

## src/ - nginx-1.7.10

- [core/](#)
- [event/](#)
- [http/](#)
- [mail/](#)
- [misc/](#)
- [os/](#)

## src/core/ - nginx-1.7.10

- [nginx.c](#)
- [nginx.h](#)
- [ngx\\_array.c](#)
- [ngx\\_array.h](#)
- [ngx\\_buf.c](#)
- [ngx\\_buf.h](#)
- [ngx\\_conf\\_file.c](#)
- [ngx\\_conf\\_file.h](#)
- [ngx\\_config.h](#)
- [ngx\\_connection.c](#)
- [ngx\\_connection.h](#)
- [ngx\\_core.h](#)
- [ngx\\_cpuid.c](#)
- [ngx\\_crc.h](#)
- [ngx\\_crc32.c](#)
- [ngx\\_crc32.h](#)
- [ngx\\_crypt.c](#)
- [ngx\\_crypt.h](#)
- [ngx\\_cycle.c](#)
- [ngx\\_cycle.h](#)
- [ngx\\_file.c](#)
- [ngx\\_file.h](#)
- [ngx\\_hash.c](#)
- [ngx\\_hash.h](#)
- [ngx\\_inet.c](#)
- [ngx\\_inet.h](#)
- [ngx\\_list.c](#)
- [ngx\\_list.h](#)
- [ngx\\_log.c](#)
- [ngx\\_log.h](#)
- [ngx\\_md5.c](#)

- [ngx\\_md5.h](#)
- [ngx\\_murmurhash.c](#)
- [ngx\\_murmurhash.h](#)
- [ngx\\_open\\_file\\_cache.c](#)
- [ngx\\_open\\_file\\_cache.h](#)
- [ngx\\_output\\_chain.c](#)
- [ngx\\_palloc.c](#)
- [ngx\\_palloc.h](#)
- [ngx\\_parse.c](#)
- [ngx\\_parse.h](#)
- [ngx\\_proxy\\_protocol.c](#)
- [ngx\\_proxy\\_protocol.h](#)
- [ngx\\_queue.c](#)
- [ngx\\_queue.h](#)
- [ngx\\_radix\\_tree.c](#)
- [ngx\\_radix\\_tree.h](#)
- [ngx\\_rbtree.c](#)
- [ngx\\_rbtree.h](#)
- [ngx\\_regex.c](#)
- [ngx\\_regex.h](#)
- [ngx\\_resolver.c](#)
- [ngx\\_resolver.h](#)
- [ngx\\_sha1.h](#)
- [ngx\\_shmtx.c](#)
- [ngx\\_shmtx.h](#)
- [ngx\\_slab.c](#)
- [ngx\\_slab.h](#)
- [ngx\\_spinlock.c](#)
- [ngx\\_string.c](#)
- [ngx\\_string.h](#)
- [ngx\\_syslog.c](#)
- [ngx\\_syslog.h](#)
- [ngx\\_times.c](#)

- [ngx\\_times.h](#)

[One Level Up](#)

[Top Level](#)

## src/core/nginx.c - nginx-1.7.10

### Global variables defined

- [ngx\\_conf\\_file](#)
- [ngx\\_conf\\_params](#)
- [ngx\\_core\\_commands](#)
- [ngx\\_core\\_module](#)
- [ngx\\_core\\_module\\_ctx](#)
- [ngx\\_debug\\_points](#)
- [ngx\\_max\\_module](#)
- [ngx\\_os\\_environ](#)
- [ngx\\_prefix](#)
- [ngx\\_show\\_configure](#)
- [ngx\\_show\\_help](#)
- [ngx\\_show\\_version](#)
- [ngx\\_signal](#)

### Functions defined

- [main](#)
- [ngx\\_add\\_inherited\\_sockets](#)
- [ngx\\_core\\_module\\_create\\_conf](#)
- [ngx\\_core\\_module\\_init\\_conf](#)
- [ngx\\_exec\\_new\\_binary](#)
- [ngx\\_get\\_cpu\\_affinity](#)
- [ngx\\_get\\_options](#)
- [ngx\\_process\\_options](#)
- [ngx\\_save\\_argv](#)
- [ngx\\_set\\_cpu\\_affinity](#)
- [ngx\\_set\\_env](#)
- [ngx\\_set\\_environment](#)
- [ngx\\_set\\_priority](#)
- [ngx\\_set\\_user](#)
- [ngx\\_set\\_worker\\_processes](#)

## Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <nginx.h>
11
12
13 static ngx_int_t ngx_add_inherited_sockets(ngx_cycle_t *cycle);
14 static ngx_int_t ngx_get_options(int argc, char *const *argv);
15 static ngx_int_t ngx_process_options(ngx_cycle_t *cycle);
16 static ngx_int_t ngx_save_argv(ngx_cycle_t *cycle, int argc, char *const *argv);
17 static void *ngx_core_module_create_conf(ngx_cycle_t *cycle);
18 static char *ngx_core_module_init_conf(ngx_cycle_t *cycle, void *conf);
19 static char *ngx_set_user(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
20 static char *ngx_set_env(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
21 static char *ngx_set_priority(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
22 static char *ngx_set_cpu_affinity(ngx_conf_t *cf, ngx_command_t *cmd,
23     void *conf);
24 static char *ngx_set_worker_processes(ngx_conf_t *cf, ngx_command_t *cmd,
25     void *conf);
26
27
28 static ngx_conf_enum_t ngx_debug_points[] = {
29     { ngx_string("stop"), NGX_DEBUG_POINTS_STOP },
30     { ngx_string("abort"), NGX_DEBUG_POINTS_ABORT },
31     { ngx_null_string, 0 }
32 };
33
34
35 static ngx_command_t ngx_core_commands[] = {
36
37     { ngx_string("daemon"),
38         NGX_MAIN_CONF|NGX_DIRECT_CONF|NGX_CONF_FLAG,
39         ngx_conf_set_flag_slot,
40         0,
41         offsetof(ngx_core_conf_t, daemon),
42         NULL },
43
44     { ngx_string("master_process"),
45         NGX_MAIN_CONF|NGX_DIRECT_CONF|NGX_CONF_FLAG,
46         ngx_conf_set_flag_slot,
47         0,
48         offsetof(ngx_core_conf_t, master),
49         NULL },
50
51     { ngx_string("timer_resolution"),
52         NGX_MAIN_CONF|NGX_DIRECT_CONF|NGX_CONF_TAKE1,
53         ngx_conf_set_msec_slot,
54         0,
55         offsetof(ngx_core_conf_t, timer_resolution),
56         NULL },
57
58     { ngx_string("pid"),
59         NGX_MAIN_CONF|NGX_DIRECT_CONF|NGX_CONF_TAKE1,
60         ngx_conf_set_str_slot,
61         0,
62         offsetof(ngx_core_conf_t, pid),
63         NULL },
64
65     { ngx_string("lock_file"),
66         NGX_MAIN_CONF|NGX_DIRECT_CONF|NGX_CONF_TAKE1,
67         ngx_conf_set_str_slot,
68         0,
69         offsetof(ngx_core_conf_t, lock_file),
70         NULL },
71
72     { ngx_string("worker_processes"),
73         NGX_MAIN_CONF|NGX_DIRECT_CONF|NGX_CONF_TAKE1,
```

```

74     ngx_set_worker_processes,
75     0,
76     0,
77     NULL },
78
79     { ngx_string("debug_points"),
80       NGX_MAIN_CONF|NGX_DIRECT_CONF|NGX_CONF_TAKE1,
81       ngx_conf_set_enum_slot,
82       0,
83       offsetof(ngx_core_conf_t, debug_points),
84       &ngx_debug_points },
85
86     { ngx_string("user"),
87       NGX_MAIN_CONF|NGX_DIRECT_CONF|NGX_CONF_TAKE12,
88       ngx_set_user,
89       0,
90       0,
91       NULL },
92
93     { ngx_string("worker_priority"),
94       NGX_MAIN_CONF|NGX_DIRECT_CONF|NGX_CONF_TAKE1,
95       ngx_set_priority,
96       0,
97       0,
98       NULL },
99
100    { ngx_string("worker_cpu_affinity"),
101      NGX_MAIN_CONF|NGX_DIRECT_CONF|NGX_CONF_1MORE,
102      ngx_set_cpu_affinity,
103      0,
104      0,
105      NULL },
106
107    { ngx_string("worker_rlimit_nofile"),
108      NGX_MAIN_CONF|NGX_DIRECT_CONF|NGX_CONF_TAKE1,
109      ngx_conf_set_num_slot,
110      0,
111      offsetof(ngx_core_conf_t, rlimit_nofile),
112      NULL },
113
114    { ngx_string("worker_rlimit_core"),
115      NGX_MAIN_CONF|NGX_DIRECT_CONF|NGX_CONF_TAKE1,
116      ngx_conf_set_off_slot,
117      0,
118      offsetof(ngx_core_conf_t, rlimit_core),
119      NULL },
120
121    { ngx_string("worker_rlimit_sigpending"),
122      NGX_MAIN_CONF|NGX_DIRECT_CONF|NGX_CONF_TAKE1,
123      ngx_conf_set_num_slot,
124      0,
125      offsetof(ngx_core_conf_t, rlimit_sigpending),
126      NULL },
127
128    { ngx_string("working_directory"),
129      NGX_MAIN_CONF|NGX_DIRECT_CONF|NGX_CONF_TAKE1,
130      ngx_conf_set_str_slot,
131      0,
132      offsetof(ngx_core_conf_t, working_directory),
133      NULL },
134
135    { ngx_string("env"),
136      NGX_MAIN_CONF|NGX_DIRECT_CONF|NGX_CONF_TAKE1,
137      ngx_set_env,
138      0,
139      0,
140      NULL },
141
142    #if (NGX_THREADS)
143
144    { ngx_string("worker_threads"),
145      NGX_MAIN_CONF|NGX_DIRECT_CONF|NGX_CONF_TAKE1,
146      ngx_conf_set_num_slot,
147      0,
148      offsetof(ngx_core_conf_t, worker_threads),
149      NULL },

```

```

150     { ngx_string("thread_stack_size"),
151       NGX_MAIN_CONF|NGX_DIRECT_CONF|NGX_CONF_TAKE1,
152       ngx_conf_set_size_slot,
153       0,
154       offsetof(ngx_core_conf_t, thread_stack_size),
155       NULL },
156
157 #endif
158
159     ngx_null_command
160 };
161
162
163
164 static ngx_core_module_t ngx_core_module_ctx = {
165     ngx_string("core"),
166     ngx_core_module_create_conf,
167     ngx_core_module_init_conf
168 };
169
170
171 ngx_module_t ngx_core_module = {
172     NGX_MODULE_V1,
173     &ngx_core_module_ctx,          /* module context */
174     ngx_core_commands,           /* module directives */
175     NGX_CORE_MODULE,           /* module type */
176     NULL,                       /* init master */
177     NULL,                       /* init module */
178     NULL,                       /* init process */
179     NULL,                       /* init thread */
180     NULL,                       /* exit thread */
181     NULL,                       /* exit process */
182     NULL,                       /* exit master */
183     NGX_MODULE_V1_PADDING
184 };
185
186
187 ngx_uint_t      ngx_max_module;
188
189 static ngx_uint_t  ngx_show_help;
190 static ngx_uint_t  ngx_show_version;
191 static ngx_uint_t  ngx_show_configure;
192 static u_char      *ngx_prefix;
193 static u_char      *ngx_conf_file;
194 static u_char      *ngx_conf_params;
195 static char        *ngx_signal;
196
197
198 static char **ngx_os_environ;
199
200
201 int ngx_cdecl
202 main(int argc, char *const *argv)
203 {
204     ngx_int_t      i;
205     ngx_log_t      *log;
206     ngx_cycle_t    *cycle, init_cycle;
207     ngx_core_conf_t *ccf;
208
209     ngx_debug_init();
210
211     if (ngx_strerror_init() != NGX_OK) {
212         return 1;
213     }
214
215     if (ngx_get_options(argc, argv) != NGX_OK) {
216         return 1;
217     }
218
219     if (ngx_show_version) {
220         ngx_write_stderr("nginx version: " NGX_VER_BUILD NGX_LINEFEED);
221
222         if (ngx_show_help) {
223             ngx_write_stderr(
224                 "Usage: nginx [-?hvVtq] [-s signal] [-c filename] "
225                 "[-p prefix] [-g directives]" NGX_LINEFEED

```



```

226         NGX\_LINEFEED
227     "Options:" NGX\_LINEFEED
228     " -?, -h      : this help" NGX\_LINEFEED
229     " -v         : show version and exit" NGX\_LINEFEED
230     " -V         : show version and configure options then exit"
231         NGX\_LINEFEED
232     " -t         : test configuration and exit" NGX\_LINEFEED
233     " -q         : suppress non-error messages "
234         "during configuration testing" NGX\_LINEFEED
235     " -s signal   : send signal to a master process: "
236         "stop, quit, reopen, reload" NGX\_LINEFEED
237 #ifdef NGX_PREFIX
238     " -p prefix   : set prefix path (default: "
239         NGX_PREFIX ")" NGX\_LINEFEED
240 #else
241     " -p prefix   : set prefix path (default: NONE)" NGX\_LINEFEED
242 #endif
243     " -c filename : set configuration file (default: "
244         NGX_CONF_PATH ")" NGX\_LINEFEED
245     " -g directives : set global directives out of configuration "
246         "file" NGX\_LINEFEED NGX\_LINEFEED
247     );
248 }
249
250     if (ngx\_show\_configure) {
251         ngx\_write\_stderr(
252 #ifdef NGX_COMPILER
253         "built by " NGX_COMPILER NGX\_LINEFEED
254 #endif
255 #if (NGX_SSL)
256 #ifdef SSL_CTRL_SET_TLSEXT_HOSTNAME
257         "TLS SNI support enabled" NGX\_LINEFEED
258 #else
259         "TLS SNI support disabled" NGX\_LINEFEED
260 #endif
261 #endif
262         "configure arguments:" NGX_CONFIGURE NGX\_LINEFEED);
263     }
264
265     if (!ngx\_test\_config) {
266         return 0;
267     }
268 }
269
270 /* TODO */ ngx\_max\_sockets = -1;
271
272     ngx\_time\_init();
273
274 #if (NGX_PCRE)
275     ngx\_regex\_init();
276 #endif
277
278     ngx\_pid = ngx\_getpid();
279
280     log = ngx\_log\_init(ngx\_prefix);
281     if (log == NULL) {
282         return 1;
283     }
284
285     /* STUB */
286 #if (NGX_OPENSSL)
287     ngx\_ssl\_init(log);
288 #endif
289
290     /*
291     * init_cycle->log is required for signal handlers and
292     * ngx\_process\_options()
293     */
294
295     ngx\_memzero(&init_cycle, sizeof(ngx\_cycle\_t));
296     init_cycle.log = log;
297     ngx\_cycle = &init_cycle;
298
299     init_cycle.pool = ngx\_create\_pool(1024, log);
300     if (init_cycle.pool == NULL) {
301         return 1;

```

```

302     }
303
304     if (ngx_save_argv(&init_cycle, argc, argv) != NGX_OK) {
305         return 1;
306     }
307
308     if (ngx_process_options(&init_cycle) != NGX_OK) {
309         return 1;
310     }
311
312     if (ngx_os_init(log) != NGX_OK) {
313         return 1;
314     }
315
316     /*
317     * ngx_crc32_table_init() requires ngx_cacheline_size set in ngx_os_init()
318     */
319
320     if (ngx_crc32_table_init() != NGX_OK) {
321         return 1;
322     }
323
324     if (ngx_add_inherited_sockets(&init_cycle) != NGX_OK) {
325         return 1;
326     }
327
328     ngx_max_module = 0;
329     for (i = 0; ngx_modules[i]; i++) {
330         ngx_modules[i]->index = ngx_max_module++;
331     }
332
333     cycle = ngx_init_cycle(&init_cycle);
334     if (cycle == NULL) {
335         if (ngx_test_config) {
336             ngx_log_stderr(0, "configuration file %s test failed",
337                 init_cycle.conf_file.data);
338         }
339
340         return 1;
341     }
342
343     if (ngx_test_config) {
344         if (!ngx_quiet_mode) {
345             ngx_log_stderr(0, "configuration file %s test is successful",
346                 cycle->conf_file.data);
347         }
348
349         return 0;
350     }
351
352     if (ngx_signal) {
353         return ngx_signal_process(cycle, ngx_signal);
354     }
355
356     ngx_os_status(cycle->log);
357
358     ngx_cycle = cycle;
359
360     ccf = (ngx_core_conf_t *) ngx_get_conf(cycle->conf_ctx, ngx_core_module);
361
362     if (ccf->master && ngx_process == NGX_PROCESS_SINGLE) {
363         ngx_process = NGX_PROCESS_MASTER;
364     }
365
366     #if !(NGX_WIN32)
367
368     if (ngx_init_signals(cycle->log) != NGX_OK) {
369         return 1;
370     }
371
372     if (!ngx_inherited && ccf->daemon) {
373         if (ngx_daemon(cycle->log) != NGX_OK) {
374             return 1;
375         }
376
377         ngx_daemonized = 1;

```

```

378     }
379
380     if (ngx_inherited) {
381         ngx_daemonized = 1;
382     }
383
384 #endif
385
386     if (ngx_create_pidfile(&ccf->pid, cycle->log) != NGX_OK) {
387         return 1;
388     }
389
390     if (ngx_log_redirect_stderr(cycle) != NGX_OK) {
391         return 1;
392     }
393
394     if (log->file->fd != ngx_stderr) {
395         if (ngx_close_file(log->file->fd) == NGX_FILE_ERROR) {
396             ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
397                 ngx_close_file_n " built-in log failed");
398         }
399     }
400
401     ngx_use_stderr = 0;
402
403     if (ngx_process == NGX_PROCESS_SINGLE) {
404         ngx_single_process_cycle(cycle);
405     } else {
406         ngx_master_process_cycle(cycle);
407     }
408 }
409
410 return 0;
411 }
412
413
414 static ngx_int_t
415 ngx_add_inherited_sockets(ngx_cycle_t *cycle)
416 {
417     u_char          *p, *v, *inherited;
418     ngx_int_t      s;
419     ngx_listening_t *ls;
420
421     inherited = (u_char *) getenv(NGINX_VAR);
422
423     if (inherited == NULL) {
424         return NGX_OK;
425     }
426
427     ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0,
428         "using inherited sockets from \"%s\"", inherited);
429
430     if (ngx_array_init(&cycle->listening, cycle->pool, 10,
431         sizeof(ngx_listening_t))
432         != NGX_OK)
433     {
434         return NGX_ERROR;
435     }
436
437     for (p = inherited, v = p; *p; p++) {
438         if (*p == ':' || *p == ';') {
439             s = ngx_atoi(v, p - v);
440             if (s == NGX_ERROR) {
441                 ngx_log_error(NGX_LOG_EMERG, cycle->log, 0,
442                     "invalid socket number \"%s\" in " NGINX_VAR
443                     " environment variable, ignoring the rest"
444                     " of the variable", v);
445                 break;
446             }
447
448             v = p + 1;
449
450             ls = ngx_array_push(&cycle->listening);
451             if (ls == NULL) {
452                 return NGX_ERROR;
453             }

```

```

454         ngx_memzero(&ls, sizeof(ngx_listening_t));
455     }
456     ls->fd = (ngx_socket_t) s;
457 }
458 }
459 }
460
461 ngx_inherited = 1;
462
463 return ngx_set_inherited_sockets(cycle);
464 }
465
466
467 char **
468 ngx_set_environment(ngx_cycle_t *cycle, ngx_uint_t *last)
469 {
470     char          **p, **env;
471     ngx_str_t      *var;
472     ngx_uint_t     i, n;
473     ngx_core_conf_t *ccf;
474
475     ccf = (ngx_core_conf_t *) ngx_get_conf(cycle->conf_ctx, ngx_core_module);
476
477     if (last == NULL && ccf->environment) {
478         return ccf->environment;
479     }
480
481     var = ccf->env.elts;
482
483     for (i = 0; i < ccf->env.nelts; i++) {
484         if (ngx_strcmp(var[i].data, "TZ") == 0
485             || ngx_strncmp(var[i].data, "TZ=", 3) == 0)
486             {
487                 goto tz_found;
488             }
489     }
490
491     var = ngx_array_push(&ccf->env);
492     if (var == NULL) {
493         return NULL;
494     }
495
496     var->len = 2;
497     var->data = (u_char *) "TZ";
498
499     var = ccf->env.elts;
500
501 tz_found:
502
503     n = 0;
504
505     for (i = 0; i < ccf->env.nelts; i++) {
506
507         if (var[i].data[var[i].len] == '=') {
508             n++;
509             continue;
510         }
511
512         for (p = ngx_os_environ; *p; p++) {
513
514             if (ngx_strncmp(*p, var[i].data, var[i].len) == 0
515                 && (*p)[var[i].len] == '=')
516                 {
517                     n++;
518                     break;
519                 }
520         }
521     }
522
523     if (last) {
524         env = ngx_alloc((*last + n + 1) * sizeof(char *), cycle->log);
525         *last = n;
526     }
527     else {
528         env = ngx_palloc(cycle->pool, (n + 1) * sizeof(char *));
529     }

```

```

530     if (env == NULL) {
531         return NULL;
532     }
533 }
534
535 n = 0;
536
537 for (i = 0; i < ccf->env.nelts; i++) {
538     if (var[i].data[var[i].len] == '=') {
539         env[n++] = (char *) var[i].data;
540         continue;
541     }
542 }
543
544 for (p = ngx_os_environ; *p; p++) {
545     if (ngx_strncmp(*p, var[i].data, var[i].len) == 0
546         && (*p)[var[i].len] == '=')
547     {
548         env[n++] = *p;
549         break;
550     }
551 }
552 }
553 }
554
555 env[n] = NULL;
556
557 if (last == NULL) {
558     ccf->environment = env;
559     environ = env;
560 }
561
562 return env;
563 }
564
565
566 ngx_pid_t
567 ngx_exec_new_binary(ngx_cycle_t *cycle, char *const *argv)
568 {
569     char          **env, *var;
570     u_char        *p;
571     ngx_uint_t    i, n;
572     ngx_pid_t     pid;
573     ngx_exec_ctx_t ctx;
574     ngx_core_conf_t *ccf;
575     ngx_listening_t *ls;
576
577     ngx_memzero(&ctx, sizeof(ngx_exec_ctx_t));
578
579     ctx.path = argv[0];
580     ctx.name = "new binary process";
581     ctx.argv = argv;
582
583     n = 2;
584     env = ngx_set_environment(cycle, &n);
585     if (env == NULL) {
586         return NGX_INVALID_PID;
587     }
588
589     var = ngx_alloc(sizeof(NGINX_VAR)
590                    + cycle->listening.nelts * (NGX_INT32_LEN + 1) + 2,
591                    cycle->log);
592     if (var == NULL) {
593         ngx_free(env);
594         return NGX_INVALID_PID;
595     }
596
597     p = ngx_cpymem(var, NGINX_VAR "=", sizeof(NGINX_VAR));
598
599     ls = cycle->listening.elts;
600     for (i = 0; i < cycle->listening.nelts; i++) {
601         p = ngx_sprintf(p, "%ud;", ls[i].fd);
602     }
603
604     *p = '\0';
605

```

```

606     env[n++] = var;
607
608     #if (NGX_SETPROCTITLE_USES_ENV)
609
610     /* allocate the spare 300 bytes for the new binary process title */
611
612     env[n++] = "SPARE=XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
613             "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
614             "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
615             "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
616             "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
617
618     #endif
619
620     env[n] = NULL;
621
622     #if (NGX_DEBUG)
623     {
624     char **e;
625     for (e = env; *e; e++) {
626         ngx_log_debug1(NGX_LOG_DEBUG_CORE, cycle->log, 0, "env: %s", *e);
627     }
628     }
629     #endif
630
631     ctx.envp = (char *const *) env;
632
633     ccf = (ngx_core_conf_t *) ngx_get_conf(cycle->conf_ctx, ngx_core_module);
634
635     if (ngx_rename_file(ccf->pid.data, ccf->oldpid.data) == NGX_FILE_ERROR) {
636         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
637             ngx_rename_file_n " %s to %s failed "
638             "before executing new binary process \"%s\"",
639             ccf->pid.data, ccf->oldpid.data, argv[0]);
640
641         ngx_free(env);
642         ngx_free(var);
643
644         return NGX_INVALID_PID;
645     }
646
647     pid = ngx_execute(cycle, &ctx);
648
649     if (pid == NGX_INVALID_PID) {
650         if (ngx_rename_file(ccf->oldpid.data, ccf->pid.data)
651             == NGX_FILE_ERROR)
652         {
653             ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
654                 ngx_rename_file_n " %s back to %s failed after "
655                 "an attempt to execute new binary process \"%s\"",
656                 ccf->oldpid.data, ccf->pid.data, argv[0]);
657         }
658     }
659
660     ngx_free(env);
661     ngx_free(var);
662
663     return pid;
664 }
665
666
667 static ngx_int_t
668 ngx_get_options(int argc, char *const *argv)
669 {
670     u_char *p;
671     ngx_int_t i;
672
673     for (i = 1; i < argc; i++) {
674
675         p = (u_char *) argv[i];
676
677         if (*p++ != '-') {
678             ngx_log_stderr(0, "invalid option: \"%s\"", argv[i]);
679             return NGX_ERROR;
680         }
681     }

```

```

682 while (*p) {
683
684     switch (*p++) {
685
686     case '?':
687     case 'h':
688         ngx_show_version = 1;
689         ngx_show_help = 1;
690         break;
691
692     case 'v':
693         ngx_show_version = 1;
694         break;
695
696     case 'V':
697         ngx_show_version = 1;
698         ngx_show_configure = 1;
699         break;
700
701     case 't':
702         ngx_test_config = 1;
703         break;
704
705     case 'q':
706         ngx_quiet_mode = 1;
707         break;
708
709     case 'p':
710         if (*p) {
711             ngx_prefix = p;
712             goto next;
713         }
714
715         if (argv[++i]) {
716             ngx_prefix = (u_char *) argv[i];
717             goto next;
718         }
719
720         ngx_log_stderr(0, "option \"-p\" requires directory name");
721         return NGX_ERROR;
722
723     case 'c':
724         if (*p) {
725             ngx_conf_file = p;
726             goto next;
727         }
728
729         if (argv[++i]) {
730             ngx_conf_file = (u_char *) argv[i];
731             goto next;
732         }
733
734         ngx_log_stderr(0, "option \"-c\" requires file name");
735         return NGX_ERROR;
736
737     case 'g':
738         if (*p) {
739             ngx_conf_params = p;
740             goto next;
741         }
742
743         if (argv[++i]) {
744             ngx_conf_params = (u_char *) argv[i];
745             goto next;
746         }
747
748         ngx_log_stderr(0, "option \"-g\" requires parameter");
749         return NGX_ERROR;
750
751     case 's':
752         if (*p) {
753             ngx_signal = (char *) p;
754
755         } else if (argv[++i]) {
756             ngx_signal = argv[i];
757

```

```

758     } else {
759         ngx_log_stderr(0, "option \"-s\" requires parameter");
760         return NGX_ERROR;
761     }
762
763     if (ngx_strcmp(ngx_signal, "stop") == 0
764         || ngx_strcmp(ngx_signal, "quit") == 0
765         || ngx_strcmp(ngx_signal, "reopen") == 0
766         || ngx_strcmp(ngx_signal, "reload") == 0)
767     {
768         ngx_process = NGX_PROCESS_SIGNALLER;
769         goto next;
770     }
771
772     ngx_log_stderr(0, "invalid option: \"-s %s\"", ngx_signal);
773     return NGX_ERROR;
774
775     default:
776         ngx_log_stderr(0, "invalid option: \"%c\"", *(p - 1));
777         return NGX_ERROR;
778     }
779 }
780
781 next:
782
783     continue;
784 }
785
786 return NGX_OK;
787 }
788
789
790 static ngx_int_t
791 ngx_save_argv(ngx_cycle_t *cycle, int argc, char *const *argv)
792 {
793     #if (NGX_FREEBSD)
794
795         ngx_os_argv = (char **) argv;
796         ngx_argc = argc;
797         ngx_argv = (char **) argv;
798
799     #else
800         size_t    len;
801         ngx_int_t i;
802
803         ngx_os_argv = (char **) argv;
804         ngx_argc = argc;
805
806         ngx_argv = ngx_alloc((argc + 1) * sizeof(char *), cycle->log);
807         if (ngx_argv == NULL) {
808             return NGX_ERROR;
809         }
810
811         for (i = 0; i < argc; i++) {
812             len = ngx_strlen(argv[i]) + 1;
813
814             ngx_argv[i] = ngx_alloc(len, cycle->log);
815             if (ngx_argv[i] == NULL) {
816                 return NGX_ERROR;
817             }
818
819             (void) ngx_cpystn((u_char *) ngx_argv[i], (u_char *) argv[i], len);
820         }
821
822         ngx_argv[i] = NULL;
823
824     #endif
825
826     ngx_os_environ = environ;
827
828     return NGX_OK;
829 }
830
831
832 static ngx_int_t
833 ngx_process_options(ngx_cycle_t *cycle)

```



```

834 {
835     u_char *p;
836     size_t len;
837
838     if (ngx_prefix) {
839         len = ngx_strlen(ngx_prefix);
840         p = ngx_prefix;
841
842         if (len && !ngx_path_separator(p[len - 1])) {
843             p = ngx_pnalloc(cycle->pool, len + 1);
844             if (p == NULL) {
845                 return NGX_ERROR;
846             }
847
848             ngx_memcpy(p, ngx_prefix, len);
849             p[len++] = '/';
850         }
851
852         cycle->conf_prefix.len = len;
853         cycle->conf_prefix.data = p;
854         cycle->prefix.len = len;
855         cycle->prefix.data = p;
856
857     } else {
858
859 #ifndef NGX_PREFIX
860
861         p = ngx_pnalloc(cycle->pool, NGX_MAX_PATH);
862         if (p == NULL) {
863             return NGX_ERROR;
864         }
865
866         if (ngx_getcwd(p, NGX_MAX_PATH) == 0) {
867             ngx_log_stderr(ngx_errno, "[emerg]: " ngx_getcwd_n " failed");
868             return NGX_ERROR;
869         }
870
871         len = ngx_strlen(p);
872
873         p[len++] = '/';
874
875         cycle->conf_prefix.len = len;
876         cycle->conf_prefix.data = p;
877         cycle->prefix.len = len;
878         cycle->prefix.data = p;
879
880 #else
881
882 #ifdef NGX_CONF_PREFIX
883         ngx_str_set(&cycle->conf_prefix, NGX_CONF_PREFIX);
884 #else
885         ngx_str_set(&cycle->conf_prefix, NGX_PREFIX);
886 #endif
887         ngx_str_set(&cycle->prefix, NGX_PREFIX);
888
889 #endif
890     }
891
892     if (ngx_conf_file) {
893         cycle->conf_file.len = ngx_strlen(ngx_conf_file);
894         cycle->conf_file.data = ngx_conf_file;
895
896     } else {
897         ngx_str_set(&cycle->conf_file, NGX_CONF_PATH);
898     }
899
900     if (ngx_conf_full_name(cycle, &cycle->conf_file, 0) != NGX_OK) {
901         return NGX_ERROR;
902     }
903
904     for (p = cycle->conf_file.data + cycle->conf_file.len - 1;
905          p > cycle->conf_file.data;
906          p--)
907     {
908         if (ngx_path_separator(*p)) {
909             cycle->conf_prefix.len = p - ngx_cycle->conf_file.data + 1;

```

```

910         cycle->conf_prefix.data = ngx_cycle->conf_file.data;
911         break;
912     }
913 }
914
915 if (ngx_conf_params) {
916     cycle->conf_param.len = ngx_strlen(ngx_conf_params);
917     cycle->conf_param.data = ngx_conf_params;
918 }
919
920 if (ngx_test_config) {
921     cycle->log->log_level = NGX_LOG_INFO;
922 }
923
924 return NGX_OK;
925 }
926
927
928 static void *
929 ngx_core_module_create_conf(ngx_cycle_t *cycle)
930 {
931     ngx_core_conf_t *ccf;
932
933     ccf = ngx_palloc(cycle->pool, sizeof(ngx_core_conf_t));
934     if (ccf == NULL) {
935         return NULL;
936     }
937
938     /*
939     * set by ngx_palloc()
940     *
941     *     ccf->pid = NULL;
942     *     ccf->oldpid = NULL;
943     *     ccf->priority = 0;
944     *     ccf->cpu_affinity_n = 0;
945     *     ccf->cpu_affinity = NULL;
946     */
947
948     ccf->daemon = NGX_CONF_UNSET;
949     ccf->master = NGX_CONF_UNSET;
950     ccf->timer_resolution = NGX_CONF_UNSET_MSEC;
951
952     ccf->worker_processes = NGX_CONF_UNSET;
953     ccf->debug_points = NGX_CONF_UNSET;
954
955     ccf->rlimit_nofile = NGX_CONF_UNSET;
956     ccf->rlimit_core = NGX_CONF_UNSET;
957     ccf->rlimit_sigpending = NGX_CONF_UNSET;
958
959     ccf->user = (ngx_uid_t) NGX_CONF_UNSET_UINT;
960     ccf->group = (ngx_gid_t) NGX_CONF_UNSET_UINT;
961
962     #if (NGX_THREADS)
963     ccf->worker_threads = NGX_CONF_UNSET;
964     ccf->thread_stack_size = NGX_CONF_UNSET_SIZE;
965     #endif
966
967     if (ngx_array_init(&ccf->env, cycle->pool, 1, sizeof(ngx_str_t))
968         != NGX_OK)
969     {
970         return NULL;
971     }
972
973     return ccf;
974 }
975
976
977 static char *
978 ngx_core_module_init_conf(ngx_cycle_t *cycle, void *conf)
979 {
980     ngx_core_conf_t *ccf = conf;
981
982     ngx_conf_init_value(ccf->daemon, 1);
983     ngx_conf_init_value(ccf->master, 1);
984     ngx_conf_init_msec_value(ccf->timer_resolution, 0);
985

```

```

986     ngx_conf_init_value(ccf->worker_processes, 1);
987     ngx_conf_init_value(ccf->debug_points, 0);
988
989     #if (NGX_HAVE_CPU_AFFINITY)
990
991     if (ccf->cpu_affinity_n
992         && ccf->cpu_affinity_n != 1
993         && ccf->cpu_affinity_n != (ngx_uint_t) ccf->worker_processes)
994     {
995         ngx_log_error(NGX_LOG_WARN, cycle->log, 0,
996             "the number of \"worker_processes\" is not equal to \"
997             \"the number of \"worker_cpu_affinity\" masks, \"
998             \"using last mask for remaining worker processes");
999     }
1000
1001     #endif
1002
1003     #if (NGX_THREADS)
1004
1005     ngx_conf_init_value(ccf->worker_threads, 0);
1006     ngx_threads_n = ccf->worker_threads;
1007     ngx_conf_init_size_value(ccf->thread_stack_size, 2 * 1024 * 1024);
1008
1009     #endif
1010
1011
1012     if (ccf->pid.len == 0) {
1013         ngx_str_set(&ccf->pid, NGX_PID_PATH);
1014     }
1015
1016     if (ngx_conf_full_name(cycle, &ccf->pid, 0) != NGX_OK) {
1017         return NGX_CONF_ERROR;
1018     }
1019
1020     ccf->oldpid.len = ccf->pid.len + sizeof(NGX_OLDPID_EXT);
1021
1022     ccf->oldpid.data = ngx_pnalloc(cycle->pool, ccf->oldpid.len);
1023     if (ccf->oldpid.data == NULL) {
1024         return NGX_CONF_ERROR;
1025     }
1026
1027     ngx_memcpy(ngx_cpymem(ccf->oldpid.data, ccf->pid.data, ccf->pid.len),
1028         NGX_OLDPID_EXT, sizeof(NGX_OLDPID_EXT));
1029
1030
1031     #if !(NGX_WIN32)
1032
1033     if (ccf->user == (uid_t) NGX_CONF_UNSET_UINT && geteuid() == 0) {
1034         struct group *grp;
1035         struct passwd *pwd;
1036
1037         ngx_set_errno(0);
1038         pwd = getpwnam(NGX_USER);
1039         if (pwd == NULL) {
1040             ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
1041                 "getpwnam(\"" NGX_USER "\" failed");
1042             return NGX_CONF_ERROR;
1043         }
1044
1045         ccf->username = NGX_USER;
1046         ccf->user = pwd->pw_uid;
1047
1048         ngx_set_errno(0);
1049         grp = getgrnam(NGX_GROUP);
1050         if (grp == NULL) {
1051             ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
1052                 "getgrnam(\"" NGX_GROUP "\" failed");
1053             return NGX_CONF_ERROR;
1054         }
1055
1056         ccf->group = grp->gr_gid;
1057     }
1058
1059
1060     if (ccf->lock_file.len == 0) {
1061         ngx_str_set(&ccf->lock_file, NGX_LOCK_PATH);

```

```

1062     }
1063
1064     if (ngx_conf_full_name(cycle, &ccf->lock_file, 0) != NGX_OK) {
1065         return NGX_CONF_ERROR;
1066     }
1067
1068     {
1069         ngx_str_t lock_file;
1070
1071         lock_file = cycle->old_cycle->lock_file;
1072
1073         if (lock_file.len) {
1074             lock_file.len--;
1075
1076             if (ccf->lock_file.len != lock_file.len
1077                 || ngx_strncmp(ccf->lock_file.data, lock_file.data, lock_file.len)
1078                     != 0)
1079             {
1080                 ngx_log_error(NGX_LOG_EMERG, cycle->log, 0,
1081                     "\"lock_file\" could not be changed, ignored");
1082             }
1083
1084             cycle->lock_file.len = lock_file.len + 1;
1085             lock_file.len += sizeof(".accept");
1086
1087             cycle->lock_file.data = ngx_pstrdup(cycle->pool, &lock_file);
1088             if (cycle->lock_file.data == NULL) {
1089                 return NGX_CONF_ERROR;
1090             }
1091         } else {
1092             cycle->lock_file.len = ccf->lock_file.len + 1;
1093             cycle->lock_file.data = ngx_pnalloc(cycle->pool,
1094                 ccf->lock_file.len + sizeof(".accept"));
1095             if (cycle->lock_file.data == NULL) {
1096                 return NGX_CONF_ERROR;
1097             }
1098         }
1099
1100         ngx_memcpy(ngx_cpymem(cycle->lock_file.data, ccf->lock_file.data,
1101             ccf->lock_file.len),
1102             ".accept", sizeof(".accept"));
1103     }
1104 }
1105
1106 #endif
1107
1108     return NGX_CONF_OK;
1109 }
1110
1111
1112 static char *
1113 ngx_set_user(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
1114 {
1115     #if (NGX_WIN32)
1116
1117         ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
1118             "\"user\" is not supported, ignored");
1119
1120         return NGX_CONF_OK;
1121
1122     #else
1123
1124         ngx_core_conf_t *ccf = conf;
1125
1126         char *group;
1127         struct passwd *pwd;
1128         struct group *grp;
1129         ngx_str_t *value;
1130
1131         if (ccf->user != (uid_t) NGX_CONF_UNSET_UINT) {
1132             return "is duplicate";
1133         }
1134
1135         if (geteuid() != 0) {
1136             ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
1137                 "the \"user\" directive makes sense only "

```

```

1138         "if the master process runs "
1139         "with super-user privileges, ignored");
1140     return NGX_CONF_OK;
1141 }
1142
1143 value = (ngx_str_t *) cf->args->elts;
1144
1145 ccf->username = (char *) value[1].data;
1146
1147 ngx_set_errno(0);
1148 pwd = getpwnam((const char *) value[1].data);
1149 if (pwd == NULL) {
1150     ngx_conf_log_error(NGX_LOG_EMERG, cf, ngx_errno,
1151         "getpwnam(\"%s\") failed", value[1].data);
1152     return NGX_CONF_ERROR;
1153 }
1154
1155 ccf->user = pwd->pw_uid;
1156
1157 group = (char *) ((cf->args->nelts == 2) ? value[1].data : value[2].data);
1158
1159 ngx_set_errno(0);
1160 grp = getgrnam(group);
1161 if (grp == NULL) {
1162     ngx_conf_log_error(NGX_LOG_EMERG, cf, ngx_errno,
1163         "getgrnam(\"%s\") failed", group);
1164     return NGX_CONF_ERROR;
1165 }
1166
1167 ccf->group = grp->gr_gid;
1168
1169 return NGX_CONF_OK;
1170
1171 #endif
1172 }
1173
1174
1175 static char *
1176 ngx_set_env(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
1177 {
1178     ngx_core_conf_t *ccf = conf;
1179
1180     ngx_str_t *value, *var;
1181     ngx_uint_t i;
1182
1183     var = ngx_array_push(&ccf->env);
1184     if (var == NULL) {
1185         return NGX_CONF_ERROR;
1186     }
1187
1188     value = cf->args->elts;
1189     *var = value[1];
1190
1191     for (i = 0; i < value[1].len; i++) {
1192         if (value[1].data[i] == '=') {
1193             var->len = i;
1194
1195             return NGX_CONF_OK;
1196         }
1197     }
1198 }
1199
1200 return NGX_CONF_OK;
1201 }
1202
1203
1204
1205 static char *
1206 ngx_set_priority(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
1207 {
1208     ngx_core_conf_t *ccf = conf;
1209
1210     ngx_str_t *value;
1211     ngx_uint_t n, minus;
1212
1213     if (ccf->priority != 0) {

```

```

1214     return "is duplicate";
1215 }
1216
1217 value = cf->args->elts;
1218
1219 if (value[1].data[0] == '-') {
1220     n = 1;
1221     minus = 1;
1222
1223 } else if (value[1].data[0] == '+') {
1224     n = 1;
1225     minus = 0;
1226
1227 } else {
1228     n = 0;
1229     minus = 0;
1230 }
1231
1232 ccf->priority = ngx_atoi(&value[1].data[n], value[1].len - n);
1233 if (ccf->priority == NGX_ERROR) {
1234     return "invalid number";
1235 }
1236
1237 if (minus) {
1238     ccf->priority = -ccf->priority;
1239 }
1240
1241 return NGX_CONF_OK;
1242 }
1243
1244
1245 static char *
1246 ngx_set_cpu_affinity(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
1247 {
1248     #if (NGX_HAVE_CPU_AFFINITY)
1249         ngx_core_conf_t *ccf = conf;
1250
1251         u_char          ch;
1252         uint64_t        *mask;
1253         ngx_str_t       *value;
1254         ngx_uint_t      i, n;
1255
1256         if (ccf->cpu_affinity) {
1257             return "is duplicate";
1258         }
1259
1260         mask = ngx_palloc(cf->pool, (cf->args->nelts - 1) * sizeof(uint64_t));
1261         if (mask == NULL) {
1262             return NGX_CONF_ERROR;
1263         }
1264
1265         ccf->cpu_affinity_n = cf->args->nelts - 1;
1266         ccf->cpu_affinity = mask;
1267
1268         value = cf->args->elts;
1269
1270         for (n = 1; n < cf->args->nelts; n++) {
1271
1272             if (value[n].len > 64) {
1273                 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1274                     "\"worker_cpu_affinity\" supports up to 64 CPUs only");
1275                 return NGX_CONF_ERROR;
1276             }
1277
1278             mask[n - 1] = 0;
1279
1280             for (i = 0; i < value[n].len; i++) {
1281
1282                 ch = value[n].data[i];
1283
1284                 if (ch == ' ') {
1285                     continue;
1286                 }
1287
1288                 mask[n - 1] <<= 1;
1289

```

```

1290     if (ch == '0') {
1291         continue;
1292     }
1293
1294     if (ch == '1') {
1295         mask[n - 1] |= 1;
1296         continue;
1297     }
1298
1299     ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
1300         "invalid character \"%c\" in \"worker_cpu_affinity\"",
1301         ch);
1302     return NGX\_CONF\_ERROR;
1303 }
1304 }
1305
1306 #else
1307
1308     ngx\_conf\_log\_error(NGX\_LOG\_WARN, cf, 0,
1309         "\"worker_cpu_affinity\" is not supported "
1310         "on this platform, ignored");
1311 #endif
1312
1313     return NGX\_CONF\_OK;
1314 }
1315
1316
1317 uint64\_t
1318 ngx\_get\_cpu\_affinity(ngx\_uint\_t n)
1319 {
1320     ngx\_core\_conf\_t *ccf;
1321
1322     ccf = (ngx\_core\_conf\_t *) ngx\_get\_conf(ngx\_cycle->conf\_ctx,
1323         ngx\_core\_module);
1324
1325     if (ccf->cpu_affinity == NULL) {
1326         return 0;
1327     }
1328
1329     if (ccf->cpu_affinity_n > n) {
1330         return ccf->cpu_affinity[n];
1331     }
1332
1333     return ccf->cpu_affinity[ccf->cpu_affinity_n - 1];
1334 }
1335
1336
1337 static char *
1338 ngx\_set\_worker\_processes(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
1339 {
1340     ngx\_str\_t *value;
1341     ngx\_core\_conf\_t *ccf;
1342
1343     ccf = (ngx\_core\_conf\_t *) conf;
1344
1345     if (ccf->worker_processes != NGX\_CONF\_UNSET) {
1346         return "is duplicate";
1347     }
1348
1349     value = (ngx\_str\_t *) cf->args->elts;
1350
1351     if (ngx\_strcmp(value[1].data, "auto") == 0) {
1352         ccf->worker_processes = ngx\_ncpu;
1353         return NGX\_CONF\_OK;
1354     }
1355
1356     ccf->worker_processes = ngx\_atoi(value[1].data, value[1].len);
1357
1358     if (ccf->worker_processes == NGX\_ERROR) {
1359         return "invalid value";
1360     }
1361
1362     return NGX\_CONF\_OK;
1363 }

```

# src/http/modules/perl/nginx\_http\_perl\_module.h - nginx-1.7.10

## Data types defined

- [nginx](#)
- [ngx\\_http\\_perl\\_ctx\\_t](#)
- [ngx\\_http\\_perl\\_var\\_t](#)

## Macros defined

- [\\_NGX\\_HTTP\\_PERL\\_MODULE\\_H\\_INCLUDED](#)
- [dTHXa](#)
- [dTHXa](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_HTTP_PERL_MODULE_H_INCLUDED_
9 #define _NGX_HTTP_PERL_MODULE_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14 #include <ngx_http.h>
15 #include <nginx.h>
16
17 #include <EXTERN.h>
18 #include <perl.h>
19
20
21 typedef ngx_http_request_t *ngxinx;
22
23 typedef struct {
24     ngx_str_t      filename;
25     ngx_str_t      redirect_uri;
26     ngx_str_t      redirect_args;
27
28     SV             *next;
29
30     ngx_uint_t     done;      /* unsigned done:1; */
31
32     ngx_array_t    *variables; /* array of ngx_http_perl_var_t */
33
34     #if (NGX_HTTP_SSI)
35     ngx_http_ssi_ctx_t *ssi;
36     #endif
37 } ngx_http_perl_ctx_t;
38
39
40 typedef struct {
41     ngx_uint_t     hash;
42     ngx_str_t      name;
43     ngx_str_t      value;
44 } ngx_http_perl_var_t;
45
46
47 extern ngx_module_t ngx_http_perl_module;
48
```



```
49
50 /*
51  * workaround for "unused variable `Perl__notused'" warning
52  * when building with perl 5.6.1
53  */
54 #ifndef PERL_IMPLICIT_CONTEXT
55 #undef dTHXa
56 #define dTHXa(a)
57 #endif
58
59
60 extern void boot_DynaLoader(pTHX_ CV* cv);
61
62
63 void ngx\_http\_perl\_handle\_request(ngx\_http\_request\_t *r);
64 void ngx\_http\_perl\_sleep\_handler(ngx\_http\_request\_t *r);
65
66
67 #endif /* \_NGX\_HTTP\_PERL\_MODULE\_H\_INCLUDED */
```

[One Level Up](#)

[Top Level](#)

[One Level Up](#)

[Top Level](#)

## src/http/modules/perl/ - nginx-1.7.10

- [nginx.pm](#)
- [ngx\\_http\\_perl\\_module.c](#)
- [ngx\\_http\\_perl\\_module.h](#)

[One Level Up](#)

[Top Level](#)

## src/http/modules/ - nginx-1.7.10

- [ngx\\_http\\_access\\_module.c](#)
- [ngx\\_http\\_addition\\_filter\\_module.c](#)
- [ngx\\_http\\_auth\\_basic\\_module.c](#)
- [ngx\\_http\\_auth\\_request\\_module.c](#)
- [ngx\\_http\\_autoindex\\_module.c](#)
- [ngx\\_http\\_browser\\_module.c](#)
- [ngx\\_http\\_charset\\_filter\\_module.c](#)
- [ngx\\_http\\_chunked\\_filter\\_module.c](#)
- [ngx\\_http\\_dav\\_module.c](#)
- [ngx\\_http\\_degradation\\_module.c](#)
- [ngx\\_http\\_empty\\_gif\\_module.c](#)
- [ngx\\_http\\_fastcgi\\_module.c](#)
- [ngx\\_http\\_flv\\_module.c](#)
- [ngx\\_http\\_geo\\_module.c](#)
- [ngx\\_http\\_geoip\\_module.c](#)
- [ngx\\_http\\_gunzip\\_filter\\_module.c](#)
- [ngx\\_http\\_gzip\\_filter\\_module.c](#)
- [ngx\\_http\\_gzip\\_static\\_module.c](#)
- [ngx\\_http\\_headers\\_filter\\_module.c](#)
- [ngx\\_http\\_image\\_filter\\_module.c](#)
- [ngx\\_http\\_index\\_module.c](#)
- [ngx\\_http\\_limit\\_conn\\_module.c](#)
- [ngx\\_http\\_limit\\_req\\_module.c](#)
- [ngx\\_http\\_log\\_module.c](#)
- [ngx\\_http\\_map\\_module.c](#)
- [ngx\\_http\\_memcached\\_module.c](#)
- [ngx\\_http\\_mp4\\_module.c](#)
- [ngx\\_http\\_not\\_modified\\_filter\\_module.c](#)
- [ngx\\_http\\_proxy\\_module.c](#)
- [ngx\\_http\\_random\\_index\\_module.c](#)
- [ngx\\_http\\_range\\_filter\\_module.c](#)

- [ngx\\_http\\_realip\\_module.c](#)
- [ngx\\_http\\_referer\\_module.c](#)
- [ngx\\_http\\_rewrite\\_module.c](#)
- [ngx\\_http\\_scgi\\_module.c](#)
- [ngx\\_http\\_secure\\_link\\_module.c](#)
- [ngx\\_http\\_split\\_clients\\_module.c](#)
- [ngx\\_http\\_ssi\\_filter\\_module.c](#)
- [ngx\\_http\\_ssi\\_filter\\_module.h](#)
- [ngx\\_http\\_ssl\\_module.c](#)
- [ngx\\_http\\_ssl\\_module.h](#)
- [ngx\\_http\\_static\\_module.c](#)
- [ngx\\_http\\_stub\\_status\\_module.c](#)
- [ngx\\_http\\_sub\\_filter\\_module.c](#)
- [ngx\\_http\\_upstream\\_hash\\_module.c](#)
- [ngx\\_http\\_upstream\\_ip\\_hash\\_module.c](#)
- [ngx\\_http\\_upstream\\_keepalive\\_module.c](#)
- [ngx\\_http\\_upstream\\_least\\_conn\\_module.c](#)
- [ngx\\_http\\_userid\\_filter\\_module.c](#)
- [ngx\\_http\\_uwsgi\\_module.c](#)
- [ngx\\_http\\_xslt\\_filter\\_module.c](#)
- [perl/](#)

[One Level Up](#)

[Top Level](#)

## src/http/ - nginx-1.7.10

- [modules/](#)
- [ngx\\_http.c](#)
- [ngx\\_http.h](#)
- [ngx\\_http\\_busy\\_lock.c](#)
- [ngx\\_http\\_busy\\_lock.h](#)
- [ngx\\_http\\_cache.h](#)
- [ngx\\_http\\_config.h](#)
- [ngx\\_http\\_copy\\_filter\\_module.c](#)
- [ngx\\_http\\_core\\_module.c](#)
- [ngx\\_http\\_core\\_module.h](#)
- [ngx\\_http\\_file\\_cache.c](#)
- [ngx\\_http\\_header\\_filter\\_module.c](#)
- [ngx\\_http\\_parse.c](#)
- [ngx\\_http\\_parse\\_time.c](#)
- [ngx\\_http\\_postpone\\_filter\\_module.c](#)
- [ngx\\_http\\_request.c](#)
- [ngx\\_http\\_request.h](#)
- [ngx\\_http\\_request\\_body.c](#)
- [ngx\\_http\\_script.c](#)
- [ngx\\_http\\_script.h](#)
- [ngx\\_http\\_spdy.c](#)
- [ngx\\_http\\_spdy.h](#)
- [ngx\\_http\\_spdy\\_filter\\_module.c](#)
- [ngx\\_http\\_spdy\\_module.c](#)
- [ngx\\_http\\_spdy\\_module.h](#)
- [ngx\\_http\\_special\\_response.c](#)
- [ngx\\_http\\_upstream.c](#)
- [ngx\\_http\\_upstream.h](#)
- [ngx\\_http\\_upstream\\_round\\_robin.c](#)
- [ngx\\_http\\_upstream\\_round\\_robin.h](#)
- [ngx\\_http\\_variables.c](#)

- [ngx\\_http\\_variables.h](#)
- [ngx\\_http\\_write\\_filter\\_module.c](#)

[One Level Up](#)

[Top Level](#)

## src/http/nginx\_http.c - nginx-1.7.10

### Global variables defined

- [ngx\\_http\\_commands](#)
- [ngx\\_http\\_html\\_default\\_types](#)
- [ngx\\_http\\_max\\_module](#)
- [ngx\\_http\\_module](#)
- [ngx\\_http\\_module\\_ctx](#)
- [ngx\\_http\\_top\\_body\\_filter](#)
- [ngx\\_http\\_top\\_header\\_filter](#)

### Functions defined

- [ngx\\_http\\_add\\_address](#)
- [ngx\\_http\\_add\\_addresses](#)
- [ngx\\_http\\_add\\_addrs](#)
- [ngx\\_http\\_add\\_addrs6](#)
- [ngx\\_http\\_add\\_listen](#)
- [ngx\\_http\\_add\\_listening](#)
- [ngx\\_http\\_add\\_location](#)
- [ngx\\_http\\_add\\_server](#)
- [ngx\\_http\\_block](#)
- [ngx\\_http\\_cmp\\_conf\\_addrs](#)
- [ngx\\_http\\_cmp\\_dns\\_wildcards](#)
- [ngx\\_http\\_cmp\\_locations](#)
- [ngx\\_http\\_create\\_locations\\_list](#)
- [ngx\\_http\\_create\\_locations\\_tree](#)
- [ngx\\_http\\_init\\_headers\\_in\\_hash](#)
- [ngx\\_http\\_init\\_listening](#)
- [ngx\\_http\\_init\\_locations](#)
- [ngx\\_http\\_init\\_phase\\_handlers](#)
- [ngx\\_http\\_init\\_phases](#)
- [ngx\\_http\\_init\\_static\\_location\\_trees](#)
- [ngx\\_http\\_join\\_exact\\_locations](#)

- [ngx http merge locations](#)
- [ngx http merge servers](#)
- [ngx http merge types](#)
- [ngx http optimize servers](#)
- [ngx http server names](#)
- [ngx http set default types](#)
- [ngx http types slot](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 static char *ngx_http_block(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
14 static ngx_int_t ngx_http_init_phases(ngx_conf_t *cf,
15     ngx_http_core_main_conf_t *cmcf);
16 static ngx_int_t ngx_http_init_headers_in_hash(ngx_conf_t *cf,
17     ngx_http_core_main_conf_t *cmcf);
18 static ngx_int_t ngx_http_init_phase_handlers(ngx_conf_t *cf,
19     ngx_http_core_main_conf_t *cmcf);
20
21 static ngx_int_t ngx_http_add_addresses(ngx_conf_t *cf,
22     ngx_http_core_srv_conf_t *cscf, ngx_http_conf_port_t *port,
23     ngx_http_listen_opt_t *lsopt);
24 static ngx_int_t ngx_http_add_address(ngx_conf_t *cf,
25     ngx_http_core_srv_conf_t *cscf, ngx_http_conf_port_t *port,
26     ngx_http_listen_opt_t *lsopt);
27 static ngx_int_t ngx_http_add_server(ngx_conf_t *cf,
28     ngx_http_core_srv_conf_t *cscf, ngx_http_conf_addr_t *addr);
29
30 static char *ngx_http_merge_servers(ngx_conf_t *cf,
31     ngx_http_core_main_conf_t *cmcf, ngx_http_module_t *module,
32     ngx_uint_t ctx_index);
33 static char *ngx_http_merge_locations(ngx_conf_t *cf,
34     ngx_queue_t *locations, void **loc_conf, ngx_http_module_t *module,
35     ngx_uint_t ctx_index);
36 static ngx_int_t ngx_http_init_locations(ngx_conf_t *cf,
37     ngx_http_core_srv_conf_t *cscf, ngx_http_core_loc_conf_t *pclcf);
38 static ngx_int_t ngx_http_init_static_location_trees(ngx_conf_t *cf,
39     ngx_http_core_loc_conf_t *pclcf);
40 static ngx_int_t ngx_http_cmp_locations(const ngx_queue_t *one,
41     const ngx_queue_t *two);
42 static ngx_int_t ngx_http_join_exact_locations(ngx_conf_t *cf,
43     ngx_queue_t *locations);
44 static void ngx_http_create_locations_list(ngx_queue_t *locations,
45     ngx_queue_t *q);
46 static ngx_http_location_tree_node_t *
47     ngx_http_create_locations_tree(ngx_conf_t *cf, ngx_queue_t *locations,
48     size_t prefix);
49
50 static ngx_int_t ngx_http_optimize_servers(ngx_conf_t *cf,
51     ngx_http_core_main_conf_t *cmcf, ngx_array_t *ports);
52 static ngx_int_t ngx_http_server_names(ngx_conf_t *cf,
53     ngx_http_core_main_conf_t *cmcf, ngx_http_conf_addr_t *addr);
54 static ngx_int_t ngx_http_cmp_conf_addrs(const void *one, const void *two);
55 static int ngx_libc_cdecl ngx_http_cmp_dns_wildcards(const void *one,
56     const void *two);

```



```

57
58 static ngx_int_t ngx_http_init_listening(ngx_conf_t *cf,
59     ngx_http_conf_port_t *port);
60 static ngx_listening_t *ngx_http_add_listening(ngx_conf_t *cf,
61     ngx_http_conf_addr_t *addr);
62 static ngx_int_t ngx_http_add_addrs(ngx_conf_t *cf, ngx_http_port_t *hport,
63     ngx_http_conf_addr_t *addr);
64 #if (NGX_HAVE_INET6)
65 static ngx_int_t ngx_http_add_addrs6(ngx_conf_t *cf, ngx_http_port_t *hport,
66     ngx_http_conf_addr_t *addr);
67 #endif
68
69 ngx_uint_t    ngx_http_max_module;
70
71
72 ngx_int_t    (*ngx_http_top_header_filter) (ngx_http_request_t *r);
73 ngx_int_t    (*ngx_http_top_body_filter) (ngx_http_request_t *r, ngx_chain_t *ch);
74
75
76 ngx_str_t    ngx_http_html_default_types[] = {
77     ngx_string("text/html"),
78     ngx_null_string
79 };
80
81
82 static ngx_command_t  ngx_http_commands[] = {
83
84     { ngx_string("http"),
85       NGX_MAIN_CONF|NGX_CONF_BLOCK|NGX_CONF_NOARGS,
86       ngx_http_block,
87       0,
88       0,
89       NULL },
90
91     ngx_null_command
92 };
93
94
95 static ngx_core_module_t  ngx_http_module_ctx = {
96     ngx_string("http"),
97     NULL,
98     NULL
99 };
100
101
102 ngx_module_t  ngx_http_module = {
103     NGX_MODULE_V1,
104     &ngx_http_module_ctx,          /* module context */
105     ngx_http_commands,            /* module directives */
106     NGX_CORE_MODULE,              /* module type */
107     NULL,                          /* init master */
108     NULL,                          /* init module */
109     NULL,                          /* init process */
110     NULL,                          /* init thread */
111     NULL,                          /* exit thread */
112     NULL,                          /* exit process */
113     NULL,                          /* exit master */
114     NGX_MODULE_V1_PADDING
115 };
116
117
118 static char *
119 ngx_http_block(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
120 {
121     char                *rv;
122     ngx_uint_t          mi, m, s;
123     ngx_conf_t          pcf;
124     ngx_http_module_t   *module;
125     ngx_http_conf_ctx_t *ctx;
126     ngx_http_core_loc_conf_t *clcf;
127     ngx_http_core_srv_conf_t **cscfp;
128     ngx_http_core_main_conf_t *cmcf;
129
130     /* the main http context */
131
132     ctx = ngx_palloc(cf->pool, sizeof(ngx_http_conf_ctx_t));

```

```

133 if (ctx == NULL) {
134     return NGX\_CONF\_ERROR;
135 }
136
137 \*\(ngx\_http\_conf\_ctx\_t \*\*\) conf = ctx;
138
139
140 /* count the number of the http modules and set up their indices */
141
142 ngx\_http\_max\_module = 0;
143 for (m = 0; ngx_modules[m]; m++) {
144     if (ngx_modules[m]->type != NGX\_HTTP\_MODULE) {
145         continue;
146     }
147
148     ngx_modules[m]->ctx_index = ngx\_http\_max\_module++;
149 }
150
151
152 /* the http main_conf context, it is the same in the all http contexts */
153
154 ctx->main_conf = ngx\_palloc(cf->pool,
155                             sizeof(void *) * ngx\_http\_max\_module);
156 if (ctx->main_conf == NULL) {
157     return NGX\_CONF\_ERROR;
158 }
159
160
161 /*
162  * the http null srv_conf context, it is used to merge
163  * the server{}s' srv_conf's
164  */
165
166 ctx->srv_conf = ngx\_palloc(cf->pool, sizeof(void *) * ngx\_http\_max\_module);
167 if (ctx->srv_conf == NULL) {
168     return NGX\_CONF\_ERROR;
169 }
170
171
172 /*
173  * the http null loc_conf context, it is used to merge
174  * the server{}s' loc_conf's
175  */
176
177 ctx->loc_conf = ngx\_palloc(cf->pool, sizeof(void *) * ngx\_http\_max\_module);
178 if (ctx->loc_conf == NULL) {
179     return NGX\_CONF\_ERROR;
180 }
181
182
183 /*
184  * create the main_conf's, the null srv_conf's, and the null loc_conf's
185  * of the all http modules
186  */
187
188 for (m = 0; ngx_modules[m]; m++) {
189     if (ngx_modules[m]->type != NGX\_HTTP\_MODULE) {
190         continue;
191     }
192
193     module = ngx_modules[m]->ctx;
194     mi = ngx_modules[m]->ctx_index;
195
196     if (module->create_main_conf) {
197         ctx->main_conf[mi] = module->create_main_conf(cf);
198         if (ctx->main_conf[mi] == NULL) {
199             return NGX\_CONF\_ERROR;
200         }
201     }
202
203     if (module->create_srv_conf) {
204         ctx->srv_conf[mi] = module->create_srv_conf(cf);
205         if (ctx->srv_conf[mi] == NULL) {
206             return NGX\_CONF\_ERROR;
207         }
208     }

```

```

209     if (module->create_loc_conf) {
210         ctx->loc_conf[mi] = module->create_loc_conf(cf);
211         if (ctx->loc_conf[mi] == NULL) {
212             return NGX_CONF_ERROR;
213         }
214     }
215 }
216
217 pcf = *cf;
218 cf->ctx = ctx;
219
220 for (m = 0; ngx_modules[m]; m++) {
221     if (ngx_modules[m]->type != NGX_HTTP_MODULE) {
222         continue;
223     }
224
225     module = ngx_modules[m]->ctx;
226
227     if (module->preconfiguration) {
228         if (module->preconfiguration(cf) != NGX_OK) {
229             return NGX_CONF_ERROR;
230         }
231     }
232 }
233
234 /* parse inside the http{} block */
235
236 cf->module_type = NGX_HTTP_MODULE;
237 cf->cmd_type = NGX_HTTP_MAIN_CONF;
238 rv = ngx_conf_parse(cf, NULL);
239
240 if (rv != NGX_CONF_OK) {
241     goto failed;
242 }
243
244 /*
245  * init http{} main_conf's, merge the server{}s' srv_conf's
246  * and its location{}s' loc_conf's
247  */
248
249 cmcf = ctx->main_conf[ngx_http_core_module.ctx_index];
250 cscfp = cmcf->servers.elts;
251
252 for (m = 0; ngx_modules[m]; m++) {
253     if (ngx_modules[m]->type != NGX_HTTP_MODULE) {
254         continue;
255     }
256
257     module = ngx_modules[m]->ctx;
258     mi = ngx_modules[m]->ctx_index;
259
260     /* init http{} main_conf's */
261
262     if (module->init_main_conf) {
263         rv = module->init_main_conf(cf, ctx->main_conf[mi]);
264         if (rv != NGX_CONF_OK) {
265             goto failed;
266         }
267     }
268 }
269
270 rv = ngx_http_merge_servers(cf, cmcf, module, mi);
271 if (rv != NGX_CONF_OK) {
272     goto failed;
273 }
274
275 /* create location trees */
276
277 for (s = 0; s < cmcf->servers.nelts; s++) {
278
279     clcf = cscfp[s]->ctx->loc_conf[ngx_http_core_module.ctx_index];
280
281     if (ngx_http_init_locations(cf, cscfp[s], clcf) != NGX_OK) {
282         return NGX_CONF_ERROR;

```

```

285     }
286
287     if (ngx\_http\_init\_static\_location\_trees(cf, clcf) != NGX\_OK) {
288         return NGX\_CONF\_ERROR;
289     }
290 }
291
292
293 if (ngx\_http\_init\_phases(cf, cmcf) != NGX\_OK) {
294     return NGX\_CONF\_ERROR;
295 }
296
297 if (ngx\_http\_init\_headers\_in\_hash(cf, cmcf) != NGX\_OK) {
298     return NGX\_CONF\_ERROR;
299 }
300
301
302 for (m = 0; ngx_modules[m]; m++) {
303     if (ngx_modules[m]->type != NGX\_HTTP\_MODULE) {
304         continue;
305     }
306
307     module = ngx_modules[m]->ctx;
308
309     if (module->postconfiguration) {
310         if (module->postconfiguration(cf) != NGX\_OK) {
311             return NGX\_CONF\_ERROR;
312         }
313     }
314 }
315
316 if (ngx\_http\_variables\_init\_vars(cf) != NGX\_OK) {
317     return NGX\_CONF\_ERROR;
318 }
319
320 /*
321  * http{}'s cf->ctx was needed while the configuration merging
322  * and in postconfiguration process
323  */
324
325 *cf = pcf;
326
327
328 if (ngx\_http\_init\_phase\_handlers(cf, cmcf) != NGX\_OK) {
329     return NGX\_CONF\_ERROR;
330 }
331
332
333 /* optimize the lists of ports, addresses and server names */
334
335 if (ngx\_http\_optimize\_servers(cf, cmcf, cmcf->ports) != NGX\_OK) {
336     return NGX\_CONF\_ERROR;
337 }
338
339 return NGX\_CONF\_OK;
340
341 failed:
342
343 *cf = pcf;
344
345 return rv;
346 }
347
348
349 static ngx\_int\_t
350 ngx\_http\_init\_phases(ngx\_conf\_t *cf, ngx\_http\_core\_main\_conf\_t *cmcf)
351 {
352     if (ngx\_array\_init(&cmcf->phases[NGX\_HTTP\_POST\_READ\_PHASE].handlers,
353         cf->pool, 1, sizeof\(ngx\_http\_handler\_pt\))
354         != NGX\_OK)
355     {
356         return NGX\_ERROR;
357     }
358
359     if (ngx\_array\_init(&cmcf->phases[NGX\_HTTP\_SERVER\_REWRITE\_PHASE].handlers,
360         cf->pool, 1, sizeof\(ngx\_http\_handler\_pt\))

```

```

361     != NGX\_OK)
362     {
363         return NGX\_ERROR;
364     }
365
366     if (ngx\_array\_init(&cmcf->phases[NGX_HTTP_REWRITE_PHASE].handlers,
367                     cf->pool, 1, sizeof\(ngx\_http\_handler\_pt\))
368         != NGX\_OK)
369     {
370         return NGX\_ERROR;
371     }
372
373     if (ngx\_array\_init(&cmcf->phases[NGX_HTTP_PREACCESS_PHASE].handlers,
374                     cf->pool, 1, sizeof\(ngx\_http\_handler\_pt\))
375         != NGX\_OK)
376     {
377         return NGX\_ERROR;
378     }
379
380     if (ngx\_array\_init(&cmcf->phases[NGX_HTTP_ACCESS_PHASE].handlers,
381                     cf->pool, 2, sizeof\(ngx\_http\_handler\_pt\))
382         != NGX\_OK)
383     {
384         return NGX\_ERROR;
385     }
386
387     if (ngx\_array\_init(&cmcf->phases[NGX_HTTP_CONTENT_PHASE].handlers,
388                     cf->pool, 4, sizeof\(ngx\_http\_handler\_pt\))
389         != NGX\_OK)
390     {
391         return NGX\_ERROR;
392     }
393
394     if (ngx\_array\_init(&cmcf->phases[NGX_HTTP_LOG_PHASE].handlers,
395                     cf->pool, 1, sizeof\(ngx\_http\_handler\_pt\))
396         != NGX\_OK)
397     {
398         return NGX\_ERROR;
399     }
400
401     return NGX\_OK;
402 }
403
404
405 static ngx\_int\_t
406 ngx\_http\_init\_headers\_in\_hash(ngx\_conf\_t *cf, ngx\_http\_core\_main\_conf\_t *cmcf)
407 {
408     ngx\_array\_t         headers_in;
409     ngx\_hash\_key\_t     *hk;
410     ngx\_hash\_init\_t     hash;
411     ngx\_http\_header\_t *header;
412
413     if (ngx\_array\_init(&headers_in, cf->temp_pool, 32, sizeof\(ngx\_hash\_key\_t\))
414         != NGX\_OK)
415     {
416         return NGX\_ERROR;
417     }
418
419     for (header = ngx\_http\_headers\_in; header->name.len; header++) {
420         hk = ngx\_array\_push(&headers_in);
421         if (hk == NULL) {
422             return NGX\_ERROR;
423         }
424
425         hk->key = header->name;
426         hk->key_hash = ngx\_hash\_key\_lc(header->name.data, header->name.len);
427         hk->value = header;
428     }
429
430     hash.hash = &cmcf->headers_in_hash;
431     hash.key = ngx\_hash\_key\_lc;
432     hash.max_size = 512;
433     hash.bucket_size = ngx\_align(64, ngx\_cacheline\_size);
434     hash.name = "headers_in_hash";
435     hash.pool = cf->pool;
436     hash.temp_pool = NULL;

```



```

513     }
514
515     continue;
516
517     case NGX_HTTP_ACCESS_PHASE:
518         checker = ngx\_http\_core\_access\_phase;
519         n++;
520         break;
521
522     case NGX_HTTP_POST_ACCESS_PHASE:
523         if (use_access) {
524             ph->checker = ngx\_http\_core\_post\_access\_phase;
525             ph->next = n;
526             ph++;
527         }
528
529         continue;
530
531     case NGX_HTTP_TRY_FILES_PHASE:
532         if (cmcf->try_files) {
533             ph->checker = ngx\_http\_core\_try\_files\_phase;
534             n++;
535             ph++;
536         }
537
538         continue;
539
540     case NGX_HTTP_CONTENT_PHASE:
541         checker = ngx\_http\_core\_content\_phase;
542         break;
543
544     default:
545         checker = ngx\_http\_core\_generic\_phase;
546     }
547
548     n += cmcf->phases[i].handlers.nelts;
549
550     for (j = cmcf->phases[i].handlers.nelts - 1; j >=0; j--) {
551         ph->checker = checker;
552         ph->handler = h[j];
553         ph->next = n;
554         ph++;
555     }
556 }
557
558 return NGX\_OK;
559 }
560
561
562 static char *
563 ngx\_http\_merge\_servers(ngx\_conf\_t *cf, ngx\_http\_core\_main\_conf\_t *cmcf,
564 ngx\_http\_module\_t *module, ngx\_uint\_t ctx_index)
565 {
566     char                *rv;
567     ngx\_uint\_t          s;
568     ngx\_http\_conf\_ctx\_t *ctx, saved;
569     ngx\_http\_core\_loc\_conf\_t *clcf;
570     ngx\_http\_core\_srv\_conf\_t **cscfp;
571
572     cscfp = cmcf->servers.elts;
573     ctx = (ngx\_http\_conf\_ctx\_t *) cf->ctx;
574     saved = *ctx;
575     rv = NGX\_CONF\_OK;
576
577     for (s = 0; s < cmcf->servers.nelts; s++) {
578
579         /* merge the server{s}' srv_conf's */
580
581         ctx->srv_conf = cscfp[s]->ctx->srv_conf;
582
583         if (module->merge_srv_conf) {
584             rv = module->merge_srv_conf(cf, saved.srv_conf[ctx_index],
585                                     cscfp[s]->ctx->srv_conf[ctx_index]);
586
587             if (rv != NGX\_CONF\_OK) {
588                 goto failed;
589             }
590         }
591     }

```

```

589     }
590
591     if (module->merge_loc_conf) {
592         /* merge the server{}'s loc_conf */
593
594         ctx->loc_conf = cscfp[s]->ctx->loc_conf;
595
596         rv = module->merge_loc_conf(cf, saved.loc_conf[ctx_index],
597                                   cscfp[s]->ctx->loc_conf[ctx_index]);
598         if (rv != NGX_CONF_OK) {
599             goto failed;
600         }
601     }
602
603     /* merge the locations{}' loc_conf's */
604
605     clcf = cscfp[s]->ctx->loc_conf[ngx_http_core_module.ctx_index];
606
607     rv = ngx_http_merge_locations(cf, clcf->locations,
608                                  cscfp[s]->ctx->loc_conf,
609                                  module, ctx_index);
610     if (rv != NGX_CONF_OK) {
611         goto failed;
612     }
613 }
614 }
615
616 failed:
617     *ctx = saved;
618
619     return rv;
620 }
621
622
623
624 static char *
625 ngx_http_merge_locations(ngx_conf_t *cf, ngx_queue_t *locations,
626                          void **loc_conf, ngx_http_module_t *module, ngx_uint_t ctx_index)
627 {
628     char *rv;
629     ngx_queue_t *q;
630     ngx_http_conf_ctx_t *ctx, saved;
631     ngx_http_core_loc_conf_t *clcf;
632     ngx_http_location_queue_t *lq;
633
634     if (locations == NULL) {
635         return NGX_CONF_OK;
636     }
637
638     ctx = (ngx_http_conf_ctx_t *) cf->ctx;
639     saved = *ctx;
640
641     for (q = ngx_queue_head(locations);
642          q != ngx_queue_sentinel(locations);
643          q = ngx_queue_next(q))
644     {
645         lq = (ngx_http_location_queue_t *) q;
646
647         clcf = lq->exact ? lq->exact : lq->inclusive;
648         ctx->loc_conf = clcf->loc_conf;
649
650         rv = module->merge_loc_conf(cf, loc_conf[ctx_index],
651                                   clcf->loc_conf[ctx_index]);
652         if (rv != NGX_CONF_OK) {
653             return rv;
654         }
655
656         rv = ngx_http_merge_locations(cf, clcf->locations, clcf->loc_conf,
657                                       module, ctx_index);
658         if (rv != NGX_CONF_OK) {
659             return rv;
660         }
661     }
662
663     *ctx = saved;
664

```



```

665     return NGX\_CONF\_OK;
666 }
667
668
669 static ngx\_int\_t
670 ngx\_http\_init\_locations(ngx\_conf\_t *cf, ngx\_http\_core\_srv\_conf\_t *cscf,
671 ngx\_http\_core\_loc\_conf\_t *pclcf)
672 {
673     ngx\_uint\_t                n;
674     ngx\_queue\_t              *q, *locations, *named, tail;
675     ngx\_http\_core\_loc\_conf\_t *clcf;
676     ngx\_http\_location\_queue\_t *lq;
677     ngx\_http\_core\_loc\_conf\_t **clcfp;
678 #if (NGX_PCRE)
679     ngx\_uint\_t                r;
680     ngx\_queue\_t              *regex;
681 #endif
682
683     locations = pclcf->locations;
684
685     if (locations == NULL) {
686         return NGX\_OK;
687     }
688
689     ngx\_queue\_sort(locations, ngx\_http\_cmp\_locations);
690
691     named = NULL;
692     n = 0;
693 #if (NGX_PCRE)
694     regex = NULL;
695     r = 0;
696 #endif
697
698     for (q = ngx\_queue\_head(locations);
699         q != ngx\_queue\_sentinel(locations);
700         q = ngx\_queue\_next(q))
701     {
702         lq = (ngx\_http\_location\_queue\_t *) q;
703
704         clcf = lq->exact ? lq->exact : lq->inclusive;
705
706         if (ngx\_http\_init\_locations(cf, NULL, clcf) != NGX\_OK) {
707             return NGX\_ERROR;
708         }
709
710 #if (NGX_PCRE)
711         if (clcf->regex) {
712             r++;
713
714             if (regex == NULL) {
715                 regex = q;
716             }
717
718             continue;
719         }
720     }
721 #endif
722
723     if (clcf->named) {
724         n++;
725
726         if (named == NULL) {
727             named = q;
728         }
729
730         continue;
731     }
732
733     if (clcf->noname) {
734         break;
735     }
736 }
737
738
739 if (q != ngx\_queue\_sentinel(locations)) {
740     ngx\_queue\_split(locations, q, &tail);

```

```

741 }
742
743 if (named) {
744     clcfp = ngx_palloc(cf->pool,
745                     (n + 1) * sizeof(ngx_http_core_loc_conf_t *));
746     if (clcfp == NULL) {
747         return NGX_ERROR;
748     }
749
750     cscf->named_locations = clcfp;
751
752     for (q = named;
753          q != ngx_queue_sentinel(locations);
754          q = ngx_queue_next(q))
755     {
756         lq = (ngx_http_location_queue_t *) q;
757
758         *(clcfp++) = lq->exact;
759     }
760
761     *clcfp = NULL;
762
763     ngx_queue_split(locations, named, &tail);
764 }
765
766 #if (NGX_PCRE)
767
768 if (regex) {
769
770     clcfp = ngx_palloc(cf->pool,
771                     (r + 1) * sizeof(ngx_http_core_loc_conf_t *));
772     if (clcfp == NULL) {
773         return NGX_ERROR;
774     }
775
776     pclcf->regex_locations = clcfp;
777
778     for (q = regex;
779          q != ngx_queue_sentinel(locations);
780          q = ngx_queue_next(q))
781     {
782         lq = (ngx_http_location_queue_t *) q;
783
784         *(clcfp++) = lq->exact;
785     }
786
787     *clcfp = NULL;
788
789     ngx_queue_split(locations, regex, &tail);
790 }
791
792 #endif
793
794 return NGX_OK;
795 }
796
797
798 static ngx_int_t
799 ngx_http_init_static_location_trees(ngx_conf_t *cf,
800 ngx_http_core_loc_conf_t *pclcf)
801 {
802     ngx_queue_t *q, *locations;
803     ngx_http_core_loc_conf_t *clcf;
804     ngx_http_location_queue_t *lq;
805
806     locations = pclcf->locations;
807
808     if (locations == NULL) {
809         return NGX_OK;
810     }
811
812     if (ngx_queue_empty(locations)) {
813         return NGX_OK;
814     }
815
816     for (q = ngx_queue_head(locations);

```

```

817     q != ngx_queue_sentinel(locations);
818     q = ngx_queue_next(q)
819 }
820     lq = (ngx_http_location_queue_t *) q;
821
822     clcf = lq->exact ? lq->exact : lq->inclusive;
823
824     if (ngx_http_init_static_location_trees(cf, clcf) != NGX_OK) {
825         return NGX_ERROR;
826     }
827 }
828
829 if (ngx_http_join_exact_locations(cf, locations) != NGX_OK) {
830     return NGX_ERROR;
831 }
832
833 ngx_http_create_locations_list(locations, ngx_queue_head(locations));
834
835 pclcf->static_locations = ngx_http_create_locations_tree(cf, locations, 0);
836 if (pclcf->static_locations == NULL) {
837     return NGX_ERROR;
838 }
839
840 return NGX_OK;
841 }
842
843
844 ngx_int_t
845 ngx_http_add_location(ngx_conf_t *cf, ngx_queue_t **locations,
846 ngx_http_core_loc_conf_t *clcf)
847 {
848     ngx_http_location_queue_t *lq;
849
850     if (*locations == NULL) {
851         *locations = ngx_palloc(cf->temp_pool,
852             sizeof(ngx_http_location_queue_t));
853         if (*locations == NULL) {
854             return NGX_ERROR;
855         }
856
857         ngx_queue_init(*locations);
858     }
859
860     lq = ngx_palloc(cf->temp_pool, sizeof(ngx_http_location_queue_t));
861     if (lq == NULL) {
862         return NGX_ERROR;
863     }
864
865     if (clcf->exact_match
866 #if (NGX_PCRE)
867         || clcf->regex
868 #endif
869         || clcf->named || clcf->noname)
870     {
871         lq->exact = clcf;
872         lq->inclusive = NULL;
873     }
874     else {
875         lq->exact = NULL;
876         lq->inclusive = clcf;
877     }
878
879     lq->name = &clcf->name;
880     lq->file_name = cf->conf_file->file.name.data;
881     lq->line = cf->conf_file->line;
882
883     ngx_queue_init(&lq->list);
884
885     ngx_queue_insert_tail(*locations, &lq->queue);
886
887     return NGX_OK;
888 }
889
890
891 static ngx_int_t
892 ngx_http_cmp_locations(const ngx_queue_t *one, const ngx_queue_t *two)

```

```

893 {
894     ngx_int_t          rc;
895     ngx_http_core_loc_conf_t  *first, *second;
896     ngx_http_location_queue_t  *lq1, *lq2;
897
898     lq1 = (ngx_http_location_queue_t *) one;
899     lq2 = (ngx_http_location_queue_t *) two;
900
901     first = lq1->exact ? lq1->exact : lq1->inclusive;
902     second = lq2->exact ? lq2->exact : lq2->inclusive;
903
904     if (first->noname && !second->noname) {
905         /* shift no named locations to the end */
906         return 1;
907     }
908
909     if (!first->noname && second->noname) {
910         /* shift no named locations to the end */
911         return -1;
912     }
913
914     if (first->noname || second->noname) {
915         /* do not sort no named locations */
916         return 0;
917     }
918
919     if (first->named && !second->named) {
920         /* shift named locations to the end */
921         return 1;
922     }
923
924     if (!first->named && second->named) {
925         /* shift named locations to the end */
926         return -1;
927     }
928
929     if (first->named && second->named) {
930         return ngx_strcmp(first->name.data, second->name.data);
931     }
932
933     #if (NGX_PCRE)
934
935     if (first->regex && !second->regex) {
936         /* shift the regex matches to the end */
937         return 1;
938     }
939
940     if (!first->regex && second->regex) {
941         /* shift the regex matches to the end */
942         return -1;
943     }
944
945     if (first->regex || second->regex) {
946         /* do not sort the regex matches */
947         return 0;
948     }
949
950     #endif
951
952     rc = ngx_filename_cmp(first->name.data, second->name.data,
953                          ngx_min(first->name.len, second->name.len) + 1);
954
955     if (rc == 0 && !first->exact_match && second->exact_match) {
956         /* an exact match must be before the same inclusive one */
957         return 1;
958     }
959
960     return rc;
961 }
962
963
964 static ngx_int_t
965 ngx_http_join_exact_locations(ngx_conf_t *cf, ngx_queue_t *locations)
966 {
967     ngx_queue_t  *q, *x;
968     ngx_http_location_queue_t  *lq, *lx;

```

```

969 q = ngx_queue_head(locations);
970
971
972 while (q != ngx_queue_last(locations)) {
973     x = ngx_queue_next(q);
974
975     lq = (ngx_http_location_queue_t *) q;
976     lx = (ngx_http_location_queue_t *) x;
977
978     if (lq->name->len == lx->name->len
979         && ngx_filename_cmp(lq->name->data, lx->name->data, lx->name->len)
980             == 0)
981     {
982         if ((lq->exact && lx->exact) || (lq->inclusive && lx->inclusive)) {
983             ngx_log_error(NGX_LOG_EMERG, cf->log, 0,
984                 "duplicate location \"%V\" in %s:%ui",
985                 lx->name, lx->file_name, lx->line);
986
987             return NGX_ERROR;
988         }
989     }
990
991     lq->inclusive = lx->inclusive;
992
993     ngx_queue_remove(x);
994
995     continue;
996 }
997
998 q = ngx_queue_next(q);
999 }
1000
1001 return NGX_OK;
1002 }
1003
1004
1005 static void
1006 ngx_http_create_locations_list(ngx_queue_t *locations, ngx_queue_t *q)
1007 {
1008     u_char          *name;
1009     size_t          len;
1010     ngx_queue_t     *x, tail;
1011     ngx_http_location_queue_t *lq, *lx;
1012
1013     if (q == ngx_queue_last(locations)) {
1014         return;
1015     }
1016
1017     lq = (ngx_http_location_queue_t *) q;
1018
1019     if (lq->inclusive == NULL) {
1020         ngx_http_create_locations_list(locations, ngx_queue_next(q));
1021         return;
1022     }
1023
1024     len = lq->name->len;
1025     name = lq->name->data;
1026
1027     for (x = ngx_queue_next(q);
1028         x != ngx_queue_sentinel(locations);
1029         x = ngx_queue_next(x))
1030     {
1031         lx = (ngx_http_location_queue_t *) x;
1032
1033         if (len > lx->name->len
1034             || ngx_filename_cmp(name, lx->name->data, len) != 0)
1035         {
1036             break;
1037         }
1038     }
1039
1040     q = ngx_queue_next(q);
1041
1042     if (q == x) {
1043         ngx_http_create_locations_list(locations, x);
1044         return;

```

```

1045     }
1046
1047     ngx_queue_split(locations, q, &tail);
1048     ngx_queue_add(&lq->list, &tail);
1049
1050     if (x == ngx_queue_sentinel(locations)) {
1051         ngx_http_create_locations_list(&lq->list, ngx_queue_head(&lq->list));
1052         return;
1053     }
1054
1055     ngx_queue_split(&lq->list, x, &tail);
1056     ngx_queue_add(locations, &tail);
1057
1058     ngx_http_create_locations_list(&lq->list, ngx_queue_head(&lq->list));
1059
1060     ngx_http_create_locations_list(locations, x);
1061 }
1062
1063
1064 /*
1065  * to keep cache locality for left leaf nodes, allocate nodes in following
1066  * order: node, left subtree, right subtree, inclusive subtree
1067  */
1068
1069 static ngx_http_location_tree_node_t *
1070 ngx_http_create_locations_tree(ngx_conf_t *cf, ngx_queue_t *locations,
1071     size_t prefix)
1072 {
1073     size_t len;
1074     ngx_queue_t *q, tail;
1075     ngx_http_location_queue_t *lq;
1076     ngx_http_location_tree_node_t *node;
1077
1078     q = ngx_queue_middle(locations);
1079
1080     lq = (ngx_http_location_queue_t *) q;
1081     len = lq->name->len - prefix;
1082
1083     node = ngx_palloc(cf->pool,
1084         offsetof(ngx_http_location_tree_node_t, name) + len);
1085     if (node == NULL) {
1086         return NULL;
1087     }
1088
1089     node->left = NULL;
1090     node->right = NULL;
1091     node->tree = NULL;
1092     node->exact = lq->exact;
1093     node->inclusive = lq->inclusive;
1094
1095     node->auto_redirect = (u_char) ((lq->exact && lq->exact->auto_redirect)
1096         || (lq->inclusive && lq->inclusive->auto_redirect));
1097
1098     node->len = (u_char) len;
1099     ngx_memcpy(node->name, &lq->name->data[prefix], len);
1100
1101     ngx_queue_split(locations, q, &tail);
1102
1103     if (ngx_queue_empty(locations)) {
1104         /*
1105          * ngx_queue_split() insures that if left part is empty,
1106          * then right one is empty too
1107          */
1108         goto inclusive;
1109     }
1110
1111     node->left = ngx_http_create_locations_tree(cf, locations, prefix);
1112     if (node->left == NULL) {
1113         return NULL;
1114     }
1115
1116     ngx_queue_remove(q);
1117
1118     if (ngx_queue_empty(&tail)) {
1119         goto inclusive;
1120     }

```

```

1121     node->right = ngx_http_create_locations_tree(cf, &tail, prefix);
1122     if (node->right == NULL) {
1123         return NULL;
1124     }
1125 }
1126
1127 inclusive:
1128
1129     if (ngx_queue_empty(&lq->list)) {
1130         return node;
1131     }
1132
1133     node->tree = ngx_http_create_locations_tree(cf, &lq->list, prefix + len);
1134     if (node->tree == NULL) {
1135         return NULL;
1136     }
1137
1138     return node;
1139 }
1140
1141
1142 ngx_int_t
1143 ngx_http_add_listen(ngx_conf_t *cf, ngx_http_core_srv_conf_t *cscf,
1144     ngx_http_listen_opt_t *lsopt)
1145 {
1146     in_port_t          p;
1147     ngx_uint_t         i;
1148     struct sockaddr    *sa;
1149     struct sockaddr_in *sin;
1150     ngx_http_conf_port_t *port;
1151     ngx_http_core_main_conf_t *cmcf;
1152     #if (NGX_HAVE_INET6)
1153     struct sockaddr_in6 *sin6;
1154     #endif
1155
1156     cmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_core_module);
1157
1158     if (cmcf->ports == NULL) {
1159         cmcf->ports = ngx_array_create(cf->temp_pool, 2,
1160             sizeof(ngx_http_conf_port_t));
1161         if (cmcf->ports == NULL) {
1162             return NGX_ERROR;
1163         }
1164     }
1165
1166     sa = &lsopt->u.sockaddr;
1167
1168     switch (sa->sa_family) {
1169
1170     #if (NGX_HAVE_INET6)
1171     case AF_INET6:
1172         sin6 = &lsopt->u.sockaddr_in6;
1173         p = sin6->sin6_port;
1174         break;
1175     #endif
1176
1177     #if (NGX_HAVE_UNIX_DOMAIN)
1178     case AF_UNIX:
1179         p = 0;
1180         break;
1181     #endif
1182
1183     default: /* AF_INET */
1184         sin = &lsopt->u.sockaddr_in;
1185         p = sin->sin_port;
1186         break;
1187     }
1188
1189     port = cmcf->ports->elts;
1190     for (i = 0; i < cmcf->ports->nelts; i++) {
1191
1192         if (p != port[i].port || sa->sa_family != port[i].family) {
1193             continue;
1194         }
1195
1196         /* a port is already in the port list */

```

```

1197     return ngx_http_add_addresses(cf, cscf, &port[i], lsopt);
1198 }
1199
1200 /* add a port to the port list */
1201
1202 port = ngx_array_push(cmcf->ports);
1203 if (port == NULL) {
1204     return NGX_ERROR;
1205 }
1206
1207 port->family = sa->sa_family;
1208 port->port = p;
1209 port->addrs.elts = NULL;
1210
1211 return ngx_http_add_address(cf, cscf, port, lsopt);
1212 }
1213
1214
1215
1216 static ngx_int_t
1217 ngx_http_add_addresses(ngx_conf_t *cf, ngx_http_core_srv_conf_t *cscf,
1218 ngx_http_conf_port_t *port, ngx_http_listen_opt_t *lsopt)
1219 {
1220     u_char                *p;
1221     size_t                len, off;
1222     ngx_uint_t           i, default_server;
1223     struct sockaddr      *sa;
1224     ngx_http_conf_addr_t *addr;
1225     #if (NGX_HAVE_UNIX_DOMAIN)
1226     struct sockaddr_un   *saun;
1227     #endif
1228     #if (NGX_HTTP_SSL)
1229     ngx_uint_t           ssl;
1230     #endif
1231     #if (NGX_HTTP_SPDY)
1232     ngx_uint_t           spdy;
1233     #endif
1234
1235     /*
1236      * we cannot compare whole sockaddr struct's as kernel
1237      * may fill some fields in inherited sockaddr struct's
1238      */
1239
1240     sa = &lsopt->u.sockaddr;
1241
1242     switch (sa->sa_family) {
1243
1244     #if (NGX_HAVE_INET6)
1245     case AF_INET6:
1246         off = offsetof(struct sockaddr_in6, sin6_addr);
1247         len = 16;
1248         break;
1249     #endif
1250
1251     #if (NGX_HAVE_UNIX_DOMAIN)
1252     case AF_UNIX:
1253         off = offsetof(struct sockaddr_un, sun_path);
1254         len = sizeof(saun->sun_path);
1255         break;
1256     #endif
1257
1258     default: /* AF_INET */
1259         off = offsetof(struct sockaddr_in, sin_addr);
1260         len = 4;
1261         break;
1262     }
1263
1264     p = lsopt->u.sockaddr_data + off;
1265
1266     addr = port->addrs.elts;
1267
1268     for (i = 0; i < port->addrs.nelts; i++) {
1269
1270         if (ngx_memcmp(p, addr[i].opt.u.sockaddr_data + off, len) != 0) {
1271             continue;
1272         }

```



```

1273     /* the address is already in the address list */
1274
1275
1276     if (ngx_http_add_server(cf, cscf, &addr[i]) != NGX_OK) {
1277         return NGX_ERROR;
1278     }
1279
1280     /* preserve default_server bit during listen options overwriting */
1281     default_server = addr[i].opt.default_server;
1282
1283     #if (NGX_HTTP_SSL)
1284         ssl = lsopt->ssl || addr[i].opt.ssl;
1285     #endif
1286     #if (NGX_HTTP_SPDY)
1287         spdy = lsopt->spdy || addr[i].opt.spdy;
1288     #endif
1289
1290     if (lsopt->set) {
1291
1292         if (addr[i].opt.set) {
1293             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1294                 "duplicate listen options for %s", addr[i].opt.addr);
1295             return NGX_ERROR;
1296         }
1297
1298         addr[i].opt = *lsopt;
1299     }
1300
1301     /* check the duplicate "default" server for this address:port */
1302
1303     if (lsopt->default_server) {
1304
1305         if (default_server) {
1306             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1307                 "a duplicate default server for %s", addr[i].opt.addr);
1308             return NGX_ERROR;
1309         }
1310
1311         default_server = 1;
1312         addr[i].default_server = cscf;
1313     }
1314
1315     addr[i].opt.default_server = default_server;
1316     #if (NGX_HTTP_SSL)
1317         addr[i].opt.ssl = ssl;
1318     #endif
1319     #if (NGX_HTTP_SPDY)
1320         addr[i].opt.spdy = spdy;
1321     #endif
1322
1323     return NGX_OK;
1324 }
1325
1326 /* add the address to the addresses list that bound to this port */
1327
1328 return ngx_http_add_address(cf, cscf, port, lsopt);
1329 }
1330
1331
1332 /*
1333  * add the server address, the server names and the server core module
1334  * configurations to the port list
1335  */
1336
1337 static ngx_int_t
1338 ngx_http_add_address(ngx_conf_t *cf, ngx_http_core_srv_conf_t *cscf,
1339     ngx_http_conf_port_t *port, ngx_http_listen_opt_t *lsopt)
1340 {
1341     ngx_http_conf_addr_t *addr;
1342
1343     if (port->addrs.elts == NULL) {
1344         if (ngx_array_init(&port->addrs, cf->temp_pool, 4,
1345             sizeof(ngx_http_conf_addr_t))
1346             != NGX_OK)
1347         {
1348             return NGX_ERROR;

```

```

1349     }
1350 }
1351
1352 #if (NGX_HTTP_SPDY && NGX_HTTP_SSL \
1353     && !defined TLSEXT_TYPE_application_layer_protocol_negotiation \
1354     && !defined TLSEXT_TYPE_next_proto_neg)
1355 if (lsopt->spdy && lsopt->ssl) {
1356     ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
1357         "nginx was built without OpenSSL ALPN or NPN "
1358         "support, SPDY is not enabled for %s", lsopt->addr);
1359 }
1360 #endif
1361
1362 addr = ngx_array_push(&port->addrs);
1363 if (addr == NULL) {
1364     return NGX_ERROR;
1365 }
1366
1367 addr->opt = *lsopt;
1368 addr->hash.buckets = NULL;
1369 addr->hash.size = 0;
1370 addr->wc_head = NULL;
1371 addr->wc_tail = NULL;
1372 #if (NGX_PCRE)
1373 addr->nregex = 0;
1374 addr->regex = NULL;
1375 #endif
1376 addr->default_server = cscf;
1377 addr->servers.elts = NULL;
1378
1379 return ngx_http_add_server(cf, cscf, addr);
1380 }
1381
1382
1383 /* add the server core module configuration to the address:port */
1384
1385 static ngx_int_t
1386 ngx_http_add_server(ngx_conf_t *cf, ngx_http_core_srv_conf_t *cscf,
1387     ngx_http_conf_addr_t *addr)
1388 {
1389     ngx_uint_t i;
1390     ngx_http_core_srv_conf_t **server;
1391
1392     if (addr->servers.elts == NULL) {
1393         if (ngx_array_init(&addr->servers, cf->temp_pool, 4,
1394             sizeof(ngx_http_core_srv_conf_t *)))
1395             != NGX_OK)
1396         {
1397             return NGX_ERROR;
1398         }
1399     }
1400     else {
1401         server = addr->servers.elts;
1402         for (i = 0; i < addr->servers.elts; i++) {
1403             if (server[i] == cscf) {
1404                 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1405                     "a duplicate listen %s", addr->opt.addr);
1406                 return NGX_ERROR;
1407             }
1408         }
1409     }
1410
1411     server = ngx_array_push(&addr->servers);
1412     if (server == NULL) {
1413         return NGX_ERROR;
1414     }
1415
1416     *server = cscf;
1417
1418     return NGX_OK;
1419 }
1420
1421
1422 static ngx_int_t
1423 ngx_http_optimize_servers(ngx_conf_t *cf, ngx_http_core_main_conf_t *cmcf,
1424     ngx_array_t *ports)

```

```

1425 {
1426     ngx_uint_t         p, a;
1427     ngx_http_conf_port_t *port;
1428     ngx_http_conf_addr_t *addr;
1429
1430     if (ports == NULL) {
1431         return NGX_OK;
1432     }
1433
1434     port = ports->elts;
1435     for (p = 0; p < ports->nelts; p++) {
1436
1437         ngx_sort(port[p].addrs.elts, (size_t) port[p].addrs.nelts,
1438             sizeof(ngx_http_conf_addr_t), ngx_http_cmp_conf_addrs);
1439
1440         /*
1441          * check whether all name-based servers have the same
1442          * configuration as a default server for given address:port
1443          */
1444
1445         addr = port[p].addrs.elts;
1446         for (a = 0; a < port[p].addrs.nelts; a++) {
1447
1448             if (addr[a].servers.nelts > 1
1449 #if (NGX_PCRE)
1450                 || addr[a].default_server->captures
1451 #endif
1452             )
1453             {
1454                 if (ngx_http_server_names(cf, cmcf, &addr[a]) != NGX_OK) {
1455                     return NGX_ERROR;
1456                 }
1457             }
1458
1459             if (ngx_http_init_listening(cf, &port[p]) != NGX_OK) {
1460                 return NGX_ERROR;
1461             }
1462         }
1463     }
1464
1465     return NGX_OK;
1466 }
1467
1468
1469 static ngx_int_t
1470 ngx_http_server_names(ngx_conf_t *cf, ngx_http_core_main_conf_t *cmcf,
1471     ngx_http_conf_addr_t *addr)
1472 {
1473     ngx_int_t         rc;
1474     ngx_uint_t         n, s;
1475     ngx_hash_init_t     hash;
1476     ngx_hash_keys_arrays_t ha;
1477     ngx_http_server_name_t *name;
1478     ngx_http_core_srv_conf_t **cscfp;
1479 #if (NGX_PCRE)
1480     ngx_uint_t         regex, i;
1481
1482     regex = 0;
1483 #endif
1484
1485     ngx_memzero(&ha, sizeof(ngx_hash_keys_arrays_t));
1486
1487     ha.temp_pool = ngx_create_pool(NGX_DEFAULT_POOL_SIZE, cf->log);
1488     if (ha.temp_pool == NULL) {
1489         return NGX_ERROR;
1490     }
1491
1492     ha.pool = cf->pool;
1493
1494     if (ngx_hash_keys_array_init(&ha, NGX_HASH_LARGE) != NGX_OK) {
1495         goto failed;
1496     }
1497
1498     cscfp = addr->servers.elts;
1499
1500     for (s = 0; s < addr->servers.nelts; s++) {

```

```

1501     name = cscfp[s]->server_names.elts;
1502
1503
1504     for (n = 0; n < cscfp[s]->server_names.nelts; n++) {
1505
1506     #if (NGX_PCRE)
1507         if (name[n].regex) {
1508             regex++;
1509             continue;
1510         }
1511     #endif
1512
1513     rc = ngx_hash_add_key(&ha, &name[n].name, name[n].server,
1514                          NGX_HASH_WILDCARD_KEY);
1515
1516     if (rc == NGX_ERROR) {
1517         return NGX_ERROR;
1518     }
1519
1520     if (rc == NGX_DECLINED) {
1521         ngx_log_error(NGX_LOG_EMERG, cf->log, 0,
1522                     "invalid server name or wildcard \"%V\" on %s",
1523                     &name[n].name, addr->opt.addr);
1524         return NGX_ERROR;
1525     }
1526
1527     if (rc == NGX_BUSY) {
1528         ngx_log_error(NGX_LOG_WARN, cf->log, 0,
1529                     "conflicting server name \"%V\" on %s, ignored",
1530                     &name[n].name, addr->opt.addr);
1531     }
1532 }
1533
1534
1535 hash.key = ngx_hash_key_lc;
1536 hash.max_size = cmcf->server_names_hash_max_size;
1537 hash.bucket_size = cmcf->server_names_hash_bucket_size;
1538 hash.name = "server_names_hash";
1539 hash.pool = cf->pool;
1540
1541 if (ha.keys.nelts) {
1542     hash.hash = &addr->hash;
1543     hash.temp_pool = NULL;
1544
1545     if (ngx_hash_init(&hash, ha.keys.elts, ha.keys.nelts) != NGX_OK) {
1546         goto failed;
1547     }
1548 }
1549
1550 if (ha.dns_wc_head.nelts) {
1551
1552     ngx_qsort(ha.dns_wc_head.elts, (size_t) ha.dns_wc_head.nelts,
1553              sizeof(ngx_hash_key_t), ngx_http_cmp_dns_wildcards);
1554
1555     hash.hash = NULL;
1556     hash.temp_pool = ha.temp_pool;
1557
1558     if (ngx_hash_wildcard_init(&hash, ha.dns_wc_head.elts,
1559                               ha.dns_wc_head.nelts)
1560         != NGX_OK)
1561     {
1562         goto failed;
1563     }
1564
1565     addr->wc_head = (ngx_hash_wildcard_t *) hash.hash;
1566 }
1567
1568 if (ha.dns_wc_tail.nelts) {
1569
1570     ngx_qsort(ha.dns_wc_tail.elts, (size_t) ha.dns_wc_tail.nelts,
1571              sizeof(ngx_hash_key_t), ngx_http_cmp_dns_wildcards);
1572
1573     hash.hash = NULL;
1574     hash.temp_pool = ha.temp_pool;
1575
1576     if (ngx_hash_wildcard_init(&hash, ha.dns_wc_tail.elts,

```

```

1577                                     ha.dns_wc_tail.nelts)
1578         != NGX\_OK)
1579     {
1580         goto failed;
1581     }
1582
1583     addr->wc_tail = (ngx\_hash\_wildcard\_t *) hash.hash;
1584 }
1585
1586 ngx\_destroy\_pool(ha.temp_pool);
1587
1588 #if (NGX\_PCRE)
1589
1590     if (regex == 0) {
1591         return NGX\_OK;
1592     }
1593
1594     addr->nregex = regex;
1595     addr->regex = ngx\_palloc(cf->pool, regex * sizeof(ngx\_http\_server\_name\_t));
1596     if (addr->regex == NULL) {
1597         return NGX\_ERROR;
1598     }
1599
1600     i = 0;
1601
1602     for (s = 0; s < addr->servers.nelts; s++) {
1603
1604         name = cscfp[s]->server_names.elts;
1605
1606         for (n = 0; n < cscfp[s]->server_names.nelts; n++) {
1607             if (name[n].regex) {
1608                 addr->regex[i++] = name[n];
1609             }
1610         }
1611     }
1612
1613 #endif
1614
1615     return NGX\_OK;
1616
1617 failed:
1618
1619     ngx\_destroy\_pool(ha.temp_pool);
1620
1621     return NGX\_ERROR;
1622 }
1623
1624
1625 static ngx\_int\_t
1626 ngx\_http\_cmp\_conf\_addrs(const void *one, const void *two)
1627 {
1628     ngx\_http\_conf\_addr\_t *first, *second;
1629
1630     first = (ngx\_http\_conf\_addr\_t *) one;
1631     second = (ngx\_http\_conf\_addr\_t *) two;
1632
1633     if (first->opt.wildcard) {
1634         /* a wildcard address must be the last resort, shift it to the end */
1635         return 1;
1636     }
1637
1638     if (second->opt.wildcard) {
1639         /* a wildcard address must be the last resort, shift it to the end */
1640         return -1;
1641     }
1642
1643     if (first->opt.bind && !second->opt.bind) {
1644         /* shift explicit bind()ed addresses to the start */
1645         return -1;
1646     }
1647
1648     if (!first->opt.bind && second->opt.bind) {
1649         /* shift explicit bind()ed addresses to the start */
1650         return 1;
1651     }
1652

```

```

1653     /* do not sort by default */
1654
1655     return 0;
1656 }
1657
1658
1659 static int ngx_libc cdecl
1660 ngx_http_cmp_dns_wildcards(const void *one, const void *two)
1661 {
1662     ngx_hash_key_t *first, *second;
1663
1664     first = (ngx_hash_key_t *) one;
1665     second = (ngx_hash_key_t *) two;
1666
1667     return ngx_dns_strcmp(first->key.data, second->key.data);
1668 }
1669
1670
1671 static ngx_int_t
1672 ngx_http_init_listening(ngx_conf_t *cf, ngx_http_conf_port_t *port)
1673 {
1674     ngx_uint_t i, last, bind_wildcard;
1675     ngx_listening_t *ls;
1676     ngx_http_port_t *hport;
1677     ngx_http_conf_addr_t *addr;
1678
1679     addr = port->addrs.elts;
1680     last = port->addrs.nelts;
1681
1682     /*
1683     * If there is a binding to an " *:port" then we need to bind() to
1684     * the " *:port" only and ignore other implicit bindings. The bindings
1685     * have been already sorted: explicit bindings are on the start, then
1686     * implicit bindings go, and wildcard binding is in the end.
1687     */
1688
1689     if (addr[last - 1].opt.wildcard) {
1690         addr[last - 1].opt.bind = 1;
1691         bind_wildcard = 1;
1692     } else {
1693         bind_wildcard = 0;
1694     }
1695
1696     i = 0;
1697
1698     while (i < last) {
1699
1700         if (bind_wildcard && !addr[i].opt.bind) {
1701             i++;
1702             continue;
1703         }
1704
1705         ls = ngx_http_add_listening(cf, &addr[i]);
1706         if (ls == NULL) {
1707             return NGX_ERROR;
1708         }
1709
1710         hport = ngx_palloc(cf->pool, sizeof(ngx_http_port_t));
1711         if (hport == NULL) {
1712             return NGX_ERROR;
1713         }
1714
1715         ls->servers = hport;
1716
1717         if (i == last - 1) {
1718             hport->naddrs = last;
1719
1720         } else {
1721             hport->naddrs = 1;
1722             i = 0;
1723         }
1724
1725         switch (ls->sockaddr->sa_family) {
1726
1727         #if (NGX_HAVE_INET6)

```

```

1729     case AF_INET6:
1730         if (ngx\_http\_add\_addrs6(cf, hport, addr) != NGX\_OK) {
1731             return NGX\_ERROR;
1732         }
1733         break;
1734 #endif
1735     default: /* AF\_INET */
1736         if (ngx\_http\_add\_addrs(cf, hport, addr) != NGX\_OK) {
1737             return NGX\_ERROR;
1738         }
1739         break;
1740     }
1741 }
1742     addr++;
1743     last--;
1744 }
1745
1746     return NGX\_OK;
1747 }
1748
1749
1750 static ngx\_listening\_t *
1751 ngx\_http\_add\_listening(ngx\_conf\_t *cf, ngx\_http\_conf\_addr\_t *addr)
1752 {
1753     ngx\_listening\_t *ls;
1754     ngx\_http\_core\_loc\_conf\_t *clcf;
1755     ngx\_http\_core\_srv\_conf\_t *cscf;
1756
1757     ls = ngx\_create\_listening(cf, &addr->opt.u.sockaddr, addr->opt.socklen);
1758     if (ls == NULL) {
1759         return NULL;
1760     }
1761
1762     ls->addr_ntop = 1;
1763
1764     ls->handler = ngx\_http\_init\_connection;
1765
1766     cscf = addr->default_server;
1767     ls->pool_size = cscf->connection_pool_size;
1768     ls->post_accept_timeout = cscf->client_header_timeout;
1769
1770     clcf = cscf->ctx->loc_conf[ngx\_http\_core\_module.ctx_index];
1771
1772     ls->logp = clcf->error_log;
1773     ls->log.data = &ls->addr_text;
1774     ls->log.handler = ngx\_accept\_log\_error;
1775
1776 #if (NGX\_WIN32)
1777     {
1778         ngx\_ioconf\_t *iocpcf = NULL;
1779
1780         if (ngx\_get\_conf(cf->cycle->conf_ctx, ngx\_events\_module)) {
1781             iocpcf = ngx\_event\_get\_conf(cf->cycle->conf_ctx, ngx\_ioconf\_module);
1782         }
1783         if (iocpcf && iocpcf->acceptex_read) {
1784             ls->post_accept_buffer_size = cscf->client_header_buffer_size;
1785         }
1786     }
1787 #endif
1788
1789     ls->backlog = addr->opt.backlog;
1790     ls->rcvbuf = addr->opt.rcvbuf;
1791     ls->sndbuf = addr->opt.sndbuf;
1792
1793     ls->keepalive = addr->opt.so_keepalive;
1794 #if (NGX\_HAVE\_KEEPALIVE\_TUNABLE)
1795     ls->keepidle = addr->opt.tcp_keepidle;
1796     ls->keepintvl = addr->opt.tcp_keepintvl;
1797     ls->keepcnt = addr->opt.tcp_keepcnt;
1798 #endif
1799
1800 #if (NGX\_HAVE\_DEFERRED\_ACCEPT && defined SO\_ACCEPTFILTER)
1801     ls->accept_filter = addr->opt.accept_filter;
1802 #endif
1803
1804 #if (NGX\_HAVE\_DEFERRED\_ACCEPT && defined TCP\_DEFER\_ACCEPT)

```

```

1805     ls->deferred_accept = addr->opt.deferred_accept;
1806 #endif
1807
1808 #if (NGX_HAVE_INET6 && defined IPV6_V6ONLY)
1809     ls->ipv6only = addr->opt.ipv6only;
1810 #endif
1811
1812 #if (NGX_HAVE_SETFIB)
1813     ls->setfib = addr->opt.setfib;
1814 #endif
1815
1816 #if (NGX_HAVE_TCP_FASTOPEN)
1817     ls->fastopen = addr->opt.fastopen;
1818 #endif
1819
1820     return ls;
1821 }
1822
1823
1824 static ngx_int_t
1825 ngx_http_add_addr(ngx_conf_t *cf, ngx_http_port_t *hport,
1826                 ngx_http_conf_addr_t *addr)
1827 {
1828     ngx_uint_t          i;
1829     ngx_http_in_addr_t  *addrs;
1830     struct sockaddr_in  *sin;
1831     ngx_http_virtual_names_t *vn;
1832
1833     hport->addrs = ngx_palloc(cf->pool,
1834                             hport->naddrs * sizeof(ngx_http_in_addr_t));
1835     if (hport->addrs == NULL) {
1836         return NGX_ERROR;
1837     }
1838
1839     addrs = hport->addrs;
1840
1841     for (i = 0; i < hport->naddrs; i++) {
1842
1843         sin = &addr[i].opt.u.sockaddr_in;
1844         addrs[i].addr = sin->sin_addr.s_addr;
1845         addrs[i].conf.default_server = addr[i].default_server;
1846 #if (NGX_HTTP_SSL)
1847         addrs[i].conf.ssl = addr[i].opt.ssl;
1848 #endif
1849 #if (NGX_HTTP_SPDY)
1850         addrs[i].conf.spdy = addr[i].opt.spdy;
1851 #endif
1852         addrs[i].conf.proxy_protocol = addr[i].opt.proxy_protocol;
1853
1854         if (addr[i].hash.buckets == NULL
1855             && (addr[i].wc_head == NULL
1856                || addr[i].wc_head->hash.buckets == NULL)
1857             && (addr[i].wc_tail == NULL
1858                || addr[i].wc_tail->hash.buckets == NULL)
1859 #if (NGX_PCRE)
1860             && addr[i].nregex == 0
1861 #endif
1862         )
1863         {
1864             continue;
1865         }
1866
1867         vn = ngx_palloc(cf->pool, sizeof(ngx_http_virtual_names_t));
1868         if (vn == NULL) {
1869             return NGX_ERROR;
1870         }
1871
1872         addrs[i].conf.virtual_names = vn;
1873
1874         vn->names.hash = addr[i].hash;
1875         vn->names.wc_head = addr[i].wc_head;
1876         vn->names.wc_tail = addr[i].wc_tail;
1877 #if (NGX_PCRE)
1878         vn->nregex = addr[i].nregex;
1879         vn->regex = addr[i].regex;
1880 #endif

```



```

1881     }
1882
1883     return NGX_OK;
1884 }
1885
1886
1887 #if (NGX_HAVE_INET6)
1888
1889 static ngx_int_t
1890 ngx_http_add_addrs6(ngx_conf_t *cf, ngx_http_port_t *hport,
1891                    ngx_http_conf_addr_t *addr)
1892 {
1893     ngx_uint_t          i;
1894     ngx_http_in6_addr_t *addrs6;
1895     struct sockaddr_in6 *sin6;
1896     ngx_http_virtual_names_t *vn;
1897
1898     hport->addrs = ngx_palloc(cf->pool,
1899                             hport->naddrs * sizeof(ngx_http_in6_addr_t));
1900     if (hport->addrs == NULL) {
1901         return NGX_ERROR;
1902     }
1903
1904     addrs6 = hport->addrs;
1905
1906     for (i = 0; i < hport->naddrs; i++) {
1907
1908         sin6 = &addr[i].opt.u.sockaddr_in6;
1909         addrs6[i].addr6 = sin6->sin6_addr;
1910         addrs6[i].conf.default_server = addr[i].default_server;
1911 #if (NGX_HTTP_SSL)
1912         addrs6[i].conf.ssl = addr[i].opt.ssl;
1913 #endif
1914 #if (NGX_HTTP_SPDY)
1915         addrs6[i].conf.spdy = addr[i].opt.spdy;
1916 #endif
1917
1918         if (addr[i].hash.buckets == NULL
1919             && (addr[i].wc_head == NULL
1920                || addr[i].wc_head->hash.buckets == NULL)
1921             && (addr[i].wc_tail == NULL
1922                || addr[i].wc_tail->hash.buckets == NULL)
1923 #if (NGX_PCRE)
1924             && addr[i].nregex == 0
1925 #endif
1926         )
1927         {
1928             continue;
1929         }
1930
1931         vn = ngx_palloc(cf->pool, sizeof(ngx_http_virtual_names_t));
1932         if (vn == NULL) {
1933             return NGX_ERROR;
1934         }
1935
1936         addrs6[i].conf.virtual_names = vn;
1937
1938         vn->names.hash = addr[i].hash;
1939         vn->names.wc_head = addr[i].wc_head;
1940         vn->names.wc_tail = addr[i].wc_tail;
1941 #if (NGX_PCRE)
1942         vn->nregex = addr[i].nregex;
1943         vn->regex = addr[i].regex;
1944 #endif
1945     }
1946
1947     return NGX_OK;
1948 }
1949
1950 #endif
1951
1952
1953 char *
1954 ngx_http_types_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
1955 {
1956     char *p = conf;

```

```

1957     ngx_array_t      **types;
1958     ngx_str_t       *value, *default_type;
1959     ngx_uint_t      i, n, hash;
1960     ngx_hash_key_t *type;
1961
1962
1963     types = (ngx_array_t **) (p + cmd->offset);
1964
1965     if (*types == (void *) -1) {
1966         return NGX_CONF_OK;
1967     }
1968
1969     default_type = cmd->post;
1970
1971     if (*types == NULL) {
1972         *types = ngx_array_create(cf->temp_pool, 1, sizeof(ngx_hash_key_t));
1973         if (*types == NULL) {
1974             return NGX_CONF_ERROR;
1975         }
1976
1977         if (default_type) {
1978             type = ngx_array_push(*types);
1979             if (type == NULL) {
1980                 return NGX_CONF_ERROR;
1981             }
1982
1983             type->key = *default_type;
1984             type->key_hash = ngx_hash_key(default_type->data,
1985                                         default_type->len);
1986             type->value = (void *) 4;
1987         }
1988     }
1989
1990     value = cf->args->elts;
1991
1992     for (i = 1; i < cf->args->nelts; i++) {
1993
1994         if (value[i].len == 1 && value[i].data[0] == '*') {
1995             *types = (void *) -1;
1996             return NGX_CONF_OK;
1997         }
1998
1999         hash = ngx_hash_strlow(value[i].data, value[i].data, value[i].len);
2000         value[i].data[value[i].len] = '\0';
2001
2002         type = (*types)->elts;
2003         for (n = 0; n < (*types)->nelts; n++) {
2004
2005             if (ngx_strcmp(value[i].data, type[n].key.data) == 0) {
2006                 ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
2007                                     "duplicate MIME type \"%V\"", &value[i]);
2008                 goto next;
2009             }
2010         }
2011
2012         type = ngx_array_push(*types);
2013         if (type == NULL) {
2014             return NGX_CONF_ERROR;
2015         }
2016
2017         type->key = value[i];
2018         type->key_hash = hash;
2019         type->value = (void *) 4;
2020
2021     next:
2022
2023         continue;
2024     }
2025
2026     return NGX_CONF_OK;
2027 }
2028
2029
2030 char *
2031 ngx_http_merge_types(ngx_conf_t *cf, ngx_array_t **keys, ngx_hash_t *types_hash,
2032                     ngx_array_t **prev_keys, ngx_hash_t *prev_types_hash,

```

```

2033     ngx\_str\_t *default_types)
2034 {
2035     ngx\_hash\_init\_t hash;
2036
2037     if (*keys) {
2038
2039         if (*keys == (void *) -1) {
2040             return NGX\_CONF\_OK;
2041         }
2042
2043         hash.hash = types_hash;
2044         hash.key = NULL;
2045         hash.max_size = 2048;
2046         hash.bucket_size = 64;
2047         hash.name = "test_types_hash";
2048         hash.pool = cf->pool;
2049         hash.temp_pool = NULL;
2050
2051         if (ngx\_hash\_init(&hash, (*keys)->elts, (*keys)->nelts) != NGX\_OK) {
2052             return NGX\_CONF\_ERROR;
2053         }
2054
2055         return NGX\_CONF\_OK;
2056     }
2057
2058     if (prev_types_hash->buckets == NULL) {
2059
2060         if (*prev_keys == NULL) {
2061
2062             if (ngx\_http\_set\_default\_types(cf, prev_keys, default_types)
2063                 != NGX\_OK)
2064             {
2065                 return NGX\_CONF\_ERROR;
2066             }
2067
2068             } else if (*prev_keys == (void *) -1) {
2069                 *keys = *prev_keys;
2070                 return NGX\_CONF\_OK;
2071             }
2072
2073             hash.hash = prev_types_hash;
2074             hash.key = NULL;
2075             hash.max_size = 2048;
2076             hash.bucket_size = 64;
2077             hash.name = "test_types_hash";
2078             hash.pool = cf->pool;
2079             hash.temp_pool = NULL;
2080
2081             if (ngx\_hash\_init(&hash, (*prev_keys)->elts, (*prev_keys)->nelts)
2082                 != NGX\_OK)
2083             {
2084                 return NGX\_CONF\_ERROR;
2085             }
2086         }
2087
2088         *types_hash = *prev_types_hash;
2089
2090         return NGX\_CONF\_OK;
2091     }
2092 }
2093
2094
2095 ngx\_int\_t
2096 ngx\_http\_set\_default\_types(ngx\_conf\_t *cf, ngx\_array\_t **types,
2097     ngx\_str\_t *default_type)
2098 {
2099     ngx\_hash\_key\_t *type;
2100
2101     *types = ngx\_array\_create(cf->temp_pool, 1, sizeof(ngx\_hash\_key\_t));
2102     if (*types == NULL) {
2103         return NGX\_ERROR;
2104     }
2105
2106     while (default_type->len) {
2107         type = ngx\_array\_push(*types);

```

```
2109     if (type == NULL) {
2110         return NGX\_ERROR;
2111     }
2112
2113     type->key = *default_type;
2114     type->key_hash = ngx\_hash\_key(default_type->data,
2115                                   default_type->len);
2116     type->value = (void *) 4;
2117
2118     default_type++;
2119 }
2120
2121 return NGX\_OK;
2122 }
```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_core.h - nginx-1.7.10

### Data types defined

- [ngx\\_chain\\_t](#)
- [ngx\\_command\\_t](#)
- [ngx\\_conf\\_t](#)
- [ngx\\_connection\\_handler\\_pt](#)
- [ngx\\_connection\\_t](#)
- [ngx\\_cycle\\_t](#)
- [ngx\\_event\\_aio\\_t](#)
- [ngx\\_event\\_handler\\_pt](#)
- [ngx\\_event\\_t](#)
- [ngx\\_file\\_t](#)
- [ngx\\_log\\_t](#)
- [ngx\\_module\\_t](#)
- [ngx\\_open\\_file\\_t](#)
- [ngx\\_pool\\_t](#)

### Macros defined

- [CR](#)
- [CRLF](#)
- [LF](#)
- [NGX\\_ABORT](#)
- [NGX\\_AGAIN](#)
- [NGX\\_BUSY](#)
- [NGX\\_DECLINED](#)
- [NGX\\_DISABLE\\_SYMLINKS\\_NOTOWNER](#)
- [NGX\\_DISABLE\\_SYMLINKS\\_OFF](#)
- [NGX\\_DISABLE\\_SYMLINKS\\_ON](#)
- [NGX\\_DONE](#)
- [NGX\\_ERROR](#)
- [NGX\\_OK](#)
- [\\_NGX\\_CORE\\_H\\_INCLUDED](#)

- [ngx\\_abs](#)
- [ngx\\_max](#)
- [ngx\\_min](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_CORE\_H\_INCLUDED
9 #define \_NGX\_CORE\_H\_INCLUDED
10
11
12 typedef struct ngx\_module\_s      ngx_module_t;
13 typedef struct ngx\_conf\_s        ngx_conf_t;
14 typedef struct ngx\_cycle\_s       ngx_cycle_t;
15 typedef struct ngx\_pool\_s        ngx_pool_t;
16 typedef struct ngx\_chain\_s       ngx_chain_t;
17 typedef struct ngx\_log\_s         ngx_log_t;
18 typedef struct ngx\_open\_file\_s    ngx_open_file_t;
19 typedef struct ngx\_command\_s       ngx_command_t;
20 typedef struct ngx\_file\_s         ngx_file_t;
21 typedef struct ngx\_event\_s         ngx_event_t;
22 typedef struct ngx\_event\_aio\_s      ngx_event_aio_t;
23 typedef struct ngx\_connection\_s    ngx_connection_t;
24
25 typedef void (*ngx\_event\_handler\_pt)(ngx\_event\_t *ev);
26 typedef void (*ngx\_connection\_handler\_pt)(ngx\_connection\_t *c);
27
28
29 #define NGX\_OK          0
30 #define NGX\_ERROR      -1
31 #define NGX\_AGAIN     -2
32 #define NGX\_BUSY       -3
33 #define NGX\_DONE       -4
34 #define NGX\_DECLINED  -5
35 #define NGX\_ABORT    -6
36
37
38 #include <ngx\_errno.h>
39 #include <ngx\_atomic.h>
40 #include <ngx\_thread.h>
41 #include <ngx\_rbtree.h>
42 #include <ngx\_time.h>
43 #include <ngx\_socket.h>
44 #include <ngx\_string.h>
45 #include <ngx\_files.h>
46 #include <ngx\_shmem.h>
47 #include <ngx\_process.h>
48 #include <ngx\_user.h>
49 #include <ngx\_parse.h>
50 #include <ngx\_log.h>
51 #include <ngx\_alloc.h>
52 #include <ngx\_palloc.h>
53 #include <ngx\_buf.h>
54 #include <ngx\_queue.h>
55 #include <ngx\_array.h>
56 #include <ngx\_list.h>
57 #include <ngx\_hash.h>
58 #include <ngx\_file.h>
59 #include <ngx\_crc.h>
60 #include <ngx\_crc32.h>
61 #include <ngx\_murmurhash.h>
62 #if (NGX_PCRE)
63 #include <ngx\_regex.h>
64 #endif
65 #include <ngx\_radix\_tree.h>

```

```
66 #include <ngx_times.h>
67 #include <ngx_shmtx.h>
68 #include <ngx_slab.h>
69 #include <ngx_inet.h>
70 #include <ngx_cycle.h>
71 #include <ngx_resolver.h>
72 #if (NGX_OPENSSL)
73 #include <ngx_event_openssl.h>
74 #endif
75 #include <ngx_process_cycle.h>
76 #include <ngx_conf_file.h>
77 #include <ngx_open_file_cache.h>
78 #include <ngx_os.h>
79 #include <ngx_connection.h>
80 #include <ngx_syslog.h>
81 #include <ngx_proxy_protocol.h>
82
83
84 #define LF      (u_char) '\n'
85 #define CR      (u_char) '\r'
86 #define CRLF    "\r\n"
87
88
89 #define ngx_abs(value)      (((value) >= 0) ? (value) : - (value))
90 #define ngx_max(val1, val2) ((val1 < val2) ? (val2) : (val1))
91 #define ngx_min(val1, val2) ((val1 > val2) ? (val2) : (val1))
92
93 void ngx_cpuidinfo(void);
94
95 #if (NGX_HAVE_OPENAT)
96 #define NGX_DISABLE_SYMLINKS_OFF      0
97 #define NGX_DISABLE_SYMLINKS_ON      1
98 #define NGX_DISABLE_SYMLINKS_NOTOWNER 2
99 #endif
100
101 #endif /* NGX_CORE_H_INCLUDED */
```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_conf\_file.h - nginx-1.7.10

### Data types defined

- [ngx\\_command\\_s](#)
- [ngx\\_conf\\_bitmask\\_t](#)
- [ngx\\_conf\\_deprecated\\_t](#)
- [ngx\\_conf\\_enum\\_t](#)
- [ngx\\_conf\\_file\\_t](#)
- [ngx\\_conf\\_handler\\_pt](#)
- [ngx\\_conf\\_num\\_bounds\\_t](#)
- [ngx\\_conf\\_post\\_handler\\_pt](#)
- [ngx\\_conf\\_post\\_t](#)
- [ngx\\_conf\\_s](#)
- [ngx\\_core\\_module\\_t](#)
- [ngx\\_module\\_s](#)
- [ngx\\_open\\_file\\_s](#)

### Macros defined

- [NGX\\_ANY\\_CONF](#)
- [NGX\\_CONF\\_1MORE](#)
- [NGX\\_CONF\\_2MORE](#)
- [NGX\\_CONF\\_ANY](#)
- [NGX\\_CONF\\_ARGS\\_NUMBER](#)
- [NGX\\_CONF\\_BITMASK\\_SET](#)
- [NGX\\_CONF\\_BLOCK](#)
- [NGX\\_CONF\\_BLOCK\\_DONE](#)
- [NGX\\_CONF\\_BLOCK\\_START](#)
- [NGX\\_CONF\\_ERROR](#)
- [NGX\\_CONF\\_FILE\\_DONE](#)
- [NGX\\_CONF\\_FLAG](#)
- [NGX\\_CONF\\_MAX\\_ARGS](#)
- [NGX\\_CONF\\_MODULE](#)
- [NGX\\_CONF\\_MULTI](#)



- [NGX\\_CONF\\_NOARGS](#)
- [NGX\\_CONF\\_OK](#)
- [NGX\\_CONF\\_TAKE1](#)
- [NGX\\_CONF\\_TAKE12](#)
- [NGX\\_CONF\\_TAKE123](#)
- [NGX\\_CONF\\_TAKE1234](#)
- [NGX\\_CONF\\_TAKE13](#)
- [NGX\\_CONF\\_TAKE2](#)
- [NGX\\_CONF\\_TAKE23](#)
- [NGX\\_CONF\\_TAKE3](#)
- [NGX\\_CONF\\_TAKE4](#)
- [NGX\\_CONF\\_TAKE5](#)
- [NGX\\_CONF\\_TAKE6](#)
- [NGX\\_CONF\\_TAKE7](#)
- [NGX\\_CONF\\_UNSET](#)
- [NGX\\_CONF\\_UNSET\\_MSEC](#)
- [NGX\\_CONF\\_UNSET\\_PTR](#)
- [NGX\\_CONF\\_UNSET\\_SIZE](#)
- [NGX\\_CONF\\_UNSET\\_UINT](#)
- [NGX\\_CORE\\_MODULE](#)
- [NGX\\_DIRECT\\_CONF](#)
- [NGX\\_MAIN\\_CONF](#)
- [NGX\\_MAX\\_CONF\\_ERRSTR](#)
- [NGX\\_MODULE\\_V1](#)
- [NGX\\_MODULE\\_V1\\_PADDING](#)
- [\\_NGX\\_CONF\\_FILE\\_H\\_INCLUDED\\_](#)
- [ngx\\_conf\\_init\\_msec\\_value](#)
- [ngx\\_conf\\_init\\_ptr\\_value](#)
- [ngx\\_conf\\_init\\_size\\_value](#)
- [ngx\\_conf\\_init\\_uint\\_value](#)
- [ngx\\_conf\\_init\\_value](#)
- [ngx\\_conf\\_merge\\_bitmask\\_value](#)
- [ngx\\_conf\\_merge\\_bufs\\_value](#)

- [ngx\\_conf\\_merge\\_msec\\_value](#)
- [ngx\\_conf\\_merge\\_off\\_value](#)
- [ngx\\_conf\\_merge\\_ptr\\_value](#)
- [ngx\\_conf\\_merge\\_sec\\_value](#)
- [ngx\\_conf\\_merge\\_size\\_value](#)
- [ngx\\_conf\\_merge\\_str\\_value](#)
- [ngx\\_conf\\_merge\\_uint\\_value](#)
- [ngx\\_conf\\_merge\\_value](#)
- [ngx\\_get\\_conf](#)
- [ngx\\_null\\_command](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_CONF_FILE_H_INCLUDED_
9 #define _NGX_CONF_FILE_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 /*
17  *      AAAA  number of arguments
18  *      FF    command flags
19  *      TT    command type, i.e. HTTP "location" or "server" command
20  */
21
22 #define NGX_CONF_NOARGS      0x00000001
23 #define NGX_CONF_TAKE1      0x00000002
24 #define NGX_CONF_TAKE2      0x00000004
25 #define NGX_CONF_TAKE3      0x00000008
26 #define NGX_CONF_TAKE4      0x00000010
27 #define NGX_CONF_TAKE5      0x00000020
28 #define NGX_CONF_TAKE6      0x00000040
29 #define NGX_CONF_TAKE7      0x00000080
30
31 #define NGX_CONF_MAX_ARGS    8
32
33 #define NGX_CONF_TAKE12      (NGX_CONF_TAKE1 | NGX_CONF_TAKE2)
34 #define NGX_CONF_TAKE13      (NGX_CONF_TAKE1 | NGX_CONF_TAKE3)
35
36 #define NGX_CONF_TAKE23      (NGX_CONF_TAKE2 | NGX_CONF_TAKE3)
37
38 #define NGX_CONF_TAKE123      (NGX_CONF_TAKE1 | NGX_CONF_TAKE2 | NGX_CONF_TAKE3)
39 #define NGX_CONF_TAKE1234      (NGX_CONF_TAKE1 | NGX_CONF_TAKE2 | NGX_CONF_TAKE3 \
40 | NGX_CONF_TAKE4)
41
42 #define NGX_CONF_ARGS_NUMBER 0x000000ff
43 #define NGX_CONF_BLOCK      0x00000100
44 #define NGX_CONF_FLAG      0x00000200
45 #define NGX_CONF_ANY      0x00000400
46 #define NGX_CONF_1MORE      0x00000800
47 #define NGX_CONF_2MORE      0x00001000
48 #define NGX_CONF_MULTII      0x00000000 /* compatibility */
49
50 #define NGX_DIRECT_CONF      0x00010000

```

```

51
52 #define NGX_MAIN_CONF          0x01000000
53 #define NGX_ANY_CONF          0x0F000000
54
55
56
57 #define NGX_CONF_UNSET        -1
58 #define NGX_CONF_UNSET_UINT  (ngx_uint_t) -1
59 #define NGX_CONF_UNSET_PTR    (void *) -1
60 #define NGX_CONF_UNSET_SIZE   (size_t) -1
61 #define NGX_CONF_UNSET_MSEC   (ngx_msec_t) -1
62
63
64 #define NGX_CONF_OK           NULL
65 #define NGX_CONF_ERROR        (void *) -1
66
67 #define NGX_CONF_BLOCK_START  1
68 #define NGX_CONF_BLOCK_DONE   2
69 #define NGX_CONF_FILE_DONE    3
70
71 #define NGX_CORE_MODULE        0x45524F43 /* "CORE" */
72 #define NGX_CONF_MODULE        0x464E4F43 /* "CONF" */
73
74
75 #define NGX_MAX_CONF_ERRSTR   1024
76
77
78 struct ngx_command_s {
79     ngx_str_t      name;
80     ngx_uint_t     type;
81     char           *(*set)(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
82     ngx_uint_t     conf;
83     ngx_uint_t     offset;
84     void           *post;
85 };
86
87 #define ngx_null_command { ngx_null_string, 0, NULL, 0, 0, NULL }
88
89
90 struct ngx_open_file_s {
91     ngx_fd_t       fd;
92     ngx_str_t      name;
93
94     void           (*flush)(ngx_open_file_t *file, ngx_log_t *log);
95     void           *data;
96 };
97
98
99 #define NGX_MODULE_V1          0, 0, 0, 0, 0, 0, 1
100 #define NGX_MODULE_V1_PADDING 0, 0, 0, 0, 0, 0, 0, 0
101
102 struct ngx_module_s {
103     ngx_uint_t     ctx_index;
104     ngx_uint_t     index;
105
106     ngx_uint_t     spare0;
107     ngx_uint_t     spare1;
108     ngx_uint_t     spare2;
109     ngx_uint_t     spare3;
110
111     ngx_uint_t     version;
112
113     void           *ctx;
114     ngx_command_t  *commands;
115     ngx_uint_t     type;
116
117     ngx_int_t      (*init_master)(ngx_log_t *log);
118
119     ngx_int_t      (*init_module)(ngx_cycle_t *cycle);
120
121     ngx_int_t      (*init_process)(ngx_cycle_t *cycle);
122     ngx_int_t      (*init_thread)(ngx_cycle_t *cycle);
123     void           (*exit_thread)(ngx_cycle_t *cycle);
124     void           (*exit_process)(ngx_cycle_t *cycle);
125
126     void           (*exit_master)(ngx_cycle_t *cycle);

```

```

127
128     uintptr_t         spare_hook0;
129     uintptr_t         spare_hook1;
130     uintptr_t         spare_hook2;
131     uintptr_t         spare_hook3;
132     uintptr_t         spare_hook4;
133     uintptr_t         spare_hook5;
134     uintptr_t         spare_hook6;
135     uintptr_t         spare_hook7;
136 };
137
138
139 typedef struct {
140     ngx_str_t         name;
141     void              *(*create_conf)(ngx_cycle_t *cycle);
142     char              *(*init_conf)(ngx_cycle_t *cycle, void *conf);
143 } ngx_core_module_t;
144
145
146 typedef struct {
147     ngx_file_t        file;
148     ngx_buf_t         *buffer;
149     ngx_uint_t        line;
150 } ngx_conf_file_t;
151
152
153 typedef char *(*ngx_conf_handler_pt)(ngx_conf_t *cf,
154     ngx_command_t *dummy, void *conf);
155
156
157 struct ngx_conf_s {
158     char              *name;
159     ngx_array_t       *args;
160
161     ngx_cycle_t       *cycle;
162     ngx_pool_t        *pool;
163     ngx_pool_t        *temp_pool;
164     ngx_conf_file_t   *conf_file;
165     ngx_log_t         *log;
166
167     void              *ctx;
168     ngx_uint_t        module_type;
169     ngx_uint_t        cmd_type;
170
171     ngx_conf_handler_pt handler;
172     char              *handler_conf;
173 };
174
175
176 typedef char *(*ngx_conf_post_handler_pt) (ngx_conf_t *cf,
177     void *data, void *conf);
178
179 typedef struct {
180     ngx_conf_post_handler_pt post_handler;
181 } ngx_conf_post_t;
182
183
184 typedef struct {
185     ngx_conf_post_handler_pt post_handler;
186     char                    *old_name;
187     char                    *new_name;
188 } ngx_conf_deprecated_t;
189
190
191 typedef struct {
192     ngx_conf_post_handler_pt post_handler;
193     ngx_int_t                low;
194     ngx_int_t                high;
195 } ngx_conf_num_bounds_t;
196
197
198 typedef struct {
199     ngx_str_t                name;
200     ngx_uint_t               value;
201 } ngx_conf_enum_t;
202

```

```

203
204 #define NGX_CONF_BITMASK_SET 1
205
206 typedef struct {
207     ngx_str_t      name;
208     ngx_uint_t     mask;
209 } ngx_conf_bitmask_t;
210
211
212
213 char * ngx_conf_deprecated(ngx_conf_t *cf, void *post, void *data);
214 char * ngx_conf_check_num_bounds(ngx_conf_t *cf, void *post, void *data);
215
216
217 #define ngx_get_conf(conf_ctx, module)  conf_ctx[module.index]
218
219
220
221 #define ngx_conf_init_value(conf, default) \
222     if (conf == NGX_CONF_UNSET) { \
223         conf = default; \
224     }
225
226 #define ngx_conf_init_ptr_value(conf, default) \
227     if (conf == NGX_CONF_UNSET_PTR) { \
228         conf = default; \
229     }
230
231 #define ngx_conf_init_uint_value(conf, default) \
232     if (conf == NGX_CONF_UNSET_UINT) { \
233         conf = default; \
234     }
235
236 #define ngx_conf_init_size_value(conf, default) \
237     if (conf == NGX_CONF_UNSET_SIZE) { \
238         conf = default; \
239     }
240
241 #define ngx_conf_init_msec_value(conf, default) \
242     if (conf == NGX_CONF_UNSET_MSEC) { \
243         conf = default; \
244     }
245
246 #define ngx_conf_merge_value(conf, prev, default) \
247     if (conf == NGX_CONF_UNSET) { \
248         conf = (prev == NGX_CONF_UNSET) ? default : prev; \
249     }
250
251 #define ngx_conf_merge_ptr_value(conf, prev, default) \
252     if (conf == NGX_CONF_UNSET_PTR) { \
253         conf = (prev == NGX_CONF_UNSET_PTR) ? default : prev; \
254     }
255
256 #define ngx_conf_merge_uint_value(conf, prev, default) \
257     if (conf == NGX_CONF_UNSET_UINT) { \
258         conf = (prev == NGX_CONF_UNSET_UINT) ? default : prev; \
259     }
260
261 #define ngx_conf_merge_msec_value(conf, prev, default) \
262     if (conf == NGX_CONF_UNSET_MSEC) { \
263         conf = (prev == NGX_CONF_UNSET_MSEC) ? default : prev; \
264     }
265
266 #define ngx_conf_merge_sec_value(conf, prev, default) \
267     if (conf == NGX_CONF_UNSET) { \
268         conf = (prev == NGX_CONF_UNSET) ? default : prev; \
269     }
270
271 #define ngx_conf_merge_size_value(conf, prev, default) \
272     if (conf == NGX_CONF_UNSET_SIZE) { \
273         conf = (prev == NGX_CONF_UNSET_SIZE) ? default : prev; \
274     }
275
276 #define ngx_conf_merge_off_value(conf, prev, default) \
277     if (conf == NGX_CONF_UNSET) { \
278         conf = (prev == NGX_CONF_UNSET) ? default : prev; \

```

```

279     }
280
281 #define ngx_conf_merge_str_value(conf, prev, default)      \
282     if (conf.data == NULL) {                               \
283         if (prev.data) {                                   \
284             conf.len = prev.len;                          \
285             conf.data = prev.data;                        \
286         } else {                                          \
287             conf.len = sizeof(default) - 1;               \
288             conf.data = (u_char *) default;               \
289         }                                                 \
290     }
291
292 #define ngx_conf_merge_bufs_value(conf, prev, default_num, default_size) \
293     if (conf.num == 0) {                                   \
294         if (prev.num) {                                   \
295             conf.num = prev.num;                          \
296             conf.size = prev.size;                        \
297         } else {                                          \
298             conf.num = default_num;                       \
299             conf.size = default_size;                     \
300         }                                                 \
301     }
302
303 #define ngx_conf_merge_bitmask_value(conf, prev, default) \
304     if (conf == 0) {                                     \
305         conf = (prev == 0) ? default : prev;             \
306     }
307
308
309 char *ngx_conf_param(ngx_conf_t *cf);
310 char *ngx_conf_parse(ngx_conf_t *cf, ngx_str_t *filename);
311 char *ngx_conf_include(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
312
313
314 ngx_int_t ngx_conf_full_name(ngx_cycle_t *cycle, ngx_str_t *name,
315     ngx_uint_t conf_prefix);
316 ngx_open_file_t *ngx_conf_open_file(ngx_cycle_t *cycle, ngx_str_t *name);
317 void ngx_cdecl ngx_conf_log_error(ngx_uint_t level, ngx_conf_t *cf,
318     ngx_err_t err, const char *fmt, ...);
319
320
321 char *ngx_conf_set_flag_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
322 char *ngx_conf_set_str_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
323 char *ngx_conf_set_str_array_slot(ngx_conf_t *cf, ngx_command_t *cmd,
324     void *conf);
325 char *ngx_conf_set_keyval_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
326 char *ngx_conf_set_num_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
327 char *ngx_conf_set_size_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
328 char *ngx_conf_set_off_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
329 char *ngx_conf_set_msec_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
330 char *ngx_conf_set_sec_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
331 char *ngx_conf_set_bufs_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
332 char *ngx_conf_set_enum_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
333 char *ngx_conf_set_bitmask_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
334
335
336 extern ngx_uint_t     ngx_max_module;
337 extern ngx_module_t *ngx_modules[];
338
339
340 #endif /* NGX_CONF_FILE_H_INCLUDED */

```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_config.h - nginx-1.7.10

### Data types defined

- [ngx\\_flag\\_t](#)
- [ngx\\_int\\_t](#)
- [ngx\\_uint\\_t](#)

### Macros defined

- [INADDR\\_NONE](#)
- [NGX\\_ALIGNMENT](#)
- [NGX\\_CHANGEBIN\\_SIGNAL](#)
- [NGX\\_CHANGEBIN\\_SIGNAL](#)
- [NGX\\_HAVE\\_SO\\_SNDLOWAT](#)
- [NGX\\_INT32\\_LEN](#)
- [NGX\\_INT64\\_LEN](#)
- [NGX\\_INT\\_T\\_LEN](#)
- [NGX\\_INT\\_T\\_LEN](#)
- [NGX\\_INVALID\\_ARRAY\\_INDEX](#)
- [NGX\\_MAXHOSTNAMELEN](#)
- [NGX\\_MAXHOSTNAMELEN](#)
- [NGX\\_MAX\\_INT32\\_VALUE](#)
- [NGX\\_MAX\\_UINT32\\_VALUE](#)
- [NGX\\_MAX\\_UINT32\\_VALUE](#)
- [NGX\\_NOACCEPT\\_SIGNAL](#)
- [NGX\\_RECONFIGURE\\_SIGNAL](#)
- [NGX\\_REOPEN\\_SIGNAL](#)
- [NGX\\_REOPEN\\_SIGNAL](#)
- [NGX\\_SHUTDOWN\\_SIGNAL](#)
- [NGX\\_TERMINATE\\_SIGNAL](#)
- [\\_NGX\\_CONFIG\\_H\\_INCLUDED\\_](#)
- [\\_\\_FreeBSD\\_\\_](#)
- [\\_\\_FreeBSD\\_version](#)
- [ngx\\_abort](#)

- [ngx\\_align](#)
- [ngx\\_align\\_ptr](#)
- [ngx\\_cdecl](#)
- [ngx\\_inline](#)
- [ngx\\_libc\\_cdecl](#)
- [ngx\\_random](#)
- [ngx\\_signal\\_helper](#)
- [ngx\\_signal\\_value](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef NGX_CONFIG_H_INCLUDED
9 #define NGX_CONFIG_H_INCLUDED
10
11
12 #include <ngx_auto_headers.h>
13
14
15 #if defined __DragonFly__ && !defined FreeBSD
16 #define __FreeBSD__ 4
17 #define __FreeBSD_version 480101
18 #endif
19
20
21 #if (NGX_FREEBSD)
22 #include <ngx_freebsd_config.h>
23
24
25 #elif (NGX_LINUX)
26 #include <ngx_linux_config.h>
27
28
29 #elif (NGX_SOLARIS)
30 #include <ngx_solaris_config.h>
31
32
33 #elif (NGX_DARWIN)
34 #include <ngx_darwin_config.h>
35
36
37 #elif (NGX_WIN32)
38 #include <ngx_win32_config.h>
39
40
41 #else /* POSIX */
42 #include <ngx_posix_config.h>
43
44 #endif
45
46
47 #ifndef NGX_HAVE_SO_SNDLOWAT
48 #define NGX_HAVE_SO_SNDLOWAT 1
49 #endif
50
51
52 #if !(NGX_WIN32)
53
54 #define ngx_signal_helper(n) SIG##n

```



```

55 #define ngx_signal_value(n)      ngx_signal_helper(n)
56
57 #define ngx_random                random
58
59 /* TODO: #ifndef */
60 #define NGX_SHUTDOWN_SIGNAL      QUIT
61 #define NGX_TERMINATE_SIGNAL     TERM
62 #define NGX_NOACCEPT_SIGNAL      WINCH
63 #define NGX_RECONFIGURE_SIGNAL   HUP
64
65 #if (NGX_LINUXTHREADS)
66 #define NGX_REOPEN_SIGNAL        INFO
67 #define NGX_CHANGEBIN_SIGNAL     XCPU
68 #else
69 #define NGX_REOPEN_SIGNAL        USR1
70 #define NGX_CHANGEBIN_SIGNAL     USR2
71 #endif
72
73 #define ngx_cdecl
74 #define ngx_libc_cdecl
75
76 #endif
77
78 typedef intptr_t      ngx_int_t;
79 typedef uintptr_t     ngx_uint_t;
80 typedef intptr_t      ngx_flag_t;
81
82
83 #define NGX_INT32_LEN      (sizeof("-2147483648") - 1)
84 #define NGX_INT64_LEN      (sizeof("-9223372036854775808") - 1)
85
86 #if (NGX_PTR_SIZE == 4)
87 #define NGX_INT_T_LEN      NGX_INT32_LEN
88 #else
89 #define NGX_INT_T_LEN      NGX_INT64_LEN
90 #endif
91
92
93 #ifndef NGX_ALIGNMENT
94 #define NGX_ALIGNMENT      sizeof(unsigned long)    /* platform word */
95 #endif
96
97 #define ngx_align(d, a)      (((d) + (a - 1)) & ~(a - 1))
98 #define ngx_align_ptr(p, a) \
99     (u_char *) (((uintptr_t) (p) + ((uintptr_t) a - 1)) & ~((uintptr_t) a - 1))
100
101
102 #define ngx_abort            abort
103
104
105 /* TODO: platform specific: array[NGX_INVALID_ARRAY_INDEX] must cause SIGSEGV */
106 #define NGX_INVALID_ARRAY_INDEX 0x80000000
107
108
109 /* TODO: auto_conf: ngx_inline inline __inline __inline__ */
110 #ifndef ngx_inline
111 #define ngx_inline          inline
112 #endif
113
114 #ifndef INADDR_NONE /* Solaris */
115 #define INADDR_NONE        ((unsigned int) -1)
116 #endif
117
118 #ifdef MAXHOSTNAMELEN
119 #define NGX_MAXHOSTNAMELEN MAXHOSTNAMELEN
120 #else
121 #define NGX_MAXHOSTNAMELEN 256
122 #endif
123
124
125 #if ((__GNU__ == 2) && (__GNUC_MINOR__ < 8))
126 #define NGX_MAX_UINT32_VALUE (uint32_t) 0xffffffffLL
127 #else
128 #define NGX_MAX_UINT32_VALUE (uint32_t) 0xffffffff
129 #endif
130

```

```
131 #define NGX_MAX_INT32_VALUE (uint32_t) 0x7fffffff
132
133
134 #endif /* NGX_CONFIG_H_INCLUDED */
```

[One Level Up](#)

[Top Level](#)

# src/os/unix/nginx\_linux\_config.h - nginx-1.7.10

## Data types defined

- [ngx\\_aiocb\\_t](#)

## Macros defined

- [NGX\\_HAVE\\_INHERITED\\_NONBLOCK](#)
- [NGX\\_HAVE\\_OS\\_SPECIFIC\\_INIT](#)
- [NGX\\_HAVE\\_SO\\_SNDLOWAT](#)
- [NGX\\_LISTEN\\_BACKLOG](#)
- [NGX\\_SENDFILE\\_LIMIT](#)
- [\\_FILE\\_OFFSET\\_BITS](#)
- [\\_GNU\\_SOURCE](#)
- [\\_NGX\\_LINUX\\_CONFIG\\_H\\_INCLUDED](#)
- [ngx\\_debug\\_init](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_LINUX_CONFIG_H_INCLUDED_
9 #define _NGX_LINUX_CONFIG_H_INCLUDED_
10
11
12 #ifndef _GNU_SOURCE
13 #define _GNU_SOURCE          /* pread(), pwrite(), gethostname() */
14 #endif
15
16 #define _FILE_OFFSET_BITS 64
17
18 #include <sys/types.h>
19 #include <sys/time.h>
20 #include <unistd.h>
21 #include <stdarg.h>
22 #include <stddef.h>          /* offsetof() */
23 #include <stdio.h>
24 #include <stdlib.h>
25 #include <ctype.h>
26 #include <errno.h>
27 #include <string.h>
28 #include <signal.h>
29 #include <pwd.h>
30 #include <grp.h>
31 #include <dirent.h>
32 #include <glob.h>
33 #include <sys/vfs.h>        /* statfs() */
34
35 #include <sys/uio.h>
36 #include <sys/stat.h>
37 #include <fcntl.h>
38
```

```

39 #include <sys/wait.h>
40 #include <sys/mman.h>
41 #include <sys/resource.h>
42 #include <sched.h>
43
44 #include <sys/socket.h>
45 #include <netinet/in.h>
46 #include <netinet/tcp.h>      /* TCP_NODELAY, TCP_CORK */
47 #include <arpa/inet.h>
48 #include <netdb.h>
49 #include <sys/un.h>
50
51 #include <time.h>              /* tzset() */
52 #include <malloc.h>           /* memalign() */
53 #include <limits.h>          /* IOV_MAX */
54 #include <sys/ioctl.h>
55 #include <crypt.h>
56 #include <sys/utsname.h>      /* uname() */
57
58
59 #include <ngx_auto_config.h>
60
61
62 #if (NGX_HAVE_POSIX_SEM)
63 #include <semaphore.h>
64 #endif
65
66
67 #if (NGX_HAVE_SYS_PRCTL_H)
68 #include <sys/prctl.h>
69 #endif
70
71
72 #if (NGX_HAVE_SENDFILE64)
73 #include <sys/sendfile.h>
74 #else
75 extern ssize_t sendfile(int s, int fd, int32_t *offset, size_t size);
76 #define NGX_SENDFILE_LIMIT 0x80000000
77 #endif
78
79
80 #if (NGX_HAVE_POLL)
81 #include <poll.h>
82 #endif
83
84
85 #if (NGX_HAVE_RT_SIG)
86 #include <poll.h>
87 #include <sys/sysctl.h>
88 #endif
89
90
91 #if (NGX_HAVE_EPOLL)
92 #include <sys/epoll.h>
93 #endif
94
95
96 #if (NGX_HAVE_FILE_AIO)
97 #if (NGX_HAVE_SYS_EVENTFD_H)
98 #include <sys/eventfd.h>
99 #endif
100 #include <sys/syscall.h>
101 #include <linux/aio_abi.h>
102 typedef struct iocb ngx_aiocb_t;
103 #endif
104
105
106 #define NGX_LISTEN_BACKLOG      511
107
108
109 #ifndef NGX_HAVE_SO_SNDLOWAT
110 /* setsockopt(SO_SNDLOWAT) returns ENOPROTOPT */
111 #define NGX_HAVE_SO_SNDLOWAT    0
112 #endif
113
114

```

```
115 #ifndef NGX\_HAVE\_INHERITED\_NONBLOCK
116 #define NGX\_HAVE\_INHERITED\_NONBLOCK 0
117 #endif
118
119
120 #define NGX\_HAVE\_OS\_SPECIFIC\_INIT 1
121 #define ngx\_debug\_init\(\)
122
123
124 extern char **environ;
125
126
127 #endif /* \_NGX\_LINUX\_CONFIG\_H\_INCLUDED */
```

[One Level Up](#)

[Top Level](#)

## src/os/unix/ - nginx-1.7.10

- [ngx\\_aio\\_read.c](#)
- [ngx\\_aio\\_read\\_chain.c](#)
- [ngx\\_aio\\_write.c](#)
- [ngx\\_aio\\_write\\_chain.c](#)
- [ngx\\_alloc.c](#)
- [ngx\\_alloc.h](#)
- [ngx\\_atomic.h](#)
- [ngx\\_channel.c](#)
- [ngx\\_channel.h](#)
- [ngx\\_daemon.c](#)
- [ngx\\_darwin.h](#)
- [ngx\\_darwin\\_config.h](#)
- [ngx\\_darwin\\_init.c](#)
- [ngx\\_darwin\\_sendfile\\_chain.c](#)
- [ngx\\_errno.c](#)
- [ngx\\_errno.h](#)
- [ngx\\_file\\_aio\\_read.c](#)
- [ngx\\_files.c](#)
- [ngx\\_files.h](#)
- [ngx\\_freebsd.h](#)
- [ngx\\_freebsd\\_config.h](#)
- [ngx\\_freebsd\\_init.c](#)
- [ngx\\_freebsd\\_rfork\\_thread.c](#)
- [ngx\\_freebsd\\_rfork\\_thread.h](#)
- [ngx\\_freebsd\\_sendfile\\_chain.c](#)
- [ngx\\_gcc\\_atomic\\_amd64.h](#)
- [ngx\\_gcc\\_atomic\\_ppc.h](#)
- [ngx\\_gcc\\_atomic\\_sparc64.h](#)
- [ngx\\_gcc\\_atomic\\_x86.h](#)
- [ngx\\_linux.h](#)
- [ngx\\_linux\\_aio\\_read.c](#)

- [ngx\\_linux\\_config.h](#)
- [ngx\\_linux\\_init.c](#)
- [ngx\\_linux\\_sendfile\\_chain.c](#)
- [ngx\\_os.h](#)
- [ngx\\_posix\\_config.h](#)
- [ngx\\_posix\\_init.c](#)
- [ngx\\_process.c](#)
- [ngx\\_process.h](#)
- [ngx\\_process\\_cycle.c](#)
- [ngx\\_process\\_cycle.h](#)
- [ngx\\_pthread\\_thread.c](#)
- [ngx\\_readv\\_chain.c](#)
- [ngx\\_recv.c](#)
- [ngx\\_send.c](#)
- [ngx\\_setaffinity.c](#)
- [ngx\\_setaffinity.h](#)
- [ngx\\_setproctitle.c](#)
- [ngx\\_setproctitle.h](#)
- [ngx\\_shmem.c](#)
- [ngx\\_shmem.h](#)
- [ngx\\_socket.c](#)
- [ngx\\_socket.h](#)
- [ngx\\_solaris.h](#)
- [ngx\\_solaris\\_config.h](#)
- [ngx\\_solaris\\_init.c](#)
- [ngx\\_solaris\\_sendfilev\\_chain.c](#)
- [ngx\\_sunpro\\_atomic\\_sparc64.h](#)
- [ngx\\_thread.h](#)
- [ngx\\_time.c](#)
- [ngx\\_time.h](#)
- [ngx\\_udp\\_recv.c](#)
- [ngx\\_user.c](#)
- [ngx\\_user.h](#)

- [ngx\\_writev\\_chain.c](#)
- [rfork\\_thread.S](#)

[One Level Up](#)

[Top Level](#)



[One Level Up](#)

[Top Level](#)

## src/os/ - nginx-1.7.10

- [unix/](#)

[One Level Up](#)

[Top Level](#)

# src/os/unix/nginx\_aio\_read.c - nginx-1.7.10

## Functions defined

- [ngx\\_aio\\_read](#)

## Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 extern int  ngx_kqueue;
14
15
16 ssize_t
17 ngx_aio_read(ngx_connection_t *c, u_char *buf, size_t size)
18 {
19     int          n;
20     ngx_event_t *rev;
21
22     rev = c->read;
23
24     if (!rev->ready) {
25         ngx_log_error(NGX_LOG_ALERT, c->log, 0, "second aio post");
26         return NGX_AGAIN;
27     }
28
29     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, c->log, 0,
30                  "rev->complete: %d", rev->complete);
31     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, c->log, 0,
32                  "aio size: %d", size);
33
34     if (!rev->complete) {
35         ngx_memzero(&rev->aiocb, sizeof(struct aiocb));
36
37         rev->aiocb.aio_fildes = c->fd;
38         rev->aiocb.aio_buf = buf;
39         rev->aiocb.aio_nbytes = size;
40
41         #if (NGX_HAVE_KQUEUE)
42             rev->aiocb.aio_sigevent.sigev_notify_kqueue = ngx_kqueue;
43             rev->aiocb.aio_sigevent.sigev_notify = SIGEV_KEVENT;
44             rev->aiocb.aio_sigevent.sigev_value.sigval_ptr = rev;
45         #endif
46
47         if (aio_read(&rev->aiocb) == -1) {
48             ngx_log_error(NGX_LOG_CRIT, rev->log, ngx_errno,
49                          "aio_read() failed");
50             rev->error = 1;
51             return NGX_ERROR;
52         }
53
54         ngx_log_debug1(NGX_LOG_DEBUG_EVENT, c->log, 0,
55                      "aio_read: #%d OK", c->fd);
56
57         rev->active = 1;
58         rev->ready = 0;
59     }
60
61     rev->complete = 0;
62
```

```

63 n = aio_error(&rev->aiocb);
64 if (n == -1) {
65     ngx_log_error(NGX_LOG_ALERT, c->log, ngx_errno, "aio_error() failed");
66     rev->error = 1;
67     return NGX_ERROR;
68 }
69
70 if (n != 0) {
71     if (n == NGX_EINPROGRESS) {
72         if (rev->ready) {
73             ngx_log_error(NGX_LOG_ALERT, c->log, n,
74                 "aio_read() still in progress");
75             rev->ready = 0;
76         }
77         return NGX_AGAIN;
78     }
79
80     ngx_log_error(NGX_LOG_CRIT, c->log, n, "aio_read() failed");
81     rev->error = 1;
82     rev->ready = 0;
83     return NGX_ERROR;
84 }
85
86 n = aio_return(&rev->aiocb);
87 if (n == -1) {
88     ngx_log_error(NGX_LOG_ALERT, c->log, ngx_errno,
89         "aio_return() failed");
90
91     rev->error = 1;
92     rev->ready = 0;
93     return NGX_ERROR;
94 }
95
96 ngx_log_debug2(NGX_LOG_DEBUG_EVENT, rev->log, 0,
97     "aio_read: #%d %d", c->fd, n);
98
99 if (n == 0) {
100     rev->eof = 1;
101     rev->ready = 0;
102 } else {
103     rev->ready = 1;
104 }
105
106 rev->active = 0;
107
108 return n;
109 }

```

[One Level Up](#)

[Top Level](#)

## src/event/modules/nginx\_kqueue\_module.c - nginx-1.7.10

### Global variables defined

- [change\\_list](#)
- [change\\_list0](#)
- [change\\_list1](#)
- [event\\_list](#)
- [kevent\\_mutex](#)
- [kqueue\\_name](#)
- [list\\_mutex](#)
- [max\\_changes](#)
- [nchanges](#)
- [nevents](#)
- [ngx\\_kqueue](#)
- [ngx\\_kqueue\\_commands](#)
- [ngx\\_kqueue\\_module](#)
- [ngx\\_kqueue\\_module\\_ctx](#)

### Data types defined

- [ngx\\_kqueue\\_conf\\_t](#)

### Functions defined

- [ngx\\_kqueue\\_add\\_event](#)
- [ngx\\_kqueue\\_create\\_conf](#)
- [ngx\\_kqueue\\_del\\_event](#)
- [ngx\\_kqueue\\_done](#)
- [ngx\\_kqueue\\_dump\\_event](#)
- [ngx\\_kqueue\\_init](#)
- [ngx\\_kqueue\\_init\\_conf](#)
- [ngx\\_kqueue\\_process\\_changes](#)
- [ngx\\_kqueue\\_process\\_events](#)
- [ngx\\_kqueue\\_set\\_event](#)

### Source code

```

2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 typedef struct {
14     ngx_uint_t  changes;
15     ngx_uint_t  events;
16 } ngx_kqueue_conf_t;
17
18
19 static ngx_int_t ngx_kqueue_init(ngx_cycle_t *cycle, ngx_msec_t timer);
20 static void ngx_kqueue_done(ngx_cycle_t *cycle);
21 static ngx_int_t ngx_kqueue_add_event(ngx_event_t *ev, ngx_int_t event,
22     ngx_uint_t flags);
23 static ngx_int_t ngx_kqueue_del_event(ngx_event_t *ev, ngx_int_t event,
24     ngx_uint_t flags);
25 static ngx_int_t ngx_kqueue_set_event(ngx_event_t *ev, ngx_int_t filter,
26     ngx_uint_t flags);
27 static ngx_int_t ngx_kqueue_process_changes(ngx_cycle_t *cycle, ngx_uint_t try);
28 static ngx_int_t ngx_kqueue_process_events(ngx_cycle_t *cycle, ngx_msec_t timer,
29     ngx_uint_t flags);
30 static ngx_inline void ngx_kqueue_dump_event(ngx_log_t *log,
31     struct kevent *kev);
32
33 static void *ngx_kqueue_create_conf(ngx_cycle_t *cycle);
34 static char *ngx_kqueue_init_conf(ngx_cycle_t *cycle, void *conf);
35
36
37 int                ngx_kqueue = -1;
38
39 /*
40 * The "change_list" should be declared as ngx_thread_volatile.
41 * However, the use of the change_list is localized in kqueue functions and
42 * is protected by the mutex so even the "icc -ipo" should not build the code
43 * with the race condition. Thus we avoid the declaration to make a more
44 * readable code.
45 */
46
47 static struct kevent *change_list, *change_list0, *change_list1;
48 static struct kevent *event_list;
49 static ngx_uint_t    max_changes, nchanges, nevents;
50
51 #if (NGX_THREADS)
52 static ngx_mutex_t  *list_mutex;
53 static ngx_mutex_t  *kevent_mutex;
54 #endif
55
56
57
58 static ngx_str_t    kqueue_name = ngx_string("kqueue");
59
60 static ngx_command_t  ngx_kqueue_commands[] = {
61
62     { ngx_string("kqueue_changes"),
63       NGX_EVENT_CONF|NGX_CONF_TAKE1,
64       ngx_conf_set_num_slot,
65       0,
66       offsetof(ngx_kqueue_conf_t, changes),
67       NULL },
68
69     { ngx_string("kqueue_events"),
70       NGX_EVENT_CONF|NGX_CONF_TAKE1,
71       ngx_conf_set_num_slot,
72       0,
73       offsetof(ngx_kqueue_conf_t, events),
74       NULL },
75
76     ngx_null_command
77 };

```

```

78
79
80 ngx_event_module_t ngx_kqueue_module_ctx = {
81     &kqueue_name,
82     ngx_kqueue_create_conf,          /* create configuration */
83     ngx_kqueue_init_conf,          /* init configuration */
84
85     {
86         ngx_kqueue_add_event,      /* add an event */
87         ngx_kqueue_del_event,      /* delete an event */
88         ngx_kqueue_add_event,      /* enable an event */
89         ngx_kqueue_del_event,      /* disable an event */
90         NULL,                      /* add an connection */
91         NULL,                      /* delete an connection */
92         ngx_kqueue_process_changes, /* process the changes */
93         ngx_kqueue_process_events, /* process the events */
94         ngx_kqueue_init,           /* init the events */
95         ngx_kqueue_done,           /* done the events */
96     }
97 };
98
99
100 ngx_module_t ngx_kqueue_module = {
101     NGX_MODULE_V1,
102     &ngx_kqueue_module_ctx,        /* module context */
103     ngx_kqueue_commands,          /* module directives */
104     NGX_EVENT_MODULE,             /* module type */
105     NULL,                         /* init master */
106     NULL,                         /* init module */
107     NULL,                         /* init process */
108     NULL,                         /* init thread */
109     NULL,                         /* exit thread */
110     NULL,                         /* exit process */
111     NULL,                         /* exit master */
112     NGX_MODULE_V1_PADDING
113 };
114
115
116 static ngx_int_t
117 ngx_kqueue_init(ngx_cycle_t *cycle, ngx_msec_t timer)
118 {
119     ngx_kqueue_conf_t *kcf;
120     struct timespec  ts;
121     #if (NGX_HAVE_TIMER_EVENT)
122     struct kevent    kev;
123     #endif
124
125     kcf = ngx_event_get_conf(cycle->conf_ctx, ngx_kqueue_module);
126
127     if (ngx_kqueue == -1) {
128         ngx_kqueue = kqueue();
129
130         if (ngx_kqueue == -1) {
131             ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
132                 "kqueue() failed");
133             return NGX_ERROR;
134         }
135
136     #if (NGX_THREADS)
137
138         list_mutex = ngx_mutex_init(cycle->log, 0);
139         if (list_mutex == NULL) {
140             return NGX_ERROR;
141         }
142
143         kevent_mutex = ngx_mutex_init(cycle->log, 0);
144         if (kevent_mutex == NULL) {
145             return NGX_ERROR;
146         }
147
148     #endif
149     }
150
151     if (max_changes < kcf->changes) {
152         if (nchanges) {
153             ts.tv_sec = 0;

```

```

154     ts.tv_nsec = 0;
155
156     if (kevent(ngx_kqueue, change_list, (int) nchanges, NULL, 0, &ts)
157         == -1)
158     {
159         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
160             "kevent() failed");
161         return NGX_ERROR;
162     }
163     nchanges = 0;
164 }
165
166 if (change_list0) {
167     ngx_free(change_list0);
168 }
169
170 change_list0 = ngx_alloc(kcf->changes * sizeof(struct kevent),
171     cycle->log);
172 if (change_list0 == NULL) {
173     return NGX_ERROR;
174 }
175
176 if (change_list1) {
177     ngx_free(change_list1);
178 }
179
180 change_list1 = ngx_alloc(kcf->changes * sizeof(struct kevent),
181     cycle->log);
182 if (change_list1 == NULL) {
183     return NGX_ERROR;
184 }
185
186 change_list = change_list0;
187 }
188
189 max_changes = kcf->changes;
190
191 if (nevents < kcf->events) {
192     if (event_list) {
193         ngx_free(event_list);
194     }
195
196     event_list = ngx_alloc(kcf->events * sizeof(struct kevent), cycle->log);
197     if (event_list == NULL) {
198         return NGX_ERROR;
199     }
200 }
201
202 ngx_event_flags = NGX_USE_ONESHOT_EVENT
203     | NGX_USE_KQUEUE_EVENT
204     | NGX_USE_VNODE_EVENT;
205
206 #if (NGX_HAVE_TIMER_EVENT)
207
208 if (timer) {
209     kev.ident = 0;
210     kev.filter = EVFILT_TIMER;
211     kev.flags = EV_ADD|EV_ENABLE;
212     kev.fflags = 0;
213     kev.data = timer;
214     kev.udata = 0;
215
216     ts.tv_sec = 0;
217     ts.tv_nsec = 0;
218
219     if (kevent(ngx_kqueue, &kev, 1, NULL, 0, &ts) == -1) {
220         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
221             "kevent(EVFILT_TIMER) failed");
222         return NGX_ERROR;
223     }
224
225     ngx_event_flags |= NGX_USE_TIMER_EVENT;
226 }
227
228 #endif
229

```

```

230 #if (NGX_HAVE_CLEAR_EVENT)
231     ngx_event_flags |= NGX_USE_CLEAR_EVENT;
232 #else
233     ngx_event_flags |= NGX_USE_LEVEL_EVENT;
234 #endif
235
236 #if (NGX_HAVE_LOWAT_EVENT)
237     ngx_event_flags |= NGX_USE_LOWAT_EVENT;
238 #endif
239
240     nevents = kcf->events;
241
242     ngx_io = ngx_os_io;
243
244     ngx_event_actions = ngx_kqueue_module_ctx.actions;
245
246     return NGX_OK;
247 }
248
249 static void
250 ngx_kqueue_done(ngx_cycle_t *cycle)
251 {
252     if (close(ngx_kqueue) == -1) {
253         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
254             "kqueue close() failed");
255     }
256
257     ngx_kqueue = -1;
258
259 #if (NGX_THREADS)
260     ngx_mutex_destroy(kevent_mutex);
261     ngx_mutex_destroy(list_mutex);
262 #endif
263
264     ngx_free(change_list1);
265     ngx_free(change_list0);
266     ngx_free(event_list);
267
268     change_list1 = NULL;
269     change_list0 = NULL;
270     change_list = NULL;
271     event_list = NULL;
272     max_changes = 0;
273     nchanges = 0;
274     nevents = 0;
275 }
276
277
278 static ngx_int_t
279 ngx_kqueue_add_event(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags)
280 {
281     ngx_int_t rc;
282
283     #if 0
284     ngx_event_t *e;
285     ngx_connection_t *c;
286     #endif
287
288     ev->active = 1;
289     ev->disabled = 0;
290     ev->oneshot = (flags & NGX_ONESHOT_EVENT) ? 1 : 0;
291
292     ngx_mutex_lock(list_mutex);
293
294     #if 0
295     if (ev->index < nchanges
296         && ((uintptr_t) change_list[ev->index].udata & (uintptr_t) ~1)
297         == (uintptr_t) ev)
298     {
299         if (change_list[ev->index].flags == EV_DISABLE) {
300             /*
301              * if the EV_DISABLE is still not passed to a kernel
302              * we will not pass it
303              */
304         }
305     }

```



```

306         ngx_log_debug2(NGX_LOG_DEBUG_EVENT, ev->log, 0,
307             "kevent activated: %d: ft:%i",
308             ngx_event_ident(ev->data), event);
309
310
311         if (ev->index < --nchanges) {
312             e = (ngx_event_t *)
313                 ((uintptr_t) change_list[nchanges].udata & (uintptr_t) ~1);
314             change_list[ev->index] = change_list[nchanges];
315             e->index = ev->index;
316         }
317
318         ngx_mutex_unlock(list_mutex);
319
320         return NGX_OK;
321     }
322
323     c = ev->data;
324
325     ngx_log_error(NGX_LOG_ALERT, ev->log, 0,
326         "previous event on #%d were not passed in kernel", c->fd);
327
328     ngx_mutex_unlock(list_mutex);
329
330     return NGX_ERROR;
331 }
332
333 #endif
334
335 rc = ngx_kqueue_set_event(ev, event, EV_ADD|EV_ENABLE|flags);
336
337 ngx_mutex_unlock(list_mutex);
338
339 return rc;
340 }
341
342
343 static ngx_int_t
344 ngx_kqueue_del_event(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags)
345 {
346     ngx_int_t    rc;
347     ngx_event_t *e;
348
349     ev->active = 0;
350     ev->disabled = 0;
351
352     ngx_mutex_lock(list_mutex);
353
354     if (ev->index < nchanges
355         && ((uintptr_t) change_list[ev->index].udata & (uintptr_t) ~1)
356         == (uintptr_t) ev)
357     {
358         ngx_log_debug2(NGX_LOG_DEBUG_EVENT, ev->log, 0,
359             "kevent deleted: %d: ft:%i",
360             ngx_event_ident(ev->data), event);
361
362         /* if the event is still not passed to a kernel we will not pass it */
363
364         nchanges--;
365
366         if (ev->index < nchanges) {
367             e = (ngx_event_t *)
368                 ((uintptr_t) change_list[nchanges].udata & (uintptr_t) ~1);
369             change_list[ev->index] = change_list[nchanges];
370             e->index = ev->index;
371         }
372
373         ngx_mutex_unlock(list_mutex);
374
375         return NGX_OK;
376     }
377
378     /*
379     * when the file descriptor is closed the kqueue automatically deletes
380     * its filters so we do not need to delete explicitly the event
381     * before the closing the file descriptor.

```

```

382     */
383
384     if (flags & NGX_CLOSE_EVENT) {
385         ngx_mutex_unlock(list_mutex);
386         return NGX_OK;
387     }
388
389     if (flags & NGX_DISABLE_EVENT) {
390         ev->disabled = 1;
391     }
392     else {
393         flags |= EV_DELETE;
394     }
395
396     rc = ngx_kqueue_set_event(ev, event, flags);
397
398     ngx_mutex_unlock(list_mutex);
399
400     return rc;
401 }
402
403
404 static ngx_int_t
405 ngx_kqueue_set_event(ngx_event_t *ev, ngx_int_t filter, ngx_uint_t flags)
406 {
407     struct kevent    *kev;
408     struct timespec  ts;
409     ngx_connection_t *c;
410
411     c = ev->data;
412
413     ngx_log_debug3(NGX_LOG_DEBUG_EVENT, ev->log, 0,
414                  "kevent set event: %d: ft:%i fl:%04Xi",
415                  c->fd, filter, flags);
416
417     if (nchanges >= max_changes) {
418         ngx_log_error(NGX_LOG_WARN, ev->log, 0,
419                     "kqueue change list is filled up");
420
421         ts.tv_sec = 0;
422         ts.tv_nsec = 0;
423
424         if (kevent(ngx_kqueue, change_list, (int) nchanges, NULL, 0, &ts)
425             == -1)
426         {
427             ngx_log_error(NGX_LOG_ALERT, ev->log, ngx_errno, "kevent() failed");
428             return NGX_ERROR;
429         }
430
431         nchanges = 0;
432     }
433
434     kev = &change_list[nchanges];
435
436     kev->ident = c->fd;
437     kev->filter = (short) filter;
438     kev->flags = (u_short) flags;
439     kev->udata = NGX_KQUEUE_UDATA_T ((uintptr_t) ev | ev->instance);
440
441     if (filter == EVFILT_VNODE) {
442         kev->fflags = NOTE_DELETE|NOTE_WRITE|NOTE_EXTEND
443                   |NOTE_ATTRIB|NOTE_RENAME
444 #if (FreeBSD == 4 && FreeBSD_version >= 430000) \
445     || FreeBSD_version >= 500018
446                   |NOTE_REVOKE
447 #endif
448     ;
449     kev->data = 0;
450
451     } else {
452 #if (NGX_HAVE_LOWAT_EVENT)
453         if (flags & NGX_LOWAT_EVENT) {
454             kev->fflags = NOTE_LOWAT;
455             kev->data = ev->available;
456
457         } else {

```

```

458         kev->fflags = 0;
459         kev->data = 0;
460     }
461 #else
462     kev->fflags = 0;
463     kev->data = 0;
464 #endif
465 }
466
467     ev->index = nchanges;
468     nchanges++;
469
470     if (flags & NGX_FLUSH_EVENT) {
471         ts.tv_sec = 0;
472         ts.tv_nsec = 0;
473
474         ngx_log_debug0(NGX_LOG_DEBUG_EVENT, ev->log, 0, "kevent flush");
475
476         if (kevent(ngx_kqueue, change_list, (int) nchanges, NULL, 0, &ts)
477             == -1)
478         {
479             ngx_log_error(NGX_LOG_ALERT, ev->log, ngx_errno, "kevent() failed");
480             return NGX_ERROR;
481         }
482
483         nchanges = 0;
484     }
485
486     return NGX_OK;
487 }
488
489
490 static ngx_int_t
491 ngx_kqueue_process_events(ngx_cycle_t *cycle, ngx_msec_t timer,
492     ngx_uint_t flags)
493 {
494     int             events, n;
495     ngx_int_t     i, instance;
496     ngx_uint_t    level;
497     ngx_err_t     err;
498     ngx_event_t  *ev;
499     ngx_queue_t  *queue;
500     struct timespec ts, *tp;
501
502     if (ngx_threaded) {
503         if (ngx_kqueue_process_changes(cycle, 0) == NGX_ERROR) {
504             return NGX_ERROR;
505         }
506
507         n = 0;
508     } else {
509         n = (int) nchanges;
510         nchanges = 0;
511     }
512
513     if (timer == NGX_TIMER_INFINITE) {
514         tp = NULL;
515     } else {
516
517         ts.tv_sec = timer / 1000;
518         ts.tv_nsec = (timer % 1000) * 1000000;
519
520         /*
521          * 64-bit Darwin kernel has the bug: kernel level ts.tv_nsec is
522          * the int32_t while user level ts.tv_nsec is the long (64-bit),
523          * so on the big endian PowerPC all nanoseconds are lost.
524          */
525
526         #if (NGX_DARWIN_KEVENT_BUG)
527             ts.tv_nsec <<= 32;
528         #endif
529
530         tp = &ts;
531     }
532 }

```

```

534 ngx_log_debug2(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
535     "kevent timer: %M, changes: %d", timer, n);
536
537
538 events = kevent(ngx_kqueue, change_list, n, event_list, (int) nevents, tp);
539
540 err = (events == -1) ? ngx_errno : 0;
541
542 if (flags & NGX_UPDATE_TIME || ngx_event_timer_alarm) {
543     ngx_time_update();
544 }
545
546 ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
547     "kevent events: %d", events);
548
549 if (err) {
550     if (err == NGX_EINTR) {
551
552         if (ngx_event_timer_alarm) {
553             ngx_event_timer_alarm = 0;
554             return NGX_OK;
555         }
556
557         level = NGX_LOG_INFO;
558
559     } else {
560         level = NGX_LOG_ALERT;
561     }
562
563     ngx_log_error(level, cycle->log, err, "kevent() failed");
564     return NGX_ERROR;
565 }
566
567 if (events == 0) {
568     if (timer != NGX_TIMER_INFINITE) {
569         return NGX_OK;
570     }
571
572     ngx_log_error(NGX_LOG_ALERT, cycle->log, 0,
573         "kevent() returned no events without timeout");
574     return NGX_ERROR;
575 }
576
577 for (i = 0; i < events; i++) {
578
579     ngx_kqueue_dump_event(cycle->log, &event_list[i]);
580
581     if (event_list[i].flags & EV_ERROR) {
582         ngx_log_error(NGX_LOG_ALERT, cycle->log, event_list[i].data,
583             "kevent() error on %d filter:%d flags:%04Xd",
584             event_list[i].ident, event_list[i].filter,
585             event_list[i].flags);
586         continue;
587     }
588
589 #if (NGX_HAVE_TIMER_EVENT)
590
591     if (event_list[i].filter == EVFILT_TIMER) {
592         ngx_time_update();
593         continue;
594     }
595 #endif
596
597 ev = (ngx_event_t *) event_list[i].udata;
598
599 switch (event_list[i].filter) {
600
601 case EVFILT_READ:
602 case EVFILT_WRITE:
603
604     instance = (uintptr_t) ev & 1;
605     ev = (ngx_event_t *) ((uintptr_t) ev & (uintptr_t) ~1);
606
607     if (ev->closed || ev->instance != instance) {

```

```

610     /*
611     * the stale event from a file descriptor
612     * that was just closed in this iteration
613     */
614
615     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
616                  "kevent: stale event %p", ev);
617     continue;
618 }
619
620 if (ev->log && (ev->log->log_level & NGX_LOG_DEBUG_CONNECTION)) {
621     ngx_kqueue_dump_event(ev->log, &event_list[i]);
622 }
623
624 if (ev->oneshot) {
625     ev->active = 0;
626 }
627
628 ev->available = event_list[i].data;
629
630 if (event_list[i].flags & EV_EOF) {
631     ev->pending_eof = 1;
632     ev->kq_errno = event_list[i].fflags;
633 }
634
635 ev->ready = 1;
636
637 break;
638
639 case EVFILT_VNODE:
640     ev->kq_vnode = 1;
641
642     break;
643
644 case EVFILT_AIO:
645     ev->complete = 1;
646     ev->ready = 1;
647
648     break;
649
650 default:
651     ngx_log_error(NGX_LOG_ALERT, cycle->log, 0,
652                  "unexpected kevent() filter %d",
653                  event_list[i].filter);
654     continue;
655 }
656
657 if (flags & NGX_POST_EVENTS) {
658     queue = ev->accept ? &ngx_posted_accept_events
659                       : &ngx_posted_events;
660
661     ngx_post_event(ev, queue);
662
663     continue;
664 }
665
666 ev->handler(ev);
667 }
668
669 return NGX_OK;
670 }
671
672
673 static ngx_int_t
674 ngx_kqueue_process_changes(ngx_cycle_t *cycle, ngx_uint_t try)
675 {
676     int n;
677     ngx_int_t rc;
678     ngx_err_t err;
679     struct timespec ts;
680     struct kevent *changes;
681
682     ngx_mutex_lock(kevent_mutex);
683
684     ngx_mutex_lock(list_mutex);
685

```

```

686     if (nchanges == 0) {
687         ngx_mutex_unlock(list_mutex);
688         ngx_mutex_unlock(kevent_mutex);
689         return NGX_OK;
690     }
691
692     changes = change_list;
693     if (change_list == change_list0) {
694         change_list = change_list1;
695     } else {
696         change_list = change_list0;
697     }
698
699     n = (int) nchanges;
700     nchanges = 0;
701
702     ngx_mutex_unlock(list_mutex);
703
704     ts.tv_sec = 0;
705     ts.tv_nsec = 0;
706
707     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
708                  "kevent changes: %d", n);
709
710     if (kevent(ngx_kqueue, changes, n, NULL, 0, &ts) == -1) {
711         err = ngx_errno;
712         ngx_log_error((err == NGX_EINTR) ? NGX_LOG_INFO : NGX_LOG_ALERT,
713                      cycle->log, err, "kevent() failed");
714         rc = NGX_ERROR;
715     } else {
716         rc = NGX_OK;
717     }
718 }
719
720 ngx_mutex_unlock(kevent_mutex);
721
722 return rc;
723 }
724
725
726 static ngx_inline void
727 ngx_kqueue_dump_event(ngx_log_t *log, struct kevent *kev)
728 {
729     ngx_log_debug6(NGX_LOG_DEBUG_EVENT, log, 0,
730                  (kev->ident > 0x80000000 && kev->ident != (unsigned) -1) ?
731                  "kevent: %p: ft:%d fl:%04Xd ff:%08Xd d:%d ud:%p":
732                  "kevent: %d: ft:%d fl:%04Xd ff:%08Xd d:%d ud:%p",
733                  kev->ident, kev->filter,
734                  kev->flags, kev->fflags,
735                  kev->data, kev->udata);
736 }
737
738
739 static void *
740 ngx_kqueue_create_conf(ngx_cycle_t *cycle)
741 {
742     ngx_kqueue_conf_t *kcf;
743
744     kcf = ngx_palloc(cycle->pool, sizeof(ngx_kqueue_conf_t));
745     if (kcf == NULL) {
746         return NULL;
747     }
748
749     kcf->changes = NGX_CONF_UNSET;
750     kcf->events = NGX_CONF_UNSET;
751
752     return kcf;
753 }
754
755
756 static char *
757 ngx_kqueue_init_conf(ngx_cycle_t *cycle, void *conf)
758 {
759     ngx_kqueue_conf_t *kcf = conf;
760
761     ngx_conf_init_uint_value(kcf->changes, 512);

```

```
762 ngx\_conf\_init\_uint\_value(kcf->events, 512);  
763  
764 return NGX\_CONF\_OK;  
765 }
```

[One Level Up](#)

[Top Level](#)

## src/event/modules/ - nginx-1.7.10

- [ngx\\_aio\\_module.c](#)
- [ngx\\_devpoll\\_module.c](#)
- [ngx\\_epoll\\_module.c](#)
- [ngx\\_eventport\\_module.c](#)
- [ngx\\_kqueue\\_module.c](#)
- [ngx\\_poll\\_module.c](#)
- [ngx\\_rtsig\\_module.c](#)
- [ngx\\_select\\_module.c](#)
- [ngx\\_win32\\_select\\_module.c](#)



## src/event/ - nginx-1.7.10

- [modules/](#)
- [ngx\\_event.c](#)
- [ngx\\_event.h](#)
- [ngx\\_event\\_accept.c](#)
- [ngx\\_event\\_busy\\_lock.c](#)
- [ngx\\_event\\_busy\\_lock.h](#)
- [ngx\\_event\\_connect.c](#)
- [ngx\\_event\\_connect.h](#)
- [ngx\\_event\\_mutex.c](#)
- [ngx\\_event\\_openssl.c](#)
- [ngx\\_event\\_openssl.h](#)
- [ngx\\_event\\_openssl\\_stapling.c](#)
- [ngx\\_event\\_pipe.c](#)
- [ngx\\_event\\_pipe.h](#)
- [ngx\\_event\\_posted.c](#)
- [ngx\\_event\\_posted.h](#)
- [ngx\\_event\\_timer.c](#)
- [ngx\\_event\\_timer.h](#)

## src/event/nginx\_event.c - nginx-1.7.10

### Global variables defined

- [connection\\_counter](#)
- [event\\_core\\_name](#)
- [ngx\\_accept\\_disabled](#)
- [ngx\\_accept\\_events](#)
- [ngx\\_accept\\_mutex](#)
- [ngx\\_accept\\_mutex\\_delay](#)
- [ngx\\_accept\\_mutex\\_held](#)
- [ngx\\_accept\\_mutex\\_ptr](#)
- [ngx\\_connection\\_counter](#)
- [ngx\\_event\\_actions](#)
- [ngx\\_event\\_core\\_commands](#)
- [ngx\\_event\\_core\\_module](#)
- [ngx\\_event\\_core\\_module\\_ctx](#)
- [ngx\\_event\\_flags](#)
- [ngx\\_event\\_max\\_module](#)
- [ngx\\_event\\_timer\\_alarm](#)
- [ngx\\_events\\_commands](#)
- [ngx\\_events\\_module](#)
- [ngx\\_events\\_module\\_ctx](#)
- [ngx\\_stat\\_accepted](#)
- [ngx\\_stat\\_accepted0](#)
- [ngx\\_stat\\_active](#)
- [ngx\\_stat\\_active0](#)
- [ngx\\_stat\\_handled](#)
- [ngx\\_stat\\_handled0](#)
- [ngx\\_stat\\_reading](#)
- [ngx\\_stat\\_reading0](#)
- [ngx\\_stat\\_requests](#)
- [ngx\\_stat\\_requests0](#)
- [ngx\\_stat\\_waiting](#)

- [ngx\\_stat\\_waiting0](#)
- [ngx\\_stat\\_writing](#)
- [ngx\\_stat\\_writing0](#)
- [ngx\\_timer\\_resolution](#)
- [ngx\\_use\\_accept\\_mutex](#)

## Functions defined

- [ngx\\_event\\_connections](#)
- [ngx\\_event\\_core\\_create\\_conf](#)
- [ngx\\_event\\_core\\_init\\_conf](#)
- [ngx\\_event\\_debug\\_connection](#)
- [ngx\\_event\\_init\\_conf](#)
- [ngx\\_event\\_module\\_init](#)
- [ngx\\_event\\_process\\_init](#)
- [ngx\\_event\\_use](#)
- [ngx\\_events\\_block](#)
- [ngx\\_handle\\_read\\_event](#)
- [ngx\\_handle\\_write\\_event](#)
- [ngx\\_process\\_events\\_and\\_timers](#)
- [ngx\\_send\\_lowat](#)
- [ngx\\_timer\\_signal\\_handler](#)

## Macros defined

- [DEFAULT\\_CONNECTIONS](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 #define DEFAULT_CONNECTIONS 512
14
15
16 extern ngx_module_t ngx_kqueue_module;
17 extern ngx_module_t ngx_eventport_module;
18 extern ngx_module_t ngx_devpoll_module;
19 extern ngx_module_t ngx_epoll_module;
20 extern ngx_module_t ngx_rtsig_module;

```

```

21 extern ngx_module_t ngx_select_module;
22
23
24 static char *ngx_event_init_conf(ngx_cycle_t *cycle, void *conf);
25 static ngx_int_t ngx_event_module_init(ngx_cycle_t *cycle);
26 static ngx_int_t ngx_event_process_init(ngx_cycle_t *cycle);
27 static char *ngx_events_block(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
28
29 static char *ngx_event_connections(ngx_conf_t *cf, ngx_command_t *cmd,
30     void *conf);
31 static char *ngx_event_use(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
32 static char *ngx_event_debug_connection(ngx_conf_t *cf, ngx_command_t *cmd,
33     void *conf);
34
35 static void *ngx_event_core_create_conf(ngx_cycle_t *cycle);
36 static char *ngx_event_core_init_conf(ngx_cycle_t *cycle, void *conf);
37
38
39 static ngx_uint_t     ngx_timer_resolution;
40 sig_atomic_t         ngx_event_timer_alarm;
41
42 static ngx_uint_t     ngx_event_max_module;
43
44 ngx_uint_t           ngx_event_flags;
45 ngx_event_actions_t  ngx_event_actions;
46
47
48 static ngx_atomic_t   connection_counter = 1;
49 ngx_atomic_t         *ngx_connection_counter = &connection_counter;
50
51
52 ngx_atomic_t         *ngx_accept_mutex_ptr;
53 ngx_shmtx_t          ngx_accept_mutex;
54 ngx_uint_t           ngx_use_accept_mutex;
55 ngx_uint_t           ngx_accept_events;
56 ngx_uint_t           ngx_accept_mutex_held;
57 ngx_msec_t           ngx_accept_mutex_delay;
58 ngx_int_t            ngx_accept_disabled;
59
60
61 #if (NGX_STAT_STUB)
62
63 ngx_atomic_t         ngx_stat_accepted0;
64 ngx_atomic_t         *ngx_stat_accepted = &ngx_stat_accepted0;
65 ngx_atomic_t         ngx_stat_handled0;
66 ngx_atomic_t         *ngx_stat_handled = &ngx_stat_handled0;
67 ngx_atomic_t         ngx_stat_requests0;
68 ngx_atomic_t         *ngx_stat_requests = &ngx_stat_requests0;
69 ngx_atomic_t         ngx_stat_active0;
70 ngx_atomic_t         *ngx_stat_active = &ngx_stat_active0;
71 ngx_atomic_t         ngx_stat_reading0;
72 ngx_atomic_t         *ngx_stat_reading = &ngx_stat_reading0;
73 ngx_atomic_t         ngx_stat_writing0;
74 ngx_atomic_t         *ngx_stat_writing = &ngx_stat_writing0;
75 ngx_atomic_t         ngx_stat_waiting0;
76 ngx_atomic_t         *ngx_stat_waiting = &ngx_stat_waiting0;
77
78 #endif
79
80
81
82 static ngx_command_t  ngx_events_commands[] = {
83
84     { ngx_string("events"),
85       NGX_MAIN_CONF|NGX_CONF_BLOCK|NGX_CONF_NOARGS,
86       ngx_events_block,
87       0,
88       0,
89       NULL },
90
91     ngx_null_command
92 };
93
94
95 static ngx_core_module_t  ngx_events_module_ctx = {
96     ngx_string("events"),

```

```

97     NULL,
98     ngx\_event\_init\_conf
99 };
100
101
102 ngx\_module\_t ngx\_events\_module = {
103     NGX\_MODULE\_V1,
104     &ngx\_events\_module\_ctx,           /* module context */
105     ngx\_events\_commands,           /* module directives */
106     NGX\_CORE\_MODULE,               /* module type */
107     NULL,                            /* init master */
108     NULL,                            /* init module */
109     NULL,                            /* init process */
110     NULL,                            /* init thread */
111     NULL,                            /* exit thread */
112     NULL,                            /* exit process */
113     NULL,                            /* exit master */
114     NGX\_MODULE\_V1\_PADDING
115 };
116
117
118 static ngx\_str\_t event\_core\_name = ngx\_string("event_core");
119
120
121 static ngx\_command\_t ngx\_event\_core\_commands[] = {
122
123     { ngx\_string("worker_connections"),
124       NGX\_EVENT\_CONF|NGX\_CONF\_TAKE1,
125       ngx\_event\_connections,
126       0,
127       0,
128       NULL },
129
130     { ngx\_string("connections"),
131       NGX\_EVENT\_CONF|NGX\_CONF\_TAKE1,
132       ngx\_event\_connections,
133       0,
134       0,
135       NULL },
136
137     { ngx\_string("use"),
138       NGX\_EVENT\_CONF|NGX\_CONF\_TAKE1,
139       ngx\_event\_use,
140       0,
141       0,
142       NULL },
143
144     { ngx\_string("multi_accept"),
145       NGX\_EVENT\_CONF|NGX\_CONF\_FLAG,
146       ngx\_conf\_set\_flag\_slot,
147       0,
148       offsetof(ngx\_event\_conf\_t, multi\_accept),
149       NULL },
150
151     { ngx\_string("accept_mutex"),
152       NGX\_EVENT\_CONF|NGX\_CONF\_FLAG,
153       ngx\_conf\_set\_flag\_slot,
154       0,
155       offsetof(ngx\_event\_conf\_t, accept\_mutex),
156       NULL },
157
158     { ngx\_string("accept_mutex_delay"),
159       NGX\_EVENT\_CONF|NGX\_CONF\_TAKE1,
160       ngx\_conf\_set\_msec\_slot,
161       0,
162       offsetof(ngx\_event\_conf\_t, accept\_mutex\_delay),
163       NULL },
164
165     { ngx\_string("debug_connection"),
166       NGX\_EVENT\_CONF|NGX\_CONF\_TAKE1,
167       ngx\_event\_debug\_connection,
168       0,
169       0,
170       NULL },
171
172     ngx\_null\_command

```

```

173 };
174
175
176 ngx_event_module_t ngx_event_core_module_ctx = {
177     &event_core_name,
178     ngx_event_core_create_conf,          /* create configuration */
179     ngx_event_core_init_conf,          /* init configuration */
180
181     { NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL }
182 };
183
184
185 ngx_module_t ngx_event_core_module = {
186     NGX_MODULE_V1,
187     &ngx_event_core_module_ctx,        /* module context */
188     ngx_event_core_commands,          /* module directives */
189     NGX_EVENT_MODULE,                 /* module type */
190     NULL,                              /* init master */
191     ngx_event_module_init,            /* init module */
192     ngx_event_process_init,          /* init process */
193     NULL,                              /* init thread */
194     NULL,                              /* exit thread */
195     NULL,                              /* exit process */
196     NULL,                              /* exit master */
197     NGX_MODULE_V1_PADDING
198 };
199
200
201 void
202 ngx_process_events_and_timers(ngx_cycle_t *cycle)
203 {
204     ngx_uint_t  flags;
205     ngx_msec_t  timer, delta;
206
207     if (ngx_timer_resolution) {
208         timer = NGX_TIMER_INFINITE;
209         flags = 0;
210     }
211     else {
212         timer = ngx_event_find_timer();
213         flags = NGX_UPDATE_TIME;
214     }
215     #if (NGX_THREADS)
216
217         if (timer == NGX_TIMER_INFINITE || timer > 500) {
218             timer = 500;
219         }
220
221     #endif
222     }
223
224     if (ngx_use_accept_mutex) {
225         if (ngx_accept_disabled > 0) {
226             ngx_accept_disabled--;
227         }
228         else {
229             if (ngx_trylock_accept_mutex(cycle) == NGX_ERROR) {
230                 return;
231             }
232
233             if (ngx_accept_mutex_held) {
234                 flags |= NGX_POST_EVENTS;
235             }
236             else {
237                 if (timer == NGX_TIMER_INFINITE
238                     || timer > ngx_accept_mutex_delay)
239                 {
240                     timer = ngx_accept_mutex_delay;
241                 }
242             }
243         }
244     }
245
246     delta = ngx_current_msec;
247
248     (void) ngx_process_events(cycle, timer, flags);

```

```

249 delta = ngx_current_msec - delta;
250
251 ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
252              "timer delta: %M", delta);
253
254 ngx_event_process_posted(cycle, &ngx_posted_accept_events);
255
256 if (ngx_accept_mutex_held) {
257     ngx_shmtx_unlock(&ngx_accept_mutex);
258 }
259
260 if (delta) {
261     ngx_event_expire_timers();
262 }
263
264 ngx_event_process_posted(cycle, &ngx_posted_events);
265 }
266
267
268 ngx_int_t
269 ngx_handle_read_event(ngx_event_t *rev, ngx_uint_t flags)
270 {
271     if (ngx_event_flags & NGX_USE_CLEAR_EVENT) {
272         /* kqueue, epoll */
273
274         if (!rev->active && !rev->ready) {
275             if (ngx_add_event(rev, NGX_READ_EVENT, NGX_CLEAR_EVENT)
276                 == NGX_ERROR)
277             {
278                 return NGX_ERROR;
279             }
280         }
281
282         return NGX_OK;
283     }
284     else if (ngx_event_flags & NGX_USE_LEVEL_EVENT) {
285         /* select, poll, /dev/poll */
286
287         if (!rev->active && !rev->ready) {
288             if (ngx_add_event(rev, NGX_READ_EVENT, NGX_LEVEL_EVENT)
289                 == NGX_ERROR)
290             {
291                 return NGX_ERROR;
292             }
293         }
294
295         return NGX_OK;
296     }
297     else if (rev->active && (rev->ready || (flags & NGX_CLOSE_EVENT))) {
298         if (ngx_del_event(rev, NGX_READ_EVENT, NGX_LEVEL_EVENT | flags)
299             == NGX_ERROR)
300         {
301             return NGX_ERROR;
302         }
303
304         return NGX_OK;
305     }
306     else if (ngx_event_flags & NGX_USE_EVENTPORT_EVENT) {
307         /* event ports */
308
309         if (!rev->active && !rev->ready) {
310             if (ngx_add_event(rev, NGX_READ_EVENT, 0) == NGX_ERROR) {
311                 return NGX_ERROR;
312             }
313         }
314
315         return NGX_OK;
316     }
317     else if (rev->oneshot && !rev->ready) {
318         if (ngx_del_event(rev, NGX_READ_EVENT, 0) == NGX_ERROR) {
319             return NGX_ERROR;
320         }
321     }
322 }

```

```

325     }
326
327     return NGX\_OK;
328 }
329 }
330
331 /* aio, iocp, rtsig */
332
333 return NGX\_OK;
334 }
335
336
337 ngx\_int\_t
338 ngx\_handle\_write\_event(ngx\_event\_t *wev, size\_t lowat)
339 {
340     ngx\_connection\_t *c;
341
342     if (lowat) {
343         c = wev->data;
344
345         if (ngx\_send\_lowat(c, lowat) == NGX\_ERROR) {
346             return NGX\_ERROR;
347         }
348     }
349
350     if (ngx\_event\_flags & NGX\_USE\_CLEAR\_EVENT) {
351
352         /* kqueue, epoll */
353
354         if (!wev->active && !wev->ready) {
355             if (ngx\_add\_event(wev, NGX\_WRITE\_EVENT,
356                 NGX\_CLEAR\_EVENT | (lowat ? NGX\_LOWAT\_EVENT : 0))
357                 == NGX\_ERROR)
358             {
359                 return NGX\_ERROR;
360             }
361         }
362
363         return NGX\_OK;
364     } else if (ngx\_event\_flags & NGX\_USE\_LEVEL\_EVENT) {
365
366         /* select, poll, /dev/poll */
367
368         if (!wev->active && !wev->ready) {
369             if (ngx\_add\_event(wev, NGX\_WRITE\_EVENT, NGX\_LEVEL\_EVENT)
370                 == NGX\_ERROR)
371             {
372                 return NGX\_ERROR;
373             }
374
375             return NGX\_OK;
376         }
377
378         if (wev->active && wev->ready) {
379             if (ngx\_del\_event(wev, NGX\_WRITE\_EVENT, NGX\_LEVEL\_EVENT)
380                 == NGX\_ERROR)
381             {
382                 return NGX\_ERROR;
383             }
384
385             return NGX\_OK;
386         }
387     }
388 } else if (ngx\_event\_flags & NGX\_USE\_EVENTPORT\_EVENT) {
389
390     /* event ports */
391
392     if (!wev->active && !wev->ready) {
393         if (ngx\_add\_event(wev, NGX\_WRITE\_EVENT, 0) == NGX\_ERROR) {
394             return NGX\_ERROR;
395         }
396     }
397
398     return NGX\_OK;
399 }
400

```



```

401     if (wev->oneshot && wev->ready) {
402         if (ngx_del_event(wev, NGX_WRITE_EVENT, 0) == NGX_ERROR) {
403             return NGX_ERROR;
404         }
405
406         return NGX_OK;
407     }
408 }
409
410 /* aio, iocp, rtsig */
411
412 return NGX_OK;
413 }
414
415
416 static char *
417 ngx_event_init_conf(ngx_cycle_t *cycle, void *conf)
418 {
419     if (ngx_get_conf(cycle->conf_ctx, ngx_events_module) == NULL) {
420         ngx_log_error(NGX_LOG_EMERG, cycle->log, 0,
421             "no \"events\" section in configuration");
422         return NGX_CONF_ERROR;
423     }
424
425     return NGX_CONF_OK;
426 }
427
428
429 static ngx_int_t
430 ngx_event_module_init(ngx_cycle_t *cycle)
431 {
432     void                ***cf;
433     u_char              *shared;
434     size_t              size, cl;
435     ngx_shm_t          shm;
436     ngx_time_t        *tp;
437     ngx_core_conf_t   *ccf;
438     ngx_event_conf_t *ecf;
439
440     cf = ngx_get_conf(cycle->conf_ctx, ngx_events_module);
441     ecf = (*cf)[ngx_event_core_module.ctx_index];
442
443     if (!ngx_test_config && ngx_process <= NGX_PROCESS_MASTER) {
444         ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0,
445             "using the \"%s\" event method", ecf->name);
446     }
447
448     ccf = (ngx_core_conf_t *) ngx_get_conf(cycle->conf_ctx, ngx_core_module);
449
450     ngx_timer_resolution = ccf->timer_resolution;
451
452 #if !(NGX_WIN32)
453 {
454     ngx_int_t    limit;
455     struct rlimit rlmt;
456
457     if (getrlimit(RLIMIT_NOFILE, &rlmt) == -1) {
458         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
459             "getrlimit(RLIMIT_NOFILE) failed, ignored");
460     }
461 } else {
462     if (ecf->connections > (ngx_uint_t) rlmt.rlim_cur
463         && (ccf->rlimit_nofile == NGX_CONF_UNSET
464             || ecf->connections > (ngx_uint_t) ccf->rlimit_nofile))
465     {
466         limit = (ccf->rlimit_nofile == NGX_CONF_UNSET) ?
467             (ngx_int_t) rlmt.rlim_cur : ccf->rlimit_nofile;
468
469         ngx_log_error(NGX_LOG_WARN, cycle->log, 0,
470             "%ui worker_connections exceed "
471             "open file resource limit: %i",
472             ecf->connections, limit);
473     }
474 }
475 }
476 #endif /* !(NGX_WIN32) */

```

```

477
478
479     if (ccf->master == 0) {
480         return NGX_OK;
481     }
482
483     if (ngx_accept_mutex_ptr) {
484         return NGX_OK;
485     }
486
487     /* cl should be equal to or greater than cache line size */
488
489     cl = 128;
490
491     size = cl          /* ngx_accept_mutex */
492           + cl        /* ngx_connection_counter */
493           + cl;      /* ngx_temp_number */
494
495 #if (NGX_STAT_STUB)
496
497     size += cl        /* ngx_stat_accepted */
498           + cl        /* ngx_stat_handled */
499           + cl        /* ngx_stat_requests */
500           + cl        /* ngx_stat_active */
501           + cl        /* ngx_stat_reading */
502           + cl        /* ngx_stat_writing */
503           + cl;      /* ngx_stat_waiting */
504
505 #endif
506
507     shm.size = size;
508     shm.name.len = sizeof("nginx_shared_zone");
509     shm.name.data = (u_char *) "nginx_shared_zone";
510     shm.log = cycle->log;
511
512     if (ngx_shm_alloc(&shm) != NGX_OK) {
513         return NGX_ERROR;
514     }
515
516     shared = shm.addr;
517
518     ngx_accept_mutex_ptr = (ngx_atomic_t *) shared;
519     ngx_accept_mutex.spin = (ngx_uint_t) -1;
520
521     if (ngx_shmtx_create(&ngx_accept_mutex, (ngx_shmtx_sh_t *) shared,
522                         cycle->lock_file.data)
523         != NGX_OK)
524     {
525         return NGX_ERROR;
526     }
527
528     ngx_connection_counter = (ngx_atomic_t *) (shared + 1 * cl);
529
530     (void) ngx_atomic_cmp_set(ngx_connection_counter, 0, 1);
531
532     ngx_log_debug2(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
533                  "counter: %p, %d",
534                  ngx_connection_counter, *ngx_connection_counter);
535
536     ngx_temp_number = (ngx_atomic_t *) (shared + 2 * cl);
537
538     tp = ngx_timeofday();
539
540     ngx_random_number = (tp->msec << 16) + ngx_pid;
541
542 #if (NGX_STAT_STUB)
543
544     ngx_stat_accepted = (ngx_atomic_t *) (shared + 3 * cl);
545     ngx_stat_handled = (ngx_atomic_t *) (shared + 4 * cl);
546     ngx_stat_requests = (ngx_atomic_t *) (shared + 5 * cl);
547     ngx_stat_active = (ngx_atomic_t *) (shared + 6 * cl);
548     ngx_stat_reading = (ngx_atomic_t *) (shared + 7 * cl);
549     ngx_stat_writing = (ngx_atomic_t *) (shared + 8 * cl);
550     ngx_stat_waiting = (ngx_atomic_t *) (shared + 9 * cl);
551
552

```

```

553 #endif
554
555     return NGX_OK;
556 }
557
558
559 #if !(NGX_WIN32)
560
561 static void
562 ngx_timer_signal_handler(int signo)
563 {
564     ngx_event_timer_alarm = 1;
565
566 #if 1
567     ngx_log_debug0(NGX_LOG_DEBUG_EVENT, ngx_cycle->log, 0, "timer signal");
568 #endif
569 }
570
571 #endif
572
573
574 static ngx_int_t
575 ngx_event_process_init(ngx_cycle_t *cycle)
576 {
577     ngx_uint_t          m, i;
578     ngx_event_t       *rev, *wev;
579     ngx_listening_t   *ls;
580     ngx_connection_t *c, *next, *old;
581     ngx_core_conf_t   *ccf;
582     ngx_event_conf_t  *ecf;
583     ngx_event_module_t *module;
584
585     ccf = (ngx_core_conf_t *) ngx_get_conf(cycle->conf_ctx, ngx_core_module);
586     ecf = ngx_event_get_conf(cycle->conf_ctx, ngx_event_core_module);
587
588     if (ccf->master && ccf->worker_processes > 1 && ecf->accept_mutex) {
589         ngx_use_accept_mutex = 1;
590         ngx_accept_mutex_held = 0;
591         ngx_accept_mutex_delay = ecf->accept_mutex_delay;
592     } else {
593         ngx_use_accept_mutex = 0;
594     }
595
596 #if (NGX_WIN32)
597
598     /*
599     * disable accept mutex on win32 as it may cause deadlock if
600     * grabbed by a process which can't accept connections
601     */
602
603     ngx_use_accept_mutex = 0;
604
605 #endif
606
607     ngx_queue_init(&ngx_posted_accept_events);
608     ngx_queue_init(&ngx_posted_events);
609
610     if (ngx_event_timer_init(cycle->log) == NGX_ERROR) {
611         return NGX_ERROR;
612     }
613
614     for (m = 0; ngx_modules[m]; m++) {
615         if (ngx_modules[m]->type != NGX_EVENT_MODULE) {
616             continue;
617         }
618
619         if (ngx_modules[m]->ctx_index != ecf->use) {
620             continue;
621         }
622
623         module = ngx_modules[m]->ctx;
624
625         if (module->actions.init(cycle, ngx_timer_resolution) != NGX_OK) {
626             /* fatal */
627             exit(2);
628         }

```

```

629     }
630
631     break;
632 }
633
634 #if !(NGX_WIN32)
635
636 if (ngx_timer_resolution && !(ngx_event_flags & NGX_USE_TIMER_EVENT)) {
637     struct sigaction sa;
638     struct itimerval itv;
639
640     ngx_memzero(&sa, sizeof(struct sigaction));
641     sa.sa_handler = ngx_timer_signal_handler;
642     sigemptyset(&sa.sa_mask);
643
644     if (sigaction(SIGALRM, &sa, NULL) == -1) {
645         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
646             "sigaction(SIGALRM) failed");
647         return NGX_ERROR;
648     }
649
650     itv.it_interval.tv_sec = ngx_timer_resolution / 1000;
651     itv.it_interval.tv_usec = (ngx_timer_resolution % 1000) * 1000;
652     itv.it_value.tv_sec = ngx_timer_resolution / 1000;
653     itv.it_value.tv_usec = (ngx_timer_resolution % 1000) * 1000;
654
655     if (setitimer(ITIMER_REAL, &itv, NULL) == -1) {
656         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
657             "setitimer() failed");
658     }
659 }
660
661 if (ngx_event_flags & NGX_USE_FD_EVENT) {
662     struct rlimit rlimt;
663
664     if (getrlimit(RLIMIT_NOFILE, &rlimt) == -1) {
665         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
666             "getrlimit(RLIMIT_NOFILE) failed");
667         return NGX_ERROR;
668     }
669
670     cycle->files_n = (ngx_uint_t) rlimt.rlim_cur;
671
672     cycle->files = ngx_alloc(sizeof(ngx_connection_t) * cycle->files_n,
673         cycle->log);
674     if (cycle->files == NULL) {
675         return NGX_ERROR;
676     }
677 }
678
679 #endif
680
681 cycle->connections =
682     ngx_alloc(sizeof(ngx_connection_t) * cycle->connection_n, cycle->log);
683 if (cycle->connections == NULL) {
684     return NGX_ERROR;
685 }
686
687 c = cycle->connections;
688
689 cycle->read_events = ngx_alloc(sizeof(ngx_event_t) * cycle->connection_n,
690     cycle->log);
691 if (cycle->read_events == NULL) {
692     return NGX_ERROR;
693 }
694
695 rev = cycle->read_events;
696 for (i = 0; i < cycle->connection_n; i++) {
697     rev[i].closed = 1;
698     rev[i].instance = 1;
699 }
700
701 cycle->write_events = ngx_alloc(sizeof(ngx_event_t) * cycle->connection_n,
702     cycle->log);
703 if (cycle->write_events == NULL) {
704     return NGX_ERROR;

```

```

705     }
706
707     wev = cycle->write_events;
708     for (i = 0; i < cycle->connection_n; i++) {
709         wev[i].closed = 1;
710     }
711
712     i = cycle->connection_n;
713     next = NULL;
714
715     do {
716         i--;
717
718         c[i].data = next;
719         c[i].read = &cycle->read_events[i];
720         c[i].write = &cycle->write_events[i];
721         c[i].fd = (ngx\_socket\_t) -1;
722
723         next = &c[i];
724
725     #if (NGX_THREADS)
726         c[i].lock = 0;
727     #endif
728     } while (i);
729
730     cycle->free_connections = next;
731     cycle->free_connection_n = cycle->connection_n;
732
733     /* for each listening socket */
734
735     ls = cycle->listening.elts;
736     for (i = 0; i < cycle->listening.nelts; i++) {
737
738         c = ngx\_get\_connection(ls[i].fd, cycle->log);
739
740         if (c == NULL) {
741             return NGX\_ERROR;
742         }
743
744         c->log = &ls[i].log;
745
746         c->listening = &ls[i];
747         ls[i].connection = c;
748
749         rev = c->read;
750
751         rev->log = c->log;
752         rev->accept = 1;
753
754     #if (NGX_HAVE_DEFERRED_ACCEPT)
755         rev->deferred_accept = ls[i].deferred_accept;
756     #endif
757
758     if (!(ngx\_event\_flags & NGX\_USE\_IOCP\_EVENT)) {
759         if (ls[i].previous) {
760
761             /*
762              * delete the old accept events that were bound to
763              * the old cycle read events array
764              */
765
766             old = ls[i].previous->connection;
767
768             if (ngx\_del\_event(old->read, NGX\_READ\_EVENT, NGX\_CLOSE\_EVENT)
769                 == NGX\_ERROR)
770             {
771                 return NGX\_ERROR;
772             }
773
774             old->fd = (ngx\_socket\_t) -1;
775         }
776     }
777
778     #if (NGX_WIN32)
779
780     if (ngx\_event\_flags & NGX\_USE\_IOCP\_EVENT) {

```

```

781     ngx_iocp_conf_t  *iocpcf;
782
783     rev->handler = ngx_event_acceptex;
784
785     if (ngx_use_accept_mutex) {
786         continue;
787     }
788
789     if (ngx_add_event(rev, 0, NGX_IOCP_ACCEPT) == NGX_ERROR) {
790         return NGX_ERROR;
791     }
792
793     ls[i].log.handler = ngx_acceptex_log_error;
794
795     iocpcf = ngx_event_get_conf(cycle->conf_ctx, ngx_iocp_module);
796     if (ngx_event_post_acceptex(&ls[i], iocpcf->post_acceptex)
797         == NGX_ERROR)
798     {
799         return NGX_ERROR;
800     }
801
802 } else {
803     rev->handler = ngx_event_accept;
804
805     if (ngx_use_accept_mutex) {
806         continue;
807     }
808
809     if (ngx_add_event(rev, NGX_READ_EVENT, 0) == NGX_ERROR) {
810         return NGX_ERROR;
811     }
812 }
813
814 #else
815
816     rev->handler = ngx_event_accept;
817
818     if (ngx_use_accept_mutex) {
819         continue;
820     }
821
822     if (ngx_event_flags & NGX_USE_RT_SIG_EVENT) {
823         if (ngx_add_conn(c) == NGX_ERROR) {
824             return NGX_ERROR;
825         }
826     }
827 } else {
828     if (ngx_add_event(rev, NGX_READ_EVENT, 0) == NGX_ERROR) {
829         return NGX_ERROR;
830     }
831 }
832
833 #endif
834
835     }
836
837     return NGX_OK;
838 }
839
840
841 ngx_int_t
842 ngx_send_lowat(ngx_connection_t *c, size_t lowat)
843 {
844     int sndlowat;
845
846     #if (NGX_HAVE_LOWAT_EVENT)
847
848     if (ngx_event_flags & NGX_USE_KQUEUE_EVENT) {
849         c->write->available = lowat;
850         return NGX_OK;
851     }
852
853 #endif
854
855     if (lowat == 0 || c->sndlowat) {
856         return NGX_OK;

```

```

857 }
858
859 sndlowat = (int) lowat;
860
861 if (setsockopt(c->fd, SOL_SOCKET, SO_SNDLOWAT,
862             (const void *) &sndlowat, sizeof(int))
863     == -1)
864 {
865     ngx_connection_error(c, ngx_socket_errno,
866                         "setsockopt(SO_SNDLOWAT) failed");
867     return NGX_ERROR;
868 }
869
870 c->sndlowat = 1;
871
872 return NGX_OK;
873 }
874
875
876 static char *
877 ngx_events_block(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
878 {
879     char          *rv;
880     void          ***ctx;
881     ngx_uint_t    i;
882     ngx_conf_t    pcf;
883     ngx_event_module_t *m;
884
885     if (*(void **) conf) {
886         return "is duplicate";
887     }
888
889     /* count the number of the event modules and set up their indices */
890
891     ngx_event_max_module = 0;
892     for (i = 0; ngx_modules[i]; i++) {
893         if (ngx_modules[i]->type != NGX_EVENT_MODULE) {
894             continue;
895         }
896
897         ngx_modules[i]->ctx_index = ngx_event_max_module++;
898     }
899
900     ctx = ngx_palloc(cf->pool, sizeof(void *));
901     if (ctx == NULL) {
902         return NGX_CONF_ERROR;
903     }
904
905     *ctx = ngx_palloc(cf->pool, ngx_event_max_module * sizeof(void *));
906     if (*ctx == NULL) {
907         return NGX_CONF_ERROR;
908     }
909
910     *(void **) conf = ctx;
911
912     for (i = 0; ngx_modules[i]; i++) {
913         if (ngx_modules[i]->type != NGX_EVENT_MODULE) {
914             continue;
915         }
916
917         m = ngx_modules[i]->ctx;
918
919         if (m->create_conf) {
920             (*ctx)[ngx_modules[i]->ctx_index] = m->create_conf(cf->cycle);
921             if ((*ctx)[ngx_modules[i]->ctx_index] == NULL) {
922                 return NGX_CONF_ERROR;
923             }
924         }
925     }
926
927     pcf = *cf;
928     cf->ctx = ctx;
929     cf->module_type = NGX_EVENT_MODULE;
930     cf->cmd_type = NGX_EVENT_CONF;
931
932     rv = ngx_conf_parse(cf, NULL);

```

```

933     *cf = pcf;
934
935
936     if (rv != NGX\_CONF\_OK)
937         return rv;
938
939     for (i = 0; ngx_modules[i]; i++) {
940         if (ngx_modules[i]->type != NGX\_EVENT\_MODULE) {
941             continue;
942         }
943
944         m = ngx_modules[i]->ctx;
945
946         if (m->init_conf) {
947             rv = m->init_conf(cf->cycle, (*ctx)[ngx_modules[i]->ctx_index]);
948             if (rv != NGX\_CONF\_OK) {
949                 return rv;
950             }
951         }
952     }
953
954     return NGX\_CONF\_OK;
955 }
956
957
958 static char *
959 ngx_event_connections(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
960 {
961     ngx\_event\_conf\_t *ecf = conf;
962
963     ngx\_str\_t *value;
964
965     if (ecf->connections != NGX\_CONF\_UNSET\_UINT) {
966         return "is duplicate";
967     }
968
969     if (ngx\_strcmp(cmd->name.data, "connections") == 0) {
970         ngx\_conf\_log\_error(NGX\_LOG\_WARN, cf, 0,
971             "the \"connections\" directive is deprecated, "
972             "use the \"worker_connections\" directive instead");
973     }
974
975     value = cf->args->elts;
976     ecf->connections = ngx\_atoi(value[1].data, value[1].len);
977     if (ecf->connections == (ngx\_uint\_t) NGX\_ERROR) {
978         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
979             "invalid number \"%V\"", &value[1]);
980
981         return NGX\_CONF\_ERROR;
982     }
983
984     cf->cycle->connection_n = ecf->connections;
985
986     return NGX\_CONF\_OK;
987 }
988
989
990 static char *
991 ngx_event_use(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
992 {
993     ngx\_event\_conf\_t *ecf = conf;
994
995     ngx\_int\_t m;
996     ngx\_str\_t *value;
997     ngx\_event\_conf\_t *old_ecf;
998     ngx\_event\_module\_t *module;
999
1000     if (ecf->use != NGX\_CONF\_UNSET\_UINT) {
1001         return "is duplicate";
1002     }
1003
1004     value = cf->args->elts;
1005
1006     if (cf->cycle->old_cycle->conf_ctx) {
1007         old_ecf = ngx\_event\_get\_conf(cf->cycle->old_cycle->conf_ctx,
1008             ngx\_event\_core\_module);

```



```

1009 } else {
1010     old_ecf = NULL;
1011 }
1012
1013
1014 for (m = 0; ngx_modules[m]; m++) {
1015     if (ngx_modules[m]->type != NGX_EVENT_MODULE) {
1016         continue;
1017     }
1018
1019     module = ngx_modules[m]->ctx;
1020     if (module->name->len == value[1].len) {
1021         if (ngx_strcmp(module->name->data, value[1].data) == 0) {
1022             ecf->use = ngx_modules[m]->ctx_index;
1023             ecf->name = module->name->data;
1024
1025             if (ngx_process == NGX_PROCESS_SINGLE
1026                 && old_ecf
1027                 && old_ecf->use != ecf->use)
1028             {
1029                 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1030                     "when the server runs without a master process "
1031                     "the \"%V\" event type must be the same as "
1032                     "in previous configuration - \"%s\" "
1033                     "and it cannot be changed on the fly, "
1034                     "to change it you need to stop server "
1035                     "and start it again",
1036                     &value[1], old_ecf->name);
1037
1038                 return NGX_CONF_ERROR;
1039             }
1040
1041             return NGX_CONF_OK;
1042         }
1043     }
1044 }
1045
1046 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1047     "invalid event type \"%V\"", &value[1]);
1048
1049 return NGX_CONF_ERROR;
1050 }
1051
1052
1053 static char *
1054 ngx_event_debug_connection(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
1055 {
1056     #if (NGX_DEBUG)
1057         ngx_event_conf_t *ecf = conf;
1058
1059         ngx_int_t         rc;
1060         ngx_str_t         *value;
1061         ngx_url_t         u;
1062         ngx_cidr_t         c, *cidr;
1063         ngx_uint_t         i;
1064         struct sockaddr_in *sin;
1065     #if (NGX_HAVE_INET6)
1066         struct sockaddr_in6 *sin6;
1067     #endif
1068
1069     value = cf->args->elts;
1070
1071     #if (NGX_HAVE_UNIX_DOMAIN)
1072
1073     if (ngx_strcmp(value[1].data, "unix:") == 0) {
1074         cidr = ngx_array_push(&ecf->debug_connection);
1075         if (cidr == NULL) {
1076             return NGX_CONF_ERROR;
1077         }
1078
1079         cidr->family = AF_UNIX;
1080         return NGX_CONF_OK;
1081     }
1082
1083 #endif
1084

```

```

1085 rc = ngx_ptocidr(&value[1], &c);
1086
1087 if (rc != NGX_ERROR) {
1088     if (rc == NGX_DONE) {
1089         ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
1090             "low address bits of %V are meaningless",
1091             &value[1]);
1092     }
1093
1094     cidr = ngx_array_push(&ecf->debug_connection);
1095     if (cidr == NULL) {
1096         return NGX_CONF_ERROR;
1097     }
1098
1099     *cidr = c;
1100
1101     return NGX_CONF_OK;
1102 }
1103
1104 ngx_memzero(&u, sizeof(ngx_url_t));
1105 u.host = value[1];
1106
1107 if (ngx_inet_resolve_host(cf->pool, &u) != NGX_OK) {
1108     if (u.err) {
1109         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1110             "%s in debug_connection \"%V\"",
1111             u.err, &u.host);
1112     }
1113
1114     return NGX_CONF_ERROR;
1115 }
1116
1117 cidr = ngx_array_push_n(&ecf->debug_connection, u.naddrs);
1118 if (cidr == NULL) {
1119     return NGX_CONF_ERROR;
1120 }
1121
1122 ngx_memzero(cidr, u.naddrs * sizeof(ngx_cidr_t));
1123
1124 for (i = 0; i < u.naddrs; i++) {
1125     cidr[i].family = u.addrs[i].sockaddr->sa_family;
1126
1127     switch (cidr[i].family) {
1128
1129 #if (NGX_HAVE_INET6)
1130     case AF_INET6:
1131         sin6 = (struct sockaddr_in6 *) u.addrs[i].sockaddr;
1132         cidr[i].u.in6.addr = sin6->sin6_addr;
1133         ngx_memset(cidr[i].u.in6.mask.s6_addr, 0xff, 16);
1134         break;
1135 #endif
1136
1137     default: /* AF_INET */
1138         sin = (struct sockaddr_in *) u.addrs[i].sockaddr;
1139         cidr[i].u.in.addr = sin->sin_addr.s_addr;
1140         cidr[i].u.in.mask = 0xffffffff;
1141         break;
1142     }
1143 }
1144
1145 #else
1146     ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
1147         "\"debug_connection\" is ignored, you need to rebuild \"
1148         \"nginx using --with-debug option to enable it");
1149 #endif
1150
1151 #endif
1152
1153     return NGX_CONF_OK;
1154 }
1155
1156
1157 static void *
1158 ngx_event_core_create_conf(ngx_cycle_t *cycle)
1159 {
1160     ngx_event_conf_t *ecf;

```

```

1161     ecf = ngx_palloc(cycle->pool, sizeof(ngx_event_conf_t));
1162     if (ecf == NULL) {
1163         return NULL;
1164     }
1165
1166     ecf->connections = NGX_CONF_UNSET_UINT;
1167     ecf->use = NGX_CONF_UNSET_UINT;
1168     ecf->multi_accept = NGX_CONF_UNSET;
1169     ecf->accept_mutex = NGX_CONF_UNSET;
1170     ecf->accept_mutex_delay = NGX_CONF_UNSET_MSEC;
1171     ecf->name = (void *) NGX_CONF_UNSET;
1172
1173     #if (NGX_DEBUG)
1174     if (ngx_array_init(&ecf->debug_connection, cycle->pool, 4,
1175         sizeof(ngx_cidr_t)) == NGX_ERROR)
1176     {
1177         return NULL;
1178     }
1179     #endif
1180
1181     return ecf;
1182 }
1183
1184 static char *
1185 ngx_event_core_init_conf(ngx_cycle_t *cycle, void *conf)
1186 {
1187     ngx_event_conf_t *ecf = conf;
1188
1189     #if (NGX_HAVE_EPOLL) && !(NGX_TEST_BUILD_EPOLL)
1190     int fd;
1191     #endif
1192
1193     #if (NGX_HAVE_RTSG)
1194     ngx_uint_t rtsig;
1195     ngx_core_conf_t *ccf;
1196     #endif
1197
1198     ngx_int_t i;
1199     ngx_module_t *module;
1200     ngx_event_module_t *event_module;
1201
1202     module = NULL;
1203
1204     #if (NGX_HAVE_EPOLL) && !(NGX_TEST_BUILD_EPOLL)
1205     fd = epoll_create(100);
1206
1207     if (fd != -1) {
1208         (void) close(fd);
1209         module = &ngx_epoll_module;
1210     } else if (ngx_errno != NGX_ENOSYS) {
1211         module = &ngx_epoll_module;
1212     }
1213     #endif
1214
1215     #if (NGX_HAVE_RTSG)
1216     if (module == NULL) {
1217         module = &ngx_rtsig_module;
1218         rtsig = 1;
1219     } else {
1220         rtsig = 0;
1221     }
1222     #endif
1223
1224     #if (NGX_HAVE_DEVPOLL)
1225     module = &ngx_devpoll_module;
1226     #endif
1227 }

```

```

1237
1238 #if (NGX_HAVE_KQUEUE)
1239
1240     module = &ngx_kqueue_module;
1241
1242 #endif
1243
1244 #if (NGX_HAVE_SELECT)
1245
1246     if (module == NULL) {
1247         module = &ngx_select_module;
1248     }
1249
1250 #endif
1251
1252     if (module == NULL) {
1253         for (i = 0; ngx_modules[i]; i++) {
1254
1255             if (ngx_modules[i]->type != NGX_EVENT_MODULE) {
1256                 continue;
1257             }
1258
1259             event_module = ngx_modules[i]->ctx;
1260
1261             if (ngx_strcmp(event_module->name->data, event_core_name.data) == 0)
1262             {
1263                 continue;
1264             }
1265
1266             module = ngx_modules[i];
1267             break;
1268         }
1269     }
1270
1271     if (module == NULL) {
1272         ngx_log_error(NGX_LOG_EMERG, cycle->log, 0, "no events module found");
1273         return NGX_CONF_ERROR;
1274     }
1275
1276     ngx_conf_init_uint_value(ecf->connections, DEFAULT_CONNECTIONS);
1277     cycle->connection_n = ecf->connections;
1278
1279     ngx_conf_init_uint_value(ecf->use, module->ctx_index);
1280
1281     event_module = module->ctx;
1282     ngx_conf_init_ptr_value(ecf->name, event_module->name->data);
1283
1284     ngx_conf_init_value(ecf->multi_accept, 0);
1285     ngx_conf_init_value(ecf->accept_mutex, 1);
1286     ngx_conf_init_msec_value(ecf->accept_mutex_delay, 500);
1287
1288
1289 #if (NGX_HAVE_RTSG)
1290
1291     if (!rtsig) {
1292         return NGX_CONF_OK;
1293     }
1294
1295     if (ecf->accept_mutex) {
1296         return NGX_CONF_OK;
1297     }
1298
1299     ccf = (ngx_core_conf_t *) ngx_get_conf(cycle->conf_ctx, ngx_core_module);
1300
1301     if (ccf->worker_processes == 0) {
1302         return NGX_CONF_OK;
1303     }
1304
1305     ngx_log_error(NGX_LOG_EMERG, cycle->log, 0,
1306                 "the \"rtsig\" method requires \"accept_mutex\" to be on");
1307
1308     return NGX_CONF_ERROR;
1309
1310 #else
1311
1312     return NGX_CONF_OK;

```

```
1313
1314 #endif
1315 }
```

[One Level Up](#)

[Top Level](#)

## src/event/modules/nginx\_eventport\_module.c - nginx-1.7.10

### Global variables defined

- [ep](#)
- [event\\_list](#)
- [event\\_timer](#)
- [eventport\\_name](#)
- [nevents](#)
- [ngx\\_eventport\\_commands](#)
- [ngx\\_eventport\\_module](#)
- [ngx\\_eventport\\_module\\_ctx](#)

### Data types defined

- [clockid\\_t](#)
- [itimerspec](#)
- [itimerspec\\_t](#)
- [ngx\\_eventport\\_conf\\_t](#)
- [port\\_event\\_t](#)
- [port\\_notify](#)
- [port\\_notify\\_t](#)
- [timer\\_t](#)

### Functions defined

- [ngx\\_eventport\\_add\\_event](#)
- [ngx\\_eventport\\_create\\_conf](#)
- [ngx\\_eventport\\_del\\_event](#)
- [ngx\\_eventport\\_done](#)
- [ngx\\_eventport\\_init](#)
- [ngx\\_eventport\\_init\\_conf](#)
- [ngx\\_eventport\\_process\\_events](#)
- [port\\_associate](#)
- [port\\_create](#)
- [port\\_dissociate](#)
- [port\\_getn](#)

- [timer\\_create](#)
- [timer\\_delete](#)
- [timer\\_settime](#)

## Macros defined

- [CLOCK\\_REALTIME](#)
- [ETIME](#)
- [PORT\\_SOURCE\\_AIO](#)
- [PORT\\_SOURCE\\_ALERT](#)
- [PORT\\_SOURCE\\_FD](#)
- [PORT\\_SOURCE\\_MQ](#)
- [PORT\\_SOURCE\\_TIMER](#)
- [PORT\\_SOURCE\\_USER](#)
- [SIGEV\\_PORT](#)
- [uint\\_t](#)
- [ushort\\_t](#)

## Source code

```

1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 #if (NGX_TEST_BUILD_EVENTPORT)
14
15 #define ushort_t  u_short
16 #define uint_t    u_int
17
18 #ifndef CLOCK_REALTIME
19 #define CLOCK_REALTIME 0
20 typedef int        clockid_t;
21 typedef void *    timer_t;
22 #endif
23
24 /* Solaris declarations */
25
26 #define PORT_SOURCE_AIO      1
27 #define PORT_SOURCE_TIMER   2
28 #define PORT_SOURCE_USER    3
29 #define PORT_SOURCE_FD      4
30 #define PORT_SOURCE_ALERT   5
31 #define PORT_SOURCE_MQ      6
32
33 #ifndef ETIME
34 #define ETIME 64
35 #endif
36
37 #define SIGEV_PORT 4

```

```

38
39 typedef struct {
40     int     portev_events; /* event data is source specific */
41     ushort_t portev_source; /* event source */
42     ushort_t portev_pad; /* port internal use */
43     uintptr_t portev_object; /* source specific object */
44     void     *portev_user; /* user cookie */
45 } port_event_t;
46
47 typedef struct port_notify {
48     int     portnfy_port; /* bind request(s) to port */
49     void     *portnfy_user; /* user defined */
50 } port_notify_t;
51
52 #if (__FreeBSD_version < 700005)
53
54 typedef struct itimerspec { /* definition per POSIX.4 */
55     struct timespec it_interval; /* timer period */
56     struct timespec it_value; /* timer expiration */
57 } itimerspec_t;
58
59 #endif
60
61 int port_create(void);
62
63 int port_create(void)
64 {
65     return -1;
66 }
67
68
69 int port_associate(int port, int source, uintptr_t object, int events,
70 void *user);
71
72 int port_associate(int port, int source, uintptr_t object, int events,
73 void *user)
74 {
75     return -1;
76 }
77
78
79 int port_dissociate(int port, int source, uintptr_t object);
80
81 int port_dissociate(int port, int source, uintptr_t object)
82 {
83     return -1;
84 }
85
86
87 int port_getn(int port, port_event_t list[], uint_t max, uint_t *nget,
88 struct timespec *timeout);
89
90 int port_getn(int port, port_event_t list[], uint_t max, uint_t *nget,
91 struct timespec *timeout)
92 {
93     return -1;
94 }
95
96
97 int timer_create(clockid_t clock_id, struct sigevent *evp, timer_t *timerid);
98
99 int timer_create(clockid_t clock_id, struct sigevent *evp, timer_t *timerid)
100 {
101     return -1;
102 }
103
104
105 int timer_settime(timer_t timerid, int flags, const struct itimerspec *value,
106 struct itimerspec *ovalue);
107
108 int timer_settime(timer_t timerid, int flags, const struct itimerspec *value,
109 struct itimerspec *ovalue)
110 {
111     return -1;
112 }
113

```



```

114 int timer_delete(timer_t timerid);
115
116 int timer_delete(timer_t timerid)
117 {
118     return -1;
119 }
120
121 #endif
122
123
124 typedef struct {
125     ngx_uint_t events;
126 } ngx_eventport_conf_t;
127
128
129 static ngx_int_t ngx_eventport_init(ngx_cycle_t *cycle, ngx_msec_t timer);
130 static void ngx_eventport_done(ngx_cycle_t *cycle);
131 static ngx_int_t ngx_eventport_add_event(ngx_event_t *ev, ngx_int_t event,
132     ngx_uint_t flags);
133 static ngx_int_t ngx_eventport_del_event(ngx_event_t *ev, ngx_int_t event,
134     ngx_uint_t flags);
135 static ngx_int_t ngx_eventport_process_events(ngx_cycle_t *cycle,
136     ngx_msec_t timer, ngx_uint_t flags);
137
138 static void *ngx_eventport_create_conf(ngx_cycle_t *cycle);
139 static char *ngx_eventport_init_conf(ngx_cycle_t *cycle, void *conf);
140
141 static int ep = -1;
142 static port_event_t *event_list;
143 static ngx_uint_t nevents;
144 static timer_t event_timer = (timer_t) -1;
145
146 static ngx_str_t eventport_name = ngx_string("eventport");
147
148 static ngx_command_t ngx_eventport_commands[] = {
149
150     { ngx_string("eventport_events"),
151       NGX_EVENT_CONF|NGX_CONF_TAKE1,
152       ngx_conf_set_num_slot,
153       0,
154       offsetof(ngx_eventport_conf_t, events),
155       NULL },
156
157     ngx_null_command
158
159 };
160
161
162 ngx_event_module_t ngx_eventport_module_ctx = {
163     &eventport_name,
164     ngx_eventport_create_conf,      /* create configuration */
165     ngx_eventport_init_conf,        /* init configuration */
166
167     {
168         ngx_eventport_add_event,    /* add an event */
169         ngx_eventport_del_event,    /* delete an event */
170         ngx_eventport_add_event,    /* enable an event */
171         ngx_eventport_del_event,    /* disable an event */
172         NULL,                       /* add an connection */
173         NULL,                       /* delete an connection */
174         NULL,                       /* process the changes */
175         ngx_eventport_process_events, /* process the events */
176         ngx_eventport_init,         /* init the events */
177         ngx_eventport_done,         /* done the events */
178     }
179 };
180
181
182 ngx_module_t ngx_eventport_module = {
183     NGX_MODULE_V1,
184     &ngx_eventport_module_ctx,     /* module context */
185     ngx_eventport_commands,        /* module directives */
186     NGX_EVENT_MODULE,             /* module type */
187     NULL,                         /* init master */
188     NULL,                         /* init module */
189

```

```

190     NULL,                                /* init process */
191     NULL,                                /* init thread */
192     NULL,                                /* exit thread */
193     NULL,                                /* exit process */
194     NULL,                                /* exit master */
195     NGX_MODULE_V1_PADDING
196 };
197
198
199 static ngx_int_t
200 ngx_eventport_init(ngx_cycle_t *cycle, ngx_msec_t timer)
201 {
202     port_notify_t          pn;
203     struct itimerspec      its;
204     struct sigevent       sev;
205     ngx_eventport_conf_t *epcf;
206
207     epcf = ngx_event_get_conf(cycle->conf_ctx, ngx_eventport_module);
208
209     if (ep == -1) {
210         ep = port_create();
211
212         if (ep == -1) {
213             ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
214                 "port_create() failed");
215             return NGX_ERROR;
216         }
217     }
218
219     if (nevents < epcf->events) {
220         if (event_list) {
221             ngx_free(event_list);
222         }
223
224         event_list = ngx_alloc(sizeof(port_event_t) * epcf->events,
225             cycle->log);
226         if (event_list == NULL) {
227             return NGX_ERROR;
228         }
229     }
230
231     ngx_event_flags = NGX_USE_EVENTPORT_EVENT;
232
233     if (timer) {
234         ngx_memzero(&pn, sizeof(port_notify_t));
235         pn.portnfy_port = ep;
236
237         ngx_memzero(&sev, sizeof(struct sigevent));
238         sev.sigev_notify = SIGEV_PORT;
239         #if !(NGX_TEST_BUILD_EVENTPORT)
240         sev.sigev_value.sival_ptr = &pn;
241         #endif
242
243         if (timer_create(CLOCK_REALTIME, &sev, &event_timer) == -1) {
244             ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
245                 "timer_create() failed");
246             return NGX_ERROR;
247         }
248
249         its.it_interval.tv_sec = timer / 1000;
250         its.it_interval.tv_nsec = (timer % 1000) * 1000000;
251         its.it_value.tv_sec = timer / 1000;
252         its.it_value.tv_nsec = (timer % 1000) * 1000000;
253
254         if (timer_settime(event_timer, 0, &its, NULL) == -1) {
255             ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
256                 "timer_settime() failed");
257             return NGX_ERROR;
258         }
259
260         ngx_event_flags |= NGX_USE_TIMER_EVENT;
261     }
262
263     nevents = epcf->events;
264
265     ngx_io = ngx_os_io;

```

```

266     ngx_event_actions = ngx_eventport_module_ctx.actions;
267
268
269     return NGX_OK;
270 }
271
272
273 static void
274 ngx_eventport_done(ngx_cycle_t *cycle)
275 {
276     if (event_timer != (timer_t) -1) {
277         if (timer_delete(event_timer) == -1) {
278             ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
279                 "timer_delete() failed");
280         }
281
282         event_timer = (timer_t) -1;
283     }
284
285     if (close(ep) == -1) {
286         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
287             "close() event port failed");
288     }
289
290     ep = -1;
291
292     ngx_free(event_list);
293
294     event_list = NULL;
295     nevents = 0;
296 }
297
298
299 static ngx_int_t
300 ngx_eventport_add_event(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags)
301 {
302     ngx_int_t      events, prev;
303     ngx_event_t    *e;
304     ngx_connection_t *c;
305
306     c = ev->data;
307
308     events = event;
309
310     if (event == NGX_READ_EVENT) {
311         e = c->write;
312         prev = POLLOUT;
313 #if (NGX_READ_EVENT != POLLIN)
314         events = POLLIN;
315 #endif
316
317     } else {
318         e = c->read;
319         prev = POLLIN;
320 #if (NGX_WRITE_EVENT != POLLOUT)
321         events = POLLOUT;
322 #endif
323     }
324
325     if (e->oneshot) {
326         events |= prev;
327     }
328
329     ngx_log_debug2(NGX_LOG_DEBUG_EVENT, ev->log, 0,
330         "eventport add event: fd:%d ev:%04Xi", c->fd, events);
331
332     if (port_associate(ep, PORT_SOURCE_FD, c->fd, events,
333         (void *) ((uintptr_t) ev | ev->instance))
334         == -1)
335     {
336         ngx_log_error(NGX_LOG_ALERT, ev->log, ngx_errno,
337             "port_associate() failed");
338         return NGX_ERROR;
339     }
340
341     ev->active = 1;

```

```

342     ev->oneshot = 1;
343
344     return NGX_OK;
345 }
346
347
348 static ngx_int_t
349 ngx_eventport_del_event(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags)
350 {
351     ngx_event_t      *e;
352     ngx_connection_t *c;
353
354     /*
355      * when the file descriptor is closed, the event port automatically
356      * dissociates it from the port, so we do not need to dissociate explicitly
357      * the event before the closing the file descriptor
358      */
359
360     if (flags & NGX_CLOSE_EVENT) {
361         ev->active = 0;
362         ev->oneshot = 0;
363         return NGX_OK;
364     }
365
366     c = ev->data;
367
368     if (event == NGX_READ_EVENT) {
369         e = c->write;
370         event = POLLOUT;
371     } else {
372         e = c->read;
373         event = POLLIN;
374     }
375
376     if (e->oneshot) {
377         ngx_log_debug2(NGX_LOG_DEBUG_EVENT, ev->log, 0,
378             "eventport change event: fd:%d ev:%04Xi", c->fd, event);
379
380         if (port_associate(ep, PORT_SOURCE_FD, c->fd, event,
381             (void *) ((u_intptr_t) ev | ev->instance))
382             == -1)
383         {
384             ngx_log_error(NGX_LOG_ALERT, ev->log, ngx_errno,
385                 "port_associate() failed");
386             return NGX_ERROR;
387         }
388     } else {
389         ngx_log_debug1(NGX_LOG_DEBUG_EVENT, ev->log, 0,
390             "eventport del event: fd:%d", c->fd);
391
392         if (port_dissociate(ep, PORT_SOURCE_FD, c->fd) == -1) {
393             ngx_log_error(NGX_LOG_ALERT, ev->log, ngx_errno,
394                 "port_dissociate() failed");
395             return NGX_ERROR;
396         }
397     }
398 }
399
400     ev->active = 0;
401     ev->oneshot = 0;
402
403     return NGX_OK;
404 }
405
406
407
408 ngx_int_t
409 ngx_eventport_process_events(ngx_cycle_t *cycle, ngx_msec_t timer,
410     ngx_uint_t flags)
411 {
412     int                n, revents;
413     u_int              events;
414     ngx_err_t        err;
415     ngx_int_t        instance;
416     ngx_uint_t        i, level;
417     ngx_event_t      *ev, *rev, *wev;

```

```

418     ngx_queue_t      *queue;
419     ngx_connection_t *c;
420     struct timespec  ts, *tp;
421
422     if (timer == NGX_TIMER_INFINITE) {
423         tp = NULL;
424
425     } else {
426         ts.tv_sec = timer / 1000;
427         ts.tv_nsec = (timer % 1000) * 1000000;
428         tp = &ts;
429     }
430
431     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
432                  "eventport timer: %M", timer);
433
434     events = 1;
435
436     n = port_getn(ep, event_list, (u_int) nevents, &events, tp);
437
438     err = ngx_errno;
439
440     if (flags & NGX_UPDATE_TIME) {
441         ngx_time_update();
442     }
443
444     if (n == -1) {
445         if (err == ETIME) {
446             if (timer != NGX_TIMER_INFINITE) {
447                 return NGX_OK;
448             }
449
450             ngx_log_error(NGX_LOG_ALERT, cycle->log, 0,
451                          "port_getn() returned no events without timeout");
452             return NGX_ERROR;
453         }
454
455         level = (err == NGX_EINTR) ? NGX_LOG_INFO : NGX_LOG_ALERT;
456         ngx_log_error(level, cycle->log, err, "port_getn() failed");
457         return NGX_ERROR;
458     }
459
460     if (events == 0) {
461         if (timer != NGX_TIMER_INFINITE) {
462             return NGX_OK;
463         }
464
465         ngx_log_error(NGX_LOG_ALERT, cycle->log, 0,
466                      "port_getn() returned no events without timeout");
467         return NGX_ERROR;
468     }
469
470     for (i = 0; i < events; i++) {
471
472         if (event_list[i].portev_source == PORT_SOURCE_TIMER) {
473             ngx_time_update();
474             continue;
475         }
476
477         ev = event_list[i].portev_user;
478
479         switch (event_list[i].portev_source) {
480
481         case PORT_SOURCE_FD:
482
483             instance = (uintptr_t) ev & 1;
484             ev = (ngx_event_t *) ((uintptr_t) ev & (uintptr_t) ~1);
485
486             if (ev->closed || ev->instance != instance) {
487
488                 /*
489                  * the stale event from a file descriptor
490                  * that was just closed in this iteration
491                  */
492
493                 ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,

```

```

494         "eventport: stale event %p", ev);
495     continue;
496 }
497
498 revents = event\_list[i].portev_events;
499
500 ngx\_log\_debug2(NGX\_LOG\_DEBUG\_EVENT, cycle->log, 0,
501     "eventport: fd:%d, ev:%04Xd",
502     event\_list[i].portev_object, revents);
503
504 if (revents & (POLLERR|POLLHUP|POLLNVAL)) {
505     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_EVENT, cycle->log, 0,
506         "port_getn() error fd:%d ev:%04Xd",
507         event\_list[i].portev_object, revents);
508 }
509
510 if (revents & ~(POLLIN|POLLOUT|POLLERR|POLLHUP|POLLNVAL)) {
511     ngx\_log\_error(NGX\_LOG\_ALERT, cycle->log, 0,
512         "strange port\_getn\(\) events fd:%d ev:%04Xd",
513         event\_list[i].portev_object, revents);
514 }
515
516 if ((revents & (POLLERR|POLLHUP|POLLNVAL))
517     && (revents & (POLLIN|POLLOUT)) == 0)
518 {
519     /*
520      * if the error events were returned without POLLIN or POLLOUT,
521      * then add these flags to handle the events at least in one
522      * active handler
523      */
524
525     revents |= POLLIN|POLLOUT;
526 }
527
528 c = ev->data;
529 rev = c->read;
530 wev = c->write;
531
532 rev->active = 0;
533 wev->active = 0;
534
535 if (revents & POLLIN) {
536     rev->ready = 1;
537
538     if (flags & NGX\_POST\_EVENTS) {
539         queue = rev->accept ? &ngx\_posted\_accept\_events
540             : &ngx\_posted\_events;
541
542         ngx\_post\_event(rev, queue);
543     } else {
544         rev->handler(rev);
545
546         if (ev->closed || ev->instance != instance) {
547             continue;
548         }
549     }
550 }
551
552 if (rev->accept) {
553     if (ngx\_use\_accept\_mutex) {
554         ngx\_accept\_events = 1;
555         continue;
556     }
557
558     if (port\_associate(ep, PORT\_SOURCE\_FD, c->fd, POLLIN,
559         (void *) ((uintptr_t) ev | ev->instance))
560         == -1)
561     {
562         ngx\_log\_error(NGX\_LOG\_ALERT, ev->log, ngx\_errno,
563             "port\_associate\(\) failed");
564         return NGX\_ERROR;
565     }
566 }
567 }
568
569 if (revents & POLLOUT) {

```

```

570         wev->ready = 1;
571
572         if (flags & NGX\_POST\_EVENTS) {
573             ngx\_post\_event(wev, &ngx\_posted\_events);
574
575         } else {
576             wev->handler(wev);
577         }
578     }
579
580     continue;
581
582     default:
583         ngx\_log\_error(NGX\_LOG\_ALERT, cycle->log, 0,
584             "unexpected even_port object %d",
585             event\_list[i].portev_object);
586         continue;
587     }
588 }
589
590 return NGX\_OK;
591 }
592
593
594 static void *
595 ngx\_eventport\_create\_conf(ngx\_cycle\_t *cycle)
596 {
597     ngx\_eventport\_conf\_t *epcf;
598
599     epcf = ngx\_palloc(cycle->pool, sizeof(ngx\_eventport\_conf\_t));
600     if (epcf == NULL) {
601         return NULL;
602     }
603
604     epcf->events = NGX\_CONF\_UNSET;
605
606     return epcf;
607 }
608
609
610 static char *
611 ngx\_eventport\_init\_conf(ngx\_cycle\_t *cycle, void *conf)
612 {
613     ngx\_eventport\_conf\_t *epcf = conf;
614
615     ngx\_conf\_init\_uint\_value(epcf->events, 32);
616
617     return NGX\_CONF\_OK;
618 }

```

[One Level Up](#)

[Top Level](#)

## src/os/unix/nginx\_time.h - nginx-1.7.10

### Data types defined

- [ngx\\_msec\\_int\\_t](#)
- [ngx\\_msec\\_t](#)
- [ngx\\_tm\\_t](#)

### Macros defined

- [\\_NGX\\_TIME\\_H\\_INCLUDED\\_](#)
- [ngx\\_gettimeofday](#)
- [ngx\\_msleep](#)
- [ngx\\_sleep](#)
- [ngx\\_timezone](#)
- [ngx\\_timezone](#)
- [ngx\\_tm\\_gmtoff](#)
- [ngx\\_tm\\_hour](#)
- [ngx\\_tm\\_hour\\_t](#)
- [ngx\\_tm\\_isdst](#)
- [ngx\\_tm\\_mday](#)
- [ngx\\_tm\\_mday\\_t](#)
- [ngx\\_tm\\_min](#)
- [ngx\\_tm\\_min\\_t](#)
- [ngx\\_tm\\_mon](#)
- [ngx\\_tm\\_mon\\_t](#)
- [ngx\\_tm\\_sec](#)
- [ngx\\_tm\\_sec\\_t](#)
- [ngx\\_tm\\_wday](#)
- [ngx\\_tm\\_wday\\_t](#)
- [ngx\\_tm\\_year](#)
- [ngx\\_tm\\_year\\_t](#)
- [ngx\\_tm\\_zone](#)

### Source code

```
1  
2 /*
```



```

3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #ifndef NGX_TIME_H_INCLUDED
9  #define NGX_TIME_H_INCLUDED
10
11
12  #include <ngx_config.h>
13  #include <ngx_core.h>
14
15
16  typedef ngx_rbtree_key_t      ngx_msec_t;
17  typedef ngx_rbtree_key_int_t ngx_msec_int_t;
18
19  typedef struct tm             ngx_tm_t;
20
21  #define ngx_tm_sec           tm_sec
22  #define ngx_tm_min           tm_min
23  #define ngx_tm_hour          tm_hour
24  #define ngx_tm_mday          tm_mday
25  #define ngx_tm_mon           tm_mon
26  #define ngx_tm_year          tm_year
27  #define ngx_tm_wday          tm_wday
28  #define ngx_tm_isdst         tm_isdst
29
30  #define ngx_tm_sec_t         int
31  #define ngx_tm_min_t         int
32  #define ngx_tm_hour_t        int
33  #define ngx_tm_mday_t        int
34  #define ngx_tm_mon_t         int
35  #define ngx_tm_year_t        int
36  #define ngx_tm_wday_t        int
37
38
39  #if (NGX_HAVE_GMTOFF)
40  #define ngx_tm_gmtoff        tm_gmtoff
41  #define ngx_tm_zone          tm_zone
42  #endif
43
44
45  #if (NGX_SOLARIS)
46
47  #define ngx_timezone(isdst) (- (isdst ? altzone : timezone) / 60)
48
49  #else
50
51  #define ngx_timezone(isdst) (- (isdst ? timezone + 3600 : timezone) / 60)
52
53  #endif
54
55
56  void ngx_timezone_update(void);
57  void ngx_localtime(time_t s, ngx_tm_t *tm);
58  void ngx_libc_localtime(time_t s, struct tm *tm);
59  void ngx_libc_gmtime(time_t s, struct tm *tm);
60
61  #define ngx_gettimeofday(tp) (void) gettimeofday(tp, NULL);
62  #define ngx_msleep(ms)       (void) usleep(ms * 1000)
63  #define ngx_sleep(s)         (void) sleep(s)
64
65
66  #endif /* NGX_TIME_H_INCLUDED */

```

[One Level Up](#)

[Top Level](#)

# src/core/nginx\_rbtrees.h - nginx-1.7.10

## Data types defined

- [ngx\\_rbtrees\\_insert\\_pt](#)
- [ngx\\_rbtrees\\_key\\_int\\_t](#)
- [ngx\\_rbtrees\\_key\\_t](#)
- [ngx\\_rbtrees\\_node\\_s](#)
- [ngx\\_rbtrees\\_node\\_t](#)
- [ngx\\_rbtrees\\_s](#)
- [ngx\\_rbtrees\\_t](#)

## Functions defined

- [ngx\\_rbtrees\\_min](#)

## Macros defined

- [\\_NGX\\_RBTREE\\_H\\_INCLUDED](#)
- [ngx\\_rbt\\_black](#)
- [ngx\\_rbt\\_copy\\_color](#)
- [ngx\\_rbt\\_is\\_black](#)
- [ngx\\_rbt\\_is\\_red](#)
- [ngx\\_rbt\\_red](#)
- [ngx\\_rbtrees\\_init](#)
- [ngx\\_rbtrees\\_sentinel\\_init](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_RBTREE_H_INCLUDED_
9 #define _NGX_RBTREE_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 typedef ngx_uint_t ngx_rbtrees_key_t;
17 typedef ngx_int_t ngx_rbtrees_key_int_t;
18
19
20 typedef struct ngx_rbtrees_node_s ngx_rbtrees_node_t;
21
```

```

22 struct ngx_rbtrees_node_s {
23     ngx_rbtrees_key_t    key;
24     ngx_rbtrees_node_t  *left;
25     ngx_rbtrees_node_t  *right;
26     ngx_rbtrees_node_t  *parent;
27     u_char               color;
28     u_char               data;
29 };
30
31
32 typedef struct ngx_rbtrees_s  ngx_rbtrees_t;
33
34 typedef void (*ngx_rbtrees_insert_pt) (ngx_rbtrees_node_t *root,
35     ngx_rbtrees_node_t *node, ngx_rbtrees_node_t *sentinel);
36
37 struct ngx_rbtrees_s {
38     ngx_rbtrees_node_t  *root;
39     ngx_rbtrees_node_t  *sentinel;
40     ngx_rbtrees_insert_pt  insert;
41 };
42
43
44 #define ngx_rbtrees_init(tree, s, i)           \
45     ngx_rbtrees_sentinel_init(s);           \
46     (tree)->root = s;                       \
47     (tree)->sentinel = s;                   \
48     (tree)->insert = i
49
50
51 void ngx_rbtrees_insert(ngx_rbtrees_t *tree, ngx_rbtrees_node_t *node);
52 void ngx_rbtrees_delete(ngx_rbtrees_t *tree, ngx_rbtrees_node_t *node);
53 void ngx_rbtrees_insert_value(ngx_rbtrees_node_t *root, ngx_rbtrees_node_t *node,
54     ngx_rbtrees_node_t *sentinel);
55 void ngx_rbtrees_insert_timer_value(ngx_rbtrees_node_t *root,
56     ngx_rbtrees_node_t *node, ngx_rbtrees_node_t *sentinel);
57
58
59 #define ngx_rbt_black(node)      ((node)->color = 1)
60 #define ngx_rbt_black(node)      ((node)->color = 0)
61 #define ngx_rbt_is_black(node)   ((node)->color)
62 #define ngx_rbt_is_black(node)   (!ngx_rbt_is_black(node))
63 #define ngx_rbt_copy_color(n1, n2) (n1->color = n2->color)
64
65
66 /* a sentinel must be black */
67
68 #define ngx_rbtrees_sentinel_init(node) ngx_rbt_black(node)
69
70
71 static ngx_inline ngx_rbtrees_node_t *
72 ngx_rbtrees_min(ngx_rbtrees_node_t *node, ngx_rbtrees_node_t *sentinel)
73 {
74     while (node->left != sentinel) {
75         node = node->left;
76     }
77
78     return node;
79 }
80
81
82 #endif /* NGX_RBTREE_H_INCLUDED */

```

[One Level Up](#)

[Top Level](#)

# src/core/nginx\_rbtrees.c - nginx-1.7.10

## Functions defined

- [ngx\\_rbtrees\\_delete](#)
- [ngx\\_rbtrees\\_insert](#)
- [ngx\\_rbtrees\\_insert\\_timer\\_value](#)
- [ngx\\_rbtrees\\_insert\\_value](#)
- [ngx\\_rbtrees\\_left\\_rotate](#)
- [ngx\\_rbtrees\\_right\\_rotate](#)

## Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10
11
12  /*
13  * The red-black tree code is based on the algorithm described in
14  * the "Introduction to Algorithms" by Cormen, Leiserson and Rivest.
15  */
16
17
18  static ngx_inline void ngx_rbtrees_left_rotate(ngx_rbtrees_node_t **root,
19  ngx_rbtrees_node_t *sentinel, ngx_rbtrees_node_t *node);
20  static ngx_inline void ngx_rbtrees_right_rotate(ngx_rbtrees_node_t **root,
21  ngx_rbtrees_node_t *sentinel, ngx_rbtrees_node_t *node);
22
23
24  void
25  ngx_rbtrees_insert(ngx_rbtrees_t *tree, ngx_rbtrees_node_t *node)
26  {
27  ngx_rbtrees_node_t **root, *temp, *sentinel;
28
29  /* a binary tree insert */
30
31  root = (ngx_rbtrees_node_t **) &tree->root;
32  sentinel = tree->sentinel;
33
34  if (*root == sentinel) {
35  node->parent = NULL;
36  node->left = sentinel;
37  node->right = sentinel;
38  ngx_rbt_black(node);
39  *root = node;
40
41  return;
42  }
43
44  tree->insert(*root, node, sentinel);
45
46  /* re-balance tree */
47
48  while (node != *root && ngx_rbt_is_red(node->parent)) {
49
50  if (node->parent == node->parent->parent->left) {
51  temp = node->parent->parent->right;
```

```

52     if (ngx_rbt_is_red(temp)) {
53         ngx_rbt_black(node->parent);
54         ngx_rbt_black(temp);
55         ngx_rbt_red(node->parent->parent);
56         node = node->parent->parent;
57     }
58     } else {
59         if (node == node->parent->right) {
60             node = node->parent;
61             ngx_rbtree_left_rotate(root, sentinel, node);
62         }
63     }
64     ngx_rbt_black(node->parent);
65     ngx_rbt_red(node->parent->parent);
66     ngx_rbtree_right_rotate(root, sentinel, node->parent->parent);
67 }
68 } else {
69     temp = node->parent->parent->left;
70     if (ngx_rbt_is_red(temp)) {
71         ngx_rbt_black(node->parent);
72         ngx_rbt_black(temp);
73         ngx_rbt_red(node->parent->parent);
74         node = node->parent->parent;
75     } else {
76         if (node == node->parent->left) {
77             node = node->parent;
78             ngx_rbtree_right_rotate(root, sentinel, node);
79         }
80         ngx_rbt_black(node->parent);
81         ngx_rbt_red(node->parent->parent);
82         ngx_rbtree_left_rotate(root, sentinel, node->parent->parent);
83     }
84 }
85 }
86 }
87 ngx_rbt_black(*root);
88 }
89
90 void
91 ngx_rbtree_insert_value(ngx_rbtree_node_t *temp, ngx_rbtree_node_t *node,
92     ngx_rbtree_node_t *sentinel)
93 {
94     ngx_rbtree_node_t **p;
95     for ( ;; ) {
96         p = (node->key < temp->key) ? &temp->left : &temp->right;
97         if (*p == sentinel) {
98             break;
99         }
100        temp = *p;
101    }
102    *p = node;
103    node->parent = temp;
104    node->left = sentinel;
105    node->right = sentinel;
106    ngx_rbt_red(node);
107 }
108
109 void
110 ngx_rbtree_insert_timer_value(ngx_rbtree_node_t *temp, ngx_rbtree_node_t *node,
111     ngx_rbtree_node_t *sentinel)
112 {
113     ngx_rbtree_node_t **p;
114     for ( ;; ) {

```

```

128
129
130     /*
131     * Timer values
132     * 1) are spread in small range, usually several minutes,
133     * 2) and overflow each 49 days, if milliseconds are stored in 32 bits.
134     * The comparison takes into account that overflow.
135     */
136     /* node->key < temp->key */
137
138     p = ((ngx_rbtree_key_int_t) (node->key - temp->key) < 0)
139         ? &temp->left : &temp->right;
140
141     if (*p == sentinel) {
142         break;
143     }
144
145     temp = *p;
146 }
147
148 *p = node;
149 node->parent = temp;
150 node->left = sentinel;
151 node->right = sentinel;
152 ngx_rbt_red(node);
153 }
154
155
156 void
157 ngx_rbtree_delete(ngx_rbtree_t *tree, ngx_rbtree_node_t *node)
158 {
159     ngx_uint_t         red;
160     ngx_rbtree_node_t **root, *sentinel, *subst, *temp, *w;
161
162     /* a binary tree delete */
163
164     root = (ngx_rbtree_node_t **) &tree->root;
165     sentinel = tree->sentinel;
166
167     if (node->left == sentinel) {
168         temp = node->right;
169         subst = node;
170
171     } else if (node->right == sentinel) {
172         temp = node->left;
173         subst = node;
174
175     } else {
176         subst = ngx_rbtree_min(node->right, sentinel);
177
178         if (subst->left != sentinel) {
179             temp = subst->left;
180         } else {
181             temp = subst->right;
182         }
183     }
184
185     if (subst == *root) {
186         *root = temp;
187         ngx_rbt_black(temp);
188
189         /* DEBUG stuff */
190         node->left = NULL;
191         node->right = NULL;
192         node->parent = NULL;
193         node->key = 0;
194
195         return;
196     }
197
198     red = ngx_rbt_is_red(subst);
199
200     if (subst == subst->parent->left) {
201         subst->parent->left = temp;
202
203     } else {

```

```

204     subst->parent->right = temp;
205 }
206
207 if (subst == node) {
208     temp->parent = subst->parent;
209
210 } else {
211
212     if (subst->parent == node) {
213         temp->parent = subst;
214
215     } else {
216         temp->parent = subst->parent;
217     }
218
219     subst->left = node->left;
220     subst->right = node->right;
221     subst->parent = node->parent;
222     ngx_rbt_copy_color(subst, node);
223
224     if (node == *root) {
225         *root = subst;
226
227     } else {
228         if (node == node->parent->left) {
229             node->parent->left = subst;
230         } else {
231             node->parent->right = subst;
232         }
233     }
234
235     if (subst->left != sentinel) {
236         subst->left->parent = subst;
237     }
238
239     if (subst->right != sentinel) {
240         subst->right->parent = subst;
241     }
242 }
243
244 /* DEBUG stuff */
245 node->left = NULL;
246 node->right = NULL;
247 node->parent = NULL;
248 node->key = 0;
249
250 if (red) {
251     return;
252 }
253
254 /* a delete fixup */
255
256 while (temp != *root && ngx_rbt_is_black(temp)) {
257
258     if (temp == temp->parent->left) {
259         w = temp->parent->right;
260
261         if (ngx_rbt_is_red(w)) {
262             ngx_rbt_black(w);
263             ngx_rbt_red(temp->parent);
264             ngx_rbtree_left_rotate(root, sentinel, temp->parent);
265             w = temp->parent->right;
266         }
267
268         if (ngx_rbt_is_black(w->left) && ngx_rbt_is_black(w->right)) {
269             ngx_rbt_red(w);
270             temp = temp->parent;
271
272         } else {
273             if (ngx_rbt_is_black(w->right)) {
274                 ngx_rbt_black(w->left);
275                 ngx_rbt_red(w);
276                 ngx_rbtree_right_rotate(root, sentinel, w);
277                 w = temp->parent->right;
278             }
279

```

```

280         ngx_rbt_copy_color(w, temp->parent);
281         ngx_rbt_black(temp->parent);
282         ngx_rbt_black(w->right);
283         ngx_rbtree_left_rotate(root, sentinel, temp->parent);
284         temp = *root;
285     }
286 }
287
288 } else {
289     w = temp->parent->left;
290
291     if (ngx_rbt_is_red(w)) {
292         ngx_rbt_black(w);
293         ngx_rbt_red(temp->parent);
294         ngx_rbtree_right_rotate(root, sentinel, temp->parent);
295         w = temp->parent->left;
296     }
297
298     if (ngx_rbt_is_black(w->left) && ngx_rbt_is_black(w->right)) {
299         ngx_rbt_red(w);
300         temp = temp->parent;
301     }
302     } else {
303         if (ngx_rbt_is_black(w->left)) {
304             ngx_rbt_black(w->right);
305             ngx_rbt_red(w);
306             ngx_rbtree_left_rotate(root, sentinel, w);
307             w = temp->parent->left;
308         }
309
310         ngx_rbt_copy_color(w, temp->parent);
311         ngx_rbt_black(temp->parent);
312         ngx_rbt_black(w->left);
313         ngx_rbtree_right_rotate(root, sentinel, temp->parent);
314         temp = *root;
315     }
316 }
317 }
318
319 ngx_rbt_black(temp);
320 }
321
322
323 static ngx_inline void
324 ngx_rbtree_left_rotate(ngx_rbtree_node_t **root, ngx_rbtree_node_t *sentinel,
325 ngx_rbtree_node_t *node)
326 {
327     ngx_rbtree_node_t *temp;
328
329     temp = node->right;
330     node->right = temp->left;
331
332     if (temp->left != sentinel) {
333         temp->left->parent = node;
334     }
335
336     temp->parent = node->parent;
337
338     if (node == *root) {
339         *root = temp;
340     }
341     } else if (node == node->parent->left) {
342         node->parent->left = temp;
343     }
344     } else {
345         node->parent->right = temp;
346     }
347
348     temp->left = node;
349     node->parent = temp;
350 }
351
352
353 static ngx_inline void
354 ngx_rbtree_right_rotate(ngx_rbtree_node_t **root, ngx_rbtree_node_t *sentinel,
355 ngx_rbtree_node_t *node)

```



```
356 {
357     ngx_rbtree_node_t *temp;
358
359     temp = node->left;
360     node->left = temp->right;
361
362     if (temp->right != sentinel) {
363         temp->right->parent = node;
364     }
365
366     temp->parent = node->parent;
367
368     if (node == *root) {
369         *root = temp;
370
371     } else if (node == node->parent->right) {
372         node->parent->right = temp;
373
374     } else {
375         node->parent->left = temp;
376     }
377
378     temp->right = node;
379     node->parent = temp;
380 }
```

[One Level Up](#)

[Top Level](#)

## src/event/modules/nginx\_rtsig\_module.c - nginx-1.7.10

### Global variables defined

- [ngx\\_linux\\_rtsig\\_max](#)
- [ngx\\_overflow\\_threshold\\_bounds](#)
- [ngx\\_rtsig\\_commands](#)
- [ngx\\_rtsig\\_module](#)
- [ngx\\_rtsig\\_module\\_ctx](#)
- [overflow](#)
- [overflow\\_current](#)
- [overflow\\_list](#)
- [rtsig\\_name](#)
- [set](#)

### Data types defined

- [ngx\\_rtsig\\_conf\\_t](#)

### Functions defined

- [ngx\\_check\\_ngx\\_overflow\\_threshold\\_bounds](#)
- [ngx\\_rtsig\\_add\\_connection](#)
- [ngx\\_rtsig\\_create\\_conf](#)
- [ngx\\_rtsig\\_del\\_connection](#)
- [ngx\\_rtsig\\_done](#)
- [ngx\\_rtsig\\_init](#)
- [ngx\\_rtsig\\_init\\_conf](#)
- [ngx\\_rtsig\\_process\\_events](#)
- [ngx\\_rtsig\\_process\\_overflow](#)
- [sigtimedwait](#)

### Macros defined

- [F\\_SETSIG](#)
- [KERN\\_RTSMAX](#)
- [KERN\\_RTSMNR](#)
- [SIGRTMIN](#)

- [SIGRTMIN](#)
- [si\\_fd](#)
- [si\\_fd](#)
- [si\\_fd](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 #if (NGX_TEST_BUILD_RTSIG)
14
15 #if (NGX_DARWIN)
16
17 #define SIGRTMIN      33
18 #define si_fd         __pad[0]
19
20 #else
21
22 #ifdef SIGRTMIN
23 #define si_fd         _reason.__spare__.__spare2__[0]
24 #else
25 #define SIGRTMIN      33
26 #define si_fd         __spare__[0]
27 #endif
28
29 #endif
30
31 #define F_SETSIG      10
32 #define KERN_RTSIGNR  30
33 #define KERN_RTSIGMAX 31
34
35 int sigtimedwait(const sigset_t *set, siginfo_t *info,
36                 const struct timespec *timeout);
37
38 int sigtimedwait(const sigset_t *set, siginfo_t *info,
39                 const struct timespec *timeout)
40 {
41     return -1;
42 }
43
44 int ngx_linux_rtsig_max;
45
46 #endif
47
48
49 typedef struct {
50     ngx_uint_t  signo;
51     ngx_uint_t  overflow_events;
52     ngx_uint_t  overflow_test;
53     ngx_uint_t  overflow_threshold;
54 } ngx_rtsig_conf_t;
55
56
57 extern ngx_event_module_t  ngx_poll_module_ctx;
58
59 static ngx_int_t ngx_rtsig_init(ngx_cycle_t *cycle, ngx_msec_t timer);
60 static void ngx_rtsig_done(ngx_cycle_t *cycle);
61 static ngx_int_t ngx_rtsig_add_connection(ngx_connection_t *c);
62 static ngx_int_t ngx_rtsig_del_connection(ngx_connection_t *c,
63     ngx_uint_t flags);

```

```

64 static ngx_int_t ngx_rtsig_process_events(ngx_cycle_t *cycle,
65     ngx_msec_t timer, ngx_uint_t flags);
66 static ngx_int_t ngx_rtsig_process_overflow(ngx_cycle_t *cycle,
67     ngx_msec_t timer, ngx_uint_t flags);
68
69 static void *ngx_rtsig_create_conf(ngx_cycle_t *cycle);
70 static char *ngx_rtsig_init_conf(ngx_cycle_t *cycle, void *conf);
71 static char *ngx_check ngx_overflow_threshold_bounds(ngx_conf_t *cf,
72     void *post, void *data);
73
74
75 static sigset_t      set;
76 static ngx_uint_t   overflow, overflow_current;
77 static struct pollfd *overflow_list;
78
79
80 static ngx_str_t     rtsig_name = ngx_string("rtsig");
81
82 static ngx_conf_num_bounds_t ngx_overflow_threshold_bounds = {
83     ngx_check ngx_overflow_threshold_bounds, 2, 10
84 };
85
86
87 static ngx_command_t ngx_rtsig_commands[] = {
88
89     { ngx_string("rtsig_signo"),
90       NGX_EVENT_CONF|NGX_CONF_TAKE1,
91       ngx_conf_set_num_slot,
92       0,
93       offsetof(ngx_rtsig_conf_t, signo),
94       NULL },
95
96     { ngx_string("rtsig_overflow_events"),
97       NGX_EVENT_CONF|NGX_CONF_TAKE1,
98       ngx_conf_set_num_slot,
99       0,
100      offsetof(ngx_rtsig_conf_t, overflow_events),
101      NULL },
102
103     { ngx_string("rtsig_overflow_test"),
104       NGX_EVENT_CONF|NGX_CONF_TAKE1,
105       ngx_conf_set_num_slot,
106       0,
107       offsetof(ngx_rtsig_conf_t, overflow_test),
108       NULL },
109
110     { ngx_string("rtsig_overflow_threshold"),
111       NGX_EVENT_CONF|NGX_CONF_TAKE1,
112       ngx_conf_set_num_slot,
113       0,
114       offsetof(ngx_rtsig_conf_t, overflow_threshold),
115       &ngx_overflow_threshold_bounds },
116
117     ngx_null_command
118 };
119
120
121 ngx_event_module_t ngx_rtsig_module_ctx = {
122     &rtsig_name,
123     ngx_rtsig_create_conf,          /* create configuration */
124     ngx_rtsig_init_conf,           /* init configuration */
125
126     {
127         NULL,                       /* add an event */
128         NULL,                       /* delete an event */
129         NULL,                       /* enable an event */
130         NULL,                       /* disable an event */
131         ngx_rtsig_add_connection,   /* add an connection */
132         ngx_rtsig_del_connection,   /* delete an connection */
133         NULL,                       /* process the changes */
134         ngx_rtsig_process_events,   /* process the events */
135         ngx_rtsig_init,             /* init the events */
136         ngx_rtsig_done,             /* done the events */
137     }
138
139 };

```

```

140 ngx_module_t ngx_rtsig_module = {
141     NGX_MODULE_V1,
142     &ngx_rtsig_module_ctx,           /* module context */
143     ngx_rtsig_commands,           /* module directives */
144     NGX_EVENT_MODULE,             /* module type */
145     NULL,                            /* init master */
146     NULL,                            /* init module */
147     NULL,                            /* init process */
148     NULL,                            /* init thread */
149     NULL,                            /* exit thread */
150     NULL,                            /* exit process */
151     NULL,                            /* exit master */
152     NGX_MODULE_V1_PADDING
153 };
154
155
156 static ngx_int_t
157 ngx_rtsig_init(ngx_cycle_t *cycle, ngx_msec_t timer)
158 {
159     ngx_rtsig_conf_t *rtscf;
160
161     rtscf = ngx_event_get_conf(cycle->conf_ctx, ngx_rtsig_module);
162
163     sigemptyset(&set);
164     sigaddset(&set, (int) rtscf->signo);
165     sigaddset(&set, (int) rtscf->signo + 1);
166     sigaddset(&set, SIGIO);
167     sigaddset(&set, SIGALRM);
168
169     if (sigprocmask(SIG_BLOCK, &set, NULL) == -1) {
170         ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
171             "sigprocmask() failed");
172     }
173     return NGX_ERROR;
174 }
175
176 if (overflow_list) {
177     ngx_free(overflow_list);
178 }
179
180 overflow_list = ngx_alloc(sizeof(struct pollfd) * rtscf->overflow_events,
181     cycle->log);
182 if (overflow_list == NULL) {
183     return NGX_ERROR;
184 }
185
186 ngx_io = ngx_os_io;
187
188 ngx_event_actions = ngx_rtsig_module_ctx.actions;
189
190 ngx_event_flags = NGX_USE_RTSIG_EVENT
191     | NGX_USE_GREEDY_EVENT
192     | NGX_USE_FD_EVENT;
193
194 return NGX_OK;
195 }
196
197
198 static void
199 ngx_rtsig_done(ngx_cycle_t *cycle)
200 {
201     ngx_free(overflow_list);
202
203     overflow_list = NULL;
204 }
205
206
207 static ngx_int_t
208 ngx_rtsig_add_connection(ngx_connection_t *c)
209 {
210     ngx_uint_t signo;
211     ngx_rtsig_conf_t *rtscf;
212
213     if (c->read->accept && c->read->disabled) {
214
215         ngx_log_debug1(NGX_LOG_DEBUG_EVENT, c->log, 0,

```

```

216         "rtsig enable connection: fd:%d", c->fd);
217
218     if (fcntl(c->fd, F_SETOWN, ngx_pid) == -1) {
219         ngx_log_error(NGX_LOG_ALERT, c->log, ngx_errno,
220             "fcntl(F_SETOWN) failed");
221         return NGX_ERROR;
222     }
223
224     c->read->active = 1;
225     c->read->disabled = 0;
226 }
227
228 rtscf = ngx_event_get_conf(ngx_cycle->conf_ctx, ngx_rtsig_module);
229
230 signo = rtscf->signo + c->read->instance;
231
232 ngx_log_debug2(NGX_LOG_DEBUG_EVENT, c->log, 0,
233     "rtsig add connection: fd:%d signo:%ui", c->fd, signo);
234
235 if (fcntl(c->fd, F_SETFL, O_RDWR|O_NONBLOCK|O_ASYNC) == -1) {
236     ngx_log_error(NGX_LOG_ALERT, c->log, ngx_errno,
237         "fcntl(O_RDWR|O_NONBLOCK|O_ASYNC) failed");
238     return NGX_ERROR;
239 }
240
241 if (fcntl(c->fd, F_SETSIG, (int) signo) == -1) {
242     ngx_log_error(NGX_LOG_ALERT, c->log, ngx_errno,
243         "fcntl(F_SETSIG) failed");
244     return NGX_ERROR;
245 }
246
247 if (fcntl(c->fd, F_SETOWN, ngx_pid) == -1) {
248     ngx_log_error(NGX_LOG_ALERT, c->log, ngx_errno,
249         "fcntl(F_SETOWN) failed");
250     return NGX_ERROR;
251 }
252
253 #if (NGX_HAVE_ONESIGFD)
254 if (fcntl(c->fd, F_SETAUXFL, O_ONESIGFD) == -1) {
255     ngx_log_error(NGX_LOG_ALERT, c->log, ngx_errno,
256         "fcntl(F_SETAUXFL) failed");
257     return NGX_ERROR;
258 }
259 #endif
260
261 c->read->active = 1;
262 c->write->active = 1;
263
264 return NGX_OK;
265 }
266
267
268 static ngx_int_t
269 ngx_rtsig_del_connection(ngx_connection_t *c, ngx_uint_t flags)
270 {
271     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, c->log, 0,
272         "rtsig del connection: fd:%d", c->fd);
273
274     if ((flags & NGX_DISABLE_EVENT) && c->read->accept) {
275
276         ngx_log_debug1(NGX_LOG_DEBUG_EVENT, c->log, 0,
277             "rtsig disable connection: fd:%d", c->fd);
278
279         c->read->active = 0;
280         c->read->disabled = 1;
281         return NGX_OK;
282     }
283
284     if (flags & NGX_CLOSE_EVENT) {
285         c->read->active = 0;
286         c->write->active = 0;
287         return NGX_OK;
288     }
289
290     if (fcntl(c->fd, F_SETFL, O_RDWR|O_NONBLOCK) == -1) {
291         ngx_log_error(NGX_LOG_ALERT, c->log, ngx_errno,

```

```

292         "fcntl(O_RDWR|O_NONBLOCK) failed");
293     return NGX\_ERROR;
294 }
295
296 c->read->active = 0;
297 c->write->active = 0;
298
299     return NGX\_OK;
300 }
301
302
303 static ngx\_int\_t
304 ngx\_rtsig\_process\_events(ngx\_cycle\_t *cycle, ngx\_msec\_t timer, ngx\_uint\_t flags)
305 {
306     int                signo;
307     ngx\_int\_t         instance;
308     ngx\_err\_t         err;
309     siginfo\_t         si;
310     ngx\_event\_t      *rev, *wev;
311     ngx\_queue\_t      *queue;
312     struct timespec   ts, *tp;
313     struct sigaction  sa;
314     ngx\_connection\_t *c;
315     ngx\_rtsig\_conf\_t *rtscf;
316
317     if (timer == NGX\_TIMER\_INFINITE) {
318         tp = NULL;
319     }
320     else {
321         ts.tv_sec = timer / 1000;
322         ts.tv_nsec = (timer % 1000) * 1000000;
323         tp = &ts;
324     }
325
326     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_EVENT, cycle->log, 0,
327                 "rtsig timer: %M", timer);
328
329     /* Linux's sigwaitinfo() is sigtimedwait() with the NULL timeout pointer */
330
331     signo = sigtimedwait(&set, &si, tp);
332
333     if (signo == -1) {
334         err = ngx\_errno;
335
336         ngx\_log\_debug1(NGX\_LOG\_DEBUG\_EVENT, cycle->log, err,
337                     "rtsig signo:%d", signo);
338
339         if (flags & NGX\_UPDATE\_TIME) {
340             ngx\_time\_update();
341         }
342
343         if (err == NGX\_EAGAIN) {
344
345             /* timeout */
346
347             if (timer != NGX\_TIMER\_INFINITE) {
348                 return NGX\_AGAIN;
349             }
350
351             ngx\_log\_error(NGX\_LOG\_ALERT, cycle->log, err,
352                         "sigtimedwait() returned EAGAIN without timeout");
353             return NGX\_ERROR;
354         }
355
356         ngx\_log\_error((err == NGX\_EINTR) ? NGX\_LOG\_INFO : NGX\_LOG\_ALERT,
357                     cycle->log, err, "sigtimedwait() failed");
358         return NGX\_ERROR;
359     }
360
361     ngx\_log\_debug3(NGX\_LOG\_DEBUG\_EVENT, cycle->log, 0,
362                 "rtsig signo:%d fd:%d band:%04Xd",
363                 signo, si.si_fd, si.si_band);
364
365     if (flags & NGX\_UPDATE\_TIME) {
366         ngx\_time\_update();
367     }

```

```

368 rtscf = ngx_event_get_conf(ngx_cycle->conf_ctx, ngx_rtsig_module);
369
370
371 if (signo == (int) rtscf->signo || signo == (int) rtscf->signo + 1) {
372
373     if (overflow && (ngx_uint_t) si.si_fd > overflow_current) {
374         return NGX_OK;
375     }
376
377     c = ngx_cycle->files[si.si_fd];
378
379     if (c == NULL) {
380
381         /* the stale event */
382
383         return NGX_OK;
384     }
385
386     instance = signo - (int) rtscf->signo;
387
388     rev = c->read;
389
390     if (rev->instance != instance) {
391
392         /*
393          * the stale event from a file descriptor
394          * that was just closed in this iteration
395          */
396
397         ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
398             "rtsig: stale event %p", c);
399
400         return NGX_OK;
401     }
402
403     if ((si.si_band & (POLLIN|POLLHUP|POLLERR)) && rev->active) {
404
405         rev->ready = 1;
406
407         if (flags & NGX_POST_EVENTS) {
408             queue = rev->accept ? &ngx_posted_accept_events
409                 : &ngx_posted_events;
410
411             ngx_post_event(rev, queue);
412
413         } else {
414             rev->handler(rev);
415         }
416     }
417
418     wev = c->write;
419
420     if ((si.si_band & (POLLOUT|POLLHUP|POLLERR)) && wev->active) {
421
422         wev->ready = 1;
423
424         if (flags & NGX_POST_EVENTS) {
425             ngx_post_event(wev, &ngx_posted_events);
426
427         } else {
428             wev->handler(wev);
429         }
430     }
431
432     return NGX_OK;
433
434 } else if (signo == SIGALRM) {
435
436     ngx_time_update();
437
438     return NGX_OK;
439
440 } else if (signo == SIGIO) {
441
442     ngx_log_error(NGX_LOG_ALERT, cycle->log, 0,
443         "rt signal queue overflowed");

```



```

444     /* flush the RT signal queue */
445
446     ngx_memzero(&sa, sizeof(struct sigaction));
447     sa.sa_handler = SIG_DFL;
448     sigemptyset(&sa.sa_mask);
449
450     if (sigaction(rtscf->signo, &sa, NULL) == -1) {
451         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
452             "sigaction(%d, SIG_DFL) failed", rtscf->signo);
453     }
454
455     if (sigaction(rtscf->signo + 1, &sa, NULL) == -1) {
456         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
457             "sigaction(%d, SIG_DFL) failed", rtscf->signo + 1);
458     }
459
460     overflow = 1;
461     overflow_current = 0;
462     ngx_event_actions.process_events = ngx_rtsig_process_overflow;
463
464     return NGX_ERROR;
465 }
466
467 ngx_log_error(NGX_LOG_ALERT, cycle->log, 0,
468     "sigtimedwait() returned unexpected signal: %d", signo);
469
470 return NGX_ERROR;
471 }
472
473 static ngx_int_t
474 ngx_rtsig_process_overflow(ngx_cycle_t *cycle, ngx_msec_t timer,
475     ngx_uint_t flags)
476 {
477     int             name[2], rtsig_max, rtsig_nr, events, ready;
478     size_t         len;
479     ngx_err_t      err;
480     ngx_uint_t     tested, n, i;
481     ngx_event_t    *rev, *wev;
482     ngx_queue_t    *queue;
483     ngx_connection_t *c;
484     ngx_rtsig_conf_t *rtscf;
485
486     ngx_log_debug0(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
487         "rtsig process overflow");
488
489     rtscf = ngx_event_get_conf(ngx_cycle->conf_ctx, ngx_rtsig_module);
490
491     tested = 0;
492
493     for ( ;; ) {
494         n = 0;
495         while (n < rtscf->overflow_events) {
496             if (overflow_current == cycle->connection_n) {
497                 break;
498             }
499             c = cycle->files[overflow_current++];
500
501             if (c == NULL || c->fd == -1) {
502                 continue;
503             }
504
505             events = 0;
506
507             if (c->read->active && c->read->handler) {
508                 events |= POLLIN;
509             }
510
511             if (c->write->active && c->write->handler) {
512                 events |= POLLOUT;
513             }
514         }
515     }

```

```

520     if (events == 0) {
521         continue;
522     }
523
524
525     overflow\_list[n].fd = c->fd;
526     overflow\_list[n].events = events;
527     overflow\_list[n].revents = 0;
528     n++;
529 }
530
531 if (n == 0) {
532     break;
533 }
534
535 for ( ;; ) {
536     ready = poll(overflow\_list, n, 0);
537
538     ngx\_log\_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
539         "rtsig overflow poll:%d", ready);
540
541     if (ready == -1) {
542         err = ngx\_errno;
543         ngx\_log\_error((err == NGX_EINTR) ? NGX_LOG_INFO : NGX_LOG_ALERT,
544             cycle->log, 0,
545             "poll() failed while the overflow recover");
546
547         if (err == NGX_EINTR) {
548             continue;
549         }
550     }
551
552     break;
553 }
554
555 if (ready <= 0) {
556     continue;
557 }
558
559 for (i = 0; i < n; i++) {
560     c = cycle->files[overflow\_list[i].fd];
561
562     if (c == NULL) {
563         continue;
564     }
565
566     rev = c->read;
567
568     if (rev->active
569         && !rev->closed
570         && rev->handler
571         && (overflow\_list[i].revents
572             & (POLLIN|POLLERR|POLLHUP|POLLNVAL)))
573     {
574         tested++;
575
576         rev->ready = 1;
577
578         if (flags & NGX\_POST\_EVENTS) {
579             queue = rev->accept ? &ngx\_posted\_accept\_events
580                 : &ngx\_posted\_events;
581
582             ngx\_post\_event(rev, queue);
583
584         } else {
585             rev->handler(rev);
586         }
587     }
588
589     wev = c->write;
590
591     if (wev->active
592         && !wev->closed
593         && wev->handler
594         && (overflow\_list[i].revents
595             & (POLLOUT|POLLERR|POLLHUP|POLLNVAL)))

```

```

596     {
597         tested++;
598
599         wev->ready = 1;
600
601         if (flags & NGX\_POST\_EVENTS) {
602             ngx\_post\_event(wev, &ngx\_posted\_events);
603
604         } else {
605             wev->handler(wev);
606         }
607     }
608 }
609
610 if (tested >= rtscf->overflow_test) {
611
612     if (ngx\_linux\_rtsig\_max) {
613
614         /*
615          * Check the current rt queue length to prevent
616          * the new overflow.
617          *
618          * learn the "/proc/sys/kernel/rtsig-max" value because
619          * it can be changed since the last checking
620          */
621
622         name[0] = CTL_KERN;
623         name[1] = KERN\_RTSIGMAX;
624         len = sizeof(rtsig_max);
625
626         if (sysctl(name, 2, &rtsig_max, &len, NULL, 0) == -1) {
627             ngx\_log\_error(NGX\_LOG\_ALERT, cycle->log, errno,
628                 "sysctl(KERN\_RTSIGMAX) failed");
629             return NGX\_ERROR;
630         }
631
632         /* name[0] = CTL_KERN; */
633         name[1] = KERN\_RTSIGNR;
634         len = sizeof(rtsig_nr);
635
636         if (sysctl(name, 2, &rtsig_nr, &len, NULL, 0) == -1) {
637             ngx\_log\_error(NGX\_LOG\_ALERT, cycle->log, errno,
638                 "sysctl(KERN\_RTSIGNR) failed");
639             return NGX\_ERROR;
640         }
641
642         /*
643          * drain the rt signal queue if the "/proc/sys/kernel/rtsig-nr"
644          * is bigger than
645          * "/proc/sys/kernel/rtsig-max" / "rtsig\_overflow\_threshold"
646          */
647
648         if (rtsig_max / (int) rtscf->overflow_threshold < rtsig_nr) {
649             ngx\_log\_debug2(NGX\_LOG\_DEBUG\_EVENT, cycle->log, 0,
650                 "rtsig queue state: %d/%d",
651                 rtsig_nr, rtsig_max);
652             while (ngx\_rtsig\_process\_events(cycle, 0, flags) == NGX\_OK)
653                 {
654                     /* void */
655                 }
656         }
657     } else {
658
659         /*
660          * Linux has not KERN\_RTSIGMAX since 2.6.6-mm2
661          * so drain the rt signal queue unconditionally
662          */
663
664         while (ngx\_rtsig\_process\_events(cycle, 0, flags) == NGX\_OK) {
665             /* void */
666         }
667     }
668 }
669
670 tested = 0;
671 }

```

```

672     }
673
674     if (flags & NGX\_UPDATE\_TIME) {
675         ngx\_time\_update();
676     }
677
678     ngx\_log\_error(NGX\_LOG\_ALERT, cycle->log, 0,
679         "rt signal queue overflow recovered");
680
681     overflow = 0;
682     ngx\_event\_actions.process_events = ngx\_rtsig\_process\_events;
683
684     return NGX\_OK;
685 }
686
687
688 static void *
689 ngx\_rtsig\_create\_conf(ngx\_cycle\_t *cycle)
690 {
691     ngx\_rtsig\_conf\_t *rtscf;
692
693     rtscf = ngx\_palloc(cycle->pool, sizeof(ngx\_rtsig\_conf\_t));
694     if (rtscf == NULL) {
695         return NULL;
696     }
697
698     rtscf->signo = NGX\_CONF\_UNSET;
699     rtscf->overflow_events = NGX\_CONF\_UNSET;
700     rtscf->overflow_test = NGX\_CONF\_UNSET;
701     rtscf->overflow_threshold = NGX\_CONF\_UNSET;
702
703     return rtscf;
704 }
705
706
707 static char *
708 ngx\_rtsig\_init\_conf(ngx\_cycle\_t *cycle, void *conf)
709 {
710     ngx\_rtsig\_conf\_t *rtscf = conf;
711
712     /* LinuxThreads use the first 3 RT signals */
713     ngx\_conf\_init\_uint\_value(rtscf->signo, SIGRTMIN + 10);
714
715     ngx\_conf\_init\_uint\_value(rtscf->overflow_events, 16);
716     ngx\_conf\_init\_uint\_value(rtscf->overflow_test, 32);
717     ngx\_conf\_init\_uint\_value(rtscf->overflow_threshold, 10);
718
719     return NGX\_CONF\_OK;
720 }
721
722
723 static char *
724 ngx\_check ngx\_overflow\_threshold\_bounds(ngx\_conf\_t *cf, void *post, void *data)
725 {
726     if (ngx\_linux\_rtsig\_max) {
727         return ngx\_conf\_check\_num\_bounds(cf, post, data);
728     }
729
730     ngx\_conf\_log\_error(NGX\_LOG\_WARN, cf, 0,
731         "\"rtsig\_overflow\_threshold\" is not supported "
732         "since Linux 2.6.6-mm2, ignored");
733
734     return NGX\_CONF\_OK;
735 }

```

[One Level Up](#)

[Top Level](#)

## src/event/nginx\_event.h - nginx-1.7.10

### Data types defined

- [ngx\\_event\\_actions\\_t](#)
- [ngx\\_event\\_aio\\_s](#)
- [ngx\\_event\\_conf\\_t](#)
- [ngx\\_event\\_module\\_t](#)
- [ngx\\_event\\_mutex\\_t](#)
- [ngx\\_event\\_ovlp\\_t](#)
- [ngx\\_event\\_s](#)

### Macros defined

- [EPOLLRDHUP](#)
- [NGX\\_CLEAR\\_EVENT](#)
- [NGX\\_CLEAR\\_EVENT](#)
- [NGX\\_CLEAR\\_EVENT](#)
- [NGX\\_CLOSE\\_EVENT](#)
- [NGX\\_CLOSE\\_EVENT](#)
- [NGX\\_CLOSE\\_EVENT](#)
- [NGX\\_DISABLE\\_EVENT](#)
- [NGX\\_DISABLE\\_EVENT](#)
- [NGX\\_DISABLE\\_EVENT](#)
- [NGX\\_EVENT\\_CONF](#)
- [NGX\\_EVENT\\_MODULE](#)
- [NGX\\_FLUSH\\_EVENT](#)
- [NGX\\_FLUSH\\_EVENT](#)
- [NGX\\_FLUSH\\_EVENT](#)
- [NGX\\_INVALID\\_INDEX](#)
- [NGX\\_IOCP\\_ACCEPT](#)
- [NGX\\_IOCP\\_CONNECT](#)
- [NGX\\_IOCP\\_IO](#)
- [NGX\\_LEVEL\\_EVENT](#)
- [NGX\\_LEVEL\\_EVENT](#)

- [NGX\\_LEVEL\\_EVENT](#)
- [NGX\\_LEVEL\\_EVENT](#)
- [NGX\\_LEVEL\\_EVENT](#)
- [NGX\\_LOWAT\\_EVENT](#)
- [NGX\\_LOWAT\\_EVENT](#)
- [NGX\\_LOWAT\\_EVENT](#)
- [NGX\\_ONESHOT\\_EVENT](#)
- [NGX\\_ONESHOT\\_EVENT](#)
- [NGX\\_ONESHOT\\_EVENT](#)
- [NGX\\_ONESHOT\\_EVENT](#)
- [NGX\\_ONESHOT\\_EVENT](#)
- [NGX\\_POST\\_EVENTS](#)
- [NGX\\_READ\\_EVENT](#)
- [NGX\\_READ\\_EVENT](#)
- [NGX\\_READ\\_EVENT](#)
- [NGX\\_READ\\_EVENT](#)
- [NGX\\_READ\\_EVENT](#)
- [NGX\\_READ\\_EVENT](#)
- [NGX\\_UPDATE\\_TIME](#)
- [NGX\\_USE\\_AIO\\_EVENT](#)
- [NGX\\_USE\\_CLEAR\\_EVENT](#)
- [NGX\\_USE\\_EPOLL\\_EVENT](#)
- [NGX\\_USE\\_EVENTPORT\\_EVENT](#)
- [NGX\\_USE\\_FD\\_EVENT](#)
- [NGX\\_USE\\_GREEDY\\_EVENT](#)
- [NGX\\_USE\\_IOCP\\_EVENT](#)
- [NGX\\_USE\\_KQUEUE\\_EVENT](#)
- [NGX\\_USE\\_LEVEL\\_EVENT](#)
- [NGX\\_USE\\_LOWAT\\_EVENT](#)
- [NGX\\_USE\\_ONESHOT\\_EVENT](#)
- [NGX\\_USE\\_RTSIG\\_EVENT](#)
- [NGX\\_USE\\_TIMER\\_EVENT](#)
- [NGX\\_USE\\_VNODE\\_EVENT](#)
- [NGX\\_VNODE\\_EVENT](#)

- [NGX\\_VNODE\\_EVENT](#)
- [NGX\\_VNODE\\_EVENT](#)
- [NGX\\_WRITE\\_EVENT](#)
- [NGX\\_WRITE\\_EVENT](#)
- [NGX\\_WRITE\\_EVENT](#)
- [NGX\\_WRITE\\_EVENT](#)
- [NGX\\_WRITE\\_EVENT](#)
- [NGX\\_WRITE\\_EVENT](#)
- [\\_NGX\\_EVENT\\_H\\_INCLUDED](#)
- [ngx\\_add\\_conn](#)
- [ngx\\_add\\_event](#)
- [ngx\\_add\\_timer](#)
- [ngx\\_del\\_conn](#)
- [ngx\\_del\\_event](#)
- [ngx\\_del\\_timer](#)
- [ngx\\_done\\_events](#)
- [ngx\\_event\\_get\\_conf](#)
- [ngx\\_event\\_ident](#)
- [ngx\\_process\\_changes](#)
- [ngx\\_process\\_events](#)
- [ngx\\_recv](#)
- [ngx\\_recv\\_chain](#)
- [ngx\\_send](#)
- [ngx\\_send\\_chain](#)
- [ngx\\_udp\\_recv](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_EVENT_H_INCLUDED_
9 #define _NGX_EVENT_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 #define NGX_INVALID_INDEX 0xd0d0d0d0
17
18
```

```

19 #if (NGX_HAVE_IOCP)
20
21 typedef struct {
22     WSAOVERLAPPED    ovlp;
23     ngx_event_t      *event;
24     int               error;
25 } ngx_event_ovlp_t;
26
27 #endif
28
29
30 typedef struct {
31     ngx_uint_t        lock;
32
33     ngx_event_t       *events;
34     ngx_event_t       *last;
35 } ngx_event_mutex_t;
36
37
38 struct ngx_event_s {
39     void               *data;
40
41     unsigned           write:1;
42
43     unsigned           accept:1;
44
45     /* used to detect the stale events in kqueue, rtsig, and epoll */
46     unsigned           instance:1;
47
48     /*
49      * the event was passed or would be passed to a kernel;
50      * in aio mode - operation was posted.
51     */
52     unsigned           active:1;
53
54     unsigned           disabled:1;
55
56     /* the ready event; in aio mode 0 means that no operation can be posted */
57     unsigned           ready:1;
58
59     unsigned           oneshot:1;
60
61     /* aio operation is complete */
62     unsigned           complete:1;
63
64     unsigned           eof:1;
65     unsigned           error:1;
66
67     unsigned           timedout:1;
68     unsigned           timer_set:1;
69
70     unsigned           delayed:1;
71
72     unsigned           deferred_accept:1;
73
74     /* the pending eof reported by kqueue, epoll or in aio chain operation */
75     unsigned           pending_eof:1;
76
77     unsigned           posted:1;
78
79 #if (NGX_WIN32)
80     /* setsockopt(SO_UPDATE_ACCEPT_CONTEXT) was successful */
81     unsigned           accept_context_updated:1;
82 #endif
83
84 #if (NGX_HAVE_KQUEUE)
85     unsigned           kq_vnode:1;
86
87     /* the pending errno reported by kqueue */
88     int               kq_errno;
89 #endif
90
91     /*
92     * kqueue only:
93     * accept:    number of sockets that wait to be accepted
94     * read:      bytes to read when event is ready

```



```

95     * or lowat when event is set with NGX_LOWAT_EVENT flag
96     * write: available space in buffer when event is ready
97     * or lowat when event is set with NGX_LOWAT_EVENT flag
98     *
99     * iocp: TODO
100    *
101    * otherwise:
102    * accept: 1 if accept many, 0 otherwise
103    */
104
105    #if (NGX_HAVE_KQUEUE) || (NGX_HAVE_IOCP)
106        int available;
107    #else
108        unsigned available:1;
109    #endif
110
111    ngx_event_handler_pt handler;
112
113
114    #if (NGX_HAVE_AIO)
115
116    #if (NGX_HAVE_IOCP)
117        ngx_event_ovlp_t ovlp;
118    #else
119        struct aiocb aiocb;
120    #endif
121
122    #endif
123
124    ngx_uint_t index;
125
126    ngx_log_t *log;
127
128    ngx_rbtree_node_t timer;
129
130    /* the posted queue */
131    ngx_queue_t queue;
132
133    unsigned closed:1;
134
135    /* to test on worker exit */
136    unsigned channel:1;
137    unsigned resolver:1;
138
139    unsigned cancelable:1;
140
141
142    #if 0
143
144    /* the threads support */
145
146    /*
147     * the event thread context, we store it here
148     * if $(CC) does not understand __thread declaration
149     * and pthread_getspecific() is too costly
150     */
151
152    void *thr_ctx;
153
154    #if (NGX_EVENT_T_PADDING)
155
156    /* event should not cross cache line in SMP */
157
158    uint32_t padding[NGX_EVENT_T_PADDING];
159    #endif
160    #endif
161 };
162
163
164    #if (NGX_HAVE_FILE_AIO)
165
166    struct ngx_event_aio_s {
167        void *data;
168        ngx_event_handler_pt handler;
169        ngx_file_t *file;
170

```

```

171     ngx_fd_t          fd;
172
173 #if (NGX_HAVE_EVENTFD)
174     int64_t          res;
175 #if (NGX_TEST_BUILD_EPOLL)
176     ngx_err_t        err;
177     size_t           nbytes;
178 #endif
179 #else
180     ngx_err_t        err;
181     size_t           nbytes;
182 #endif
183
184 #if (NGX_HAVE_AIO_SENDFILE)
185     off_t            last_offset;
186 #endif
187
188     ngx_aiocb_t      aiocb;
189     ngx_event_t      event;
190 };
191
192 #endif
193
194
195 typedef struct {
196     ngx_int_t        (*add)(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags);
197     ngx_int_t        (*del)(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags);
198
199     ngx_int_t        (*enable)(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags);
200     ngx_int_t        (*disable)(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags);
201
202     ngx_int_t        (*add_conn)(ngx_connection_t *c);
203     ngx_int_t        (*del_conn)(ngx_connection_t *c, ngx_uint_t flags);
204
205     ngx_int_t        (*process_changes)(ngx_cycle_t *cycle, ngx_uint_t nowait);
206     ngx_int_t        (*process_events)(ngx_cycle_t *cycle, ngx_msec_t timer,
207                                     ngx_uint_t flags);
208
209     ngx_int_t        (*init)(ngx_cycle_t *cycle, ngx_msec_t timer);
210     void             (*done)(ngx_cycle_t *cycle);
211 } ngx_event_actions_t;
212
213
214 extern ngx_event_actions_t  ngx_event_actions;
215
216
217 /*
218  * The event filter requires to read/write the whole data:
219  * select, poll, /dev/poll, kqueue, epoll.
220  */
221 #define NGX_USE_LEVEL_EVENT      0x00000001
222
223 /*
224  * The event filter is deleted after a notification without an additional
225  * syscall: kqueue, epoll.
226  */
227 #define NGX_USE_ONESHOT_EVENT    0x00000002
228
229 /*
230  * The event filter notifies only the changes and an initial level:
231  * kqueue, epoll.
232  */
233 #define NGX_USE_CLEAR_EVENT      0x00000004
234
235 /*
236  * The event filter has kqueue features: the eof flag, errno,
237  * available data, etc.
238  */
239 #define NGX_USE_KQUEUE_EVENT     0x00000008
240
241 /*
242  * The event filter supports low water mark: kqueue's NOTE_LOWAT.
243  * kqueue in FreeBSD 4.1-4.2 has no NOTE_LOWAT so we need a separate flag.
244  */
245 #define NGX_USE_LOWAT_EVENT      0x00000010
246

```

```

247 /*
248 * The event filter requires to do i/o operation until EAGAIN: epoll, rtsig.
249 */
250 #define NGX_USE_GREEDY_EVENT    0x00000020
251
252 /*
253 * The event filter is epoll.
254 */
255 #define NGX_USE_EPOLL_EVENT     0x00000040
256
257 /*
258 * No need to add or delete the event filters: rtsig.
259 */
260 #define NGX_USE_RTSIG_EVENT     0x00000080
261
262 /*
263 * No need to add or delete the event filters: overlapped, aio_read,
264 * aioread, io\_submit.
265 */
266 #define NGX_USE_AIO_EVENT       0x00000100
267
268 /*
269 * Need to add socket or handle only once: i/o completion port.
270 * It also requires NGX_HAVE_AIO and NGX\_USE\_AIO\_EVENT to be set.
271 */
272 #define NGX_USE_IOCP_EVENT      0x00000200
273
274 /*
275 * The event filter has no opaque data and requires file descriptors table:
276 * poll, /dev/poll, rtsig.
277 */
278 #define NGX_USE_FD_EVENT       0x00000400
279
280 /*
281 * The event module handles periodic or absolute timer event by itself:
282 * kqueue in FreeBSD 4.4, NetBSD 2.0, and MacOSX 10.4, Solaris 10's event ports.
283 */
284 #define NGX_USE_TIMER_EVENT     0x00000800
285
286 /*
287 * All event filters on file descriptor are deleted after a notification:
288 * Solaris 10's event ports.
289 */
290 #define NGX_USE_EVENTPORT_EVENT 0x00001000
291
292 /*
293 * The event filter support vnode notifications: kqueue.
294 */
295 #define NGX_USE_VNODE_EVENT     0x00002000
296
297
298 /*
299 * The event filter is deleted just before the closing file.
300 * Has no meaning for select and poll.
301 * kqueue, epoll, rtsig, eventport: allows to avoid explicit delete,
302 * because filter automatically is deleted
303 * on file close,
304 *
305 * /dev/poll: we need to flush POLLREMOVE event
306 * before closing file.
307 */
308 #define NGX_CLOSE_EVENT        1
309
310 /*
311 * disable temporarily event filter, this may avoid locks
312 * in kernel malloc()/free(): kqueue.
313 */
314 #define NGX_DISABLE_EVENT      2
315
316 /*
317 * event must be passed to kernel right now, do not wait until batch processing.
318 */
319 #define NGX_FLUSH_EVENT        4
320
321
322 /* these flags have a meaning only for kqueue */

```

```

323 #define NGX_LOWAT_EVENT      0
324 #define NGX_VNODE_EVENT     0
325
326
327 #if (NGX_HAVE_EPOLL) && !(NGX_HAVE_EPOLLRDHUP)
328 #define EPOLLRDHUP          0
329 #endif
330
331
332 #if (NGX_HAVE_KQUEUE)
333
334 #define NGX_READ_EVENT      EVFILT_READ
335 #define NGX_WRITE_EVENT     EVFILT_WRITE
336
337 #undef  NGX_VNODE_EVENT
338 #define NGX_VNODE_EVENT     EVFILT_VNODE
339
340 /*
341  * NGX_CLOSE_EVENT, NGX_LOWAT_EVENT, and NGX_FLUSH_EVENT are the module flags
342  * and they must not go into a kernel so we need to choose the value
343  * that must not interfere with any existent and future kqueue flags.
344  * kqueue has such values - EV_FLAG1, EV_EOF, and EV_ERROR:
345  * they are reserved and cleared on a kernel entrance.
346  */
347 #undef  NGX_CLOSE_EVENT
348 #define NGX_CLOSE_EVENT     EV_EOF
349
350 #undef  NGX_LOWAT_EVENT
351 #define NGX_LOWAT_EVENT     EV_FLAG1
352
353 #undef  NGX_FLUSH_EVENT
354 #define NGX_FLUSH_EVENT     EV_ERROR
355
356 #define NGX_LEVEL_EVENT     0
357 #define NGX_ONESHOT_EVENT   EV_ONESHOT
358 #define NGX_CLEAR_EVENT     EV_CLEAR
359
360 #undef  NGX_DISABLE_EVENT
361 #define NGX_DISABLE_EVENT   EV_DISABLE
362
363
364 #elif (NGX_HAVE_DEVPOLL || NGX_HAVE_EVENTPORT)
365
366 #define NGX_READ_EVENT      POLLIN
367 #define NGX_WRITE_EVENT     POLLOUT
368
369 #define NGX_LEVEL_EVENT     0
370 #define NGX_ONESHOT_EVENT   1
371
372
373 #elif (NGX_HAVE_EPOLL)
374
375 #define NGX_READ_EVENT      (EPOLLIN|EPOLLRDHUP)
376 #define NGX_WRITE_EVENT     EPOLLOUT
377
378 #define NGX_LEVEL_EVENT     0
379 #define NGX_CLEAR_EVENT     EPOLLET
380 #define NGX_ONESHOT_EVENT   0x70000000
381 #if 0
382 #define NGX_ONESHOT_EVENT EPOLLONESHOT
383 #endif
384
385
386 #elif (NGX_HAVE_POLL)
387
388 #define NGX_READ_EVENT      POLLIN
389 #define NGX_WRITE_EVENT     POLLOUT
390
391 #define NGX_LEVEL_EVENT     0
392 #define NGX_ONESHOT_EVENT   1
393
394
395 #else /* select */
396
397 #define NGX_READ_EVENT      0
398 #define NGX_WRITE_EVENT     1

```

```

399
400 #define NGX_LEVEL_EVENT      0
401 #define NGX_ONESHOT_EVENT    1
402
403 #endif /* NGX_HAVE_KQUEUE */
404
405
406 #if (NGX_HAVE_IOCP)
407 #define NGX_IOCP_ACCEPT      0
408 #define NGX_IOCP_IO         1
409 #define NGX_IOCP_CONNECT    2
410 #endif
411
412
413 #ifndef NGX_CLEAR_EVENT
414 #define NGX_CLEAR_EVENT      0 /* dummy declaration */
415 #endif
416
417
418 #define ngx_process_changes  ngx_event_actions.process_changes
419 #define ngx_process_events  ngx_event_actions.process_events
420 #define ngx_done_events     ngx_event_actions.done
421
422 #define ngx_add_event        ngx_event_actions.add
423 #define ngx_del_event        ngx_event_actions.del
424 #define ngx_add_conn        ngx_event_actions.add_conn
425 #define ngx_del_conn        ngx_event_actions.del_conn
426
427 #define ngx_add_timer        ngx_event_add_timer
428 #define ngx_del_timer        ngx_event_del_timer
429
430
431 extern ngx_os_io_t  ngx_io;
432
433 #define ngx_recv        ngx_io.recv
434 #define ngx_recv_chain ngx_io.recv_chain
435 #define ngx_udp_recv   ngx_io.udp_recv
436 #define ngx_send       ngx_io.send
437 #define ngx_send_chain ngx_io.send_chain
438
439
440 #define NGX_EVENT_MODULE      0x544E5645 /* "EVNT" */
441 #define NGX_EVENT_CONF       0x02000000
442
443
444 typedef struct {
445     ngx_uint_t  connections;
446     ngx_uint_t  use;
447
448     ngx_flag_t  multi_accept;
449     ngx_flag_t  accept_mutex;
450
451     ngx_msec_t  accept_mutex_delay;
452
453     u_char      *name;
454
455     #if (NGX_DEBUG)
456     ngx_array_t  debug_connection;
457     #endif
458 } ngx_event_conf_t;
459
460
461 typedef struct {
462     ngx_str_t      *name;
463
464     void            (*create_conf)(ngx_cycle_t *cycle);
465     char            (*init_conf)(ngx_cycle_t *cycle, void *conf);
466
467     ngx_event_actions_t  actions;
468 } ngx_event_module_t;
469
470
471 extern ngx_atomic_t      *ngx_connection_counter;
472
473 extern ngx_atomic_t      *ngx_accept_mutex_ptr;
474 extern ngx_shmtx_t       ngx_accept_mutex;

```

```

475 extern ngx_uint_t          ngx_use_accept_mutex;
476 extern ngx_uint_t          ngx_accept_events;
477 extern ngx_uint_t          ngx_accept_mutex_held;
478 extern ngx_msec_t          ngx_accept_mutex_delay;
479 extern ngx_int_t           ngx_accept_disabled;
480
481
482 #if (NGX_STAT_STUB)
483
484 extern ngx_atomic_t         *ngx_stat_accepted;
485 extern ngx_atomic_t         *ngx_stat_handled;
486 extern ngx_atomic_t         *ngx_stat_requests;
487 extern ngx_atomic_t         *ngx_stat_active;
488 extern ngx_atomic_t         *ngx_stat_reading;
489 extern ngx_atomic_t         *ngx_stat_writing;
490 extern ngx_atomic_t         *ngx_stat_waiting;
491
492 #endif
493
494
495 #define NGX_UPDATE_TIME     1
496 #define NGX_POST_EVENTS     2
497
498
499 extern sig_atomic_t         ngx_event_timer_alarm;
500 extern ngx_uint_t          ngx_event_flags;
501 extern ngx_module_t        ngx_events_module;
502 extern ngx_module_t        ngx_event_core_module;
503
504
505 #define ngx_event_get_conf(conf_ctx, module) \
506     (*(ngx_get_conf(conf_ctx, ngx_events_module))) [module.ctx_index];
507
508
509
510 void ngx_event_accept(ngx_event_t *ev);
511 ngx_int_t ngx_trylock_accept_mutex(ngx_cycle_t *cycle);
512 u_char *ngx_accept_log_error(ngx_log_t *log, u_char *buf, size_t len);
513
514
515 void ngx_process_events_and_timers(ngx_cycle_t *cycle);
516 ngx_int_t ngx_handle_read_event(ngx_event_t *rev, ngx_uint_t flags);
517 ngx_int_t ngx_handle_write_event(ngx_event_t *wev, size_t lowat);
518
519
520 #if (NGX_WIN32)
521 void ngx_event_acptex(ngx_event_t *ev);
522 ngx_int_t ngx_event_post_acptex(ngx_listening_t *ls, ngx_uint_t n);
523 u_char *ngx_acptex_log_error(ngx_log_t *log, u_char *buf, size_t len);
524 #endif
525
526
527 ngx_int_t ngx_send_lowat(ngx_connection_t *c, size_t lowat);
528
529
530 /* used in ngx_log_debugX() */
531 #define ngx_event_ident(p) ((ngx_connection_t *) (p))->fd
532
533
534 #include <ngx_event_timer.h>
535 #include <ngx_event_posted.h>
536 #include <ngx_event_busy_lock.h>
537
538 #if (NGX_WIN32)
539 #include <ngx_ioep_module.h>
540 #endif
541
542
543 #endif /* _NGX_EVENT_H_INCLUDED */

```

[One Level Up](#)

[Top Level](#)

# src/core/nginx\_queue.h - nginx-1.7.10

## Data types defined

- [ngx\\_queue\\_s](#)
- [ngx\\_queue\\_t](#)

## Macros defined

- [\\_NGX\\_QUEUE\\_H\\_INCLUDED\\_](#)
- [ngx\\_queue\\_add](#)
- [ngx\\_queue\\_data](#)
- [ngx\\_queue\\_empty](#)
- [ngx\\_queue\\_head](#)
- [ngx\\_queue\\_init](#)
- [ngx\\_queue\\_insert\\_after](#)
- [ngx\\_queue\\_insert\\_head](#)
- [ngx\\_queue\\_insert\\_tail](#)
- [ngx\\_queue\\_last](#)
- [ngx\\_queue\\_next](#)
- [ngx\\_queue\\_prev](#)
- [ngx\\_queue\\_remove](#)
- [ngx\\_queue\\_remove](#)
- [ngx\\_queue\\_sentinel](#)
- [ngx\\_queue\\_split](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10
11
12 #ifndef _NGX_QUEUE_H_INCLUDED_
13 #define _NGX_QUEUE_H_INCLUDED_
14
15
16 typedef struct ngx_queue_s ngx_queue_t;
17
18 struct ngx_queue_s {
19     ngx_queue_t *prev;
20     ngx_queue_t *next;
```

```

21 };
22
23
24 #define ngx_queue_init(q) \
25     (q)->prev = q; \
26     (q)->next = q
27
28
29 #define ngx_queue_empty(h) \
30     (h == (h)->prev)
31
32
33 #define ngx_queue_insert_head(h, x) \
34     (x)->next = (h)->next; \
35     (x)->next->prev = x; \
36     (x)->prev = h; \
37     (h)->next = x
38
39
40 #define ngx_queue_insert_after ngx\_queue\_insert\_head
41
42
43 #define ngx_queue_insert_tail(h, x) \
44     (x)->prev = (h)->prev; \
45     (x)->prev->next = x; \
46     (x)->next = h; \
47     (h)->prev = x
48
49
50 #define ngx_queue_head(h) \
51     (h)->next
52
53
54 #define ngx_queue_last(h) \
55     (h)->prev
56
57
58 #define ngx_queue_sentinel(h) \
59     (h)
60
61
62 #define ngx_queue_next(q) \
63     (q)->next
64
65
66 #define ngx_queue_prev(q) \
67     (q)->prev
68
69
70 #if (NGX_DEBUG)
71
72 #define ngx_queue_remove(x) \
73     (x)->next->prev = (x)->prev; \
74     (x)->prev->next = (x)->next; \
75     (x)->prev = NULL; \
76     (x)->next = NULL
77
78 #else
79
80 #define ngx_queue_remove(x) \
81     (x)->next->prev = (x)->prev; \
82     (x)->prev->next = (x)->next
83
84 #endif
85
86
87 #define ngx_queue_split(h, q, n) \
88     (n)->prev = (h)->prev; \
89     (n)->prev->next = n; \
90     (n)->next = q; \
91     (h)->prev = (q)->prev; \
92     (h)->prev->next = h; \
93     (q)->prev = n;
94
95
96 #define ngx_queue_add(h, n) \

```



```
97     (h)->prev->next = (n)->next;           \  
98     (n)->next->prev = (h)->prev;         \  
99     (h)->prev = (n)->prev;              \  
100    (h)->prev->next = h;                  \  
101                                         \  
102                                         \  
103 #define ngx_queue_data(q, type, link)    \  
104     (type *) ((u_char *) q - offsetof(type, link)) \  
105                                         \  
106                                         \  
107 ngx_queue_t *ngx_queue_middle(ngx_queue_t *queue); \  
108 void ngx_queue_sort(ngx_queue_t *queue, \  
109     ngx_int_t (*cmp)(const ngx_queue_t *, const ngx_queue_t *)); \  
110                                         \  
111                                         \  
112 #endif /* NGX_QUEUE_H_INCLUDED */
```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_queue.c - nginx-1.7.10

### Functions defined

- [ngx\\_queue\\_middle](#)
- [ngx\\_queue\\_sort](#)

### Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10
11
12  /*
13  * find the middle queue element if the queue has odd number of elements
14  * or the first element of the queue's second part otherwise
15  */
16
17  ngx_queue_t *
18  ngx_queue_middle(ngx_queue_t *queue)
19  {
20     ngx_queue_t *middle, *next;
21
22     middle = ngx_queue_head(queue);
23
24     if (middle == ngx_queue_last(queue)) {
25         return middle;
26     }
27
28     next = ngx_queue_head(queue);
29
30     for ( ;; ) {
31         middle = ngx_queue_next(middle);
32
33         next = ngx_queue_next(next);
34
35         if (next == ngx_queue_last(queue)) {
36             return middle;
37         }
38
39         next = ngx_queue_next(next);
40
41         if (next == ngx_queue_last(queue)) {
42             return middle;
43         }
44     }
45 }
46
47
48  /* the stable insertion sort */
49
50  void
51  ngx_queue_sort(ngx_queue_t *queue,
52               ngx_int_t (*cmp)(const ngx_queue_t *, const ngx_queue_t *))
53  {
54     ngx_queue_t *q, *prev, *next;
55
56     q = ngx_queue_head(queue);
57
58     if (q == ngx_queue_last(queue)) {
59         return;
60     }
```

```
61
62 for (q = ngx\_queue\_next(q); q != ngx\_queue\_sentinel(queue); q = next) {
63
64     prev = ngx\_queue\_prev(q);
65     next = ngx\_queue\_next(q);
66
67     ngx\_queue\_remove(q);
68
69     do {
70         if (cmp(prev, q) <= 0) {
71             break;
72         }
73
74         prev = ngx\_queue\_prev(prev);
75
76     } while (prev != ngx\_queue\_sentinel(queue));
77
78     ngx\_queue\_insert\_after(prev, q);
79 }
80 }
```

[One Level Up](#)

[Top Level](#)

## src/os/unix/nginx\_files.h - nginx-1.7.10

### Data types defined

- [ngx\\_dir\\_t](#)
- [ngx\\_fd\\_t](#)
- [ngx\\_file\\_info\\_t](#)
- [ngx\\_file\\_mapping\\_t](#)
- [ngx\\_file\\_uniq\\_t](#)
- [ngx\\_glob\\_t](#)

### Functions defined

- [ngx\\_de\\_info](#)
- [ngx\\_write\\_fd](#)

### Macros defined

- [NGX\\_AT\\_FDCWD](#)
- [NGX\\_DIR\\_MASK\\_LEN](#)
- [NGX\\_FILE\\_APPEND](#)
- [NGX\\_FILE\\_CREATE\\_OR\\_OPEN](#)
- [NGX\\_FILE\\_DEFAULT\\_ACCESS](#)
- [NGX\\_FILE\\_DIRECTORY](#)
- [NGX\\_FILE\\_DIRECTORY](#)
- [NGX\\_FILE\\_ERROR](#)
- [NGX\\_FILE\\_NOFOLLOW](#)
- [NGX\\_FILE\\_NONBLOCK](#)
- [NGX\\_FILE\\_OPEN](#)
- [NGX\\_FILE\\_OWNER\\_ACCESS](#)
- [NGX\\_FILE\\_RDONLY](#)
- [NGX\\_FILE\\_RDWR](#)
- [NGX\\_FILE\\_SEARCH](#)
- [NGX\\_FILE\\_SEARCH](#)
- [NGX\\_FILE\\_SEARCH](#)
- [NGX\\_FILE\\_SEARCH](#)
- [NGX\\_FILE\\_TRUNCATE](#)

- [NGX\\_FILE\\_WRONLY](#)
- [NGX\\_HAVE\\_CASELESS\\_FILESYSTEM](#)
- [NGX\\_HAVE\\_MAX\\_PATH](#)
- [NGX\\_HAVE\\_READ\\_AHEAD](#)
- [NGX\\_HAVE\\_READ\\_AHEAD](#)
- [NGX\\_INVALID\\_FILE](#)
- [NGX\\_LINEFEED](#)
- [NGX\\_LINEFEED\\_SIZE](#)
- [NGX\\_MAX\\_PATH](#)
- [NGX\\_MAX\\_PATH](#)
- [\\_NGX\\_FILES\\_H\\_INCLUDED](#)
- [ngx\\_change\\_file\\_access](#)
- [ngx\\_change\\_file\\_access\\_n](#)
- [ngx\\_close\\_dir](#)
- [ngx\\_close\\_dir\\_n](#)
- [ngx\\_close\\_file](#)
- [ngx\\_close\\_file\\_n](#)
- [ngx\\_create\\_dir](#)
- [ngx\\_create\\_dir\\_n](#)
- [ngx\\_de\\_access](#)
- [ngx\\_de\\_fs\\_size](#)
- [ngx\\_de\\_info\\_n](#)
- [ngx\\_de\\_is\\_dir](#)
- [ngx\\_de\\_is\\_dir](#)
- [ngx\\_de\\_is\\_file](#)
- [ngx\\_de\\_is\\_file](#)
- [ngx\\_de\\_is\\_link](#)
- [ngx\\_de\\_is\\_link](#)
- [ngx\\_de\\_link\\_info](#)
- [ngx\\_de\\_link\\_info\\_n](#)
- [ngx\\_de\\_mtime](#)
- [ngx\\_de\\_name](#)
- [ngx\\_de\\_namelen](#)

- [ngx\\_de\\_namelen](#)
- [ngx\\_de\\_size](#)
- [ngx\\_delete\\_dir](#)
- [ngx\\_delete\\_dir\\_n](#)
- [ngx\\_delete\\_file](#)
- [ngx\\_delete\\_file\\_n](#)
- [ngx\\_dir\\_access](#)
- [ngx\\_directio\\_off\\_n](#)
- [ngx\\_directio\\_on](#)
- [ngx\\_directio\\_on](#)
- [ngx\\_directio\\_on](#)
- [ngx\\_directio\\_on\\_n](#)
- [ngx\\_directio\\_on\\_n](#)
- [ngx\\_directio\\_on\\_n](#)
- [ngx\\_directio\\_on\\_n](#)
- [ngx\\_directio\\_on\\_n](#)
- [ngx\\_fd\\_info](#)
- [ngx\\_fd\\_info\\_n](#)
- [ngx\\_file\\_access](#)
- [ngx\\_file\\_at\\_info](#)
- [ngx\\_file\\_at\\_info\\_n](#)
- [ngx\\_file\\_fs\\_size](#)
- [ngx\\_file\\_info](#)
- [ngx\\_file\\_info\\_n](#)
- [ngx\\_file\\_mtime](#)
- [ngx\\_file\\_size](#)
- [ngx\\_file\\_uniq](#)
- [ngx\\_getcwd](#)
- [ngx\\_getcwd\\_n](#)
- [ngx\\_is\\_dir](#)
- [ngx\\_is\\_exec](#)
- [ngx\\_is\\_file](#)
- [ngx\\_is\\_link](#)
- [ngx\\_linefeed](#)

- [ngx\\_link\\_info](#)
- [ngx\\_link\\_info\\_n](#)
- [ngx\\_lock\\_fd\\_n](#)
- [ngx\\_open\\_dir\\_n](#)
- [ngx\\_open\\_file](#)
- [ngx\\_open\\_file](#)
- [ngx\\_open\\_file\\_n](#)
- [ngx\\_open\\_glob\\_n](#)
- [ngx\\_open\\_tempfile\\_n](#)
- [ngx\\_openat\\_file](#)
- [ngx\\_openat\\_file\\_n](#)
- [ngx\\_path\\_separator](#)
- [ngx\\_read\\_ahead](#)
- [ngx\\_read\\_ahead](#)
- [ngx\\_read\\_ahead\\_n](#)
- [ngx\\_read\\_ahead\\_n](#)
- [ngx\\_read\\_ahead\\_n](#)
- [ngx\\_read\\_ahead\\_n](#)
- [ngx\\_read\\_dir\\_n](#)
- [ngx\\_read\\_fd](#)
- [ngx\\_read\\_fd\\_n](#)
- [ngx\\_read\\_file\\_n](#)
- [ngx\\_read\\_file\\_n](#)
- [ngx\\_realpath](#)
- [ngx\\_realpath\\_n](#)
- [ngx\\_rename\\_file](#)
- [ngx\\_rename\\_file\\_n](#)
- [ngx\\_set\\_file\\_time\\_n](#)
- [ngx\\_set\\_stderr](#)
- [ngx\\_set\\_stderr\\_n](#)
- [ngx\\_stderr](#)
- [ngx\\_trylock\\_fd\\_n](#)
- [ngx\\_unlock\\_fd\\_n](#)
- [ngx\\_write\\_console](#)

- [ngx\\_write\\_fd\\_n](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_FILES\_H\_INCLUDED
9 #define \_NGX\_FILES\_H\_INCLUDED
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 typedef int                ngx_fd_t;
17 typedef struct stat        ngx_file_info_t;
18 typedef ino_t              ngx_file_uniq_t;
19
20
21 typedef struct {
22     u_char                  *name;
23     size_t                  size;
24     void                    *addr;
25     ngx\_fd\_t                  fd;
26     ngx\_log\_t                *log;
27 } ngx_file_mapping_t;
28
29
30 typedef struct {
31     DIR                     *dir;
32     struct dirent           *de;
33     struct stat              info;
34
35     unsigned                 type:8;
36     unsigned                 valid_info:1;
37 } ngx_dir_t;
38
39
40 typedef struct {
41     size_t                  n;
42     glob_t                  pglob;
43     u_char                  *pattern;
44     ngx\_log\_t                *log;
45     ngx\_uint\_t              test;
46 } ngx_glob_t;
47
48
49 #define NGX\_INVALID\_FILE        -1
50 #define NGX\_FILE\_ERROR          -1
51
52
53
54 #ifdef \_\_CYGWIN\_\_
55
56 #ifndef NGX\_HAVE\_CASELESS\_FILESYSTEM
57 #define NGX\_HAVE\_CASELESS\_FILESYSTEM 1
58 #endif
59
60 #define ngx\_open\_file(name, mode, create, access) \
61     open((const char *) name, mode|create|O_BINARY, access)
62
63 #else
64
65 #define ngx\_open\_file(name, mode, create, access) \
66     open((const char *) name, mode|create, access)
67
68 #endif
69
70 #define ngx\_open\_file\_n        "open()"

```



```

71
72 #define NGX_FILE_RDONLY          O_RDONLY
73 #define NGX_FILE_WRONLY          O_WRONLY
74 #define NGX_FILE_RDWR           O_RDWR
75 #define NGX_FILE_CREATE_OR_OPEN  O_CREAT
76 #define NGX_FILE_OPEN            0
77 #define NGX_FILE_TRUNCATE        (O_CREAT|O_TRUNC)
78 #define NGX_FILE_APPEND          (O_WRONLY|O_APPEND)
79 #define NGX_FILE_NONBLOCK        O_NONBLOCK
80
81 #if (NGX_HAVE_OPENAT)
82 #define NGX_FILE_NOFOLLOW        O_NOFOLLOW
83
84 #if defined(O_DIRECTORY)
85 #define NGX_FILE_DIRECTORY       O_DIRECTORY
86 #else
87 #define NGX_FILE_DIRECTORY       0
88 #endif
89
90 #if defined(O_SEARCH)
91 #define NGX_FILE_SEARCH           (O_SEARCH|NGX_FILE_DIRECTORY)
92
93 #elif defined(O_EXEC)
94 #define NGX_FILE_SEARCH           (O_EXEC|NGX_FILE_DIRECTORY)
95
96 #elif (NGX_HAVE_O_PATH)
97 #define NGX_FILE_SEARCH           (O_PATH|O_RDONLY|NGX_FILE_DIRECTORY)
98
99 #else
100 #define NGX_FILE_SEARCH           (O_RDONLY|NGX_FILE_DIRECTORY)
101 #endif
102
103 #endif /* NGX_HAVE_OPENAT */
104
105 #define NGX_FILE_DEFAULT_ACCESS  0644
106 #define NGX_FILE_OWNER_ACCESS    0600
107
108
109 #define ngx_close_file            close
110 #define ngx_close_file_n         "close()"
111
112
113 #define ngx_delete_file(name)     unlink((const char *) name)
114 #define ngx_delete_file_n        "unlink()"
115
116
117 ngx_fd_t ngx_open_tempfile(u_char *name, ngx_uint_t persistent,
118     ngx_uint_t access);
119 #define ngx_open_tempfile_n      "open()"
120
121
122 ssize_t ngx_read_file(ngx_file_t *file, u_char *buf, size_t size, off_t offset);
123 #if (NGX_HAVE_PREAD)
124 #define ngx_read_file_n          "pread()"
125 #else
126 #define ngx_read_file_n          "read()"
127 #endif
128
129 ssize_t ngx_write_file(ngx_file_t *file, u_char *buf, size_t size,
130     off_t offset);
131
132 ssize_t ngx_write_chain_to_file(ngx_file_t *file, ngx_chain_t *ce,
133     off_t offset, ngx_pool_t *pool);
134
135
136 #define ngx_read_fd              read
137 #define ngx_read_fd_n            "read()"
138
139 /*
140  * we use inlined function instead of simple #define
141  * because glibc 2.3 sets warn_unused_result attribute for write()
142  * and in this case gcc 4.3 ignores (void) cast
143  */
144 static ngx_inline ssize_t
145 ngx_write_fd(ngx_fd_t fd, void *buf, size_t n)
146 {

```

```

147     return write(fd, buf, n);
148 }
149
150 #define ngx_write_fd_n          "write()"
151
152
153 #define ngx_write_console      ngx_write_fd
154
155
156 #define ngx_linefeed(p)        *p++ = LF;
157 #define NGX_LINEFEED_SIZE     1
158 #define NGX_LINEFEED          "\x0a"
159
160
161 #define ngx_rename_file(o, n)  rename((const char *) o, (const char *) n)
162 #define ngx_rename_file_n     "rename()"
163
164
165 #define ngx_change_file_access(n, a) chmod((const char *) n, a)
166 #define ngx_change_file_access_n "chmod()"
167
168
169 ngx_int_t ngx_set_file_time(u_char *name, ngx_fd_t fd, time_t s);
170 #define ngx_set_file_time_n    "utimes()"
171
172
173 #define ngx_file_info(file, sb) stat((const char *) file, sb)
174 #define ngx_file_info_n       "stat()"
175
176 #define ngx_fd_info(fd, sb)    fstat(fd, sb)
177 #define ngx_fd_info_n         "fstat()"
178
179 #define ngx_link_info(file, sb) lstat((const char *) file, sb)
180 #define ngx_link_info_n       "lstat()"
181
182 #define ngx_is_dir(sb)         (S_ISDIR((sb)->st_mode))
183 #define ngx_is_file(sb)        (S_ISREG((sb)->st_mode))
184 #define ngx_is_link(sb)        (S_ISLNK((sb)->st_mode))
185 #define ngx_is_exec(sb)        (((sb)->st_mode & S_IXUSR) == S_IXUSR)
186 #define ngx_file_access(sb)     ((sb)->st_mode & 0777)
187 #define ngx_file_size(sb)       (sb)->st_size
188 #define ngx_file_fs_size(sb)    ngx_max((sb)->st_size, (sb)->st_blocks * 512)
189 #define ngx_file_mtime(sb)      (sb)->st_mtime
190 #define ngx_file_uniq(sb)       (sb)->st_ino
191
192
193 ngx_int_t ngx_create_file_mapping(ngx_file_mapping_t *fm);
194 void ngx_close_file_mapping(ngx_file_mapping_t *fm);
195
196
197 #define ngx_realpath(p, r)      (u_char *) realpath((char *) p, (char *) r)
198 #define ngx_realpath_n         "realpath()"
199 #define ngx_getcwd(buf, size)  (getcwd((char *) buf, size) != NULL)
200 #define ngx_getcwd_n           "getcwd()"
201 #define ngx_path_separator(c)  ((c) == '/')
202
203
204 #if defined(PATH_MAX)
205
206 #define NGX_HAVE_MAX_PATH      1
207 #define NGX_MAX_PATH           PATH_MAX
208
209 #else
210
211 #define NGX_MAX_PATH           4096
212
213 #endif
214
215
216 #define NGX_DIR_MASK_LEN       0
217
218
219 ngx_int_t ngx_open_dir(ngx_str_t *name, ngx_dir_t *dir);
220 #define ngx_open_dir_n         "opendir()"
221
222

```

```

223 #define ngx_close_dir(d)          closedir((d)->dir)
224 #define ngx_close_dir_n          "closedir()"
225
226
227 ngx_int_t ngx_read_dir(ngx_dir_t *dir);
228 #define ngx_read_dir_n            "readdir()"
229
230
231 #define ngx_create_dir(name, access) mkdir((const char *) name, access)
232 #define ngx_create_dir_n          "mkdir()"
233
234
235 #define ngx_delete_dir(name)      rmdir((const char *) name)
236 #define ngx_delete_dir_n          "rmdir()"
237
238
239 #define ngx_dir_access(a)         (a | (a & 0444) >> 2)
240
241
242 #define ngx_de_name(dir)          ((u_char *) (dir)->de->d_name)
243 #if (NGX_HAVE_D_NAMLEN)
244 #define ngx_de_namelen(dir)      (dir)->de->d_namlen
245 #else
246 #define ngx_de_namelen(dir)      ngx_strlen((dir)->de->d_name)
247 #endif
248
249 static ngx_inline ngx_int_t
250 ngx_de_info(u_char *name, ngx_dir_t *dir)
251 {
252     dir->type = 0;
253     return stat((const char *) name, &dir->info);
254 }
255
256 #define ngx_de_info_n              "stat()"
257 #define ngx_de_link_info(name, dir) lstat((const char *) name, &(dir)->info)
258 #define ngx_de_link_info_n        "lstat()"
259
260 #if (NGX_HAVE_D_TYPE)
261
262 /*
263  * some file systems (e.g. XFS on Linux and CD9660 on FreeBSD)
264  * do not set dirent.d_type
265  */
266
267 #define ngx_de_is_dir(dir)         \
268     (((dir)->type) ? ((dir)->type == DT_DIR) : (S_ISDIR((dir)->info.st_mode))) \
269 #define ngx_de_is_file(dir)       \
270     (((dir)->type) ? ((dir)->type == DT_REG) : (S_ISREG((dir)->info.st_mode))) \
271 #define ngx_de_is_link(dir)       \
272     (((dir)->type) ? ((dir)->type == DT_LNK) : (S_ISLNK((dir)->info.st_mode)))
273
274 #else
275
276 #define ngx_de_is_dir(dir)        (S_ISDIR((dir)->info.st_mode))
277 #define ngx_de_is_file(dir)       (S_ISREG((dir)->info.st_mode))
278 #define ngx_de_is_link(dir)       (S_ISLNK((dir)->info.st_mode))
279
280 #endif
281
282 #define ngx_de_access(dir)        (((dir)->info.st_mode) & 0777)
283 #define ngx_de_size(dir)          (dir)->info.st_size
284 #define ngx_de_fs_size(dir)       \
285     ngx_max((dir)->info.st_size, (dir)->info.st_blocks * 512) \
286 #define ngx_de_mtime(dir)         (dir)->info.st_mtime
287
288
289 ngx_int_t ngx_open_glob(ngx_glob_t *gl);
290 #define ngx_open_glob_n            "glob()"
291 ngx_int_t ngx_read_glob(ngx_glob_t *gl, ngx_str_t *name);
292 void ngx_close_glob(ngx_glob_t *gl);
293
294
295 ngx_err_t ngx_trylock_fd(ngx_fd_t fd);
296 ngx_err_t ngx_lock_fd(ngx_fd_t fd);
297 ngx_err_t ngx_unlock_fd(ngx_fd_t fd);
298

```

```

299 #define ngx_trylock_fd_n      "fcntl(F_SETLK, F_WRLCK)"
300 #define ngx_lock_fd_n        "fcntl(F_SETLKW, F_WRLCK)"
301 #define ngx_unlock_fd_n      "fcntl(F_SETLK, F_UNLCK)"
302
303
304 #if (NGX_HAVE_F_READAHEAD)
305
306 #define NGX_HAVE_READ_AHEAD    1
307
308 #define ngx_read_ahead(fd, n)  fcntl(fd, F_READAHEAD, (int) n)
309 #define ngx_read_ahead_n      "fcntl(fd, F_READAHEAD)"
310
311 #elif (NGX_HAVE_POSIX_FADVISE)
312
313 #define NGX_HAVE_READ_AHEAD    1
314
315 ngx_int_t ngx_read_ahead(ngx_fd_t fd, size_t n);
316 #define ngx_read_ahead_n      "posix_fadvise(POSIX_FADV_SEQUENTIAL)"
317
318 #else
319
320 #define ngx_read_ahead(fd, n)  0
321 #define ngx_read_ahead_n      "ngx_read_ahead_n"
322
323 #endif
324
325
326 #if (NGX_HAVE_O_DIRECT)
327
328 ngx_int_t ngx_directio_on(ngx_fd_t fd);
329 #define ngx_directio_on_n      "fcntl(O_DIRECT)"
330
331 ngx_int_t ngx_directio_off(ngx_fd_t fd);
332 #define ngx_directio_off_n     "fcntl(!O_DIRECT)"
333
334 #elif (NGX_HAVE_F_NOCACHE)
335
336 #define ngx_directio_on(fd)    fcntl(fd, F_NOCACHE, 1)
337 #define ngx_directio_on_n     "fcntl(F_NOCACHE, 1)"
338
339 #elif (NGX_HAVE_DIRECTIO)
340
341 #define ngx_directio_on(fd)    directio(fd, DIRECTIO_ON)
342 #define ngx_directio_on_n     "directio(DIRECTIO_ON)"
343
344 #else
345
346 #define ngx_directio_on(fd)    0
347 #define ngx_directio_on_n     "ngx_directio_on_n"
348
349 #endif
350
351 size_t ngx_fs_bsize(u_char *name);
352
353
354 #if (NGX_HAVE_OPENAT)
355
356 #define ngx_openat_file(fd, name, mode, create, access)      \
357     openat(fd, (const char *) name, mode|create, access)
358
359 #define ngx_openat_file_n      "openat()"
360
361 #define ngx_file_at_info(fd, name, sb, flag)                \
362     fstatat(fd, (const char *) name, sb, flag)
363
364 #define ngx_file_at_info_n     "fstatat()"
365
366 #define NGX_AT_FDCWD          (ngx_fd_t) AT_FDCWD
367
368 #endif
369
370
371 #define ngx_stderr            STDERR_FILENO
372 #define ngx_set_stderr(fd)   dup2(fd, STDERR_FILENO)
373 #define ngx_set_stderr_n     "dup2(STDERR_FILENO)"
374

```

```
375
376 #if (NGX_HAVE_FILE_AIO)
377
378 ssize_t ngx_file_aio_read(ngx_file_t *file, u_char *buf, size_t size,
379     off_t offset, ngx_pool_t *pool);
380
381 extern ngx_uint_t  ngx_file_aio;
382
383 #endif
384
385
386 #endif /* NGX_FILES_H_INCLUDED */
```

[One Level Up](#)

[Top Level](#)

## src/os/unix/nginx\_files.c - nginx-1.7.10

### Global variables defined

- [ngx\\_file\\_aio](#)

### Functions defined

- [ngx\\_close\\_file\\_mapping](#)
- [ngx\\_close\\_glob](#)
- [ngx\\_create\\_file\\_mapping](#)
- [ngx\\_directio\\_off](#)
- [ngx\\_directio\\_on](#)
- [ngx\\_fs\\_bsize](#)
- [ngx\\_fs\\_bsize](#)
- [ngx\\_fs\\_bsize](#)
- [ngx\\_lock\\_fd](#)
- [ngx\\_open\\_dir](#)
- [ngx\\_open\\_glob](#)
- [ngx\\_open\\_tempfile](#)
- [ngx\\_read\\_ahead](#)
- [ngx\\_read\\_dir](#)
- [ngx\\_read\\_file](#)
- [ngx\\_read\\_glob](#)
- [ngx\\_set\\_file\\_time](#)
- [ngx\\_trylock\\_fd](#)
- [ngx\\_unlock\\_fd](#)
- [ngx\\_write\\_chain\\_to\\_file](#)
- [ngx\\_write\\_file](#)

### Macros defined

- [NGX\\_IOVS](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
```

```

6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10
11
12 #if (NGX_HAVE_FILE_AIO)
13
14 ngx_uint_t ngx_file_aio = 1;
15
16 #endif
17
18
19 ssize_t
20 ngx_read_file(ngx_file_t *file, u_char *buf, size_t size, off_t offset)
21 {
22     ssize_t n;
23
24     ngx_log_debug4(NGX_LOG_DEBUG_CORE, file->log, 0,
25                  "read: %d, %p, %uz, %O", file->fd, buf, size, offset);
26
27     #if (NGX_HAVE_PREAD)
28
29         n = pread(file->fd, buf, size, offset);
30
31         if (n == -1) {
32             ngx_log_error(NGX_LOG_CRIT, file->log, ngx_errno,
33                          "pread() \"%s\" failed", file->name.data);
34             return NGX_ERROR;
35         }
36
37     #else
38
39         if (file->sys_offset != offset) {
40             if (lseek(file->fd, offset, SEEK_SET) == -1) {
41                 ngx_log_error(NGX_LOG_CRIT, file->log, ngx_errno,
42                              "lseek() \"%s\" failed", file->name.data);
43                 return NGX_ERROR;
44             }
45
46             file->sys_offset = offset;
47         }
48
49         n = read(file->fd, buf, size);
50
51         if (n == -1) {
52             ngx_log_error(NGX_LOG_CRIT, file->log, ngx_errno,
53                          "read() \"%s\" failed", file->name.data);
54             return NGX_ERROR;
55         }
56
57         file->sys_offset += n;
58
59     #endif
60
61     file->offset += n;
62
63     return n;
64 }
65
66
67 ssize_t
68 ngx_write_file(ngx_file_t *file, u_char *buf, size_t size, off_t offset)
69 {
70     ssize_t n, written;
71
72     ngx_log_debug4(NGX_LOG_DEBUG_CORE, file->log, 0,
73                  "write: %d, %p, %uz, %O", file->fd, buf, size, offset);
74
75     written = 0;
76
77     #if (NGX_HAVE_PWRITE)
78
79     for ( ;; ) {
80         n = pwrite(file->fd, buf + written, size, offset);
81

```

```

82     if (n == -1) {
83         ngx_log_error(NGX_LOG_CRIT, file->log, ngx_errno,
84             "pwrite() \"%s\" failed", file->name.data);
85         return NGX_ERROR;
86     }
87
88     file->offset += n;
89     written += n;
90
91     if ((size_t) n == size) {
92         return written;
93     }
94
95     offset += n;
96     size -= n;
97 }
98
99 #else
100
101     if (file->sys_offset != offset) {
102         if (lseek(file->fd, offset, SEEK_SET) == -1) {
103             ngx_log_error(NGX_LOG_CRIT, file->log, ngx_errno,
104                 "lseek() \"%s\" failed", file->name.data);
105             return NGX_ERROR;
106         }
107
108         file->sys_offset = offset;
109     }
110
111     for ( ;; ) {
112         n = write(file->fd, buf + written, size);
113
114         if (n == -1) {
115             ngx_log_error(NGX_LOG_CRIT, file->log, ngx_errno,
116                 "write() \"%s\" failed", file->name.data);
117             return NGX_ERROR;
118         }
119
120         file->offset += n;
121         written += n;
122
123         if ((size_t) n == size) {
124             return written;
125         }
126
127         size -= n;
128     }
129 #endif
130 }
131
132
133 ngx_fd_t
134 ngx_open_tempfile(u_char *name, ngx_uint_t persistent, ngx_uint_t access)
135 {
136     ngx_fd_t fd;
137
138     fd = open((const char *) name, O_CREAT|O_EXCL|O_RDWR,
139         access ? access : 0600);
140
141     if (fd != -1 && !persistent) {
142         (void) unlink((const char *) name);
143     }
144
145     return fd;
146 }
147
148
149 #define NGX_IOVS 8
150
151 ssize_t
152 ngx_write_chain_to_file(ngx_file_t *file, ngx_chain_t *cl, off_t offset,
153     ngx_pool_t *pool)
154 {
155     u_char *prev;
156     size_t size;
157     ssize_t total, n;

```



```

158     ngx_array_t    vec;
159     struct iovec   *iov, iovs[NGX_IOVS];
160
161     /* use pwrite() if there is the only buf in a chain */
162
163     if (cl->next == NULL) {
164         return ngx_write_file(file, cl->buf->pos,
165                               (size_t) (cl->buf->last - cl->buf->pos),
166                               offset);
167     }
168
169     total = 0;
170
171     vec.elts = iovs;
172     vec.size = sizeof(struct iovec);
173     vec.nalloc = NGX_IOVS;
174     vec.pool = pool;
175
176     do {
177         prev = NULL;
178         iov = NULL;
179         size = 0;
180
181         vec.nelts = 0;
182
183         /* create the iovec and coalesce the neighbouring bufs */
184
185         while (cl && vec.nelts < IOV_MAX) {
186             if (prev == cl->buf->pos) {
187                 iov->iov_len += cl->buf->last - cl->buf->pos;
188
189             } else {
190                 iov = ngx_array_push(&vec);
191                 if (iov == NULL) {
192                     return NGX_ERROR;
193                 }
194
195                 iov->iov_base = (void *) cl->buf->pos;
196                 iov->iov_len = cl->buf->last - cl->buf->pos;
197             }
198
199             size += cl->buf->last - cl->buf->pos;
200             prev = cl->buf->last;
201             cl = cl->next;
202         }
203
204         /* use pwrite() if there is the only iovec buffer */
205
206         if (vec.nelts == 1) {
207             iov = vec.elts;
208
209             n = ngx_write_file(file, (u_char *) iov[0].iov_base,
210                               iov[0].iov_len, offset);
211
212             if (n == NGX_ERROR) {
213                 return n;
214             }
215
216             return total + n;
217         }
218
219         if (file->sys_offset != offset) {
220             if (lseek(file->fd, offset, SEEK_SET) == -1) {
221                 ngx_log_error(NGX_LOG_CRIT, file->log, ngx_errno,
222                               "lseek() \"%s\" failed", file->name.data);
223                 return NGX_ERROR;
224             }
225
226             file->sys_offset = offset;
227         }
228
229         n = writev(file->fd, vec.elts, vec.nelts);
230
231         if (n == -1) {
232             ngx_log_error(NGX_LOG_CRIT, file->log, ngx_errno,
233                           "writev() \"%s\" failed", file->name.data);

```

```

234     return NGX\_ERROR;
235 }
236
237 if ((size_t) n != size) {
238     ngx\_log\_error(NGX\_LOG\_CRIT, file->log, 0,
239         "writev() \"%s\" has written only %z of %uz",
240         file->name.data, n, size);
241     return NGX\_ERROR;
242 }
243
244 ngx\_log\_debug2(NGX\_LOG\_DEBUG\_CORE, file->log, 0,
245     "writev: %d, %z", file->fd, n);
246
247 file->sys_offset += n;
248 file->offset += n;
249 offset += n;
250 total += n;
251
252 } while (c1);
253
254 return total;
255 }
256
257
258 ngx\_int\_t
259 ngx\_set\_file\_time(u_char *name, ngx\_fd\_t fd, time_t s)
260 {
261     struct timeval tv[2];
262
263     tv[0].tv_sec = ngx\_time();
264     tv[0].tv_usec = 0;
265     tv[1].tv_sec = s;
266     tv[1].tv_usec = 0;
267
268     if (utimes((char *) name, tv) != -1) {
269         return NGX\_OK;
270     }
271
272     return NGX\_ERROR;
273 }
274
275
276 ngx\_int\_t
277 ngx\_create\_file\_mapping(ngx\_file\_mapping\_t *fm)
278 {
279     fm->fd = ngx\_open\_file(fm->name, NGX\_FILE\_RDWR, NGX\_FILE\_TRUNCATE,
280         NGX\_FILE\_DEFAULT\_ACCESS);
281     if (fm->fd == NGX\_INVALID\_FILE) {
282         ngx\_log\_error(NGX\_LOG\_CRIT, fm->log, ngx\_errno,
283             ngx\_open\_file\_n " \"%s\" failed", fm->name);
284         return NGX\_ERROR;
285     }
286
287     if (ftruncate(fm->fd, fm->size) == -1) {
288         ngx\_log\_error(NGX\_LOG\_CRIT, fm->log, ngx\_errno,
289             "ftruncate() \"%s\" failed", fm->name);
290         goto failed;
291     }
292
293     fm->addr = mmap(NULL, fm->size, PROT_READ|PROT_WRITE, MAP_SHARED,
294         fm->fd, 0);
295     if (fm->addr != MAP_FAILED) {
296         return NGX\_OK;
297     }
298
299     ngx\_log\_error(NGX\_LOG\_CRIT, fm->log, ngx\_errno,
300         "mmap(%uz) \"%s\" failed", fm->size, fm->name);
301
302 failed:
303
304     if (ngx\_close\_file(fm->fd) == NGX\_FILE\_ERROR) {
305         ngx\_log\_error(NGX\_LOG\_ALERT, fm->log, ngx\_errno,
306             ngx\_close\_file\_n " \"%s\" failed", fm->name);
307     }
308
309     return NGX\_ERROR;

```

```

310 }
311
312
313 void
314 ngx_close_file_mapping(ngx_file_mapping_t *fm)
315 {
316     if (munmap(fm->addr, fm->size) == -1) {
317         ngx_log_error(NGX_LOG_CRIT, fm->log, ngx_errno,
318             "munmap(%uz) \"%s\" failed", fm->size, fm->name);
319     }
320
321     if (ngx_close_file(fm->fd) == NGX_FILE_ERROR) {
322         ngx_log_error(NGX_LOG_ALERT, fm->log, ngx_errno,
323             ngx_close_file_n " \"%s\" failed", fm->name);
324     }
325 }
326
327
328 ngx_int_t
329 ngx_open_dir(ngx_str_t *name, ngx_dir_t *dir)
330 {
331     dir->dir = opendir((const char *) name->data);
332
333     if (dir->dir == NULL) {
334         return NGX_ERROR;
335     }
336
337     dir->valid_info = 0;
338
339     return NGX_OK;
340 }
341
342
343 ngx_int_t
344 ngx_read_dir(ngx_dir_t *dir)
345 {
346     dir->de = readdir(dir->dir);
347
348     if (dir->de) {
349 #if (NGX_HAVE_D_TYPE)
350         dir->type = dir->de->d_type;
351 #else
352         dir->type = 0;
353 #endif
354         return NGX_OK;
355     }
356
357     return NGX_ERROR;
358 }
359
360
361 ngx_int_t
362 ngx_open_glob(ngx_glob_t *gl)
363 {
364     int n;
365
366     n = glob((char *) gl->pattern, 0, NULL, &gl->pglob);
367
368     if (n == 0) {
369         return NGX_OK;
370     }
371
372 #ifdef GLOB_NOMATCH
373     if (n == GLOB_NOMATCH && gl->test) {
374         return NGX_OK;
375     }
376 #endif
377
378 #endif
379
380     return NGX_ERROR;
381 }
382
383
384 ngx_int_t
385 ngx_read_glob(ngx_glob_t *gl, ngx_str_t *name)

```

```

386 {
387     size_t count;
388
389 #ifdef GLOB_NOMATCH
390     count = (size_t) gl->pglob.gl_pathc;
391 #else
392     count = (size_t) gl->pglob.gl_matchc;
393 #endif
394
395     if (gl->n < count) {
396
397         name->len = (size_t) ngx_strlen(gl->pglob.gl_pathv[gl->n]);
398         name->data = (u_char *) gl->pglob.gl_pathv[gl->n];
399         gl->n++;
400
401         return NGX_OK;
402     }
403
404     return NGX_DONE;
405 }
406
407 void
408 ngx_close_glob(ngx_glob_t *gl)
409 {
410     globfree(&gl->pglob);
411 }
412
413
414
415 ngx_err_t
416 ngx_trylock_fd(ngx_fd_t fd)
417 {
418     struct flock fl;
419
420     ngx_memzero(&fl, sizeof(struct flock));
421     fl.l_type = F_WRLCK;
422     fl.l_whence = SEEK_SET;
423
424     if (fcntl(fd, F_SETLK, &fl) == -1) {
425         return ngx_errno;
426     }
427
428     return 0;
429 }
430
431
432 ngx_err_t
433 ngx_lock_fd(ngx_fd_t fd)
434 {
435     struct flock fl;
436
437     ngx_memzero(&fl, sizeof(struct flock));
438     fl.l_type = F_WRLCK;
439     fl.l_whence = SEEK_SET;
440
441     if (fcntl(fd, F_SETLKW, &fl) == -1) {
442         return ngx_errno;
443     }
444
445     return 0;
446 }
447
448
449 ngx_err_t
450 ngx_unlock_fd(ngx_fd_t fd)
451 {
452     struct flock fl;
453
454     ngx_memzero(&fl, sizeof(struct flock));
455     fl.l_type = F_UNLCK;
456     fl.l_whence = SEEK_SET;
457
458     if (fcntl(fd, F_SETLK, &fl) == -1) {
459         return ngx_errno;
460     }
461

```

```

462     return 0;
463 }
464
465
466 #if (NGX_HAVE_POSIX_FADVISE) && !(NGX_HAVE_F_READAHEAD)
467
468 ngx_int_t
469 ngx_read_ahead(ngx_fd_t fd, size_t n)
470 {
471     int err;
472
473     err = posix_fadvise(fd, 0, 0, POSIX_FADV_SEQUENTIAL);
474
475     if (err == 0) {
476         return 0;
477     }
478
479     ngx_set_errno(err);
480     return NGX_FILE_ERROR;
481 }
482
483 #endif
484
485
486 #if (NGX_HAVE_O_DIRECT)
487
488 ngx_int_t
489 ngx_directio_on(ngx_fd_t fd)
490 {
491     int flags;
492
493     flags = fcntl(fd, F_GETFL);
494
495     if (flags == -1) {
496         return NGX_FILE_ERROR;
497     }
498
499     return fcntl(fd, F_SETFL, flags | O_DIRECT);
500 }
501
502
503 ngx_int_t
504 ngx_directio_off(ngx_fd_t fd)
505 {
506     int flags;
507
508     flags = fcntl(fd, F_GETFL);
509
510     if (flags == -1) {
511         return NGX_FILE_ERROR;
512     }
513
514     return fcntl(fd, F_SETFL, flags & ~O_DIRECT);
515 }
516
517 #endif
518
519
520 #if (NGX_HAVE_STATFS)
521
522 size_t
523 ngx_fs_bsize(u_char *name)
524 {
525     struct statfs fs;
526
527     if (statfs((char *) name, &fs) == -1) {
528         return 512;
529     }
530
531     if ((fs.f_bsize % 512) != 0) {
532         return 512;
533     }
534
535     return (size_t) fs.f_bsize;
536 }
537

```

```
538 #elif (NGX_HAVE_STATVFS)
539
540 size_t
541 ngx_fs_bsize(u_char *name)
542 {
543     struct statvfs fs;
544
545     if (statvfs((char *) name, &fs) == -1) {
546         return 512;
547     }
548
549     if ((fs.f_frsize % 512) != 0) {
550         return 512;
551     }
552
553     return (size_t) fs.f_frsize;
554 }
555
556 #else
557
558 size_t
559 ngx_fs_bsize(u_char *name)
560 {
561     return 512;
562 }
563
564 #endif
```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_log.h - nginx-1.7.10

### Data types defined

- [ngx\\_log\\_handler\\_pt](#)
- [ngx\\_log\\_s](#)
- [ngx\\_log\\_writer\\_pt](#)

### Functions defined

- [ngx\\_write\\_stderr](#)

### Macros defined

- [NGX\\_HAVE\\_VARIADIC\\_MACROS](#)
- [NGX\\_HAVE\\_VARIADIC\\_MACROS](#)
- [NGX\\_HAVE\\_VARIADIC\\_MACROS](#)
- [NGX\\_LOG\\_ALERT](#)
- [NGX\\_LOG\\_CRIT](#)
- [NGX\\_LOG\\_DEBUG](#)
- [NGX\\_LOG\\_DEBUG\\_ALL](#)
- [NGX\\_LOG\\_DEBUG\\_ALLOC](#)
- [NGX\\_LOG\\_DEBUG\\_CONNECTION](#)
- [NGX\\_LOG\\_DEBUG\\_CORE](#)
- [NGX\\_LOG\\_DEBUG\\_EVENT](#)
- [NGX\\_LOG\\_DEBUG\\_FIRST](#)
- [NGX\\_LOG\\_DEBUG\\_HTTP](#)
- [NGX\\_LOG\\_DEBUG\\_LAST](#)
- [NGX\\_LOG\\_DEBUG\\_MAIL](#)
- [NGX\\_LOG\\_DEBUG\\_MUTEX](#)
- [NGX\\_LOG\\_DEBUG\\_MYSQL](#)
- [NGX\\_LOG\\_EMERG](#)
- [NGX\\_LOG\\_ERR](#)
- [NGX\\_LOG\\_INFO](#)
- [NGX\\_LOG\\_NOTICE](#)
- [NGX\\_LOG\\_STDERR](#)
- [NGX\\_LOG\\_WARN](#)

- [NGX\\_MAX\\_ERROR\\_STR](#)
- [\\_NGX\\_LOG\\_H\\_INCLUDED](#)
- [ngx\\_log\\_debug](#)
- [ngx\\_log\\_debug](#)
- [ngx\\_log\\_debug0](#)
- [ngx\\_log\\_debug0](#)
- [ngx\\_log\\_debug0](#)
- [ngx\\_log\\_debug1](#)
- [ngx\\_log\\_debug1](#)
- [ngx\\_log\\_debug1](#)
- [ngx\\_log\\_debug2](#)
- [ngx\\_log\\_debug2](#)
- [ngx\\_log\\_debug2](#)
- [ngx\\_log\\_debug3](#)
- [ngx\\_log\\_debug3](#)
- [ngx\\_log\\_debug3](#)
- [ngx\\_log\\_debug4](#)
- [ngx\\_log\\_debug4](#)
- [ngx\\_log\\_debug4](#)
- [ngx\\_log\\_debug5](#)
- [ngx\\_log\\_debug5](#)
- [ngx\\_log\\_debug5](#)
- [ngx\\_log\\_debug6](#)
- [ngx\\_log\\_debug6](#)
- [ngx\\_log\\_debug6](#)
- [ngx\\_log\\_debug7](#)
- [ngx\\_log\\_debug7](#)
- [ngx\\_log\\_debug7](#)
- [ngx\\_log\\_debug8](#)
- [ngx\\_log\\_debug8](#)
- [ngx\\_log\\_debug8](#)
- [ngx\\_log\\_error](#)
- [ngx\\_log\\_error](#)



## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef NGX_LOG_H_INCLUDED
9 #define NGX_LOG_H_INCLUDED
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 #define NGX_LOG_STDERR          0
17 #define NGX_LOG_EMERG          1
18 #define NGX_LOG_ALERT          2
19 #define NGX_LOG_CRIT           3
20 #define NGX_LOG_ERR            4
21 #define NGX_LOG_WARN           5
22 #define NGX_LOG_NOTICE         6
23 #define NGX_LOG_INFO           7
24 #define NGX_LOG_DEBUG          8
25
26 #define NGX_LOG_DEBUG_CORE      0x010
27 #define NGX_LOG_DEBUG_ALLOC     0x020
28 #define NGX_LOG_DEBUG_MUTEX    0x040
29 #define NGX_LOG_DEBUG_EVENT    0x080
30 #define NGX_LOG_DEBUG_HTTP     0x100
31 #define NGX_LOG_DEBUG_MAIL     0x200
32 #define NGX_LOG_DEBUG_MYSQL    0x400
33
34 /*
35  * do not forget to update debug_levels[] in src/core/nginx_log.c
36  * after the adding a new debug level
37  */
38
39 #define NGX_LOG_DEBUG_FIRST      NGX_LOG_DEBUG_CORE
40 #define NGX_LOG_DEBUG_LAST      NGX_LOG_DEBUG_MYSQL
41 #define NGX_LOG_DEBUG_CONNECTION 0x80000000
42 #define NGX_LOG_DEBUG_ALL       0x7fffffff
43
44
45 typedef u_char *(*ngx_log_handler_pt) (ngx_log_t *log, u_char *buf, size_t len);
46 typedef void (*ngx_log_writer_pt) (ngx_log_t *log, ngx_uint_t level,
47     u_char *buf, size_t len);
48
49
50 struct ngx_log_s {
51     ngx_uint_t          log_level;
52     ngx_open_file_t   *file;
53
54     ngx_atomic_uint_t  connection;
55
56     time_t              disk_full_time;
57
58     ngx_log_handler_pt handler;
59     void                *data;
60
61     ngx_log_writer_pt  writer;
62     void                *wdata;
63
64     /*
65      * we declare "action" as "char *" because the actions are usually
66      * the static strings and in the "u_char *" case we have to override
67      * their types all the time
68      */
69
70     char                *action;
71
72     ngx_log_t          *next;
73 };
```

```

74
75
76 #define NGX_MAX_ERROR_STR 2048
77
78
79 /*****/
80
81 #if (NGX_HAVE_C99_VARIADIC_MACROS)
82
83 #define NGX_HAVE_VARIADIC_MACROS 1
84
85 #define ngx_log_error(level, log, ...) \
86     if ((log)->log_level >= level) ngx_log_error_core(level, log, __VA_ARGS__)
87
88 void ngx_log_error_core(ngx_uint_t level, ngx_log_t *log, ngx_err_t err,
89     const char *fmt, ...);
90
91 #define ngx_log_debug(level, log, ...) \
92     if ((log)->log_level & level) \
93         ngx_log_error_core(NGX_LOG_DEBUG, log, __VA_ARGS__)
94
95 /*****/
96
97 #elif (NGX_HAVE_GCC_VARIADIC_MACROS)
98
99 #define NGX_HAVE_VARIADIC_MACROS 1
100
101 #define ngx_log_error(level, log, args...) \
102     if ((log)->log_level >= level) ngx_log_error_core(level, log, args)
103
104 void ngx_log_error_core(ngx_uint_t level, ngx_log_t *log, ngx_err_t err,
105     const char *fmt, ...);
106
107 #define ngx_log_debug(level, log, args...) \
108     if ((log)->log_level & level) \
109         ngx_log_error_core(NGX_LOG_DEBUG, log, args)
110
111 /*****/
112
113 #else /* NO VARIADIC MACROS */
114
115 #define NGX_HAVE_VARIADIC_MACROS 0
116
117 void ngx_cdecl ngx_log_error(ngx_uint_t level, ngx_log_t *log, ngx_err_t err,
118     const char *fmt, ...);
119 void ngx_log_error_core(ngx_uint_t level, ngx_log_t *log, ngx_err_t err,
120     const char *fmt, va_list args);
121 void ngx_cdecl ngx_log_debug_core(ngx_log_t *log, ngx_err_t err,
122     const char *fmt, ...);
123
124
125 #endif /* VARIADIC MACROS */
126
127
128 /*****/
129
130 #if (NGX_DEBUG)
131
132 #if (NGX_HAVE_VARIADIC_MACROS)
133
134 #define ngx_log_debug0(level, log, err, fmt) \
135     ngx_log_debug(level, log, err, fmt)
136
137 #define ngx_log_debug1(level, log, err, fmt, arg1) \
138     ngx_log_debug(level, log, err, fmt, arg1)
139
140 #define ngx_log_debug2(level, log, err, fmt, arg1, arg2) \
141     ngx_log_debug(level, log, err, fmt, arg1, arg2)
142
143 #define ngx_log_debug3(level, log, err, fmt, arg1, arg2, arg3) \
144     ngx_log_debug(level, log, err, fmt, arg1, arg2, arg3)
145
146 #define ngx_log_debug4(level, log, err, fmt, arg1, arg2, arg3, arg4) \
147     ngx_log_debug(level, log, err, fmt, arg1, arg2, arg3, arg4)
148
149 #define ngx_log_debug5(level, log, err, fmt, arg1, arg2, arg3, arg4, arg5) \

```

```

150     ngx_log_debug(level, log, err, fmt, arg1, arg2, arg3, arg4, arg5)
151
152 #define ngx_log_debug6(level, log, err, fmt,                                \
153     arg1, arg2, arg3, arg4, arg5, arg6)                                    \
154     ngx_log_debug(level, log, err, fmt,                                    \
155     arg1, arg2, arg3, arg4, arg5, arg6)
156
157 #define ngx_log_debug7(level, log, err, fmt,                                \
158     arg1, arg2, arg3, arg4, arg5, arg6, arg7)                            \
159     ngx_log_debug(level, log, err, fmt,                                    \
160     arg1, arg2, arg3, arg4, arg5, arg6, arg7)
161
162 #define ngx_log_debug8(level, log, err, fmt,                                \
163     arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8)                      \
164     ngx_log_debug(level, log, err, fmt,                                    \
165     arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8)
166
167
168 #else /* NO VARIADIC MACROS */
169
170 #define ngx_log_debug0(level, log, err, fmt)                                \
171     if ((log)->log_level & level)                                        \
172     ngx_log_debug_core(log, err, fmt)
173
174 #define ngx_log_debug1(level, log, err, fmt, arg1)                        \
175     if ((log)->log_level & level)                                        \
176     ngx_log_debug_core(log, err, fmt, arg1)
177
178 #define ngx_log_debug2(level, log, err, fmt, arg1, arg2)                  \
179     if ((log)->log_level & level)                                        \
180     ngx_log_debug_core(log, err, fmt, arg1, arg2)
181
182 #define ngx_log_debug3(level, log, err, fmt, arg1, arg2, arg3)            \
183     if ((log)->log_level & level)                                        \
184     ngx_log_debug_core(log, err, fmt, arg1, arg2, arg3)
185
186 #define ngx_log_debug4(level, log, err, fmt, arg1, arg2, arg3, arg4)      \
187     if ((log)->log_level & level)                                        \
188     ngx_log_debug_core(log, err, fmt, arg1, arg2, arg3, arg4)
189
190 #define ngx_log_debug5(level, log, err, fmt, arg1, arg2, arg3, arg4, arg5) \
191     if ((log)->log_level & level)                                        \
192     ngx_log_debug_core(log, err, fmt, arg1, arg2, arg3, arg4, arg5)
193
194 #define ngx_log_debug6(level, log, err, fmt,                                \
195     arg1, arg2, arg3, arg4, arg5, arg6)                                    \
196     if ((log)->log_level & level)                                        \
197     ngx_log_debug_core(log, err, fmt, arg1, arg2, arg3, arg4, arg5, arg6)
198
199 #define ngx_log_debug7(level, log, err, fmt,                                \
200     arg1, arg2, arg3, arg4, arg5, arg6, arg7)                            \
201     if ((log)->log_level & level)                                        \
202     ngx_log_debug_core(log, err, fmt,                                    \
203     arg1, arg2, arg3, arg4, arg5, arg6, arg7)
204
205 #define ngx_log_debug8(level, log, err, fmt,                                \
206     arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8)                      \
207     if ((log)->log_level & level)                                        \
208     ngx_log_debug_core(log, err, fmt,                                    \
209     arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8)
210
211 #endif
212
213 #else /* NO NGX_DEBUG */
214
215 #define ngx_log_debug0(level, log, err, fmt)
216 #define ngx_log_debug1(level, log, err, fmt, arg1)
217 #define ngx_log_debug2(level, log, err, fmt, arg1, arg2)
218 #define ngx_log_debug3(level, log, err, fmt, arg1, arg2, arg3)
219 #define ngx_log_debug4(level, log, err, fmt, arg1, arg2, arg3, arg4)
220 #define ngx_log_debug5(level, log, err, fmt, arg1, arg2, arg3, arg4, arg5)
221 #define ngx_log_debug6(level, log, err, fmt, arg1, arg2, arg3, arg4, arg5, arg6)
222 #define ngx_log_debug7(level, log, err, fmt, arg1, arg2, arg3, arg4, arg5, \
223     arg6, arg7)
224 #define ngx_log_debug8(level, log, err, fmt, arg1, arg2, arg3, arg4, arg5, \
225     arg6, arg7, arg8)

```

```

226
227 #endif
228
229 /******
230
231 ngx\_log\_t *ngx\_log\_init(u_char *prefix);
232 void ngx\_cdecl ngx\_log\_abort(ngx\_err\_t err, const char *fmt, ...);
233 void ngx\_cdecl ngx\_log\_stderr(ngx\_err\_t err, const char *fmt, ...);
234 u_char *ngx\_log\_errno(u_char *buf, u_char *last, ngx\_err\_t err);
235 ngx\_int\_t ngx\_log\_open\_default(ngx\_cycle\_t *cycle);
236 ngx\_int\_t ngx\_log\_redirect\_stderr(ngx\_cycle\_t *cycle);
237 ngx\_log\_t *ngx\_log\_get\_file\_log(ngx\_log\_t *head);
238 char *ngx\_log\_set\_log(ngx\_conf\_t *cf, ngx\_log\_t **head);
239
240
241 /*
242 * ngx\_write\_stderr\(\) cannot be implemented as macro, since
243 * MSVC does not allow to use #ifdef inside macro parameters.
244 *
245 * ngx\_write\_fd\(\) is used instead of ngx\_write\_console\(\), since
246 * CharToOemBuff\(\) inside ngx\_write\_console\(\) cannot be used with
247 * read only buffer as destination and CharToOemBuff\(\) is not needed
248 * for ngx\_write\_stderr\(\) anyway.
249 */
250 static ngx\_inline void
251 ngx\_write\_stderr(char *text)
252 {
253     (void) ngx\_write\_fd(ngx\_stderr, text, ngx\_strlen(text));
254 }
255
256
257 extern ngx\_module\_t ngx\_errlog\_module;
258 extern ngx\_uint\_t ngx\_use\_stderr;
259
260
261 #endif /* \_NGX\_LOG\_H\_INCLUDED */

```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_log.c - nginx-1.7.10

### Global variables defined

- [debug\\_levels](#)
- [err\\_levels](#)
- [ngx\\_errlog\\_commands](#)
- [ngx\\_errlog\\_module](#)
- [ngx\\_errlog\\_module\\_ctx](#)
- [ngx\\_log](#)
- [ngx\\_log\\_file](#)
- [ngx\\_use\\_stderr](#)

### Functions defined

- [ngx\\_error\\_log](#)
- [ngx\\_log\\_abort](#)
- [ngx\\_log\\_debug\\_core](#)
- [ngx\\_log\\_errno](#)
- [ngx\\_log\\_error](#)
- [ngx\\_log\\_error\\_core](#)
- [ngx\\_log\\_get\\_file\\_log](#)
- [ngx\\_log\\_init](#)
- [ngx\\_log\\_insert](#)
- [ngx\\_log\\_open\\_default](#)
- [ngx\\_log\\_redirect\\_stderr](#)
- [ngx\\_log\\_set\\_levels](#)
- [ngx\\_log\\_set\\_log](#)
- [ngx\\_log\\_stderr](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10
11
```

```

12 static char *ngx_error_log(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
13 static char *ngx_log_set_levels(ngx_conf_t *cf, ngx_log_t *log);
14 static void ngx_log_insert(ngx_log_t *log, ngx_log_t *new_log);
15
16
17 static ngx_command_t ngx_errlog_commands[] = {
18     {ngx_string("error_log"),
19      NGX_MAIN_CONF|NGX_CONF_1MORE,
20      ngx_error_log,
21      0,
22      0,
23      NULL},
24     ngx_null_command
25 };
26
27
28
29
30 static ngx_core_module_t ngx_errlog_module_ctx = {
31     ngx_string("errlog"),
32     NULL,
33     NULL
34 };
35
36
37 ngx_module_t ngx_errlog_module = {
38     NGX_MODULE_V1,
39     &ngx_errlog_module_ctx,          /* module context */
40     ngx_errlog_commands,           /* module directives */
41     NGX_CORE_MODULE,              /* module type */
42     NULL,                          /* init master */
43     NULL,                          /* init module */
44     NULL,                          /* init process */
45     NULL,                          /* init thread */
46     NULL,                          /* exit thread */
47     NULL,                          /* exit process */
48     NULL,                          /* exit master */
49     NGX_MODULE_V1_PADDING
50 };
51
52
53 static ngx_log_t      ngx_log;
54 static ngx_open_file_t ngx_log_file;
55 ngx_uint_t          ngx_use_stderr = 1;
56
57
58 static ngx_str_t err_levels[] = {
59     ngx_null_string,
60     ngx_string("emerg"),
61     ngx_string("alert"),
62     ngx_string("crit"),
63     ngx_string("error"),
64     ngx_string("warn"),
65     ngx_string("notice"),
66     ngx_string("info"),
67     ngx_string("debug")
68 };
69
70 static const char *debug_levels[] = {
71     "debug_core", "debug_alloc", "debug_mutex", "debug_event",
72     "debug_http", "debug_mail", "debug_mysql"
73 };
74
75
76 #if (NGX_HAVE_VARIADIC_MACROS)
77
78 void
79 ngx_log_error_core(ngx_uint_t level, ngx_log_t *log, ngx_err_t err,
80                  const char *fmt, ...)
81
82 #else
83
84 void
85 ngx_log_error_core(ngx_uint_t level, ngx_log_t *log, ngx_err_t err,
86                  const char *fmt, va_list args)
87

```

```

88 #endif
89 {
90 #if (NGX_HAVE_VARIADIC_MACROS)
91     va_list     args;
92 #endif
93     u_char      *p, *last, *msg;
94     ssize_t     n;
95     ngx_uint_t  wrote_stderr, debug_connection;
96     u_char      errstr[NGX_MAX_ERROR_STR];
97
98     last = errstr + NGX_MAX_ERROR_STR;
99
100     ngx_memcpy(errstr, ngx_cached_err_log_time.data,
101               ngx_cached_err_log_time.len);
102
103     p = errstr + ngx_cached_err_log_time.len;
104
105     p = ngx_slprintf(p, last, " [%V] ", &err_levels[level]);
106
107     /* pid#tid */
108     p = ngx_slprintf(p, last, "%P#" NGX_TID_T_FMT ": ",
109                     ngx_log_pid, ngx_log_tid);
110
111     if (log->connection) {
112         p = ngx_slprintf(p, last, "%uA ", log->connection);
113     }
114
115     msg = p;
116
117 #if (NGX_HAVE_VARIADIC_MACROS)
118     va_start(args, fmt);
119     p = ngx_vslprintf(p, last, fmt, args);
120     va_end(args);
121 #else
122
123     p = ngx_vslprintf(p, last, fmt, args);
124 #endif
125
126 #endif
127
128     if (err) {
129         p = ngx_log_errno(p, last, err);
130     }
131
132     if (level != NGX_LOG_DEBUG && log->handler) {
133         p = log->handler(log, p, last - p);
134     }
135
136     if (p > last - NGX_LINEFEED_SIZE) {
137         p = last - NGX_LINEFEED_SIZE;
138     }
139
140     ngx_linefeed(p);
141
142     wrote_stderr = 0;
143     debug_connection = (log->log_level & NGX_LOG_DEBUG_CONNECTION) != 0;
144
145     while (log) {
146         if (log->log_level < level && !debug_connection) {
147             break;
148         }
149
150         if (log->writer) {
151             log->writer(log, level, errstr, p - errstr);
152             goto next;
153         }
154
155         if (ngx_time() == log->disk_full_time) {
156             /*
157              * on FreeBSD writing to a full filesystem with enabled softupdates
158              * may block process for much longer time than writing to non-full
159              * filesystem, so we skip writing to a log for one second
160              */

```

```

164         goto next;
165     }
166
167     n = ngx_write_fd(log->file->fd, errstr, p - errstr);
168
169     if (n == -1 && ngx_errno == NGX_ENOSPC) {
170         log->disk_full_time = ngx_time();
171     }
172
173     if (log->file->fd == ngx_stderr) {
174         wrote_stderr = 1;
175     }
176
177     next:
178     log = log->next;
179 }
180
181 if (!ngx_use_stderr
182     || level > NGX_LOG_WARN
183     || wrote_stderr)
184 {
185     return;
186 }
187
188 msg -= (7 + err_levels[level].len + 3);
189
190 (void) ngx_sprintf(msg, "nginx: [%V] ", &err_levels[level]);
191
192 (void) ngx_write_console(ngx_stderr, msg, p - msg);
193 }
194
195 #if !(NGX_HAVE_VARIADIC_MACROS)
196
197 void ngx_cdecl
198 ngx_log_error(ngx_uint_t level, ngx_log_t *log, ngx_err_t err,
199     const char *fmt, ...)
200 {
201     va_list args;
202
203     if (log->log_level >= level) {
204         va_start(args, fmt);
205         ngx_log_error_core(level, log, err, fmt, args);
206         va_end(args);
207     }
208 }
209
210 void ngx_cdecl
211 ngx_log_debug_core(ngx_log_t *log, ngx_err_t err, const char *fmt, ...)
212 {
213     va_list args;
214
215     va_start(args, fmt);
216     ngx_log_error_core(NGX_LOG_DEBUG, log, err, fmt, args);
217     va_end(args);
218 }
219
220 #endif
221
222 void ngx_cdecl
223 ngx_log_abort(ngx_err_t err, const char *fmt, ...)
224 {
225     u_char *p;
226     va_list args;
227     u_char errstr[NGX_MAX_CONF_ERRSTR];
228
229     va_start(args, fmt);
230     p = ngx_vsprintf(errstr, sizeof(errstr) - 1, fmt, args);
231     va_end(args);
232
233     ngx_log_error(NGX_LOG_ALERT, ngx_cycle->log, err,
234         "%*s", p - errstr, errstr);

```



```

240 }
241
242
243 void ngx_cdecl
244 ngx_log_stderr(ngx_err_t err, const char *fmt, ...)
245 {
246     u_char *p, *last;
247     va_list args;
248     u_char errstr[NGX_MAX_ERROR_STR];
249
250     last = errstr + NGX_MAX_ERROR_STR;
251     p = errstr + 7;
252
253     ngx_memcpy(errstr, "nginx: ", 7);
254
255     va_start(args, fmt);
256     p = ngx_vslprintf(p, last, fmt, args);
257     va_end(args);
258
259     if (err) {
260         p = ngx_log_errno(p, last, err);
261     }
262
263     if (p > last - NGX_LINEFEED_SIZE) {
264         p = last - NGX_LINEFEED_SIZE;
265     }
266
267     ngx_linefeed(p);
268
269     (void) ngx_write_console(ngx_stderr, errstr, p - errstr);
270 }
271
272
273 u_char *
274 ngx_log_errno(u_char *buf, u_char *last, ngx_err_t err)
275 {
276     if (buf > last - 50) {
277
278         /* Leave a space for an error code */
279
280         buf = last - 50;
281         *buf++ = '.';
282         *buf++ = '.';
283         *buf++ = '.';
284     }
285
286     #if (NGX_WIN32)
287     buf = ngx_slprintf(buf, last, ((unsigned) err < 0x80000000)
288         ? "%d: " : "%Xd: ", err);
289     #else
290     buf = ngx_slprintf(buf, last, "%d: ", err);
291     #endif
292
293     buf = ngx_strerror(err, buf, last - buf);
294
295     if (buf < last) {
296         *buf++ = ' ';
297     }
298
299     return buf;
300 }
301
302
303 ngx_log_t *
304 ngx_log_init(u_char *prefix)
305 {
306     u_char *p, *name;
307     size_t nlen, plen;
308
309     ngx_log.file = &ngx_log_file;
310     ngx_log.log_level = NGX_LOG_NOTICE;
311
312     name = (u_char *) NGX_ERROR_LOG_PATH;
313
314     /*
315     * we use ngx_strlen() here since BCC warns about

```

```

316     * condition is always false and unreachable code
317     */
318
319     nlen = ngx_strlen(name);
320
321     if (nlen == 0) {
322         ngx_log_file.fd = ngx_stderr;
323         return &ngx_log;
324     }
325
326     p = NULL;
327
328     #if (NGX_WIN32)
329         if (name[1] != ':') {
330     #else
331         if (name[0] != '/') {
332     #endif
333
334         if (prefix) {
335             plen = ngx_strlen(prefix);
336
337         } else {
338     #ifdef NGX_PREFIX
339             prefix = (u_char *) NGX_PREFIX;
340             plen = ngx_strlen(prefix);
341     #else
342             plen = 0;
343     #endif
344         }
345
346         if (plen) {
347             name = malloc(plen + nlen + 2);
348             if (name == NULL) {
349                 return NULL;
350             }
351
352             p = ngx_cpymem(name, prefix, plen);
353
354             if (!ngx_path_separator(*(p - 1))) {
355                 *p++ = '/';
356             }
357
358             ngx_cpystrn(p, (u_char *) NGX_ERROR_LOG_PATH, nlen + 1);
359
360             p = name;
361         }
362     }
363
364     ngx_log_file.fd = ngx_open_file(name, NGX_FILE_APPEND,
365                                     NGX_FILE_CREATE_OR_OPEN,
366                                     NGX_FILE_DEFAULT_ACCESS);
367
368     if (ngx_log_file.fd == NGX_INVALID_FILE) {
369         ngx_log_stderr(ngx_errno,
370                       "[alert] could not open error log file: "
371                       ngx_open_file_n " \"%s\" failed", name);
372     #if (NGX_WIN32)
373         ngx_event_log(ngx_errno,
374                      "could not open error log file: "
375                      ngx_open_file_n " \"%s\" failed", name);
376     #endif
377
378     ngx_log_file.fd = ngx_stderr;
379 }
380
381 if (p) {
382     ngx_free(p);
383 }
384
385 return &ngx_log;
386 }
387
388
389 ngx_int_t
390 ngx_log_open_default(ngx_cycle_t *cycle)
391 {

```

```

392     ngx_log_t      *log;
393     static ngx_str_t  error_log = ngx_string(NGX_ERROR_LOG_PATH);
394
395     if (ngx_log_get_file_log(&cycle->new_log) != NULL) {
396         return NGX_OK;
397     }
398
399     if (cycle->new_log.log_level != 0) {
400         /* there are some error logs, but no files */
401
402         log = ngx_palloc(cycle->pool, sizeof(ngx_log_t));
403         if (log == NULL) {
404             return NGX_ERROR;
405         }
406
407     } else {
408         /* no error logs at all */
409         log = &cycle->new_log;
410     }
411
412     log->log_level = NGX_LOG_ERR;
413
414     log->file = ngx_conf_open_file(cycle, &error_log);
415     if (log->file == NULL) {
416         return NGX_ERROR;
417     }
418
419     if (log != &cycle->new_log) {
420         ngx_log_insert(&cycle->new_log, log);
421     }
422
423     return NGX_OK;
424 }
425
426
427 ngx_int_t
428 ngx_log_redirect_stderr(ngx_cycle_t *cycle)
429 {
430     ngx_fd_t  fd;
431
432     if (cycle->log_use_stderr) {
433         return NGX_OK;
434     }
435
436     /* file log always exists when we are called */
437     fd = ngx_log_get_file_log(cycle->log)->file->fd;
438
439     if (fd != ngx_stderr) {
440         if (ngx_set_stderr(fd) == NGX_FILE_ERROR) {
441             ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
442                 ngx_set_stderr_n " failed");
443
444             return NGX_ERROR;
445         }
446     }
447
448     return NGX_OK;
449 }
450
451
452 ngx_log_t *
453 ngx_log_get_file_log(ngx_log_t *head)
454 {
455     ngx_log_t  *log;
456
457     for (log = head; log; log = log->next) {
458         if (log->file != NULL) {
459             return log;
460         }
461     }
462
463     return NULL;
464 }
465
466
467 static char *

```

```

468 ngx_log_set_levels(ngx_conf_t *cf, ngx_log_t *log)
469 {
470     ngx_uint_t  i, n, d, found;
471     ngx_str_t   *value;
472
473     if (cf->args->nelts == 2) {
474         log->log_level = NGX_LOG_ERR;
475         return NGX_CONF_OK;
476     }
477
478     value = cf->args->elts;
479
480     for (i = 2; i < cf->args->nelts; i++) {
481         found = 0;
482
483         for (n = 1; n <= NGX_LOG_DEBUG; n++) {
484             if (ngx_strcmp(value[i].data, err_levels[n].data) == 0) {
485
486                 if (log->log_level != 0) {
487                     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
488                         "duplicate log level \"%V\"",
489                         &value[i]);
490                     return NGX_CONF_ERROR;
491                 }
492
493                 log->log_level = n;
494                 found = 1;
495                 break;
496             }
497         }
498
499         for (n = 0, d = NGX_LOG_DEBUG_FIRST; d <= NGX_LOG_DEBUG_LAST; d <= 1) {
500             if (ngx_strcmp(value[i].data, debug_levels[n++]) == 0) {
501                 if (log->log_level & ~NGX_LOG_DEBUG_ALL) {
502                     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
503                         "invalid log level \"%V\"",
504                         &value[i]);
505                     return NGX_CONF_ERROR;
506                 }
507
508                 log->log_level |= d;
509                 found = 1;
510                 break;
511             }
512         }
513
514
515         if (!found) {
516             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
517                 "invalid log level \"%V\"", &value[i]);
518             return NGX_CONF_ERROR;
519         }
520     }
521
522     if (log->log_level == NGX_LOG_DEBUG) {
523         log->log_level = NGX_LOG_DEBUG_ALL;
524     }
525
526     return NGX_CONF_OK;
527 }
528
529
530 static char *
531 ngx_error_log(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
532 {
533     ngx_log_t   *dummy;
534
535     dummy = &cf->cycle->new_log;
536
537     return ngx_log_set_log(cf, &dummy);
538 }
539
540
541 char *
542 ngx_log_set_log(ngx_conf_t *cf, ngx_log_t **head)
543 {

```

```

544     ngx_log_t      *new_log;
545     ngx_str_t      *value, name;
546     ngx_syslog_peer_t *peer;
547
548     if (*head != NULL && (*head)->log_level == 0) {
549         new_log = *head;
550
551     } else {
552
553         new_log = ngx_palloc(cf->pool, sizeof(ngx_log_t));
554         if (new_log == NULL) {
555             return NGX_CONF_ERROR;
556         }
557
558         if (*head == NULL) {
559             *head = new_log;
560         }
561     }
562
563     value = cf->args->elts;
564
565     if (ngx_strcmp(value[1].data, "stderr") == 0) {
566         ngx_str_null(&name);
567         cf->cycle->log_use_stderr = 1;
568
569         new_log->file = ngx_conf_open_file(cf->cycle, &name);
570         if (new_log->file == NULL) {
571             return NGX_CONF_ERROR;
572         }
573
574     } else if (ngx_strncmp(value[1].data, "syslog:", 7) == 0) {
575         peer = ngx_palloc(cf->pool, sizeof(ngx_syslog_peer_t));
576         if (peer == NULL) {
577             return NGX_CONF_ERROR;
578         }
579
580         if (ngx_syslog_process_conf(cf, peer) != NGX_CONF_OK) {
581             return NGX_CONF_ERROR;
582         }
583
584         new_log->writer = ngx_syslog_writer;
585         new_log->wdata = peer;
586
587     } else {
588         new_log->file = ngx_conf_open_file(cf->cycle, &value[1]);
589         if (new_log->file == NULL) {
590             return NGX_CONF_ERROR;
591         }
592     }
593 }
594
595 if (ngx_log_set_levels(cf, new_log) != NGX_CONF_OK) {
596     return NGX_CONF_ERROR;
597 }
598
599 if (*head != new_log) {
600     ngx_log_insert(*head, new_log);
601 }
602
603 return NGX_CONF_OK;
604 }
605
606
607 static void
608 ngx_log_insert(ngx_log_t *log, ngx_log_t *new_log)
609 {
610     ngx_log_t tmp;
611
612     if (new_log->log_level > log->log_level) {
613
614         /*
615          * list head address is permanent, insert new log after
616          * head and swap its contents with head
617          */
618
619         tmp = *log;

```

```
620     *log = *new_log;
621     *new_log = tmp;
622
623     log->next = new_log;
624     return;
625 }
626
627 while (log->next) {
628     if (new_log->log_level > log->next->log_level) {
629         new_log->next = log->next;
630         log->next = new_log;
631         return;
632     }
633
634     log = log->next;
635 }
636
637 log->next = new_log;
638 }
```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_string.h - nginx-1.7.10

### Data types defined

- [ngx\\_keyval\\_t](#)
- [ngx\\_str\\_node\\_t](#)
- [ngx\\_str\\_t](#)
- [ngx\\_variable\\_value\\_t](#)

### Functions defined

- [ngx\\_copy](#)
- [ngx\\_strlchr](#)

### Macros defined

- [NGX\\_ESCAPE\\_ARGS](#)
- [NGX\\_ESCAPE\\_HTML](#)
- [NGX\\_ESCAPE\\_MAIL\\_AUTH](#)
- [NGX\\_ESCAPE\\_MEMCACHED](#)
- [NGX\\_ESCAPE\\_REFRESH](#)
- [NGX\\_ESCAPE\\_URI](#)
- [NGX\\_ESCAPE\\_URI\\_COMPONENT](#)
- [NGX\\_UNESCAPE\\_REDIRECT](#)
- [NGX\\_UNESCAPE\\_URI](#)
- [\\_NGX\\_STRING\\_H\\_INCLUDED\\_](#)
- [ngx\\_base64\\_decoded\\_length](#)
- [ngx\\_base64\\_encoded\\_length](#)
- [ngx\\_copy](#)
- [ngx\\_cpymem](#)
- [ngx\\_cpymem](#)
- [ngx\\_memcmp](#)
- [ngx\\_memcpy](#)
- [ngx\\_memmove](#)
- [ngx\\_memset](#)
- [ngx\\_memzero](#)
- [ngx\\_movemem](#)

- [ngx\\_null\\_string](#)
- [ngx\\_qsort](#)
- [ngx\\_str\\_null](#)
- [ngx\\_str\\_set](#)
- [ngx\\_strchr](#)
- [ngx\\_strcmp](#)
- [ngx\\_string](#)
- [ngx\\_strlen](#)
- [ngx\\_strncmp](#)
- [ngx\\_strstr](#)
- [ngx\\_tolower](#)
- [ngx\\_toupper](#)
- [ngx\\_value](#)
- [ngx\\_value\\_helper](#)
- [ngx\\_vsnprintf](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_STRING_H_INCLUDED_
9 #define _NGX_STRING_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 typedef struct {
17     size_t    len;
18     u_char    *data;
19 } ngx_str_t;
20
21
22 typedef struct {
23     ngx_str_t  key;
24     ngx_str_t  value;
25 } ngx_keyval_t;
26
27
28 typedef struct {
29     unsigned   len:28;
30
31     unsigned   valid:1;
32     unsigned   no_cacheable:1;
33     unsigned   not_found:1;
34     unsigned   escape:1;
35
36     u_char     *data;
37 } ngx_variable_value_t;
38
```



```

39
40 #define ngx_string(str)      { sizeof(str) - 1, (u_char *) str }
41 #define ngx_null_string    { 0, NULL }
42 #define ngx_str_set(str, text) \
43     (str)->len = sizeof(text) - 1; (str)->data = (u_char *) text
44 #define ngx_str_null(str)   (str)->len = 0; (str)->data = NULL
45
46
47 #define ngx_tolower(c)      (u_char) ((c >= 'A' && c <= 'Z') ? (c | 0x20) : c)
48 #define ngx_toupper(c)     (u_char) ((c >= 'a' && c <= 'z') ? (c & ~0x20) : c)
49
50 void ngx_strlow(u_char *dst, u_char *src, size_t n);
51
52
53 #define ngx_strncmp(s1, s2, n)  strncmp((const char *) s1, (const char *) s2, n)
54
55
56 /* msvc and icc7 compile strcmp() to inline loop */
57 #define ngx_strcmp(s1, s2)  strcmp((const char *) s1, (const char *) s2)
58
59
60 #define ngx_strstr(s1, s2)  strstr((const char *) s1, (const char *) s2)
61 #define ngx_strlen(s)      strlen((const char *) s)
62
63 #define ngx_strchr(s1, c)   strchr((const char *) s1, (int) c)
64
65 static ngx_inline u_char *
66 ngx_strlchr(u_char *p, u_char *last, u_char c)
67 {
68     while (p < last) {
69         if (*p == c) {
70             return p;
71         }
72     }
73     p++;
74 }
75
76
77     return NULL;
78 }
79
80
81 /*
82 * msvc and icc7 compile memset() to the inline "rep stos"
83 * while ZeroMemory() and bzero() are the calls.
84 * icc7 may also inline several mov's of a zeroed register for small blocks.
85 */
86 #define ngx_memzero(buf, n)      (void) memset(buf, 0, n)
87 #define ngx_memset(buf, c, n)   (void) memset(buf, c, n)
88
89
90 #if (NGX_MEMCPY_LIMIT)
91
92 void *ngx_memcpy(void *dst, const void *src, size_t n);
93 #define ngx_cpymem(dst, src, n)  (((u_char *) ngx_memcpy(dst, src, n)) + (n))
94
95 #else
96
97 /*
98 * gcc3, msvc, and icc7 compile memcpy() to the inline "rep movs".
99 * gcc3 compiles memcpy(d, s, 4) to the inline "mov"es.
100 * icc8 compile memcpy(d, s, 4) to the inline "mov"es or XMM moves.
101 */
102 #define ngx_memcpy(dst, src, n)  (void) memcpy(dst, src, n)
103 #define ngx_cpymem(dst, src, n)  (((u_char *) memcpy(dst, src, n)) + (n))
104
105 #endif
106
107
108 #if ( __INTEL_COMPILER >= 800 )
109
110 /*
111 * the simple inline cycle copies the variable length strings up to 16
112 * bytes faster than icc8 autodetecting _intel_fast_memcpy()
113 */
114

```

```

115 static ngx_inline u_char *
116 ngx_copy(u_char *dst, u_char *src, size_t len)
117 {
118     if (len < 17) {
119
120         while (len) {
121             *dst++ = *src++;
122             len--;
123         }
124
125         return dst;
126
127     } else {
128         return ngx_cpymem(dst, src, len);
129     }
130 }
131
132 #else
133
134 #define ngx_copy                ngx_cpymem
135
136 #endif
137
138
139 #define ngx_memmove(dst, src, n)  (void) memmove(dst, src, n)
140 #define ngx_movemem(dst, src, n) (((u_char *) memmove(dst, src, n)) + (n))
141
142
143 /* msvc and icc7 compile memcmp() to the inline loop */
144 #define ngx_memcmp(s1, s2, n) memcmp((const char *) s1, (const char *) s2, n)
145
146
147 u_char *ngx_cpystrn(u_char *dst, u_char *src, size_t n);
148 u_char *ngx_pstrdup(ngx_pool_t *pool, ngx_str_t *src);
149 u_char * ngx_cdecl ngx_sprintf(u_char *buf, const char *fmt, ...);
150 u_char * ngx_cdecl ngx_snprintf(u_char *buf, size_t max, const char *fmt, ...);
151 u_char * ngx_cdecl ngx_slprintf(u_char *buf, u_char *last, const char *fmt,
152     ...);
153 u_char *ngx_vslprintf(u_char *buf, u_char *last, const char *fmt, va_list args);
154 #define ngx_vsnprintf(buf, max, fmt, args) \
155     ngx_vslprintf(buf, buf + (max), fmt, args)
156
157 ngx_int_t ngx_strcasecmp(u_char *s1, u_char *s2);
158 ngx_int_t ngx_strncasecmp(u_char *s1, u_char *s2, size_t n);
159
160 u_char *ngx_strnstr(u_char *s1, char *s2, size_t n);
161
162 u_char *ngx_strstrn(u_char *s1, char *s2, size_t n);
163 u_char *ngx_strcasestrn(u_char *s1, char *s2, size_t n);
164 u_char *ngx_strlcasestrn(u_char *s1, u_char *last, u_char *s2, size_t n);
165
166 ngx_int_t ngx_rstrncmp(u_char *s1, u_char *s2, size_t n);
167 ngx_int_t ngx_rstrncasecmp(u_char *s1, u_char *s2, size_t n);
168 ngx_int_t ngx_memn2cmp(u_char *s1, u_char *s2, size_t n1, size_t n2);
169 ngx_int_t ngx_dns_strcmp(u_char *s1, u_char *s2);
170 ngx_int_t ngx_filename_cmp(u_char *s1, u_char *s2, size_t n);
171
172 ngx_int_t ngx_atoi(u_char *line, size_t n);
173 ngx_int_t ngx_atofp(u_char *line, size_t n, size_t point);
174 ssize_t ngx_atosz(u_char *line, size_t n);
175 off_t ngx_atoof(u_char *line, size_t n);
176 time_t ngx_atotm(u_char *line, size_t n);
177 ngx_int_t ngx_hextoi(u_char *line, size_t n);
178
179 u_char *ngx_hex_dump(u_char *dst, u_char *src, size_t len);
180
181
182 #define ngx_base64_encoded_length(len) (((len + 2) / 3) * 4)
183 #define ngx_base64_decoded_length(len) (((len + 3) / 4) * 3)
184
185 void ngx_encode_base64(ngx_str_t *dst, ngx_str_t *src);
186 void ngx_encode_base64url(ngx_str_t *dst, ngx_str_t *src);
187 ngx_int_t ngx_decode_base64(ngx_str_t *dst, ngx_str_t *src);
188 ngx_int_t ngx_decode_base64url(ngx_str_t *dst, ngx_str_t *src);
189
190 uint32_t ngx_utf8_decode(u_char **p, size_t n);

```

```

191 size_t ngx_utf8_length(u_char *p, size_t n);
192 u_char *ngx_utf8_cpystri(u_char *dst, u_char *src, size_t n, size_t len);
193
194
195 #define NGX_ESCAPE_URI          0
196 #define NGX_ESCAPE_ARGS        1
197 #define NGX_ESCAPE_URI_COMPONENT 2
198 #define NGX_ESCAPE_HTML        3
199 #define NGX_ESCAPE_REFRESH      4
200 #define NGX_ESCAPE_MEMCACHED    5
201 #define NGX_ESCAPE_MAIL_AUTH    6
202
203 #define NGX_UNESCAPE_URI        1
204 #define NGX_UNESCAPE_REDIRECT   2
205
206 uintptr_t ngx_escape_uri(u_char *dst, u_char *src, size_t size,
207     ngx_uint_t type);
208 void ngx_unescape_uri(u_char **dst, u_char **src, size_t size, ngx_uint_t type);
209 uintptr_t ngx_escape_html(u_char *dst, u_char *src, size_t size);
210 uintptr_t ngx_escape_json(u_char *dst, u_char *src, size_t size);
211
212
213 typedef struct {
214     ngx_rbtree_node_t    node;
215     ngx_str_t            str;
216 } ngx_str_node_t;
217
218
219 void ngx_str_rbtree_insert_value(ngx_rbtree_node_t *temp,
220     ngx_rbtree_node_t *node, ngx_rbtree_node_t *sentinel);
221 ngx_str_node_t *ngx_str_rbtree_lookup(ngx_rbtree_t *rbtree, ngx_str_t *name,
222     uint32_t hash);
223
224
225 void ngx_sort(void *base, size_t n, size_t size,
226     ngx_int_t (*cmp)(const void *, const void *));
227 #define ngx_qsort          qsort
228
229
230 #define ngx_value_helper(n)    #n
231 #define ngx_value(n)          ngx_value_helper(n)
232
233
234 #endif /* NGX_STRING_H_INCLUDED */

```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_string.c - nginx-1.7.10

### Functions defined

- [ngx\\_atofp](#)
- [ngx\\_atoi](#)
- [ngx\\_atoof](#)
- [ngx\\_atosz](#)
- [ngx\\_atotm](#)
- [ngx\\_cpystn](#)
- [ngx\\_decode\\_base64](#)
- [ngx\\_decode\\_base64\\_internal](#)
- [ngx\\_decode\\_base64url](#)
- [ngx\\_dns\\_strcmp](#)
- [ngx\\_encode\\_base64](#)
- [ngx\\_encode\\_base64\\_internal](#)
- [ngx\\_encode\\_base64url](#)
- [ngx\\_escape\\_html](#)
- [ngx\\_escape\\_json](#)
- [ngx\\_escape\\_uri](#)
- [ngx\\_filename\\_cmp](#)
- [ngx\\_hex\\_dump](#)
- [ngx\\_hextoi](#)
- [ngx\\_memcpy](#)
- [ngx\\_memn2cmp](#)
- [ngx\\_pstrdup](#)
- [ngx\\_rstrncasecmp](#)
- [ngx\\_rstrncmp](#)
- [ngx\\_slprintf](#)
- [ngx\\_sprintf](#)
- [ngx\\_sort](#)
- [ngx\\_sprintf](#)
- [ngx\\_sprintf\\_num](#)
- [ngx\\_str\\_rbtree\\_insert\\_value](#)

- [ngx\\_str\\_rbtree\\_lookup](#)
- [ngx\\_strcasecmp](#)
- [ngx\\_strcasestr](#)
- [ngx\\_strlcasestr](#)
- [ngx\\_strlow](#)
- [ngx\\_strncasecmp](#)
- [ngx\\_strnstr](#)
- [ngx\\_strstrn](#)
- [ngx\\_unescape\\_uri](#)
- [ngx\\_utf8\\_cpymstrn](#)
- [ngx\\_utf8\\_decode](#)
- [ngx\\_utf8\\_length](#)
- [ngx\\_vslprintf](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10
11
12 static u_char *ngx_sprintf_num(u_char *buf, u_char *last, uint64_t ui64,
13     u_char zero, ngx_uint_t hexadecimal, ngx_uint_t width);
14 static void ngx_encode_base64_internal(ngx_str_t *dst, ngx_str_t *src,
15     const u_char *basis, ngx_uint_t padding);
16 static ngx_int_t ngx_decode_base64_internal(ngx_str_t *dst, ngx_str_t *src,
17     const u_char *basis);
18
19
20 void
21 ngx_strlow(u_char *dst, u_char *src, size_t n)
22 {
23     while (n) {
24         *dst = ngx_tolower(*src);
25         dst++;
26         src++;
27         n--;
28     }
29 }
30
31
32 u_char *
33 ngx_cpymstrn(u_char *dst, u_char *src, size_t n)
34 {
35     if (n == 0) {
36         return dst;
37     }
38
39     while (--n) {
40         *dst = *src;
41
42         if (*dst == '\0') {
43             return dst;

```

```

44     }
45
46     dst++;
47     src++;
48 }
49
50 *dst = '\0';
51
52 return dst;
53 }
54
55
56 u_char *
57 ngx_pstrdup(ngx_pool_t *pool, ngx_str_t *src)
58 {
59     u_char *dst;
60
61     dst = ngx_pnalloc(pool, src->len);
62     if (dst == NULL) {
63         return NULL;
64     }
65
66     ngx_memcpy(dst, src->data, src->len);
67
68     return dst;
69 }
70
71
72 /*
73 * supported formats:
74 *   %[0][width][x][X]O      off_t
75 *   %[0][width]T           time_t
76 *   %[0][width][u][x|X]z   ssize_t/size_t
77 *   %[0][width][u][x|X]d   int/u_int
78 *   %[0][width][u][x|X]l   long
79 *   %[0][width|m][u][x|X]i ngx_int_t/ngx_uint_t
80 *   %[0][width][u][x|X]D   int32_t/uint32_t
81 *   %[0][width][u][x|X]L   int64_t/uint64_t
82 *   %[0][width|m][u][x|X]A ngx_atomic_int_t/ngx_atomic_uint_t
83 *   %[0][width][.width]f   double, max valid number fits to %18.15f
84 *   %P                      ngx_pid_t
85 *   %M                      ngx_msec_t
86 *   %r                      rlim_t
87 *   %p                      void *
88 *   %V                      ngx_str_t *
89 *   %v                      ngx_variable_value_t *
90 *   %s                      null-terminated string
91 *   %*s                      length and string
92 *   %Z                      '\0'
93 *   %N                      '\n'
94 *   %C                      char
95 *   %%                      %
96 *
97 * reserved:
98 *   %t                      ptrdiff_t
99 *   %S                      null-terminated wchar string
100 *   %C                      wchar
101 */
102
103
104 u_char * ngx_cdecl
105 ngx_sprintf(u_char *buf, const char *fmt, ...)
106 {
107     u_char *p;
108     va_list args;
109
110     va_start(args, fmt);
111     p = ngx_vsprintf(buf, (void *) -1, fmt, args);
112     va_end(args);
113
114     return p;
115 }
116
117
118 u_char * ngx_cdecl
119 ngx_snprintf(u_char *buf, size_t max, const char *fmt, ...)

```

```

120 {
121     u_char    *p;
122     va_list   args;
123
124     va_start(args, fmt);
125     p = ngx_vslprintf(buf, buf + max, fmt, args);
126     va_end(args);
127
128     return p;
129 }
130
131
132 u_char * ngx_cdecl
133 ngx_slprintf(u_char *buf, u_char *last, const char *fmt, ...)
134 {
135     u_char    *p;
136     va_list   args;
137
138     va_start(args, fmt);
139     p = ngx_vslprintf(buf, last, fmt, args);
140     va_end(args);
141
142     return p;
143 }
144
145
146 u_char *
147 ngx_vslprintf(u_char *buf, u_char *last, const char *fmt, va_list args)
148 {
149     u_char    *p, zero;
150     int        d;
151     double     f;
152     size_t     len, slen;
153     int64_t    i64;
154     uint64_t   ui64, frac;
155     ngx_msec_t ms;
156     ngx_uint_t width, sign, hex, max_width, frac_width, scale, n;
157     ngx_str_t  *v;
158     ngx_variable_value_t *vv;
159
160     while (*fmt && buf < last) {
161
162         /*
163          * "buf < last" means that we could copy at least one character:
164          * the plain character, "%%", "%c", and minus without the checking
165          */
166
167         if (*fmt == '%') {
168
169             i64 = 0;
170             ui64 = 0;
171
172             zero = (u_char) ((*++fmt == '0') ? '0' : ' ');
173             width = 0;
174             sign = 1;
175             hex = 0;
176             max_width = 0;
177             frac_width = 0;
178             slen = (size_t) -1;
179
180             while (*fmt >= '0' && *fmt <= '9') {
181                 width = width * 10 + *fmt++ - '0';
182             }
183
184             for ( ;; ) {
185                 switch (*fmt) {
186
187                     case 'u':
188                         sign = 0;
189                         fmt++;
190                         continue;
191
192                     case 'm':
193                         max_width = 1;
194                         fmt++;

```

```

196         continue;
197
198     case 'X':
199         hex = 2;
200         sign = 0;
201         fmt++;
202         continue;
203
204     case 'x':
205         hex = 1;
206         sign = 0;
207         fmt++;
208         continue;
209
210     case '.':
211         fmt++;
212
213         while (*fmt >= '0' && *fmt <= '9') {
214             frac_width = frac_width * 10 + *fmt++ - '0';
215         }
216
217         break;
218
219     case '*':
220         slen = va_arg(args, size_t);
221         fmt++;
222         continue;
223
224     default:
225         break;
226     }
227
228     break;
229 }
230
231
232 switch (*fmt) {
233
234 case 'v':
235     v = va_arg(args, ngx_str_t *);
236
237     len = ngx_min(((size_t) (last - buf)), v->len);
238     buf = ngx_cpymem(buf, v->data, len);
239     fmt++;
240
241     continue;
242
243 case 'V':
244     vv = va_arg(args, ngx_variable_value_t *);
245
246     len = ngx_min(((size_t) (last - buf)), vv->len);
247     buf = ngx_cpymem(buf, vv->data, len);
248     fmt++;
249
250     continue;
251
252 case 's':
253     p = va_arg(args, u_char *);
254
255     if (slen == (size_t) -1) {
256         while (*p && buf < last) {
257             *buf++ = *p++;
258         }
259     } else {
260         len = ngx_min(((size_t) (last - buf)), slen);
261         buf = ngx_cpymem(buf, p, len);
262     }
263
264     fmt++;
265
266     continue;
267
268 case 'O':
269     i64 = (int64_t) va_arg(args, off_t);
270     sign = 1;
271

```



```
272         break;
273
274     case 'P':
275         i64 = (int64_t) va_arg(args, ngx_pid_t);
276         sign = 1;
277         break;
278
279     case 'T':
280         i64 = (int64_t) va_arg(args, time_t);
281         sign = 1;
282         break;
283
284     case 'M':
285         ms = (ngx_msec_t) va_arg(args, ngx_msec_t);
286         if ((ngx_msec_int_t) ms == -1) {
287             sign = 1;
288             i64 = -1;
289         } else {
290             sign = 0;
291             ui64 = (uint64_t) ms;
292         }
293         break;
294
295     case 'z':
296         if (sign) {
297             i64 = (int64_t) va_arg(args, ssize_t);
298         } else {
299             ui64 = (uint64_t) va_arg(args, size_t);
300         }
301         break;
302
303     case 'i':
304         if (sign) {
305             i64 = (int64_t) va_arg(args, ngx_int_t);
306         } else {
307             ui64 = (uint64_t) va_arg(args, ngx_uint_t);
308         }
309
310         if (max_width) {
311             width = NGX_INT_T_LEN;
312         }
313
314         break;
315
316     case 'd':
317         if (sign) {
318             i64 = (int64_t) va_arg(args, int);
319         } else {
320             ui64 = (uint64_t) va_arg(args, u_int);
321         }
322         break;
323
324     case 'l':
325         if (sign) {
326             i64 = (int64_t) va_arg(args, long);
327         } else {
328             ui64 = (uint64_t) va_arg(args, u_long);
329         }
330         break;
331
332     case 'D':
333         if (sign) {
334             i64 = (int64_t) va_arg(args, int32_t);
335         } else {
336             ui64 = (uint64_t) va_arg(args, uint32_t);
337         }
338         break;
339
340     case 'L':
341         if (sign) {
342             i64 = va_arg(args, int64_t);
343         } else {
344             ui64 = va_arg(args, uint64_t);
345         }
346         break;
347
```

```

348     case 'A':
349         if (sign) {
350             i64 = (int64_t) va_arg(args, ngx_atomic_int_t);
351         } else {
352             ui64 = (uint64_t) va_arg(args, ngx_atomic_uint_t);
353         }
354
355         if (max_width) {
356             width = NGX_ATOMIC_T_LEN;
357         }
358
359         break;
360
361     case 'f':
362         f = va_arg(args, double);
363
364         if (f < 0) {
365             *buf++ = '-';
366             f = -f;
367         }
368
369         ui64 = (int64_t) f;
370         frac = 0;
371
372         if (frac_width) {
373             scale = 1;
374             for (n = frac_width; n; n--) {
375                 scale *= 10;
376             }
377
378             frac = (uint64_t) ((f - (double) ui64) * scale + 0.5);
379
380             if (frac == scale) {
381                 ui64++;
382                 frac = 0;
383             }
384         }
385     }
386
387     buf = ngx_sprintf_num(buf, last, ui64, zero, 0, width);
388
389     if (frac_width) {
390         if (buf < last) {
391             *buf++ = '.';
392         }
393
394         buf = ngx_sprintf_num(buf, last, frac, '0', 0, frac_width);
395     }
396
397     fmt++;
398
399     continue;
400
401 #if !(NGX_WIN32)
402     case 'r':
403         i64 = (int64_t) va_arg(args, rlim_t);
404         sign = 1;
405         break;
406 #endif
407
408     case 'p':
409         ui64 = (uintptr_t) va_arg(args, void *);
410         hex = 2;
411         sign = 0;
412         zero = '0';
413         width = NGX_PTR_SIZE * 2;
414         break;
415
416     case 'c':
417         d = va_arg(args, int);
418         *buf++ = (u_char) (d & 0xff);
419         fmt++;
420
421         continue;
422
423     case 'Z':

```

```

424         *buf++ = '\0';
425         fmt++;
426
427         continue;
428
429         case 'N':
430 #if (NGX_WIN32)
431         *buf++ = CR;
432         if (buf < last) {
433             *buf++ = LF;
434         }
435 #else
436         *buf++ = LF;
437 #endif
438         fmt++;
439
440         continue;
441
442         case '%':
443         *buf++ = '%';
444         fmt++;
445
446         continue;
447
448         default:
449         *buf++ = *fmt++;
450
451         continue;
452     }
453
454     if (sign) {
455         if (i64 < 0) {
456             *buf++ = '-';
457             ui64 = (uint64_t) -i64;
458
459         } else {
460             ui64 = (uint64_t) i64;
461         }
462     }
463
464     buf = ngx_sprintf_num(buf, last, ui64, zero, hex, width);
465
466     fmt++;
467
468     } else {
469         *buf++ = *fmt++;
470     }
471 }
472
473 return buf;
474 }
475
476
477 static u_char *
478 ngx_sprintf_num(u_char *buf, u_char *last, uint64_t ui64, u_char zero,
479 ngx_uint_t hexadecimal, ngx_uint_t width)
480 {
481     u_char      *p, temp[NGX_INT64_LEN + 1];
482
483     /*
484      * we need temp[NGX_INT64_LEN] only,
485      * but icc issues the warning
486      */
487     size_t      len;
488     uint32_t    ui32;
489     static u_char hex[] = "0123456789abcdef";
490     static u_char HEX[] = "0123456789ABCDEF";
491
492     p = temp + NGX_INT64_LEN;
493
494     if (hexadecimal == 0) {
495         if (ui64 <= (uint64_t) NGX_MAX_UINT32_VALUE) {
496
497             /*
498              * To divide 64-bit numbers and to find remainders
499              * on the x86 platform gcc and icc call the libc functions

```

```

500     * [u]divdi3() and [u]moddi3(), they call another function
501     * in its turn. On FreeBSD it is the qdivrem() function,
502     * its source code is about 170 lines of the code.
503     * The glibc counterpart is about 150 lines of the code.
504     *
505     * For 32-bit numbers and some divisors gcc and icc use
506     * a inlined multiplication and shifts. For example,
507     * unsigned "i32 / 10" is compiled to
508     *
509     *     (i32 * 0xCCCCCCCD) >> 35
510     */
511
512     ui32 = (uint32_t) ui64;
513
514     do {
515         *--p = (u_char) (ui32 % 10 + '0');
516     } while (ui32 /= 10);
517
518     } else {
519         do {
520             *--p = (u_char) (ui64 % 10 + '0');
521         } while (ui64 /= 10);
522     }
523
524 } else if (hexadecimal == 1) {
525
526     do {
527
528         /* the "(uint32_t)" cast disables the BCC's warning */
529         *--p = hex[(uint32_t) (ui64 & 0xf)];
530
531     } while (ui64 >>= 4);
532
533 } else { /* hexadecimal == 2 */
534
535     do {
536
537         /* the "(uint32_t)" cast disables the BCC's warning */
538         *--p = HEX[(uint32_t) (ui64 & 0xf)];
539
540     } while (ui64 >>= 4);
541 }
542
543 /* zero or space padding */
544
545 len = (temp + NGX\_INT64\_LEN) - p;
546
547 while (len++ < width && buf < last) {
548     *buf++ = zero;
549 }
550
551 /* number safe copy */
552
553 len = (temp + NGX\_INT64\_LEN) - p;
554
555 if (buf + len > last) {
556     len = last - buf;
557 }
558
559 return ngx\_cpymem(buf, p, len);
560 }
561
562
563 /*
564 * We use ngx\_strcasecmp\(\)/ngx\_strncasecmp\(\) for 7-bit ASCII strings only,
565 * and implement our own ngx\_strcasecmp\(\)/ngx\_strncasecmp\(\)
566 * to avoid libc locale overhead. Besides, we use the ngx\_uint\_t's
567 * instead of the u_char's, because they are slightly faster.
568 */
569
570 ngx\_int\_t
571 ngx\_strcasecmp(u_char *s1, u_char *s2)
572 {
573     ngx\_uint\_t c1, c2;
574
575     for ( ;; ) {

```

```

576     c1 = (ngx\_uint\_t) *s1++;
577     c2 = (ngx\_uint\_t) *s2++;
578
579     c1 = (c1 >= 'A' && c1 <= 'Z') ? (c1 | 0x20) : c1;
580     c2 = (c2 >= 'A' && c2 <= 'Z') ? (c2 | 0x20) : c2;
581
582     if (c1 == c2) {
583
584         if (c1) {
585             continue;
586         }
587
588         return 0;
589     }
590
591     return c1 - c2;
592 }
593 }

```

```

594
595
596 ngx\_int\_t
597 ngx_strncasecmp(u_char *s1, u_char *s2, size\_t n)
598 {
599     ngx\_uint\_t c1, c2;
600
601     while (n) {
602         c1 = (ngx\_uint\_t) *s1++;
603         c2 = (ngx\_uint\_t) *s2++;
604
605         c1 = (c1 >= 'A' && c1 <= 'Z') ? (c1 | 0x20) : c1;
606         c2 = (c2 >= 'A' && c2 <= 'Z') ? (c2 | 0x20) : c2;
607
608         if (c1 == c2) {
609
610             if (c1) {
611                 n--;
612                 continue;
613             }
614
615             return 0;
616         }
617
618         return c1 - c2;
619     }
620
621     return 0;
622 }

```

```

623
624
625 u_char *
626 ngx_strnstr(u_char *s1, char *s2, size\_t len)
627 {
628     u_char c1, c2;
629     size\_t n;
630
631     c2 = *(u_char *) s2++;
632
633     n = ngx\_strlen(s2);
634
635     do {
636         do {
637             if (len-- == 0) {
638                 return NULL;
639             }
640
641             c1 = *s1++;
642
643             if (c1 == 0) {
644                 return NULL;
645             }
646
647         } while (c1 != c2);
648
649         if (n > len) {
650             return NULL;
651         }

```

```

652     } while (ngx\_strncmp(s1, (u_char *) s2, n) != 0);
653
654     return --s1;
655 }
656
657
658
659 /*
660 * ngx\_strstrn\(\) and ngx\_strcasestrn\(\) are intended to search for static
661 * substring with known length in null-terminated string. The argument n
662 * must be length of the second substring - 1.
663 */
664
665 u_char *
666 ngx\_strstrn(u_char *s1, char *s2, size\_t n)
667 {
668     u_char  c1, c2;
669
670     c2 = *(u_char *) s2++;
671
672     do {
673         do {
674             c1 = *s1++;
675
676             if (c1 == 0) {
677                 return NULL;
678             }
679
680             } while (c1 != c2);
681
682     } while (ngx\_strncmp(s1, (u_char *) s2, n) != 0);
683
684     return --s1;
685 }
686
687
688 u_char *
689 ngx\_strcasestrn(u_char *s1, char *s2, size\_t n)
690 {
691     ngx\_uint\_t  c1, c2;
692
693     c2 = (ngx\_uint\_t) *s2++;
694     c2 = (c2 >= 'A' && c2 <= 'Z') ? (c2 | 0x20) : c2;
695
696     do {
697         do {
698             c1 = (ngx\_uint\_t) *s1++;
699
700             if (c1 == 0) {
701                 return NULL;
702             }
703
704             c1 = (c1 >= 'A' && c1 <= 'Z') ? (c1 | 0x20) : c1;
705
706             } while (c1 != c2);
707
708     } while (ngx\_strncasecmp(s1, (u_char *) s2, n) != 0);
709
710     return --s1;
711 }
712
713
714 /*
715 * ngx\_strlcasestrn\(\) is intended to search for static substring
716 * with known length in string until the argument last. The argument n
717 * must be length of the second substring - 1.
718 */
719
720 u_char *
721 ngx\_strlcasestrn(u_char *s1, u_char *last, u_char *s2, size\_t n)
722 {
723     ngx\_uint\_t  c1, c2;
724
725     c2 = (ngx\_uint\_t) *s2++;
726     c2 = (c2 >= 'A' && c2 <= 'Z') ? (c2 | 0x20) : c2;
727     last -= n;

```

```

728
729     do {
730         do {
731             if (s1 >= last) {
732                 return NULL;
733             }
734
735             c1 = (ngx_uint_t) *s1++;
736
737             c1 = (c1 >= 'A' && c1 <= 'Z') ? (c1 | 0x20) : c1;
738
739         } while (c1 != c2);
740
741     } while (ngx_strncasecmp(s1, s2, n) != 0);
742
743     return --s1;
744 }
745
746
747 ngx_int_t
748 ngx_rstrncmp(u_char *s1, u_char *s2, size_t n)
749 {
750     if (n == 0) {
751         return 0;
752     }
753
754     n--;
755
756     for ( ;; ) {
757         if (s1[n] != s2[n]) {
758             return s1[n] - s2[n];
759         }
760
761         if (n == 0) {
762             return 0;
763         }
764
765         n--;
766     }
767 }
768
769
770 ngx_int_t
771 ngx_rstrncasecmp(u_char *s1, u_char *s2, size_t n)
772 {
773     u_char  c1, c2;
774
775     if (n == 0) {
776         return 0;
777     }
778
779     n--;
780
781     for ( ;; ) {
782         c1 = s1[n];
783         if (c1 >= 'a' && c1 <= 'z') {
784             c1 -= 'a' - 'A';
785         }
786
787         c2 = s2[n];
788         if (c2 >= 'a' && c2 <= 'z') {
789             c2 -= 'a' - 'A';
790         }
791
792         if (c1 != c2) {
793             return c1 - c2;
794         }
795
796         if (n == 0) {
797             return 0;
798         }
799
800         n--;
801     }
802 }
803

```

```

804 ngx\_int\_t
805 ngx\_memn2cmp(u_char *s1, u_char *s2, size\_t n1, size\_t n2)
806 {
807     size\_t    n;
808     ngx\_int\_t m, z;
809
810     if (n1 <= n2) {
811         n = n1;
812         z = -1;
813     } else {
814         n = n2;
815         z = 1;
816     }
817
818     m = ngx\_memcmp(s1, s2, n);
819
820     if (m || n1 == n2) {
821         return m;
822     }
823
824     return z;
825 }
826
827 ngx\_int\_t
828 ngx\_dns\_strcmp(u_char *s1, u_char *s2)
829 {
830     ngx\_uint\_t c1, c2;
831
832     for ( ;; ) {
833         c1 = (ngx\_uint\_t) *s1++;
834         c2 = (ngx\_uint\_t) *s2++;
835
836         c1 = (c1 >= 'A' && c1 <= 'Z') ? (c1 | 0x20) : c1;
837         c2 = (c2 >= 'A' && c2 <= 'Z') ? (c2 | 0x20) : c2;
838
839         if (c1 == c2) {
840             if (c1) {
841                 continue;
842             }
843
844             return 0;
845         }
846
847         /* in ASCII '.' > '-', but we need '.' to be the lowest character */
848
849         c1 = (c1 == '.') ? ' ' : c1;
850         c2 = (c2 == '.') ? ' ' : c2;
851
852         return c1 - c2;
853     }
854 }
855
856 ngx\_int\_t
857 ngx\_filename\_cmp(u_char *s1, u_char *s2, size\_t n)
858 {
859     ngx\_uint\_t c1, c2;
860
861     while (n) {
862         c1 = (ngx\_uint\_t) *s1++;
863         c2 = (ngx\_uint\_t) *s2++;
864
865         #if (NGX\_HAVE\_CASELESS\_FILESYSTEM)
866             c1 = tolower(c1);
867             c2 = tolower(c2);
868         #endif
869
870         if (c1 == c2) {
871             if (c1) {
872                 n--;
873                 continue;

```



```

880     }
881
882     return 0;
883 }
884
885 /* we need '/' to be the lowest character */
886
887 if (c1 == 0 || c2 == 0) {
888     return c1 - c2;
889 }
890
891 c1 = (c1 == '/') ? 0 : c1;
892 c2 = (c2 == '/') ? 0 : c2;
893
894 return c1 - c2;
895 }
896
897 return 0;
898 }
899
900
901 ngx_int_t
902 ngx_atoi(u_char *line, size_t n)
903 {
904     ngx_int_t value;
905
906     if (n == 0) {
907         return NGX_ERROR;
908     }
909
910     for (value = 0; n--; line++) {
911         if (*line < '0' || *line > '9') {
912             return NGX_ERROR;
913         }
914
915         value = value * 10 + (*line - '0');
916     }
917
918     if (value < 0) {
919         return NGX_ERROR;
920     }
921     else {
922         return value;
923     }
924 }
925
926
927 /* parse a fixed point number, e.g., ngx_atofp("10.5", 4, 2) returns 1050 */
928
929 ngx_int_t
930 ngx_atofp(u_char *line, size_t n, size_t point)
931 {
932     ngx_int_t value;
933     ngx_uint_t dot;
934
935     if (n == 0) {
936         return NGX_ERROR;
937     }
938
939     dot = 0;
940
941     for (value = 0; n--; line++) {
942
943         if (point == 0) {
944             return NGX_ERROR;
945         }
946
947         if (*line == '.') {
948             if (dot) {
949                 return NGX_ERROR;
950             }
951
952             dot = 1;
953             continue;
954         }
955

```

```

956     if (*line < '0' || *line > '9') {
957         return NGX\_ERROR;
958     }
959
960     value = value * 10 + (*line - '0');
961     point -= dot;
962 }
963
964 while (point--) {
965     value = value * 10;
966 }
967
968 if (value < 0) {
969     return NGX\_ERROR;
970 }
971 else {
972     return value;
973 }
974 }
975
976
977 ssize_t
978 ngx_atosz(u_char *line, size_t n)
979 {
980     ssize_t value;
981
982     if (n == 0) {
983         return NGX\_ERROR;
984     }
985
986     for (value = 0; n--; line++) {
987         if (*line < '0' || *line > '9') {
988             return NGX\_ERROR;
989         }
990
991         value = value * 10 + (*line - '0');
992     }
993
994     if (value < 0) {
995         return NGX\_ERROR;
996     }
997     else {
998         return value;
999     }
1000 }
1001
1002
1003 off_t
1004 ngx_atoof(u_char *line, size_t n)
1005 {
1006     off_t value;
1007
1008     if (n == 0) {
1009         return NGX\_ERROR;
1010     }
1011
1012     for (value = 0; n--; line++) {
1013         if (*line < '0' || *line > '9') {
1014             return NGX\_ERROR;
1015         }
1016
1017         value = value * 10 + (*line - '0');
1018     }
1019
1020     if (value < 0) {
1021         return NGX\_ERROR;
1022     }
1023     else {
1024         return value;
1025     }
1026 }
1027
1028
1029 time_t
1030 ngx_atotm(u_char *line, size_t n)
1031 {

```

```

1032     time_t value;
1033
1034     if (n == 0) {
1035         return NGX\_ERROR;
1036     }
1037
1038     for (value = 0; n--; line++) {
1039         if (*line < '0' || *line > '9') {
1040             return NGX\_ERROR;
1041         }
1042
1043         value = value * 10 + (*line - '0');
1044     }
1045
1046     if (value < 0) {
1047         return NGX\_ERROR;
1048     } else {
1049         return value;
1050     }
1051 }
1052
1053
1054
1055 ngx\_int\_t
1056 ngx\_hextoi(u_char *line, size\_t n)
1057 {
1058     u_char    c, ch;
1059     ngx\_int\_t value;
1060
1061     if (n == 0) {
1062         return NGX\_ERROR;
1063     }
1064
1065     for (value = 0; n--; line++) {
1066         ch = *line;
1067
1068         if (ch >= '0' && ch <= '9') {
1069             value = value * 16 + (ch - '0');
1070             continue;
1071         }
1072
1073         c = (u_char) (ch | 0x20);
1074
1075         if (c >= 'a' && c <= 'f') {
1076             value = value * 16 + (c - 'a' + 10);
1077             continue;
1078         }
1079
1080         return NGX\_ERROR;
1081     }
1082
1083     if (value < 0) {
1084         return NGX\_ERROR;
1085     } else {
1086         return value;
1087     }
1088 }
1089
1090
1091 u_char *
1092 ngx\_hex\_dump(u_char *dst, u_char *src, size\_t len)
1093 {
1094     static u_char hex[] = "0123456789abcdef";
1095
1096     while (len--) {
1097         *dst++ = hex[*src >> 4];
1098         *dst++ = hex[*src & 0xf];
1099     }
1100
1101     return dst;
1102 }
1103
1104
1105
1106 void
1107 ngx\_encode\_base64(ngx\_str\_t *dst, ngx\_str\_t *src)

```

```

1108 {
1109     static u_char  basis64[] =
1110         "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
1111
1112     ngx_encode_base64_internal(dst, src, basis64, 1);
1113 }
1114
1115
1116 void
1117 ngx_encode_base64url(ngx_str_t *dst, ngx_str_t *src)
1118 {
1119     static u_char  basis64[] =
1120         "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789-_" ;
1121
1122     ngx_encode_base64_internal(dst, src, basis64, 0);
1123 }
1124
1125
1126 static void
1127 ngx_encode_base64_internal(ngx_str_t *dst, ngx_str_t *src, const u_char *basis,
1128     ngx_uint_t padding)
1129 {
1130     u_char          *d, *s;
1131     size_t          len;
1132
1133     len = src->len;
1134     s = src->data;
1135     d = dst->data;
1136
1137     while (len > 2) {
1138         *d++ = basis[(s[0] >> 2) & 0x3f];
1139         *d++ = basis[((s[0] & 3) << 4) | (s[1] >> 4)];
1140         *d++ = basis[((s[1] & 0x0f) << 2) | (s[2] >> 6)];
1141         *d++ = basis[s[2] & 0x3f];
1142
1143         s += 3;
1144         len -= 3;
1145     }
1146
1147     if (len) {
1148         *d++ = basis[(s[0] >> 2) & 0x3f];
1149
1150         if (len == 1) {
1151             *d++ = basis[(s[0] & 3) << 4];
1152             if (padding) {
1153                 *d++ = '=';
1154             }
1155         } else {
1156             *d++ = basis[((s[0] & 3) << 4) | (s[1] >> 4)];
1157             *d++ = basis[(s[1] & 0x0f) << 2];
1158         }
1159
1160         if (padding) {
1161             *d++ = '=';
1162         }
1163     }
1164 }
1165
1166 dst->len = d - dst->data;
1167 }
1168
1169
1170 ngx_int_t
1171 ngx_decode_base64(ngx_str_t *dst, ngx_str_t *src)
1172 {
1173     static u_char  basis64[] = {
1174         77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77,
1175         77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77,
1176         77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 62, 77, 77, 77, 63,
1177         52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 77, 77, 77, 77, 77, 77,
1178         77, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
1179         15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 77, 77, 77, 77, 77,
1180         77, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
1181         41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 77, 77, 77, 77, 77,
1182
1183         77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77,

```

```

1184     77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77,
1185     77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77,
1186     77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77,
1187     77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77,
1188     77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77,
1189     77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77,
1190     77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77,
1191 };
1192
1193     return ngx_decode_base64_internal(dst, src, basis64);
1194 }
1195
1196
1197 ngx_int_t
1198 ngx_decode_base64url(ngx_str_t *dst, ngx_str_t *src)
1199 {
1200     static u_char    basis64[] = {
1201         77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77,
1202         77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77,
1203         77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 62, 77, 77,
1204         52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 77, 77, 77, 77, 77, 77,
1205         77, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
1206         15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 77, 77, 77, 77, 63,
1207         77, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
1208         41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 77, 77, 77, 77, 77,
1209
1210         77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77,
1211         77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77,
1212         77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77,
1213         77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77,
1214         77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77,
1215         77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77,
1216         77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77,
1217         77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77, 77,
1218     };
1219
1220     return ngx_decode_base64_internal(dst, src, basis64);
1221 }
1222
1223
1224 static ngx_int_t
1225 ngx_decode_base64_internal(ngx_str_t *dst, ngx_str_t *src, const u_char *basis)
1226 {
1227     size_t    len;
1228     u_char    *d, *s;
1229
1230     for (len = 0; len < src->len; len++) {
1231         if (src->data[len] == '=') {
1232             break;
1233         }
1234
1235         if (basis[src->data[len]] == 77) {
1236             return NGX_ERROR;
1237         }
1238     }
1239
1240     if (len % 4 == 1) {
1241         return NGX_ERROR;
1242     }
1243
1244     s = src->data;
1245     d = dst->data;
1246
1247     while (len > 3) {
1248         *d++ = (u_char) (basis[s[0]] << 2 | basis[s[1]] >> 4);
1249         *d++ = (u_char) (basis[s[1]] << 4 | basis[s[2]] >> 2);
1250         *d++ = (u_char) (basis[s[2]] << 6 | basis[s[3]]);
1251
1252         s += 4;
1253         len -= 4;
1254     }
1255
1256     if (len > 1) {
1257         *d++ = (u_char) (basis[s[0]] << 2 | basis[s[1]] >> 4);
1258     }
1259

```

```

1260     if (len > 2) {
1261         *d++ = (u_char) (basis[s[1]] << 4 | basis[s[2]] >> 2);
1262     }
1263
1264     dst->len = d - dst->data;
1265
1266     return NGX_OK;
1267 }
1268
1269
1270 /*
1271  * ngx_utf8_decode() decodes two and more bytes UTF sequences only
1272  * the return values:
1273  * 0x80 - 0x10ffff    valid character
1274  * 0x110000 - 0xffffffff    invalid sequence
1275  * 0xffffffffe    incomplete sequence
1276  * 0xffffffff    error
1277 */
1278
1279 uint32_t
1280 ngx_utf8_decode(u_char **p, size_t n)
1281 {
1282     size_t    len;
1283     uint32_t  u, i, valid;
1284
1285     u = **p;
1286
1287     if (u >= 0xf0) {
1288
1289         u &= 0x07;
1290         valid = 0xffff;
1291         len = 3;
1292
1293     } else if (u >= 0xe0) {
1294
1295         u &= 0x0f;
1296         valid = 0x7ff;
1297         len = 2;
1298
1299     } else if (u >= 0xc2) {
1300
1301         u &= 0x1f;
1302         valid = 0x7f;
1303         len = 1;
1304
1305     } else {
1306         (*p)++;
1307         return 0xffffffff;
1308     }
1309
1310     if (n - 1 < len) {
1311         return 0xffffffffe;
1312     }
1313
1314     (*p)++;
1315
1316     while (len) {
1317         i = *(*p)++;
1318
1319         if (i < 0x80) {
1320             return 0xffffffff;
1321         }
1322
1323         u = (u << 6) | (i & 0x3f);
1324
1325         len--;
1326     }
1327
1328     if (u > valid) {
1329         return u;
1330     }
1331
1332     return 0xffffffff;
1333 }
1334
1335

```

```

1336 size_t
1337 ngx_utf8_length(u_char *p, size_t n)
1338 {
1339     u_char c, *last;
1340     size_t len;
1341
1342     last = p + n;
1343
1344     for (len = 0; p < last; len++) {
1345
1346         c = *p;
1347
1348         if (c < 0x80) {
1349             p++;
1350             continue;
1351         }
1352
1353         if (ngx_utf8_decode(&p, n) > 0x10ffff) {
1354             /* invalid UTF-8 */
1355             return n;
1356         }
1357     }
1358
1359     return len;
1360 }
1361
1362
1363 u_char *
1364 ngx_utf8_cpystirn(u_char *dst, u_char *src, size_t n, size_t len)
1365 {
1366     u_char c, *next;
1367
1368     if (n == 0) {
1369         return dst;
1370     }
1371
1372     while (--n) {
1373
1374         c = *src;
1375         *dst = c;
1376
1377         if (c < 0x80) {
1378
1379             if (c != '\0') {
1380                 dst++;
1381                 src++;
1382                 len--;
1383
1384                 continue;
1385             }
1386
1387             return dst;
1388         }
1389
1390         next = src;
1391
1392         if (ngx_utf8_decode(&next, len) > 0x10ffff) {
1393             /* invalid UTF-8 */
1394             break;
1395         }
1396
1397         while (src < next) {
1398             *dst++ = *src++;
1399             len--;
1400         }
1401     }
1402
1403     *dst = '\0';
1404
1405     return dst;
1406 }
1407
1408
1409 uintptr_t
1410 ngx_escape_uri(u_char *dst, u_char *src, size_t size, ngx_uint_t type)
1411 {

```

```

1412 ngx_uint_t n;
1413 uint32_t *escape;
1414 static u_char hex[] = "0123456789ABCDEF";
1415
1416 /* " ", "#", "%", "?", %00-%1F, %7F-%FF */
1417
1418 static uint32_t uri[] = {
1419     0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
1420
1421     /* ?>=< ;:98 7654 3210 /.-, +*)( '&%$ #"! */
1422     0x80000029, /* 1000 0000 0000 0000 0000 0000 0010 1001 */
1423
1424     /* _^]\ [ZYX WVUT SRQP ONML KJIH GFED CBA@ */
1425     0x00000000, /* 0000 0000 0000 0000 0000 0000 0000 0000 */
1426
1427     /* ~}| {zyx wvut srqp onml kjih gfed cba` */
1428     0x80000000, /* 1000 0000 0000 0000 0000 0000 0000 0000 */
1429
1430     0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
1431     0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
1432     0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
1433     0xffffffff /* 1111 1111 1111 1111 1111 1111 1111 1111 */
1434 };
1435
1436 /* " ", "#", "%", "&", "+", "?", %00-%1F, %7F-%FF */
1437
1438 static uint32_t args[] = {
1439     0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
1440
1441     /* ?>=< ;:98 7654 3210 /.-, +*)( '&%$ #"! */
1442     0x88000869, /* 1000 1000 0000 0000 0000 1000 0110 1001 */
1443
1444     /* _^]\ [ZYX WVUT SRQP ONML KJIH GFED CBA@ */
1445     0x00000000, /* 0000 0000 0000 0000 0000 0000 0000 0000 */
1446
1447     /* ~}| {zyx wvut srqp onml kjih gfed cba` */
1448     0x80000000, /* 1000 0000 0000 0000 0000 0000 0000 0000 */
1449
1450     0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
1451     0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
1452     0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
1453     0xffffffff /* 1111 1111 1111 1111 1111 1111 1111 1111 */
1454 };
1455
1456 /* not ALPHA, DIGIT, "-", ".", "_", "~" */
1457
1458 static uint32_t uri_component[] = {
1459     0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
1460
1461     /* ?>=< ;:98 7654 3210 /.-, +*)( '&%$ #"! */
1462     0xfc009fff, /* 1111 1100 0000 0000 1001 1111 1111 1111 */
1463
1464     /* _^]\ [ZYX WVUT SRQP ONML KJIH GFED CBA@ */
1465     0x78000001, /* 0111 1000 0000 0000 0000 0000 0000 0001 */
1466
1467     /* ~}| {zyx wvut srqp onml kjih gfed cba` */
1468     0xb8000001, /* 1011 1000 0000 0000 0000 0000 0000 0001 */
1469
1470     0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
1471     0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
1472     0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
1473     0xffffffff /* 1111 1111 1111 1111 1111 1111 1111 1111 */
1474 };
1475
1476 /* " ", "#", "", "%", "'", %00-%1F, %7F-%FF */
1477
1478 static uint32_t html[] = {
1479     0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
1480
1481     /* ?>=< ;:98 7654 3210 /.-, +*)( '&%$ #"! */
1482     0x000000ad, /* 0000 0000 0000 0000 0000 0000 1010 1101 */
1483
1484     /* _^]\ [ZYX WVUT SRQP ONML KJIH GFED CBA@ */
1485     0x00000000, /* 0000 0000 0000 0000 0000 0000 0000 0000 */
1486
1487     /* ~}| {zyx wvut srqp onml kjih gfed cba` */

```



```

1488     0x80000000, /* 1000 0000 0000 0000 0000 0000 0000 0000 */
1489
1490     0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
1491     0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
1492     0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
1493     0xffffffff /* 1111 1111 1111 1111 1111 1111 1111 1111 */
1494 };
1495
1496     /* " ", "'", "%", "'", %00-%1F, %7F-%FF */
1497
1498 static uint32_t refresh[] = {
1499     0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
1500
1501     /* ?>=< ;:98 7654 3210 /.-, +*)( '&%$ #"! */
1502     0x00000085, /* 0000 0000 0000 0000 0000 0000 1000 0101 */
1503
1504     /* _^]\ [ZYX WVUT SRQP ONML KJIH GFED CBA@ */
1505     0x00000000, /* 0000 0000 0000 0000 0000 0000 0000 0000 */
1506
1507     /* ~}| {zyx wvut srqp onml kjih gfed cba` */
1508     0x80000000, /* 1000 0000 0000 0000 0000 0000 0000 0000 */
1509
1510     0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
1511     0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
1512     0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
1513     0xffffffff /* 1111 1111 1111 1111 1111 1111 1111 1111 */
1514 };
1515
1516     /* " ", "%", %00-%1F */
1517
1518 static uint32_t memcached[] = {
1519     0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
1520
1521     /* ?>=< ;:98 7654 3210 /.-, +*)( '&%$ #"! */
1522     0x00000021, /* 0000 0000 0000 0000 0000 0000 0010 0001 */
1523
1524     /* _^]\ [ZYX WVUT SRQP ONML KJIH GFED CBA@ */
1525     0x00000000, /* 0000 0000 0000 0000 0000 0000 0000 0000 */
1526
1527     /* ~}| {zyx wvut srqp onml kjih gfed cba` */
1528     0x00000000, /* 0000 0000 0000 0000 0000 0000 0000 0000 */
1529
1530     0x00000000, /* 0000 0000 0000 0000 0000 0000 0000 0000 */
1531     0x00000000, /* 0000 0000 0000 0000 0000 0000 0000 0000 */
1532     0x00000000, /* 0000 0000 0000 0000 0000 0000 0000 0000 */
1533     0x00000000, /* 0000 0000 0000 0000 0000 0000 0000 0000 */
1534 };
1535
1536     /* mail_auth is the same as memcached */
1537
1538 static uint32_t *map[] =
1539     { uri, args, uri_component, html, refresh, memcached, memcached };
1540
1541
1542 escape = map[type];
1543
1544 if (dst == NULL) {
1545
1546     /* find the number of the characters to be escaped */
1547
1548     n = 0;
1549
1550     while (size) {
1551         if (escape[*src >> 5] & (1 << (*src & 0x1f))) {
1552             n++;
1553         }
1554         src++;
1555         size--;
1556     }
1557
1558     return (uintptr_t) n;
1559 }
1560
1561 while (size) {
1562     if (escape[*src >> 5] & (1 << (*src & 0x1f))) {
1563         *dst++ = '%';

```

```

1564         *dst++ = hex[*src >> 4];
1565         *dst++ = hex[*src & 0xf];
1566         src++;
1567
1568     } else {
1569         *dst++ = *src++;
1570     }
1571     size--;
1572 }
1573
1574 return (uintptr_t) dst;
1575 }
1576
1577
1578 void
1579 ngx_unescape_uri(u_char **dst, u_char **src, size_t size, ngx_uint_t type)
1580 {
1581     u_char *d, *s, ch, c, decoded;
1582     enum {
1583         sw_usual = 0,
1584         sw_quoted,
1585         sw_quoted_second
1586     } state;
1587
1588     d = *dst;
1589     s = *src;
1590
1591     state = 0;
1592     decoded = 0;
1593
1594     while (size--) {
1595
1596         ch = *s++;
1597
1598         switch (state) {
1599         case sw_usual:
1600             if (ch == '?'
1601                 && (type & (NGX_UNESCAPE_URI|NGX_UNESCAPE_REDIRECT)))
1602             {
1603                 *d++ = ch;
1604                 goto done;
1605             }
1606
1607             if (ch == '%') {
1608                 state = sw_quoted;
1609                 break;
1610             }
1611
1612             *d++ = ch;
1613             break;
1614
1615         case sw_quoted:
1616
1617             if (ch >= '0' && ch <= '9') {
1618                 decoded = (u_char) (ch - '0');
1619                 state = sw_quoted_second;
1620                 break;
1621             }
1622
1623             c = (u_char) (ch | 0x20);
1624             if (c >= 'a' && c <= 'f') {
1625                 decoded = (u_char) (c - 'a' + 10);
1626                 state = sw_quoted_second;
1627                 break;
1628             }
1629
1630             /* the invalid quoted character */
1631
1632             state = sw_usual;
1633
1634             *d++ = ch;
1635
1636             break;
1637
1638         case sw_quoted_second:

```

```

1640     state = sw_usual;
1641
1642     if (ch >= '0' && ch <= '9') {
1643         ch = (u_char) ((decoded << 4) + ch - '0');
1644
1645         if (type & NGX\_UNESCAPE\_REDIRECT) {
1646             if (ch > '%' && ch < 0x7f) {
1647                 *d++ = ch;
1648                 break;
1649             }
1650
1651             *d++ = '%'; *d++ = *(s - 2); *d++ = *(s - 1);
1652
1653             break;
1654         }
1655
1656         *d++ = ch;
1657
1658         break;
1659     }
1660
1661     c = (u_char) (ch | 0x20);
1662     if (c >= 'a' && c <= 'f') {
1663         ch = (u_char) ((decoded << 4) + c - 'a' + 10);
1664
1665         if (type & NGX\_UNESCAPE\_URI) {
1666             if (ch == '?') {
1667                 *d++ = ch;
1668                 goto done;
1669             }
1670
1671             *d++ = ch;
1672             break;
1673         }
1674
1675         if (type & NGX\_UNESCAPE\_REDIRECT) {
1676             if (ch == '?') {
1677                 *d++ = ch;
1678                 goto done;
1679             }
1680
1681             if (ch > '%' && ch < 0x7f) {
1682                 *d++ = ch;
1683                 break;
1684             }
1685
1686             *d++ = '%'; *d++ = *(s - 2); *d++ = *(s - 1);
1687             break;
1688         }
1689
1690         *d++ = ch;
1691
1692         break;
1693     }
1694
1695     /* the invalid quoted character */
1696
1697     break;
1698 }
1699 }
1700
1701 done:
1702
1703     *dst = d;
1704     *src = s;
1705 }
1706
1707
1708 uintptr\_t
1709 ngx\_escape\_html(u_char *dst, u_char *src, size\_t size)
1710 {
1711     u_char     ch;
1712     ngx\_uint\_t len;
1713
1714     if (dst == NULL) {
1715

```

```

1716     len = 0;
1717
1718     while (size) {
1719         switch (*src++) {
1720
1721             case '<':
1722                 len += sizeof("&lt;") - 2;
1723                 break;
1724
1725             case '>':
1726                 len += sizeof("&gt;") - 2;
1727                 break;
1728
1729             case '&':
1730                 len += sizeof("&amp;") - 2;
1731                 break;
1732
1733             case '"':
1734                 len += sizeof("&quot;") - 2;
1735                 break;
1736
1737             default:
1738                 break;
1739         }
1740         size--;
1741     }
1742
1743     return (uintptr_t) len;
1744 }
1745
1746 while (size) {
1747     ch = *src++;
1748
1749     switch (ch) {
1750
1751         case '<':
1752             *dst++ = '&'; *dst++ = 'l'; *dst++ = 't'; *dst++ = ';';
1753             break;
1754
1755         case '>':
1756             *dst++ = '&'; *dst++ = 'g'; *dst++ = 't'; *dst++ = ';';
1757             break;
1758
1759         case '&':
1760             *dst++ = '&'; *dst++ = 'a'; *dst++ = 'm'; *dst++ = 'p';
1761             *dst++ = ';';
1762             break;
1763
1764         case '"':
1765             *dst++ = '&'; *dst++ = 'q'; *dst++ = 'u'; *dst++ = 'o';
1766             *dst++ = 't'; *dst++ = ';';
1767             break;
1768
1769         default:
1770             *dst++ = ch;
1771             break;
1772     }
1773     size--;
1774 }
1775
1776 return (uintptr_t) dst;
1777 }
1778
1779
1780 uintptr_t
1781 ngx_escape_json(u_char *dst, u_char *src, size_t size)
1782 {
1783     u_char    ch;
1784     ngx_uint_t len;
1785
1786     if (dst == NULL) {
1787         len = 0;
1788
1789         while (size) {
1790             ch = *src++;

```

```

1792         if (ch == '\\\\' || ch == '') {
1793             len++;
1794
1795         } else if (ch <= 0x1f) {
1796             len += sizeof("\\u001F") - 2;
1797         }
1798
1799         size--;
1800     }
1801
1802     return (uintptr_t) len;
1803 }
1804
1805 while (size) {
1806     ch = *src++;
1807
1808     if (ch > 0x1f) {
1809
1810         if (ch == '\\\\' || ch == '') {
1811             *dst++ = '\\\\';
1812         }
1813
1814         *dst++ = ch;
1815
1816     } else {
1817         *dst++ = '\\\\'; *dst++ = 'u'; *dst++ = '0'; *dst++ = '0';
1818         *dst++ = '0' + (ch >> 4);
1819
1820         ch &= 0xf;
1821
1822         *dst++ = (ch < 10) ? ('0' + ch) : ('A' + ch - 10);
1823     }
1824
1825     size--;
1826 }
1827
1828 return (uintptr_t) dst;
1829 }
1830
1831
1832 void
1833 ngx_str_rbtree_insert_value(ngx_rbtree_node_t *temp,
1834 ngx_rbtree_node_t *node, ngx_rbtree_node_t *sentinel)
1835 {
1836     ngx_str_node_t    *n, *t;
1837     ngx_rbtree_node_t **p;
1838
1839     for ( ;; ) {
1840
1841         n = (ngx_str_node_t *) node;
1842         t = (ngx_str_node_t *) temp;
1843
1844         if (node->key != temp->key) {
1845
1846             p = (node->key < temp->key) ? &temp->left : &temp->right;
1847
1848         } else if (n->str.len != t->str.len) {
1849
1850             p = (n->str.len < t->str.len) ? &temp->left : &temp->right;
1851
1852         } else {
1853             p = (ngx_memcmp(n->str.data, t->str.data, n->str.len) < 0)
1854                 ? &temp->left : &temp->right;
1855         }
1856
1857         if (*p == sentinel) {
1858             break;
1859         }
1860
1861         temp = *p;
1862     }
1863
1864     *p = node;
1865     node->parent = temp;
1866     node->left = sentinel;
1867     node->right = sentinel;

```

```

1868     ngx_rbt_red(node);
1869 }
1870
1871
1872 ngx_str_node_t *
1873 ngx_str_rbtree_lookup(ngx_rbtree_t *rbtree, ngx_str_t *val, uint32_t hash)
1874 {
1875     ngx_int_t      rc;
1876     ngx_str_node_t *n;
1877     ngx_rbtree_node_t *node, *sentinel;
1878
1879     node = rbtree->root;
1880     sentinel = rbtree->sentinel;
1881
1882     while (node != sentinel) {
1883
1884         n = (ngx_str_node_t *) node;
1885
1886         if (hash != node->key) {
1887             node = (hash < node->key) ? node->left : node->right;
1888             continue;
1889         }
1890
1891         if (val->len != n->str.len) {
1892             node = (val->len < n->str.len) ? node->left : node->right;
1893             continue;
1894         }
1895
1896         rc = ngx_memcmp(val->data, n->str.data, val->len);
1897
1898         if (rc < 0) {
1899             node = node->left;
1900             continue;
1901         }
1902
1903         if (rc > 0) {
1904             node = node->right;
1905             continue;
1906         }
1907
1908         return n;
1909     }
1910
1911     return NULL;
1912 }
1913
1914
1915 /* ngx_sort() is implemented as insertion sort because we need stable sort */
1916
1917 void
1918 ngx_sort(void *base, size_t n, size_t size,
1919         ngx_int_t (*cmp)(const void *, const void *))
1920 {
1921     u_char *p1, *p2, *p;
1922
1923     p = ngx_alloc(size, ngx_cycle->log);
1924     if (p == NULL) {
1925         return;
1926     }
1927
1928     for (p1 = (u_char *) base + size;
1929         p1 < (u_char *) base + n * size;
1930         p1 += size)
1931     {
1932         ngx_memcpy(p, p1, size);
1933
1934         for (p2 = p1;
1935             p2 > (u_char *) base && cmp(p2 - size, p) > 0;
1936             p2 -= size)
1937         {
1938             ngx_memcpy(p2, p2 - size, size);
1939         }
1940
1941         ngx_memcpy(p2, p, size);
1942     }
1943

```

```
1944     ngx_free(p);
1945 }
1946
1947
1948 #if (NGX_MEMCPY_LIMIT)
1949
1950 void *
1951 ngx_memcpy(void *dst, const void *src, size_t n)
1952 {
1953     if (n > NGX_MEMCPY_LIMIT) {
1954         ngx_log_error(NGX_LOG_ALERT, ngx_cycle->log, 0, "memcpy %uz bytes", n);
1955         ngx_debug_point();
1956     }
1957
1958     return memcpy(dst, src, n);
1959 }
1960
1961 #endif
```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_palloc.c - nginx-1.7.10

### Functions defined

- [ngx\\_create\\_pool](#)
- [ngx\\_destroy\\_pool](#)
- [ngx\\_palloc](#)
- [ngx\\_palloc\\_block](#)
- [ngx\\_palloc\\_large](#)
- [ngx\\_pcalloc](#)
- [ngx\\_pfree](#)
- [ngx\\_pmemalign](#)
- [ngx\\_pnalloc](#)
- [ngx\\_pool\\_cleanup\\_add](#)
- [ngx\\_pool\\_cleanup\\_file](#)
- [ngx\\_pool\\_delete\\_file](#)
- [ngx\\_pool\\_run\\_cleanup\\_file](#)
- [ngx\\_reset\\_pool](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10
11
12 static void *ngx_palloc_block(ngx_pool_t *pool, size_t size);
13 static void *ngx_palloc_large(ngx_pool_t *pool, size_t size);
14
15
16 ngx_pool_t *
17 ngx_create_pool(size_t size, ngx_log_t *log)
18 {
19     ngx_pool_t *p;
20
21     p = ngx_memalign(NGX_POOL_ALIGNMENT, size, log);
22     if (p == NULL) {
23         return NULL;
24     }
25
26     p->d.last = (u_char *) p + sizeof(ngx_pool_t);
27     p->d.end = (u_char *) p + size;
28     p->d.next = NULL;
29     p->d.failed = 0;
30
31     size = size - sizeof(ngx_pool_t);
32     p->max = (size < NGX_MAX_ALLOC_FROM_POOL) ? size : NGX_MAX_ALLOC_FROM_POOL;
33 }
```



```

34     p->current = p;
35     p->chain = NULL;
36     p->large = NULL;
37     p->cleanup = NULL;
38     p->log = log;
39
40     return p;
41 }
42
43
44 void
45 ngx_destroy_pool(ngx_pool_t *pool)
46 {
47     ngx_pool_t      *p, *n;
48     ngx_pool_large_t *l;
49     ngx_pool_cleanup_t *c;
50
51     for (c = pool->cleanup; c; c = c->next) {
52         if (c->handler) {
53             ngx_log_debug1(NGX_LOG_DEBUG_ALLOC, pool->log, 0,
54                 "run cleanup: %p", c);
55             c->handler(c->data);
56         }
57     }
58
59     for (l = pool->large; l; l = l->next) {
60
61         ngx_log_debug1(NGX_LOG_DEBUG_ALLOC, pool->log, 0, "free: %p", l->alloc);
62
63         if (l->alloc) {
64             ngx_free(l->alloc);
65         }
66     }
67
68     #if (NGX_DEBUG)
69     /*
70      * we could allocate the pool->log from this pool
71      * so we cannot use this log while free()ing the pool
72      */
73
74     for (p = pool, n = pool->d.next; /* void */; p = n, n = n->d.next) {
75         ngx_log_debug2(NGX_LOG_DEBUG_ALLOC, pool->log, 0,
76             "free: %p, unused: %uz", p, p->d.end - p->d.last);
77
78         if (n == NULL) {
79             break;
80         }
81     }
82
83     #endif
84
85     for (p = pool, n = pool->d.next; /* void */; p = n, n = n->d.next) {
86         ngx_free(p);
87
88         if (n == NULL) {
89             break;
90         }
91     }
92 }
93
94
95 void
96 ngx_reset_pool(ngx_pool_t *pool)
97 {
98     ngx_pool_t      *p;
99     ngx_pool_large_t *l;
100
101     for (l = pool->large; l; l = l->next) {
102         if (l->alloc) {
103             ngx_free(l->alloc);
104         }
105     }
106
107     for (p = pool; p; p = p->d.next) {
108         p->d.last = (u_char *) p + sizeof(ngx_pool_t);
109     }

```

```

110     p->d.failed = 0;
111 }
112
113 pool->current = pool;
114 pool->chain = NULL;
115 pool->large = NULL;
116 }
117
118
119 void *
120 ngx_palloc(ngx_pool_t *pool, size_t size)
121 {
122     u_char    *m;
123     ngx_pool_t *p;
124
125     if (size <= pool->max) {
126
127         p = pool->current;
128
129         do {
130             m = ngx_align_ptr(p->d.last, NGX_ALIGNMENT);
131
132             if ((size_t) (p->d.end - m) >= size) {
133                 p->d.last = m + size;
134
135                 return m;
136             }
137
138             p = p->d.next;
139
140         } while (p);
141
142         return ngx_palloc_block(pool, size);
143     }
144
145     return ngx_palloc_large(pool, size);
146 }
147
148
149 void *
150 ngx_pnalloc(ngx_pool_t *pool, size_t size)
151 {
152     u_char    *m;
153     ngx_pool_t *p;
154
155     if (size <= pool->max) {
156
157         p = pool->current;
158
159         do {
160             m = p->d.last;
161
162             if ((size_t) (p->d.end - m) >= size) {
163                 p->d.last = m + size;
164
165                 return m;
166             }
167
168             p = p->d.next;
169
170         } while (p);
171
172         return ngx_palloc_block(pool, size);
173     }
174
175     return ngx_palloc_large(pool, size);
176 }
177
178
179 static void *
180 ngx_palloc_block(ngx_pool_t *pool, size_t size)
181 {
182     u_char    *m;
183     size_t    psize;
184     ngx_pool_t *p, *new;
185

```

```

186     psize = (size_t) (pool->d.end - (u_char *) pool);
187
188     m = ngx_memalign(NGX_POOL_ALIGNMENT, psize, pool->log);
189     if (m == NULL) {
190         return NULL;
191     }
192
193     new = (ngx_pool_t *) m;
194
195     new->d.end = m + psize;
196     new->d.next = NULL;
197     new->d.failed = 0;
198
199     m += sizeof(ngx_pool_data_t);
200     m = ngx_align_ptr(m, NGX_ALIGNMENT);
201     new->d.last = m + size;
202
203     for (p = pool->current; p->d.next; p = p->d.next) {
204         if (p->d.failed++ > 4) {
205             pool->current = p->d.next;
206         }
207     }
208
209     p->d.next = new;
210
211     return m;
212 }
213
214
215 static void *
216 ngx_palloc_large(ngx_pool_t *pool, size_t size)
217 {
218     void *p;
219     ngx_uint_t n;
220     ngx_pool_large_t *large;
221
222     p = ngx_alloc(size, pool->log);
223     if (p == NULL) {
224         return NULL;
225     }
226
227     n = 0;
228
229     for (large = pool->large; large; large = large->next) {
230         if (large->alloc == NULL) {
231             large->alloc = p;
232             return p;
233         }
234
235         if (n++ > 3) {
236             break;
237         }
238     }
239
240     large = ngx_palloc(pool, sizeof(ngx_pool_large_t));
241     if (large == NULL) {
242         ngx_free(p);
243         return NULL;
244     }
245
246     large->alloc = p;
247     large->next = pool->large;
248     pool->large = large;
249
250     return p;
251 }
252
253
254 void *
255 ngx_pmemalign(ngx_pool_t *pool, size_t size, size_t alignment)
256 {
257     void *p;
258     ngx_pool_large_t *large;
259
260     p = ngx_memalign(alignment, size, pool->log);
261     if (p == NULL) {

```

```

262     return NULL;
263 }
264
265 large = ngx_palloc(pool, sizeof(ngx_pool_large_t));
266 if (large == NULL) {
267     ngx_free(p);
268     return NULL;
269 }
270
271 large->alloc = p;
272 large->next = pool->large;
273 pool->large = large;
274
275 return p;
276 }
277
278
279 ngx_int_t
280 ngx_pfree(ngx_pool_t *pool, void *p)
281 {
282     ngx_pool_large_t *l;
283
284     for (l = pool->large; l; l = l->next) {
285         if (p == l->alloc) {
286             ngx_log_debug1(NGX_LOG_DEBUG_ALLOC, pool->log, 0,
287                 "free: %p", l->alloc);
288             ngx_free(l->alloc);
289             l->alloc = NULL;
290
291             return NGX_OK;
292         }
293     }
294
295     return NGX_DECLINED;
296 }
297
298
299 void *
300 ngx_pcalloc(ngx_pool_t *pool, size_t size)
301 {
302     void *p;
303
304     p = ngx_palloc(pool, size);
305     if (p) {
306         ngx_memzero(p, size);
307     }
308
309     return p;
310 }
311
312
313 ngx_pool_cleanup_t *
314 ngx_pool_cleanup_add(ngx_pool_t *p, size_t size)
315 {
316     ngx_pool_cleanup_t *c;
317
318     c = ngx_palloc(p, sizeof(ngx_pool_cleanup_t));
319     if (c == NULL) {
320         return NULL;
321     }
322
323     if (size) {
324         c->data = ngx_palloc(p, size);
325         if (c->data == NULL) {
326             return NULL;
327         }
328
329     } else {
330         c->data = NULL;
331     }
332
333     c->handler = NULL;
334     c->next = p->cleanup;
335
336     p->cleanup = c;
337

```

```

338     ngx_log_debug1(NGX_LOG_DEBUG_ALLOC, p->log, 0, "add cleanup: %p", c);
339
340     return c;
341 }
342
343
344 void
345 ngx_pool_run_cleanup_file(ngx_pool_t *p, ngx_fd_t fd)
346 {
347     ngx_pool_cleanup_t *c;
348     ngx_pool_cleanup_file_t *cf;
349
350     for (c = p->cleanup; c; c = c->next) {
351         if (c->handler == ngx_pool_cleanup_file) {
352
353             cf = c->data;
354
355             if (cf->fd == fd) {
356                 c->handler(cf);
357                 c->handler = NULL;
358                 return;
359             }
360         }
361     }
362 }
363
364
365 void
366 ngx_pool_cleanup_file(void *data)
367 {
368     ngx_pool_cleanup_file_t *c = data;
369
370     ngx_log_debug1(NGX_LOG_DEBUG_ALLOC, c->log, 0, "file cleanup: fd:%d",
371                 c->fd);
372
373     if (ngx_close_file(c->fd) == NGX_FILE_ERROR) {
374         ngx_log_error(NGX_LOG_ALERT, c->log, ngx_errno,
375                 ngx_close_file_n " \"%s\" failed", c->name);
376     }
377 }
378
379
380 void
381 ngx_pool_delete_file(void *data)
382 {
383     ngx_pool_cleanup_file_t *c = data;
384
385     ngx_err_t err;
386
387     ngx_log_debug2(NGX_LOG_DEBUG_ALLOC, c->log, 0, "file cleanup: fd:%d %s",
388                 c->fd, c->name);
389
390     if (ngx_delete_file(c->name) == NGX_FILE_ERROR) {
391         err = ngx_errno;
392
393         if (err != NGX_ENOENT) {
394             ngx_log_error(NGX_LOG_CRIT, c->log, err,
395                 ngx_delete_file_n " \"%s\" failed", c->name);
396         }
397     }
398
399     if (ngx_close_file(c->fd) == NGX_FILE_ERROR) {
400         ngx_log_error(NGX_LOG_ALERT, c->log, ngx_errno,
401                 ngx_close_file_n " \"%s\" failed", c->name);
402     }
403 }
404
405
406 #if 0
407
408 static void *
409 ngx_get_cached_block(size_t size)
410 {
411     void *p;
412     ngx_cached_block_slot_t *slot;
413

```

```
414     if (ngx\_cycle->cache == NULL) {
415         return NULL;
416     }
417
418     slot = &ngx\_cycle->cache[(size + ngx\_pagesize - 1) / ngx\_pagesize];
419
420     slot->tries++;
421
422     if (slot->number) {
423         p = slot->block;
424         slot->block = slot->block->next;
425         slot->number--;
426         return p;
427     }
428
429     return NULL;
430 }
431
432 #endif
```

[One Level Up](#)

[Top Level](#)

# src/os/unix/nginx\_alloc.c - nginx-1.7.10

## Global variables defined

- [ngx\\_cacheline\\_size](#)
- [ngx\\_pagesize](#)
- [ngx\\_pagesize\\_shift](#)

## Functions defined

- [ngx\\_alloc](#)
- [ngx\\_calloc](#)
- [ngx\\_memalign](#)
- [ngx\\_memalign](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10
11
12 ngx_uint_t  ngx_pagesize;
13 ngx_uint_t  ngx_pagesize_shift;
14 ngx_uint_t  ngx_cacheline_size;
15
16
17 void *
18 ngx_alloc(size_t size, ngx_log_t *log)
19 {
20     void *p;
21
22     p = malloc(size);
23     if (p == NULL) {
24         ngx_log_error(NGX_LOG_EMERG, log, ngx_errno,
25                     "malloc(%uz) failed", size);
26     }
27
28     ngx_log_debug2(NGX_LOG_DEBUG_ALLOC, log, 0, "malloc: %p:%uz", p, size);
29
30     return p;
31 }
32
33
34 void *
35 ngx_calloc(size_t size, ngx_log_t *log)
36 {
37     void *p;
38
39     p = ngx_alloc(size, log);
40
41     if (p) {
42         ngx_memzero(p, size);
43     }
44
45     return p;

```

```

46 }
47
48
49 #if (NGX_HAVE_POSIX_MEMALIGN)
50
51 void *
52 ngx_memalign(size_t alignment, size_t size, ngx_log_t *log)
53 {
54     void *p;
55     int err;
56
57     err = posix_memalign(&p, alignment, size);
58
59     if (err) {
60         ngx_log_error(NGX_LOG_EMERG, log, err,
61             "posix_memalign(%uz, %uz) failed", alignment, size);
62         p = NULL;
63     }
64
65     ngx_log_debug3(NGX_LOG_DEBUG_ALLOC, log, 0,
66         "posix_memalign: %p:%uz @%uz", p, size, alignment);
67
68     return p;
69 }
70
71 #elif (NGX_HAVE_MEMALIGN)
72
73 void *
74 ngx_memalign(size_t alignment, size_t size, ngx_log_t *log)
75 {
76     void *p;
77
78     p = memalign(alignment, size);
79     if (p == NULL) {
80         ngx_log_error(NGX_LOG_EMERG, log, ngx_errno,
81             "memalign(%uz, %uz) failed", alignment, size);
82     }
83
84     ngx_log_debug3(NGX_LOG_DEBUG_ALLOC, log, 0,
85         "memalign: %p:%uz @%uz", p, size, alignment);
86
87     return p;
88 }
89
90 #endif

```

[One Level Up](#)

[Top Level](#)



## src/os/unix/nginx\_errno.h - nginx-1.7.10

### Data types defined

- [ngx\\_err\\_t](#)

### Macros defined

- [NGX\\_EACCES](#)
- [NGX\\_EADDRINUSE](#)
- [NGX\\_EAGAIN](#)
- [NGX\\_EAGAIN](#)
- [NGX\\_EBADF](#)
- [NGX\\_EBUSY](#)
- [NGX\\_ECANCELED](#)
- [NGX\\_ECHILD](#)
- [NGX\\_ECONNABORTED](#)
- [NGX\\_ECONNREFUSED](#)
- [NGX\\_ECONNRESET](#)
- [NGX\\_EEXIST](#)
- [NGX\\_EHOSTDOWN](#)
- [NGX\\_EHOSTUNREACH](#)
- [NGX\\_EILSEQ](#)
- [NGX\\_EINPROGRESS](#)
- [NGX\\_EINTR](#)
- [NGX\\_EINVAL](#)
- [NGX\\_EISDIR](#)
- [NGX\\_ELOOP](#)
- [NGX\\_EMFILE](#)
- [NGX\\_EMLINK](#)
- [NGX\\_ENAMETOOLONG](#)
- [NGX\\_ENETDOWN](#)
- [NGX\\_ENETUNREACH](#)
- [NGX\\_ENFILE](#)
- [NGX\\_ENOENT](#)

- [NGX\\_ENOMEM](#)
- [NGX\\_ENOMOREFILES](#)
- [NGX\\_ENOPATH](#)
- [NGX\\_ENOPROTOOPT](#)
- [NGX\\_ENOSPC](#)
- [NGX\\_ENOSYS](#)
- [NGX\\_ENOTCONN](#)
- [NGX\\_ENOTDIR](#)
- [NGX\\_EOPNOTSUPP](#)
- [NGX\\_EPERM](#)
- [NGX\\_EPIPE](#)
- [NGX\\_ESRCH](#)
- [NGX\\_ETIMEDOUT](#)
- [NGX\\_EXDEV](#)
- [\\_NGX\\_ERRNO\\_H\\_INCLUDED\\_](#)
- [ngx\\_errno](#)
- [ngx\\_set\\_errno](#)
- [ngx\\_set\\_socket\\_errno](#)
- [ngx\\_socket\\_errno](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_ERRNO\_H\_INCLUDED\_
9 #define \_NGX\_ERRNO\_H\_INCLUDED\_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 typedef int          ngx_err_t;
17
18 #define NGX_EPERM          EPERM
19 #define NGX_ENOENT         ENOENT
20 #define NGX_ENOPATH        ENOENT
21 #define NGX_ESRCH          ESRCH
22 #define NGX_EINTR          EINTR
23 #define NGX_ECHILD         ECHILD
24 #define NGX_ENOMEM         ENOMEM
25 #define NGX_EACCES         EACCES
26 #define NGX_EBUSY          EBUSY
27 #define NGX_EEXIST         EEXIST
28 #define NGX_EXDEV          EXDEV
29 #define NGX_ENOTDIR        ENOTDIR

```

```

30 #define NGX_EISDIR      EISDIR
31 #define NGX_EINVAL     EINVAL
32 #define NGX_ENFILE     ENFILE
33 #define NGX_EMFILE     EMFILE
34 #define NGX_ENOSPC     ENOSPC
35 #define NGX_EPIPE      EPIPE
36 #define NGX_EINPROGRESS EINPROGRESS
37 #define NGX_ENOPROTOPT ENOPROTOPT
38 #define NGX_EOPNOTSUPP EOPNOTSUPP
39 #define NGX_EADDRINUSE EADDRINUSE
40 #define NGX_ECONNABORTED ECONNABORTED
41 #define NGX_ECONNRESET ECONNRESET
42 #define NGX_ENOTCONN   ENOTCONN
43 #define NGX_ETIMEDOUT  ETIMEDOUT
44 #define NGX_ECONNREFUSED ECONNREFUSED
45 #define NGX_ENAMETOOLONG ENAMETOOLONG
46 #define NGX_ENETDOWN   ENETDOWN
47 #define NGX_ENETUNREACH ENETUNREACH
48 #define NGX_EHOSTDOWN  EHOSTDOWN
49 #define NGX_EHOSTUNREACH EHOSTUNREACH
50 #define NGX_ENOSYS     ENOSYS
51 #define NGX_ECANCELED  ECANCELED
52 #define NGX_EILSEQ     EILSEQ
53 #define NGX_ENOMOREFILES 0
54 #define NGX_ELOOP      ELOOP
55 #define NGX_EBADF      EBADF
56
57 #if (NGX_HAVE_OPENAT)
58 #define NGX_EMLINK     EMLINK
59 #endif
60
61 #if (__hpux__)
62 #define NGX_EAGAIN     EWOULDBLOCK
63 #else
64 #define NGX_EAGAIN     EAGAIN
65 #endif
66
67
68 #define ngx_errno      errno
69 #define ngx_socket_errno  errno
70 #define ngx_set_errno(err)  errno = err
71 #define ngx_set_socket_errno(err)  errno = err
72
73
74 u_char *ngx_strerror(ngx_err_t err, u_char *errstr, size_t size);
75 ngx_int_t ngx_strerror_init(void);
76
77
78 #endif /* NGX_ERRNO_H_INCLUDED */

```

[One Level Up](#)

[Top Level](#)

## src/os/unix/nginx\_errno.c - nginx-1.7.10

### Global variables defined

- [ngx\\_sys\\_errlist](#)
- [ngx\\_unknown\\_error](#)

### Functions defined

- [ngx\\_strerror](#)
- [ngx\\_strerror\\_init](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10
11
12 /*
13  * The strerror() messages are copied because:
14  *
15  * 1) strerror() and strerror_r() functions are not Async-Signal-Safe,
16  *    therefore, they cannot be used in signal handlers;
17  *
18  * 2) a direct sys_errlist[] array may be used instead of these functions,
19  *    but Linux linker warns about its usage:
20  *
21  * warning: `sys_errlist' is deprecated; use `strerror' or `strerror_r' instead
22  * warning: `sys_nerr' is deprecated; use `strerror' or `strerror_r' instead
23  *
24  *    causing false bug reports.
25  */
26
27
28 static ngx_str_t *ngx_sys_errlist;
29 static ngx_str_t  ngx_unknown_error = ngx_string("Unknown error");
30
31
32 u_char *
33 ngx_strerror(ngx_err_t err, u_char *errstr, size_t size)
34 {
35     ngx_str_t *msg;
36
37     msg = ((ngx_uint_t) err < NGX_SYS_NERR) ? &ngx_sys_errlist[err]:
38                                               &ngx_unknown_error;
39     size = ngx_min(size, msg->len);
40
41     return ngx_cpymem(errstr, msg->data, size);
42 }
43
44
45 ngx_int_t
46 ngx_strerror_init(void)
47 {
48     char      *msg;
49     u_char    *p;
50     size_t    len;
51     ngx_err_t  err;
52
```

```

53  /*
54  * ngx\_strerror\(\) is not ready to work at this stage, therefore,
55  * malloc\(\) is used and possible errors are logged using strerror\(\).
56  */
57
58  len = NGX_SYS_NERR * sizeof\(ngx\_str\_t\);
59
60  ngx\_sys\_errlist = malloc(len);
61  if (ngx\_sys\_errlist == NULL) {
62      goto failed;
63  }
64
65  for (err = 0; err < NGX_SYS_NERR; err++) {
66      msg = strerror(err);
67      len = ngx\_strlen(msg);
68
69      p = malloc(len);
70      if (p == NULL) {
71          goto failed;
72      }
73
74      ngx\_memcpy(p, msg, len);
75      ngx\_sys\_errlist[err].len = len;
76      ngx\_sys\_errlist[err].data = p;
77  }
78
79  return NGX\_OK;
80
81  failed:
82
83  err = errno;
84  ngx\_log\_stderr(0, "malloc\(%uz\) failed (%d: %s)", len, err, strerror(err));
85
86  return NGX\_ERROR;
87  }

```

[One Level Up](#)

[Top Level](#)

# src/core/nginx\_palloc.h - nginx-1.7.10

## Data types defined

- [ngx\\_pool\\_cleanup\\_file\\_t](#)
- [ngx\\_pool\\_cleanup\\_pt](#)
- [ngx\\_pool\\_cleanup\\_s](#)
- [ngx\\_pool\\_cleanup\\_t](#)
- [ngx\\_pool\\_data\\_t](#)
- [ngx\\_pool\\_large\\_s](#)
- [ngx\\_pool\\_large\\_t](#)
- [ngx\\_pool\\_s](#)

## Macros defined

- [NGX\\_DEFAULT\\_POOL\\_SIZE](#)
- [NGX\\_MAX\\_ALLOC\\_FROM\\_POOL](#)
- [NGX\\_MIN\\_POOL\\_SIZE](#)
- [NGX\\_POOL\\_ALIGNMENT](#)
- [\\_NGX\\_PALLOC\\_H\\_INCLUDED](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_PALLOC_H_INCLUDED
9 #define _NGX_PALLOC_H_INCLUDED
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 /*
17  * NGX\_MAX\_ALLOC\_FROM\_POOL should be (ngx\_pagesize - 1), i.e. 4095 on x86.
18  * On Windows NT it decreases a number of locked pages in a kernel.
19  */
20 #define NGX\_MAX\_ALLOC\_FROM\_POOL (ngx\_pagesize - 1)
21
22 #define NGX\_DEFAULT\_POOL\_SIZE (16 * 1024)
23
24 #define NGX\_POOL\_ALIGNMENT 16
25 #define NGX\_MIN\_POOL\_SIZE \
26     ngx\_align((sizeof(ngx\_pool\_t) + 2 * sizeof(ngx\_pool\_large\_t)), \
27             NGX\_POOL\_ALIGNMENT)
28
29
30 typedef void (*ngx\_pool\_cleanup\_pt)(void *data);
31
32 typedef struct ngx\_pool\_cleanup\_s ngx\_pool\_cleanup\_t;
```

```

33
34 struct ngx_pool_cleanup_s {
35     ngx_pool_cleanup_pt handler;
36     void *data;
37     ngx_pool_cleanup_t *next;
38 };
39
40
41 typedef struct ngx_pool_large_s ngx_pool_large_t;
42
43 struct ngx_pool_large_s {
44     ngx_pool_large_t *next;
45     void *alloc;
46 };
47
48
49 typedef struct {
50     u_char *last;
51     u_char *end;
52     ngx_pool_t *next;
53     ngx_uint_t failed;
54 } ngx_pool_data_t;
55
56
57 struct ngx_pool_s {
58     ngx_pool_data_t d;
59     size_t max;
60     ngx_pool_t *current;
61     ngx_chain_t *chain;
62     ngx_pool_large_t *large;
63     ngx_pool_cleanup_t *cleanup;
64     ngx_log_t *log;
65 };
66
67
68 typedef struct {
69     ngx_fd_t fd;
70     u_char *name;
71     ngx_log_t *log;
72 } ngx_pool_cleanup_file_t;
73
74
75 void *ngx_alloc(size_t size, ngx_log_t *log);
76 void *ngx_calloc(size_t size, ngx_log_t *log);
77
78 ngx_pool_t *ngx_create_pool(size_t size, ngx_log_t *log);
79 void ngx_destroy_pool(ngx_pool_t *pool);
80 void ngx_reset_pool(ngx_pool_t *pool);
81
82 void *ngx_palloc(ngx_pool_t *pool, size_t size);
83 void *ngx_pnalloc(ngx_pool_t *pool, size_t size);
84 void *ngx_pcalloc(ngx_pool_t *pool, size_t size);
85 void *ngx_pmemalign(ngx_pool_t *pool, size_t size, size_t alignment);
86 ngx_int_t ngx_pfree(ngx_pool_t *pool, void *p);
87
88
89 ngx_pool_cleanup_t *ngx_pool_cleanup_add(ngx_pool_t *p, size_t size);
90 void ngx_pool_run_cleanup_file(ngx_pool_t *p, ngx_fd_t fd);
91 void ngx_pool_cleanup_file(void *data);
92 void ngx_pool_delete_file(void *data);
93
94
95 #endif /* NGX_PALLOC_H_INCLUDED */

```

[One Level Up](#)

[Top Level](#)

# src/os/unix/nginx\_alloc.h - nginx-1.7.10

## Macros defined

- [\\_NGX\\_ALLOC\\_H\\_INCLUDED](#)
- [ngx\\_free](#)
- [ngx\\_memalign](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_ALLOC\_H\_INCLUDED
9 #define \_NGX\_ALLOC\_H\_INCLUDED
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 void *ngx\_alloc(size_t size, ngx\_log\_t *log);
17 void *ngx\_calloc(size_t size, ngx\_log\_t *log);
18
19 #define ngx\_free          free
20
21
22 /*
23  * Linux has memalign() or posix_memalign()
24  * Solaris has memalign()
25  * FreeBSD 7.0 has posix_memalign(), besides, early version's malloc()
26  * aligns allocations bigger than page size at the page boundary
27  */
28
29 #if (NGX_HAVE_POSIX_MEMALIGN || NGX_HAVE_MEMALIGN)
30
31 void *ngx\_memalign(size_t alignment, size_t size, ngx\_log\_t *log);
32
33 #else
34
35 #define ngx\_memalign(alignment, size, log) ngx\_alloc(size, log)
36
37 #endif
38
39
40 extern ngx\_uint\_t ngx\_pagesize;
41 extern ngx\_uint\_t ngx\_pagesize\_shift;
42 extern ngx\_uint\_t ngx\_cacheline\_size;
43
44
45 #endif /* \_NGX\_ALLOC\_H\_INCLUDED */
```



# src/os/unix/nginx\_solaris\_config.h - nginx-1.7.10

## Macros defined

- [NGX\\_ALIGNMENT](#)
- [NGX\\_HAVE\\_INHERITED\\_NONBLOCK](#)
- [NGX\\_HAVE\\_OS\\_SPECIFIC\\_INIT](#)
- [NGX\\_HAVE\\_SO\\_SNDLOWAT](#)
- [NGX\\_LISTEN\\_BACKLOG](#)
- [\\_FILE\\_OFFSET\\_BITS](#)
- [\\_NGX\\_SOLARIS\\_CONFIG\\_H\\_INCLUDED](#)
- [\\_REENTRANT](#)
- [ngx\\_debug\\_init](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_SOLARIS\_CONFIG\_H\_INCLUDED
9 #define \_NGX\_SOLARIS\_CONFIG\_H\_INCLUDED
10
11
12 #ifndef \_REENTRANT
13 #define \_REENTRANT
14 #endif
15
16 #define \_FILE\_OFFSET\_BITS 64 /* must be before <sys/types.h> */
17
18 #include <sys/types.h>
19 #include <sys/time.h>
20 #include <unistd.h>
21 #include <stdarg.h>
22 #include <stddef.h> /* offsetof() */
23 #include <stdio.h>
24 #include <stdlib.h>
25 #include <ctype.h>
26 #include <errno.h>
27 #include <string.h>
28 #include <signal.h>
29 #include <pwd.h>
30 #include <grp.h>
31 #include <dirent.h>
32 #include <glob.h>
33 #include <time.h>
34 #include <sys/statvfs.h> /* statvfs() */
35
36 #include <sys/filio.h> /* FIONBIO */
37 #include <sys/uio.h>
38 #include <sys/stat.h>
39 #include <fcntl.h>
40
41 #include <sys/wait.h>
42 #include <sys/mman.h>
43 #include <sys/resource.h>
44 #include <sched.h>
```

```

45
46 #include <sys/socket.h>
47 #include <netinet/in.h>
48 #include <netinet/tcp.h>          /* TCP_NODELAY */
49 #include <arpa/inet.h>
50 #include <netdb.h>
51 #include <sys/un.h>
52
53 #include <sys/systeminfo.h>
54 #include <limits.h>              /* IOV_MAX */
55 #include <inttypes.h>
56 #include <crypt.h>
57
58 #define NGX_ALIGNMENT  _MAX_ALIGNMENT
59
60 #include <ngx_auto_config.h>
61
62
63 #if (NGX_HAVE_POSIX_SEM)
64 #include <semaphore.h>
65 #endif
66
67
68 #if (NGX_HAVE_POLL)
69 #include <poll.h>
70 #endif
71
72
73 #if (NGX_HAVE_DEVPOLL)
74 #include <sys/ioctl.h>
75 #include <sys/devpoll.h>
76 #endif
77
78
79 #if (NGX_HAVE_EVENTPORT)
80 #include <port.h>
81 #endif
82
83
84 #if (NGX_HAVE_SENDFILE)
85 #include <sys/sendfile.h>
86 #endif
87
88
89 #define NGX_LISTEN_BACKLOG      511
90
91
92 #ifndef NGX_HAVE_INHERITED_NONBLOCK
93 #define NGX_HAVE_INHERITED_NONBLOCK  1
94 #endif
95
96
97 #ifndef NGX_HAVE_SO_SNDLOWAT
98 /* setsockopt(SO_SNDLOWAT) returns ENOPROTOPT */
99 #define NGX_HAVE_SO_SNDLOWAT        0
100 #endif
101
102
103 #define NGX_HAVE_OS_SPECIFIC_INIT    1
104 #define ngx_debug_init()
105
106
107 extern char **environ;
108
109
110 #endif /* NGX_SOLARIS_CONFIG_H_INCLUDED */

```

[One Level Up](#)

[Top Level](#)

## src/os/unix/nginx\_posix\_config.h - nginx-1.7.10

### Data types defined

- [ngx\\_aiocb\\_t](#)

### Macros defined

- [CMSG\\_DATA](#)
- [CMSG\\_DATA](#)
- [CMSG\\_LEN](#)
- [CMSG\\_LEN](#)
- [CMSG\\_SPACE](#)
- [CMSG\\_SPACE](#)
- [IOV\\_MAX](#)
- [NGX\\_BROKEN\\_SCM\\_RIGHTS](#)
- [NGX\\_LISTEN\\_BACKLOG](#)
- [\\_HPUX\\_ALT\\_XOPEN\\_SOCKET\\_API](#)
- [\\_NGX\\_POSIX\\_CONFIG\\_H\\_INCLUDED](#)
- [\\_REENTRANT](#)
- [\\_XOPEN\\_SOURCE](#)
- [\\_XOPEN\\_SOURCE\\_EXTENDED](#)
- [ngx\\_debug\\_init](#)
- [timezonevar](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_POSIX\_CONFIG\_H\_INCLUDED
9 #define \_NGX\_POSIX\_CONFIG\_H\_INCLUDED
10
11
12 #if (NGX_HPUX)
13 #define \_XOPEN\_SOURCE
14 #define \_XOPEN\_SOURCE\_EXTENDED 1
15 #define \_HPUX\_ALT\_XOPEN\_SOCKET\_API
16 #endif
17
18
19 #if (NGX_TRU64)
20 #define \_REENTRANT
21 #endif
22
23
```

```

24 #ifdef __CYGWIN__
25 #define timezonevar      /* timezone is variable */
26 #define NGX_BROKEN_SCM_RIGHTS 1
27 #endif
28
29
30 #include <sys/types.h>
31 #include <sys/time.h>
32 #if (NGX_HAVE_UNISTD_H)
33 #include <unistd.h>
34 #endif
35 #if (NGX_HAVE_INTTYPES_H)
36 #include <inttypes.h>
37 #endif
38 #include <stdarg.h>
39 #include <stddef.h>      /* offsetof() */
40 #include <stdio.h>
41 #include <stdlib.h>
42 #include <ctype.h>
43 #include <errno.h>
44 #include <string.h>
45 #include <signal.h>
46 #include <pwd.h>
47 #include <grp.h>
48 #include <dirent.h>
49 #include <glob.h>
50 #include <time.h>
51 #if (NGX_HAVE_SYS_PARAM_H)
52 #include <sys/param.h>  /* statfs() */
53 #endif
54 #if (NGX_HAVE_SYS_MOUNT_H)
55 #include <sys/mount.h>  /* statfs() */
56 #endif
57 #if (NGX_HAVE_SYS_STATVFS_H)
58 #include <sys/statvfs.h> /* statvfs() */
59 #endif
60
61 #if (NGX_HAVE_SYS_FILIO_H)
62 #include <sys/filio.h>  /* FIONBIO */
63 #endif
64 #include <sys/ioctl.h>  /* FIONBIO */
65
66 #include <sys/uio.h>
67 #include <sys/stat.h>
68 #include <fcntl.h>
69
70 #include <sys/wait.h>
71 #include <sys/mman.h>
72 #include <sys/resource.h>
73 #include <sched.h>
74
75 #include <sys/socket.h>
76 #include <netinet/in.h>
77 #include <netinet/tcp.h> /* TCP_NODELAY */
78 #include <arpa/inet.h>
79 #include <netdb.h>
80 #include <sys/un.h>
81
82 #if (NGX_HAVE_LIMITS_H)
83 #include <limits.h>     /* IOV_MAX */
84 #endif
85
86 #ifdef __CYGWIN__
87 #include <malloc.h>     /* memalign() */
88 #endif
89
90 #if (NGX_HAVE_CRYPT_H)
91 #include <crypt.h>
92 #endif
93
94
95 #ifndef IOV_MAX
96 #define IOV_MAX 16
97 #endif
98
99

```

```

100 #include <ngx_auto_config.h>
101
102
103 #if (NGX_HAVE_POSIX_SEM)
104 #include <semaphore.h>
105 #endif
106
107
108 #if (NGX_HAVE_POLL)
109 #include <poll.h>
110 #endif
111
112
113 #if (NGX_HAVE_KQUEUE)
114 #include <sys/event.h>
115 #endif
116
117
118 #if (NGX_HAVE_DEVPOLL)
119 #include <sys/ioctl.h>
120 #include <sys/devpoll.h>
121 #endif
122
123
124 #if (NGX_HAVE_FILE_AIO)
125 #include <aio.h>
126 typedef struct aiocb ngx_aiocb_t;
127 #endif
128
129
130 #define NGX_LISTEN_BACKLOG 511
131
132 #define ngx_debug_init()
133
134
135 #if ( __FreeBSD__ ) && ( __FreeBSD_version < 400017 )
136
137 #include <sys/param.h>          /* ALIGN() */
138
139 /*
140 * FreeBSD 3.x has no CMSG_SPACE() and CMSG_LEN() and has the broken CMSG_DATA()
141 */
142
143 #undef CMSG_SPACE
144 #define CMSG_SPACE(l)          (ALIGN(sizeof(struct cmsghdr)) + ALIGN(l))
145
146 #undef CMSG_LEN
147 #define CMSG_LEN(l)           (ALIGN(sizeof(struct cmsghdr)) + (l))
148
149 #undef CMSG_DATA
150 #define CMSG_DATA(cmsg)       ((u_char *) (cmsg) + ALIGN(sizeof(struct cmsghdr)))
151
152 #endif
153
154
155 extern char **environ;
156
157
158 #endif /* NGX_POSIX_CONFIG_H_INCLUDED */

```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_times.c - nginx-1.7.10

### Global variables defined

- [cached\\_err\\_log\\_time](#)
- [cached\\_gmtoff](#)
- [cached\\_http\\_log\\_iso8601](#)
- [cached\\_http\\_log\\_time](#)
- [cached\\_http\\_time](#)
- [cached\\_syslog\\_time](#)
- [cached\\_time](#)
- [months](#)
- [ngx\\_cached\\_err\\_log\\_time](#)
- [ngx\\_cached\\_http\\_log\\_iso8601](#)
- [ngx\\_cached\\_http\\_log\\_time](#)
- [ngx\\_cached\\_http\\_time](#)
- [ngx\\_cached\\_syslog\\_time](#)
- [ngx\\_cached\\_time](#)
- [ngx\\_current\\_msec](#)
- [ngx\\_time\\_lock](#)
- [slot](#)
- [week](#)

### Functions defined

- [ngx\\_gmtime](#)
- [ngx\\_http\\_cookie\\_time](#)
- [ngx\\_http\\_time](#)
- [ngx\\_next\\_time](#)
- [ngx\\_time\\_init](#)
- [ngx\\_time\\_sigsafe\\_update](#)
- [ngx\\_time\\_update](#)

### Macros defined

- [NGX\\_TIME\\_SLOTS](#)

### Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10
11
12 /*
13  * The time may be updated by signal handler or by several threads.
14  * The time update operations are rare and require to hold the ngx_time_lock.
15  * The time read operations are frequent, so they are lock-free and get time
16  * values and strings from the current slot. Thus thread may get the corrupted
17  * values only if it is preempted while copying and then it is not scheduled
18  * to run more than NGX_TIME_SLOTS seconds.
19  */
20
21 #define NGX_TIME_SLOTS    64
22
23 static ngx_uint_t          slot;
24 static ngx_atomic_t       ngx_time_lock;
25
26 volatile ngx_msec_t       ngx_current_msec;
27 volatile ngx_time_t       *ngx_cached_time;
28 volatile ngx_str_t        ngx_cached_err_log_time;
29 volatile ngx_str_t        ngx_cached_http_time;
30 volatile ngx_str_t        ngx_cached_http_log_time;
31 volatile ngx_str_t        ngx_cached_http_log_iso8601;
32 volatile ngx_str_t        ngx_cached_syslog_time;
33
34 #if !(NGX_WIN32)
35
36 /*
37  * localtime() and localtime_r() are not Async-Signal-Safe functions, therefore,
38  * they must not be called by a signal handler, so we use the cached
39  * GMT offset value. Fortunately the value is changed only two times a year.
40  */
41
42 static ngx_int_t          cached_gmtoff;
43 #endif
44
45 static ngx_time_t         cached_time[NGX_TIME_SLOTS];
46 static u_char              cached_err_log_time[NGX_TIME_SLOTS]
47                             [sizeof("1970/09/28 12:00:00")];
48 static u_char              cached_http_time[NGX_TIME_SLOTS]
49                             [sizeof("Mon, 28 Sep 1970 06:00:00 GMT")];
50 static u_char              cached_http_log_time[NGX_TIME_SLOTS]
51                             [sizeof("28/Sep/1970:12:00:00 +0600")];
52 static u_char              cached_http_log_iso8601[NGX_TIME_SLOTS]
53                             [sizeof("1970-09-28T12:00:00+06:00")];
54 static u_char              cached_syslog_time[NGX_TIME_SLOTS]
55                             [sizeof("Sep 28 12:00:00")];
56
57
58 static char *week[] = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
59 static char *months[] = { "Jan", "Feb", "Mar", "Apr", "May", "Jun",
60                           "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };
61
62 void
63 ngx_time_init(void)
64 {
65     ngx_cached_err_log_time.len = sizeof("1970/09/28 12:00:00") - 1;
66     ngx_cached_http_time.len = sizeof("Mon, 28 Sep 1970 06:00:00 GMT") - 1;
67     ngx_cached_http_log_time.len = sizeof("28/Sep/1970:12:00:00 +0600") - 1;
68     ngx_cached_http_log_iso8601.len = sizeof("1970-09-28T12:00:00+06:00") - 1;
69     ngx_cached_syslog_time.len = sizeof("Sep 28 12:00:00") - 1;
70
71     ngx_cached_time = &cached_time[0];
72
73     ngx_time_update();
74 }
75
76

```

```

77 void
78 ngx_time_update(void)
79 {
80     u_char          *p0, *p1, *p2, *p3, *p4;
81     ngx_tm_t        tm, gmt;
82     time_t          sec;
83     ngx_uint_t      msec;
84     ngx_time_t      *tp;
85     struct timeval   tv;
86
87     if (!ngx_trylock(&ngx_time_lock)) {
88         return;
89     }
90
91     ngx_gettimeofday(&tv);
92
93     sec = tv.tv_sec;
94     msec = tv.tv_usec / 1000;
95
96     ngx_current_msec = (ngx_msec_t) sec * 1000 + msec;
97
98     tp = &cached_time[slot];
99
100    if (tp->sec == sec) {
101        tp->msec = msec;
102        ngx_unlock(&ngx_time_lock);
103        return;
104    }
105
106    if (slot == NGX_TIME_SLOTS - 1) {
107        slot = 0;
108    } else {
109        slot++;
110    }
111
112    tp = &cached_time[slot];
113
114    tp->sec = sec;
115    tp->msec = msec;
116
117    ngx_gmtime(sec, &gmt);
118
119
120    p0 = &cached_http_time[slot][0];
121
122    (void) ngx_sprintf(p0, "%s, %02d %s %4d %02d:%02d:%02d GMT",
123                    week[gmt.ngx_tm_wday], gmt.ngx_tm_mday,
124                    months[gmt.ngx_tm_mon - 1], gmt.ngx_tm_year,
125                    gmt.ngx_tm_hour, gmt.ngx_tm_min, gmt.ngx_tm_sec);
126
127    #if (NGX_HAVE_GETTIMEZONE)
128
129        tp->gmtoff = ngx_gettimezone();
130        ngx_gmtime(sec + tp->gmtoff * 60, &tm);
131
132    #elif (NGX_HAVE_GMTOFF)
133
134        ngx_localtime(sec, &tm);
135        cached_gmtoff = (ngx_int_t) (tm.ngx_tm_gmtoff / 60);
136        tp->gmtoff = cached_gmtoff;
137
138    #else
139
140        ngx_localtime(sec, &tm);
141        cached_gmtoff = ngx_timezone(tm.ngx_tm_isdst);
142        tp->gmtoff = cached_gmtoff;
143
144    #endif
145
146
147    p1 = &cached_err_log_time[slot][0];
148
149    (void) ngx_sprintf(p1, "%4d/%02d/%02d %02d:%02d:%02d",
150                    tm.ngx_tm_year, tm.ngx_tm_mon,
151                    tm.ngx_tm_mday, tm.ngx_tm_hour,
152                    tm.ngx_tm_min, tm.ngx_tm_sec);

```



```

153
154
155 p2 = &cached_http_log_time[slot][0];
156
157 (void) ngx_sprintf(p2, "%02d/%s/%d:%02d:%02d:%02d %c%02d%02d",
158                 tm.ngx_tm_mday, months[tm.ngx_tm_mon - 1],
159                 tm.ngx_tm_year, tm.ngx_tm_hour,
160                 tm.ngx_tm_min, tm.ngx_tm_sec,
161                 tp->gmtoff < 0 ? '-' : '+',
162                 ngx_abs(tp->gmtoff / 60), ngx_abs(tp->gmtoff % 60));
163
164 p3 = &cached_http_log_iso8601[slot][0];
165
166 (void) ngx_sprintf(p3, "%4d-%02d-%02dT%02d:%02d:%02d%c%02d:%02d",
167                 tm.ngx_tm_year, tm.ngx_tm_mon,
168                 tm.ngx_tm_mday, tm.ngx_tm_hour,
169                 tm.ngx_tm_min, tm.ngx_tm_sec,
170                 tp->gmtoff < 0 ? '-' : '+',
171                 ngx_abs(tp->gmtoff / 60), ngx_abs(tp->gmtoff % 60));
172
173 p4 = &cached_syslog_time[slot][0];
174
175 (void) ngx_sprintf(p4, "%s %2d %02d:%02d:%02d",
176                 months[tm.ngx_tm_mon - 1], tm.ngx_tm_mday,
177                 tm.ngx_tm_hour, tm.ngx_tm_min, tm.ngx_tm_sec);
178
179 ngx_memory_barrier();
180
181 ngx_cached_time = tp;
182 ngx_cached_http_time.data = p0;
183 ngx_cached_err_log_time.data = p1;
184 ngx_cached_http_log_time.data = p2;
185 ngx_cached_http_log_iso8601.data = p3;
186 ngx_cached_syslog_time.data = p4;
187
188 ngx_unlock(&ngx_time_lock);
189 }
190
191
192 #if !(NGX_WIN32)
193
194 void
195 ngx_time_sigsafe_update(void)
196 {
197     u_char          *p, *p2;
198     ngx_tm_t        tm;
199     time_t          sec;
200     ngx_time_t      *tp;
201     struct timeval  tv;
202
203     if (!ngx_trylock(&ngx_time_lock)) {
204         return;
205     }
206
207     ngx_gettimeofday(&tv);
208
209     sec = tv.tv_sec;
210
211     tp = &cached_time[slot];
212
213     if (tp->sec == sec) {
214         ngx_unlock(&ngx_time_lock);
215         return;
216     }
217
218     if (slot == NGX_TIME_SLOTS - 1) {
219         slot = 0;
220     } else {
221         slot++;
222     }
223
224     tp = &cached_time[slot];
225
226     tp->sec = 0;
227
228     ngx_gmtime(sec + cached_gmtoff * 60, &tm);

```

```

229 p = &cached_err_log_time[slot][0];
230
231
232 (void) ngx_sprintf(p, "%4d/%02d/%02d %02d:%02d:%02d",
233                 tm.ngx_tm_year, tm.ngx_tm_mon,
234                 tm.ngx_tm_mday, tm.ngx_tm_hour,
235                 tm.ngx_tm_min, tm.ngx_tm_sec);
236
237 p2 = &cached_syslog_time[slot][0];
238
239 (void) ngx_sprintf(p2, "%s %2d %02d:%02d:%02d",
240                 months[tm.ngx_tm_mon - 1], tm.ngx_tm_mday,
241                 tm.ngx_tm_hour, tm.ngx_tm_min, tm.ngx_tm_sec);
242
243 ngx_memory_barrier();
244
245 ngx_cached_err_log_time.data = p;
246 ngx_cached_syslog_time.data = p2;
247
248 ngx_unlock(&ngx_time_lock);
249 }
250
251 #endif
252
253
254 u_char *
255 ngx_http_time(u_char *buf, time_t t)
256 {
257     ngx_tm_t  tm;
258
259     ngx_gmtime(t, &tm);
260
261     return ngx_sprintf(buf, "%s, %02d %s %4d %02d:%02d:%02d GMT",
262                     week[tm.ngx_tm_wday],
263                     tm.ngx_tm_mday,
264                     months[tm.ngx_tm_mon - 1],
265                     tm.ngx_tm_year,
266                     tm.ngx_tm_hour,
267                     tm.ngx_tm_min,
268                     tm.ngx_tm_sec);
269 }
270
271
272 u_char *
273 ngx_http_cookie_time(u_char *buf, time_t t)
274 {
275     ngx_tm_t  tm;
276
277     ngx_gmtime(t, &tm);
278
279     /*
280      * Netscape 3.x does not understand 4-digit years at all and
281      * 2-digit years more than "37"
282      */
283
284     return ngx_sprintf(buf,
285                     (tm.ngx_tm_year > 2037) ?
286                         "%s, %02d-%s-%d %02d:%02d:%02d GMT":
287                         "%s, %02d-%s-%02d %02d:%02d:%02d GMT",
288                     week[tm.ngx_tm_wday],
289                     tm.ngx_tm_mday,
290                     months[tm.ngx_tm_mon - 1],
291                     (tm.ngx_tm_year > 2037) ? tm.ngx_tm_year :
292                         tm.ngx_tm_year % 100,
293                     tm.ngx_tm_hour,
294                     tm.ngx_tm_min,
295                     tm.ngx_tm_sec);
296 }
297
298
299 void
300 ngx_gmtime(time_t t, ngx_tm_t *tp)
301 {
302     ngx_int_t  yday;
303     ngx_uint_t n, sec, min, hour, mday, mon, year, wday, days, leap;
304

```

```

305  /* the calculation is valid for positive time_t only */
306
307  n = (ngx_uint_t) t;
308
309  days = n / 86400;
310
311  /* January 1, 1970 was Thursday */
312
313  wday = (4 + days) % 7;
314
315  n %= 86400;
316  hour = n / 3600;
317  n %= 3600;
318  min = n / 60;
319  sec = n % 60;
320
321  /*
322   * the algorithm based on Gauss' formula,
323   * see src/http/ngx_http_parse_time.c
324   */
325
326  /* days since March 1, 1 BC */
327  days = days - (31 + 28) + 719527;
328
329  /*
330   * The "days" should be adjusted to 1 only, however, some March 1st's go
331   * to previous year, so we adjust them to 2. This causes also shift of the
332   * last February days to next year, but we catch the case when "yday"
333   * becomes negative.
334   */
335
336  year = (days + 2) * 400 / (365 * 400 + 100 - 4 + 1);
337
338  yday = days - (365 * year + year / 4 - year / 100 + year / 400);
339
340  if (yday < 0) {
341      leap = (year % 4 == 0) && (year % 100 || (year % 400 == 0));
342      yday = 365 + leap + yday;
343      year--;
344  }
345
346  /*
347   * The empirical formula that maps "yday" to month.
348   * There are at least 10 variants, some of them are:
349   *     mon = (yday + 31) * 15 / 459
350   *     mon = (yday + 31) * 17 / 520
351   *     mon = (yday + 31) * 20 / 612
352   */
353
354  mon = (yday + 31) * 10 / 306;
355
356  /* the Gauss' formula that evaluates days before the month */
357
358  mday = yday - (367 * mon / 12 - 30) + 1;
359
360  if (yday >= 306) {
361
362      year++;
363      mon -= 10;
364
365      /*
366       * there is no "yday" in Win32 SYSTEMTIME
367       *
368       * yday -= 306;
369       */
370
371  } else {
372
373      mon += 2;
374
375      /*
376       * there is no "yday" in Win32 SYSTEMTIME
377       *
378       * yday += 31 + 28 + leap;
379       */
380  }

```

```

381 tp->ngx_tm_sec = (ngx_tm_sec_t) sec;
382 tp->ngx_tm_min = (ngx_tm_min_t) min;
383 tp->ngx_tm_hour = (ngx_tm_hour_t) hour;
384 tp->ngx_tm_mday = (ngx_tm_mday_t) mday;
385 tp->ngx_tm_mon = (ngx_tm_mon_t) mon;
386 tp->ngx_tm_year = (ngx_tm_year_t) year;
387 tp->ngx_tm_wday = (ngx_tm_wday_t) wday;
388 }
389 }
390
391 time_t
392 ngx_next_time(time_t when)
393 {
394     time_t    now, next;
395     struct tm  tm;
396
397     now = ngx_time();
398
399     ngx_libc_localtime(now, &tm);
400
401     tm.tm_hour = (int) (when / 3600);
402     when %= 3600;
403     tm.tm_min = (int) (when / 60);
404     tm.tm_sec = (int) (when % 60);
405
406     next = mktime(&tm);
407
408     if (next == -1) {
409         return -1;
410     }
411
412     if (next - now > 0) {
413         return next;
414     }
415
416     tm.tm_mday++;
417
418     /* mktime() should normalize a date (Jan 32, etc) */
419
420     next = mktime(&tm);
421
422     if (next != -1) {
423         return next;
424     }
425
426     return -1;
427 }
428 }

```

[One Level Up](#)

[Top Level](#)



## Functions defined

- [ngx\\_atomic\\_cmp\\_set](#)
- [ngx\\_atomic\\_fetch\\_add](#)

## Macros defined

- [AO\\_REQUIRE\\_CAS](#)
- [NGX\\_ATOMIC\\_T\\_LEN](#)
- [NGX\\_ATOMIC\\_T\\_LEN](#)
- [NGX\\_ATOMIC\\_T\\_LEN](#)
- [NGX\\_ATOMIC\\_T\\_LEN](#)
- [NGX\\_ATOMIC\\_T\\_LEN](#)
- [NGX\\_ATOMIC\\_T\\_LEN](#)
- [NGX\\_ATOMIC\\_T\\_LEN](#)
- [NGX\\_ATOMIC\\_T\\_LEN](#)
- [NGX\\_ATOMIC\\_T\\_LEN](#)
- [NGX\\_ATOMIC\\_T\\_LEN](#)
- [NGX\\_ATOMIC\\_T\\_LEN](#)
- [NGX\\_ATOMIC\\_T\\_LEN](#)
- [NGX\\_ATOMIC\\_T\\_LEN](#)
- [NGX\\_ATOMIC\\_T\\_LEN](#)
- [NGX\\_ATOMIC\\_T\\_LEN](#)
- [NGX\\_ATOMIC\\_T\\_LEN](#)
- [NGX\\_ATOMIC\\_T\\_LEN](#)
- [NGX\\_ATOMIC\\_T\\_LEN](#)
- [NGX\\_ATOMIC\\_T\\_LEN](#)
- [NGX\\_ATOMIC\\_T\\_LEN](#)
- [NGX\\_ATOMIC\\_T\\_LEN](#)
- [NGX\\_HAVE\\_ATOMIC\\_OPS](#)
- [NGX\\_HAVE\\_ATOMIC\\_OPS](#)
- [NGX\\_HAVE\\_ATOMIC\\_OPS](#)
- [NGX\\_HAVE\\_ATOMIC\\_OPS](#)
- [NGX\\_HAVE\\_ATOMIC\\_OPS](#)
- [NGX\\_HAVE\\_ATOMIC\\_OPS](#)
- [NGX\\_HAVE\\_ATOMIC\\_OPS](#)
- [NGX\\_HAVE\\_ATOMIC\\_OPS](#)
- [NGX\\_HAVE\\_ATOMIC\\_OPS](#)
- [NGX\\_HAVE\\_ATOMIC\\_OPS](#)
- [NGX\\_HAVE\\_ATOMIC\\_OPS](#)
- [NGX\\_HAVE\\_ATOMIC\\_OPS](#)
- [NGX\\_HAVE\\_ATOMIC\\_OPS](#)
- [NGX\\_HAVE\\_ATOMIC\\_OPS](#)
- [NGX\\_HAVE\\_ATOMIC\\_OPS](#)
- [NGX\\_HAVE\\_ATOMIC\\_OPS](#)
- [NGX\\_HAVE\\_ATOMIC\\_OPS](#)
- [NGX\\_HAVE\\_ATOMIC\\_OPS](#)
- [NGX\\_HAVE\\_ATOMIC\\_OPS](#)
- [\\_NGX\\_ATOMIC\\_H\\_INCLUDED](#)
- [ngx\\_atomic\\_cmp\\_set](#)
- [ngx\\_atomic\\_cmp\\_set](#)

- [ngx\\_atomic\\_cmp\\_set](#)
- [ngx\\_atomic\\_cmp\\_set](#)
- [ngx\\_atomic\\_fetch\\_add](#)
- [ngx\\_atomic\\_fetch\\_add](#)
- [ngx\\_atomic\\_fetch\\_add](#)
- [ngx\\_atomic\\_fetch\\_add](#)
- [ngx\\_atomic\\_fetch\\_add](#)
- [ngx\\_cpu\\_pause](#)
- [ngx\\_cpu\\_pause](#)
- [ngx\\_cpu\\_pause](#)
- [ngx\\_cpu\\_pause](#)
- [ngx\\_cpu\\_pause](#)
- [ngx\\_cpu\\_pause](#)
- [ngx\\_memory\\_barrier](#)
- [ngx\\_memory\\_barrier](#)
- [ngx\\_memory\\_barrier](#)
- [ngx\\_memory\\_barrier](#)
- [ngx\\_memory\\_barrier](#)
- [ngx\\_memory\\_barrier](#)
- [ngx\\_memory\\_barrier](#)
- [ngx\\_memory\\_barrier](#)
- [ngx\\_trylock](#)
- [ngx\\_unlock](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef NGX_ATOMIC_H_INCLUDED
9 #define _NGX_ATOMIC_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 #if (NGX_HAVE_LIBATOMIC)
17
18 #define AO_REQUIRE_CAS
19 #include <atomic_ops.h>
20
21 #define NGX_HAVE_ATOMIC_OPS 1
22
23 typedef long ngx_atomic_int_t;
24 typedef AO_t ngx_atomic_uint_t;
25 typedef volatile ngx_atomic_uint_t ngx_atomic_t;
26
27 #if (NGX_PTR_SIZE == 8)
28 #define NGX_ATOMIC_T_LEN (sizeof("-9223372036854775808") - 1)
29 #else

```

```

30 #define NGX_ATOMIC_T_LEN          (sizeof("-2147483648") - 1)
31 #endif
32
33 #define ngx_atomic_cmp_set(lock, old, new)          \
34     AO_compare_and_swap(lock, old, new)           \
35 #define ngx_atomic_fetch_add(value, add)          \
36     AO_fetch_and_add(value, add)                  \
37 #define ngx_memory_barrier()          AO_nop()
38 #define ngx_cpu_pause()
39
40
41 #elif (NGX_DARWIN_ATOMIC)
42
43 /*
44  * use Darwin 8 atomic(3) and barrier(3) operations
45  * optimized at run-time for UP and SMP
46  */
47
48 #include <libkern/OSAtomic.h>
49
50 /* "bool" conflicts with perl's CORE/handy.h */
51 #if 0
52 #undef bool
53 #endif
54
55 #define NGX_HAVE_ATOMIC_OPS 1
56
57 #if (NGX_PTR_SIZE == 8)
58
59 typedef int64_t          ngx_atomic_int_t;
60 typedef uint64_t         ngx_atomic_uint_t;
61 #define NGX_ATOMIC_T_LEN          (sizeof("-9223372036854775808") - 1)
62
63 #define ngx_atomic_cmp_set(lock, old, new)          \
64     OSAtomicCompareAndSwap64Barrier(old, new, (int64_t *) lock) \
65 #define ngx_atomic_fetch_add(value, add)          \
66     (OSAtomicAdd64(add, (int64_t *) value) - add) \
67 #else
68
69 typedef int32_t          ngx_atomic_int_t;
70 typedef uint32_t         ngx_atomic_uint_t;
71 #define NGX_ATOMIC_T_LEN          (sizeof("-2147483648") - 1)
72
73 #define ngx_atomic_cmp_set(lock, old, new)          \
74     OSAtomicCompareAndSwap32Barrier(old, new, (int32_t *) lock) \
75 #define ngx_atomic_fetch_add(value, add)          \
76     (OSAtomicAdd32(add, (int32_t *) value) - add) \
77 #endif
78
79 #define ngx_memory_barrier()          OSMemoryBarrier()
80 #define ngx_cpu_pause()
81
82 typedef volatile ngx\_atomic\_uint\_t ngx_atomic_t;
83
84 #elif (NGX_HAVE_GCC_ATOMIC)
85
86 /* GCC 4.1 builtin atomic operations */
87
88 #define NGX_HAVE_ATOMIC_OPS 1
89
90 typedef long             ngx_atomic_int_t;
91 typedef unsigned long    ngx_atomic_uint_t;
92
93 #if (NGX_PTR_SIZE == 8)
94 #define NGX_ATOMIC_T_LEN          (sizeof("-9223372036854775808") - 1)
95 #else
96 #define NGX_ATOMIC_T_LEN          (sizeof("-2147483648") - 1)
97 #endif
98 #endif
99
100
101
102
103
104
105

```



```

106 typedef volatile ngx_atomic_uint_t ngx_atomic_t;
107
108
109 #define ngx_atomic_cmp_set(lock, old, set) \
110     __sync_bool_compare_and_swap(lock, old, set) \
111
112 #define ngx_atomic_fetch_add(value, add) \
113     __sync_fetch_and_add(value, add) \
114
115 #define ngx_memory_barrier() __sync_synchronize()
116
117 #if ( __i386__ || __i386 || __amd64__ || __amd64 )
118 #define ngx_cpu_pause() __asm__ ("pause")
119 #else
120 #define ngx_cpu_pause()
121 #endif
122
123
124 #elif ( __i386__ || __i386 )
125
126 typedef int32_t ngx_atomic_int_t;
127 typedef uint32_t ngx_atomic_uint_t;
128 typedef volatile ngx_atomic_uint_t ngx_atomic_t;
129 #define NGX_ATOMIC_T_LEN (sizeof("-2147483648") - 1)
130
131
132 #if ( __SUNPRO_C )
133
134 #define NGX_HAVE_ATOMIC_OPS 1
135
136 ngx_atomic_uint_t
137 ngx_atomic_cmp_set(ngx_atomic_t *lock, ngx_atomic_uint_t old,
138     ngx_atomic_uint_t set);
139
140 ngx_atomic_int_t
141 ngx_atomic_fetch_add(ngx_atomic_t *value, ngx_atomic_int_t add);
142
143 /*
144  * Sun Studio 12 exits with segmentation fault on '__asm ("pause")',
145  * so ngx_cpu_pause is declared in src/os/unix/nginx_sunpro_x86.il
146  */
147
148 void
149 ngx_cpu_pause(void);
150
151 /* the code in src/os/unix/nginx_sunpro_x86.il */
152
153 #define ngx_memory_barrier() __asm__ (".volatile"); __asm__ (".nonvolatile")
154
155
156 #else /* ( __GNUC__ || __INTEL_COMPILER ) */
157
158 #define NGX_HAVE_ATOMIC_OPS 1
159
160 #include "ngx_gcc_atomic_x86.h"
161
162 #endif
163
164
165 #elif ( __amd64__ || __amd64 )
166
167 typedef int64_t ngx_atomic_int_t;
168 typedef uint64_t ngx_atomic_uint_t;
169 typedef volatile ngx_atomic_uint_t ngx_atomic_t;
170 #define NGX_ATOMIC_T_LEN (sizeof("-9223372036854775808") - 1)
171
172
173 #if ( __SUNPRO_C )
174
175 #define NGX_HAVE_ATOMIC_OPS 1
176
177 ngx_atomic_uint_t
178 ngx_atomic_cmp_set(ngx_atomic_t *lock, ngx_atomic_uint_t old,
179     ngx_atomic_uint_t set);
180
181 ngx_atomic_int_t

```

```

182 ngx_atomic_fetch_add(ngx_atomic_t *value, ngx_atomic_int_t add);
183
184 /*
185  * Sun Studio 12 exits with segmentation fault on '__asm ("pause")',
186  * so ngx_cpu_pause is declared in src/os/unix/nginx_sunpro_amd64.il
187  */
188
189 void
190 ngx_cpu_pause(void);
191
192 /* the code in src/os/unix/nginx_sunpro_amd64.il */
193
194 #define ngx_memory_barrier()    __asm (".volatile"); __asm (".nonvolatile")
195
196
197 #else /* ( __GNUC__ || __INTEL_COMPILER ) */
198
199 #define NGX_HAVE_ATOMIC_OPS 1
200
201 #include "ngx_gcc_atomic_amd64.h"
202
203 #endif
204
205
206 #elif ( __sparc__ || __sparc || __sparcv9 )
207
208 #if (NGX_PTR_SIZE == 8)
209
210 typedef int64_t                ngx_atomic_int_t;
211 typedef uint64_t               ngx_atomic_uint_t;
212 #define NGX_ATOMIC_T_LEN      (sizeof("-9223372036854775808") - 1)
213
214 #else
215
216 typedef int32_t                ngx_atomic_int_t;
217 typedef uint32_t               ngx_atomic_uint_t;
218 #define NGX_ATOMIC_T_LEN      (sizeof("-2147483648") - 1)
219
220 #endif
221
222 typedef volatile ngx_atomic_uint_t ngx_atomic_t;
223
224
225 #if ( __SUNPRO_C )
226
227 #define NGX_HAVE_ATOMIC_OPS 1
228
229 #include "ngx_sunpro_atomic_sparc64.h"
230
231
232 #else /* ( __GNUC__ || __INTEL_COMPILER ) */
233
234 #define NGX_HAVE_ATOMIC_OPS 1
235
236 #include "ngx_gcc_atomic_sparc64.h"
237
238 #endif
239
240
241 #elif ( __powerpc__ || __POWERPC__ )
242
243 #define NGX_HAVE_ATOMIC_OPS 1
244
245 #if (NGX_PTR_SIZE == 8)
246
247 typedef int64_t                ngx_atomic_int_t;
248 typedef uint64_t               ngx_atomic_uint_t;
249 #define NGX_ATOMIC_T_LEN      (sizeof("-9223372036854775808") - 1)
250
251 #else
252
253 typedef int32_t                ngx_atomic_int_t;
254 typedef uint32_t               ngx_atomic_uint_t;
255 #define NGX_ATOMIC_T_LEN      (sizeof("-2147483648") - 1)
256
257 #endif

```

```

258
259 typedef volatile ngx\_atomic\_uint\_t ngx_atomic_t;
260
261
262 #include "ngx_gcc_atomic_ppc.h"
263
264 #endif
265
266
267 #if !(NGX_HAVE_ATOMIC_OPS)
268
269 #define NGX_HAVE_ATOMIC_OPS 0
270
271 typedef int32_t ngx_atomic_int_t;
272 typedef uint32_t ngx_atomic_uint_t;
273 typedef volatile ngx\_atomic\_uint\_t ngx_atomic_t;
274 #define NGX_ATOMIC_T_LEN (sizeof("-2147483648") - 1)
275
276
277 static ngx\_inline ngx\_atomic\_uint\_t
278 ngx_atomic_cmp_set(ngx\_atomic\_t *lock, ngx\_atomic\_uint\_t old,
279 ngx\_atomic\_uint\_t set)
280 {
281     if (*lock == old) {
282         *lock = set;
283         return 1;
284     }
285
286     return 0;
287 }
288
289
290 static ngx\_inline ngx\_atomic\_int\_t
291 ngx_atomic_fetch_add(ngx\_atomic\_t *value, ngx\_atomic\_int\_t add)
292 {
293     ngx\_atomic\_int\_t old;
294
295     old = *value;
296     *value += add;
297
298     return old;
299 }
300
301 #define ngx_memory_barrier()
302 #define ngx_cpu_pause()
303
304 #endif
305
306
307 void ngx_spinlock(ngx\_atomic\_t *lock, ngx\_atomic\_int\_t value, ngx\_uint\_t spin);
308
309 #define ngx_trylock(lock) ((*lock) == 0 && ngx_atomic_cmp_set(lock, 0, 1))
310 #define ngx_unlock(lock) (*lock) = 0
311
312
313 #endif /* NGX\_ATOMIC\_H\_INCLUDED */

```

[One Level Up](#)

[Top Level](#)

## src/http/modules/perl/nginx\_http\_perl\_module.c - nginx-1.7.10

### Global variables defined

- [nginx\\_stash](#)
- [ngx\\_http\\_perl\\_commands](#)
- [ngx\\_http\\_perl\\_module](#)
- [ngx\\_http\\_perl\\_module\\_ctx](#)
- [ngx\\_http\\_perl\\_ssi\\_command](#)
- [ngx\\_http\\_perl\\_ssi\\_params](#)
- [ngx\\_null\\_name](#)
- [ngx\\_perl\\_term](#)
- [perl](#)

### Data types defined

- [ngx\\_http\\_perl\\_loc\\_conf\\_t](#)
- [ngx\\_http\\_perl\\_main\\_conf\\_t](#)
- [ngx\\_http\\_perl\\_variable\\_t](#)

### Functions defined

- [ngx\\_http\\_perl](#)
- [ngx\\_http\\_perl\\_call\\_handler](#)
- [ngx\\_http\\_perl\\_cleanup\\_perl](#)
- [ngx\\_http\\_perl\\_create\\_interpreter](#)
- [ngx\\_http\\_perl\\_create\\_loc\\_conf](#)
- [ngx\\_http\\_perl\\_create\\_main\\_conf](#)
- [ngx\\_http\\_perl\\_eval\\_anon\\_sub](#)
- [ngx\\_http\\_perl\\_exit](#)
- [ngx\\_http\\_perl\\_handle\\_request](#)
- [ngx\\_http\\_perl\\_handler](#)
- [ngx\\_http\\_perl\\_init\\_interpreter](#)
- [ngx\\_http\\_perl\\_init\\_main\\_conf](#)
- [ngx\\_http\\_perl\\_init\\_worker](#)
- [ngx\\_http\\_perl\\_merge\\_loc\\_conf](#)
- [ngx\\_http\\_perl\\_preconfiguration](#)

- [ngx\\_http\\_perl\\_run\\_requires](#)
- [ngx\\_http\\_perl\\_set](#)
- [ngx\\_http\\_perl\\_sleep\\_handler](#)
- [ngx\\_http\\_perl\\_ssi](#)
- [ngx\\_http\\_perl\\_variable](#)
- [ngx\\_http\\_perl\\_xs\\_init](#)

## Macros defined

- [NGX\\_HTTP\\_PERL\\_SSI\\_ARG](#)
- [NGX\\_HTTP\\_PERL\\_SSI\\_SUB](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11 #include <ngx_http_perl_module.h>
12
13
14 typedef struct {
15     PerlInterpreter    *perl;
16     HV                 *nginx;
17     ngx_array_t        *modules;
18     ngx_array_t        *requires;
19 } ngx_http_perl_main_conf_t;
20
21
22 typedef struct {
23     SV                 *sub;
24     ngx_str_t          handler;
25 } ngx_http_perl_loc_conf_t;
26
27
28 typedef struct {
29     SV                 *sub;
30     ngx_str_t          handler;
31 } ngx_http_perl_variable_t;
32
33
34 #if (NGX_HTTP_SSI)
35 static ngx_int_t ngx_http_perl_ssi(ngx_http_request_t *r,
36     ngx_http_ssi_ctx_t *ssi_ctx, ngx_str_t **params);
37 #endif
38
39 static char *ngx_http_perl_init_interpreter(ngx_conf_t *cf,
40     ngx_http_perl_main_conf_t *pconf);
41 static PerlInterpreter *ngx_http_perl_create_interpreter(ngx_conf_t *cf,
42     ngx_http_perl_main_conf_t *pconf);
43 static ngx_int_t ngx_http_perl_run_requires(pTHX_ ngx_array_t *requires,
44     ngx_log_t *log);
45 static ngx_int_t ngx_http_perl_call_handler(pTHX_ ngx_http_request_t *r,
46     HV *nginx, SV *sub, SV **args, ngx_str_t *handler, ngx_str_t *rv);
47 static void ngx_http_perl_eval_anon_sub(pTHX_ ngx_str_t *handler, SV **sv);
48
49 static ngx_int_t ngx_http_perl_preconfiguration(ngx_conf_t *cf);
50 static void *ngx_http_perl_create_main_conf(ngx_conf_t *cf);
51 static char *ngx_http_perl_init_main_conf(ngx_conf_t *cf, void *conf);

```

```

52 static void *ngx_http_perl_create_loc_conf(ngx_conf_t *cf);
53 static char *ngx_http_perl_merge_loc_conf(ngx_conf_t *cf, void *parent,
54 void *child);
55 static char *ngx_http_perl(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
56 static char *ngx_http_perl_set(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
57
58 #if (NGX_HAVE_PERL_MULTIPLICITY)
59 static void ngx_http_perl_cleanup_perl(void *data);
60 #endif
61
62 static ngx_int_t ngx_http_perl_init_worker(ngx_cycle_t *cycle);
63 static void ngx_http_perl_exit(ngx_cycle_t *cycle);
64
65
66 static ngx_command_t ngx_http_perl_commands[] = {
67
68     { ngx_string("perl_modules"),
69       NGX_HTTP_MAIN_CONF|NGX_CONF_TAKE1,
70       ngx_conf_set_str_array_slot,
71       NGX_HTTP_MAIN_CONF_OFFSET,
72       offsetof(ngx_http_perl_main_conf_t, modules),
73       NULL },
74
75     { ngx_string("perl_require"),
76       NGX_HTTP_MAIN_CONF|NGX_CONF_TAKE1,
77       ngx_conf_set_str_array_slot,
78       NGX_HTTP_MAIN_CONF_OFFSET,
79       offsetof(ngx_http_perl_main_conf_t, requires),
80       NULL },
81
82     { ngx_string("perl"),
83       NGX_HTTP_LOC_CONF|NGX_HTTP_LMT_CONF|NGX_CONF_TAKE1,
84       ngx_http_perl,
85       NGX_HTTP_LOC_CONF_OFFSET,
86       0,
87       NULL },
88
89     { ngx_string("perl_set"),
90       NGX_HTTP_MAIN_CONF|NGX_CONF_TAKE2,
91       ngx_http_perl_set,
92       NGX_HTTP_LOC_CONF_OFFSET,
93       0,
94       NULL },
95
96     ngx_null_command
97 };
98
99
100 static ngx_http_module_t ngx_http_perl_module_ctx = {
101     ngx_http_perl_preconfiguration,      /* preconfiguration */
102     NULL,                                /* postconfiguration */
103
104     ngx_http_perl_create_main_conf,      /* create main configuration */
105     ngx_http_perl_init_main_conf,        /* init main configuration */
106
107     NULL,                                 /* create server configuration */
108     NULL,                                 /* merge server configuration */
109
110     ngx_http_perl_create_loc_conf,       /* create location configuration */
111     ngx_http_perl_merge_loc_conf         /* merge location configuration */
112 };
113
114
115 ngx_module_t ngx_http_perl_module = {
116     NGX_MODULE_V1,
117     &ngx_http_perl_module_ctx,          /* module context */
118     ngx_http_perl_commands,             /* module directives */
119     NGX_HTTP_MODULE,                    /* module type */
120     NULL,                                /* init master */
121     NULL,                                /* init module */
122     ngx_http_perl_init_worker,           /* init process */
123     NULL,                                /* init thread */
124     NULL,                                /* exit thread */
125     NULL,                                /* exit process */
126     ngx_http_perl_exit,                  /* exit master */
127     NGX_MODULE_V1_PADDING

```

```

128 };
129
130
131 #if (NGX_HTTP_SSI)
132
133 #define NGX_HTTP_PERL_SSI_SUB 0
134 #define NGX_HTTP_PERL_SSI_ARG 1
135
136
137 static ngx_http_ssi_param_t ngx_http_perl_ssi_params[] = {
138     { ngx_string("sub"), NGX_HTTP_PERL_SSI_SUB, 1, 0 },
139     { ngx_string("arg"), NGX_HTTP_PERL_SSI_ARG, 0, 1 },
140     { ngx_null_string, 0, 0, 0 }
141 };
142
143 static ngx_http_ssi_command_t ngx_http_perl_ssi_command = {
144     ngx_string("perl"), ngx_http_perl_ssi, ngx_http_perl_ssi_params, 0, 0, 1
145 };
146
147 #endif
148
149
150 static ngx_str_t          ngx_null_name = ngx_null_string;
151 static HV                *ngxinx_stash;
152
153 #if (NGX_HAVE_PERL_MULTIPLICITY)
154 static ngx_uint_t        ngx_perl_term;
155 #else
156 static PerlInterpreter   *perl;
157 #endif
158
159
160 static void
161 ngx_http_perl_xs_init(pTHX)
162 {
163     newXS("DynaLoader::boot_DynaLoader", boot_DynaLoader, __FILE__);
164
165     ngxinx_stash = gv_stashpv("ngxinx", TRUE);
166 }
167
168
169 static ngx_int_t
170 ngx_http_perl_handler(ngx_http_request_t *r)
171 {
172     r->main->count++;
173
174     ngx_http_perl_handle_request(r);
175
176     return NGX_DONE;
177 }
178
179
180 void
181 ngx_http_perl_handle_request(ngx_http_request_t *r)
182 {
183     SV                *sub;
184     ngx_int_t         rc;
185     ngx_str_t         uri, args, *handler;
186     ngx_http_perl_ctx_t *ctx;
187     ngx_http_perl_loc_conf_t *plcf;
188     ngx_http_perl_main_conf_t *pmcf;
189
190     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0, "perl handler");
191
192     ctx = ngx_http_get_module_ctx(r, ngx_http_perl_module);
193
194     if (ctx == NULL) {
195         ctx = ngx_palloc(r->pool, sizeof(ngx_http_perl_ctx_t));
196         if (ctx == NULL) {
197             ngx_http_finalize_request(r, NGX_ERROR);
198             return;
199         }
200
201         ngx_http_set_ctx(r, ctx, ngx_http_perl_module);
202     }
203

```

```

204 pmcf = ngx_http_get_module_main_conf(r, ngx_http_perl_module);
205
206 {
207
208     dTHXa(pmcf->perl);
209     PERL_SET_CONTEXT(pmcf->perl);
210
211     if (ctx->next == NULL) {
212         plcf = ngx_http_get_module_loc_conf(r, ngx_http_perl_module);
213         sub = plcf->sub;
214         handler = &plcf->handler;
215
216     } else {
217         sub = ctx->next;
218         handler = &ngx_null_name;
219         ctx->next = NULL;
220     }
221
222     rc = ngx_http_perl_call_handler(aTHX_ r, pmcf->ngxinx, sub, NULL, handler,
223                                   NULL);
224
225 }
226
227 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
228               "perl handler done: %i", rc);
229
230 if (rc == NGX_DONE) {
231     ngx_http_finalize_request(r, rc);
232     return;
233 }
234
235 if (rc > 600) {
236     rc = NGX_OK;
237 }
238
239 if (ctx->redirect_uri.len) {
240     uri = ctx->redirect_uri;
241     args = ctx->redirect_args;
242
243 } else {
244     uri.len = 0;
245 }
246
247 ctx->filename.data = NULL;
248 ctx->redirect_uri.len = 0;
249
250 if (ctx->done || ctx->next) {
251     ngx_http_finalize_request(r, NGX_DONE);
252     return;
253 }
254
255 if (uri.len) {
256     ngx_http_internal_redirect(r, &uri, &args);
257     ngx_http_finalize_request(r, NGX_DONE);
258     return;
259 }
260
261 if (rc == NGX_OK || rc == NGX_HTTP_OK) {
262     ngx_http_send_special(r, NGX_HTTP_LAST);
263     ctx->done = 1;
264 }
265
266 ngx_http_finalize_request(r, rc);
267 }
268
269
270 void
271 ngx_http_perl_sleep_handler(ngx_http_request_t *r)
272 {
273     ngx_event_t *wev;
274
275     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
276                   "perl sleep handler");
277
278     wev = r->connection->write;
279

```



```

280     if (wev->timedout) {
281         wev->timedout = 0;
282         ngx_http_perl_handle_request(r);
283         return;
284     }
285
286     if (ngx_handle_write_event(wev, 0) != NGX_OK) {
287         ngx_http_finalize_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
288     }
289 }
290
291
292 static ngx_int_t
293 ngx_http_perl_variable(ngx_http_request_t *r, ngx_http_variable_value_t *v,
294     uintptr_t data)
295 {
296     ngx_http_perl_variable_t *pv = (ngx_http_perl_variable_t *) data;
297
298     ngx_int_t          rc;
299     ngx_str_t          value;
300     ngx_http_perl_ctx_t *ctx;
301     ngx_http_perl_main_conf_t *pmcf;
302
303     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
304         "perl variable handler");
305
306     ctx = ngx_http_get_module_ctx(r, ngx_http_perl_module);
307
308     if (ctx == NULL) {
309         ctx = ngx_palloc(r->pool, sizeof(ngx_http_perl_ctx_t));
310         if (ctx == NULL) {
311             return NGX_ERROR;
312         }
313
314         ngx_http_set_ctx(r, ctx, ngx_http_perl_module);
315     }
316
317     pmcf = ngx_http_get_module_main_conf(r, ngx_http_perl_module);
318
319     value.data = NULL;
320
321     {
322
323         dTHXa(pmcf->perl);
324         PERL_SET_CONTEXT(pmcf->perl);
325
326         rc = ngx_http_perl_call_handler(aTHX_ r, pmcf->nginx, pv->sub, NULL,
327             &pv->handler, &value);
328
329     }
330
331     if (value.data) {
332         v->len = value.len;
333         v->valid = 1;
334         v->no_cacheable = 0;
335         v->not_found = 0;
336         v->data = value.data;
337
338     } else {
339         v->not_found = 1;
340     }
341
342     ctx->filename.data = NULL;
343     ctx->redirect_uri.len = 0;
344
345     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
346         "perl variable done");
347
348     return rc;
349 }
350
351
352 #if (NGX_HTTP_SSI)
353
354 static ngx_int_t
355 ngx_http_perl_ssi(ngx_http_request_t *r, ngx_http_ssi_ctx_t *ssi_ctx,

```

```

356     ngx_str_t **params)
357 {
358     SV                *sv, **asv;
359     ngx_int_t         rc;
360     ngx_str_t         *handler, **args;
361     ngx_uint_t        i;
362     ngx_http_perl_ctx_t *ctx;
363     ngx_http_perl_main_conf_t *pmcf;
364
365     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
366                  "perl ssi handler");
367
368     ctx = ngx_http_get_module_ctx(r, ngx_http_perl_module);
369
370     if (ctx == NULL) {
371         ctx = ngx_palloc(r->pool, sizeof(ngx_http_perl_ctx_t));
372         if (ctx == NULL) {
373             return NGX_ERROR;
374         }
375
376         ngx_http_set_ctx(r, ctx, ngx_http_perl_module);
377     }
378
379     pmcf = ngx_http_get_module_main_conf(r, ngx_http_perl_module);
380
381     ctx->ssi = ssi_ctx;
382
383     handler = params[NGX_HTTP_PERL_SSI_SUB];
384     handler->data[handler->len] = '\0';
385
386     {
387
388         dTHXa(pmcf->perl);
389         PERL_SET_CONTEXT(pmcf->perl);
390
391 #if 0
392
393     /* the code is disabled to force the precompiled perl code using only */
394
395     ngx_http_perl_eval_anon_sub(aTHX_ handler, &sv);
396
397     if (sv == &PL_sv_undef) {
398         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
399                     "eval_pv(\"%V\") failed", handler);
400         return NGX_ERROR;
401     }
402
403     if (sv == NULL) {
404         sv = newSVpvn((char *) handler->data, handler->len);
405     }
406 #endif
407 #endif
408
409     sv = newSVpvn((char *) handler->data, handler->len);
410
411     args = &params[NGX_HTTP_PERL_SSI_ARG];
412
413     if (args) {
414
415         for (i = 0; args[i]; i++) { /* void */ }
416
417         asv = ngx_palloc(r->pool, (i + 1) * sizeof(SV *));
418
419         if (asv == NULL) {
420             SvREFCNT_dec(sv);
421             return NGX_ERROR;
422         }
423
424         asv[0] = (SV *) (uintptr_t) i;
425
426         for (i = 0; args[i]; i++) {
427             asv[i + 1] = newSVpvn((char *) args[i]->data, args[i]->len);
428         }
429
430     } else {
431         asv = NULL;

```

```

432     }
433
434     rc = ngx_http_perl_call_handler(aTHX_ r, pmcf->ngxinx, sv, asv, handler,
435                                     NULL);
436
437     SvREFCNT_dec(sv);
438
439     }
440
441     ctx->filename.data = NULL;
442     ctx->redirect_uri.len = 0;
443     ctx->ssi = NULL;
444
445     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0, "perl ssi done");
446
447     return rc;
448 }
449
450 #endif
451
452
453 static char *
454 ngx_http_perl_init_interpreter(ngx_conf_t *cf, ngx_http_perl_main_conf_t *pmcf)
455 {
456     ngx_str_t      *m;
457     ngx_uint_t      i;
458     #if (NGX_HAVE_PERL_MULTIPLICITY)
459     ngx_pool_cleanup_t *cfn;
460
461     cfn = ngx_pool_cleanup_add(cf->pool, 0);
462     if (cfn == NULL) {
463         return NGX_CONF_ERROR;
464     }
465
466 #endif
467
468 #ifdef NGX_PERL_MODULES
469     if (pmcf->modules == NGX_CONF_UNSET_PTR) {
470
471         pmcf->modules = ngx_array_create(cf->pool, 1, sizeof(ngx_str_t));
472         if (pmcf->modules == NULL) {
473             return NGX_CONF_ERROR;
474         }
475
476         m = ngx_array_push(pmcf->modules);
477         if (m == NULL) {
478             return NGX_CONF_ERROR;
479         }
480
481         ngx_str_set(m, NGX_PERL_MODULES);
482     }
483 #endif
484
485     if (pmcf->modules != NGX_CONF_UNSET_PTR) {
486         m = pmcf->modules->elts;
487         for (i = 0; i < pmcf->modules->nelts; i++) {
488             if (ngx_conf_full_name(cf->cycle, &m[i], 0) != NGX_OK) {
489                 return NGX_CONF_ERROR;
490             }
491         }
492     }
493
494     #if !(NGX_HAVE_PERL_MULTIPLICITY)
495     if (perl) {
496         if (ngx_set_environment(cf->cycle, NULL) == NULL) {
497             return NGX_CONF_ERROR;
498         }
499
500     }
501
502     if (ngx_http_perl_run_requires(aTHX_ pmcf->requires, cf->log)
503         != NGX_OK)
504     {
505         return NGX_CONF_ERROR;
506     }
507

```

```

508     pmcf->perl = perl;
509     pmcf->nginx = nginx_stash;
510
511     return NGX_CONF_OK;
512 }
513
514 #endif
515
516 if (nginx_stash == NULL) {
517     PERL_SYS_INIT(&ngx_argc, &ngx_argv);
518 }
519
520 pmcf->perl = ngx_http_perl_create_interpreter(cf, pmcf);
521
522 if (pmcf->perl == NULL) {
523     return NGX_CONF_ERROR;
524 }
525
526 pmcf->nginx = nginx_stash;
527
528 #if (NGX_HAVE_PERL_MULTIPLICITY)
529
530     cln->handler = ngx_http_perl_cleanup_perl;
531     cln->data = pmcf->perl;
532
533 #else
534
535     perl = pmcf->perl;
536
537 #endif
538
539     return NGX_CONF_OK;
540 }
541
542
543 static PerlInterpreter *
544 ngx_http_perl_create_interpreter(ngx_conf_t *cf,
545     ngx_http_perl_main_conf_t *pmcf)
546 {
547     int             n;
548     STRLEN         len;
549     SV             *sv;
550     char           *ver, **embedding;
551     ngx_str_t      *m;
552     ngx_uint_t     i;
553     PerlInterpreter *perl;
554
555     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, cf->log, 0, "create perl interpreter");
556
557     if (ngx_set_environment(cf->cycle, NULL) == NULL) {
558         return NULL;
559     }
560
561     perl = perl_alloc();
562     if (perl == NULL) {
563         ngx_log_error(NGX_LOG_ALERT, cf->log, 0, "perl_alloc() failed");
564         return NULL;
565     }
566
567     {
568
569         dTHXa(perl);
570         PERL_SET_CONTEXT(perl);
571
572         perl_construct(perl);
573
574 #ifndef PERL_EXIT_DESTRUCT_END
575         PL_exit_flags |= PERL_EXIT_DESTRUCT_END;
576 #endif
577
578         n = (pmcf->modules != NGX_CONF_UNSET_PTR) ? pmcf->modules->nelts * 2 : 0;
579
580         embedding = ngx_palloc(cf->pool, (5 + n) * sizeof(char *));
581         if (embedding == NULL) {
582             goto fail;
583         }

```

```

584 embedding[0] = "";
585
586
587 if (n++) {
588     m = pmcf->modules->elts;
589     for (i = 0; i < pmcf->modules->nelts; i++) {
590         embedding[2 * i + 1] = "-I";
591         embedding[2 * i + 2] = (char *) m[i].data;
592     }
593 }
594
595 embedding[n++] = "-Mnginx";
596 embedding[n++] = "-e";
597 embedding[n++] = "0";
598 embedding[n] = NULL;
599
600 n = perl_parse(perl, ngx_http_perl_xs_init, n, embedding, NULL);
601
602 if (n != 0) {
603     ngx_log_error(NGX_LOG_ALERT, cf->log, 0, "perl_parse() failed: %d", n);
604     goto fail;
605 }
606
607 sv = get_sv("nginx::VERSION", FALSE);
608 ver = SvPV(sv, len);
609
610 if (ngx_strcmp(ver, NGX_VERSION) != 0) {
611     ngx_log_error(NGX_LOG_ALERT, cf->log, 0,
612                 "version " NGX_VERSION " of nginx.pm is required, "
613                 "but %s was found", ver);
614     goto fail;
615 }
616
617 if (ngx_http_perl_run_requires(aTHX_ pmcf->requires, cf->log) != NGX_OK) {
618     goto fail;
619 }
620
621 }
622
623 return perl;
624
625 fail:
626
627 (void) perl_destruct(perl);
628
629 perl_free(perl);
630
631 return NULL;
632 }
633
634
635 static ngx_int_t
636 ngx_http_perl_run_requires(pTHX_ ngx_array_t *requires, ngx_log_t *log)
637 {
638     u_char      *err;
639     STRLEN      len;
640     ngx_str_t   *script;
641     ngx_uint_t  i;
642
643     if (requires == NGX_CONF_UNSET_PTR) {
644         return NGX_OK;
645     }
646
647     script = requires->elts;
648     for (i = 0; i < requires->nelts; i++) {
649         require_pv((char *) script[i].data);
650
651         if (SvTRUE(ERRSV)) {
652             err = (u_char *) SvPV(ERRSV, len);
653             while (--len && (err[len] == CR || err[len] == LF)) { /* void */ }
654
655             ngx_log_error(NGX_LOG_EMERG, log, 0,
656                         "require_pv(\"%s\") failed: \"%s\"",
657                         script[i].data, len + 1, err);
658         }
659     }

```

```

660         return NGX_ERROR;
661     }
662 }
663
664
665 return NGX_OK;
666 }
667
668
669 static ngx_int_t
670 ngx_http_perl_call_handler(pTHX_ ngx_http_request_t *r, HV *ngxin, SV *sub,
671 SV **args, ngx_str_t *handler, ngx_str_t *rv)
672 {
673     SV          *sv;
674     int          n, status;
675     char        *line;
676     u_char      *err;
677     STRLEN      len, n_a;
678     ngx_uint_t  i;
679     ngx_connection_t *c;
680
681     dSP;
682
683     status = 0;
684
685     ENTER;
686     SAVETMPS;
687
688     PUSHMARK(sp);
689
690     sv = sv_2mortal(sv_bless(newRV_noinc(newSViv(PTR2IV(r))), ngxin));
691     XPUSHs(sv);
692
693     if (args) {
694         EXTEND(sp, (intptr_t) args[0]);
695
696         for (i = 1; i <= (uintptr_t) args[0]; i++) {
697             PUSHs(sv_2mortal(args[i]));
698         }
699     }
700
701     PUTBACK;
702
703     c = r->connection;
704
705     n = call_sv(sub, G_EVAL);
706
707     SPAGAIN;
708
709     if (n) {
710         if (rv == NULL) {
711             status = POPi;
712
713             ngx_log_debug1(NGX_LOG_DEBUG_HTTP, c->log, 0,
714                 "call_sv: %d", status);
715
716         } else {
717             line = SvPVx(POPs, n_a);
718             rv->len = n_a;
719
720             rv->data = ngx_pnalloc(r->pool, n_a);
721             if (rv->data == NULL) {
722                 return NGX_ERROR;
723             }
724
725             ngx_memcpy(rv->data, line, n_a);
726         }
727     }
728
729     PUTBACK;
730
731     FREETMPS;
732     LEAVE;
733
734     /* check $@ */
735

```

```

736     if (SVTRUE(ERRSV)) {
737
738         err = (u_char *) SvPV(ERRSV, len);
739         while (--len && (err[len] == CR || err[len] == LF)) { /* void */
740
741             ngx_log_error(NGX_LOG_ERR, c->log, 0,
742                 "call_sv(\"%V\") failed: \"%s\"", handler, len + 1, err);
743
744             if (rv) {
745                 return NGX_ERROR;
746             }
747
748             return NGX_HTTP_INTERNAL_SERVER_ERROR;
749         }
750
751         if (n != 1) {
752             ngx_log_error(NGX_LOG_ALERT, c->log, 0,
753                 "call_sv(\"%V\") returned %d results", handler, n);
754             status = NGX_OK;
755         }
756
757         if (rv) {
758             return NGX_OK;
759         }
760
761         return (ngx_int_t) status;
762     }
763
764
765     static void
766     ngx_http_perl_eval_anon_sub(pTHX_ ngx_str_t *handler, SV **sv)
767     {
768         u_char *p;
769
770         for (p = handler->data; *p; p++) {
771             if (*p != ' ' && *p != '\t' && *p != CR && *p != LF) {
772                 break;
773             }
774         }
775
776         if (ngx_strncmp(p, "sub ", 4) == 0
777             || ngx_strncmp(p, "sub{", 4) == 0
778             || ngx_strncmp(p, "use ", 4) == 0)
779         {
780             *sv = eval_pv((char *) p, FALSE);
781
782             /* eval_pv() does not set ERRSV on failure */
783
784             return;
785         }
786
787         *sv = NULL;
788     }
789
790
791     static void *
792     ngx_http_perl_create_main_conf(ngx_conf_t *cf)
793     {
794         ngx_http_perl_main_conf_t *pmcf;
795
796         pmcf = ngx_palloc(cf->pool, sizeof(ngx_http_perl_main_conf_t));
797         if (pmcf == NULL) {
798             return NULL;
799         }
800
801         pmcf->modules = NGX_CONF_UNSET_PTR;
802         pmcf->requires = NGX_CONF_UNSET_PTR;
803
804         return pmcf;
805     }
806
807
808     static char *
809     ngx_http_perl_init_main_conf(ngx_conf_t *cf, void *conf)
810     {
811         ngx_http_perl_main_conf_t *pmcf = conf;

```

```

812     if (pmcf->perl == NULL) {
813         if (ngx_http_perl_init_interpreter(cf, pmcf) != NGX_CONF_OK) {
814             return NGX_CONF_ERROR;
815         }
816     }
817 }
818
819 return NGX_CONF_OK;
820 }
821
822
823 #if (NGX_HAVE_PERL_MULTIPLICITY)
824
825 static void
826 ngx_http_perl_cleanup_perl(void *data)
827 {
828     PerlInterpreter *perl = data;
829
830     PERL_SET_CONTEXT(perl);
831
832     (void) perl_destruct(perl);
833
834     perl_free(perl);
835
836     if (ngx_perl_term) {
837         ngx_log_debug0(NGX_LOG_DEBUG_HTTP, ngx_cycle->log, 0, "perl term");
838
839         PERL_SYS_TERM();
840     }
841 }
842
843 #endif
844
845
846 static ngx_int_t
847 ngx_http_perl_preconfiguration(ngx_conf_t *cf)
848 {
849     #if (NGX_HTTP_SSI)
850         ngx_int_t rc;
851         ngx_http_ssi_main_conf_t *smcf;
852
853         smcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_ssi_filter_module);
854
855         rc = ngx_hash_add_key(&smcf->commands, &ngx_http_perl_ssi_command.name,
856                             &ngx_http_perl_ssi_command, NGX_HASH_READONLY_KEY);
857
858         if (rc != NGX_OK) {
859             if (rc == NGX_BUSY) {
860                 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
861                                     "conflicting SSI command \"%V\"",
862                                     &ngx_http_perl_ssi_command.name);
863             }
864
865             return NGX_ERROR;
866         }
867     #endif
868
869     return NGX_OK;
870 }
871
872
873 static void *
874 ngx_http_perl_create_loc_conf(ngx_conf_t *cf)
875 {
876     ngx_http_perl_loc_conf_t *plcf;
877
878     plcf = ngx_palloc(cf->pool, sizeof(ngx_http_perl_loc_conf_t));
879     if (plcf == NULL) {
880         return NULL;
881     }
882
883     /*
884     * set by ngx_palloc():
885     *
886     *     plcf->handler = { 0, NULL };
887     */

```



```

888     return plcf;
889 }
890
891
892
893 static char *
894 ngx_http_perl_merge_loc_conf(ngx_conf_t *cf, void *parent, void *child)
895 {
896     ngx_http_perl_loc_conf_t *prev = parent;
897     ngx_http_perl_loc_conf_t *conf = child;
898
899     if (conf->sub == NULL) {
900         conf->sub = prev->sub;
901         conf->handler = prev->handler;
902     }
903
904     return NGX_CONF_OK;
905 }
906
907
908 static char *
909 ngx_http_perl(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
910 {
911     ngx_http_perl_loc_conf_t *plcf = conf;
912
913     ngx_str_t          *value;
914     ngx_http_core_loc_conf_t *clcf;
915     ngx_http_perl_main_conf_t *pmcf;
916
917     value = cf->args->elts;
918
919     if (plcf->handler.data) {
920         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
921             "duplicate perl handler \"%V\"", &value[1]);
922         return NGX_CONF_ERROR;
923     }
924
925     pmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_perl_module);
926
927     if (pmcf->perl == NULL) {
928         if (ngx_http_perl_init_interpreter(cf, pmcf) != NGX_CONF_OK) {
929             return NGX_CONF_ERROR;
930         }
931     }
932
933     plcf->handler = value[1];
934
935     {
936
937         dTHXa(pmcf->perl);
938         PERL_SET_CONTEXT(pmcf->perl);
939
940         ngx_http_perl_eval_anon_sub(aTHX_ &value[1], &plcf->sub);
941
942         if (plcf->sub == &PL_sv_undef) {
943             ngx_conf_log_error(NGX_LOG_ERR, cf, 0,
944                 "eval_pv(\"%V\") failed", &value[1]);
945             return NGX_CONF_ERROR;
946         }
947
948         if (plcf->sub == NULL) {
949             plcf->sub = newSVpvn((char *) value[1].data, value[1].len);
950         }
951     }
952
953     clcf = ngx_http_conf_get_module_loc_conf(cf, ngx_http_core_module);
954     clcf->handler = ngx_http_perl_handler;
955
956     return NGX_CONF_OK;
957 }
958
959
960
961 static char *
962 ngx_http_perl_set(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
963 {

```

```

964     ngx_int_t             index;
965     ngx_str_t             *value;
966     ngx_http_variable_t  *v;
967     ngx_http_perl_variable_t *pv;
968     ngx_http_perl_main_conf_t *pmcf;
969
970     value = cf->args->elts;
971
972     if (value[1].data[0] != '$') {
973         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
974             "invalid variable name \"%V\"", &value[1]);
975         return NGX_CONF_ERROR;
976     }
977
978     value[1].len--;
979     value[1].data++;
980
981     v = ngx_http_add_variable(cf, &value[1], NGX_HTTP_VAR_CHANGEABLE);
982     if (v == NULL) {
983         return NGX_CONF_ERROR;
984     }
985
986     pv = ngx_palloc(cf->pool, sizeof(ngx_http_perl_variable_t));
987     if (pv == NULL) {
988         return NGX_CONF_ERROR;
989     }
990
991     index = ngx_http_get_variable_index(cf, &value[1]);
992     if (index == NGX_ERROR) {
993         return NGX_CONF_ERROR;
994     }
995
996     pmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_perl_module);
997
998     if (pmcf->perl == NULL) {
999         if (ngx_http_perl_init_interpreter(cf, pmcf) != NGX_CONF_OK) {
1000             return NGX_CONF_ERROR;
1001         }
1002     }
1003
1004     pv->handler = value[2];
1005
1006     {
1007
1008         dTHXa(pmcf->perl);
1009         PERL_SET_CONTEXT(pmcf->perl);
1010
1011         ngx_http_perl_eval_anon_sub(aTHX_ &value[2], &pv->sub);
1012
1013         if (pv->sub == &PL_sv_undef) {
1014             ngx_conf_log_error(NGX_LOG_ERR, cf, 0,
1015                 "eval_pv(\"%V\") failed", &value[2]);
1016             return NGX_CONF_ERROR;
1017         }
1018
1019         if (pv->sub == NULL) {
1020             pv->sub = newSVpvn((char *) value[2].data, value[2].len);
1021         }
1022     }
1023
1024     v->get_handler = ngx_http_perl_variable;
1025     v->data = (uintptr_t) pv;
1026
1027     return NGX_CONF_OK;
1028 }
1029
1030
1031
1032 static ngx_int_t
1033 ngx_http_perl_init_worker(ngx_cycle_t *cycle)
1034 {
1035     ngx_http_perl_main_conf_t *pmcf;
1036
1037     pmcf = ngx_http_cycle_get_module_main_conf(cycle, ngx_http_perl_module);
1038
1039     if (pmcf) {

```

```

1040     dTHXa(pmcF->perl);
1041     PERL_SET_CONTEXT(pmcF->perl);
1042
1043     /* set worker's $$ */
1044     sv_setiv(GvSV(gv_fetchpv("$", TRUE, SVT_PV)), (I32) ngx_pid);
1045 }
1046
1047 return NGX_OK;
1048 }
1049
1050
1051 static void
1052 ngx_http_perl_exit(ngx_cycle_t *cycle)
1053 {
1054     #if (NGX_HAVE_PERL_MULTIPLICITY)
1055         /*
1056          * the master exit hook is run before global pool cleanup,
1057          * therefore just set flag here
1058          */
1059         ngx_perl_term = 1;
1060     #else
1061         if (ngx_inx_stash) {
1062             ngx_log_debug0(NGX_LOG_DEBUG_HTTP, cycle->log, 0, "perl term");
1063
1064             (void) perl_destruct(perl);
1065
1066             perl_free(perl);
1067
1068             PERL_SYS_TERM();
1069         }
1070     #endif
1071 }
1072
1073 #endif
1074 }

```

[One Level Up](#)

[Top Level](#)

# src/core/nginx\_array.h - nginx-1.7.10

## Data types defined

- [ngx\\_array\\_t](#)

## Functions defined

- [ngx\\_array\\_init](#)

## Macros defined

- [\\_NGX\\_ARRAY\\_H\\_INCLUDED](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_ARRAY_H_INCLUDED_
9 #define _NGX_ARRAY_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 typedef struct {
17     void *elts;
18     ngx_uint_t nelts;
19     size_t size;
20     ngx_uint_t nalloc;
21     ngx_pool_t *pool;
22 } ngx_array_t;
23
24
25 ngx_array_t *ngx_array_create(ngx_pool_t *p, ngx_uint_t n, size_t size);
26 void ngx_array_destroy(ngx_array_t *a);
27 void *ngx_array_push(ngx_array_t *a);
28 void *ngx_array_push_n(ngx_array_t *a, ngx_uint_t n);
29
30
31 static ngx_inline ngx_int_t
32 ngx_array_init(ngx_array_t *array, ngx_pool_t *pool, ngx_uint_t n, size_t size)
33 {
34     /*
35      * set "array->nelts" before "array->elts", otherwise MSVC thinks
36      * that "array->nelts" may be used without having been initialized
37      */
38
39     array->nelts = 0;
40     array->size = size;
41     array->nalloc = n;
42     array->pool = pool;
43
44     array->elts = ngx_palloc(pool, n * size);
45     if (array->elts == NULL) {
46         return NGX_ERROR;
47     }
48
49     return NGX_OK;
50 }
51
```

52

53 #endif /\* [NGX\\_ARRAY\\_H\\_INCLUDED](#) \*/

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_array.c - nginx-1.7.10

### Functions defined

- [ngx\\_array\\_create](#)
- [ngx\\_array\\_destroy](#)
- [ngx\\_array\\_push](#)
- [ngx\\_array\\_push\\_n](#)

### Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10
11
12  ngx_array_t *
13  ngx_array_create(ngx_pool_t *p, ngx_uint_t n, size_t size)
14  {
15      ngx_array_t *a;
16
17      a = ngx_palloc(p, sizeof(ngx_array_t));
18      if (a == NULL) {
19          return NULL;
20      }
21
22      if (ngx_array_init(a, p, n, size) != NGX_OK) {
23          return NULL;
24      }
25
26      return a;
27  }
28
29
30  void
31  ngx_array_destroy(ngx_array_t *a)
32  {
33      ngx_pool_t *p;
34
35      p = a->pool;
36
37      if ((u_char *) a->elts + a->size * a->nalloc == p->d.last) {
38          p->d.last -= a->size * a->nalloc;
39      }
40
41      if ((u_char *) a + sizeof(ngx_array_t) == p->d.last) {
42          p->d.last = (u_char *) a;
43      }
44  }
45
46
47  void *
48  ngx_array_push(ngx_array_t *a)
49  {
50      void *elt, *new;
51      size_t size;
52      ngx_pool_t *p;
53
54      if (a->nelts == a->nalloc) {
55          /* the array is full */
56
```

```

57     size = a->size * a->nalloc;
58
59     p = a->pool;
60
61     if ((u_char *) a->elts + size == p->d.last
62         && p->d.last + a->size <= p->d.end)
63     {
64         /*
65          * the array allocation is the last in the pool
66          * and there is space for new allocation
67          */
68
69         p->d.last += a->size;
70         a->nalloc++;
71
72     } else {
73         /* allocate a new array */
74
75         new = ngx_palloc(p, 2 * size);
76         if (new == NULL) {
77             return NULL;
78         }
79
80         ngx_memcpy(new, a->elts, size);
81         a->elts = new;
82         a->nalloc *= 2;
83     }
84 }
85
86 elt = (u_char *) a->elts + a->size * a->nelts;
87 a->nelts++;
88
89 return elt;
90 }
91
92
93
94 void *
95 ngx_array_push_n(ngx_array_t *a, ngx_uint_t n)
96 {
97     void      *elt, *new;
98     size_t     size;
99     ngx_uint_t nalloc;
100    ngx_pool_t *p;
101
102    size = n * a->size;
103
104    if (a->nelts + n > a->nalloc) {
105
106        /* the array is full */
107
108        p = a->pool;
109
110        if ((u_char *) a->elts + a->size * a->nalloc == p->d.last
111            && p->d.last + size <= p->d.end)
112        {
113            /*
114             * the array allocation is the last in the pool
115             * and there is space for new allocation
116             */
117
118            p->d.last += size;
119            a->nalloc += n;
120
121        } else {
122            /* allocate a new array */
123
124            nalloc = 2 * ((n >= a->nalloc) ? n : a->nalloc);
125
126            new = ngx_palloc(p, nalloc * a->size);
127            if (new == NULL) {
128                return NULL;
129            }
130
131            ngx_memcpy(new, a->elts, a->nelts * a->size);
132            a->elts = new;

```

```
133         a->nalloc = nalloc;
134     }
135 }
136
137 elt = (u_char *) a->elts + a->size * a->nelts;
138 a->nelts += n;
139
140 return elt;
141 }
```

[One Level Up](#)

[Top Level](#)



# src/http/nginx\_http.h - nginx-1.7.10

## Data types defined

- [ngx\\_http\\_cache\\_t](#)
- [ngx\\_http\\_chunked\\_s](#)
- [ngx\\_http\\_chunked\\_t](#)
- [ngx\\_http\\_file\\_cache\\_t](#)
- [ngx\\_http\\_header\\_handler\\_pt](#)
- [ngx\\_http\\_log\\_ctx\\_s](#)
- [ngx\\_http\\_log\\_ctx\\_t](#)
- [ngx\\_http\\_log\\_handler\\_pt](#)
- [ngx\\_http\\_request\\_t](#)
- [ngx\\_http\\_spdy\\_stream\\_t](#)
- [ngx\\_http\\_status\\_t](#)
- [ngx\\_http\\_upstream\\_t](#)

## Macros defined

- [NGX\\_HTTP\\_FLUSH](#)
- [NGX\\_HTTP\\_LAST](#)
- [\\_NGX\\_HTTP\\_H\\_INCLUDED\\_](#)
- [ngx\\_http\\_ephemeral](#)
- [ngx\\_http\\_get\\_module\\_ctx](#)
- [ngx\\_http\\_set\\_ctx](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_HTTP_H_INCLUDED_
9 #define _NGX_HTTP_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 typedef struct ngx_http_request_s    ngx_http_request_t;
17 typedef struct ngx_http_upstream_s   ngx_http_upstream_t;
18 typedef struct ngx_http_cache_s      ngx_http_cache_t;
19 typedef struct ngx_http_file_cache_s ngx_http_file_cache_t;
20 typedef struct ngx_http_log_ctx_s    ngx_http_log_ctx_t;
```

```

21 typedef struct ngx_http_chunked_s      ngx_http_chunked_t;
22
23 #if (NGX_HTTP_SPDY)
24 typedef struct ngx_http_spdy_stream_s  ngx_http_spdy_stream_t;
25 #endif
26
27 typedef ngx_int_t (*ngx_http_header_handler_pt)(ngx_http_request_t *r,
28     ngx_table_elt_t *h, ngx_uint_t offset);
29 typedef u_char *(*ngx_http_log_handler_pt)(ngx_http_request_t *r,
30     ngx_http_request_t *sr, u_char *buf, size_t len);
31
32
33 #include <ngx_http_variables.h>
34 #include <ngx_http_config.h>
35 #include <ngx_http_request.h>
36 #include <ngx_http_script.h>
37 #include <ngx_http_upstream.h>
38 #include <ngx_http_upstream_round_robin.h>
39 #include <ngx_http_busy_lock.h>
40 #include <ngx_http_core_module.h>
41
42 #if (NGX_HTTP_SPDY)
43 #include <ngx_http_spdy.h>
44 #endif
45 #if (NGX_HTTP_CACHE)
46 #include <ngx_http_cache.h>
47 #endif
48 #if (NGX_HTTP_SSI)
49 #include <ngx_http_ssi_filter_module.h>
50 #endif
51 #if (NGX_HTTP_SSL)
52 #include <ngx_http_ssl_module.h>
53 #endif
54
55
56 struct ngx_http_log_ctx_s {
57     ngx_connection_t    *connection;
58     ngx_http_request_t *request;
59     ngx_http_request_t *current_request;
60 };
61
62
63 struct ngx_http_chunked_s {
64     ngx_uint_t          state;
65     off_t                size;
66     off_t                length;
67 };
68
69
70 typedef struct {
71     ngx_uint_t          http_version;
72     ngx_uint_t          code;
73     ngx_uint_t          count;
74     u_char               *start;
75     u_char               *end;
76 } ngx_http_status_t;
77
78
79 #define ngx_http_get_module_ctx(r, module) (r)->ctx[module.ctx_index]
80 #define ngx_http_set_ctx(r, c, module)    r->ctx[module.ctx_index] = c;
81
82
83 ngx_int_t ngx_http_add_location(ngx_conf_t *cf, ngx_queue_t **locations,
84     ngx_http_core_loc_conf_t *clcf);
85 ngx_int_t ngx_http_add_listen(ngx_conf_t *cf, ngx_http_core_srv_conf_t *cscf,
86     ngx_http_listen_opt_t *lsopt);
87
88
89 void ngx_http_init_connection(ngx_connection_t *c);
90 void ngx_http_close_connection(ngx_connection_t *c);
91
92 #if (NGX_HTTP_SSL && defined SSL_CTRL_SET_TLSEXT_HOSTNAME)
93 int ngx_http_ssl_servername(ngx_ssl_conn_t *ssl_conn, int *ad, void *arg);
94 #endif
95
96 ngx_int_t ngx_http_parse_request_line(ngx_http_request_t *r, ngx_buf_t *b);

```

```

97 ngx_int_t ngx_http_parse_uri(ngx_http_request_t *r);
98 ngx_int_t ngx_http_parse_complex_uri(ngx_http_request_t *r,
99     ngx_uint_t merge_slashes);
100 ngx_int_t ngx_http_parse_status_line(ngx_http_request_t *r, ngx_buf_t *b,
101     ngx_http_status_t *status);
102 ngx_int_t ngx_http_parse_unsafe_uri(ngx_http_request_t *r, ngx_str_t *uri,
103     ngx_str_t *args, ngx_uint_t *flags);
104 ngx_int_t ngx_http_parse_header_line(ngx_http_request_t *r, ngx_buf_t *b,
105     ngx_uint_t allow_underscores);
106 ngx_int_t ngx_http_parse_multi_header_lines(ngx_array_t *headers,
107     ngx_str_t *name, ngx_str_t *value);
108 ngx_int_t ngx_http_parse_set_cookie_lines(ngx_array_t *headers,
109     ngx_str_t *name, ngx_str_t *value);
110 ngx_int_t ngx_http_arg(ngx_http_request_t *r, u_char *name, size_t len,
111     ngx_str_t *value);
112 void ngx_http_split_args(ngx_http_request_t *r, ngx_str_t *uri,
113     ngx_str_t *args);
114 ngx_int_t ngx_http_parse_chunked(ngx_http_request_t *r, ngx_buf_t *b,
115     ngx_http_chunked_t *ctx);
116
117
118 ngx_http_request_t *ngx_http_create_request(ngx_connection_t *c);
119 ngx_int_t ngx_http_process_request_uri(ngx_http_request_t *r);
120 ngx_int_t ngx_http_process_request_header(ngx_http_request_t *r);
121 void ngx_http_process_request(ngx_http_request_t *r);
122 void ngx_http_update_location_config(ngx_http_request_t *r);
123 void ngx_http_handler(ngx_http_request_t *r);
124 void ngx_http_run_posted_requests(ngx_connection_t *c);
125 ngx_int_t ngx_http_post_request(ngx_http_request_t *r,
126     ngx_http_posted_request_t *pr);
127 void ngx_http_finalize_request(ngx_http_request_t *r, ngx_int_t rc);
128 void ngx_http_free_request(ngx_http_request_t *r, ngx_int_t rc);
129
130 void ngx_http_empty_handler(ngx_event_t *wev);
131 void ngx_http_request_empty_handler(ngx_http_request_t *r);
132
133
134 #define ngx_http_ephemeral(r) (void *) (&r->uri_start)
135
136
137 #define NGX_HTTP_LAST 1
138 #define NGX_HTTP_FLUSH 2
139
140 ngx_int_t ngx_http_send_special(ngx_http_request_t *r, ngx_uint_t flags);
141
142
143 ngx_int_t ngx_http_read_client_request_body(ngx_http_request_t *r,
144     ngx_http_client_body_handler_pt post_handler);
145
146 ngx_int_t ngx_http_send_header(ngx_http_request_t *r);
147 ngx_int_t ngx_http_special_response_handler(ngx_http_request_t *r,
148     ngx_int_t error);
149 ngx_int_t ngx_http_filter_finalize_request(ngx_http_request_t *r,
150     ngx_module_t *m, ngx_int_t error);
151 void ngx_http_clean_header(ngx_http_request_t *r);
152
153
154 time_t ngx_http_parse_time(u_char *value, size_t len);
155 size_t ngx_http_get_time(char *buf, time_t t);
156
157
158
159 ngx_int_t ngx_http_discard_request_body(ngx_http_request_t *r);
160 void ngx_http_discarded_request_body_handler(ngx_http_request_t *r);
161 void ngx_http_block_reading(ngx_http_request_t *r);
162 void ngx_http_test_reading(ngx_http_request_t *r);
163
164
165 char *ngx_http_types_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
166 char *ngx_http_merge_types(ngx_conf_t *cf, ngx_array_t **keys,
167     ngx_hash_t *types_hash, ngx_array_t **prev_keys,
168     ngx_hash_t *prev_types_hash, ngx_str_t *default_types);
169 ngx_int_t ngx_http_set_default_types(ngx_conf_t *cf, ngx_array_t **types,
170     ngx_str_t *default_type);
171
172 #if (NGX_HTTP_DEGRADATION)

```

```
173 ngx\_uint\_t ngx\_http\_degraded(ngx\_http\_request\_t *);
174 #endif
175
176
177 extern ngx\_module\_t ngx\_http\_module;
178
179 extern ngx\_str\_t ngx\_http\_html\_default\_types[];
180
181
182 extern ngx\_http\_output\_header\_filter\_pt ngx\_http\_top\_header\_filter;
183 extern ngx\_http\_output\_body\_filter\_pt ngx\_http\_top\_body\_filter;
184
185
186 #endif /* \_NGX\_HTTP\_H\_INCLUDED */
```

[One Level Up](#)

[Top Level](#)

## src/http/nginx\_http\_request.h - nginx-1.7.10

### Data types defined

- [ngx\\_http\\_addr\\_conf\\_t](#)
- [ngx\\_http\\_cleanup\\_pt](#)
- [ngx\\_http\\_cleanup\\_s](#)
- [ngx\\_http\\_cleanup\\_t](#)
- [ngx\\_http\\_client\\_body\\_handler\\_pt](#)
- [ngx\\_http\\_connection\\_t](#)
- [ngx\\_http\\_ephemeral\\_t](#)
- [ngx\\_http\\_event\\_handler\\_pt](#)
- [ngx\\_http\\_handler\\_pt](#)
- [ngx\\_http\\_header\\_out\\_t](#)
- [ngx\\_http\\_header\\_t](#)
- [ngx\\_http\\_headers\\_in\\_t](#)
- [ngx\\_http\\_headers\\_out\\_t](#)
- [ngx\\_http\\_post\\_subrequest\\_pt](#)
- [ngx\\_http\\_post\\_subrequest\\_t](#)
- [ngx\\_http\\_posted\\_request\\_s](#)
- [ngx\\_http\\_posted\\_request\\_t](#)
- [ngx\\_http\\_postponed\\_request\\_s](#)
- [ngx\\_http\\_postponed\\_request\\_t](#)
- [ngx\\_http\\_request\\_body\\_t](#)
- [ngx\\_http\\_request\\_s](#)
- [ngx\\_http\\_state\\_e](#)

### Macros defined

- [NGX\\_HTTPS\\_CERT\\_ERROR](#)
- [NGX\\_HTTPS\\_NO\\_CERT](#)
- [NGX\\_HTTP\\_ACCEPTED](#)
- [NGX\\_HTTP\\_BAD\\_GATEWAY](#)
- [NGX\\_HTTP\\_BAD\\_REQUEST](#)
- [NGX\\_HTTP\\_CLIENT\\_CLOSED\\_REQUEST](#)

- [NGX HTTP CLIENT ERROR](#)
- [NGX HTTP CLOSE](#)
- [NGX HTTP CONFLICT](#)
- [NGX HTTP CONNECTION CLOSE](#)
- [NGX HTTP CONNECTION KEEP ALIVE](#)
- [NGX HTTP CONTINUE](#)
- [NGX HTTP COPY](#)
- [NGX HTTP COPY BUFFERED](#)
- [NGX HTTP CREATED](#)
- [NGX HTTP DELETE](#)
- [NGX HTTP DISCARD BUFFER SIZE](#)
- [NGX HTTP FORBIDDEN](#)
- [NGX HTTP GATEWAY TIME OUT](#)
- [NGX HTTP GET](#)
- [NGX HTTP GZIP BUFFERED](#)
- [NGX HTTP HEAD](#)
- [NGX HTTP INSUFFICIENT STORAGE](#)
- [NGX HTTP INTERNAL SERVER ERROR](#)
- [NGX HTTP LC HEADER LEN](#)
- [NGX HTTP LENGTH REQUIRED](#)
- [NGX HTTP LINGERING BUFFER SIZE](#)
- [NGX HTTP LOCK](#)
- [NGX HTTP LOG UNSAFE](#)
- [NGX HTTP LOWLEVEL BUFFERED](#)
- [NGX HTTP MAX SUBREQUESTS](#)
- [NGX HTTP MAX URI CHANGES](#)
- [NGX HTTP MKCOL](#)
- [NGX HTTP MOVE](#)
- [NGX HTTP MOVED PERMANENTLY](#)
- [NGX HTTP MOVED TEMPORARILY](#)
- [NGX HTTP NGINX CODES](#)
- [NGX HTTP NOT ALLOWED](#)
- [NGX HTTP NOT FOUND](#)

- [NGX HTTP NOT IMPLEMENTED](#)
- [NGX HTTP NOT MODIFIED](#)
- [NGX HTTP NO CONTENT](#)
- [NGX HTTP OK](#)
- [NGX HTTP OPTIONS](#)
- [NGX HTTP PARSE HEADER DONE](#)
- [NGX HTTP PARSE INVALID 09 METHOD](#)
- [NGX HTTP PARSE INVALID HEADER](#)
- [NGX HTTP PARSE INVALID METHOD](#)
- [NGX HTTP PARSE INVALID REQUEST](#)
- [NGX HTTP PARTIAL CONTENT](#)
- [NGX HTTP PATCH](#)
- [NGX HTTP POST](#)
- [NGX HTTP PRECONDITION FAILED](#)
- [NGX HTTP PROCESSING](#)
- [NGX HTTP PROPFIND](#)
- [NGX HTTP PROPPATCH](#)
- [NGX HTTP PUT](#)
- [NGX HTTP RANGE NOT SATISFIABLE](#)
- [NGX HTTP REQUEST ENTITY TOO LARGE](#)
- [NGX HTTP REQUEST HEADER TOO LARGE](#)
- [NGX HTTP REQUEST TIME OUT](#)
- [NGX HTTP REQUEST URI TOO LARGE](#)
- [NGX HTTP SEE OTHER](#)
- [NGX HTTP SERVICE UNAVAILABLE](#)
- [NGX HTTP SPECIAL RESPONSE](#)
- [NGX HTTP SSI BUFFERED](#)
- [NGX HTTP SUBREQUEST IN MEMORY](#)
- [NGX HTTP SUBREQUEST WAITED](#)
- [NGX HTTP SUB\\_BUFFERED](#)
- [NGX HTTP SWITCHING PROTOCOLS](#)
- [NGX HTTP TEMPORARY REDIRECT](#)
- [NGX HTTP TO HTTPS](#)

- [NGX\\_HTTP\\_TRACE](#)
- [NGX\\_HTTP\\_UNAUTHORIZED](#)
- [NGX\\_HTTP\\_UNKNOWN](#)
- [NGX\\_HTTP\\_UNLOCK](#)
- [NGX\\_HTTP\\_UNSUPPORTED\\_MEDIA\\_TYPE](#)
- [NGX\\_HTTP\\_VERSION\\_10](#)
- [NGX\\_HTTP\\_VERSION\\_11](#)
- [NGX\\_HTTP\\_VERSION\\_9](#)
- [NGX\\_HTTP\\_WRITE\\_BUFFERED](#)
- [NGX\\_NONE](#)
- [\\_NGX\\_HTTP\\_REQUEST\\_H\\_INCLUDED](#)
- [ngx\\_http\\_set\\_connection\\_log](#)
- [ngx\\_http\\_set\\_log\\_request](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_HTTP_REQUEST_H_INCLUDED_
9 #define _NGX_HTTP_REQUEST_H_INCLUDED_
10
11
12 #define NGX_HTTP_MAX_URI_CHANGES      10
13 #define NGX_HTTP_MAX_SUBREQUESTS      200
14
15 /* must be 2^n */
16 #define NGX_HTTP_LC_HEADER_LEN        32
17
18
19 #define NGX_HTTP_DISCARD_BUFFER_SIZE   4096
20 #define NGX_HTTP_LINGERING_BUFFER_SIZE 4096
21
22
23 #define NGX_HTTP_VERSION_9            9
24 #define NGX_HTTP_VERSION_10          1000
25 #define NGX_HTTP_VERSION_11          1001
26
27 #define NGX_HTTP_UNKNOWN              0x0001
28 #define NGX_HTTP_GET                  0x0002
29 #define NGX_HTTP_HEAD                 0x0004
30 #define NGX_HTTP_POST                 0x0008
31 #define NGX_HTTP_PUT                  0x0010
32 #define NGX_HTTP_DELETE               0x0020
33 #define NGX_HTTP_MKCOL                0x0040
34 #define NGX_HTTP_COPY                 0x0080
35 #define NGX_HTTP_MOVE                 0x0100
36 #define NGX_HTTP_OPTIONS              0x0200
37 #define NGX_HTTP_PROPFIND             0x0400
38 #define NGX_HTTP_PROPPATCH           0x0800
39 #define NGX_HTTP_LOCK                 0x1000
40 #define NGX_HTTP_UNLOCK               0x2000
41 #define NGX_HTTP_PATCH                0x4000
42 #define NGX_HTTP_TRACE                0x8000
43

```



```

44 #define NGX_HTTP_CONNECTION_CLOSE 1
45 #define NGX_HTTP_CONNECTION_KEEP_ALIVE 2
46
47
48 #define NGX_NONE 1
49
50
51 #define NGX_HTTP_PARSE_HEADER_DONE 1
52
53 #define NGX_HTTP_CLIENT_ERROR 10
54 #define NGX_HTTP_PARSE_INVALID_METHOD 10
55 #define NGX_HTTP_PARSE_INVALID_REQUEST 11
56 #define NGX_HTTP_PARSE_INVALID_09_METHOD 12
57
58 #define NGX_HTTP_PARSE_INVALID_HEADER 13
59
60
61 /* unused 1 */
62 #define NGX_HTTP_SUBREQUEST_IN_MEMORY 2
63 #define NGX_HTTP_SUBREQUEST_WAITED 4
64 #define NGX_HTTP_LOG_UNSAFE 8
65
66
67 #define NGX_HTTP_CONTINUE 100
68 #define NGX_HTTP_SWITCHING_PROTOCOLS 101
69 #define NGX_HTTP_PROCESSING 102
70
71 #define NGX_HTTP_OK 200
72 #define NGX_HTTP_CREATED 201
73 #define NGX_HTTP_ACCEPTED 202
74 #define NGX_HTTP_NO_CONTENT 204
75 #define NGX_HTTP_PARTIAL_CONTENT 206
76
77 #define NGX_HTTP_SPECIAL_RESPONSE 300
78 #define NGX_HTTP_MOVED_PERMANENTLY 301
79 #define NGX_HTTP_MOVED_TEMPORARILY 302
80 #define NGX_HTTP_SEE_OTHER 303
81 #define NGX_HTTP_NOT_MODIFIED 304
82 #define NGX_HTTP_TEMPORARY_REDIRECT 307
83
84 #define NGX_HTTP_BAD_REQUEST 400
85 #define NGX_HTTP_UNAUTHORIZED 401
86 #define NGX_HTTP_FORBIDDEN 403
87 #define NGX_HTTP_NOT_FOUND 404
88 #define NGX_HTTP_NOT_ALLOWED 405
89 #define NGX_HTTP_REQUEST_TIME_OUT 408
90 #define NGX_HTTP_CONFLICT 409
91 #define NGX_HTTP_LENGTH_REQUIRED 411
92 #define NGX_HTTP_PRECONDITION_FAILED 412
93 #define NGX_HTTP_REQUEST_ENTITY_TOO_LARGE 413
94 #define NGX_HTTP_REQUEST_URI_TOO_LARGE 414
95 #define NGX_HTTP_UNSUPPORTED_MEDIA_TYPE 415
96 #define NGX_HTTP_RANGE_NOT_SATISFIABLE 416
97
98
99 /* Our own HTTP codes */
100
101 /* The special code to close connection without any response */
102 #define NGX_HTTP_CLOSE 444
103
104 #define NGX_HTTP_NGINX_CODES 494
105
106 #define NGX_HTTP_REQUEST_HEADER_TOO_LARGE 494
107
108 #define NGX_HTTPS_CERT_ERROR 495
109 #define NGX_HTTPS_NO_CERT 496
110
111 /*
112 * We use the special code for the plain HTTP requests that are sent to
113 * HTTPS port to distinguish it from 4XX in an error page redirection
114 */
115 #define NGX_HTTP_TO_HTTPS 497
116
117 /* 498 is the canceled code for the requests with invalid host name */
118
119 /*

```

```

120 * HTTP does not define the code for the case when a client closed
121 * the connection while we are processing its request so we introduce
122 * own code to log such situation when a client has closed the connection
123 * before we even try to send the HTTP header to it
124 */
125 #define NGX_HTTP_CLIENT_CLOSED_REQUEST      499
126
127
128 #define NGX_HTTP_INTERNAL_SERVER_ERROR      500
129 #define NGX_HTTP_NOT_IMPLEMENTED           501
130 #define NGX_HTTP_BAD_GATEWAY                502
131 #define NGX_HTTP_SERVICE_UNAVAILABLE        503
132 #define NGX_HTTP_GATEWAY_TIME_OUT           504
133 #define NGX_HTTP_INSUFFICIENT_STORAGE       507
134
135
136 #define NGX_HTTP_LOWLEVEL_BUFFERED          0xf0
137 #define NGX_HTTP_WRITE_BUFFERED            0x10
138 #define NGX_HTTP_GZIP_BUFFERED             0x20
139 #define NGX_HTTP_SSI_BUFFERED              0x01
140 #define NGX_HTTP_SUB_BUFFERED              0x02
141 #define NGX_HTTP_COPY_BUFFERED             0x04
142
143
144 typedef enum {
145     NGX_HTTP_INITING_REQUEST_STATE = 0,
146     NGX_HTTP_READING_REQUEST_STATE,
147     NGX_HTTP_PROCESS_REQUEST_STATE,
148
149     NGX_HTTP_CONNECT_UPSTREAM_STATE,
150     NGX_HTTP_WRITING_UPSTREAM_STATE,
151     NGX_HTTP_READING_UPSTREAM_STATE,
152
153     NGX_HTTP_WRITING_REQUEST_STATE,
154     NGX_HTTP_LINGERING_CLOSE_STATE,
155     NGX_HTTP_KEEPALIVE_STATE
156 } ngx_http_state_e;
157
158
159 typedef struct {
160     ngx_str_t          name;
161     ngx_uint_t         offset;
162     ngx_http_header_pt handler;
163 } ngx_http_header_t;
164
165
166 typedef struct {
167     ngx_str_t          name;
168     ngx_uint_t         offset;
169 } ngx_http_header_out_t;
170
171
172 typedef struct {
173     ngx_list_t         headers;
174
175     ngx_table_elt_t   *host;
176     ngx_table_elt_t   *connection;
177     ngx_table_elt_t   *if_modified_since;
178     ngx_table_elt_t   *if_unmodified_since;
179     ngx_table_elt_t   *if_match;
180     ngx_table_elt_t   *if_none_match;
181     ngx_table_elt_t   *user_agent;
182     ngx_table_elt_t   *referer;
183     ngx_table_elt_t   *content_length;
184     ngx_table_elt_t   *content_type;
185
186     ngx_table_elt_t   *range;
187     ngx_table_elt_t   *if_range;
188
189     ngx_table_elt_t   *transfer_encoding;
190     ngx_table_elt_t   *expect;
191     ngx_table_elt_t   *upgrade;
192
193 #if (NGX_HTTP_GZIP)
194     ngx_table_elt_t   *accept_encoding;
195     ngx_table_elt_t   *via;

```

```

196 #endif
197
198     ngx_table_elt_t           *authorization;
199
200     ngx_table_elt_t           *keep_alive;
201
202     #if (NGX_HTTP_X_FORWARDED_FOR)
203     ngx_array_t                x_forwarded_for;
204 #endif
205
206     #if (NGX_HTTP_REALIP)
207     ngx_table_elt_t           *x_real_ip;
208 #endif
209
210     #if (NGX_HTTP_HEADERS)
211     ngx_table_elt_t           *accept;
212     ngx_table_elt_t           *accept_language;
213 #endif
214
215     #if (NGX_HTTP_DAV)
216     ngx_table_elt_t           *depth;
217     ngx_table_elt_t           *destination;
218     ngx_table_elt_t           *overwrite;
219     ngx_table_elt_t           *date;
220 #endif
221
222     ngx_str_t                  user;
223     ngx_str_t                  passwd;
224
225     ngx_array_t                cookies;
226
227     ngx_str_t                  server;
228     off_t                      content_length_n;
229     time_t                    keep_alive_n;
230
231     unsigned                   connection_type:2;
232     unsigned                   chunked:1;
233     unsigned                   msie:1;
234     unsigned                   msie6:1;
235     unsigned                   opera:1;
236     unsigned                   gecko:1;
237     unsigned                   chrome:1;
238     unsigned                   safari:1;
239     unsigned                   konqueror:1;
240 } ngx_http_headers_in_t;
241
242
243 typedef struct {
244     ngx_list_t                 headers;
245
246     ngx_uint_t                 status;
247     ngx_str_t                  status_line;
248
249     ngx_table_elt_t           *server;
250     ngx_table_elt_t           *date;
251     ngx_table_elt_t           *content_length;
252     ngx_table_elt_t           *content_encoding;
253     ngx_table_elt_t           *location;
254     ngx_table_elt_t           *refresh;
255     ngx_table_elt_t           *last_modified;
256     ngx_table_elt_t           *content_range;
257     ngx_table_elt_t           *accept_ranges;
258     ngx_table_elt_t           *www_authenticate;
259     ngx_table_elt_t           *expires;
260     ngx_table_elt_t           *etag;
261
262     ngx_str_t                  *override_charset;
263
264     size_t                     content_type_len;
265     ngx_str_t                  content_type;
266     ngx_str_t                  charset;
267     u_char                     *content_type_lowercase;
268     ngx_uint_t                 content_type_hash;
269
270     ngx_array_t                cache_control;
271

```

```

272     off_t                content_length_n;
273     time_t               date_time;
274     time_t               last_modified_time;
275 } ngx_http_headers_out_t;
276
277
278 typedef void (*ngx_http_client_body_handler_pt)(ngx_http_request_t *r);
279
280 typedef struct {
281     ngx_temp_file_t      *temp_file;
282     ngx_chain_t          *bufs;
283     ngx_buf_t            *buf;
284     off_t                 rest;
285     ngx_chain_t          *free;
286     ngx_chain_t          *busy;
287     ngx_http_chunked_t   *chunked;
288     ngx_http_client_body_handler_pt post_handler;
289 } ngx_http_request_body_t;
290
291
292 typedef struct ngx_http_addr_conf_s ngx_http_addr_conf_t;
293
294 typedef struct {
295     ngx_http_addr_conf_t *addr_conf;
296     ngx_http_conf_ctx_t  *conf_ctx;
297
298     #if (NGX_HTTP_SSL && defined SSL_CTRL_SET_TLSEXT_HOSTNAME)
299     ngx_str_t              *ssl_servername;
300     #if (NGX_PCRE)
301     ngx_http_regex_t      *ssl_servername_regex;
302     #endif
303     #endif
304
305     ngx_buf_t              **busy;
306     ngx_int_t              nbusy;
307
308     ngx_buf_t              **free;
309     ngx_int_t              nfree;
310
311     #if (NGX_HTTP_SSL)
312     unsigned               ssl:1;
313     #endif
314     unsigned               proxy_protocol:1;
315 } ngx_http_connection_t;
316
317
318 typedef void (*ngx_http_cleanup_pt)(void *data);
319
320 typedef struct ngx_http_cleanup_s ngx_http_cleanup_t;
321
322 struct ngx_http_cleanup_s {
323     ngx_http_cleanup_pt  handler;
324     void                 *data;
325     ngx_http_cleanup_t  *next;
326 };
327
328
329 typedef ngx_int_t (*ngx_http_post_subrequest_pt)(ngx_http_request_t *r,
330     void *data, ngx_int_t rc);
331
332 typedef struct {
333     ngx_http_post_subrequest_pt handler;
334     void                 *data;
335 } ngx_http_post_subrequest_t;
336
337
338 typedef struct ngx_http_postponed_request_s ngx_http_postponed_request_t;
339
340 struct ngx_http_postponed_request_s {
341     ngx_http_request_t    *request;
342     ngx_chain_t           *out;
343     ngx_http_postponed_request_t *next;
344 };
345
346
347 typedef struct ngx_http_posted_request_s ngx_http_posted_request_t;

```

```

348 struct ngx_http_posted_request_s {
349     ngx_http_request_t      *request;
350     ngx_http_posted_request_t *next;
351 };
352
353
354
355 typedef ngx_int_t (*ngx_http_handler_pt)(ngx_http_request_t *r);
356 typedef void (*ngx_http_event_handler_pt)(ngx_http_request_t *r);
357
358
359 struct ngx_http_request_s {
360     uint32_t                signature;           /* "HTTP" */
361
362     ngx_connection_t        *connection;
363
364     void                    **ctx;
365     void                    **main_conf;
366     void                    **srv_conf;
367     void                    **loc_conf;
368
369     ngx_http_event_handler_pt read_event_handler;
370     ngx_http_event_handler_pt write_event_handler;
371
372     #if (NGX_HTTP_CACHE)
373     ngx_http_cache_t        *cache;
374     #endif
375
376     ngx_http_upstream_t     *upstream;
377     ngx_array_t             *upstream_states;
378                             /* of ngx_http_upstream_state_t */
379
380     ngx_pool_t              *pool;
381     ngx_buf_t               *header_in;
382
383     ngx_http_headers_in_t   headers_in;
384     ngx_http_headers_out_t  headers_out;
385
386     ngx_http_request_body_t *request_body;
387
388     time_t                  lingering_time;
389     time_t                  start_sec;
390     ngx_msec_t              start_msec;
391
392     ngx_uint_t              method;
393     ngx_uint_t              http_version;
394
395     ngx_str_t               request_line;
396     ngx_str_t               uri;
397     ngx_str_t               args;
398     ngx_str_t               exten;
399     ngx_str_t               unparsed_uri;
400
401     ngx_str_t               method_name;
402     ngx_str_t               http_protocol;
403
404     ngx_chain_t             *out;
405     ngx_http_request_t     *main;
406     ngx_http_request_t     *parent;
407     ngx_http_postponed_request_t *postponed;
408     ngx_http_post_subrequest_t *post_subrequest;
409     ngx_http_posted_request_t *posted_requests;
410
411     ngx_int_t               phase_handler;
412     ngx_http_handler_pt     content_handler;
413     ngx_uint_t              access_code;
414
415     ngx_http_variable_value_t *variables;
416
417     #if (NGX_PCRE)
418     ngx_uint_t              ncaptures;
419     int                     *captures;
420     u_char                  *captures_data;
421     #endif
422
423     size_t                  limit_rate;

```

```

424     size_t                limit_rate_after;
425
426     /* used to learn the Apache compatible response length without a header */
427     size_t                header_size;
428
429     off_t                 request_length;
430
431     ngx_uint_t           err_status;
432
433     ngx_http_connection_t *http_connection;
434     #if (NGX_HTTP_SPDY)
435     ngx_http_spdy_stream_t *spdy_stream;
436     #endif
437
438     ngx_http_log_handler_pt log_handler;
439
440     ngx_http_cleanup_t   *cleanup;
441
442     unsigned               subrequests:8;
443     unsigned               count:8;
444     unsigned               blocked:8;
445
446     unsigned               aio:1;
447
448     unsigned               http_state:4;
449
450     /* URI with "/" and on Win32 with "/" */
451     unsigned               complex_uri:1;
452
453     /* URI with "%" */
454     unsigned               quoted_uri:1;
455
456     /* URI with "+" */
457     unsigned               plus_in_uri:1;
458
459     /* URI with " " */
460     unsigned               space_in_uri:1;
461
462     unsigned               invalid_header:1;
463
464     unsigned               add_uri_to_alias:1;
465     unsigned               valid_location:1;
466     unsigned               valid_unparsed_uri:1;
467     unsigned               uri_changed:1;
468     unsigned               uri_changes:4;
469
470     unsigned               request_body_in_single_buf:1;
471     unsigned               request_body_in_file_only:1;
472     unsigned               request_body_in_persistent_file:1;
473     unsigned               request_body_in_clean_file:1;
474     unsigned               request_body_file_group_access:1;
475     unsigned               request_body_file_log_level:3;
476
477     unsigned               subrequest_in_memory:1;
478     unsigned               waited:1;
479
480     #if (NGX_HTTP_CACHE)
481     unsigned               cached:1;
482     #endif
483
484     #if (NGX_HTTP_GZIP)
485     unsigned               gzip_tested:1;
486     unsigned               gzip_ok:1;
487     unsigned               gzip_vary:1;
488     #endif
489
490     unsigned               proxy:1;
491     unsigned               bypass_cache:1;
492     unsigned               no_cache:1;
493
494     /*
495      * instead of using the request context data in
496      * ngx_http_limit_conn_module and ngx_http_limit_req_module
497      * we use the single bits in the request structure
498      */
499     unsigned               limit_conn_set:1;

```

```

500     unsigned                limit_req_set:1;
501
502 #if 0
503     unsigned                cacheable:1;
504 #endif
505
506     unsigned                pipeline:1;
507     unsigned                chunked:1;
508     unsigned                header_only:1;
509     unsigned                keepalive:1;
510     unsigned                lingering_close:1;
511     unsigned                discard_body:1;
512     unsigned                internal:1;
513     unsigned                error_page:1;
514     unsigned                filter_finalize:1;
515     unsigned                post_action:1;
516     unsigned                request_complete:1;
517     unsigned                request_output:1;
518     unsigned                header_sent:1;
519     unsigned                expect_tested:1;
520     unsigned                root_tested:1;
521     unsigned                done:1;
522     unsigned                logged:1;
523
524     unsigned                buffered:4;
525
526     unsigned                main_filter_need_in_memory:1;
527     unsigned                filter_need_in_memory:1;
528     unsigned                filter_need_temporary:1;
529     unsigned                allow_ranges:1;
530     unsigned                single_range:1;
531     unsigned                disable_not_modified:1;
532
533 #if (NGX_STAT_STUB)
534     unsigned                stat_reading:1;
535     unsigned                stat_writing:1;
536 #endif
537
538     /* used to parse HTTP headers */
539
540     ngx_uint_t              state;
541
542     ngx_uint_t              header_hash;
543     ngx_uint_t              lowercase_index;
544     u_char                  lowercase_header[NGX_HTTP_LC_HEADER_LEN];
545
546     u_char                  *header_name_start;
547     u_char                  *header_name_end;
548     u_char                  *header_start;
549     u_char                  *header_end;
550
551     /*
552      * a memory that can be reused after parsing a request line
553      * via ngx\_http\_ephemeral\_t
554      */
555
556     u_char                  *uri_start;
557     u_char                  *uri_end;
558     u_char                  *uri_ext;
559     u_char                  *args_start;
560     u_char                  *request_start;
561     u_char                  *request_end;
562     u_char                  *method_end;
563     u_char                  *schema_start;
564     u_char                  *schema_end;
565     u_char                  *host_start;
566     u_char                  *host_end;
567     u_char                  *port_start;
568     u_char                  *port_end;
569
570     unsigned                http_minor:16;
571     unsigned                http_major:16;
572 };
573
574
575 typedef struct {

```

```

576     ngx\_http\_posted\_request\_t         terminal_posted_request;
577 #if (NGX_HAVE_AIO_SENDFILE)
578     u_char                             aio_preload;
579 #endif
580 } ngx\_http\_ephemeral\_t;
581
582
583 extern ngx\_http\_header\_t             ngx\_http\_headers\_in[];
584 extern ngx\_http\_header\_out\_t        ngx\_http\_headers\_out[];
585
586
587 #define ngx\_http\_set\_connection\_log(c, l)           \
588                                                     \
589     c->log->file = l->file;                          \
590     c->log->next = l->next;                          \
591     c->log->writer = l->writer;                      \
592     c->log->wdata = l->wdata;                        \
593     if (!(c->log->log_level & NGX\_LOG\_DEBUG\_CONNECTION)) { \
594         c->log->log_level = l->log_level;            \
595     }
596
597
598 #define ngx\_http\_set\_log\_request(log, r)           \
599     ((ngx\_http\_log\_ctx\_t *) log->data)->current_request = r
600
601
602 #endif /* NGX\_HTTP\_REQUEST\_H\_INCLUDED */

```

[One Level Up](#)

[Top Level](#)



## src/core/nginx\_list.h - nginx-1.7.10

### Data types defined

- [ngx\\_list\\_part\\_s](#)
- [ngx\\_list\\_part\\_t](#)
- [ngx\\_list\\_t](#)

### Functions defined

- [ngx\\_list\\_init](#)

### Macros defined

- [\\_NGX\\_LIST\\_H\\_INCLUDED](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_LIST_H_INCLUDED
9 #define _NGX_LIST_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 typedef struct ngx\_list\_part\_s ngx_list_part_t;
17
18 struct ngx_list_part_s {
19     void *elts;
20     ngx\_uint\_t nelts;
21     ngx\_list\_part\_t *next;
22 };
23
24
25 typedef struct {
26     ngx\_list\_part\_t *last;
27     ngx\_list\_part\_t part;
28     size_t size;
29     ngx\_uint\_t nalloc;
30     ngx\_pool\_t *pool;
31 } ngx_list_t;
32
33
34 ngx\_list\_t *ngx_list_create(ngx\_pool\_t *pool, ngx\_uint\_t n, size_t size);
35
36 static ngx\_inline ngx\_int\_t
37 ngx_list_init(ngx\_list\_t *list, ngx\_pool\_t *pool, ngx\_uint\_t n, size_t size)
38 {
39     list->part.elts = ngx\_palloc(pool, n * size);
40     if (list->part.elts == NULL) {
41         return NGX\_ERROR;
42     }
43
44     list->part.nelts = 0;
45     list->part.next = NULL;
46     list->last = &list->part;
```

```

47     list->size = size;
48     list->nalloc = n;
49     list->pool = pool;
50
51     return NGX\_OK;
52 }
53
54
55 /*
56 *
57 * the iteration through the list:
58 *
59 * part = &list.part;
60 * data = part->elts;
61 *
62 * for (i = 0 ;; i++) {
63 *
64 *     if (i >= part->nelts) {
65 *         if (part->next == NULL) {
66 *             break;
67 *         }
68 *
69 *         part = part->next;
70 *         data = part->elts;
71 *         i = 0;
72 *     }
73 *
74 *     ... data[i] ...
75 * }
76 */
77
78
79
80 void *ngx\_list\_push(ngx\_list\_t *list);
81
82
83 #endif /* \_NGX\_LIST\_H\_INCLUDED */

```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_list.c - nginx-1.7.10

### Functions defined

- [ngx\\_list\\_create](#)
- [ngx\\_list\\_push](#)

### Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10
11
12  ngx_list_t *
13  ngx_list_create(ngx_pool_t *pool, ngx_uint_t n, size_t size)
14  {
15      ngx_list_t *list;
16
17      list = ngx_palloc(pool, sizeof(ngx_list_t));
18      if (list == NULL) {
19          return NULL;
20      }
21
22      if (ngx_list_init(list, pool, n, size) != NGX_OK) {
23          return NULL;
24      }
25
26      return list;
27  }
28
29
30  void *
31  ngx_list_push(ngx_list_t *l)
32  {
33      void *elt;
34      ngx_list_part_t *last;
35
36      last = l->last;
37
38      if (last->nelts == l->nalloc) {
39          /* the last part is full, allocate a new list part */
40
41          last = ngx_palloc(l->pool, sizeof(ngx_list_part_t));
42          if (last == NULL) {
43              return NULL;
44          }
45
46          last->elts = ngx_palloc(l->pool, l->nalloc * l->size);
47          if (last->elts == NULL) {
48              return NULL;
49          }
50
51          last->nelts = 0;
52          last->next = NULL;
53
54          l->last->next = last;
55          l->last = last;
56      }
57
58      elt = (char *) last->elts + l->size * last->nelts;
59      last->nelts++;
60  }
```

```
61  
62 return elt;  
63 }
```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_hash.h - nginx-1.7.10

### Data types defined

- [ngx\\_hash\\_combined\\_t](#)
- [ngx\\_hash\\_elt\\_t](#)
- [ngx\\_hash\\_init\\_t](#)
- [ngx\\_hash\\_key\\_pt](#)
- [ngx\\_hash\\_key\\_t](#)
- [ngx\\_hash\\_keys\\_arrays\\_t](#)
- [ngx\\_hash\\_t](#)
- [ngx\\_hash\\_wildcard\\_t](#)
- [ngx\\_table\\_elt\\_t](#)

### Macros defined

- [NGX\\_HASH\\_LARGE](#)
- [NGX\\_HASH\\_LARGE\\_ASIZE](#)
- [NGX\\_HASH\\_LARGE\\_HSIZE](#)
- [NGX\\_HASH\\_READONLY\\_KEY](#)
- [NGX\\_HASH\\_SMALL](#)
- [NGX\\_HASH\\_WILDCARD\\_KEY](#)
- [\\_NGX\\_HASH\\_H\\_INCLUDED\\_](#)
- [ngx\\_hash](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_HASH_H_INCLUDED_
9 #define _NGX_HASH_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 typedef struct {
17     void *value;
18     u_short len;
19     u_char name[1];
20 } ngx_hash_elt_t;
21
22
23 typedef struct {
```

```

24     ngx_hash_elt_t  **buckets;
25     ngx_uint_t      size;
26 } ngx_hash_t;
27
28
29 typedef struct {
30     ngx_hash_t      hash;
31     void            *value;
32 } ngx_hash_wildcard_t;
33
34
35 typedef struct {
36     ngx_str_t        key;
37     ngx_uint_t       key_hash;
38     void            *value;
39 } ngx_hash_key_t;
40
41
42 typedef ngx_uint_t (*ngx_hash_key_pt) (u_char *data, size_t len);
43
44
45 typedef struct {
46     ngx_hash_t        hash;
47     ngx_hash_wildcard_t *wc_head;
48     ngx_hash_wildcard_t *wc_tail;
49 } ngx_hash_combined_t;
50
51
52 typedef struct {
53     ngx_hash_t        *hash;
54     ngx_hash_key_pt    key;
55
56     ngx_uint_t        max_size;
57     ngx_uint_t        bucket_size;
58
59     char              *name;
60     ngx_pool_t        *pool;
61     ngx_pool_t        *temp_pool;
62 } ngx_hash_init_t;
63
64
65 #define NGX_HASH_SMALL          1
66 #define NGX_HASH_LARGE         2
67
68 #define NGX_HASH_LARGE_ASIZE    16384
69 #define NGX_HASH_LARGE_HSIZE   10007
70
71 #define NGX_HASH_WILDCARD_KEY   1
72 #define NGX_HASH_READONLY_KEY  2
73
74
75 typedef struct {
76     ngx_uint_t        hsize;
77
78     ngx_pool_t        *pool;
79     ngx_pool_t        *temp_pool;
80
81     ngx_array_t       keys;
82     ngx_array_t       *keys_hash;
83
84     ngx_array_t       dns_wc_head;
85     ngx_array_t       *dns_wc_head_hash;
86
87     ngx_array_t       dns_wc_tail;
88     ngx_array_t       *dns_wc_tail_hash;
89 } ngx_hash_keys_arrays_t;
90
91
92 typedef struct {
93     ngx_uint_t        hash;
94     ngx_str_t         key;
95     ngx_str_t         value;
96     u_char            *lowercase_key;
97 } ngx_table_elt_t;
98
99

```

```
100 void *ngx_hash_find(ngx_hash_t *hash, ngx_uint_t key, u_char *name, size_t len);
101 void *ngx_hash_find_wc_head(ngx_hash_wildcard_t *hwc, u_char *name, size_t len);
102 void *ngx_hash_find_wc_tail(ngx_hash_wildcard_t *hwc, u_char *name, size_t len);
103 void *ngx_hash_find_combined(ngx_hash_combined_t *hash, ngx_uint_t key,
104     u_char *name, size_t len);
105
106 ngx_int_t ngx_hash_init(ngx_hash_init_t *hinit, ngx_hash_key_t *names,
107     ngx_uint_t nelts);
108 ngx_int_t ngx_hash_wildcard_init(ngx_hash_init_t *hinit, ngx_hash_key_t *names,
109     ngx_uint_t nelts);
110
111 #define ngx_hash(key, c) ((ngx_uint_t) key * 31 + c)
112 ngx_uint_t ngx_hash_key(u_char *data, size_t len);
113 ngx_uint_t ngx_hash_key_lc(u_char *data, size_t len);
114 ngx_uint_t ngx_hash_strlow(u_char *dst, u_char *src, size_t n);
115
116
117 ngx_int_t ngx_hash_keys_array_init(ngx_hash_keys_arrays_t *ha, ngx_uint_t type);
118 ngx_int_t ngx_hash_add_key(ngx_hash_keys_arrays_t *ha, ngx_str_t *key,
119     void *value, ngx_uint_t flags);
120
121
122 #endif /* _NGX_HASH_H_INCLUDED_ */
```

[One Level Up](#)

[Top Level](#)

# src/core/nginx\_hash.c - nginx-1.7.10

## Functions defined

- [ngx\\_hash\\_add\\_key](#)
- [ngx\\_hash\\_find](#)
- [ngx\\_hash\\_find\\_combined](#)
- [ngx\\_hash\\_find\\_wc\\_head](#)
- [ngx\\_hash\\_find\\_wc\\_tail](#)
- [ngx\\_hash\\_init](#)
- [ngx\\_hash\\_key](#)
- [ngx\\_hash\\_key\\_lc](#)
- [ngx\\_hash\\_keys\\_array\\_init](#)
- [ngx\\_hash\\_strlow](#)
- [ngx\\_hash\\_wildcard\\_init](#)

## Macros defined

- [NGX\\_HASH\\_ELT\\_SIZE](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10
11
12 void *
13 ngx_hash_find(ngx_hash_t *hash, ngx_uint_t key, u_char *name, size_t len)
14 {
15     ngx_uint_t i;
16     ngx_hash_elt_t *elt;
17
18     #if 0
19     ngx_log_error(NGX_LOG_ALERT, ngx_cycle->log, 0, "hf: \"%s\"", len, name);
20     #endif
21
22     elt = hash->buckets[key % hash->size];
23
24     if (elt == NULL) {
25         return NULL;
26     }
27
28     while (elt->value) {
29         if (len != (size_t) elt->len) {
30             goto next;
31         }
32
33         for (i = 0; i < len; i++) {
34             if (name[i] != elt->name[i]) {
```



```

35         goto next;
36     }
37 }
38
39     return elt->value;
40
41 next:
42
43     elt = (ngx_hash_elt_t *) ngx_align_ptr(&elt->name[0] + elt->len,
44                                         sizeof(void *));
45     continue;
46 }
47
48 return NULL;
49 }
50
51
52 void *
53 ngx_hash_find_wc_head(ngx_hash_wildcard_t *hwc, u_char *name, size_t len)
54 {
55     void *value;
56     ngx_uint_t i, n, key;
57
58 #if 0
59     ngx_log_error(NGX_LOG_ALERT, ngx_cycle->log, 0, "wch:\">%*s\\"", len, name);
60 #endif
61
62     n = len;
63
64     while (n) {
65         if (name[n - 1] == '.') {
66             break;
67         }
68
69         n--;
70     }
71
72     key = 0;
73
74     for (i = n; i < len; i++) {
75         key = ngx_hash(key, name[i]);
76     }
77
78 #if 0
79     ngx_log_error(NGX_LOG_ALERT, ngx_cycle->log, 0, "key:\">%ui\\"", key);
80 #endif
81
82     value = ngx_hash_find(&hwc->hash, key, &name[n], len - n);
83
84 #if 0
85     ngx_log_error(NGX_LOG_ALERT, ngx_cycle->log, 0, "value:\">%p\\"", value);
86 #endif
87
88     if (value) {
89
90         /*
91          * the 2 low bits of value have the special meaning:
92          * 00 - value is data pointer for both "example.com"
93          *      and "*.example.com";
94          * 01 - value is data pointer for "*.example.com" only;
95          * 10 - value is pointer to wildcard hash allowing
96          *      both "example.com" and "*.example.com";
97          * 11 - value is pointer to wildcard hash allowing
98          *      "*.example.com" only.
99          */
100
101         if ((uintptr_t) value & 2) {
102
103             if (n == 0) {
104
105                 /* "example.com" */
106
107                 if ((uintptr_t) value & 1) {
108                     return NULL;
109                 }
110

```

```

111         hwc = (ngx\_hash\_wildcard\_t *)
112             ((uintptr\_t) value & (uintptr\_t) ~3);
113         return hwc->value;
114     }
115
116     hwc = (ngx\_hash\_wildcard\_t *) ((uintptr\_t) value & (uintptr\_t) ~3);
117
118     value = ngx\_hash\_find\_wc\_head(hwc, name, n - 1);
119
120     if (value) {
121         return value;
122     }
123
124     return hwc->value;
125 }
126
127 if ((uintptr\_t) value & 1) {
128
129     if (n == 0) {
130
131         /* "example.com" */
132
133         return NULL;
134     }
135
136     return (void *) ((uintptr\_t) value & (uintptr\_t) ~3);
137 }
138
139 return value;
140 }
141
142 return hwc->value;
143 }
144
145
146 void *
147 ngx\_hash\_find\_wc\_tail(ngx\_hash\_wildcard\_t *hwc, u\_char *name, size\_t len)
148 {
149     void *value;
150     ngx\_uint\_t i, key;
151
152     #if 0
153     ngx\_log\_error(NGX\_LOG\_ALERT, ngx\_cycle->log, 0, "wct:\">%s\\"", len, name);
154     #endif
155
156     key = 0;
157
158     for (i = 0; i < len; i++) {
159         if (name[i] == '.') {
160             break;
161         }
162
163         key = ngx\_hash(key, name[i]);
164     }
165
166     if (i == len) {
167         return NULL;
168     }
169
170     #if 0
171     ngx\_log\_error(NGX\_LOG\_ALERT, ngx\_cycle->log, 0, "key:\">%ui\\"", key);
172     #endif
173
174     value = ngx\_hash\_find(&hwc->hash, key, name, i);
175
176     #if 0
177     ngx\_log\_error(NGX\_LOG\_ALERT, ngx\_cycle->log, 0, "value:\">%p\\"", value);
178     #endif
179
180     if (value) {
181
182         /*
183          * the 2 low bits of value have the special meaning:
184          * 00 - value is data pointer;
185          * 11 - value is pointer to wildcard hash allowing "example.*".
186          */

```

```

187     if ((uintptr_t) value & 2) {
188
189         i++;
190
191         hwc = (ngx\_hash\_wildcard\_t *) ((uintptr_t) value & (uintptr_t) ~3);
192
193         value = ngx\_hash\_find\_wc\_tail(hwc, &name[i], len - i);
194
195         if (value) {
196             return value;
197         }
198
199         return hwc->value;
200     }
201
202     return value;
203 }
204
205 return hwc->value;
206 }
207
208
209 void *
210 ngx\_hash\_find\_combined(ngx\_hash\_combined\_t *hash, ngx\_uint\_t key, u_char *name,
211 size\_t len)
212 {
213     void *value;
214
215     if (hash->hash.buckets) {
216         value = ngx\_hash\_find(&hash->hash, key, name, len);
217
218         if (value) {
219             return value;
220         }
221     }
222
223     if (len == 0) {
224         return NULL;
225     }
226
227     if (hash->wc_head && hash->wc_head->hash.buckets) {
228         value = ngx\_hash\_find\_wc\_head(hash->wc_head, name, len);
229
230         if (value) {
231             return value;
232         }
233     }
234
235     if (hash->wc_tail && hash->wc_tail->hash.buckets) {
236         value = ngx\_hash\_find\_wc\_tail(hash->wc_tail, name, len);
237
238         if (value) {
239             return value;
240         }
241     }
242
243     return NULL;
244 }
245
246
247 #define NGX\_HASH\_ELT\_SIZE(name) \
248     (sizeof(void *) + ngx\_align((name)->key.len + 2, sizeof(void *)))
249
250
251 ngx\_int\_t
252 ngx\_hash\_init(ngx\_hash\_init\_t *hinit, ngx\_hash\_key\_t *names, ngx\_uint\_t nelts)
253 {
254     u_char *elts;
255     size\_t len;
256     u_short *test;
257     ngx\_uint\_t i, n, key, size, start, bucket_size;
258     ngx\_hash\_elt\_t *elt, **buckets;
259
260     for (n = 0; n < nelts; n++) {
261         if (hinit->bucket_size < NGX\_HASH\_ELT\_SIZE(&names[n]) + sizeof(void *))
262     {

```

```

263     ngx_log_error(NGX_LOG_EMERG, hinit->pool->log, 0,
264                 "could not build the %s, you should "
265                 "increase %s_bucket_size: %i",
266                 hinit->name, hinit->name, hinit->bucket_size);
267     return NGX_ERROR;
268 }
269 }
270
271 test = ngx_alloc(hinit->max_size * sizeof(u_short), hinit->pool->log);
272 if (test == NULL) {
273     return NGX_ERROR;
274 }
275
276 bucket_size = hinit->bucket_size - sizeof(void *);
277
278 start = nelts / (bucket_size / (2 * sizeof(void *)));
279 start = start ? start : 1;
280
281 if (hinit->max_size > 10000 && nelts && hinit->max_size / nelts < 100) {
282     start = hinit->max_size - 1000;
283 }
284
285 for (size = start; size <= hinit->max_size; size++) {
286
287     ngx_memzero(test, size * sizeof(u_short));
288
289     for (n = 0; n < nelts; n++) {
290         if (names[n].key.data == NULL) {
291             continue;
292         }
293
294         key = names[n].key_hash % size;
295         test[key] = (u_short) (test[key] + NGX_HASH_ELT_SIZE(&names[n]));
296
297 #if 0
298         ngx_log_error(NGX_LOG_ALERT, hinit->pool->log, 0,
299                     "%ui: %ui %ui \"%V\"",
300                     size, key, test[key], &names[n].key);
301 #endif
302
303         if (test[key] > (u_short) bucket_size) {
304             goto next;
305         }
306     }
307
308     goto found;
309
310 next:
311
312     continue;
313 }
314
315 size--;
316
317 ngx_log_error(NGX_LOG_WARN, hinit->pool->log, 0,
318             "could not build optimal %s, you should increase "
319             "either %s_max_size: %i or %s_bucket_size: %i; "
320             "ignoring %s_bucket_size",
321             hinit->name, hinit->name, hinit->max_size,
322             hinit->name, hinit->bucket_size, hinit->name);
323
324 found:
325
326 for (i = 0; i < size; i++) {
327     test[i] = sizeof(void *);
328 }
329
330 for (n = 0; n < nelts; n++) {
331     if (names[n].key.data == NULL) {
332         continue;
333     }
334
335     key = names[n].key_hash % size;
336     test[key] = (u_short) (test[key] + NGX_HASH_ELT_SIZE(&names[n]));
337 }
338

```

```

339 len = 0;
340
341 for (i = 0; i < size; i++) {
342     if (test[i] == sizeof(void *)) {
343         continue;
344     }
345
346     test[i] = (u_short) (ngx_align(test[i], ngx_cacheline_size));
347
348     len += test[i];
349 }
350
351 if (hinit->hash == NULL) {
352     hinit->hash = ngx_pcalloc(hinit->pool, sizeof(ngx_hash_wildcard_t)
353                             + size * sizeof(ngx_hash_elt_t *));
354     if (hinit->hash == NULL) {
355         ngx_free(test);
356         return NGX_ERROR;
357     }
358
359     buckets = (ngx_hash_elt_t **)
360         ((u_char *) hinit->hash + sizeof(ngx_hash_wildcard_t));
361
362 } else {
363     buckets = ngx_pcalloc(hinit->pool, size * sizeof(ngx_hash_elt_t *));
364     if (buckets == NULL) {
365         ngx_free(test);
366         return NGX_ERROR;
367     }
368 }
369
370 elts = ngx_palloc(hinit->pool, len + ngx_cacheline_size);
371 if (elts == NULL) {
372     ngx_free(test);
373     return NGX_ERROR;
374 }
375
376 elts = ngx_align_ptr(elts, ngx_cacheline_size);
377
378 for (i = 0; i < size; i++) {
379     if (test[i] == sizeof(void *)) {
380         continue;
381     }
382
383     buckets[i] = (ngx_hash_elt_t *) elts;
384     elts += test[i];
385
386 }
387
388 for (i = 0; i < size; i++) {
389     test[i] = 0;
390 }
391
392 for (n = 0; n < nelts; n++) {
393     if (names[n].key.data == NULL) {
394         continue;
395     }
396
397     key = names[n].key_hash % size;
398     elt = (ngx_hash_elt_t *) ((u_char *) buckets[key] + test[key]);
399
400     elt->value = names[n].value;
401     elt->len = (u_short) names[n].key.len;
402
403     ngx_strlow(elt->name, names[n].key.data, names[n].key.len);
404
405     test[key] = (u_short) (test[key] + NGX_HASH_ELT_SIZE(&names[n]));
406 }
407
408 for (i = 0; i < size; i++) {
409     if (buckets[i] == NULL) {
410         continue;
411     }
412
413     elt = (ngx_hash_elt_t *) ((u_char *) buckets[i] + test[i]);
414

```

```

415     elt->value = NULL;
416 }
417
418 ngx_free(test);
419
420 hinit->hash->buckets = buckets;
421 hinit->hash->size = size;
422
423 #if 0
424
425 for (i = 0; i < size; i++) {
426     ngx_str_t  val;
427     ngx_uint_t key;
428
429     elt = buckets[i];
430
431     if (elt == NULL) {
432         ngx_log_error(NGX_LOG_ALERT, hinit->pool->log, 0,
433             "%ui: NULL", i);
434         continue;
435     }
436
437     while (elt->value) {
438         val.len = elt->len;
439         val.data = &elt->name[0];
440
441         key = hinit->key(val.data, val.len);
442
443         ngx_log_error(NGX_LOG_ALERT, hinit->pool->log, 0,
444             "%ui: %p \"%V\" %ui", i, elt, &val, key);
445
446         elt = (ngx_hash_elt_t *) ngx_align_ptr(&elt->name[0] + elt->len,
447             sizeof(void *));
448     }
449 }
450
451 #endif
452
453 return NGX_OK;
454 }
455
456 ngx_int_t
457 ngx_hash_wildcard_init(ngx_hash_init_t *hinit, ngx_hash_key_t *names,
458     ngx_uint_t nelts)
459 {
460     size_t      len, dot_len;
461     ngx_uint_t  i, n, dot;
462     ngx_array_t curr_names, next_names;
463     ngx_hash_key_t *name, *next_name;
464     ngx_hash_init_t h;
465     ngx_hash_wildcard_t *wdc;
466
467     if (ngx_array_init(&curr_names, hinit->temp_pool, nelts,
468         sizeof(ngx_hash_key_t))
469         != NGX_OK)
470     {
471         return NGX_ERROR;
472     }
473
474     if (ngx_array_init(&next_names, hinit->temp_pool, nelts,
475         sizeof(ngx_hash_key_t))
476         != NGX_OK)
477     {
478         return NGX_ERROR;
479     }
480
481     for (n = 0; n < nelts; n = i) {
482
483     #if 0
484         ngx_log_error(NGX_LOG_ALERT, hinit->pool->log, 0,
485             "wc0: \"%V\"", &names[n].key);
486     #endif
487
488     dot = 0;
489
490

```

```

491     for (len = 0; len < names[n].key.len; len++) {
492         if (names[n].key.data[len] == '.') {
493             dot = 1;
494             break;
495         }
496     }
497
498     name = ngx_array_push(&curr_names);
499     if (name == NULL) {
500         return NGX_ERROR;
501     }
502
503     name->key.len = len;
504     name->key.data = names[n].key.data;
505     name->key_hash = hinit->key(name->key.data, name->key.len);
506     name->value = names[n].value;
507
508     #if 0
509     ngx_log_error(NGX_LOG_ALERT, hinit->pool->log, 0,
510                 "wc1: \"%V\" %ui", &name->key, dot);
511     #endif
512
513     dot_len = len + 1;
514
515     if (dot) {
516         len++;
517     }
518
519     next_names.nelts = 0;
520
521     if (names[n].key.len != len) {
522         next_name = ngx_array_push(&next_names);
523         if (next_name == NULL) {
524             return NGX_ERROR;
525         }
526
527         next_name->key.len = names[n].key.len - len;
528         next_name->key.data = names[n].key.data + len;
529         next_name->key_hash = 0;
530         next_name->value = names[n].value;
531
532         #if 0
533         ngx_log_error(NGX_LOG_ALERT, hinit->pool->log, 0,
534                     "wc2: \"%V\"", &next_name->key);
535         #endif
536     }
537
538     for (i = n + 1; i < nelts; i++) {
539         if (ngx_strncmp(names[n].key.data, names[i].key.data, len) != 0) {
540             break;
541         }
542
543         if (!dot
544             && names[i].key.len > len
545             && names[i].key.data[len] != '.')
546         {
547             break;
548         }
549
550         next_name = ngx_array_push(&next_names);
551         if (next_name == NULL) {
552             return NGX_ERROR;
553         }
554
555         next_name->key.len = names[i].key.len - dot_len;
556         next_name->key.data = names[i].key.data + dot_len;
557         next_name->key_hash = 0;
558         next_name->value = names[i].value;
559
560         #if 0
561         ngx_log_error(NGX_LOG_ALERT, hinit->pool->log, 0,
562                     "wc3: \"%V\"", &next_name->key);
563         #endif
564     }
565
566     if (next_names.nelts) {

```

```

567         h = *hinit;
568         h.hash = NULL;
569
570
571         if (ngx\_hash\_wildcard\_init(&h, (ngx\_hash\_key\_t *) next_names.elts,
572                                 next_names.nelts)
573             != NGX\_OK)
574         {
575             return NGX\_ERROR;
576         }
577
578         wdc = (ngx\_hash\_wildcard\_t *) h.hash;
579
580         if (names[n].key.len == len) {
581             wdc->value = names[n].value;
582         }
583
584         name->value = (void *) ((uintptr_t) wdc | (dot ? 3 : 2));
585
586     } else if (dot) {
587         name->value = (void *) ((uintptr_t) name->value | 1);
588     }
589 }
590
591 if (ngx\_hash\_init(hinit, (ngx\_hash\_key\_t *) curr_names.elts,
592                 curr_names.nelts)
593     != NGX\_OK)
594 {
595     return NGX\_ERROR;
596 }
597
598 return NGX\_OK;
599 }

```

```

600
601
602 ngx\_uint\_t
603 ngx\_hash\_key(u_char *data, size\_t len)
604 {
605     ngx\_uint\_t i, key;
606
607     key = 0;
608
609     for (i = 0; i < len; i++) {
610         key = ngx\_hash(key, data[i]);
611     }
612
613     return key;
614 }

```

```

615
616
617 ngx\_uint\_t
618 ngx\_hash\_key\_lc(u_char *data, size\_t len)
619 {
620     ngx\_uint\_t i, key;
621
622     key = 0;
623
624     for (i = 0; i < len; i++) {
625         key = ngx\_hash(key, ngx\_tolower(data[i]));
626     }
627
628     return key;
629 }

```

```

630
631
632 ngx\_uint\_t
633 ngx\_hash\_strlow(u_char *dst, u_char *src, size\_t n)
634 {
635     ngx\_uint\_t key;
636
637     key = 0;
638
639     while (n--) {
640         *dst = ngx\_tolower(*src);
641         key = ngx\_hash(key, *dst);
642         dst++;

```



```

643     src++;
644 }
645
646 return key;
647 }
648
649
650 ngx_int_t
651 ngx_hash_keys_array_init(ngx_hash_keys_arrays_t *ha, ngx_uint_t type)
652 {
653     ngx_uint_t asize;
654
655     if (type == NGX_HASH_SMALL) {
656         asize = 4;
657         ha->hsize = 107;
658
659     } else {
660         asize = NGX_HASH_LARGE_ASIZE;
661         ha->hsize = NGX_HASH_LARGE_HSIZE;
662     }
663
664     if (ngx_array_init(&ha->keys, ha->temp_pool, asize, sizeof(ngx_hash_key_t))
665         != NGX_OK)
666     {
667         return NGX_ERROR;
668     }
669
670     if (ngx_array_init(&ha->dns_wc_head, ha->temp_pool, asize,
671         sizeof(ngx_hash_key_t))
672         != NGX_OK)
673     {
674         return NGX_ERROR;
675     }
676
677     if (ngx_array_init(&ha->dns_wc_tail, ha->temp_pool, asize,
678         sizeof(ngx_hash_key_t))
679         != NGX_OK)
680     {
681         return NGX_ERROR;
682     }
683
684     ha->keys_hash = ngx_palloc(ha->temp_pool, sizeof(ngx_array_t) * ha->hsize);
685     if (ha->keys_hash == NULL) {
686         return NGX_ERROR;
687     }
688
689     ha->dns_wc_head_hash = ngx_palloc(ha->temp_pool,
690         sizeof(ngx_array_t) * ha->hsize);
691     if (ha->dns_wc_head_hash == NULL) {
692         return NGX_ERROR;
693     }
694
695     ha->dns_wc_tail_hash = ngx_palloc(ha->temp_pool,
696         sizeof(ngx_array_t) * ha->hsize);
697     if (ha->dns_wc_tail_hash == NULL) {
698         return NGX_ERROR;
699     }
700
701     return NGX_OK;
702 }
703
704
705 ngx_int_t
706 ngx_hash_add_key(ngx_hash_keys_arrays_t *ha, ngx_str_t *key, void *value,
707     ngx_uint_t flags)
708 {
709     size_t len;
710     u_char *p;
711     ngx_str_t *name;
712     ngx_uint_t i, k, n, skip, last;
713     ngx_array_t *keys, *hwc;
714     ngx_hash_key_t *hk;
715
716     last = key->len;
717
718     if (flags & NGX_HASH_WILDCARD_KEY) {

```

```

719
720 /*
721  * supported wildcards:
722  *   "*.example.com", ".example.com", and "www.example.*"
723  */
724
725 n = 0;
726
727 for (i = 0; i < key->len; i++) {
728     if (key->data[i] == '*') {
729         if (++n > 1) {
730             return NGX\_DECLINED;
731         }
732     }
733
734     if (key->data[i] == '.' && key->data[i + 1] == '.') {
735         return NGX\_DECLINED;
736     }
737 }
738
739 if (key->len > 1 && key->data[0] == '.') {
740     skip = 1;
741     goto wildcard;
742 }
743
744 if (key->len > 2) {
745     if (key->data[0] == '*' && key->data[1] == '.') {
746         skip = 2;
747         goto wildcard;
748     }
749
750     if (key->data[i - 2] == '.' && key->data[i - 1] == '*') {
751         skip = 0;
752         last -= 2;
753         goto wildcard;
754     }
755 }
756
757 if (n) {
758     return NGX\_DECLINED;
759 }
760
761 }
762
763 /* exact hash */
764
765 k = 0;
766
767 for (i = 0; i < last; i++) {
768     if (!(flags & NGX\_HASH\_READONLY\_KEY)) {
769         key->data[i] = ngx\_tolower(key->data[i]);
770     }
771     k = ngx\_hash(k, key->data[i]);
772 }
773
774 k %= ha->hsize;
775
776 /* check conflicts in exact hash */
777
778 name = ha->keys_hash[k].elts;
779
780 if (name) {
781     for (i = 0; i < ha->keys_hash[k].nelts; i++) {
782         if (last != name[i].len) {
783             continue;
784         }
785
786         if (ngx\_strncmp(key->data, name[i].data, last) == 0) {
787             return NGX\_BUSY;
788         }
789     }
790 }
791
792 } else {
793     if (ngx\_array\_init(&ha->keys_hash[k], ha->temp_pool, 4,
794         sizeof(ngx\_str\_t))

```

```

795         != NGX_OK)
796     {
797         return NGX_ERROR;
798     }
799 }
800
801 name = ngx_array_push(&ha->keys_hash[k]);
802 if (name == NULL) {
803     return NGX_ERROR;
804 }
805
806 *name = *key;
807
808 hk = ngx_array_push(&ha->keys);
809 if (hk == NULL) {
810     return NGX_ERROR;
811 }
812
813 hk->key = *key;
814 hk->key_hash = ngx_hash_key(key->data, last);
815 hk->value = value;
816
817 return NGX_OK;
818
819 wildcard:
820
821     /* wildcard hash */
822
823 k = ngx_hash_strlow(&key->data[skip], &key->data[skip], last - skip);
824
825 k %= ha->hsize;
826
827 if (skip == 1) {
828
829     /* check conflicts in exact hash for ".example.com" */
830
831     name = ha->keys_hash[k].elts;
832
833     if (name) {
834         len = last - skip;
835
836         for (i = 0; i < ha->keys_hash[k].nelts; i++) {
837             if (len != name[i].len) {
838                 continue;
839             }
840
841             if (ngx_strncmp(&key->data[1], name[i].data, len) == 0) {
842                 return NGX_BUSY;
843             }
844         }
845     }
846
847 } else {
848     if (ngx_array_init(&ha->keys_hash[k], ha->temp_pool, 4,
849         sizeof(ngx_str_t))
850         != NGX_OK)
851     {
852         return NGX_ERROR;
853     }
854 }
855
856 name = ngx_array_push(&ha->keys_hash[k]);
857 if (name == NULL) {
858     return NGX_ERROR;
859 }
860
861 name->len = last - 1;
862 name->data = ngx_pnalloc(ha->temp_pool, name->len);
863 if (name->data == NULL) {
864     return NGX_ERROR;
865 }
866
867 ngx_memcpy(name->data, &key->data[1], name->len);
868 }
869
870

```

```

871 if (skip) {
872
873     /*
874      * convert "*.example.com" to "com.example.\0"
875      * and ".example.com" to "com.example\0"
876      */
877
878     p = ngx_pnalloc(ha->temp_pool, last);
879     if (p == NULL) {
880         return NGX_ERROR;
881     }
882
883     len = 0;
884     n = 0;
885
886     for (i = last - 1; i; i--) {
887         if (key->data[i] == '.') {
888             ngx_memcpy(&p[n], &key->data[i + 1], len);
889             n += len;
890             p[n++] = '.';
891             len = 0;
892             continue;
893         }
894
895         len++;
896     }
897
898     if (len) {
899         ngx_memcpy(&p[n], &key->data[1], len);
900         n += len;
901     }
902
903     p[n] = '\0';
904
905     hwc = &ha->dns_wc_head;
906     keys = &ha->dns_wc_head_hash[k];
907
908 } else {
909
910     /* convert "www.example.*" to "www.example\0" */
911
912     last++;
913
914     p = ngx_pnalloc(ha->temp_pool, last);
915     if (p == NULL) {
916         return NGX_ERROR;
917     }
918
919     ngx_cpystn(p, key->data, last);
920
921     hwc = &ha->dns_wc_tail;
922     keys = &ha->dns_wc_tail_hash[k];
923 }
924
925 /* check conflicts in wildcard hash */
926
927 name = keys->elts;
928
929 if (name) {
930     len = last - skip;
931
932     for (i = 0; i < keys->nelts; i++) {
933         if (len != name[i].len) {
934             continue;
935         }
936
937         if (ngx_strncmp(key->data + skip, name[i].data, len) == 0) {
938             return NGX_BUSY;
939         }
940     }
941 }
942
943 } else {
944     if (ngx_array_init(keys, ha->temp_pool, 4, sizeof(ngx_str_t)) != NGX_OK)
945     {
946         return NGX_ERROR;

```

```
947     }
948 }
949
950 name = ngx_array_push(keys);
951 if (name == NULL) {
952     return NGX_ERROR;
953 }
954
955 name->len = last - skip;
956 name->data = ngx_pnalloc(ha->temp_pool, name->len);
957 if (name->data == NULL) {
958     return NGX_ERROR;
959 }
960
961 ngx_memcpy(name->data, key->data + skip, name->len);
962
963
964 /* add to wildcard hash */
965
966 hk = ngx_array_push(hwc);
967 if (hk == NULL) {
968     return NGX_ERROR;
969 }
970
971 hk->key.len = last - 1;
972 hk->key.data = p;
973 hk->key_hash = 0;
974 hk->value = value;
975
976 return NGX_OK;
977 }
```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_cycle.c - nginx-1.7.10

### Global variables defined

- [dumb](#)
- [ngx\\_cleaner\\_event](#)
- [ngx\\_core\\_tls\\_key](#)
- [ngx\\_cycle](#)
- [ngx\\_old\\_cycles](#)
- [ngx\\_quiet\\_mode](#)
- [ngx\\_temp\\_pool](#)
- [ngx\\_test\\_config](#)

### Functions defined

- [ngx\\_clean\\_old\\_cycles](#)
- [ngx\\_create\\_pidfile](#)
- [ngx\\_delete\\_pidfile](#)
- [ngx\\_destroy\\_cycle\\_pools](#)
- [ngx\\_init\\_cycle](#)
- [ngx\\_init\\_zone\\_pool](#)
- [ngx\\_reopen\\_files](#)
- [ngx\\_shared\\_memory\\_add](#)
- [ngx\\_signal\\_process](#)
- [ngx\\_test\\_lockfile](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 static void ngx_destroy_cycle_pools(ngx_conf_t *conf);
14 static ngx_int_t ngx_init_zone_pool(ngx_cycle_t *cycle,
15     ngx_shm_zone_t *shm_zone);
16 static ngx_int_t ngx_test_lockfile(u_char *file, ngx_log_t *log);
17 static void ngx_clean_old_cycles(ngx_event_t *ev);
18
19
20 volatile ngx_cycle_t *ngx_cycle;
```

```

21 ngx_array_t          ngx_old_cycles;
22
23 static ngx_pool_t    *ngx_temp_pool;
24 static ngx_event_t   ngx_cleaner_event;
25
26 ngx_uint_t           ngx_test_config;
27 ngx_uint_t           ngx_quiet_mode;
28
29 #if (NGX_THREADS)
30 ngx_tls_key_t        ngx_core_tls_key;
31 #endif
32
33
34 /* STUB NAME */
35 static ngx_connection_t dumb;
36 /* STUB */
37
38
39 ngx_cycle_t *
40 ngx_init_cycle(ngx_cycle_t *old_cycle)
41 {
42     void          *rv;
43     char          **senv, **env;
44     ngx_uint_t     i, n;
45     ngx_log_t     *log;
46     ngx_time_t    *tp;
47     ngx_conf_t    conf;
48     ngx_pool_t    *pool;
49     ngx_cycle_t   *cycle, **old;
50     ngx_shm_zone_t *shm_zone, *oshm_zone;
51     ngx_list_part_t *part, *opart;
52     ngx_open_file_t *file;
53     ngx_listening_t *ls, *nls;
54     ngx_core_conf_t *ccf, *old_ccf;
55     ngx_core_module_t *module;
56     char          hostname[NGX_MAXHOSTNAMELEN];
57
58     ngx_timezone_update();
59
60     /* force localtime update with a new timezone */
61
62     tp = ngx_timeofday();
63     tp->sec = 0;
64
65     ngx_time_update();
66
67
68     log = old_cycle->log;
69
70     pool = ngx_create_pool(NGX_CYCLE_POOL_SIZE, log);
71     if (pool == NULL) {
72         return NULL;
73     }
74     pool->log = log;
75
76     cycle = ngx_palloc(pool, sizeof(ngx_cycle_t));
77     if (cycle == NULL) {
78         ngx_destroy_pool(pool);
79         return NULL;
80     }
81
82     cycle->pool = pool;
83     cycle->log = log;
84     cycle->old_cycle = old_cycle;
85
86     cycle->conf_prefix.len = old_cycle->conf_prefix.len;
87     cycle->conf_prefix.data = ngx_pstrdup(pool, &old_cycle->conf_prefix);
88     if (cycle->conf_prefix.data == NULL) {
89         ngx_destroy_pool(pool);
90         return NULL;
91     }
92
93     cycle->prefix.len = old_cycle->prefix.len;
94     cycle->prefix.data = ngx_pstrdup(pool, &old_cycle->prefix);
95     if (cycle->prefix.data == NULL) {
96         ngx_destroy_pool(pool);

```

```

97     return NULL;
98 }
99
100 cycle->conf_file.len = old_cycle->conf_file.len;
101 cycle->conf_file.data = ngx_pnalloc(pool, old_cycle->conf_file.len + 1);
102 if (cycle->conf_file.data == NULL) {
103     ngx_destroy_pool(pool);
104     return NULL;
105 }
106 ngx_cpystn(cycle->conf_file.data, old_cycle->conf_file.data,
107           old_cycle->conf_file.len + 1);
108
109 cycle->conf_param.len = old_cycle->conf_param.len;
110 cycle->conf_param.data = ngx_pstrdup(pool, &old_cycle->conf_param);
111 if (cycle->conf_param.data == NULL) {
112     ngx_destroy_pool(pool);
113     return NULL;
114 }
115
116 n = old_cycle->paths.nelts ? old_cycle->paths.nelts : 10;
117
118 cycle->paths.elts = ngx_pcalloc(pool, n * sizeof(ngx_path_t *));
119 if (cycle->paths.elts == NULL) {
120     ngx_destroy_pool(pool);
121     return NULL;
122 }
123
124
125 cycle->paths.nelts = 0;
126 cycle->paths.size = sizeof(ngx_path_t *);
127 cycle->paths.nalloc = n;
128 cycle->paths.pool = pool;
129
130
131 if (old_cycle->open_files.part.nelts) {
132     n = old_cycle->open_files.part.nelts;
133     for (part = old_cycle->open_files.part.next; part; part = part->next) {
134         n += part->nelts;
135     }
136 } else {
137     n = 20;
138 }
139
140
141 if (ngx_list_init(&cycle->open_files, pool, n, sizeof(ngx_open_file_t))
142     != NGX_OK)
143 {
144     ngx_destroy_pool(pool);
145     return NULL;
146 }
147
148
149 if (old_cycle->shared_memory.part.nelts) {
150     n = old_cycle->shared_memory.part.nelts;
151     for (part = old_cycle->shared_memory.part.next; part; part = part->next)
152     {
153         n += part->nelts;
154     }
155 } else {
156     n = 1;
157 }
158
159
160 if (ngx_list_init(&cycle->shared_memory, pool, n, sizeof(ngx_shm_zone_t))
161     != NGX_OK)
162 {
163     ngx_destroy_pool(pool);
164     return NULL;
165 }
166
167 n = old_cycle->listening.nelts ? old_cycle->listening.nelts : 10;
168
169 cycle->listening.elts = ngx_pcalloc(pool, n * sizeof(ngx_listening_t));
170 if (cycle->listening.elts == NULL) {
171     ngx_destroy_pool(pool);
172     return NULL;

```



```

173 }
174
175 cycle->listening.nelts = 0;
176 cycle->listening.size = sizeof(ngx\_listening\_t);
177 cycle->listening.nalloc = n;
178 cycle->listening.pool = pool;
179
180
181 ngx\_queue\_init(&cycle->reusable_connections_queue);
182
183
184 cycle->conf_ctx = ngx\_palloc(pool, ngx\_max\_module * sizeof(void *));
185 if (cycle->conf_ctx == NULL) {
186     ngx\_destroy\_pool(pool);
187     return NULL;
188 }
189
190
191 if (gethostname(hostname, NGX\_MAXHOSTNAMELEN) == -1) {
192     ngx\_log\_error(NGX\_LOG\_EMERG, log, ngx\_errno, "gethostname() failed");
193     ngx\_destroy\_pool(pool);
194     return NULL;
195 }
196
197 /* on Linux gethostname() silently truncates name that does not fit */
198
199 hostname[NGX\_MAXHOSTNAMELEN - 1] = '\0';
200 cycle->hostname.len = ngx\_strlen(hostname);
201
202 cycle->hostname.data = ngx\_palloc(pool, cycle->hostname.len);
203 if (cycle->hostname.data == NULL) {
204     ngx\_destroy\_pool(pool);
205     return NULL;
206 }
207
208 ngx\_strlow(cycle->hostname.data, (u_char *) hostname, cycle->hostname.len);
209
210
211 for (i = 0; ngx_modules[i]; i++) {
212     if (ngx_modules[i]->type != NGX\_CORE\_MODULE) {
213         continue;
214     }
215
216     module = ngx_modules[i]->ctx;
217
218     if (module->create_conf) {
219         rv = module->create_conf(cycle);
220         if (rv == NULL) {
221             ngx\_destroy\_pool(pool);
222             return NULL;
223         }
224         cycle->conf_ctx[ngx_modules[i]->index] = rv;
225     }
226 }
227
228
229 senv = environ;
230
231
232 ngx\_memzero(&conf, sizeof(ngx\_conf\_t));
233 /* STUB: init array ? */
234 conf.args = ngx\_array\_create(pool, 10, sizeof(ngx\_str\_t));
235 if (conf.args == NULL) {
236     ngx\_destroy\_pool(pool);
237     return NULL;
238 }
239
240 conf.temp_pool = ngx\_create\_pool(NGX\_CYCLE\_POOL\_SIZE, log);
241 if (conf.temp_pool == NULL) {
242     ngx\_destroy\_pool(pool);
243     return NULL;
244 }
245
246
247 conf.ctx = cycle->conf_ctx;
248 conf.cycle = cycle;

```

```

249     conf.pool = pool;
250     conf.log = log;
251     conf.module_type = NGX\_CORE\_MODULE;
252     conf.cmd_type = NGX\_MAIN\_CONF;
253
254     #if 0
255         log->log_level = NGX\_LOG\_DEBUG\_ALL;
256     #endif
257
258     if (ngx\_conf\_param(&conf) != NGX\_CONF\_OK) {
259         environ = serv;
260         ngx\_destroy\_cycle\_pools(&conf);
261         return NULL;
262     }
263
264     if (ngx\_conf\_parse(&conf, &cycle->conf_file) != NGX\_CONF\_OK) {
265         environ = serv;
266         ngx\_destroy\_cycle\_pools(&conf);
267         return NULL;
268     }
269
270     if (ngx\_test\_config && !ngx\_quiet\_mode) {
271         ngx\_log\_stderr(0, "the configuration file %s syntax is ok",
272             cycle->conf_file.data);
273     }
274
275     for (i = 0; ngx_modules[i]; i++) {
276         if (ngx_modules[i]->type != NGX\_CORE\_MODULE) {
277             continue;
278         }
279
280         module = ngx_modules[i]->ctx;
281
282         if (module->init_conf) {
283             if (module->init_conf(cycle, cycle->conf_ctx[ngx_modules[i]->index])
284                 == NGX\_CONF\_ERROR)
285                 {
286                     environ = serv;
287                     ngx\_destroy\_cycle\_pools(&conf);
288                     return NULL;
289                 }
290         }
291     }
292
293     if (ngx\_process == NGX\_PROCESS\_SIGNALLER) {
294         return cycle;
295     }
296
297     ccf = (ngx\_core\_conf\_t *) ngx\_get\_conf(cycle->conf_ctx, ngx\_core\_module);
298
299     if (ngx\_test\_config) {
300
301         if (ngx\_create\_pidfile(&ccf->pid, log) != NGX\_OK) {
302             goto failed;
303         }
304
305     } else if (!ngx\_is\_init\_cycle(old_cycle)) {
306
307         /*
308          * we do not create the pid file in the first ngx\_init\_cycle\(\) call
309          * because we need to write the demonized process pid
310          */
311
312         old_ccf = (ngx\_core\_conf\_t *) ngx\_get\_conf(old_cycle->conf_ctx,
313             ngx\_core\_module);
314         if (ccf->pid.len != old_ccf->pid.len
315             || ngx\_strcmp(ccf->pid.data, old_ccf->pid.data) != 0)
316             {
317                 /* new pid file name */
318
319                 if (ngx\_create\_pidfile(&ccf->pid, log) != NGX\_OK) {
320                     goto failed;
321                 }
322
323                 ngx\_delete\_pidfile(old_cycle);
324             }

```

```

325 }
326
327
328 if (ngx_test_lockfile(cycle->lock_file.data, log) != NGX_OK) {
329     goto failed;
330 }
331
332
333 if (ngx_create_paths(cycle, ccf->user) != NGX_OK) {
334     goto failed;
335 }
336
337
338 if (ngx_log_open_default(cycle) != NGX_OK) {
339     goto failed;
340 }
341
342 /* open the new files */
343
344 part = &cycle->open_files.part;
345 file = part->elts;
346
347 for (i = 0; /* void */ ; i++) {
348
349     if (i >= part->nelts) {
350         if (part->next == NULL) {
351             break;
352         }
353         part = part->next;
354         file = part->elts;
355         i = 0;
356     }
357
358     if (file[i].name.len == 0) {
359         continue;
360     }
361
362     file[i].fd = ngx_open_file(file[i].name.data,
363                               NGX_FILE_APPEND,
364                               NGX_FILE_CREATE_OR_OPEN,
365                               NGX_FILE_DEFAULT_ACCESS);
366
367     ngx_log_debug3(NGX_LOG_DEBUG_CORE, log, 0,
368                  "log: %p %d \"%s\"",
369                  &file[i], file[i].fd, file[i].name.data);
370
371     if (file[i].fd == NGX_INVALID_FILE) {
372         ngx_log_error(NGX_LOG_EMERG, log, ngx_errno,
373                      ngx_open_file_n " \"%s\" failed",
374                      file[i].name.data);
375         goto failed;
376     }
377
378     #if !(NGX_WIN32)
379     if (fcntl(file[i].fd, F_SETFD, FD_CLOEXEC) == -1) {
380         ngx_log_error(NGX_LOG_EMERG, log, ngx_errno,
381                      "fcntl(FD_CLOEXEC) \"%s\" failed",
382                      file[i].name.data);
383         goto failed;
384     }
385     #endif
386 }
387
388 cycle->log = &cycle->new_log;
389 pool->log = &cycle->new_log;
390
391
392 /* create shared memory */
393
394 part = &cycle->shared_memory.part;
395 shm_zone = part->elts;
396
397 for (i = 0; /* void */ ; i++) {
398
399     if (i >= part->nelts) {
400         if (part->next == NULL) {

```

```

401         break;
402     }
403     part = part->next;
404     shm_zone = part->elts;
405     i = 0;
406 }
407
408 if (shm_zone[i].shm.size == 0) {
409     ngx_log_error(NGX_LOG_EMERG, log, 0,
410                 "zero size shared memory zone \"%V\"",
411                 &shm_zone[i].shm.name);
412     goto failed;
413 }
414
415 shm_zone[i].shm.log = cycle->log;
416
417 opart = &old_cycle->shared_memory.part;
418 oshm_zone = opart->elts;
419
420 for (n = 0; /* void */ ; n++) {
421
422     if (n >= opart->nelts) {
423         if (opart->next == NULL) {
424             break;
425         }
426         opart = opart->next;
427         oshm_zone = opart->elts;
428         n = 0;
429     }
430
431     if (shm_zone[i].shm.name.len != oshm_zone[n].shm.name.len) {
432         continue;
433     }
434
435     if (ngx_strncmp(shm_zone[i].shm.name.data,
436                   oshm_zone[n].shm.name.data,
437                   shm_zone[i].shm.name.len)
438         != 0)
439     {
440         continue;
441     }
442
443     if (shm_zone[i].tag == oshm_zone[n].tag
444         && shm_zone[i].shm.size == oshm_zone[n].shm.size)
445     {
446         shm_zone[i].shm.addr = oshm_zone[n].shm.addr;
447
448         if (shm_zone[i].init(&shm_zone[i], oshm_zone[n].data)
449             != NGX_OK)
450         {
451             goto failed;
452         }
453
454         goto shm_zone_found;
455     }
456
457     ngx_shm_free(&oshm_zone[n].shm);
458
459     break;
460 }
461
462 if (ngx_shm_alloc(&shm_zone[i].shm) != NGX_OK) {
463     goto failed;
464 }
465
466 if (ngx_init_zone_pool(cycle, &shm_zone[i]) != NGX_OK) {
467     goto failed;
468 }
469
470 if (shm_zone[i].init(&shm_zone[i], NULL) != NGX_OK) {
471     goto failed;
472 }
473
474 shm_zone_found:
475
476     continue;

```

```

477     }
478
479     /* handle the listening sockets */
480
481     if (old_cycle->listening.nelts) {
482         ls = old_cycle->listening.elts;
483         for (i = 0; i < old_cycle->listening.nelts; i++) {
484             ls[i].remain = 0;
485         }
486
487         nls = cycle->listening.elts;
488         for (n = 0; n < cycle->listening.nelts; n++) {
489
490             for (i = 0; i < old_cycle->listening.nelts; i++) {
491                 if (ls[i].ignore) {
492                     continue;
493                 }
494
495                 if (ngx_cmp_sockaddr(nls[n].sockaddr, nls[n].socklen,
496                                     ls[i].sockaddr, ls[i].socklen, 1)
497                     == NGX_OK)
498                 {
499                     nls[n].fd = ls[i].fd;
500                     nls[n].previous = &ls[i];
501                     ls[i].remain = 1;
502
503                     if (ls[i].backlog != nls[n].backlog) {
504                         nls[n].listen = 1;
505                     }
506                 }
507
508                 #if (NGX_HAVE_DEFERRED_ACCEPT && defined SO_ACCEPTFILTER)
509
510                     /*
511                      * FreeBSD, except the most recent versions,
512                      * could not remove accept filter
513                      */
514                     nls[n].deferred_accept = ls[i].deferred_accept;
515
516                     if (ls[i].accept_filter && nls[n].accept_filter) {
517                         if (ngx_strcmp(ls[i].accept_filter,
518                                         nls[n].accept_filter)
519                             != 0)
520                         {
521                             nls[n].delete_deferred = 1;
522                             nls[n].add_deferred = 1;
523                         }
524
525                     } else if (ls[i].accept_filter) {
526                         nls[n].delete_deferred = 1;
527
528                     } else if (nls[n].accept_filter) {
529                         nls[n].add_deferred = 1;
530                     }
531                 #endif
532
533                 #if (NGX_HAVE_DEFERRED_ACCEPT && defined TCP_DEFER_ACCEPT)
534
535                     if (ls[i].deferred_accept && !nls[n].deferred_accept) {
536                         nls[n].delete_deferred = 1;
537
538                     } else if (ls[i].deferred_accept != nls[n].deferred_accept)
539                     {
540                         nls[n].add_deferred = 1;
541                     }
542                 #endif
543
544                 break;
545             }
546
547             if (nls[n].fd == (ngx_socket_t) -1) {
548                 nls[n].open = 1;
549                 #if (NGX_HAVE_DEFERRED_ACCEPT && defined SO_ACCEPTFILTER)
550                     if (nls[n].accept_filter) {
551                         nls[n].add_deferred = 1;
552                     }

```

```

553 #endif
554 #if (NGX_HAVE_DEFERRED_ACCEPT && defined TCP_DEFER_ACCEPT)
555     if (nls[n].deferred_accept) {
556         nls[n].add_deferred = 1;
557     }
558 #endif
559     }
560 }
561
562 } else {
563     ls = cycle->listening.elts;
564     for (i = 0; i < cycle->listening.nelts; i++) {
565         ls[i].open = 1;
566 #if (NGX_HAVE_DEFERRED_ACCEPT && defined SO_ACCEPTFILTER)
567         if (ls[i].accept_filter) {
568             ls[i].add_deferred = 1;
569         }
570 #endif
571 #if (NGX_HAVE_DEFERRED_ACCEPT && defined TCP_DEFER_ACCEPT)
572         if (ls[i].deferred_accept) {
573             ls[i].add_deferred = 1;
574         }
575 #endif
576     }
577 }
578
579 if (ngx_open_listening_sockets(cycle) != NGX_OK) {
580     goto failed;
581 }
582
583 if (!ngx_test_config) {
584     ngx_configure_listening_sockets(cycle);
585 }
586
587 /* commit the new cycle configuration */
588
589 if (!ngx_use_stderr) {
590     (void) ngx_log_redirect_stderr(cycle);
591 }
592
593 pool->log = cycle->log;
594
595 for (i = 0; ngx_modules[i]; i++) {
596     if (ngx_modules[i]->init_module) {
597         if (ngx_modules[i]->init_module(cycle) != NGX_OK) {
598             /* fatal */
599             exit(1);
600         }
601     }
602 }
603
604
605 /* close and delete stuff that lefts from an old cycle */
606
607 /* free the unnecessary shared memory */
608
609 opart = &old_cycle->shared_memory.part;
610 oshm_zone = opart->elts;
611
612 for (i = 0; /* void */; i++) {
613
614     if (i >= opart->nelts) {
615         if (opart->next == NULL) {
616             goto old_shm_zone_done;
617         }
618         opart = opart->next;
619         oshm_zone = opart->elts;
620         i = 0;
621     }
622
623     part = &cycle->shared_memory.part;
624     shm_zone = part->elts;
625
626     for (n = 0; /* void */; n++) {
627
628

```

```

629     if (n >= part->nelts) {
630         if (part->next == NULL) {
631             break;
632         }
633         part = part->next;
634         shm_zone = part->elts;
635         n = 0;
636     }
637
638     if (oshm_zone[i].shm.name.len == shm_zone[n].shm.name.len
639         && ngx_strncmp(oshm_zone[i].shm.name.data,
640                     shm_zone[n].shm.name.data,
641                     oshm_zone[i].shm.name.len)
642         == 0)
643     {
644         goto live_shm_zone;
645     }
646 }
647
648 ngx_shm_free(&oshm_zone[i].shm);
649
650 live_shm_zone:
651     continue;
652 }
653
654 old_shm_zone_done:
655
656
657     /* close the unnecessary listening sockets */
658
659     ls = old_cycle->listening.elts;
660     for (i = 0; i < old_cycle->listening.nelts; i++) {
661
662         if (ls[i].remain || ls[i].fd == (ngx_socket_t) -1) {
663             continue;
664         }
665
666         if (ngx_close_socket(ls[i].fd) == -1) {
667             ngx_log_error(NGX_LOG_EMERG, log, ngx_socket_errno,
668                 ngx_close_socket_n " listening socket on %V failed",
669                 &ls[i].addr_text);
670         }
671     }
672
673 #if (NGX_HAVE_UNIX_DOMAIN)
674
675     if (ls[i].sockaddr->sa_family == AF_UNIX) {
676         u_char *name;
677
678         name = ls[i].addr_text.data + sizeof("unix:") - 1;
679
680         ngx_log_error(NGX_LOG_WARN, cycle->log, 0,
681             "deleting socket %s", name);
682
683         if (ngx_delete_file(name) == NGX_FILE_ERROR) {
684             ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_socket_errno,
685                 ngx_delete_file_n " %s failed", name);
686         }
687     }
688
689 #endif
690 }
691
692
693     /* close the unnecessary open files */
694
695     part = &old_cycle->open_files.part;
696     file = part->elts;
697
698     for (i = 0; /* void */ ; i++) {
699
700         if (i >= part->nelts) {
701             if (part->next == NULL) {
702                 break;
703             }
704             part = part->next;

```

```

705     file = part->elts;
706     i = 0;
707 }
708
709 if (file[i].fd == NGX\_INVALID\_FILE || file[i].fd == ngx\_stderr) {
710     continue;
711 }
712
713 if (ngx\_close\_file(file[i].fd) == NGX\_FILE\_ERROR) {
714     ngx\_log\_error(NGX\_LOG\_EMERG, log, ngx\_errno,
715                 ngx\_close\_file n " \"%s\" failed",
716                 file[i].name.data);
717 }
718 }
719
720 ngx\_destroy\_pool(conf.temp_pool);
721
722 if (ngx\_process == NGX\_PROCESS\_MASTER || ngx\_is\_init\_cycle(old_cycle)) {
723
724     /*
725      * perl\_destruct\(\) frees environ, if it is not the same as it was at
726      * perl\_construct\(\) time, therefore we save the previous cycle
727      * environment before ngx\_conf\_parse\(\) where it will be changed.
728      */
729
730     env = environ;
731     environ = senv;
732
733     ngx\_destroy\_pool(old_cycle->pool);
734     cycle->old_cycle = NULL;
735
736     environ = env;
737
738     return cycle;
739 }
740
741
742 if (ngx\_temp\_pool == NULL) {
743     ngx\_temp\_pool = ngx\_create\_pool(128, cycle->log);
744     if (ngx\_temp\_pool == NULL) {
745         ngx\_log\_error(NGX\_LOG\_EMERG, cycle->log, 0,
746                     "could not create ngx\_temp\_pool");
747         exit(1);
748     }
749
750     n = 10;
751     ngx\_old\_cycles.elts = ngx\_palloc(ngx\_temp\_pool,
752                                   n * sizeof(ngx\_cycle\_t *));
753     if (ngx\_old\_cycles.elts == NULL) {
754         exit(1);
755     }
756     ngx\_old\_cycles.nelts = 0;
757     ngx\_old\_cycles.size = sizeof(ngx\_cycle\_t *);
758     ngx\_old\_cycles.nalloc = n;
759     ngx\_old\_cycles.pool = ngx\_temp\_pool;
760
761     ngx\_cleaner\_event.handler = ngx\_clean\_old\_cycles;
762     ngx\_cleaner\_event.log = cycle->log;
763     ngx\_cleaner\_event.data = &dumb;
764     dumb.fd = (ngx\_socket\_t) -1;
765 }
766
767 ngx\_temp\_pool->log = cycle->log;
768
769 old = ngx\_array\_push(&ngx\_old\_cycles);
770 if (old == NULL) {
771     exit(1);
772 }
773 *old = old_cycle;
774
775 if (!ngx\_cleaner\_event.timer_set) {
776     ngx\_add\_timer(&ngx\_cleaner\_event, 30000);
777     ngx\_cleaner\_event.timer_set = 1;
778 }
779
780 return cycle;

```



```

781
782
783 failed:
784
785     if (!ngx_is_init_cycle(old_cycle)) {
786         old_ccf = (ngx_core_conf_t *) ngx_get_conf(old_cycle->conf_ctx,
787                                                     ngx_core_module);
788         if (old_ccf->environment) {
789             environ = old_ccf->environment;
790         }
791     }
792
793     /* rollback the new cycle configuration */
794
795     part = &cycle->open_files.part;
796     file = part->elts;
797
798     for (i = 0; /* void */ ; i++) {
799
800         if (i >= part->nelts) {
801             if (part->next == NULL) {
802                 break;
803             }
804             part = part->next;
805             file = part->elts;
806             i = 0;
807         }
808
809         if (file[i].fd == NGX_INVALID_FILE || file[i].fd == ngx_stderr) {
810             continue;
811         }
812
813         if (ngx_close_file(file[i].fd) == NGX_FILE_ERROR) {
814             ngx_log_error(NGX_LOG_EMERG, log, ngx_errno,
815                           ngx_close_file_n " \"%s\" failed",
816                           file[i].name.data);
817         }
818     }
819
820     if (ngx_test_config) {
821         ngx_destroy_cycle_pools(&conf);
822         return NULL;
823     }
824
825     ls = cycle->listening.elts;
826     for (i = 0; i < cycle->listening.nelts; i++) {
827         if (ls[i].fd == (ngx_socket_t) -1 || !ls[i].open) {
828             continue;
829         }
830
831         if (ngx_close_socket(ls[i].fd) == -1) {
832             ngx_log_error(NGX_LOG_EMERG, log, ngx_socket_errno,
833                           ngx_close_socket_n " %V failed",
834                           &ls[i].addr_text);
835         }
836     }
837
838     ngx_destroy_cycle_pools(&conf);
839
840     return NULL;
841 }
842
843
844 static void
845 ngx_destroy_cycle_pools(ngx_conf_t *conf)
846 {
847     ngx_destroy_pool(conf->temp_pool);
848     ngx_destroy_pool(conf->pool);
849 }
850
851
852 static ngx_int_t
853 ngx_init_zone_pool(ngx_cycle_t *cycle, ngx_shm_zone_t *zn)
854 {
855     u_char          *file;
856     ngx_slab_pool_t *sp;

```

```

857 sp = (ngx\_slab\_pool\_t *) zn->shm.addr;
858
859
860 if (zn->shm.exists) {
861     if (sp == sp->addr) {
862         return NGX\_OK;
863     }
864
865     ngx\_log\_error(NGX\_LOG\_EMERG, cycle->log, 0,
866         "shared zone \"%V\" has no equal addresses: %p vs %p",
867         &zn->shm.name, sp->addr, sp);
868     return NGX\_ERROR;
869 }
870
871
872 sp->end = zn->shm.addr + zn->shm.size;
873 sp->min_shift = 3;
874 sp->addr = zn->shm.addr;
875
876 #if (NGX\_HAVE\_ATOMIC\_OPS)
877     file = NULL;
878
879 #else
880     file = ngx\_pnalloc(cycle->pool, cycle->lock_file.len + zn->shm.name.len);
881     if (file == NULL) {
882         return NGX\_ERROR;
883     }
884
885     (void) ngx\_sprintf(file, "%V%Z", &cycle->lock_file, &zn->shm.name);
886
887 #endif
888
889     if (ngx\_shmtx\_create(&sp->mutex, &sp->lock, file) != NGX\_OK) {
890         return NGX\_ERROR;
891     }
892
893     ngx\_slab\_init(sp);
894
895     return NGX\_OK;
896 }
897
898
899
900
901 ngx\_int\_t
902 ngx\_create\_pidfile(ngx\_str\_t *name, ngx\_log\_t *log)
903 {
904     size\_t len;
905     ngx\_uint\_t create;
906     ngx\_file\_t file;
907     u\_char pid[NGX\_INT64\_LEN + 2];
908
909     if (ngx\_process > NGX\_PROCESS\_MASTER) {
910         return NGX\_OK;
911     }
912
913     ngx\_memzero(&file, sizeof(ngx\_file\_t));
914
915     file.name = *name;
916     file.log = log;
917
918     create = ngx\_test\_config ? NGX\_FILE\_CREATE\_OR\_OPEN : NGX\_FILE\_TRUNCATE;
919
920     file.fd = ngx\_open\_file(file.name.data, NGX\_FILE\_RDWR,
921         create, NGX\_FILE\_DEFAULT\_ACCESS);
922
923     if (file.fd == NGX\_INVALID\_FILE) {
924         ngx\_log\_error(NGX\_LOG\_EMERG, log, ngx\_errno,
925             ngx\_open\_file\_n " \"%s\" failed", file.name.data);
926         return NGX\_ERROR;
927     }
928
929     if (!ngx\_test\_config) {
930         len = ngx\_snprintf(pid, NGX\_INT64\_LEN + 2, "%P%N", ngx\_pid) - pid;
931
932         if (ngx\_write\_file(&file, pid, len, 0) == NGX\_ERROR) {

```

```

933     return NGX\_ERROR;
934 }
935 }
936
937 if (ngx\_close\_file(file.fd) == NGX\_FILE\_ERROR) {
938     ngx\_log\_error(NGX\_LOG\_ALERT, log, ngx\_errno,
939                 ngx\_close\_file\_n " \"%s\" failed", file.name.data);
940 }
941
942 return NGX\_OK;
943 }
944
945
946 void
947 ngx\_delete\_pidfile(ngx\_cycle\_t *cycle)
948 {
949     u_char          *name;
950     ngx\_core\_conf\_t *ccf;
951
952     ccf = (ngx\_core\_conf\_t *) ngx\_get\_conf(cycle->conf_ctx, ngx\_core\_module);
953
954     name = ngx\_new\_binary ? ccf->oldpid.data : ccf->pid.data;
955
956     if (ngx\_delete\_file(name) == NGX\_FILE\_ERROR) {
957         ngx\_log\_error(NGX\_LOG\_ALERT, cycle->log, ngx\_errno,
958                 ngx\_delete\_file\_n " \"%s\" failed", name);
959     }
960 }
961
962
963 ngx\_int\_t
964 ngx\_signal\_process(ngx\_cycle\_t *cycle, char *sig)
965 {
966     ssize_t         n;
967     ngx\_int\_t       pid;
968     ngx\_file\_t      file;
969     ngx\_core\_conf\_t *ccf;
970     u_char          buf[NGX\_INT64\_LEN + 2];
971
972     ngx\_log\_error(NGX\_LOG\_NOTICE, cycle->log, 0, "signal process started");
973
974     ccf = (ngx\_core\_conf\_t *) ngx\_get\_conf(cycle->conf_ctx, ngx\_core\_module);
975
976     ngx\_memzero(&file, sizeof(ngx\_file\_t));
977
978     file.name = ccf->pid;
979     file.log = cycle->log;
980
981     file.fd = ngx\_open\_file(file.name.data, NGX\_FILE\_RDONLY,
982                             NGX\_FILE\_OPEN, NGX\_FILE\_DEFAULT\_ACCESS);
983
984     if (file.fd == NGX\_INVALID\_FILE) {
985         ngx\_log\_error(NGX\_LOG\_ERR, cycle->log, ngx\_errno,
986                 ngx\_open\_file\_n " \"%s\" failed", file.name.data);
987         return 1;
988     }
989
990     n = ngx\_read\_file(&file, buf, NGX\_INT64\_LEN + 2, 0);
991
992     if (ngx\_close\_file(file.fd) == NGX\_FILE\_ERROR) {
993         ngx\_log\_error(NGX\_LOG\_ALERT, cycle->log, ngx\_errno,
994                 ngx\_close\_file\_n " \"%s\" failed", file.name.data);
995     }
996
997     if (n == NGX\_ERROR) {
998         return 1;
999     }
1000
1001     while (n-- && (buf[n] == CR || buf[n] == LF)) { /* void */ }
1002
1003     pid = ngx\_atoi(buf, ++n);
1004
1005     if (pid == NGX\_ERROR) {
1006         ngx\_log\_error(NGX\_LOG\_ERR, cycle->log, 0,
1007                 "invalid PID number \"%s\" in \"%s\"",
1008                 n, buf, file.name.data);

```

```

1009     return 1;
1010 }
1011
1012 return ngx_os_signal_process(cycle, sig, pid);
1013
1014 }
1015
1016
1017 static ngx_int_t
1018 ngx_test_lockfile(u_char *file, ngx_log_t *log)
1019 {
1020     #if !(NGX_HAVE_ATOMIC_OPS)
1021         ngx_fd_t  fd;
1022
1023         fd = ngx_open_file(file, NGX_FILE_RDWR, NGX_FILE_CREATE_OR_OPEN,
1024             NGX_FILE_DEFAULT_ACCESS);
1025
1026         if (fd == NGX_INVALID_FILE) {
1027             ngx_log_error(NGX_LOG_EMERG, log, ngx_errno,
1028                 ngx_open_file_n " \"%s\" failed", file);
1029             return NGX_ERROR;
1030         }
1031
1032         if (ngx_close_file(fd) == NGX_FILE_ERROR) {
1033             ngx_log_error(NGX_LOG_ALERT, log, ngx_errno,
1034                 ngx_close_file_n " \"%s\" failed", file);
1035         }
1036
1037         if (ngx_delete_file(file) == NGX_FILE_ERROR) {
1038             ngx_log_error(NGX_LOG_ALERT, log, ngx_errno,
1039                 ngx_delete_file_n " \"%s\" failed", file);
1040         }
1041     #endif
1042
1043     return NGX_OK;
1044 }
1045
1046
1047
1048 void
1049 ngx_reopen_files(ngx_cycle_t *cycle, ngx_uid_t user)
1050 {
1051     ngx_fd_t      fd;
1052     ngx_uint_t    i;
1053     ngx_list_part_t *part;
1054     ngx_open_file_t *file;
1055
1056     part = &cycle->open_files.part;
1057     file = part->elts;
1058
1059     for (i = 0; /* void */ ; i++) {
1060
1061         if (i >= part->nelts) {
1062             if (part->next == NULL) {
1063                 break;
1064             }
1065             part = part->next;
1066             file = part->elts;
1067             i = 0;
1068         }
1069
1070         if (file[i].name.len == 0) {
1071             continue;
1072         }
1073
1074         if (file[i].flush) {
1075             file[i].flush(&file[i], cycle->log);
1076         }
1077
1078         fd = ngx_open_file(file[i].name.data, NGX_FILE_APPEND,
1079             NGX_FILE_CREATE_OR_OPEN, NGX_FILE_DEFAULT_ACCESS);
1080
1081         ngx_log_debug3(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
1082             "reopen file \"%s\", old:%d new:%d",
1083             file[i].name.data, file[i].fd, fd);
1084

```

```

1085     if (fd == NGX_INVALID_FILE) {
1086         ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
1087             ngx_open_file_n " \"%s\" failed", file[i].name.data);
1088     }
1089     continue;
1090 }
1091 #if !(NGX_WIN32)
1092     if (user != (ngx_uid_t) NGX_CONF_UNSET_UINT) {
1093         ngx_file_info_t fi;
1094
1095         if (ngx_file_info((const char *) file[i].name.data, &fi)
1096             == NGX_FILE_ERROR)
1097         {
1098             ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
1099                 ngx_file_info_n " \"%s\" failed",
1100                 file[i].name.data);
1101
1102             if (ngx_close_file(fd) == NGX_FILE_ERROR) {
1103                 ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
1104                     ngx_close_file_n " \"%s\" failed",
1105                     file[i].name.data);
1106             }
1107
1108             continue;
1109         }
1110
1111         if (fi.st_uid != user) {
1112             if (chown((const char *) file[i].name.data, user, -1) == -1) {
1113                 ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
1114                     "chown(\"%s\", %d) failed",
1115                     file[i].name.data, user);
1116
1117                 if (ngx_close_file(fd) == NGX_FILE_ERROR) {
1118                     ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
1119                         ngx_close_file_n " \"%s\" failed",
1120                         file[i].name.data);
1121                 }
1122
1123                 continue;
1124             }
1125         }
1126
1127         if ((fi.st_mode & (S_IRUSR|S_IWUSR)) != (S_IRUSR|S_IWUSR)) {
1128             fi.st_mode |= (S_IRUSR|S_IWUSR);
1129
1130             if (chmod((const char *) file[i].name.data, fi.st_mode) == -1) {
1131                 ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
1132                     "chmod() \"%s\" failed", file[i].name.data);
1133
1134                 if (ngx_close_file(fd) == NGX_FILE_ERROR) {
1135                     ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
1136                         ngx_close_file_n " \"%s\" failed",
1137                         file[i].name.data);
1138                 }
1139
1140                 continue;
1141             }
1142         }
1143     }
1144 }
1145
1146     if (fcntl(fd, F_SETFD, FD_CLOEXEC) == -1) {
1147         ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
1148             "fcntl(FD_CLOEXEC) \"%s\" failed",
1149             file[i].name.data);
1150
1151         if (ngx_close_file(fd) == NGX_FILE_ERROR) {
1152             ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
1153                 ngx_close_file_n " \"%s\" failed",
1154                 file[i].name.data);
1155         }
1156
1157         continue;
1158     }
1159 #endif
1160

```

```

1161     if (ngx_close_file(file[i].fd) == NGX_FILE_ERROR) {
1162         ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
1163             ngx_close_file_n "%s\" failed",
1164             file[i].name.data);
1165     }
1166
1167     file[i].fd = fd;
1168 }
1169
1170 (void) ngx_log_redirect_stderr(cycle);
1171 }
1172
1173
1174 ngx_shm_zone_t *
1175 ngx_shared_memory_add(ngx_conf_t *cf, ngx_str_t *name, size_t size, void *tag)
1176 {
1177     ngx_uint_t     i;
1178     ngx_shm_zone_t *shm_zone;
1179     ngx_list_part_t *part;
1180
1181     part = &cf->cycle->shared_memory.part;
1182     shm_zone = part->elts;
1183
1184     for (i = 0; /* void */ ; i++) {
1185
1186         if (i >= part->nelts) {
1187             if (part->next == NULL) {
1188                 break;
1189             }
1190             part = part->next;
1191             shm_zone = part->elts;
1192             i = 0;
1193         }
1194
1195         if (name->len != shm_zone[i].shm.name.len) {
1196             continue;
1197         }
1198
1199         if (ngx_strncmp(name->data, shm_zone[i].shm.name.data, name->len)
1200             != 0)
1201         {
1202             continue;
1203         }
1204
1205         if (tag != shm_zone[i].tag) {
1206             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1207                 "the shared memory zone \"%V\" is "
1208                 "already declared for a different use",
1209                 &shm_zone[i].shm.name);
1210             return NULL;
1211         }
1212
1213         if (size && size != shm_zone[i].shm.size) {
1214             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1215                 "the size %uz of shared memory zone \"%V\" "
1216                 "conflicts with already declared size %uz",
1217                 size, &shm_zone[i].shm.name, shm_zone[i].shm.size);
1218             return NULL;
1219         }
1220
1221         return &shm_zone[i];
1222     }
1223
1224     shm_zone = ngx_list_push(&cf->cycle->shared_memory);
1225
1226     if (shm_zone == NULL) {
1227         return NULL;
1228     }
1229
1230     shm_zone->data = NULL;
1231     shm_zone->shm.log = cf->cycle->log;
1232     shm_zone->shm.size = size;
1233     shm_zone->shm.name = *name;
1234     shm_zone->shm.exists = 0;
1235     shm_zone->init = NULL;
1236     shm_zone->tag = tag;

```

```

1237     return shm_zone;
1238 }
1239 }
1240
1241
1242 static void
1243 ngx_clean_old_cycles(ngx_event_t *ev)
1244 {
1245     ngx_uint_t    i, n, found, live;
1246     ngx_log_t     *log;
1247     ngx_cycle_t   **cycle;
1248
1249     log = ngx_cycle->log;
1250     ngx_temp_pool->log = log;
1251
1252     ngx_log_debug0(NGX_LOG_DEBUG_CORE, log, 0, "clean old cycles");
1253
1254     live = 0;
1255
1256     cycle = ngx_old_cycles.elts;
1257     for (i = 0; i < ngx_old_cycles.nelts; i++) {
1258
1259         if (cycle[i] == NULL) {
1260             continue;
1261         }
1262
1263         found = 0;
1264
1265         for (n = 0; n < cycle[i]->connection_n; n++) {
1266             if (cycle[i]->connections[n].fd != (ngx_socket_t) -1) {
1267                 found = 1;
1268
1269                 ngx_log_debug1(NGX_LOG_DEBUG_CORE, log, 0, "live fd:%d", n);
1270
1271                 break;
1272             }
1273         }
1274
1275         if (found) {
1276             live = 1;
1277             continue;
1278         }
1279
1280         ngx_log_debug1(NGX_LOG_DEBUG_CORE, log, 0, "clean old cycle: %d", i);
1281
1282         ngx_destroy_pool(cycle[i]->pool);
1283         cycle[i] = NULL;
1284     }
1285
1286     ngx_log_debug1(NGX_LOG_DEBUG_CORE, log, 0, "old cycles status: %d", live);
1287
1288     if (live) {
1289         ngx_add_timer(ev, 30000);
1290
1291     } else {
1292         ngx_destroy_pool(ngx_temp_pool);
1293         ngx_temp_pool = NULL;
1294         ngx_old_cycles.nelts = 0;
1295     }
1296 }

```

[One Level Up](#)

[Top Level](#)

# src/core/nginx\_cycle.h - nginx-1.7.10

## Data types defined

- [ngx\\_core\\_conf\\_t](#)
- [ngx\\_core\\_tls\\_t](#)
- [ngx\\_cycle\\_s](#)
- [ngx\\_shm\\_zone\\_init\\_pt](#)
- [ngx\\_shm\\_zone\\_s](#)
- [ngx\\_shm\\_zone\\_t](#)

## Macros defined

- [NGX\\_CYCLE\\_POOL\\_SIZE](#)
- [NGX\\_DEBUG\\_POINTS\\_ABORT](#)
- [NGX\\_DEBUG\\_POINTS\\_STOP](#)
- [\\_NGX\\_CYCLE\\_H\\_INCLUDED](#)
- [ngx\\_is\\_init\\_cycle](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_CYCLE_H_INCLUDED_
9 #define _NGX_CYCLE_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 #ifndef NGX_CYCLE_POOL_SIZE
17 #define NGX_CYCLE_POOL_SIZE    NGX_DEFAULT_POOL_SIZE
18 #endif
19
20
21 #define NGX_DEBUG_POINTS_STOP    1
22 #define NGX_DEBUG_POINTS_ABORT  2
23
24
25 typedef struct ngx_shm_zone_s    ngx_shm_zone_t;
26
27 typedef ngx_int_t (*ngx_shm_zone_init_pt) (ngx_shm_zone_t *zone, void *data);
28
29 struct ngx_shm_zone_s {
30     void                *data;
31     ngx_shm_t           shm;
32     ngx_shm_zone_init_pt  init;
33     void                *tag;
34 };
35
36
```



```

37 struct ngx_cycle_s {
38     void                ****conf_ctx;
39     ngx_pool_t          *pool;
40
41     ngx_log_t           *log;
42     ngx_log_t           new_log;
43
44     ngx_uint_t          log_use_stderr; /* unsigned log_use_stderr:1; */
45
46     ngx_connection_t    **files;
47     ngx_connection_t    *free_connections;
48     ngx_uint_t          free_connection_n;
49
50     ngx_queue_t         reusable_connections_queue;
51
52     ngx_array_t         listening;
53     ngx_array_t         paths;
54     ngx_list_t          open_files;
55     ngx_list_t          shared_memory;
56
57     ngx_uint_t          connection_n;
58     ngx_uint_t          files_n;
59
60     ngx_connection_t    *connections;
61     ngx_event_t         *read_events;
62     ngx_event_t         *write_events;
63
64     ngx_cycle_t         *old_cycle;
65
66     ngx_str_t           conf_file;
67     ngx_str_t           conf_param;
68     ngx_str_t           conf_prefix;
69     ngx_str_t           prefix;
70     ngx_str_t           lock_file;
71     ngx_str_t           hostname;
72 };
73
74
75 typedef struct {
76     ngx_flag_t          daemon;
77     ngx_flag_t          master;
78
79     ngx_msec_t          timer_resolution;
80
81     ngx_int_t           worker_processes;
82     ngx_int_t           debug_points;
83
84     ngx_int_t           rlimit_nofile;
85     ngx_int_t           rlimit_sigpending;
86     off_t               rlimit_core;
87
88     int                 priority;
89
90     ngx_uint_t          cpu_affinity_n;
91     uint64_t            *cpu_affinity;
92
93     char                *username;
94     ngx_uid_t           user;
95     ngx_gid_t           group;
96
97     ngx_str_t           working_directory;
98     ngx_str_t           lock_file;
99
100    ngx_str_t           pid;
101    ngx_str_t           oldpid;
102
103    ngx_array_t         env;
104    char                **environment;
105
106    #if (NGX_THREADS)
107        ngx_int_t       worker_threads;
108        size_t          thread_stack_size;
109    #endif
110
111 } ngx_core_conf_t;
112

```

```

113
114 typedef struct {
115     ngx\_pool\_t          *pool;    /* pcre's malloc() pool */
116 } ngx\_core\_tls\_t;
117
118
119 #define ngx\_is\_init\_cycle(cycle) (cycle->conf_ctx == NULL)
120
121
122 ngx\_cycle\_t *ngx\_init\_cycle(ngx\_cycle\_t *old_cycle);
123 ngx\_int\_t ngx\_create\_pidfile(ngx\_str\_t *name, ngx\_log\_t *log);
124 void ngx\_delete\_pidfile(ngx\_cycle\_t *cycle);
125 ngx\_int\_t ngx\_signal\_process(ngx\_cycle\_t *cycle, char *sig);
126 void ngx\_reopen\_files(ngx\_cycle\_t *cycle, ngx\_uid\_t user);
127 char **ngx\_set\_environment(ngx\_cycle\_t *cycle, ngx\_uint\_t *last);
128 ngx\_pid\_t ngx\_exec\_new\_binary(ngx\_cycle\_t *cycle, char *const *argv);
129 uint64\_t ngx\_get\_cpu\_affinity(ngx\_uint\_t n);
130 ngx\_shm\_zone\_t *ngx\_shared\_memory\_add(ngx\_conf\_t *cf, ngx\_str\_t *name,
131     size_t size, void *tag);
132
133
134 extern volatile ngx\_cycle\_t *ngx\_cycle;
135 extern ngx\_array\_t          ngx\_old\_cycles;
136 extern ngx\_module\_t        ngx\_core\_module;
137 extern ngx\_uint\_t          ngx\_test\_config;
138 extern ngx\_uint\_t          ngx\_quiet\_mode;
139 #if (NGX_THREADS)
140 extern ngx\_tls\_key\_t       ngx\_core\_tls\_key;
141 #endif
142
143
144 #endif /* \_NGX\_CYCLE\_H\_INCLUDED */

```

[One Level Up](#)

[Top Level](#)

## src/os/unix/nginx\_shmem.h - nginx-1.7.10

### Data types defined

- [ngx\\_shm\\_t](#)

### Macros defined

- [\\_NGX\\_SHMEM\\_H\\_INCLUDED](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_SHMEM\_H\_INCLUDED
9 #define \_NGX\_SHMEM\_H\_INCLUDED
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 typedef struct {
17     u_char      *addr;
18     size_t      size;
19     ngx\_str\_t    name;
20     ngx\_log\_t   *log;
21     ngx\_uint\_t  exists; /* unsigned exists:1; */
22 } ngx_shm_t;
23
24
25 ngx\_int\_t ngx_shm_alloc(ngx\_shm\_t *shm);
26 void ngx\_shm\_free(ngx\_shm\_t *shm);
27
28
29 #endif /* \_NGX\_SHMEM\_H\_INCLUDED */
```

## src/os/unix/nginx\_shmem.c - nginx-1.7.10

### Functions defined

- [ngx\\_shm\\_alloc](#)
- [ngx\\_shm\\_alloc](#)
- [ngx\\_shm\\_alloc](#)
- [ngx\\_shm\\_free](#)
- [ngx\\_shm\\_free](#)
- [ngx\\_shm\\_free](#)

### Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10
11
12  #if (NGX_HAVE_MAP_ANON)
13
14  ngx_int_t
15  ngx_shm_alloc(ngx_shm_t *shm)
16  {
17      shm->addr = (u_char *) mmap(NULL, shm->size,
18                                PROT_READ|PROT_WRITE,
19                                MAP_ANON|MAP_SHARED, -1, 0);
20
21      if (shm->addr == MAP_FAILED) {
22          ngx_log_error(NGX_LOG_ALERT, shm->log, ngx_errno,
23                      "mmap(MAP_ANON|MAP_SHARED, %uz) failed", shm->size);
24          return NGX_ERROR;
25      }
26
27      return NGX_OK;
28  }
29
30
31  void
32  ngx_shm_free(ngx_shm_t *shm)
33  {
34      if (munmap((void *) shm->addr, shm->size) == -1) {
35          ngx_log_error(NGX_LOG_ALERT, shm->log, ngx_errno,
36                      "munmap(%p, %uz) failed", shm->addr, shm->size);
37      }
38  }
39
40  #elif (NGX_HAVE_MAP_DEVZERO)
41
42  ngx_int_t
43  ngx_shm_alloc(ngx_shm_t *shm)
44  {
45      ngx_fd_t  fd;
46
47      fd = open("/dev/zero", O_RDWR);
48
49      if (fd == -1) {
50          ngx_log_error(NGX_LOG_ALERT, shm->log, ngx_errno,
51                      "open(\"/dev/zero\") failed");

```

```

52     return NGX_ERROR;
53 }
54
55 shm->addr = (u_char *) mmap(NULL, shm->size, PROT_READ|PROT_WRITE,
56                             MAP_SHARED, fd, 0);
57
58 if (shm->addr == MAP_FAILED) {
59     ngx_log_error(NGX_LOG_ALERT, shm->log, ngx_errno,
60                 "mmap(/dev/zero, MAP_SHARED, %uz) failed", shm->size);
61 }
62
63 if (close(fd) == -1) {
64     ngx_log_error(NGX_LOG_ALERT, shm->log, ngx_errno,
65                 "close(\"/dev/zero\") failed");
66 }
67
68 return (shm->addr == MAP_FAILED) ? NGX_ERROR : NGX_OK;
69 }
70
71
72 void
73 ngx_shm_free(ngx_shm_t *shm)
74 {
75     if (munmap((void *) shm->addr, shm->size) == -1) {
76         ngx_log_error(NGX_LOG_ALERT, shm->log, ngx_errno,
77                     "munmap(%p, %uz) failed", shm->addr, shm->size);
78     }
79 }
80
81 #elif (NGX_HAVE_SYSVSHM)
82
83 #include <sys/ipc.h>
84 #include <sys/shm.h>
85
86
87 ngx_int_t
88 ngx_shm_alloc(ngx_shm_t *shm)
89 {
90     int id;
91
92     id = shmget(IPC_PRIVATE, shm->size, (SHM_R|SHM_W|IPC_CREAT));
93
94     if (id == -1) {
95         ngx_log_error(NGX_LOG_ALERT, shm->log, ngx_errno,
96                     "shmget(%uz) failed", shm->size);
97         return NGX_ERROR;
98     }
99
100    ngx_log_debug1(NGX_LOG_DEBUG_CORE, shm->log, 0, "shmget id: %d", id);
101
102    shm->addr = shmat(id, NULL, 0);
103
104    if (shm->addr == (void *) -1) {
105        ngx_log_error(NGX_LOG_ALERT, shm->log, ngx_errno, "shmat() failed");
106    }
107
108    if (shmctl(id, IPC_RMID, NULL) == -1) {
109        ngx_log_error(NGX_LOG_ALERT, shm->log, ngx_errno,
110                    "shmctl(IPC_RMID) failed");
111    }
112
113    return (shm->addr == (void *) -1) ? NGX_ERROR : NGX_OK;
114 }
115
116
117 void
118 ngx_shm_free(ngx_shm_t *shm)
119 {
120     if (shmdt(shm->addr) == -1) {
121         ngx_log_error(NGX_LOG_ALERT, shm->log, ngx_errno,
122                     "shmdt(%p) failed", shm->addr);
123     }
124 }
125
126 #endif

```

[One Level Up](#)

[Top Level](#)

## src/os/unix/nginx\_user.h - nginx-1.7.10

### Data types defined

- [ngx\\_gid\\_t](#)
- [ngx\\_uid\\_t](#)

### Macros defined

- [\\_NGX\\_USER\\_H\\_INCLUDED](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_USER\_H\_INCLUDED
9 #define \_NGX\_USER\_H\_INCLUDED
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 typedef uid_t  ngx_uid_t;
17 typedef gid_t  ngx_gid_t;
18
19
20 ngx_int_t ngx_libc_crypt(ngx_pool_t *pool, u_char *key, u_char *salt,
21     u_char **encrypted);
22
23
24 #endif /* \_NGX\_USER\_H\_INCLUDED */
```

## src/os/unix/nginx\_user.c - nginx-1.7.10

### Functions defined

- [ngx\\_libc\\_crypt](#)
- [ngx\\_libc\\_crypt](#)

### Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10
11
12  /*
13  * Solaris has thread-safe crypt()
14  * Linux has crypt_r(); "struct crypt_data" is more than 128K
15  * FreeBSD needs the mutex to protect crypt()
16  *
17  * TODO:
18  *     ngx_crypt_init() to init mutex
19  */
20
21
22  #if (NGX_CRYPT)
23
24  #if (NGX_HAVE_GNU_CRYPT_R)
25
26  ngx_int_t
27  ngx_libc_crypt(ngx_pool_t *pool, u_char *key, u_char *salt, u_char **encrypted)
28  {
29      char          *value;
30      size_t        len;
31      struct crypt_data  cd;
32
33      cd.initialized = 0;
34  #ifndef __GLIBC__
35      /* work around the glibc bug */
36      cd.current_salt[0] = ~salt[0];
37  #endif
38
39      value = crypt_r((char *) key, (char *) salt, &cd);
40
41      if (value) {
42          len = ngx_strlen(value) + 1;
43
44          *encrypted = ngx_pnalloc(pool, len);
45          if (*encrypted == NULL) {
46              return NGX_ERROR;
47          }
48
49          ngx_memcpy(*encrypted, value, len);
50          return NGX_OK;
51      }
52
53      ngx_log_error(NGX_LOG_CRIT, pool->log, ngx_errno, "crypt_r() failed");
54
55      return NGX_ERROR;
56  }
57
58  #else
59
60  ngx_int_t
```



```

61 ngx_libc_crypt(ngx_pool_t *pool, u_char *key, u_char *salt, u_char **encrypted)
62 {
63     char      *value;
64     size_t    len;
65     ngx_err_t  err;
66
67     #if (NGX_THREADS && NGX_NONREENTRANT_CRYPT)
68
69         /* crypt() is a time consuming function, so we only try to lock */
70
71         if (ngx_mutex_trylock(ngx_crypt_mutex) != NGX_OK) {
72             return NGX_AGAIN;
73         }
74
75     #endif
76
77     value = crypt((char *) key, (char *) salt);
78
79     if (value) {
80         len = ngx_strlen(value) + 1;
81
82         *encrypted = ngx_pnalloc(pool, len);
83         if (*encrypted == NULL) {
84             #if (NGX_THREADS && NGX_NONREENTRANT_CRYPT)
85                 ngx_mutex_unlock(ngx_crypt_mutex);
86             #endif
87             return NGX_ERROR;
88         }
89
90         ngx_memcpy(*encrypted, value, len);
91         #if (NGX_THREADS && NGX_NONREENTRANT_CRYPT)
92             ngx_mutex_unlock(ngx_crypt_mutex);
93         #endif
94         return NGX_OK;
95     }
96
97     err = ngx_errno;
98
99     #if (NGX_THREADS && NGX_NONREENTRANT_CRYPT)
100         ngx_mutex_unlock(ngx_crypt_mutex);
101     #endif
102
103     ngx_log_error(NGX_LOG_CRIT, pool->log, err, "crypt() failed");
104
105     return NGX_ERROR;
106 }
107
108 #endif
109
110 #endif /* NGX_CRYPT */

```

[One Level Up](#)

[Top Level](#)

## src/os/unix/nginx\_pthread\_thread.c - nginx-1.7.10

### Global variables defined

- [max\\_threads](#)
- [nthreads](#)
- [thr\\_attr](#)

### Functions defined

- [ngx\\_cond\\_destroy](#)
- [ngx\\_cond\\_init](#)
- [ngx\\_cond\\_signal](#)
- [ngx\\_cond\\_wait](#)
- [ngx\\_create\\_thread](#)
- [ngx\\_init\\_threads](#)
- [ngx\\_mutex\\_destroy](#)
- [ngx\\_mutex\\_init](#)
- [ngx\\_mutex\\_lock](#)
- [ngx\\_mutex\\_trylock](#)
- [ngx\\_mutex\\_unlock](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10
11
12 static ngx_uint_t  nthreads;
13 static ngx_uint_t  max_threads;
14
15
16 static pthread_attr_t  thr_attr;
17
18
19 ngx_err_t
20 ngx_create_thread(ngx_tid_t *tid, ngx_thread_value_t (*func)(void *arg),
21                 void *arg, ngx_log_t *log)
22 {
23     int  err;
24
25     if (nthreads >= max_threads) {
26         ngx_log_error(NGX_LOG_CRIT, log, 0,
27                     "no more than %ui threads can be created", max_threads);
28         return NGX_ERROR;
29     }
```

```

30
31     err = pthread_create(tid, &thr_attr, func, arg);
32
33     if (err != 0) {
34         ngx_log_error(NGX_LOG_ALERT, log, err, "pthread_create() failed");
35         return err;
36     }
37
38     ngx_log_debug1(NGX_LOG_DEBUG_CORE, log, 0,
39                 "thread is created: " NGX_TID_T_FMT, *tid);
40
41     nthreads++;
42
43     return err;
44 }
45
46
47 ngx_int_t
48 ngx_init_threads(int n, size_t size, ngx_cycle_t *cycle)
49 {
50     int err;
51
52     max_threads = n;
53
54     err = pthread_attr_init(&thr_attr);
55
56     if (err != 0) {
57         ngx_log_error(NGX_LOG_ALERT, cycle->log, err,
58                 "pthread_attr_init() failed");
59         return NGX_ERROR;
60     }
61
62     err = pthread_attr_setstacksize(&thr_attr, size);
63
64     if (err != 0) {
65         ngx_log_error(NGX_LOG_ALERT, cycle->log, err,
66                 "pthread_attr_setstacksize() failed");
67         return NGX_ERROR;
68     }
69
70     ngx_threaded = 1;
71
72     return NGX_OK;
73 }
74
75
76 ngx_mutex_t *
77 ngx_mutex_init(ngx_log_t *log, ngx_uint_t flags)
78 {
79     int err;
80     ngx_mutex_t *m;
81
82     m = ngx_alloc(sizeof(ngx_mutex_t), log);
83     if (m == NULL) {
84         return NULL;
85     }
86
87     m->log = log;
88
89     err = pthread_mutex_init(&m->mutex, NULL);
90
91     if (err != 0) {
92         ngx_log_error(NGX_LOG_ALERT, m->log, err,
93                 "pthread_mutex_init() failed");
94         return NULL;
95     }
96
97     return m;
98 }
99
100
101 void
102 ngx_mutex_destroy(ngx_mutex_t *m)
103 {
104     int err;
105

```

```

106     err = pthread_mutex_destroy(&m->mutex);
107
108     if (err != 0) {
109         ngx_log_error(NGX_LOG_ALERT, m->log, err,
110                     "pthread_mutex_destroy(%p) failed", m);
111     }
112
113     ngx_free(m);
114 }
115
116
117 void
118 ngx_mutex_lock(ngx_mutex_t *m)
119 {
120     int err;
121
122     if (!ngx_threaded) {
123         return;
124     }
125
126     ngx_log_debug1(NGX_LOG_DEBUG_MUTEX, m->log, 0, "lock mutex %p", m);
127
128     err = pthread_mutex_lock(&m->mutex);
129
130     if (err != 0) {
131         ngx_log_error(NGX_LOG_ALERT, m->log, err,
132                     "pthread_mutex_lock(%p) failed", m);
133         ngx_abort();
134     }
135
136     ngx_log_debug1(NGX_LOG_DEBUG_MUTEX, m->log, 0, "mutex %p is locked", m);
137
138     return;
139 }
140
141
142 ngx_int_t
143 ngx_mutex_trylock(ngx_mutex_t *m)
144 {
145     int err;
146
147     if (!ngx_threaded) {
148         return NGX_OK;
149     }
150
151     ngx_log_debug1(NGX_LOG_DEBUG_MUTEX, m->log, 0, "try lock mutex %p", m);
152
153     err = pthread_mutex_trylock(&m->mutex);
154
155     if (err == NGX_EBUSY) {
156         return NGX_AGAIN;
157     }
158
159     if (err != 0) {
160         ngx_log_error(NGX_LOG_ALERT, m->log, err,
161                     "pthread_mutex_trylock(%p) failed", m);
162         ngx_abort();
163     }
164
165     ngx_log_debug1(NGX_LOG_DEBUG_MUTEX, m->log, 0, "mutex %p is locked", m);
166
167     return NGX_OK;
168 }
169
170
171 void
172 ngx_mutex_unlock(ngx_mutex_t *m)
173 {
174     int err;
175
176     if (!ngx_threaded) {
177         return;
178     }
179
180     ngx_log_debug1(NGX_LOG_DEBUG_MUTEX, m->log, 0, "unlock mutex %p", m);
181

```

```

182     err = pthread_mutex_unlock(&m->mutex);
183
184     if (err != 0) {
185         ngx_log_error(NGX_LOG_ALERT, m->log, err,
186                     "pthread_mutex_unlock(%p) failed", m);
187         ngx_abort();
188     }
189
190     ngx_log_debug1(NGX_LOG_DEBUG_MUTEX, m->log, 0, "mutex %p is unlocked", m);
191
192     return;
193 }
194
195
196 ngx_cond_t *
197 ngx_cond_init(ngx_log_t *log)
198 {
199     int     err;
200     ngx_cond_t *cv;
201
202     cv = ngx_alloc(sizeof(ngx_cond_t), log);
203     if (cv == NULL) {
204         return NULL;
205     }
206
207     cv->log = log;
208
209     err = pthread_cond_init(&cv->cond, NULL);
210
211     if (err != 0) {
212         ngx_log_error(NGX_LOG_ALERT, cv->log, err,
213                     "pthread_cond_init() failed");
214         return NULL;
215     }
216
217     return cv;
218 }
219
220
221 void
222 ngx_cond_destroy(ngx_cond_t *cv)
223 {
224     int err;
225
226     err = pthread_cond_destroy(&cv->cond);
227
228     if (err != 0) {
229         ngx_log_error(NGX_LOG_ALERT, cv->log, err,
230                     "pthread_cond_destroy(%p) failed", cv);
231     }
232
233     ngx_free(cv);
234 }
235
236
237 ngx_int_t
238 ngx_cond_wait(ngx_cond_t *cv, ngx_mutex_t *m)
239 {
240     int err;
241
242     ngx_log_debug1(NGX_LOG_DEBUG_CORE, cv->log, 0, "cv %p wait", cv);
243
244     err = pthread_cond_wait(&cv->cond, &m->mutex);
245
246     if (err != 0) {
247         ngx_log_error(NGX_LOG_ALERT, cv->log, err,
248                     "pthread_cond_wait(%p) failed", cv);
249         return NGX_ERROR;
250     }
251
252     ngx_log_debug1(NGX_LOG_DEBUG_CORE, cv->log, 0, "cv %p is waked up", cv);
253
254     ngx_log_debug1(NGX_LOG_DEBUG_MUTEX, m->log, 0, "mutex %p is locked", m);
255
256     return NGX_OK;
257 }

```

```
258
259
260 ngx\_int\_t
261 ngx\_cond\_signal(ngx\_cond\_t *cv)
262 {
263     int err;
264
265     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_CORE, cv->log, 0, "cv %p to signal", cv);
266
267     err = pthread_cond_signal(&cv->cond);
268
269     if (err != 0) {
270         ngx\_log\_error(NGX\_LOG\_ALERT, cv->log, err,
271                     "pthread_cond_signal(%p) failed", cv);
272         return NGX\_ERROR;
273     }
274
275     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_CORE, cv->log, 0, "cv %p is signaled", cv);
276
277     return NGX\_OK;
278 }
```

[One Level Up](#)

[Top Level](#)

## src/os/unix/nginx\_thread.h - nginx-1.7.10

### Data types defined

- [ngx\\_cond\\_t](#)
- [ngx\\_mutex\\_t](#)
- [ngx\\_thread\\_t](#)
- [ngx\\_thread\\_value\\_t](#)
- [ngx\\_tid\\_t](#)
- [ngx\\_tls\\_key\\_t](#)

### Macros defined

- [NGX\\_MAX\\_THREADS](#)
- [NGX\\_MUTEX\\_LIGHT](#)
- [NGX\\_THREAD\\_BUSY](#)
- [NGX\\_THREAD\\_DONE](#)
- [NGX\\_THREAD\\_EXIT](#)
- [NGX\\_THREAD\\_FREE](#)
- [NGX\\_TID\\_T\\_FMT](#)
- [NGX\\_TID\\_T\\_FMT](#)
- [NGX\\_TID\\_T\\_FMT](#)
- [\\_NGX\\_THREAD\\_H\\_INCLUDED\\_](#)
- [ngx\\_cond\\_signal](#)
- [ngx\\_log\\_tid](#)
- [ngx\\_log\\_tid](#)
- [ngx\\_mutex\\_lock](#)
- [ngx\\_mutex\\_trylock](#)
- [ngx\\_mutex\\_unlock](#)
- [ngx\\_setthrtile](#)
- [ngx\\_thread\\_get\\_tls](#)
- [ngx\\_thread\\_join](#)
- [ngx\\_thread\\_key\\_create](#)
- [ngx\\_thread\\_key\\_create\\_n](#)
- [ngx\\_thread\\_main](#)

- [ngx\\_thread\\_self](#)
- [ngx\\_thread\\_set\\_tls](#)
- [ngx\\_thread\\_set\\_tls\\_n](#)
- [ngx\\_thread\\_sigmask](#)
- [ngx\\_thread\\_sigmask\\_n](#)
- [ngx\\_thread\\_volatile](#)
- [ngx\\_thread\\_volatile](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_THREAD_H_INCLUDED_
9 #define _NGX_THREAD_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15 #if (NGX_THREADS)
16
17 #define NGX_MAX_THREADS      128
18
19 #if (NGX_USE_RFORK)
20 #include <ngx_freebsd_rfork_thread.h>
21
22
23 #else /* use pthreads */
24
25 #include <pthread.h>
26
27 typedef pthread_t          ngx_tid_t;
28
29 #define ngx_thread_self()  pthread_self()
30 #define ngx_log_tid      (int) ngx_thread_self()
31
32 #if (NGX_FREEBSD) && !(NGX_LINUXTHREADS)
33 #define NGX_TID_T_FMT      "%p"
34 #else
35 #define NGX_TID_T_FMT      "%d"
36 #endif
37
38
39 typedef pthread_key_t      ngx_tls_key_t;
40
41 #define ngx_thread_key_create(key)  pthread_key_create(key, NULL)
42 #define ngx_thread_key_create_n    "pthread_key_create()"
43 #define ngx_thread_set_tls         pthread_setspecific
44 #define ngx_thread_set_tls_n      "pthread_setspecific()"
45 #define ngx_thread_get_tls         pthread_getspecific
46
47
48 #define NGX_MUTEX_LIGHT      0
49
50 typedef struct {
51     pthread_mutex_t  mutex;
52     ngx_log_t        *log;
53 } ngx_mutex_t;
54
55 typedef struct {
56     pthread_cond_t   cond;

```



```

57     ngx_log_t          *log;
58 } ngx_cond_t;
59
60 #define ngx_thread_sigmask    pthread_sigmask
61 #define ngx_thread_sigmask_n "pthread_sigmask()"
62
63 #define ngx_thread_join(t, p) pthread_join(t, p)
64
65 #define ngx_setthrtitle(n)
66
67
68
69 ngx_int_t ngx_mutex_trylock(ngx_mutex_t *m);
70 void ngx_mutex_lock(ngx_mutex_t *m);
71 void ngx_mutex_unlock(ngx_mutex_t *m);
72
73 #endif
74
75
76 #define ngx_thread_volatile    volatile
77
78
79 typedef struct {
80     ngx_tid_t    tid;
81     ngx_cond_t  *cv;
82     ngx_uint_t   state;
83 } ngx_thread_t;
84
85 #define NGX_THREAD_FREE    1
86 #define NGX_THREAD_BUSY   2
87 #define NGX_THREAD_EXIT    3
88 #define NGX_THREAD_DONE    4
89
90 extern ngx_int_t          ngx_threads_n;
91 extern volatile ngx_thread_t ngx_threads[NGX_MAX_THREADS];
92
93
94 typedef void * ngx_thread_value_t;
95
96 ngx_int_t ngx_init_threads(int n, size_t size, ngx_cycle_t *cycle);
97 ngx_err_t ngx_create_thread(ngx_tid_t *tid,
98     ngx_thread_value_t (*func)(void *arg), void *arg, ngx_log_t *log);
99
100 ngx_mutex_t *ngx_mutex_init(ngx_log_t *log, ngx_uint_t flags);
101 void ngx_mutex_destroy(ngx_mutex_t *m);
102
103
104 ngx_cond_t *ngx_cond_init(ngx_log_t *log);
105 void ngx_cond_destroy(ngx_cond_t *cv);
106 ngx_int_t ngx_cond_wait(ngx_cond_t *cv, ngx_mutex_t *m);
107 ngx_int_t ngx_cond_signal(ngx_cond_t *cv);
108
109
110 #else /* !NGX_THREADS */
111
112 #define ngx_thread_volatile
113
114 #define ngx_log_tid          0
115 #define NGX_TID_T_FMT       "%d"
116
117 #define ngx_mutex_trylock(m) NGX_OK
118 #define ngx_mutex_lock(m)
119 #define ngx_mutex_unlock(m)
120
121 #define ngx_cond_signal(cv)
122
123 #define ngx_thread_main()    1
124
125 #endif
126
127
128 #endif /* _NGX_THREAD_H_INCLUDED */

```

## src/os/unix/nginx\_freebsd\_rfork\_thread.c - nginx-1.7.10

### Global variables defined

- [errno0](#)
- [errnos](#)
- [last\\_stack](#)
- [max\\_threads](#)
- [ngx\\_freebsd\\_kern\\_usrstack](#)
- [ngx\\_thread\\_stack\\_size](#)
- [ngx\\_tls](#)
- [nkeys](#)
- [nthreads](#)
- [rz\\_size](#)
- [tids](#)
- [usable\\_stack\\_size](#)

### Functions defined

- [\\_\\_error](#)
- [\\_spinlock](#)
- [\\_spinunlock](#)
- [ngx\\_cond\\_destroy](#)
- [ngx\\_cond\\_init](#)
- [ngx\\_cond\\_signal](#)
- [ngx\\_cond\\_wait](#)
- [ngx\\_create\\_thread](#)
- [ngx\\_init\\_threads](#)
- [ngx\\_mutex\\_destroy](#)
- [ngx\\_mutex\\_dolock](#)
- [ngx\\_mutex\\_init](#)
- [ngx\\_mutex\\_unlock](#)
- [ngx\\_thread\\_key\\_create](#)
- [ngx\\_thread\\_self](#)
- [ngx\\_thread\\_set\\_tls](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10
11 /*
12  * The threads implementation uses the rfork(RFPROC|RFTHREAD|RFMEM) syscall
13  * to create threads. All threads use the stacks of the same size mmap()'ed
14  * below the main stack. Thus the current thread id is determined via
15  * the stack pointer value.
16  *
17  * The mutex implementation uses the ngx_atomic_cmp_set() operation
18  * to acquire a mutex and the SysV semaphore to wait on a mutex and to wake up
19  * the waiting threads. The light mutex does not use semaphore, so after
20  * spinning in the lock the thread calls sched_yield(). However the light
21  * mutexes are intended to be used with the "trylock" operation only.
22  * The SysV semop() is a cheap syscall, particularly if it has little sembuf's
23  * and does not use SEM_UNDO.
24  *
25  * The condition variable implementation uses the signal #64.
26  * The signal handler is SIG_IGN so the kill() is a cheap syscall.
27  * The thread waits a signal in kevent(). The use of the EVFILT_SIGNAL
28  * is safe since FreeBSD 4.10-STABLE.
29  *
30  * This threads implementation currently works on i386 (486+) and amd64
31  * platforms only.
32  */
33
34
35 char          *ngx_freebsd_kern_usrstack;
36 size_t        ngx_thread_stack_size;
37
38
39 static size_t  rz_size;
40 static size_t  usable_stack_size;
41 static char    *last_stack;
42
43 static ngx_uint_t  nthreads;
44 static ngx_uint_t  max_threads;
45
46 static ngx_uint_t  nkeys;
47 static ngx_tid_t  *tids;      /* the threads tids array */
48 void                **ngx_tls; /* the threads tls's array */
49
50 /* the thread-safe libc errno */
51
52 static int  errno0; /* the main thread's errno */
53 static int  *errnos; /* the threads errno's array */
54
55 int *
56 __error()
57 {
58     int tid;
59
60     tid = ngx_gettid();
61
62     return tid ? &errnos[tid - 1] : &errno0;
63 }
64
65
66 /*
67  * __isthreaded enables the spinlocks in some libc functions, i.e. in malloc()
68  * and some other places. Nevertheless we protect our malloc()/free() calls
69  * by own mutex that is more efficient than the spinlock.
70  *
71  * spinlock() is a weak referenced stub in src/lib/libc/gen/_spinlock_stub.c
72  * that does nothing.
73  */
```

```

74
75 extern int __isthreaded;
76
77 void
78 _spinlock(ngx_atomic_t *lock)
79 {
80     ngx_int_t tries;
81
82     tries = 0;
83
84     for ( ;; ) {
85
86         if (*lock) {
87             if (ngx_ncpu > 1 && tries++ < 1000) {
88                 continue;
89             }
90
91             sched_yield();
92             tries = 0;
93
94         } else {
95             if (ngx_atomic_cmp_set(lock, 0, 1)) {
96                 return;
97             }
98         }
99     }
100 }
101
102
103 /*
104 * Before FreeBSD 5.1 spinunlock() is a simple #define in
105 * src/lib/libc/include/spinlock.h that zeroes lock.
106 *
107 * Since FreeBSD 5.1 spinunlock() is a weak referenced stub in
108 * src/lib/libc/gen/_spinlock_stub.c that does nothing.
109 */
110
111 #ifndef spinunlock
112
113 void
114 _spinunlock(ngx_atomic_t *lock)
115 {
116     *lock = 0;
117 }
118
119 #endif
120
121
122 ngx_err_t
123 ngx_create_thread(ngx_tid_t *tid, ngx_thread_value_t (*func)(void *arg),
124                 void *arg, ngx_log_t *log)
125 {
126     ngx_pid_t id;
127     ngx_err_t err;
128     char *stack, *stack_top;
129
130     if (nthreads >= max_threads) {
131         ngx_log_error(NGX_LOG_CRIT, log, 0,
132                     "no more than %ui threads can be created", max_threads);
133         return NGX_ERROR;
134     }
135
136     last_stack -= ngx_thread_stack_size;
137
138     stack = mmap(last_stack, usable_stack_size, PROT_READ|PROT_WRITE,
139                MAP_STACK, -1, 0);
140
141     if (stack == MAP_FAILED) {
142         ngx_log_error(NGX_LOG_ALERT, log, ngx_errno,
143                     "mmap(%p:%uz, MAP_STACK) thread stack failed",
144                     last_stack, usable_stack_size);
145         return NGX_ERROR;
146     }
147
148     if (stack != last_stack) {
149         ngx_log_error(NGX_LOG_ALERT, log, 0,

```

```

150     "stack %p address was changed to %p", last\_stack, stack);
151     return NGX\_ERROR;
152 }
153
154 stack_top = stack + usable\_stack\_size;
155
156 ngx\_log\_debug2(NGX\_LOG\_DEBUG\_CORE, log, 0,
157     "thread stack: %p-%p", stack, stack_top);
158
159 ngx\_set\_errno(0);
160
161 id = rfork_thread(RFPROC|RFTHREAD|RFMEM, stack_top,
162     (ngx\_rfork\_thread\_func\_pt) func, arg);
163
164 err = ngx\_errno;
165
166 if (id == -1) {
167     ngx\_log\_error(NGX\_LOG\_ALERT, log, err, "rfork() failed");
168 } else {
169     *tid = id;
170     nthreads = (ngx\_freebsd\_kern\_usrstack - stack_top)
171                 / ngx\_thread\_stack\_size;
172     tids[nthreads] = id;
173
174     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_CORE, log, 0, "rfork()ed thread: %P", id);
175 }
176
177 return err;
178 }
179
180
181
182 ngx\_int\_t
183 ngx\_init\_threads(int n, size_t size, ngx\_cycle\_t *cycle)
184 {
185     char            *red_zone, *zone;
186     size_t          len;
187     ngx\_int\_t       i;
188     struct sigaction sa;
189
190     max\_threads = n + 1;
191
192     for (i = 0; i < n; i++) {
193         ngx\_memzero(&sa, sizeof(struct sigaction));
194         sa.sa_handler = SIG_IGN;
195         sigemptyset(&sa.sa_mask);
196         if (sigaction(NGX\_CV\_SIGNAL, &sa, NULL) == -1) {
197             ngx\_log\_error(NGX\_LOG\_ALERT, cycle->log, ngx\_errno,
198                 "sigaction(%d, SIG_IGN) failed", NGX\_CV\_SIGNAL);
199             return NGX\_ERROR;
200         }
201     }
202
203     len = sizeof(ngx\_freebsd\_kern\_usrstack);
204     if (sysctlbyname("kern.usrstack", &ngx\_freebsd\_kern\_usrstack, &len,
205         NULL, 0) == -1)
206     {
207         ngx\_log\_error(NGX\_LOG\_ALERT, cycle->log, ngx\_errno,
208             "sysctlbyname(kern.usrstack) failed");
209         return NGX\_ERROR;
210     }
211
212     /* the main thread stack red zone */
213     rz\_size = ngx\_pagesize;
214     red_zone = ngx\_freebsd\_kern\_usrstack - (size + rz\_size);
215
216     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_CORE, cycle->log, 0,
217         "usrstack: %p red zone: %p",
218         ngx\_freebsd\_kern\_usrstack, red_zone);
219
220     zone = mmap(red_zone, rz\_size, PROT_NONE, MAP_ANON, -1, 0);
221     if (zone == MAP_FAILED) {
222         ngx\_log\_error(NGX\_LOG\_ALERT, cycle->log, ngx\_errno,
223             "mmap(%p:%uz, PROT_NONE, MAP_ANON) red zone failed",
224             red_zone, rz\_size);
225         return NGX\_ERROR;

```

```

226 }
227
228 if (zone != red_zone) {
229     ngx_log_error(NGX_LOG_ALERT, cycle->log, 0,
230         "red zone %p address was changed to %p", red_zone, zone);
231     return NGX_ERROR;
232 }
233
234 /* create the thread errno' array */
235
236 errnos = ngx_calloc(n * sizeof(int), cycle->log);
237 if (errnos == NULL) {
238     return NGX_ERROR;
239 }
240
241 /* create the thread tids array */
242
243 tids = ngx_calloc((n + 1) * sizeof(ngx_tid_t), cycle->log);
244 if (tids == NULL) {
245     return NGX_ERROR;
246 }
247
248 tids[0] = ngx_pid;
249
250 /* create the thread tls' array */
251
252 ngx_tls = ngx_calloc(NGX_THREAD_KEYS_MAX * (n + 1) * sizeof(void *),
253     cycle->log);
254 if (ngx_tls == NULL) {
255     return NGX_ERROR;
256 }
257
258 nthreads = 1;
259
260 last_stack = zone + rz_size;
261 usable_stack_size = size;
262 ngx_thread_stack_size = size + rz_size;
263
264 /* allow the spinlock in libc malloc() */
265 __isthreaded = 1;
266
267 ngx_threaded = 1;
268
269 return NGX_OK;
270 }
271
272
273 ngx_tid_t
274 ngx_thread_self(void)
275 {
276     ngx_int_t tid;
277
278     tid = ngx_gettid();
279
280     if (tids == NULL) {
281         return ngx_pid;
282     }
283
284     return tids[tid];
285 }
286
287
288 ngx_err_t
289 ngx_thread_key_create(ngx_tls_key_t *key)
290 {
291     if (nkeys >= NGX_THREAD_KEYS_MAX) {
292         return NGX_ENOMEM;
293     }
294
295     *key = nkeys++;
296
297     return 0;
298 }
299
300
301 ngx_err_t

```

```

302 ngx_thread_set_tls(ngx_tls_key_t key, void *value)
303 {
304     if (key >= NGX_THREAD_KEYS_MAX) {
305         return NGX_EINVAL;
306     }
307
308     ngx_tls[key * NGX_THREAD_KEYS_MAX + ngx_gettid()] = value;
309     return 0;
310 }
311
312
313 ngx_mutex_t *
314 ngx_mutex_init(ngx_log_t *log, ngx_uint_t flags)
315 {
316     ngx_mutex_t *m;
317     union semun  op;
318
319     m = ngx_alloc(sizeof(ngx_mutex_t), log);
320     if (m == NULL) {
321         return NULL;
322     }
323
324     m->lock = 0;
325     m->log = log;
326
327     if (flags & NGX_MUTEX_LIGHT) {
328         m->semid = -1;
329         return m;
330     }
331
332     m->semid = semget(IPC_PRIVATE, 1, SEM_R|SEM_A);
333     if (m->semid == -1) {
334         ngx_log_error(NGX_LOG_ALERT, log, ngx_errno, "semget() failed");
335         return NULL;
336     }
337
338     op.val = 0;
339
340     if (semctl(m->semid, 0, SETVAL, op) == -1) {
341         ngx_log_error(NGX_LOG_ALERT, log, ngx_errno, "semctl(SETVAL) failed");
342
343         if (semctl(m->semid, 0, IPC_RMID) == -1) {
344             ngx_log_error(NGX_LOG_ALERT, log, ngx_errno,
345                 "semctl(IPC_RMID) failed");
346         }
347
348         return NULL;
349     }
350
351     return m;
352 }
353
354
355 void
356 ngx_mutex_destroy(ngx_mutex_t *m)
357 {
358     if (semctl(m->semid, 0, IPC_RMID) == -1) {
359         ngx_log_error(NGX_LOG_ALERT, m->log, ngx_errno,
360             "semctl(IPC_RMID) failed");
361     }
362
363     ngx_free((void *) m);
364 }
365
366
367 ngx_int_t
368 ngx_mutex_dolock(ngx_mutex_t *m, ngx_int_t try)
369 {
370     uint32_t    lock, old;
371     ngx_uint_t  tries;
372     struct sembuf  op;
373
374     if (!ngx_threaded) {
375         return NGX_OK;
376     }
377

```

```

378 #if (NGX_DEBUG)
379     if (try) {
380         ngx_log_debug2(NGX_LOG_DEBUG_MUTEX, m->log, 0,
381             "try lock mutex %p lock:%XD", m, m->lock);
382     } else {
383         ngx_log_debug2(NGX_LOG_DEBUG_MUTEX, m->log, 0,
384             "lock mutex %p lock:%XD", m, m->lock);
385     }
386 #endif
387
388     old = m->lock;
389     tries = 0;
390
391     for ( ;; ) {
392         if (old & NGX_MUTEX_LOCK_BUSY) {
393
394             if (try) {
395                 return NGX_AGAIN;
396             }
397
398             if (ngx_ncpu > 1 && tries++ < 1000) {
399
400                 /* the spinlock is used only on the SMP system */
401
402                 old = m->lock;
403                 continue;
404             }
405
406             if (m->semid == -1) {
407                 sched_yield();
408
409                 tries = 0;
410                 old = m->lock;
411                 continue;
412             }
413
414             ngx_log_debug2(NGX_LOG_DEBUG_MUTEX, m->log, 0,
415                 "mutex %p lock:%XD", m, m->lock);
416
417             /*
418              * The mutex is locked so we increase a number
419              * of the threads that are waiting on the mutex
420              */
421
422             lock = old + 1;
423
424             if ((lock & ~NGX_MUTEX_LOCK_BUSY) > nthreads) {
425                 ngx_log_error(NGX_LOG_ALERT, m->log, ngx_errno,
426                     "%D threads wait for mutex %p, "
427                     "while only %ui threads are available",
428                     lock & ~NGX_MUTEX_LOCK_BUSY, m, nthreads);
429                 ngx_abort();
430             }
431
432             if (ngx_atomic_cmp_set(&m->lock, old, lock)) {
433
434                 ngx_log_debug2(NGX_LOG_DEBUG_MUTEX, m->log, 0,
435                     "wait mutex %p lock:%XD", m, m->lock);
436
437                 /*
438                  * The number of the waiting threads has been increased
439                  * and we would wait on the SysV semaphore.
440                  * A semaphore should wake up us more efficiently than
441                  * a simple sched_yield() or usleep().
442                  */
443
444                 op.sem_num = 0;
445                 op.sem_op = -1;
446                 op.sem_flg = 0;
447
448                 if (semop(m->semid, &op, 1) == -1) {
449                     ngx_log_error(NGX_LOG_ALERT, m->log, ngx_errno,
450                         "semop() failed while waiting on mutex %p", m);
451                     ngx_abort();
452                 }
453

```



```

454         ngx_log_debug2(NGX_LOG_DEBUG_MUTEX, m->log, 0,
455             "mutex waked up %p lock:%XD", m, m->lock);
456
457         tries = 0;
458         old = m->lock;
459         continue;
460     }
461
462     old = m->lock;
463
464     } else {
465         lock = old | NGX_MUTEX_LOCK_BUSY;
466
467         if (ngx_atomic_cmp_set(&m->lock, old, lock)) {
468
469             /* we locked the mutex */
470
471             break;
472         }
473
474         old = m->lock;
475     }
476
477     if (tries++ > 1000) {
478
479         ngx_log_debug1(NGX_LOG_DEBUG_MUTEX, m->log, 0,
480             "mutex %p is contested", m);
481
482         /* the mutex is probably contested so we are giving up now */
483
484         sched_yield();
485
486         tries = 0;
487         old = m->lock;
488     }
489 }
490
491 ngx_log_debug2(NGX_LOG_DEBUG_MUTEX, m->log, 0,
492     "mutex %p is locked, lock:%XD", m, m->lock);
493
494 return NGX_OK;
495 }
496
497 void
498 ngx_mutex_unlock(ngx_mutex_t *m)
499 {
500     uint32_t    lock, old;
501     struct sembuf op;
502
503     if (!ngx_threaded) {
504         return;
505     }
506
507     old = m->lock;
508
509     if (!(old & NGX_MUTEX_LOCK_BUSY)) {
510         ngx_log_error(NGX_LOG_ALERT, m->log, 0,
511             "trying to unlock the free mutex %p", m);
512         ngx_abort();
513     }
514
515     /* free the mutex */
516
517 #if 0
518     ngx_log_debug2(NGX_LOG_DEBUG_MUTEX, m->log, 0,
519         "unlock mutex %p lock:%XD", m, old);
520 #endif
521
522     for ( ;; ) {
523         lock = old & ~NGX_MUTEX_LOCK_BUSY;
524
525         if (ngx_atomic_cmp_set(&m->lock, old, lock)) {
526             break;
527         }
528     }
529 }

```

```

530     old = m->lock;
531 }
532
533 if (m->semid == -1) {
534     ngx_log_debug1(NGX_LOG_DEBUG_MUTEX, m->log, 0,
535                 "mutex %p is unlocked", m);
536
537     return;
538 }
539
540 /* check whether we need to wake up a waiting thread */
541
542 old = m->lock;
543
544 for ( ;; ) {
545     if (old & NGX_MUTEX_LOCK_BUSY) {
546
547         /* the mutex is just locked by another thread */
548
549         break;
550     }
551
552     if (old == 0) {
553         break;
554     }
555
556     /* there are the waiting threads */
557
558     lock = old - 1;
559
560     if (ngx_atomic_cmp_set(&m->lock, old, lock)) {
561
562         /* wake up the thread that waits on semaphore */
563
564         ngx_log_debug1(NGX_LOG_DEBUG_MUTEX, m->log, 0,
565                 "wake up mutex %p", m);
566
567         op.sem_num = 0;
568         op.sem_op = 1;
569         op.sem_flg = 0;
570
571         if (semop(m->semid, &op, 1) == -1) {
572             ngx_log_error(NGX_LOG_ALERT, m->log, ngx_errno,
573                 "semop() failed while waking up on mutex %p", m);
574             ngx_abort();
575         }
576
577         break;
578     }
579
580     old = m->lock;
581 }
582
583 ngx_log_debug1(NGX_LOG_DEBUG_MUTEX, m->log, 0,
584                 "mutex %p is unlocked", m);
585
586 return;
587 }
588
589 ngx_cond_t *
590 ngx_cond_init(ngx_log_t *log)
591 {
592     ngx_cond_t *cv;
593
594     cv = ngx_alloc(sizeof(ngx_cond_t), log);
595     if (cv == NULL) {
596         return NULL;
597     }
598
599     cv->signo = NGX_CV_SIGNAL;
600     cv->tid = -1;
601     cv->log = log;
602     cv->kq = -1;
603
604     return cv;
605 }

```

```

606 }
607
608
609 void
610 ngx_cond_destroy(ngx_cond_t *cv)
611 {
612     if (close(cv->kq) == -1) {
613         ngx_log_error(NGX_LOG_ALERT, cv->log, ngx_errno,
614             "kqueue close() failed");
615     }
616
617     ngx_free(cv);
618 }
619
620
621 ngx_int_t
622 ngx_cond_wait(ngx_cond_t *cv, ngx_mutex_t *m)
623 {
624     int n;
625     ngx_err_t err;
626     struct kevent kev;
627     struct timespec ts;
628
629     if (cv->kq == -1) {
630
631         /*
632          * We have to add the EVFILT_SIGNAL filter in the rfork()ed thread.
633          * Otherwise the thread would not get a signal event.
634          *
635          * However, we have not to open the kqueue in the thread,
636          * it is simply handy do it together.
637          */
638
639         cv->kq = kqueue();
640         if (cv->kq == -1) {
641             ngx_log_error(NGX_LOG_ALERT, cv->log, ngx_errno, "kqueue() failed");
642             return NGX_ERROR;
643         }
644
645         ngx_log_debug2(NGX_LOG_DEBUG_CORE, cv->log, 0,
646             "cv kq:%d signo:%d", cv->kq, cv->signo);
647
648         kev.ident = cv->signo;
649         kev.filter = EVFILT_SIGNAL;
650         kev.flags = EV_ADD;
651         kev.fflags = 0;
652         kev.data = 0;
653         kev.udata = NULL;
654
655         ts.tv_sec = 0;
656         ts.tv_nsec = 0;
657
658         if (kevent(cv->kq, &kev, 1, NULL, 0, &ts) == -1) {
659             ngx_log_error(NGX_LOG_ALERT, cv->log, ngx_errno, "kevent() failed");
660             return NGX_ERROR;
661         }
662
663         cv->tid = ngx_thread_self();
664     }
665
666     ngx_mutex_unlock(m);
667
668     ngx_log_debug3(NGX_LOG_DEBUG_CORE, cv->log, 0,
669         "cv %p wait, kq:%d, signo:%d", cv, cv->kq, cv->signo);
670
671     for ( ;; ) {
672         n = kevent(cv->kq, NULL, 0, &kev, 1, NULL);
673
674         ngx_log_debug2(NGX_LOG_DEBUG_CORE, cv->log, 0,
675             "cv %p kevent: %d", cv, n);
676
677         if (n == -1) {
678             err = ngx_errno;
679             ngx_log_error((err == NGX_EINTR) ? NGX_LOG_INFO : NGX_LOG_ALERT,
680                 cv->log, ngx_errno,
681                 "kevent() failed while waiting condition variable %p",

```

```

682         cv);
683
684     if (err == NGX\_EINTR) {
685         break;
686     }
687
688     return NGX\_ERROR;
689 }
690
691 if (n == 0) {
692     ngx\_log\_error(NGX\_LOG\_ALERT, cv->log, 0,
693         "kevent() returned no events "
694         "while waiting condition variable %p",
695         cv);
696     continue;
697 }
698
699 if (kev.filter != EVFILT_SIGNAL) {
700     ngx\_log\_error(NGX\_LOG\_ALERT, cv->log, 0,
701         "kevent() returned unexpected events: %d "
702         "while waiting condition variable %p",
703         kev.filter, cv);
704     continue;
705 }
706
707 if (kev.ident != (uintptr_t) cv->signo) {
708     ngx\_log\_error(NGX\_LOG\_ALERT, cv->log, 0,
709         "kevent() returned unexpected signal: %d ",
710         "while waiting condition variable %p",
711         kev.ident, cv);
712     continue;
713 }
714
715 break;
716 }
717
718 ngx\_log\_debug1(NGX\_LOG\_DEBUG\_CORE, cv->log, 0, "cv %p is waked up", cv);
719
720 ngx\_mutex\_lock(m);
721
722 return NGX\_OK;
723 }
724
725
726 ngx\_int\_t
727 ngx\_cond\_signal(ngx\_cond\_t *cv)
728 {
729     ngx\_err\_t err;
730
731     ngx\_log\_debug3(NGX\_LOG\_DEBUG\_CORE, cv->log, 0,
732         "cv %p to signal %P %d",
733         cv, cv->tid, cv->signo);
734
735     if (cv->tid == -1) {
736         return NGX\_OK;
737     }
738
739     if (kill(cv->tid, cv->signo) == -1) {
740
741         err = ngx\_errno;
742
743         ngx\_log\_error(NGX\_LOG\_ALERT, cv->log, err,
744             "kill() failed while signaling condition variable %p", cv);
745
746         if (err == NGX\_ESRCH) {
747             cv->tid = -1;
748         }
749
750         return NGX\_ERROR;
751     }
752
753     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_CORE, cv->log, 0, "cv %p is signaled", cv);
754
755     return NGX\_OK;
756 }

```

[One Level Up](#)

[Top Level](#)

## src/os/unix/nginx\_gcc\_atomic\_sparc64.h - nginx-1.7.10

### Functions defined

- [ngx\\_atomic\\_cmp\\_set](#)
- [ngx\\_atomic\\_fetch\\_add](#)

### Macros defined

- [NGX\\_CASA](#)
- [NGX\\_CASA](#)
- [ngx\\_cpu\\_pause](#)
- [ngx\\_memory\\_barrier](#)
- [ngx\\_memory\\_barrier](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 /*
9  * "casa [r1] 0x80, r2, r0" and
10 * "casxa [r1] 0x80, r2, r0" do the following:
11 *
12 *     if ([r1] == r2) {
13 *         swap(r0, [r1]);
14 *     } else {
15 *         r0 = [r1];
16 *     }
17 *
18 * so "r0 == r2" means that the operation was successful.
19 *
20 *
21 * The "r" means the general register.
22 * The "+r" means the general register used for both input and output.
23 */
24
25
26 #if (NGX_PTR_SIZE == 4)
27 #define NGX_CASA "casa"
28 #else
29 #define NGX_CASA "casxa"
30 #endif
31
32
33 static ngx_inline ngx_atomic_uint_t
34 ngx_atomic_cmp_set(ngx_atomic_t *lock, ngx_atomic_uint_t old,
35 ngx_atomic_uint_t set)
36 {
37     __asm__ volatile (
38
39         NGX_CASA " [%1] 0x80, %2, %0"
40
41         : "+r" (set) : "r" (lock), "r" (old) : "memory");
42
43     return (set == old);
44 }
45
```

```

46
47 static ngx_inline ngx_atomic_int_t
48 ngx_atomic_fetch_add(ngx_atomic_t *value, ngx_atomic_int_t add)
49 {
50     ngx_atomic_uint_t  old, res;
51
52     old = *value;
53
54     for ( ;; ) {
55
56         res = old + add;
57
58         __asm__ volatile (
59
60             NGX_CASA " [%1] 0x80, %2, %0"
61
62             : "+r" (res) : "r" (value), "r" (old) : "memory");
63
64         if (res == old) {
65             return res;
66         }
67
68         old = res;
69     }
70 }
71
72
73 #if (NGX_SMP)
74 #define ngx_memory_barrier() \
75     __asm__ volatile ( \
76         "membar #LoadLoad | #LoadStore | #StoreStore | #StoreLoad" \
77         ::: "memory")
78 #else
79 #define ngx_memory_barrier() __asm__ volatile (" ::: "memory")
80 #endif
81
82 #define ngx_cpu_pause()

```

[One Level Up](#)

[Top Level](#)

## src/os/unix/nginx\_freebsd\_rfork\_thread.h - nginx-1.7.10

### Data types defined

- [ngx\\_cond\\_t](#)
- [ngx\\_mutex\\_t](#)
- [ngx\\_rfork\\_thread\\_func\\_pt](#)
- [ngx\\_tid\\_t](#)
- [ngx\\_tls\\_key\\_t](#)

### Functions defined

- [ngx\\_gettid](#)
- [ngx\\_thread\\_get\\_tls](#)

### Macros defined

- [NGX CV SIGNAL](#)
- [NGX MUTEX LIGHT](#)
- [NGX MUTEX LOCK BUSY](#)
- [NGX THREAD KEYS MAX](#)
- [NGX TID T FMT](#)
- [\\_NGX\\_FREEBSD\\_RFORK\\_THREAD\\_H\\_INCLUDED](#)
- [ngx\\_log\\_pid](#)
- [ngx\\_log\\_tid](#)
- [ngx\\_mutex\\_lock](#)
- [ngx\\_mutex\\_trylock](#)
- [ngx\\_setthrtitle](#)
- [ngx\\_thread\\_join](#)
- [ngx\\_thread\\_key\\_create\\_n](#)
- [ngx\\_thread\\_set\\_tls\\_n](#)
- [ngx\\_thread\\_sigmask](#)
- [ngx\\_thread\\_sigmask\\_n](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
```



```

6
7
8 #ifndef NGX_FREEBSD_RFORK_THREAD_H_INCLUDED
9 #define NGX_FREEBSD_RFORK_THREAD_H_INCLUDED
10
11
12 #include <sys/ipc.h>
13 #include <sys/sem.h>
14 #include <sched.h>
15
16 typedef pid_t ngx_tid_t;
17
18 #define ngx_log_pid ngx_thread_self()
19 #define ngx_log_tid 0
20
21 #define NGX_TID_T_FMT "%P"
22
23
24 #define NGX_MUTEX_LIGHT 1
25
26 #define NGX_MUTEX_LOCK_BUSY 0x80000000
27
28 typedef volatile struct {
29     ngx_atomic_t lock;
30     ngx_log_t *log;
31     int semid;
32 } ngx_mutex_t;
33
34
35 #define NGX_CV_SIGNAL 64
36
37 typedef struct {
38     int signo;
39     int kq;
40     ngx_tid_t tid;
41     ngx_log_t *log;
42 } ngx_cond_t;
43
44
45 #define ngx_thread_sigmask(how, set, oset) \
46     (sigprocmask(how, set, oset) == -1) ? ngx_errno : 0
47
48 #define ngx_thread_sigmask_n "sigprocmask()"
49
50 #define ngx_thread_join(t, p)
51
52 #define ngx_setthrtitle(n) setproctitle(n)
53
54
55 extern char *ngx_freebsd_kern_usrstack;
56 extern size_t ngx_thread_stack_size;
57
58
59 static ngx_inline ngx_int_t
60 ngx_gettid(void)
61 {
62     char *sp;
63
64     if (ngx_thread_stack_size == 0) {
65         return 0;
66     }
67
68     #if ( __i386__ )
69
70     __asm__ volatile ("mov %%esp, %0" : "=q" (sp));
71
72     #elif ( __amd64__ )
73
74     __asm__ volatile ("mov %%rsp, %0" : "=q" (sp));
75
76     #else
77
78     #error "rfork()ed threads are not supported on this platform"
79
80     #endif
81

```

```

82     return (ngx\_freebsd kern usrstack - sp) / ngx\_thread\_stack\_size;
83 }
84
85
86 ngx\_tid\_t ngx\_thread\_self(void);
87
88
89 typedef ngx\_uint\_t          ngx\_tls\_key\_t;
90
91 #define NGX\_THREAD\_KEYS\_MAX    16
92
93 extern void    **ngx\_tls;
94
95 ngx\_err\_t ngx\_thread\_key\_create(ngx\_tls\_key\_t *key);
96 #define ngx\_thread\_key\_create\_n  "the tls key creation"
97
98 ngx\_err\_t ngx\_thread\_set\_tls(ngx\_tls\_key\_t key, void *value);
99 #define ngx\_thread\_set\_tls\_n     "the tls key setting"
100
101
102 static void *
103 ngx\_thread\_get\_tls(ngx\_tls\_key\_t key)
104 {
105     if (key >= NGX\_THREAD\_KEYS\_MAX) {
106         return NULL;
107     }
108
109     return ngx\_tls[key * NGX\_THREAD\_KEYS\_MAX + ngx\_gettid()];
110 }
111
112
113 #define ngx\_mutex\_trylock(m) ngx\_mutex\_dolock(m, 1)
114 #define ngx\_mutex\_lock(m)      (void) ngx\_mutex\_dolock(m, 0)
115 ngx\_int\_t ngx\_mutex\_dolock(ngx\_mutex\_t *m, ngx\_int\_t try);
116 void ngx\_mutex\_unlock(ngx\_mutex\_t *m);
117
118
119 typedef int (*ngx\_rfork\_thread\_func\_pt)(void *arg);
120
121
122 #endif /* \_NGX\_FREEBSD\_RFORK\_THREAD\_H\_INCLUDED */

```

[One Level Up](#)

[Top Level](#)

# src/os/unix/nginx\_posix\_init.c - nginx-1.7.10

## Global variables defined

- [ngx\\_inherited\\_nonblocking](#)
- [ngx\\_max\\_sockets](#)
- [ngx\\_ncpu](#)
- [ngx\\_os\\_io](#)
- [ngx\\_tcp\\_nodelay\\_and\\_tcp\\_nopush](#)
- [rlmt](#)

## Functions defined

- [ngx\\_os\\_init](#)
- [ngx\\_os\\_status](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <nginx.h>
11
12
13 ngx_int_t    ngx_ncpu;
14 ngx_int_t    ngx_max_sockets;
15 ngx_uint_t   ngx_inherited_nonblocking;
16 ngx_uint_t   ngx_tcp_nodelay_and_tcp_nopush;
17
18
19 struct rlimit  rlmt;
20
21
22 ngx_os_io_t  ngx_os_io = {
23     ngx_unix_recv,
24     ngx_readv_chain,
25     ngx_udp_unix_recv,
26     ngx_unix_send,
27     ngx_writev_chain,
28     0
29 };
30
31
32 ngx_int_t
33 ngx_os_init(ngx_log_t *log)
34 {
35     ngx_uint_t  n;
36
37     #if (NGX_HAVE_OS_SPECIFIC_INIT)
38         if (ngx_os_specific_init(log) != NGX_OK) {
39             return NGX_ERROR;
40         }
41     #endif
42
43     if (ngx_init_setproctitle(log) != NGX_OK) {
```

```

44     return NGX_ERROR;
45 }
46
47 ngx_pagesize = getpagesize();
48 ngx_cacheline_size = NGX_CPU_CACHE_LINE;
49
50 for (n = ngx_pagesize; n >= 1; ngx_pagesize_shift++) { /* void */ }
51
52 #if (NGX_HAVE_SC_NPROCESSORS_ONLN)
53     if (ngx_ncpu == 0) {
54         ngx_ncpu = sysconf(_SC_NPROCESSORS_ONLN);
55     }
56 #endif
57
58     if (ngx_ncpu < 1) {
59         ngx_ncpu = 1;
60     }
61
62     ngx_cpuinfo();
63
64     if (getrlimit(RLIMIT_NOFILE, &rlmt) == -1) {
65         ngx_log_error(NGX_LOG_ALERT, log, errno,
66             "getrlimit(RLIMIT_NOFILE) failed");
67         return NGX_ERROR;
68     }
69
70     ngx_max_sockets = (ngx_int_t) rlmt.rlim_cur;
71
72 #if (NGX_HAVE_INHERITED_NONBLOCK || NGX_HAVE_ACCEPT4)
73     ngx_inherited_nonblocking = 1;
74 #else
75     ngx_inherited_nonblocking = 0;
76 #endif
77
78     srand(ngx_time());
79
80     return NGX_OK;
81 }
82
83
84 void
85 ngx_os_status(ngx_log_t *log)
86 {
87     ngx_log_error(NGX_LOG_NOTICE, log, 0, NGINX_VER_BUILD);
88
89 #ifdef NGX_COMPILER
90     ngx_log_error(NGX_LOG_NOTICE, log, 0, "built by " NGX_COMPILER);
91 #endif
92
93 #if (NGX_HAVE_OS_SPECIFIC_INIT)
94     ngx_os_specific_status(log);
95 #endif
96
97     ngx_log_error(NGX_LOG_NOTICE, log, 0,
98         "getrlimit(RLIMIT_NOFILE): %r:%r",
99         rlmt.rlim_cur, rlmt.rlim_max);
100 }
101
102
103 #if 0
104
105 ngx_int_t
106 ngx_posix_post_conf_init(ngx_log_t *log)
107 {
108     ngx_fd_t pp[2];
109
110     if (pipe(pp) == -1) {
111         ngx_log_error(NGX_LOG_EMERG, log, ngx_errno, "pipe() failed");
112         return NGX_ERROR;
113     }
114
115     if (dup2(pp[1], STDERR_FILENO) == -1) {
116         ngx_log_error(NGX_LOG_EMERG, log, errno, "dup2(STDERR) failed");
117         return NGX_ERROR;
118     }
119

```

```
120     if (pp[1] > STDERR_FILENO) {
121         if (close(pp[1]) == -1) {
122             ngx_log_error(NGX_LOG_EMERG, log, errno, "close() failed");
123             return NGX_ERROR;
124         }
125     }
126
127     return NGX_OK;
128 }
129
130 #endif
```

[One Level Up](#)

[Top Level](#)

## src/os/unix/nginx\_os.h - nginx-1.7.10

### Data types defined

- [ngx\\_iovec\\_t](#)
- [ngx\\_os\\_io\\_t](#)
- [ngx\\_recv\\_chain\\_pt](#)
- [ngx\\_recv\\_pt](#)
- [ngx\\_send\\_chain\\_pt](#)
- [ngx\\_send\\_pt](#)

### Macros defined

- [NGX\\_IOVS\\_PREALLOCATE](#)
- [NGX\\_IOVS\\_PREALLOCATE](#)
- [NGX\\_IO\\_SENDFILE](#)
- [\\_NGX\\_OS\\_H\\_INCLUDED\\_](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_OS_H_INCLUDED_
9 #define _NGX_OS_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 #define NGX_IO_SENDFILE    1
17
18
19 typedef ssize_t (*ngx_recv_pt)(ngx_connection_t *c, u_char *buf, size_t size);
20 typedef ssize_t (*ngx_recv_chain_pt)(ngx_connection_t *c, ngx_chain_t *in,
21     off_t limit);
22 typedef ssize_t (*ngx_send_pt)(ngx_connection_t *c, u_char *buf, size_t size);
23 typedef ngx_chain_t *(*ngx_send_chain_pt)(ngx_connection_t *c, ngx_chain_t *in,
24     off_t limit);
25
26 typedef struct {
27     ngx_recv_pt      recv;
28     ngx_recv_chain_pt  recv_chain;
29     ngx_recv_pt      udp_recv;
30     ngx_send_pt      send;
31     ngx_send_chain_pt  send_chain;
32     ngx_uint_t       flags;
33 } ngx_os_io_t;
34
35
36 ngx_int_t ngx_os_init(ngx_log_t *log);
37 void ngx_os_status(ngx_log_t *log);
38 ngx_int_t ngx_os_specific_init(ngx_log_t *log);
```

```

39 void ngx_os_specific_status(ngx_log_t *log);
40 ngx_int_t ngx_daemon(ngx_log_t *log);
41 ngx_int_t ngx_os_signal_process(ngx_cycle_t *cycle, char *sig, ngx_int_t pid);
42
43
44 ssize_t ngx_unix_recv(ngx_connection_t *c, u_char *buf, size_t size);
45 ssize_t ngx_readv_chain(ngx_connection_t *c, ngx_chain_t *entry, off_t limit);
46 ssize_t ngx_udp_unix_recv(ngx_connection_t *c, u_char *buf, size_t size);
47 ssize_t ngx_unix_send(ngx_connection_t *c, u_char *buf, size_t size);
48 ngx_chain_t *ngx_writev_chain(ngx_connection_t *c, ngx_chain_t *in,
49     off_t limit);
50
51 #if (NGX_HAVE_AIO)
52 ssize_t ngx_aio_read(ngx_connection_t *c, u_char *buf, size_t size);
53 ssize_t ngx_aio_read_chain(ngx_connection_t *c, ngx_chain_t *cl, off_t limit);
54 ssize_t ngx_aio_write(ngx_connection_t *c, u_char *buf, size_t size);
55 ngx_chain_t *ngx_aio_write_chain(ngx_connection_t *c, ngx_chain_t *in,
56     off_t limit);
57 #endif
58
59
60 #if (IOV_MAX > 64)
61 #define NGX_IOVS_PREALLOCATE 64
62 #else
63 #define NGX_IOVS_PREALLOCATE IOV_MAX
64 #endif
65
66
67 typedef struct {
68     struct iovec *iovs;
69     ngx_uint_t count;
70     size_t size;
71     ngx_uint_t nalloc;
72 } ngx_iovec_t;
73
74 ngx_chain_t *ngx_output_chain_to_iovec(ngx_iovec_t *vec, ngx_chain_t *in,
75     size_t limit, ngx_log_t *log);
76
77
78 ssize_t ngx_writev(ngx_connection_t *c, ngx_iovec_t *vec);
79
80
81 extern ngx_os_io_t ngx_os_io;
82 extern ngx_int_t ngx_ncpu;
83 extern ngx_int_t ngx_max_sockets;
84 extern ngx_uint_t ngx_inherited_nonblocking;
85 extern ngx_uint_t ngx_tcp_nodelay_and_tcp_nopush;
86
87
88 #if (NGX_FREEBSD)
89 #include <ngx_freebsd.h>
90
91
92 #elif (NGX_LINUX)
93 #include <ngx_linux.h>
94
95
96 #elif (NGX_SOLARIS)
97 #include <ngx_solaris.h>
98
99
100 #elif (NGX_DARWIN)
101 #include <ngx_darwin.h>
102 #endif
103
104
105 #endif /* NGX_OS_H_INCLUDED */

```

[One Level Up](#)

[Top Level](#)

# src/os/unix/nginx\_freebsd\_init.c - nginx-1.7.10

## Global variables defined

- [ngx\\_debug\\_malloc](#)
- [ngx\\_freebsd\\_hw\\_ncpu](#)
- [ngx\\_freebsd\\_io](#)
- [ngx\\_freebsd\\_kern\\_ipc\\_somaxconn](#)
- [ngx\\_freebsd\\_kern\\_osreldate](#)
- [ngx\\_freebsd\\_kern\\_osrelease](#)
- [ngx\\_freebsd\\_kern\\_ostype](#)
- [ngx\\_freebsd\\_machdep\\_hlt\\_logical\\_cpus](#)
- [ngx\\_freebsd\\_net\\_inet\\_tcp\\_sendspace](#)
- [ngx\\_freebsd\\_sendfile\\_nbytes\\_bug](#)
- [ngx\\_freebsd\\_use\\_tcp\\_nopush](#)
- [sysctls](#)

## Data types defined

- [sysctl\\_t](#)

## Functions defined

- [ngx\\_debug\\_init](#)
- [ngx\\_os\\_specific\\_init](#)
- [ngx\\_os\\_specific\\_status](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10
11
12 /* FreeBSD 3.0 at least */
13 char    ngx_freebsd_kern_ostype[16];
14 char    ngx_freebsd_kern_osrelease[128];
15 int     ngx_freebsd_kern_osreldate;
16 int     ngx_freebsd_hw_ncpu;
17 int     ngx_freebsd_kern_ipc_somaxconn;
18 u_long  ngx_freebsd_net_inet_tcp_sendspace;
19
20 /* FreeBSD 4.9 */
21 int     ngx_freebsd_machdep_hlt_logical_cpus;
```



```

22
23
24 ngx\_uint\_t ngx_freebsd_sendfile_nbytes_bug;
25 ngx\_uint\_t ngx_freebsd_use_tcp_nopush;
26
27 ngx\_uint\_t ngx_debug_malloc;
28
29
30 static ngx\_os\_io\_t ngx_freebsd_io = {
31     ngx\_unix\_recv,
32     ngx\_readv\_chain,
33     ngx\_udp\_unix\_recv,
34     ngx\_unix\_send,
35     #if (NGX_HAVE_SENDFILE)
36         ngx\_freebsd\_sendfile\_chain,
37         NGX\_IO\_SENDFILE
38     #else
39         ngx\_writev\_chain,
40         0
41     #endif
42 };
43
44
45 typedef struct {
46     char      *name;
47     void      *value;
48     size_t    size;
49     ngx\_uint\_t exists;
50 } sysctl_t;
51
52
53 sysctl\_t sysctls[] = {
54     { "hw.ncpu",
55       &ngx\_freebsd\_hw\_ncpu,
56       sizeof(ngx\_freebsd\_hw\_ncpu), 0 },
57
58     { "machdep.hlt_logical_cpus",
59       &ngx\_freebsd\_machdep\_hlt\_logical\_cpus,
60       sizeof(ngx\_freebsd\_machdep\_hlt\_logical\_cpus), 0 },
61
62     { "net.inet.tcp.sendspace",
63       &ngx\_freebsd\_net\_inet\_tcp\_sendspace,
64       sizeof(ngx\_freebsd\_net\_inet\_tcp\_sendspace), 0 },
65
66     { "kern.ipc.somaxconn",
67       &ngx\_freebsd\_kern\_ipc\_somaxconn,
68       sizeof(ngx\_freebsd\_kern\_ipc\_somaxconn), 0 },
69
70     { NULL, NULL, 0, 0 }
71 };
72
73
74 void
75 ngx_debug_init(void)
76 {
77     #if (NGX_DEBUG_MALLOC)
78
79     #if \_\_FreeBSD\_version >= 500014 && \_\_FreeBSD\_version < 1000011
80         _malloc_options = "J";
81     #elif \_\_FreeBSD\_version < 500014
82         malloc_options = "J";
83     #endif
84
85     ngx\_debug\_malloc = 1;
86
87 #else
88     char *mo;
89
90     mo = getenv("MALLOC_OPTIONS");
91
92     if (mo && ngx\_strchr(mo, 'J')) {
93         ngx\_debug\_malloc = 1;
94     }
95 #endif
96 }
97

```

```

98
99 ngx_int_t
100 ngx_os_specific_init(ngx_log_t *log)
101 {
102     int         version;
103     size_t      size;
104     ngx_err_t   err;
105     ngx_uint_t  i;
106
107     size = sizeof(ngx_freebsd_kern_ostype);
108     if (sysctlbyname("kern.ostype",
109         ngx_freebsd_kern_ostype, &size, NULL, 0) == -1) {
110         ngx_log_error(NGX_LOG_ALERT, log, ngx_errno,
111             "sysctlbyname(kern.ostype) failed");
112
113         if (ngx_errno != NGX_ENOMEM) {
114             return NGX_ERROR;
115         }
116
117         ngx_freebsd_kern_ostype[size - 1] = '\0';
118     }
119
120     size = sizeof(ngx_freebsd_kern_osrelease);
121     if (sysctlbyname("kern.osrelease",
122         ngx_freebsd_kern_osrelease, &size, NULL, 0) == -1) {
123         ngx_log_error(NGX_LOG_ALERT, log, ngx_errno,
124             "sysctlbyname(kern.osrelease) failed");
125
126         if (ngx_errno != NGX_ENOMEM) {
127             return NGX_ERROR;
128         }
129
130         ngx_freebsd_kern_osrelease[size - 1] = '\0';
131     }
132
133
134     size = sizeof(int);
135     if (sysctlbyname("kern.osreldate",
136         &ngx_freebsd_kern_osreldate, &size, NULL, 0) == -1) {
137         ngx_log_error(NGX_LOG_ALERT, log, ngx_errno,
138             "sysctlbyname(kern.osreldate) failed");
139         return NGX_ERROR;
140     }
141
142     version = ngx_freebsd_kern_osreldate;
143
144
145     #if (NGX_HAVE_SENDFILE)
146
147     /*
148      * The determination of the sendfile() "nbytes bug" is complex enough.
149      * There are two sendfile() syscalls: a new #393 has no bug while
150      * an old #336 has the bug in some versions and has not in others.
151      * Besides libc_r wrapper also emulates the bug in some versions.
152      * There is no way to say exactly if syscall #336 in FreeBSD circa 4.6
153      * has the bug. We use the algorithm that is correct at least for
154      * RELEASEs and for syscalls only (not libc_r wrapper).
155      *
156      * 4.6.1-RELEASE and below have the bug
157      * 4.6.2-RELEASE and above have the new syscall
158      *
159      * We detect the new sendfile() syscall available at the compile time
160      * to allow an old binary to run correctly on an updated FreeBSD system.
161      */
162
163     #if ( __FreeBSD__ == 4 && __FreeBSD_version >= 460102 ) \
164         || __FreeBSD_version == 460002 || __FreeBSD_version >= 500039
165
166         /* a new syscall without the bug */
167
168         ngx_freebsd_sendfile_nbytes_bug = 0;
169
170     #else
171
172         /* an old syscall that may have the bug */
173

```

```

174     ngx_freebsd_sendfile_nbytes_bug = 1;
175
176 #endif
177
178 #endif /* NGX_HAVE_SENDFILE */
179
180
181     if ((version < 500000 && version >= 440003) || version >= 500017) {
182         ngx_freebsd_use_tcp_nopush = 1;
183     }
184
185     for (i = 0; sysctls[i].name; i++) {
186         size = sysctls[i].size;
187
188         if (sysctlbyname(sysctls[i].name, sysctls[i].value, &size, NULL, 0)
189             == 0)
190         {
191             sysctls[i].exists = 1;
192             continue;
193         }
194
195         err = ngx_errno;
196
197         if (err == NGX_ENOENT) {
198             continue;
199         }
200     }
201
202     ngx_log_error(NGX_LOG_ALERT, log, err,
203                 "sysctlbyname(%s) failed", sysctls[i].name);
204     return NGX_ERROR;
205 }
206
207 if (ngx_freebsd_machdep_hlt_logical_cpus) {
208     ngx_ncpu = ngx_freebsd_hw_ncpu / 2;
209 }
210 else {
211     ngx_ncpu = ngx_freebsd_hw_ncpu;
212 }
213
214 if (version < 600008 && ngx_freebsd_kern_ipc_somaxconn > 32767) {
215     ngx_log_error(NGX_LOG_ALERT, log, 0,
216                 "sysctl kern.ipc.somaxconn must be less than 32768");
217     return NGX_ERROR;
218 }
219
220 ngx_tcp_nodelay_and_tcp_nopush = 1;
221
222 ngx_os_io = ngx_freebsd_io;
223
224 return NGX_OK;
225 }
226
227
228 void
229 ngx_os_specific_status(ngx_log_t *log)
230 {
231     u_long    value;
232     ngx_uint_t i;
233
234     ngx_log_error(NGX_LOG_NOTICE, log, 0, "OS: %s %s",
235                 ngx_freebsd_kern_osrdate, ngx_freebsd_kern_osrelease);
236
237 #ifdef __DragonFly_version
238     ngx_log_error(NGX_LOG_NOTICE, log, 0,
239                 "kern.osreldate: %d, built on %d",
240                 ngx_freebsd_kern_osreldate, __DragonFly_version);
241 #else
242     ngx_log_error(NGX_LOG_NOTICE, log, 0,
243                 "kern.osreldate: %d, built on %d",
244                 ngx_freebsd_kern_osreldate, __FreeBSD_version);
245 #endif
246
247     for (i = 0; sysctls[i].name; i++) {
248         if (sysctls[i].exists) {
249             if (sysctls[i].size == sizeof(long)) {

```

```
250         value = *(long *) sysctls[i].value;
251
252     } else {
253         value = *(int *) sysctls[i].value;
254     }
255
256     ngx\_log\_error(NGX\_LOG\_NOTICE, log, 0, "%s: %l",
257                 sysctls[i].name, value);
258 }
259 }
260 }
```

[One Level Up](#)

[Top Level](#)

## src/os/unix/nginx\_recv.c - nginx-1.7.10

### Functions defined

- [ngx\\_unix\\_recv](#)
- [ngx\\_unix\\_recv](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 #if (NGX_HAVE_KQUEUE)
14
15 ssize_t
16 ngx_unix_recv(ngx_connection_t *c, u_char *buf, size_t size)
17 {
18     ssize_t    n;
19     ngx_err_t  err;
20     ngx_event_t *rev;
21
22     rev = c->read;
23
24     if (ngx_event_flags & NGX_USE_KQUEUE_EVENT) {
25         ngx_log_debug3(NGX_LOG_DEBUG_EVENT, c->log, 0,
26             "recv: eof:%d, avail:%d, err:%d",
27             rev->pending_eof, rev->available, rev->kq_errno);
28
29         if (rev->available == 0) {
30             if (rev->pending_eof) {
31                 rev->ready = 0;
32                 rev->eof = 1;
33
34                 if (rev->kq_errno) {
35                     rev->error = 1;
36                     ngx_set_socket_errno(rev->kq_errno);
37
38                     return ngx_connection_error(c, rev->kq_errno,
39                         "kevent() reported about an closed connection");
40                 }
41
42                 return 0;
43
44             } else {
45                 rev->ready = 0;
46                 return NGX_AGAIN;
47             }
48         }
49     }
50
51     do {
52         n = recv(c->fd, buf, size, 0);
53
54         ngx_log_debug3(NGX_LOG_DEBUG_EVENT, c->log, 0,
55             "recv: fd:%d %d of %d", c->fd, n, size);
56
57         if (n >= 0) {
58             if (ngx_event_flags & NGX_USE_KQUEUE_EVENT) {
59                 rev->available -= n;
60
```

```

61      /*
62      * rev->available may be negative here because some additional
63      * bytes may be received between kevent() and recv()
64      */
65
66      if (rev->available <= 0) {
67          if (!rev->pending_eof) {
68              rev->ready = 0;
69          }
70
71          if (rev->available < 0) {
72              rev->available = 0;
73          }
74      }
75
76      if (n == 0) {
77
78          /*
79          * on FreeBSD recv() may return 0 on closed socket
80          * even if kqueue reported about available data
81          */
82
83          rev->ready = 0;
84          rev->eof = 1;
85          rev->available = 0;
86      }
87
88      return n;
89  }
90
91  if ((size_t) n < size
92      && !(ngx_event_flags & NGX_USE_GREEDY_EVENT))
93  {
94      rev->ready = 0;
95  }
96
97  if (n == 0) {
98      rev->eof = 1;
99  }
100
101      return n;
102  }
103
104  err = ngx_socket_errno;
105
106  if (err == NGX_EAGAIN || err == NGX_EINTR) {
107      ngx_log_debug0(NGX_LOG_DEBUG_EVENT, c->log, err,
108                    "recv() not ready");
109      n = NGX_AGAIN;
110  } else {
111      n = ngx_connection_error(c, err, "recv() failed");
112      break;
113  }
114  }
115
116  } while (err == NGX_EINTR);
117
118  rev->ready = 0;
119
120  if (n == NGX_ERROR) {
121      rev->error = 1;
122  }
123
124  return n;
125 }
126
127 #else /* ! NGX_HAVE_KQUEUE */
128
129 ssize_t
130 ngx_unix_recv(ngx_connection_t *c, u_char *buf, size_t size)
131 {
132     ssize_t    n;
133     ngx_err_t  err;
134     ngx_event_t *rev;
135
136     rev = c->read;

```

```

137
138 do {
139     n = recv(c->fd, buf, size, 0);
140
141     ngx_log_debug3(NGX_LOG_DEBUG_EVENT, c->log, 0,
142                  "recv: fd:%d %d of %d", c->fd, n, size);
143
144     if (n == 0) {
145         rev->ready = 0;
146         rev->eof = 1;
147         return n;
148
149     } else if (n > 0) {
150
151         if ((size_t) n < size
152             && !(ngx_event_flags & NGX_USE_GREEDY_EVENT))
153         {
154             rev->ready = 0;
155         }
156
157         return n;
158     }
159
160     err = ngx_socket_errno;
161
162     if (err == NGX_EAGAIN || err == NGX_EINTR) {
163         ngx_log_debug0(NGX_LOG_DEBUG_EVENT, c->log, err,
164                       "recv() not ready");
165         n = NGX_AGAIN;
166
167     } else {
168         n = ngx_connection_error(c, err, "recv() failed");
169         break;
170     }
171
172 } while (err == NGX_EINTR);
173
174 rev->ready = 0;
175
176 if (n == NGX_ERROR) {
177     rev->error = 1;
178 }
179
180 return n;
181 }
182
183 #endif /* NGX_HAVE_KQUEUE */

```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_connection.c - nginx-1.7.10

### Global variables defined

- [ngx\\_io](#)

### Functions defined

- [ngx\\_close\\_connection](#)
- [ngx\\_close\\_listening\\_sockets](#)
- [ngx\\_configure\\_listening\\_sockets](#)
- [ngx\\_connection\\_error](#)
- [ngx\\_connection\\_local\\_sockaddr](#)
- [ngx\\_create\\_listening](#)
- [ngx\\_drain\\_connections](#)
- [ngx\\_free\\_connection](#)
- [ngx\\_get\\_connection](#)
- [ngx\\_open\\_listening\\_sockets](#)
- [ngx\\_reusable\\_connection](#)
- [ngx\\_set\\_inherited\\_sockets](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 ngx_os_io_t  ngx_io;
14
15
16 static void ngx_drain_connections(void);
17
18
19 ngx_listening_t *
20 ngx_create_listening(ngx_conf_t *cf, void *sockaddr, socklen_t socklen)
21 {
22     size_t      len;
23     ngx_listening_t *ls;
24     struct sockaddr *sa;
25     u_char      text[NGX_SOCKADDR_STRLEN];
26
27     ls = ngx_array_push(&cf->cycle->listening);
28     if (ls == NULL) {
29         return NULL;
30     }
31
32     ngx_memzero(ls, sizeof(ngx_listening_t));
```



```

33
34 sa = ngx_palloc(cf->pool, socklen);
35 if (sa == NULL) {
36     return NULL;
37 }
38
39 ngx_memcpy(sa, sockaddr, socklen);
40
41 ls->sockaddr = sa;
42 ls->socklen = socklen;
43
44 len = ngx_sock_ntop(sa, socklen, text, NGX_SOCKADDR_STRLEN, 1);
45 ls->addr_text.len = len;
46
47 switch (ls->sockaddr->sa_family) {
48 #if (NGX_HAVE_INET6)
49     case AF_INET6:
50         ls->addr_text_max_len = NGX_INET6_ADDRSTRLEN;
51         break;
52 #endif
53 #if (NGX_HAVE_UNIX_DOMAIN)
54     case AF_UNIX:
55         ls->addr_text_max_len = NGX_UNIX_ADDRSTRLEN;
56         len++;
57         break;
58 #endif
59     case AF_INET:
60         ls->addr_text_max_len = NGX_INET_ADDRSTRLEN;
61         break;
62     default:
63         ls->addr_text_max_len = NGX_SOCKADDR_STRLEN;
64         break;
65 }
66
67 ls->addr_text.data = ngx_pnalloc(cf->pool, len);
68 if (ls->addr_text.data == NULL) {
69     return NULL;
70 }
71
72 ngx_memcpy(ls->addr_text.data, text, len);
73
74 ls->fd = (ngx_socket_t) -1;
75 ls->type = SOCK_STREAM;
76
77 ls->backlog = NGX_LISTEN_BACKLOG;
78 ls->rcvbuf = -1;
79 ls->sndbuf = -1;
80
81 #if (NGX_HAVE_SETFIB)
82     ls->setfib = -1;
83 #endif
84
85 #if (NGX_HAVE_TCP_FASTOPEN)
86     ls->fastopen = -1;
87 #endif
88
89     return ls;
90 }
91
92
93 ngx_int_t
94 ngx_set_inherited_sockets(ngx_cycle_t *cycle)
95 {
96     size_t          len;
97     ngx_uint_t      i;
98     ngx_listening_t *ls;
99     socklen_t       olen;
100 #if (NGX_HAVE_DEFERRED_ACCEPT || NGX_HAVE_TCP_FASTOPEN)
101     ngx_err_t       err;
102 #endif
103 #if (NGX_HAVE_DEFERRED_ACCEPT && defined SO_ACCEPTFILTER)
104     struct accept_filter_arg af;
105 #endif
106 #if (NGX_HAVE_DEFERRED_ACCEPT && defined TCP_DEFER_ACCEPT)
107     int             timeout;
108 #endif

```

```

109 ls = cycle->listening.elts;
110 for (i = 0; i < cycle->listening.nelts; i++) {
111     ls[i].sockaddr = ngx_palloc(cycle->pool, NGX_SOCKADDRLEN);
112     if (ls[i].sockaddr == NULL) {
113         return NGX_ERROR;
114     }
115
116     ls[i].socklen = NGX_SOCKADDRLEN;
117     if (getsockname(ls[i].fd, ls[i].sockaddr, &ls[i].socklen) == -1) {
118         ngx_log_error(NGX_LOG_CRIT, cycle->log, ngx_socket_errno,
119             "getsockname() of the inherited "
120             "socket ##d failed", ls[i].fd);
121     }
122     ls[i].ignore = 1;
123     continue;
124 }
125
126 switch (ls[i].sockaddr->sa_family) {
127
128 #if (NGX_HAVE_INET6)
129     case AF_INET6:
130         ls[i].addr_text_max_len = NGX_INET6_ADDRSTRLEN;
131         len = NGX_INET6_ADDRSTRLEN + sizeof("[:65535]") - 1;
132         break;
133 #endif
134
135 #if (NGX_HAVE_UNIX_DOMAIN)
136     case AF_UNIX:
137         ls[i].addr_text_max_len = NGX_UNIX_ADDRSTRLEN;
138         len = NGX_UNIX_ADDRSTRLEN;
139         break;
140 #endif
141
142     case AF_INET:
143         ls[i].addr_text_max_len = NGX_INET_ADDRSTRLEN;
144         len = NGX_INET_ADDRSTRLEN + sizeof(":65535") - 1;
145         break;
146
147     default:
148         ngx_log_error(NGX_LOG_CRIT, cycle->log, ngx_socket_errno,
149             "the inherited socket ##d has "
150             "an unsupported protocol family", ls[i].fd);
151     }
152     ls[i].ignore = 1;
153     continue;
154 }
155
156 ls[i].addr_text.data = ngx_pnalloc(cycle->pool, len);
157 if (ls[i].addr_text.data == NULL) {
158     return NGX_ERROR;
159 }
160
161 len = ngx_sock_ntop(ls[i].sockaddr, ls[i].socklen,
162     ls[i].addr_text.data, len, 1);
163 if (len == 0) {
164     return NGX_ERROR;
165 }
166
167 ls[i].addr_text.len = len;
168
169 ls[i].backlog = NGX_LISTEN_BACKLOG;
170
171 olen = sizeof(int);
172
173 if (getsockopt(ls[i].fd, SOL_SOCKET, SO_RCVBUF, (void *) &ls[i].rcvbuf,
174     &olen)
175     == -1)
176 {
177     ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_socket_errno,
178         "getsockopt(SO_RCVBUF) %V failed, ignored",
179         &ls[i].addr_text);
180
181     ls[i].rcvbuf = -1;
182 }
183
184 olen = sizeof(int);

```

```

185
186     if (getsockopt(ls[i].fd, SOL_SOCKET, SO_SNDBUF, (void *) &ls[i].sndbuf,
187                 &olen)
188         == -1)
189     {
190         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_socket_errno,
191                     "getsockopt(SO_SNDBUF) %V failed, ignored",
192                     &ls[i].addr_text);
193
194         ls[i].sndbuf = -1;
195     }
196
197     #if 0
198     /* SO_SETFIB is currently a set only option */
199
200     #if (NGX_HAVE_SETFIB)
201
202         olen = sizeof(int);
203
204         if (getsockopt(ls[i].fd, SOL_SOCKET, SO_SETFIB,
205                     (void *) &ls[i].setfib, &olen)
206             == -1)
207         {
208             ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_socket_errno,
209                         "getsockopt(SO_SETFIB) %V failed, ignored",
210                         &ls[i].addr_text);
211
212             ls[i].setfib = -1;
213         }
214
215     #endif
216     #endif
217
218     #if (NGX_HAVE_TCP_FASTOPEN)
219
220         olen = sizeof(int);
221
222         if (getsockopt(ls[i].fd, IPPROTO_TCP, TCP_FASTOPEN,
223                     (void *) &ls[i].fastopen, &olen)
224             == -1)
225         {
226             err = ngx_socket_errno;
227
228             if (err != NGX_EOPNOTSUPP && err != NGX_ENOPROTOOPT) {
229                 ngx_log_error(NGX_LOG_NOTICE, cycle->log, err,
230                             "getsockopt(TCP_FASTOPEN) %V failed, ignored",
231                             &ls[i].addr_text);
232             }
233
234             ls[i].fastopen = -1;
235         }
236
237     #endif
238
239     #if (NGX_HAVE_DEFERRED_ACCEPT && defined SO_ACCEPTFILTER)
240
241         ngx_memzero(&af, sizeof(struct accept_filter_arg));
242         olen = sizeof(struct accept_filter_arg);
243
244         if (getsockopt(ls[i].fd, SOL_SOCKET, SO_ACCEPTFILTER, &af, &olen)
245             == -1)
246         {
247             err = ngx_socket_errno;
248
249             if (err == NGX_EINVAL) {
250                 continue;
251             }
252
253             ngx_log_error(NGX_LOG_NOTICE, cycle->log, err,
254                         "getsockopt(SO_ACCEPTFILTER) for %V failed, ignored",
255                         &ls[i].addr_text);
256
257             continue;
258         }
259
260         if (olen < sizeof(struct accept_filter_arg) || af.af_name[0] == '\0') {
261             continue;
262         }

```

```

261     }
262
263     ls[i].accept_filter = ngx\_palloc(cycle->pool, 16);
264     if (ls[i].accept_filter == NULL) {
265         return NGX\_ERROR;
266     }
267
268     (void) ngx\_cpystn((u_char *) ls[i].accept_filter,
269                     (u_char *) af.af_name, 16);
270 #endif
271
272 #if (NGX_HAVE_DEFERRED_ACCEPT && defined TCP_DEFER_ACCEPT)
273
274     timeout = 0;
275     olen = sizeof(int);
276
277     if (getsockopt(ls[i].fd, IPPROTO_TCP, TCP_DEFER_ACCEPT, &timeout, &olen)
278         == -1)
279     {
280         err = ngx\_socket\_errno;
281
282         if (err == NGX\_EOPNOTSUPP) {
283             continue;
284         }
285
286         ngx\_log\_error(NGX\_LOG\_NOTICE, cycle->log, err,
287                     "getsockopt(TCP_DEFER_ACCEPT) for %V failed, ignored",
288                     &ls[i].addr_text);
289         continue;
290     }
291
292     if (olen < sizeof(int) || timeout == 0) {
293         continue;
294     }
295
296     ls[i].deferred_accept = 1;
297 #endif
298     }
299
300     return NGX\_OK;
301 }
302
303
304 ngx\_int\_t
305 ngx\_open\_listening\_sockets(ngx\_cycle\_t *cycle)
306 {
307     int             reuseaddr;
308     ngx\_uint\_t     i, tries, failed;
309     ngx\_err\_t      err;
310     ngx\_log\_t      *log;
311     ngx\_socket\_t   s;
312     ngx\_listening\_t *ls;
313
314     reuseaddr = 1;
315 #if (NGX_SUPPRESS_WARN)
316     failed = 0;
317 #endif
318
319     log = cycle->log;
320
321     /* TODO: configurable try number */
322
323     for (tries = 5; tries; tries--) {
324         failed = 0;
325
326         /* for each listening socket */
327
328         ls = cycle->listening.elts;
329         for (i = 0; i < cycle->listening.nelts; i++) {
330
331             if (ls[i].ignore) {
332                 continue;
333             }
334
335             if (ls[i].fd != (ngx\_socket\_t) -1) {
336                 continue;

```

```

337     }
338
339     if (ls[i].inherited) {
340
341         /* TODO: close on exit */
342         /* TODO: nonblocking */
343         /* TODO: deferred accept */
344
345         continue;
346     }
347
348     s = ngx_socket(ls[i].sockaddr->sa_family, ls[i].type, 0);
349
350     if (s == (ngx_socket_t) -1) {
351         ngx_log_error(NGX_LOG_EMERG, log, ngx_socket_errno,
352                     ngx_socket_n " %V failed", &ls[i].addr_text);
353         return NGX_ERROR;
354     }
355
356     if (setsockopt(s, SOL_SOCKET, SO_REUSEADDR,
357                 (const void *) &reuseaddr, sizeof(int))
358         == -1)
359     {
360         ngx_log_error(NGX_LOG_EMERG, log, ngx_socket_errno,
361                     "setsockopt(SO_REUSEADDR) %V failed",
362                     &ls[i].addr_text);
363
364         if (ngx_close_socket(s) == -1) {
365             ngx_log_error(NGX_LOG_EMERG, log, ngx_socket_errno,
366                         ngx_close_socket_n " %V failed",
367                         &ls[i].addr_text);
368         }
369
370         return NGX_ERROR;
371     }
372
373     #if (NGX_HAVE_INET6 && defined IPV6_V6ONLY)
374
375     if (ls[i].sockaddr->sa_family == AF_INET6) {
376         int ipv6only;
377
378         ipv6only = ls[i].ipv6only;
379
380         if (setsockopt(s, IPPROTO_IPV6, IPV6_V6ONLY,
381                     (const void *) &ipv6only, sizeof(int))
382             == -1)
383         {
384             ngx_log_error(NGX_LOG_EMERG, log, ngx_socket_errno,
385                         "setsockopt(IPV6_V6ONLY) %V failed, ignored",
386                         &ls[i].addr_text);
387         }
388     }
389     #endif
390
391     /* TODO: close on exit */
392
393     if (!(ngx_event_flags & NGX_USE_AIO_EVENT)) {
394         if (ngx_nonblocking(s) == -1) {
395             ngx_log_error(NGX_LOG_EMERG, log, ngx_socket_errno,
396                         ngx_nonblocking_n " %V failed",
397                         &ls[i].addr_text);
398
399             if (ngx_close_socket(s) == -1) {
400                 ngx_log_error(NGX_LOG_EMERG, log, ngx_socket_errno,
401                             ngx_close_socket_n " %V failed",
402                             &ls[i].addr_text);
403             }
404
405             return NGX_ERROR;
406         }
407     }
408
409     ngx_log_debug2(NGX_LOG_DEBUG_CORE, log, 0,
410                 "bind() %V %#d ", &ls[i].addr_text, s);
411
412     if (bind(s, ls[i].sockaddr, ls[i].socklen) == -1) {
413         err = ngx_socket_errno;

```

```

413     if (err != NGX_EADDRINUSE || !ngx_test_config) {
414         ngx_log_error(NGX_LOG_EMERG, log, err,
415             "bind() to %V failed", &ls[i].addr_text);
416     }
417
418
419     if (ngx_close_socket(s) == -1) {
420         ngx_log_error(NGX_LOG_EMERG, log, ngx_socket_errno,
421             ngx_close_socket_n " %V failed",
422             &ls[i].addr_text);
423     }
424
425     if (err != NGX_EADDRINUSE) {
426         return NGX_ERROR;
427     }
428
429     if (!ngx_test_config) {
430         failed = 1;
431     }
432
433     continue;
434 }
435
436 #if (NGX_HAVE_UNIX_DOMAIN)
437
438     if (ls[i].sockaddr->sa_family == AF_UNIX) {
439         mode_t mode;
440         u_char *name;
441
442         name = ls[i].addr_text.data + sizeof("unix:") - 1;
443         mode = (S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH);
444
445         if (chmod((char *) name, mode) == -1) {
446             ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
447                 "chmod() \"%s\" failed", name);
448         }
449
450         if (ngx_test_config) {
451             if (ngx_delete_file(name) == NGX_FILE_ERROR) {
452                 ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
453                     ngx_delete_file_n " %s failed", name);
454             }
455         }
456     }
457 #endif
458
459     if (listen(s, ls[i].backlog) == -1) {
460         ngx_log_error(NGX_LOG_EMERG, log, ngx_socket_errno,
461             "listen() to %V, backlog %d failed",
462             &ls[i].addr_text, ls[i].backlog);
463
464         if (ngx_close_socket(s) == -1) {
465             ngx_log_error(NGX_LOG_EMERG, log, ngx_socket_errno,
466                 ngx_close_socket_n " %V failed",
467                 &ls[i].addr_text);
468         }
469
470         return NGX_ERROR;
471     }
472
473     ls[i].listen = 1;
474
475     ls[i].fd = s;
476 }
477
478 if (!failed) {
479     break;
480 }
481
482 /* TODO: delay configurable */
483
484 ngx_log_error(NGX_LOG_NOTICE, log, 0,
485     "try again to bind() after 500ms");
486
487 ngx_msleep(500);
488 }

```

```

489     if (failed) {
490         ngx_log_error(NGX_LOG_EMERG, log, 0, "still could not bind()");
491         return NGX_ERROR;
492     }
493 }
494
495 return NGX_OK;
496 }
497
498
499 void
500 ngx_configure_listening_sockets(ngx_cycle_t *cycle)
501 {
502     int             value;
503     ngx_uint_t     i;
504     ngx_listening_t *ls;
505
506     #if (NGX_HAVE_DEFERRED_ACCEPT && defined SO_ACCEPTFILTER)
507     struct accept_filter_arg af;
508     #endif
509
510     ls = cycle->listening.elts;
511     for (i = 0; i < cycle->listening.nelts; i++) {
512
513         ls[i].log = *ls[i].logp;
514
515         if (ls[i].rcvbuf != -1) {
516             if (setsockopt(ls[i].fd, SOL_SOCKET, SO_RCVBUF,
517                 (const void *) &ls[i].rcvbuf, sizeof(int))
518                 == -1)
519             {
520                 ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_socket_errno,
521                     "setsockopt(SO_RCVBUF, %d) %V failed, ignored",
522                     ls[i].rcvbuf, &ls[i].addr_text);
523             }
524         }
525
526         if (ls[i].sndbuf != -1) {
527             if (setsockopt(ls[i].fd, SOL_SOCKET, SO_SNDBUF,
528                 (const void *) &ls[i].sndbuf, sizeof(int))
529                 == -1)
530             {
531                 ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_socket_errno,
532                     "setsockopt(SO_SNDBUF, %d) %V failed, ignored",
533                     ls[i].sndbuf, &ls[i].addr_text);
534             }
535         }
536
537         if (ls[i].keepalive) {
538             value = (ls[i].keepalive == 1) ? 1 : 0;
539
540             if (setsockopt(ls[i].fd, SOL_SOCKET, SO_KEEPALIVE,
541                 (const void *) &value, sizeof(int))
542                 == -1)
543             {
544                 ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_socket_errno,
545                     "setsockopt(SO_KEEPALIVE, %d) %V failed, ignored",
546                     value, &ls[i].addr_text);
547             }
548         }
549
550         #if (NGX_HAVE_KEEPALIVE_TUNABLE)
551
552         if (ls[i].keepidle) {
553             value = ls[i].keepidle;
554
555             #if (NGX_KEEPALIVE_FACTOR)
556             value *= NGX_KEEPALIVE_FACTOR;
557             #endif
558
559             if (setsockopt(ls[i].fd, IPPROTO_TCP, TCP_KEEPIIDLE,
560                 (const void *) &value, sizeof(int))
561                 == -1)
562             {
563                 ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_socket_errno,
564                     "setsockopt(TCP_KEEPIIDLE, %d) %V failed, ignored",

```

```

565         value, &ls[i].addr_text);
566     }
567 }
568
569     if (ls[i].keepintvl) {
570         value = ls[i].keepintvl;
571
572     #if (NGX_KEEPALIVE_FACTOR)
573         value *= NGX_KEEPALIVE_FACTOR;
574     #endif
575
576     if (setsockopt(ls[i].fd, IPPROTO_TCP, TCP_KEEPINTVL,
577         (const void *) &value, sizeof(int))
578         == -1)
579     {
580         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_socket_errno,
581             "setsockopt(TCP_KEEPINTVL, %d) %V failed, ignored",
582             value, &ls[i].addr_text);
583     }
584 }
585
586     if (ls[i].keepcnt) {
587         if (setsockopt(ls[i].fd, IPPROTO_TCP, TCP_KEEPCNT,
588             (const void *) &ls[i].keepcnt, sizeof(int))
589             == -1)
590         {
591             ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_socket_errno,
592                 "setsockopt(TCP_KEEPCNT, %d) %V failed, ignored",
593                 ls[i].keepcnt, &ls[i].addr_text);
594         }
595     }
596 }
597 #endif
598
599 #if (NGX_HAVE_SETFIB)
600     if (ls[i].setfib != -1) {
601         if (setsockopt(ls[i].fd, SOL_SOCKET, SO_SETFIB,
602             (const void *) &ls[i].setfib, sizeof(int))
603             == -1)
604         {
605             ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_socket_errno,
606                 "setsockopt(SO_SETFIB, %d) %V failed, ignored",
607                 ls[i].setfib, &ls[i].addr_text);
608         }
609     }
610 #endif
611
612 #if (NGX_HAVE_TCP_FASTOPEN)
613     if (ls[i].fastopen != -1) {
614         if (setsockopt(ls[i].fd, IPPROTO_TCP, TCP_FASTOPEN,
615             (const void *) &ls[i].fastopen, sizeof(int))
616             == -1)
617         {
618             ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_socket_errno,
619                 "setsockopt(TCP_FASTOPEN, %d) %V failed, ignored",
620                 ls[i].fastopen, &ls[i].addr_text);
621         }
622     }
623 #endif
624
625 #if 0
626     if (1) {
627         int tcp_nodelay = 1;
628
629         if (setsockopt(ls[i].fd, IPPROTO_TCP, TCP_NODELAY,
630             (const void *) &tcp_nodelay, sizeof(int))
631             == -1)
632         {
633             ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_socket_errno,
634                 "setsockopt(TCP_NODELAY) %V failed, ignored",
635                 &ls[i].addr_text);
636         }
637     }
638 #endif
639
640     if (ls[i].listen) {

```



```

641         /* change backlog via listen() */
642
643
644         if (listen(ls[i].fd, ls[i].backlog) == -1) {
645             ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_socket_errno,
646                 "listen() to %V, backlog %d failed, ignored",
647                 &ls[i].addr_text, ls[i].backlog);
648         }
649     }
650
651     /*
652     * setting deferred mode should be last operation on socket,
653     * because code may prematurely continue cycle on failure
654     */
655
656     #if (NGX_HAVE_DEFERRED_ACCEPT)
657
658     #ifdef SO_ACCEPTFILTER
659
660         if (ls[i].delete_deferred) {
661             if (setsockopt(ls[i].fd, SOL_SOCKET, SO_ACCEPTFILTER, NULL, 0)
662                 == -1)
663             {
664                 ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_socket_errno,
665                     "setsockopt(SO_ACCEPTFILTER, NULL) "
666                     "for %V failed, ignored",
667                     &ls[i].addr_text);
668
669                 if (ls[i].accept_filter) {
670                     ngx_log_error(NGX_LOG_ALERT, cycle->log, 0,
671                         "could not change the accept filter "
672                         "to \"%s\" for %V, ignored",
673                         ls[i].accept_filter, &ls[i].addr_text);
674                 }
675
676                 continue;
677             }
678
679             ls[i].deferred_accept = 0;
680         }
681
682         if (ls[i].add_deferred) {
683             ngx_memzero(&af, sizeof(struct accept_filter_arg));
684             (void) ngx_cpystn((u_char *) af.af_name,
685                 (u_char *) ls[i].accept_filter, 16);
686
687             if (setsockopt(ls[i].fd, SOL_SOCKET, SO_ACCEPTFILTER,
688                 &af, sizeof(struct accept_filter_arg))
689                 == -1)
690             {
691                 ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_socket_errno,
692                     "setsockopt(SO_ACCEPTFILTER, \"%s\") "
693                     "for %V failed, ignored",
694                     ls[i].accept_filter, &ls[i].addr_text);
695                 continue;
696             }
697
698             ls[i].deferred_accept = 1;
699         }
700     }
701 #endif
702
703     #ifdef TCP_DEFER_ACCEPT
704
705         if (ls[i].add_deferred || ls[i].delete_deferred) {
706
707             if (ls[i].add_deferred) {
708                 /*
709                 * There is no way to find out how long a connection was
710                 * in queue (and a connection may bypass deferred queue at all
711                 * if syncookies were used), hence we use 1 second timeout
712                 * here.
713                 */
714                 value = 1;
715             }
716             else {

```

```

717         value = 0;
718     }
719
720     if (setsockopt(ls[i].fd, IPPROTO_TCP, TCP_DEFER_ACCEPT,
721                 &value, sizeof(int))
722         == -1)
723     {
724         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_socket_errno,
725                     "setsockopt(TCP_DEFER_ACCEPT, %d) for %V failed, "
726                     "ignored",
727                     value, &ls[i].addr_text);
728
729         continue;
730     }
731 }
732
733 if (ls[i].add_deferred) {
734     ls[i].deferred_accept = 1;
735 }
736
737 #endif
738
739 #endif /* NGX_HAVE_DEFERRED_ACCEPT */
740 }
741
742 return;
743 }
744
745
746 void
747 ngx_close_listening_sockets(ngx_cycle_t *cycle)
748 {
749     ngx_uint_t      i;
750     ngx_listening_t *ls;
751     ngx_connection_t *c;
752
753     if (ngx_event_flags & NGX_USE_IOCP_EVENT) {
754         return;
755     }
756
757     ngx_accept_mutex_held = 0;
758     ngx_use_accept_mutex = 0;
759
760     ls = cycle->listening.elts;
761     for (i = 0; i < cycle->listening.nelts; i++) {
762
763         c = ls[i].connection;
764
765         if (c) {
766             if (c->read->active) {
767                 if (ngx_event_flags & NGX_USE_RTSG_EVENT) {
768                     ngx_del_conn(c, NGX_CLOSE_EVENT);
769
770                 } else if (ngx_event_flags & NGX_USE_EPOLL_EVENT) {
771
772                     /*
773                      * it seems that Linux-2.6.x OpenVZ sends events
774                      * for closed shared listening sockets unless
775                      * the events was explicitly deleted
776                      */
777
778                     ngx_del_event(c->read, NGX_READ_EVENT, 0);
779
780                 } else {
781                     ngx_del_event(c->read, NGX_READ_EVENT, NGX_CLOSE_EVENT);
782                 }
783             }
784
785             ngx_free_connection(c);
786
787             c->fd = (ngx_socket_t) -1;
788         }
789
790         ngx_log_debug2(NGX_LOG_DEBUG_CORE, cycle->log, 0,
791                     "close listening %V #%d ", &ls[i].addr_text, ls[i].fd);
792     }

```

```

793     if (ngx_close_socket(ls[i].fd) == -1) {
794         ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_socket_errno,
795             ngx_close_socket_n " %V failed", &ls[i].addr_text);
796     }
797
798 #if (NGX_HAVE_UNIX_DOMAIN)
799
800     if (ls[i].sockaddr->sa_family == AF_UNIX
801         && ngx_process <= NGX_PROCESS_MASTER
802         && ngx_new_binary == 0)
803     {
804         u_char *name = ls[i].addr_text.data + sizeof("unix:") - 1;
805
806         if (ngx_delete_file(name) == NGX_FILE_ERROR) {
807             ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_socket_errno,
808                 ngx_delete_file_n " %s failed", name);
809         }
810     }
811
812 #endif
813
814     ls[i].fd = (ngx_socket_t) -1;
815 }
816
817 cycle->listening.nelts = 0;
818 }
819
820
821 ngx_connection_t *
822 ngx_get_connection(ngx_socket_t s, ngx_log_t *log)
823 {
824     ngx_uint_t         instance;
825     ngx_event_t      *rev, *wev;
826     ngx_connection_t *c;
827
828     /* disable warning: Win32 SOCKET is u_int while UNIX socket is int */
829
830     if (ngx_cycle->files && (ngx_uint_t) s >= ngx_cycle->files_n) {
831         ngx_log_error(NGX_LOG_ALERT, log, 0,
832             "the new socket has number %d, "
833             "but only %ui files are available",
834             s, ngx_cycle->files_n);
835         return NULL;
836     }
837
838     /* ngx_mutex_lock */
839
840     c = ngx_cycle->free_connections;
841
842     if (c == NULL) {
843         ngx_drain_connections();
844         c = ngx_cycle->free_connections;
845     }
846
847     if (c == NULL) {
848         ngx_log_error(NGX_LOG_ALERT, log, 0,
849             "%ui worker_connections are not enough",
850             ngx_cycle->connection_n);
851
852         /* ngx_mutex_unlock */
853
854         return NULL;
855     }
856
857     ngx_cycle->free_connections = c->data;
858     ngx_cycle->free_connection_n--;
859
860     /* ngx_mutex_unlock */
861
862     if (ngx_cycle->files) {
863         ngx_cycle->files[s] = c;
864     }
865
866     rev = c->read;
867     wev = c->write;
868

```

```

869     ngx_memzero(c, sizeof(ngx_connection_t));
870
871     c->read = rev;
872     c->write = wev;
873     c->fd = s;
874     c->log = log;
875
876     instance = rev->instance;
877
878     ngx_memzero(rev, sizeof(ngx_event_t));
879     ngx_memzero(wev, sizeof(ngx_event_t));
880
881     rev->instance = !instance;
882     wev->instance = !instance;
883
884     rev->index = NGX_INVALID_INDEX;
885     wev->index = NGX_INVALID_INDEX;
886
887     rev->data = c;
888     wev->data = c;
889
890     wev->write = 1;
891
892     return c;
893 }
894
895
896 void
897 ngx_free_connection(ngx_connection_t *c)
898 {
899     /* ngx_mutex_lock */
900
901     c->data = ngx_cycle->free_connections;
902     ngx_cycle->free_connections = c;
903     ngx_cycle->free_connection_n++;
904
905     /* ngx_mutex_unlock */
906
907     if (ngx_cycle->files) {
908         ngx_cycle->files[c->fd] = NULL;
909     }
910 }
911
912
913 void
914 ngx_close_connection(ngx_connection_t *c)
915 {
916     ngx_err_t    err;
917     ngx_uint_t   log_error, level;
918     ngx_socket_t fd;
919
920     if (c->fd == (ngx_socket_t) -1) {
921         ngx_log_error(NGX_LOG_ALERT, c->log, 0, "connection already closed");
922         return;
923     }
924
925     if (c->read->timer_set) {
926         ngx_del_timer(c->read);
927     }
928
929     if (c->write->timer_set) {
930         ngx_del_timer(c->write);
931     }
932
933     if (ngx_del_conn) {
934         ngx_del_conn(c, NGX_CLOSE_EVENT);
935     }
936 } else {
937     if (c->read->active || c->read->disabled) {
938         ngx_del_event(c->read, NGX_READ_EVENT, NGX_CLOSE_EVENT);
939     }
940
941     if (c->write->active || c->write->disabled) {
942         ngx_del_event(c->write, NGX_WRITE_EVENT, NGX_CLOSE_EVENT);
943     }
944 }

```

```

945
946 #if (NGX_THREADS)
947
948 /*
949  * we have to clean the connection information before the closing
950  * because another thread may reopen the same file descriptor
951  * before we clean the connection
952  */
953
954 ngx_unlock(&c->lock);
955
956 #endif
957
958 if (c->read->posted) {
959     ngx_delete_posted_event(c->read);
960 }
961
962 if (c->write->posted) {
963     ngx_delete_posted_event(c->write);
964 }
965
966 c->read->closed = 1;
967 c->write->closed = 1;
968
969 ngx_reusable_connection(c, 0);
970
971 log_error = c->log_error;
972
973 ngx_free_connection(c);
974
975 fd = c->fd;
976 c->fd = (ngx_socket_t) -1;
977
978 if (ngx_close_socket(fd) == -1) {
979
980     err = ngx_socket_errno;
981
982     if (err == NGX_ECONNRESET || err == NGX_ENOTCONN) {
983
984         switch (log_error) {
985
986             case NGX_ERROR_INFO:
987                 level = NGX_LOG_INFO;
988                 break;
989
990             case NGX_ERROR_ERR:
991                 level = NGX_LOG_ERR;
992                 break;
993
994             default:
995                 level = NGX_LOG_CRIT;
996         }
997
998     } else {
999         level = NGX_LOG_CRIT;
1000     }
1001
1002     /* we use ngx_cycle->log because c->log was in c->pool */
1003
1004     ngx_log_error(level, ngx_cycle->log, err,
1005                  ngx_close_socket_n " %d failed", fd);
1006 }
1007 }
1008
1009
1010 void
1011 ngx_reusable_connection(ngx_connection_t *c, ngx_uint_t reusable)
1012 {
1013     ngx_log_debug1(NGX_LOG_DEBUG_CORE, c->log, 0,
1014                  "reusable connection: %ui", reusable);
1015
1016     if (c->reusable) {
1017         ngx_queue_remove(&c->queue);
1018     }
1019 #if (NGX_STAT_STUB)
1020     (void) ngx_atomic_fetch_add(ngx_stat_waiting, -1);

```

```

1021 #endif
1022     }
1023
1024     c->reusable = reusable;
1025
1026     if (reusable) {
1027         /* need cast as ngx_cycle is volatile */
1028
1029         ngx_queue_insert_head(
1030             (ngx_queue_t *) &ngx_cycle->reusable_connections_queue, &c->queue);
1031
1032     #if (NGX_STAT_STUB)
1033         (void) ngx_atomic_fetch_add(ngx_stat_waiting, 1);
1034     #endif
1035     }
1036 }
1037
1038
1039 static void
1040 ngx_drain_connections(void)
1041 {
1042     ngx_int_t     i;
1043     ngx_queue_t  *q;
1044     ngx_connection_t *c;
1045
1046     for (i = 0; i < 32; i++) {
1047         if (ngx_queue_empty(&ngx_cycle->reusable_connections_queue)) {
1048             break;
1049         }
1050
1051         q = ngx_queue_last(&ngx_cycle->reusable_connections_queue);
1052         c = ngx_queue_data(q, ngx_connection_t, queue);
1053
1054         ngx_log_debug0(NGX_LOG_DEBUG_CORE, c->log, 0,
1055             "reusing connection");
1056
1057         c->close = 1;
1058         c->read->handler(c->read);
1059     }
1060 }
1061
1062
1063 ngx_int_t
1064 ngx_connection_local_sockaddr(ngx_connection_t *c, ngx_str_t *s,
1065     ngx_uint_t port)
1066 {
1067     socklen_t     len;
1068     ngx_uint_t   addr;
1069     u_char       sa[NGX_SOCKADDRLEN];
1070     struct sockaddr_in *sin;
1071     #if (NGX_HAVE_INET6)
1072     ngx_uint_t   i;
1073     struct sockaddr_in6 *sin6;
1074     #endif
1075
1076     if (c->local_socklen == 0) {
1077         return NGX_ERROR;
1078     }
1079
1080     switch (c->local_sockaddr->sa_family) {
1081
1082     #if (NGX_HAVE_INET6)
1083     case AF_INET6:
1084         sin6 = (struct sockaddr_in6 *) c->local_sockaddr;
1085
1086         for (addr = 0, i = 0; addr == 0 && i < 16; i++) {
1087             addr |= sin6->sin6_addr.s6_addr[i];
1088         }
1089
1090         break;
1091     #endif
1092
1093     #if (NGX_HAVE_UNIX_DOMAIN)
1094     case AF_UNIX:
1095         addr = 1;
1096         break;

```

```

1097 #endif
1098
1099     default: /* AF_INET */
1100         sin = (struct sockaddr_in *) c->local_sockaddr;
1101         addr = sin->sin_addr.s_addr;
1102         break;
1103     }
1104
1105     if (addr == 0) {
1106
1107         len = NGX_SOCKADDRLEN;
1108
1109         if (getsockname(c->fd, (struct sockaddr *) &sa, &len) == -1) {
1110             ngx_connection_error(c, ngx_socket_errno, "getsockname() failed");
1111             return NGX_ERROR;
1112         }
1113
1114         c->local_sockaddr = ngx_palloc(c->pool, len);
1115         if (c->local_sockaddr == NULL) {
1116             return NGX_ERROR;
1117         }
1118
1119         ngx_memcpy(c->local_sockaddr, &sa, len);
1120
1121         c->local_socklen = len;
1122     }
1123
1124     if (s == NULL) {
1125         return NGX_OK;
1126     }
1127
1128     s->len = ngx_sock_ntop(c->local_sockaddr, c->local_socklen,
1129                             s->data, s->len, port);
1130
1131     return NGX_OK;
1132 }
1133
1134
1135 ngx_int_t
1136 ngx_connection_error(ngx_connection_t *c, ngx_err_t err, char *text)
1137 {
1138     ngx_uint_t level;
1139
1140     /* Winsock may return NGX_ECONNABORTED instead of NGX_ECONNRESET */
1141
1142     if ((err == NGX_ECONNRESET
1143 #if (NGX_WIN32)
1144         || err == NGX_ECONNABORTED
1145 #endif
1146         ) && c->log_error == NGX_ERROR_IGNORE_ECONNRESET)
1147     {
1148         return 0;
1149     }
1150
1151     #if (NGX_SOLARIS)
1152     if (err == NGX_EINVAL && c->log_error == NGX_ERROR_IGNORE_EINVAL) {
1153         return 0;
1154     }
1155     #endif
1156
1157     if (err == 0
1158         || err == NGX_ECONNRESET
1159 #if (NGX_WIN32)
1160         || err == NGX_ECONNABORTED
1161 #else
1162         || err == NGX_EPIPE
1163 #endif
1164         || err == NGX_ENOTCONN
1165         || err == NGX_ETIMEDOUT
1166         || err == NGX_ECONNREFUSED
1167         || err == NGX_ENETDOWN
1168         || err == NGX_ENETUNREACH
1169         || err == NGX_EHOSTDOWN
1170         || err == NGX_EHOSTUNREACH)
1171     {
1172         switch (c->log_error) {

```

```
1173
1174     case NGX_ERROR_IGNORE_EINVAL:
1175     case NGX_ERROR_IGNORE_ECONNRESET:
1176     case NGX_ERROR_INFO:
1177         level = NGX\_LOG\_INFO;
1178         break;
1179
1180     default:
1181         level = NGX\_LOG\_ERR;
1182     }
1183 } else {
1184     level = NGX\_LOG\_ALERT;
1185 }
1186
1187 ngx\_log\_error(level, c->log, err, text);
1188
1189 return NGX\_ERROR;
1191 }
```

[One Level Up](#)

[Top Level](#)



# src/core/nginx\_connection.h - nginx-1.7.10

## Data types defined

- [ngx\\_connection\\_log\\_error\\_e](#)
- [ngx\\_connection\\_s](#)
- [ngx\\_connection\\_tcp\\_nodelay\\_e](#)
- [ngx\\_connection\\_tcp\\_nopush\\_e](#)
- [ngx\\_listening\\_s](#)
- [ngx\\_listening\\_t](#)

## Macros defined

- [NGX\\_LOWLEVEL\\_BUFFERED](#)
- [NGX\\_SPDY\\_BUFFERED](#)
- [NGX\\_SSL\\_BUFFERED](#)
- [\\_NGX\\_CONNECTION\\_H\\_INCLUDED](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_CONNECTION_H_INCLUDED
9 #define _NGX_CONNECTION_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 typedef struct ngx_listening_s ngx_listening_t;
17
18 struct ngx_listening_s {
19     ngx_socket_t      fd;
20
21     struct sockaddr   *sockaddr;
22     socklen_t         socklen;    /* size of sockaddr */
23     size_t            addr_text_max_len;
24     ngx_str_t         addr_text;
25
26     int               type;
27
28     int               backlog;
29     int               rcvbuf;
30     int               sndbuf;
31 #if (NGX_HAVE_KEEPALIVE_TUNABLE)
32     int               keepidle;
33     int               keepintvl;
34     int               keepcnt;
35 #endif
36
37     /* handler of accepted connection */
38     ngx_connection_handler_pt handler;
```

```

39 void                *servers; /* array of ngx_http_in_addr_t, for example */
40
41 ngx_log_t          log;
42 ngx_log_t          *logp;
43
44
45 size_t              pool_size;
46 /* should be here because of the AcceptEx() preread */
47 size_t              post_accept_buffer_size;
48 /* should be here because of the deferred accept */
49 ngx_msec_t         post_accept_timeout;
50
51 ngx_listening_t   *previous;
52 ngx_connection_t *connection;
53
54 unsigned            open:1;
55 unsigned            remain:1;
56 unsigned            ignore:1;
57
58 unsigned            bound:1;      /* already bound */
59 unsigned            inherited:1;  /* inherited from previous process */
60 unsigned            nonblocking_accept:1;
61 unsigned            listen:1;
62 unsigned            nonblocking:1;
63 unsigned            shared:1;     /* shared between threads or processes */
64 unsigned            addr_ntop:1;
65
66 #if (NGX_HAVE_INET6 && defined IPV6_V6ONLY)
67     unsigned        ipv6only:1;
68 #endif
69     unsigned        keepalive:2;
70
71 #if (NGX_HAVE_DEFERRED_ACCEPT)
72     unsigned        deferred_accept:1;
73     unsigned        delete_deferred:1;
74     unsigned        add_deferred:1;
75 #ifdef SO_ACCEPTFILTER
76     char            *accept_filter;
77 #endif
78 #endif
79 #if (NGX_HAVE_SETFIB)
80     int             setfib;
81 #endif
82
83 #if (NGX_HAVE_TCP_FASTOPEN)
84     int             fastopen;
85 #endif
86
87 };
88
89
90 typedef enum {
91     NGX_ERROR_ALERT = 0,
92     NGX_ERROR_ERR,
93     NGX_ERROR_INFO,
94     NGX_ERROR_IGNORE_ECONNRESET,
95     NGX_ERROR_IGNORE_EINVAL
96 } ngx_connection_log_error_e;
97
98
99 typedef enum {
100     NGX_TCP_NODELAY_UNSET = 0,
101     NGX_TCP_NODELAY_SET,
102     NGX_TCP_NODELAY_DISABLED
103 } ngx_connection_tcp_nodelay_e;
104
105
106 typedef enum {
107     NGX_TCP_NOPUSH_UNSET = 0,
108     NGX_TCP_NOPUSH_SET,
109     NGX_TCP_NOPUSH_DISABLED
110 } ngx_connection_tcp_nopush_e;
111
112
113 #define NGX_LOWLEVEL_BUFFERED 0x0f
114 #define NGX_SSL_BUFFERED     0x01

```

```

115 #define NGX_SPDY_BUFFERED      0x02
116
117
118 struct ngx_connection_s {
119     void                *data;
120     ngx_event_t         *read;
121     ngx_event_t         *write;
122
123     ngx_socket_t        fd;
124
125     ngx_recv_pt         recv;
126     ngx_send_pt         send;
127     ngx_recv_chain_pt   recv_chain;
128     ngx_send_chain_pt   send_chain;
129
130     ngx_listening_t     *listening;
131
132     off_t                sent;
133
134     ngx_log_t           *log;
135
136     ngx_pool_t          *pool;
137
138     struct sockaddr     *sockaddr;
139     socklen_t           socklen;
140     ngx_str_t           addr_text;
141
142     ngx_str_t           proxy_protocol_addr;
143
144     #if (NGX_SSL)
145     ngx_ssl_connection_t *ssl;
146     #endif
147
148     struct sockaddr     *local_sockaddr;
149     socklen_t           local_socklen;
150
151     ngx_buf_t           *buffer;
152
153     ngx_queue_t         queue;
154
155     ngx_atomic_uint_t   number;
156
157     ngx_uint_t          requests;
158
159     unsigned            buffered:8;
160
161     unsigned            log_error:3;    /* ngx_connection_log_error_e */
162
163     unsigned            unexpected_eof:1;
164     unsigned            timedout:1;
165     unsigned            error:1;
166     unsigned            destroyed:1;
167
168     unsigned            idle:1;
169     unsigned            reusable:1;
170     unsigned            close:1;
171
172     unsigned            sendfile:1;
173     unsigned            sndlowat:1;
174     unsigned            tcp_nodelay:2; /* ngx_connection_tcp_nodelay_e */
175     unsigned            tcp_nopush:2; /* ngx_connection_tcp_nopush_e */
176
177     unsigned            need_last_buf:1;
178
179     #if (NGX_HAVE_IOCP)
180     unsigned            accept_context_updated:1;
181     #endif
182
183     #if (NGX_HAVE_AIO_SENDFILE)
184     unsigned            aio_sendfile:1;
185     unsigned            busy_count:2;
186     ngx_buf_t           *busy_sendfile;
187     #endif
188
189     #if (NGX_THREADS)
190     ngx_atomic_t        lock;

```

```
191 #endif
192 };
193
194
195 ngx\_listening\_t *ngx\_create\_listening(ngx\_conf\_t *cf, void *sockaddr,
196     socklen_t socklen);
197 ngx\_int\_t ngx\_set\_inherited\_sockets(ngx\_cycle\_t *cycle);
198 ngx\_int\_t ngx\_open\_listening\_sockets(ngx\_cycle\_t *cycle);
199 void ngx\_configure\_listening\_sockets(ngx\_cycle\_t *cycle);
200 void ngx\_close\_listening\_sockets(ngx\_cycle\_t *cycle);
201 void ngx\_close\_connection(ngx\_connection\_t *c);
202 ngx\_int\_t ngx\_connection\_local\_sockaddr(ngx\_connection\_t *c, ngx\_str\_t *s,
203     ngx\_uint\_t port);
204 ngx\_int\_t ngx\_connection\_error(ngx\_connection\_t *c, ngx\_err\_t err, char *text);
205
206 ngx\_connection\_t *ngx\_get\_connection(ngx\_socket\_t s, ngx\_log\_t *log);
207 void ngx\_free\_connection(ngx\_connection\_t *c);
208
209 void ngx\_reusable\_connection(ngx\_connection\_t *c, ngx\_uint\_t reusable);
210
211 #endif /* \_NGX\_CONNECTION\_H\_INCLUDED */
```

[One Level Up](#)

[Top Level](#)

## src/os/unix/nginx\_socket.h - nginx-1.7.10

### Data types defined

- [ngx\\_socket\\_t](#)

### Macros defined

- [NGX\\_WRITE\\_SHUTDOWN](#)
- [\\_NGX\\_SOCKET\\_H\\_INCLUDED](#)
- [ngx\\_blocking](#)
- [ngx\\_blocking\\_n](#)
- [ngx\\_blocking\\_n](#)
- [ngx\\_close\\_socket](#)
- [ngx\\_close\\_socket\\_n](#)
- [ngx\\_nonblocking](#)
- [ngx\\_nonblocking\\_n](#)
- [ngx\\_nonblocking\\_n](#)
- [ngx\\_shutdown\\_socket](#)
- [ngx\\_shutdown\\_socket\\_n](#)
- [ngx\\_socket](#)
- [ngx\\_socket\\_n](#)
- [ngx\\_tcp\\_nopush\\_n](#)
- [ngx\\_tcp\\_nopush\\_n](#)
- [ngx\\_tcp\\_push\\_n](#)
- [ngx\\_tcp\\_push\\_n](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_SOCKET\_H\_INCLUDED
9 #define \_NGX\_SOCKET\_H\_INCLUDED
10
11
12 #include <ngx_config.h>
13
14
15 #define NGX\_WRITE\_SHUTDOWN SHUT_WR
16
17 typedef int ngx\_socket\_t;
18
```

```

19 #define ngx_socket      socket
20 #define ngx_socket_n    "socket()"
21
22
23 #if (NGX_HAVE_FIONBIO)
24
25 int ngx_nonblocking(ngx_socket_t s);
26 int ngx_blocking(ngx_socket_t s);
27
28 #define ngx_nonblocking_n    "ioctl(FIONBIO)"
29 #define ngx_blocking_n      "ioctl(!FIONBIO)"
30
31 #else
32
33 #define ngx_nonblocking(s)    fcntl(s, F_SETFL, fcntl(s, F_GETFL) | O_NONBLOCK)
34 #define ngx_nonblocking_n    "fcntl(O_NONBLOCK)"
35
36 #define ngx_blocking(s)      fcntl(s, F_SETFL, fcntl(s, F_GETFL) & ~O_NONBLOCK)
37 #define ngx_blocking_n      "fcntl(!O_NONBLOCK)"
38
39 #endif
40
41 int ngx_tcp_nopush(ngx_socket_t s);
42 int ngx_tcp_push(ngx_socket_t s);
43
44 #if (NGX_LINUX)
45
46 #define ngx_tcp_nopush_n    "setsockopt(TCP_CORK)"
47 #define ngx_tcp_push_n      "setsockopt(!TCP_CORK)"
48
49 #else
50
51 #define ngx_tcp_nopush_n    "setsockopt(TCP_NOPUSH)"
52 #define ngx_tcp_push_n      "setsockopt(!TCP_NOPUSH)"
53
54 #endif
55
56
57 #define ngx_shutdown_socket  shutdown
58 #define ngx_shutdown_socket_n "shutdown()"
59
60 #define ngx_close_socket     close
61 #define ngx_close_socket_n   "close() socket"
62
63
64 #endif /* NGX_SOCKET_H_INCLUDED */

```

[One Level Up](#)

[Top Level](#)

## src/os/unix/nginx\_socket.c - nginx-1.7.10

### Functions defined

- [ngx\\_blocking](#)
- [ngx\\_nonblocking](#)
- [ngx\\_tcp\\_nopush](#)
- [ngx\\_tcp\\_nopush](#)
- [ngx\\_tcp\\_nopush](#)
- [ngx\\_tcp\\_push](#)
- [ngx\\_tcp\\_push](#)
- [ngx\\_tcp\\_push](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10
11
12 /*
13  * ioctl(FIONBIO) sets a non-blocking mode with the single syscall
14  * while fcntl(F_SETFL, O_NONBLOCK) needs to learn the current state
15  * using fcntl(F_GETFL).
16  *
17  * ioctl() and fcntl() are syscalls at least in FreeBSD 2.x, Linux 2.2
18  * and Solaris 7.
19  *
20  * ioctl() in Linux 2.4 and 2.6 uses BKL, however, fcntl(F_SETFL) uses it too.
21  */
22
23
24 #if (NGX_HAVE_FIONBIO)
25
26 int
27 ngx_nonblocking(ngx_socket_t s)
28 {
29     int nb;
30
31     nb = 1;
32
33     return ioctl(s, FIONBIO, &nb);
34 }
35
36 int
37 ngx_blocking(ngx_socket_t s)
38 {
39     int nb;
40
41     nb = 0;
42
43     return ioctl(s, FIONBIO, &nb);
44 }
45
46 #endif
```

```

48
49
50 #if (NGX_FREEBSD)
51
52 int
53 ngx_tcp_nopush(ngx_socket_t s)
54 {
55     int tcp_nopush;
56
57     tcp_nopush = 1;
58
59     return setsockopt(s, IPPROTO_TCP, TCP_NOPUSH,
60                     (const void *) &tcp_nopush, sizeof(int));
61 }
62
63
64 int
65 ngx_tcp_push(ngx_socket_t s)
66 {
67     int tcp_nopush;
68
69     tcp_nopush = 0;
70
71     return setsockopt(s, IPPROTO_TCP, TCP_NOPUSH,
72                     (const void *) &tcp_nopush, sizeof(int));
73 }
74
75 #elif (NGX_LINUX)
76
77
78 int
79 ngx_tcp_nopush(ngx_socket_t s)
80 {
81     int cork;
82
83     cork = 1;
84
85     return setsockopt(s, IPPROTO_TCP, TCP_CORK,
86                     (const void *) &cork, sizeof(int));
87 }
88
89
90 int
91 ngx_tcp_push(ngx_socket_t s)
92 {
93     int cork;
94
95     cork = 0;
96
97     return setsockopt(s, IPPROTO_TCP, TCP_CORK,
98                     (const void *) &cork, sizeof(int));
99 }
100
101 #else
102
103 int
104 ngx_tcp_nopush(ngx_socket_t s)
105 {
106     return 0;
107 }
108
109
110 int
111 ngx_tcp_push(ngx_socket_t s)
112 {
113     return 0;
114 }
115
116 #endif

```

[One Level Up](#)

[Top Level](#)



## src/http/nginx\_http\_core\_module.h - nginx-1.7.10

### Data types defined

- [ngx\\_http\\_addr\\_conf\\_s](#)
- [ngx\\_http\\_conf\\_addr\\_t](#)
- [ngx\\_http\\_conf\\_port\\_t](#)
- [ngx\\_http\\_core\\_loc\\_conf\\_s](#)
- [ngx\\_http\\_core\\_loc\\_conf\\_t](#)
- [ngx\\_http\\_core\\_main\\_conf\\_t](#)
- [ngx\\_http\\_core\\_srv\\_conf\\_t](#)
- [ngx\\_http\\_err\\_page\\_t](#)
- [ngx\\_http\\_in6\\_addr\\_t](#)
- [ngx\\_http\\_in\\_addr\\_t](#)
- [ngx\\_http\\_listen\\_opt\\_t](#)
- [ngx\\_http\\_location\\_queue\\_t](#)
- [ngx\\_http\\_location\\_tree\\_node\\_s](#)
- [ngx\\_http\\_location\\_tree\\_node\\_t](#)
- [ngx\\_http\\_output\\_body\\_filter\\_pt](#)
- [ngx\\_http\\_output\\_header\\_filter\\_pt](#)
- [ngx\\_http\\_phase\\_engine\\_t](#)
- [ngx\\_http\\_phase\\_handler\\_pt](#)
- [ngx\\_http\\_phase\\_handler\\_s](#)
- [ngx\\_http\\_phase\\_handler\\_t](#)
- [ngx\\_http\\_phase\\_t](#)
- [ngx\\_http\\_phases](#)
- [ngx\\_http\\_port\\_t](#)
- [ngx\\_http\\_server\\_name\\_t](#)
- [ngx\\_http\\_try\\_file\\_t](#)
- [ngx\\_http\\_virtual\\_names\\_t](#)

### Macros defined

- [NGX\\_HTTP\\_AIO\\_OFF](#)
- [NGX\\_HTTP\\_AIO\\_ON](#)

- [NGX HTTP AIO SENDFILE](#)
- [NGX HTTP GZIP PROXIED ANY](#)
- [NGX HTTP GZIP PROXIED AUTH](#)
- [NGX HTTP GZIP PROXIED EXPIRED](#)
- [NGX HTTP GZIP PROXIED NO CACHE](#)
- [NGX HTTP GZIP PROXIED NO ETAG](#)
- [NGX HTTP GZIP PROXIED NO LM](#)
- [NGX HTTP GZIP PROXIED NO STORE](#)
- [NGX HTTP GZIP PROXIED OFF](#)
- [NGX HTTP GZIP PROXIED PRIVATE](#)
- [NGX HTTP IMS BEFORE](#)
- [NGX HTTP IMS EXACT](#)
- [NGX HTTP IMS OFF](#)
- [NGX HTTP KEEPALIVE DISABLE MSIE6](#)
- [NGX HTTP KEEPALIVE DISABLE NONE](#)
- [NGX HTTP KEEPALIVE DISABLE SAFARI](#)
- [NGX HTTP LINGERING ALWAYS](#)
- [NGX HTTP LINGERING OFF](#)
- [NGX HTTP LINGERING ON](#)
- [NGX HTTP SATISFY ALL](#)
- [NGX HTTP SATISFY ANY](#)
- [\\_NGX\\_HTTP\\_CORE\\_H\\_INCLUDED\\_](#)
- [ngx http clear accept ranges](#)
- [ngx http clear content length](#)
- [ngx http clear etag](#)
- [ngx http clear last modified](#)
- [ngx http clear location](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_HTTP_CORE_H_INCLUDED_
9 #define _NGX_HTTP_CORE_H_INCLUDED_
10
11
```

```

12 #include <ngx_config.h>
13 #include <ngx_core.h>
14 #include <ngx_http.h>
15
16
17 #define NGX_HTTP_GZIP_PROXIED_OFF      0x0002
18 #define NGX_HTTP_GZIP_PROXIED_EXPIRED 0x0004
19 #define NGX_HTTP_GZIP_PROXIED_NO_CACHE 0x0008
20 #define NGX_HTTP_GZIP_PROXIED_NO_STORE 0x0010
21 #define NGX_HTTP_GZIP_PROXIED_PRIVATE  0x0020
22 #define NGX_HTTP_GZIP_PROXIED_NO_LM    0x0040
23 #define NGX_HTTP_GZIP_PROXIED_NO_ETAG  0x0080
24 #define NGX_HTTP_GZIP_PROXIED_AUTH     0x0100
25 #define NGX_HTTP_GZIP_PROXIED_ANY      0x0200
26
27
28 #define NGX_HTTP_AIO_OFF                0
29 #define NGX_HTTP_AIO_ON                  1
30 #define NGX_HTTP_AIO_SENDFILE           2
31
32
33 #define NGX_HTTP_SATISFY_ALL             0
34 #define NGX_HTTP_SATISFY_ANY            1
35
36
37 #define NGX_HTTP_LINGERING_OFF           0
38 #define NGX_HTTP_LINGERING_ON            1
39 #define NGX_HTTP_LINGERING_ALWAYS        2
40
41
42 #define NGX_HTTP_IMS_OFF                  0
43 #define NGX_HTTP_IMS_EXACT                1
44 #define NGX_HTTP_IMS_BEFORE              2
45
46
47 #define NGX_HTTP_KEEPALIVE_DISABLE_NONE  0x0002
48 #define NGX_HTTP_KEEPALIVE_DISABLE_MSIE6 0x0004
49 #define NGX_HTTP_KEEPALIVE_DISABLE_SAFARI 0x0008
50
51
52 typedef struct ngx\_http\_location\_tree\_node\_s ngx_http_location_tree_node_t;
53 typedef struct ngx\_http\_core\_loc\_conf\_s ngx_http_core_loc_conf_t;
54
55
56 typedef struct {
57     union {
58         struct sockaddr      sockaddr;
59         struct sockaddr_in    sockaddr_in;
60 #if (NGX_HAVE_INET6)
61         struct sockaddr_in6   sockaddr_in6;
62 #endif
63 #if (NGX_HAVE_UNIX_DOMAIN)
64         struct sockaddr_un     sockaddr_un;
65 #endif
66         u_char                sockaddr_data[NGX_SOCKADDRLEN];
67     } u;
68
69     socklen_t                  socklen;
70
71     unsigned                   set:1;
72     unsigned                   default_server:1;
73     unsigned                   bind:1;
74     unsigned                   wildcard:1;
75 #if (NGX_HTTP_SSL)
76     unsigned                   ssl:1;
77 #endif
78 #if (NGX_HTTP_SPDY)
79     unsigned                   spdy:1;
80 #endif
81 #if (NGX_HAVE_INET6 && defined IPV6_V6ONLY)
82     unsigned                   ipv6only:1;
83 #endif
84     unsigned                   so_keepalive:2;
85     unsigned                   proxy_protocol:1;
86
87     int                        backlog;

```

```

88     int                rcvbuf;
89     int                sndbuf;
90 #if (NGX_HAVE_SETFIB)
91     int                setfib;
92 #endif
93 #if (NGX_HAVE_TCP_FASTOPEN)
94     int                fastopen;
95 #endif
96 #if (NGX_HAVE_KEEPALIVE_TUNABLE)
97     int                tcp_keepidle;
98     int                tcp_keepintvl;
99     int                tcp_keepcnt;
100 #endif
101
102 #if (NGX_HAVE_DEFERRED_ACCEPT && defined SO_ACCEPTFILTER)
103     char                *accept_filter;
104 #endif
105 #if (NGX_HAVE_DEFERRED_ACCEPT && defined TCP_DEFER_ACCEPT)
106     ngx_uint_t         deferred_accept;
107 #endif
108
109     u_char              addr[NGX_SOCKADDR_STRLEN + 1];
110 } ngx_http_listen_opt_t;
111
112
113 typedef enum {
114     NGX_HTTP_POST_READ_PHASE = 0,
115
116     NGX_HTTP_SERVER_REWRITE_PHASE,
117
118     NGX_HTTP_FIND_CONFIG_PHASE,
119     NGX_HTTP_REWRITE_PHASE,
120     NGX_HTTP_POST_REWRITE_PHASE,
121
122     NGX_HTTP_PREACCESS_PHASE,
123
124     NGX_HTTP_ACCESS_PHASE,
125     NGX_HTTP_POST_ACCESS_PHASE,
126
127     NGX_HTTP_TRY_FILES_PHASE,
128     NGX_HTTP_CONTENT_PHASE,
129
130     NGX_HTTP_LOG_PHASE
131 } ngx_http_phases;
132
133 typedef struct ngx_http_phase_handler_s ngx_http_phase_handler_t;
134
135 typedef ngx_int_t (*ngx_http_phase_handler_pt)(ngx_http_request_t *r,
136     ngx_http_phase_handler_t *ph);
137
138 struct ngx_http_phase_handler_s {
139     ngx_http_phase_handler_pt checker;
140     ngx_http_handler_pt handler;
141     ngx_uint_t next;
142 };
143
144
145 typedef struct {
146     ngx_http_phase_handler_t *handlers;
147     ngx_uint_t server_rewrite_index;
148     ngx_uint_t location_rewrite_index;
149 } ngx_http_phase_engine_t;
150
151
152 typedef struct {
153     ngx_array_t handlers;
154 } ngx_http_phase_t;
155
156
157 typedef struct {
158     ngx_array_t servers; /* ngx_http_core_srv_conf_t */
159
160     ngx_http_phase_engine_t phase_engine;
161
162     ngx_hash_t headers_in_hash;
163

```

```

164     ngx\_hash\_t                variables_hash;
165
166     ngx\_array\_t               variables;        /* ngx\_http\_variable\_t */
167     ngx\_uint\_t                ncaptures;
168
169     ngx\_uint\_t                server_names_hash_max_size;
170     ngx\_uint\_t                server_names_hash_bucket_size;
171
172     ngx\_uint\_t                variables_hash_max_size;
173     ngx\_uint\_t                variables_hash_bucket_size;
174
175     ngx\_hash\_keys\_arrays\_t    *variables_keys;
176
177     ngx\_array\_t               *ports;
178
179     ngx\_uint\_t                try_files;        /* unsigned try_files:1 */
180
181     ngx\_http\_phase\_t        phases[NGX_HTTP_LOG_PHASE + 1];
182 } ngx\_http\_core\_main\_conf\_t;
183
184
185 typedef struct {
186     /* array of the ngx\_http\_server\_name\_t, "server_name" directive */
187     ngx\_array\_t               server_names;
188
189     /* server ctx */
190     ngx\_http\_conf\_ctx\_t      *ctx;
191
192     ngx\_str\_t                 server_name;
193
194     size\_t                    connection_pool_size;
195     size\_t                    request_pool_size;
196     size\_t                    client_header_buffer_size;
197
198     ngx\_bufs\_t                large_client_header_buffers;
199
200     ngx\_msec\_t                client_header_timeout;
201
202     ngx\_flag\_t                ignore_invalid_headers;
203     ngx\_flag\_t                merge_slashes;
204     ngx\_flag\_t                underscores_in_headers;
205
206     unsigned                 listen:1;
207 #if (NGX_PCRE)
208     unsigned                 captures:1;
209 #endif
210
211     ngx\_http\_core\_loc\_conf\_t **named_locations;
212 } ngx\_http\_core\_srv\_conf\_t;
213
214
215 /* list of structures to find core_srv_conf quickly at run time */
216
217
218 typedef struct {
219 #if (NGX_PCRE)
220     ngx\_http\_regex\_t         *regex;
221 #endif
222     ngx\_http\_core\_srv\_conf\_t *server;    /* virtual name server conf */
223     ngx\_str\_t                 name;
224 } ngx\_http\_server\_name\_t;
225
226
227 typedef struct {
228     ngx\_hash\_combined\_t      names;
229
230     ngx\_uint\_t                nregex;
231     ngx\_http\_server\_name\_t   *regex;
232 } ngx\_http\_virtual\_names\_t;
233
234
235 struct ngx\_http\_addr\_conf\_s {
236     /* the default server configuration for this address:port */
237     ngx\_http\_core\_srv\_conf\_t *default_server;
238
239     ngx\_http\_virtual\_names\_t *virtual_names;

```

```

240
241 #if (NGX_HTTP_SSL)
242     unsigned                ssl:1;
243 #endif
244 #if (NGX_HTTP_SPDY)
245     unsigned                spdy:1;
246 #endif
247     unsigned                proxy_protocol:1;
248 };
249
250
251 typedef struct {
252     in_addr_t                addr;
253     ngx_http_addr_conf_t    conf;
254 } ngx_http_in_addr_t;
255
256
257 #if (NGX_HAVE_INET6)
258
259 typedef struct {
260     struct in6_addr          addr6;
261     ngx_http_addr_conf_t    conf;
262 } ngx_http_in6_addr_t;
263
264 #endif
265
266
267 typedef struct {
268     /* ngx_http_in_addr_t or ngx_http_in6_addr_t */
269     void                    *addrs;
270     ngx_uint_t              naddrs;
271 } ngx_http_port_t;
272
273
274 typedef struct {
275     ngx_int_t                family;
276     in_port_t                port;
277     ngx_array_t              addrs;    /* array of ngx_http_conf_addr_t */
278 } ngx_http_conf_port_t;
279
280
281 typedef struct {
282     ngx_http_listen_opt_t    opt;
283
284     ngx_hash_t               hash;
285     ngx_hash_wildcard_t      *wc_head;
286     ngx_hash_wildcard_t      *wc_tail;
287
288     #if (NGX_PCRE)
289     ngx_uint_t               nregex;
290     ngx_http_server_name_t   *regex;
291 #endif
292
293     /* the default server configuration for this address:port */
294     ngx_http_core_srv_conf_t *default_server;
295     ngx_array_t              servers; /* array of ngx_http_core_srv_conf_t */
296 } ngx_http_conf_addr_t;
297
298
299 typedef struct {
300     ngx_int_t                status;
301     ngx_int_t                overwrite;
302     ngx_http_complex_value_t value;
303     ngx_str_t                args;
304 } ngx_http_err_page_t;
305
306
307 typedef struct {
308     ngx_array_t              *lengths;
309     ngx_array_t              *values;
310     ngx_str_t                name;
311
312     unsigned                 code:10;
313     unsigned                 test_dir:1;
314 } ngx_http_try_file_t;
315

```

```

316 struct ngx_http_core_loc_conf_s {
317     ngx_str_t      name;          /* location name */
318
319
320     #if (NGX_PCRE)
321     ngx_http_regex_t *regex;
322     #endif
323
324     unsigned       noname:1;     /* "if () {}" block or limit_except */
325     unsigned       lmt_except:1;
326     unsigned       named:1;
327
328     unsigned       exact_match:1;
329     unsigned       noregex:1;
330
331     unsigned       auto_redirect:1;
332     #if (NGX_HTTP_GZIP)
333     unsigned       gzip_disable_msie6:2;
334     #if (NGX_HTTP_DEGRADATION)
335     unsigned       gzip_disable_degradation:2;
336     #endif
337     #endif
338
339     ngx_http_location_tree_node_t *static_locations;
340     #if (NGX_PCRE)
341     ngx_http_core_loc_conf_t **regex_locations;
342     #endif
343
344     /* pointer to the modules' loc_conf */
345     void           **loc_conf;
346
347     uint32_t       limit_except;
348     void           **limit_except_loc_conf;
349
350     ngx_http_handler_pt handler;
351
352     /* location name length for inclusive location with inherited alias */
353     size_t         alias;
354     ngx_str_t      root;          /* root, alias */
355     ngx_str_t      post_action;
356
357     ngx_array_t    *root_lengths;
358     ngx_array_t    *root_values;
359
360     ngx_array_t    *types;
361     ngx_hash_t     types_hash;
362     ngx_str_t      default_type;
363
364     off_t          client_max_body_size; /* client_max_body_size */
365     off_t          directio;          /* directio */
366     off_t          directio_alignment; /* directio_alignment */
367
368     size_t         client_body_buffer_size; /* client_body_buffer_size */
369     size_t         send_lowat;        /* send_lowat */
370     size_t         postpone_output;   /* postpone_output */
371     size_t         limit_rate;        /* limit_rate */
372     size_t         limit_rate_after;  /* limit_rate_after */
373     size_t         sendfile_max_chunk; /* sendfile_max_chunk */
374     size_t         read_ahead;        /* read_ahead */
375
376     ngx_msec_t     client_body_timeout; /* client_body_timeout */
377     ngx_msec_t     send_timeout;       /* send_timeout */
378     ngx_msec_t     keepalive_timeout; /* keepalive_timeout */
379     ngx_msec_t     lingering_time;    /* lingering_time */
380     ngx_msec_t     lingering_timeout; /* lingering_timeout */
381     ngx_msec_t     resolver_timeout;   /* resolver_timeout */
382
383     ngx_resolver_t *resolver;         /* resolver */
384
385     time_t         keepalive_header;   /* keepalive_timeout */
386
387     ngx_uint_t     keepalive_requests; /* keepalive_requests */
388     ngx_uint_t     keepalive_disable;  /* keepalive_disable */
389     ngx_uint_t     satisfy;            /* satisfy */
390     ngx_uint_t     lingering_close;    /* lingering_close */
391     ngx_uint_t     if_modified_since;  /* if_modified_since */

```

```

392     ngx_uint_t    max_ranges;                /* max_ranges */
393     ngx_uint_t    client_body_in_file_only; /* client_body_in_file_only */
394
395     ngx_flag_t    client_body_in_single_buffer;
396                                     /* client_body_in_singe_buffer */
397     ngx_flag_t    internal;                /* internal */
398     ngx_flag_t    sendfile;                /* sendfile */
399 #if (NGX_HAVE_FILE_AIO)
400     ngx_flag_t    aio;                    /* aio */
401 #endif
402     ngx_flag_t    tcp_nopush;              /* tcp_nopush */
403     ngx_flag_t    tcp_nodelay;            /* tcp_nodelay */
404     ngx_flag_t    reset_timedout_connection; /* reset_timedout_connection */
405     ngx_flag_t    server_name_in_redirect; /* server_name_in_redirect */
406     ngx_flag_t    port_in_redirect;        /* port_in_redirect */
407     ngx_flag_t    msie_padding;           /* msie_padding */
408     ngx_flag_t    msie_refresh;           /* msie_refresh */
409     ngx_flag_t    log_not_found;          /* log_not_found */
410     ngx_flag_t    log_subrequest;         /* log_subrequest */
411     ngx_flag_t    recursive_error_pages;  /* recursive_error_pages */
412     ngx_flag_t    server_tokens;          /* server_tokens */
413     ngx_flag_t    chunked_transfer_encoding; /* chunked_transfer_encoding */
414     ngx_flag_t    etag;                   /* etag */
415
416 #if (NGX_HTTP_GZIP)
417     ngx_flag_t    gzip_vary;              /* gzip_vary */
418
419     ngx_uint_t    gzip_http_version;      /* gzip_http_version */
420     ngx_uint_t    gzip_proxied;           /* gzip_proxied */
421
422 #if (NGX_PCRE)
423     ngx_array_t   *gzip_disable;          /* gzip_disable */
424 #endif
425 #endif
426
427 #if (NGX_HAVE_OPENAT)
428     ngx_uint_t    disable_symlinks;        /* disable_symlinks */
429     ngx_http_complex_value_t *disable_symlinks_from;
430 #endif
431
432     ngx_array_t   *error_pages;           /* error_page */
433     ngx_http_try_file_t *try_files;       /* try_files */
434
435     ngx_path_t    *client_body_temp_path; /* client_body_temp_path */
436
437     ngx_open_file_cache_t *open_file_cache;
438     time_t         open_file_cache_valid;
439     ngx_uint_t     open_file_cache_min_uses;
440     ngx_flag_t     open_file_cache_errors;
441     ngx_flag_t     open_file_cache_events;
442
443     ngx_log_t      *error_log;
444
445     ngx_uint_t     types_hash_max_size;
446     ngx_uint_t     types_hash_bucket_size;
447
448     ngx_queue_t    *locations;
449
450 #if 0
451     ngx_http_core_loc_conf_t *prev_location;
452 #endif
453 };
454
455
456 typedef struct {
457     ngx_queue_t    queue;
458     ngx_http_core_loc_conf_t *exact;
459     ngx_http_core_loc_conf_t *inclusive;
460     ngx_str_t      *name;
461     u_char          *file_name;
462     ngx_uint_t     line;
463     ngx_queue_t    list;
464 } ngx_http_location_queue_t;
465
466
467 struct ngx_http_location_tree_node_s {

```



```

468     ngx_http_location_tree_node_t *left;
469     ngx_http_location_tree_node_t *right;
470     ngx_http_location_tree_node_t *tree;
471
472     ngx_http_core_loc_conf_t *exact;
473     ngx_http_core_loc_conf_t *inclusive;
474
475     u_char auto_redirect;
476     u_char len;
477     u_char name[1];
478 };
479
480
481 void ngx_http_core_run_phases(ngx_http_request_t *r);
482 ngx_int_t ngx_http_core_generic_phase(ngx_http_request_t *r,
483     ngx_http_phase_handler_t *ph);
484 ngx_int_t ngx_http_core_rewrite_phase(ngx_http_request_t *r,
485     ngx_http_phase_handler_t *ph);
486 ngx_int_t ngx_http_core_find_config_phase(ngx_http_request_t *r,
487     ngx_http_phase_handler_t *ph);
488 ngx_int_t ngx_http_core_post_rewrite_phase(ngx_http_request_t *r,
489     ngx_http_phase_handler_t *ph);
490 ngx_int_t ngx_http_core_access_phase(ngx_http_request_t *r,
491     ngx_http_phase_handler_t *ph);
492 ngx_int_t ngx_http_core_post_access_phase(ngx_http_request_t *r,
493     ngx_http_phase_handler_t *ph);
494 ngx_int_t ngx_http_core_try_files_phase(ngx_http_request_t *r,
495     ngx_http_phase_handler_t *ph);
496 ngx_int_t ngx_http_core_content_phase(ngx_http_request_t *r,
497     ngx_http_phase_handler_t *ph);
498
499
500 void *ngx_http_test_content_type(ngx_http_request_t *r, ngx_hash_t *types_hash);
501 ngx_int_t ngx_http_set_content_type(ngx_http_request_t *r);
502 void ngx_http_set_exten(ngx_http_request_t *r);
503 ngx_int_t ngx_http_set_etag(ngx_http_request_t *r);
504 void ngx_http_weak_etag(ngx_http_request_t *r);
505 ngx_int_t ngx_http_send_response(ngx_http_request_t *r, ngx_uint_t status,
506     ngx_str_t *ct, ngx_http_complex_value_t *cv);
507 u_char *ngx_http_map_uri_to_path(ngx_http_request_t *r, ngx_str_t *name,
508     size_t *root_length, size_t reserved);
509 ngx_int_t ngx_http_auth_basic_user(ngx_http_request_t *r);
510 #if (NGX_HTTP_GZIP)
511 ngx_int_t ngx_http_gzip_ok(ngx_http_request_t *r);
512 #endif
513
514
515 ngx_int_t ngx_http_subrequest(ngx_http_request_t *r,
516     ngx_str_t *uri, ngx_str_t *args, ngx_http_request_t **sr,
517     ngx_http_post_subrequest_t *psr, ngx_uint_t flags);
518 ngx_int_t ngx_http_internal_redirect(ngx_http_request_t *r,
519     ngx_str_t *uri, ngx_str_t *args);
520 ngx_int_t ngx_http_named_location(ngx_http_request_t *r, ngx_str_t *name);
521
522
523 ngx_http_cleanup_t *ngx_http_cleanup_add(ngx_http_request_t *r, size_t size);
524
525
526 typedef ngx_int_t (*ngx_http_output_header_filter_pt)(ngx_http_request_t *r);
527 typedef ngx_int_t (*ngx_http_output_body_filter_pt)
528     (ngx_http_request_t *r, ngx_chain_t *chain);
529
530
531 ngx_int_t ngx_http_output_filter(ngx_http_request_t *r, ngx_chain_t *chain);
532 ngx_int_t ngx_http_write_filter(ngx_http_request_t *r, ngx_chain_t *chain);
533
534
535 ngx_int_t ngx_http_set_disable_symlinks(ngx_http_request_t *r,
536     ngx_http_core_loc_conf_t *clcf, ngx_str_t *path, ngx_open_file_info_t *of);
537
538 ngx_int_t ngx_http_get_forwarded_addr(ngx_http_request_t *r, ngx_addr_t *addr,
539     ngx_array_t *headers, ngx_str_t *value, ngx_array_t *proxies,
540     int recursive);
541
542
543 extern ngx_module_t ngx_http_core_module;

```

```

544 extern ngx\_uint\_t ngx\_http\_max\_module;
546
547 extern ngx\_str\_t ngx\_http\_core\_get\_method;
548
549
550 #define ngx\_http\_clear\_content\_length(r) \
551 \
552     r->headers_out.content_length_n = -1; \
553     if (r->headers_out.content_length) { \
554         r->headers_out.content_length->hash = 0; \
555         r->headers_out.content_length = NULL; \
556     }
557
558 #define ngx\_http\_clear\_accept\_ranges(r) \
559 \
560     r->allow_ranges = 0; \
561     if (r->headers_out.accept_ranges) { \
562         r->headers_out.accept_ranges->hash = 0; \
563         r->headers_out.accept_ranges = NULL; \
564     }
565
566 #define ngx\_http\_clear\_last\_modified(r) \
567 \
568     r->headers_out.last_modified_time = -1; \
569     if (r->headers_out.last_modified) { \
570         r->headers_out.last_modified->hash = 0; \
571         r->headers_out.last_modified = NULL; \
572     }
573
574 #define ngx\_http\_clear\_location(r) \
575 \
576     if (r->headers_out.location) { \
577         r->headers_out.location->hash = 0; \
578         r->headers_out.location = NULL; \
579     }
580
581 #define ngx\_http\_clear\_etag(r) \
582 \
583     if (r->headers_out.etag) { \
584         r->headers_out.etag->hash = 0; \
585         r->headers_out.etag = NULL; \
586     }
587
588
589 #endif /* NGX\_HTTP\_CORE\_H\_INCLUDED */

```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_inet.h - nginx-1.7.10

### Data types defined

- [ngx\\_addr\\_t](#)
- [ngx\\_cidr\\_t](#)
- [ngx\\_in6\\_cidr\\_t](#)
- [ngx\\_in\\_cidr\\_t](#)
- [ngx\\_url\\_t](#)

### Macros defined

- [NGX\\_INET6\\_ADDRSTRLEN](#)
- [NGX\\_INET\\_ADDRSTRLEN](#)
- [NGX\\_SOCKADDRLEN](#)
- [NGX\\_SOCKADDRLEN](#)
- [NGX\\_SOCKADDR\\_STRLEN](#)
- [NGX\\_SOCKADDR\\_STRLEN](#)
- [NGX\\_UNIX\\_ADDRSTRLEN](#)
- [\\_NGX\\_INET\\_H\\_INCLUDED](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_INET_H_INCLUDED
9 #define _NGX_INET_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 /*
17  * TODO: autoconfigure NGX\_SOCKADDRLEN and NGX\_SOCKADDR\_STRLEN as
18  * sizeof(struct sockaddr_storage)
19  * sizeof(struct sockaddr_un)
20  * sizeof(struct sockaddr_in6)
21  * sizeof(struct sockaddr_in)
22  */
23
24 #define NGX_INET_ADDRSTRLEN (sizeof("255.255.255.255") - 1)
25 #define NGX_INET6_ADDRSTRLEN (sizeof("ffff:ffff:ffff:ffff:ffff:ffff:255.255.255.255") - 1)
26 #define NGX_UNIX_ADDRSTRLEN (sizeof(struct sockaddr_un) - offsetof(struct sockaddr_un, sun_path))
27
28
29
30 #if (NGX_HAVE_UNIX_DOMAIN)
31 #define NGX_SOCKADDR_STRLEN (sizeof("unix:") - 1 + NGX\_UNIX\_ADDRSTRLEN)
32 #else
```

```

33 #define NGX_SOCKADDR_STRLEN (NGX_INET6_ADDRSTRLEN + sizeof("[]:65535") - 1)
34 #endif
35
36 #if (NGX_HAVE_UNIX_DOMAIN)
37 #define NGX_SOCKADDRLEN sizeof(struct sockaddr_un)
38 #else
39 #define NGX_SOCKADDRLEN 512
40 #endif
41
42
43 typedef struct {
44     in_addr_t      addr;
45     in_addr_t      mask;
46 } ngx_in_cidr_t;
47
48
49 #if (NGX_HAVE_INET6)
50
51 typedef struct {
52     struct in6_addr  addr;
53     struct in6_addr  mask;
54 } ngx_in6_cidr_t;
55
56 #endif
57
58
59 typedef struct {
60     ngx_uint_t      family;
61     union {
62         ngx_in_cidr_t  in;
63 #if (NGX_HAVE_INET6)
64         ngx_in6_cidr_t in6;
65 #endif
66     } u;
67 } ngx_cidr_t;
68
69
70 typedef struct {
71     struct sockaddr *sockaddr;
72     socklen_t      socklen;
73     ngx_str_t      name;
74 } ngx_addr_t;
75
76
77 typedef struct {
78     ngx_str_t      url;
79     ngx_str_t      host;
80     ngx_str_t      port_text;
81     ngx_str_t      uri;
82
83     in_port_t      port;
84     in_port_t      default_port;
85     int            family;
86
87     unsigned       listen:1;
88     unsigned       uri_part:1;
89     unsigned       no_resolve:1;
90     unsigned       one_addr:1; /* compatibility */
91
92     unsigned       no_port:1;
93     unsigned       wildcard:1;
94
95     socklen_t      socklen;
96     u_char         sockaddr[NGX_SOCKADDRLEN];
97
98     ngx_addr_t     *addrs;
99     ngx_uint_t     naddrs;
100
101     char           *err;
102 } ngx_url_t;
103
104
105 in_addr_t ngx_inet_addr(u_char *text, size_t len);
106 #if (NGX_HAVE_INET6)
107 ngx_int_t ngx_inet6_addr(u_char *p, size_t len, u_char *addr);
108 size_t ngx_inet6_ntop(u_char *p, u_char *text, size_t len);

```

```
109 #endif
110 size_t ngx_sock_ntop(struct sockaddr *sa, socklen_t socklen, u_char *text,
111     size_t len, ngx_uint_t port);
112 size_t ngx_inet_ntop(int family, void *addr, u_char *text, size_t len);
113 ngx_int_t ngx_ptocidr(ngx_str_t *text, ngx_cidr_t *cidr);
114 ngx_int_t ngx_parse_addr(ngx_pool_t *pool, ngx_addr_t *addr, u_char *text,
115     size_t len);
116 ngx_int_t ngx_parse_url(ngx_pool_t *pool, ngx_url_t *u);
117 ngx_int_t ngx_inet_resolve_host(ngx_pool_t *pool, ngx_url_t *u);
118 ngx_int_t ngx_cmp_sockaddr(struct sockaddr *sa1, socklen_t slen1,
119     struct sockaddr *sa2, socklen_t slen2, ngx_uint_t cmp_port);
120
121
122 #endif /* NGX_INET_H_INCLUDED */
```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_inet.c - nginx-1.7.10

### Functions defined

- [ngx\\_cmp\\_sockaddr](#)
- [ngx\\_inet6\\_addr](#)
- [ngx\\_inet6\\_ntop](#)
- [ngx\\_inet\\_addr](#)
- [ngx\\_inet\\_ntop](#)
- [ngx\\_inet\\_resolve\\_host](#)
- [ngx\\_inet\\_resolve\\_host](#)
- [ngx\\_parse\\_addr](#)
- [ngx\\_parse\\_inet6\\_url](#)
- [ngx\\_parse\\_inet\\_url](#)
- [ngx\\_parse\\_unix\\_domain\\_url](#)
- [ngx\\_parse\\_url](#)
- [ngx\\_ptocidr](#)
- [ngx\\_sock\\_ntop](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10
11
12 static ngx_int_t ngx_parse_unix_domain_url(ngx_pool_t *pool, ngx_url_t *u);
13 static ngx_int_t ngx_parse_inet_url(ngx_pool_t *pool, ngx_url_t *u);
14 static ngx_int_t ngx_parse_inet6_url(ngx_pool_t *pool, ngx_url_t *u);
15
16
17 in_addr_t
18 ngx_inet_addr(u_char *text, size_t len)
19 {
20     u_char      *p, c;
21     in_addr_t    addr;
22     ngx_uint_t  octet, n;
23
24     addr = 0;
25     octet = 0;
26     n = 0;
27
28     for (p = text; p < text + len; p++) {
29
30         c = *p;
31
32         if (c >= '0' && c <= '9') {
33             octet = octet * 10 + (c - '0');
```

```

34     continue;
35 }
36
37 if (c == '.' && octet < 256) {
38     addr = (addr << 8) + octet;
39     octet = 0;
40     n++;
41     continue;
42 }
43
44 return INADDR_NONE;
45 }
46
47 if (n == 3 && octet < 256) {
48     addr = (addr << 8) + octet;
49     return htonl(addr);
50 }
51
52 return INADDR_NONE;
53 }
54
55
56 #if (NGX_HAVE_INET6)
57
58 ngx_int_t
59 ngx_inet6_addr(u_char *p, size_t len, u_char *addr)
60 {
61     u_char    c, *zero, *digit, *s, *d;
62     size_t    len4;
63     ngx_uint_t n, nibbles, word;
64
65     if (len == 0) {
66         return NGX_ERROR;
67     }
68
69     zero = NULL;
70     digit = NULL;
71     len4 = 0;
72     nibbles = 0;
73     word = 0;
74     n = 8;
75
76     if (p[0] == ':') {
77         p++;
78         len--;
79     }
80
81     for (/* void */; len; len--) {
82         c = *p++;
83
84         if (c == ':') {
85             if (nibbles) {
86                 digit = p;
87                 len4 = len;
88                 *addr++ = (u_char) (word >> 8);
89                 *addr++ = (u_char) (word & 0xff);
90
91                 if (--n) {
92                     nibbles = 0;
93                     word = 0;
94                     continue;
95                 }
96             }
97             else {
98                 if (zero == NULL) {
99                     digit = p;
100                    len4 = len;
101                    zero = addr;
102                    continue;
103                }
104            }
105
106            return NGX_ERROR;
107        }
108
109        if (c == '.' && nibbles) {

```

```

110     if (n < 2 || digit == NULL) {
111         return NGX\_ERROR;
112     }
113
114     word = ngx\_inet\_addr(digit, len4 - 1);
115     if (word == INADDR\_NONE) {
116         return NGX\_ERROR;
117     }
118
119     word = ntohl(word);
120     *addr++ = (u_char) ((word >> 24) & 0xff);
121     *addr++ = (u_char) ((word >> 16) & 0xff);
122     n--;
123     break;
124 }
125
126 if (++nibbles > 4) {
127     return NGX\_ERROR;
128 }
129
130 if (c >= '0' && c <= '9') {
131     word = word * 16 + (c - '0');
132     continue;
133 }
134
135 c |= 0x20;
136
137 if (c >= 'a' && c <= 'f') {
138     word = word * 16 + (c - 'a') + 10;
139     continue;
140 }
141
142 return NGX\_ERROR;
143 }
144
145 if (nibbles == 0 && zero == NULL) {
146     return NGX\_ERROR;
147 }
148
149 *addr++ = (u_char) (word >> 8);
150 *addr++ = (u_char) (word & 0xff);
151
152 if (--n) {
153     if (zero) {
154         n *= 2;
155         s = addr - 1;
156         d = s + n;
157         while (s >= zero) {
158             *d-- = *s--;
159         }
160         ngx\_memzero(zero, n);
161         return NGX\_OK;
162     }
163 } else {
164     if (zero == NULL) {
165         return NGX\_OK;
166     }
167 }
168 }
169
170 return NGX\_ERROR;
171 }
172
173 #endif
174
175
176 size_t
177 ngx\_sock\_ntop(struct sockaddr *sa, socklen_t socklen, u_char *text, size_t len,
178 ngx\_uint\_t port)
179 {
180     u_char                *p;
181     struct sockaddr_in     *sin;
182     #if (NGX_HAVE_INET6)
183     size_t                n;
184     struct sockaddr_in6   *sin6;
185     #endif

```



```

186 #if (NGX_HAVE_UNIX_DOMAIN)
187     struct sockaddr_un  *saun;
188 #endif
189
190     switch (sa->sa_family) {
191
192     case AF_INET:
193
194         sin = (struct sockaddr_in *) sa;
195         p = (u_char *) &sin->sin_addr;
196
197         if (port) {
198             p = ngx_snprintf(text, len, "%ud.%ud.%ud.%ud:%d",
199                             p[0], p[1], p[2], p[3], ntohs(sin->sin_port));
200         } else {
201             p = ngx_snprintf(text, len, "%ud.%ud.%ud.%ud",
202                             p[0], p[1], p[2], p[3]);
203         }
204
205         return (p - text);
206
207 #if (NGX_HAVE_INET6)
208
209     case AF_INET6:
210
211         sin6 = (struct sockaddr_in6 *) sa;
212
213         n = 0;
214
215         if (port) {
216             text[n++] = '[';
217         }
218
219         n = ngx_inet6_ntop(sin6->sin6_addr.s6_addr, &text[n], len);
220
221         if (port) {
222             n = ngx_sprintf(&text[1 + n], "]:%d",
223                             ntohs(sin6->sin6_port)) - text;
224         }
225
226         return n;
227 #endif
228
229 #if (NGX_HAVE_UNIX_DOMAIN)
230
231     case AF_UNIX:
232         saun = (struct sockaddr_un *) sa;
233
234         /* on Linux sockaddr might not include sun_path at all */
235
236         if (socklen <= (socklen_t) offsetof(struct sockaddr_un, sun_path)) {
237             p = ngx_snprintf(text, len, "unix:%Z");
238         } else {
239             p = ngx_snprintf(text, len, "unix:%s%Z", saun->sun_path);
240         }
241
242         /* we do not include trailing zero in address length */
243
244         return (p - text - 1);
245
246 #endif
247
248     default:
249         return 0;
250     }
251 }
252
253
254
255 size_t
256 ngx_inet_ntop(int family, void *addr, u_char *text, size_t len)
257 {
258     u_char  *p;
259
260     switch (family) {
261

```

```

262     case AF_INET:
263
264         p = addr;
265
266         return ngx_sprintf(text, len, "%ud.%ud.%ud.%ud",
267                               p[0], p[1], p[2], p[3])
268             - text;
269
270 #if (NGX_HAVE_INET6)
271
272     case AF_INET6:
273         return ngx_inet6_ntop(addr, text, len);
274
275 #endif
276
277     default:
278         return 0;
279 }
280 }
281
282 #if (NGX_HAVE_INET6)
283
284 size_t
285 ngx_inet6_ntop(u_char *p, u_char *text, size_t len)
286 {
287     u_char      *dst;
288     size_t      max, n;
289     ngx_uint_t  i, zero, last;
290
291     if (len < NGX_INET6_ADDRSTRLEN) {
292         return 0;
293     }
294
295     zero = (ngx_uint_t) -1;
296     last = (ngx_uint_t) -1;
297     max = 1;
298     n = 0;
299
300     for (i = 0; i < 16; i += 2) {
301
302         if (p[i] || p[i + 1]) {
303
304             if (max < n) {
305                 zero = last;
306                 max = n;
307             }
308
309             n = 0;
310             continue;
311         }
312
313         if (n++ == 0) {
314             last = i;
315         }
316     }
317
318     if (max < n) {
319         zero = last;
320         max = n;
321     }
322
323     dst = text;
324     n = 16;
325
326     if (zero == 0) {
327
328         if ((max == 5 && p[10] == 0xff && p[11] == 0xff)
329             || (max == 6)
330             || (max == 7 && p[14] != 0 && p[15] != 1))
331         {
332             n = 12;
333         }
334
335         *dst++ = ':';
336     }
337 }

```

```

338
339     for (i = 0; i < n; i += 2) {
340
341         if (i == zero) {
342             *dst++ = ':';
343             i += (max - 1) * 2;
344             continue;
345         }
346
347         dst = ngx_sprintf(dst, "%uxi", p[i] * 256 + p[i + 1]);
348
349         if (i < 14) {
350             *dst++ = ':';
351         }
352     }
353
354     if (n == 12) {
355         dst = ngx_sprintf(dst, "%ud.%ud.%ud.%ud", p[12], p[13], p[14], p[15]);
356     }
357
358     return dst - text;
359 }
360
361 #endif
362
363
364 ngx_int_t
365 ngx_ptocidr(ngx_str_t *text, ngx_cidr_t *cidr)
366 {
367     u_char      *addr, *mask, *last;
368     size_t      len;
369     ngx_int_t    shift;
370 #if (NGX_HAVE_INET6)
371     ngx_int_t    rc;
372     ngx_uint_t   s, i;
373 #endif
374
375     addr = text->data;
376     last = addr + text->len;
377
378     mask = ngx_strlchr(addr, last, '/');
379     len = (mask ? mask : last) - addr;
380
381     cidr->u.in.addr = ngx_inet_addr(addr, len);
382
383     if (cidr->u.in.addr != INADDR_NONE) {
384         cidr->family = AF_INET;
385
386         if (mask == NULL) {
387             cidr->u.in.mask = 0xffffffff;
388             return NGX_OK;
389         }
390
391 #if (NGX_HAVE_INET6)
392     } else if (ngx_inet6_addr(addr, len, cidr->u.in6.addr.s6_addr) == NGX_OK) {
393         cidr->family = AF_INET6;
394
395         if (mask == NULL) {
396             ngx_memset(cidr->u.in6.mask.s6_addr, 0xff, 16);
397             return NGX_OK;
398         }
399
400 #endif
401     } else {
402         return NGX_ERROR;
403     }
404
405     mask++;
406
407     shift = ngx_atoi(mask, last - mask);
408     if (shift == NGX_ERROR) {
409         return NGX_ERROR;
410     }
411
412     switch (cidr->family) {
413

```

```

414 #if (NGX_HAVE_INET6)
415     case AF_INET6:
416         if (shift > 128) {
417             return NGX\_ERROR;
418         }
419
420         addr = cidr->u.in6.addr.s6_addr;
421         mask = cidr->u.in6.mask.s6_addr;
422         rc = NGX\_OK;
423
424         for (i = 0; i < 16; i++) {
425
426             s = (shift > 8) ? 8 : shift;
427             shift -= s;
428
429             mask[i] = (u_char) (0xffu << (8 - s));
430
431             if (addr[i] != (addr[i] & mask[i])) {
432                 rc = NGX\_DONE;
433                 addr[i] &= mask[i];
434             }
435         }
436
437         return rc;
438 #endif
439
440     default: /* AF_INET */
441         if (shift > 32) {
442             return NGX\_ERROR;
443         }
444
445         if (shift) {
446             cidr->u.in.mask = htonl((uint32_t) (0xffffffffu << (32 - shift)));
447
448         } else {
449             /* x86 compilers use a shl instruction that shifts by modulo 32 */
450             cidr->u.in.mask = 0;
451         }
452
453         if (cidr->u.in.addr == (cidr->u.in.addr & cidr->u.in.mask)) {
454             return NGX\_OK;
455         }
456
457         cidr->u.in.addr &= cidr->u.in.mask;
458
459         return NGX\_DONE;
460     }
461 }
462
463
464 ngx\_int\_t
465 ngx\_parse\_addr(ngx\_pool\_t *pool, ngx\_addr\_t *addr, u_char *text, size\_t len)
466 {
467     in\_addr\_t          inaddr;
468     ngx\_uint\_t        family;
469     struct sockaddr_in *sin;
470 #if (NGX_HAVE_INET6)
471     struct in6_addr    inaddr6;
472     struct sockaddr_in6 *sin6;
473
474     /*
475      * prevent MSVC8 warning:
476      * potentially uninitialized local variable 'inaddr6' used
477      */
478     ngx\_memzero(&inaddr6, sizeof(struct in6_addr));
479 #endif
480
481     inaddr = ngx\_inet\_addr(text, len);
482
483     if (inaddr != INADDR\_NONE) {
484         family = AF_INET;
485         len = sizeof(struct sockaddr_in);
486
487     #if (NGX_HAVE_INET6)
488     } else if (ngx\_inet6\_addr(text, len, inaddr6.s6_addr) == NGX\_OK) {
489         family = AF_INET6;

```

```

490         len = sizeof(struct sockaddr_in6);
491
492     #endif
493     } else {
494         return NGX\_DECLINED;
495     }
496
497     addr->sockaddr = ngx\_palloc(pool, len);
498     if (addr->sockaddr == NULL) {
499         return NGX\_ERROR;
500     }
501
502     addr->sockaddr->sa_family = (u_char) family;
503     addr->socklen = len;
504
505     switch (family) {
506
507     #if (NGX_HAVE_INET6)
508     case AF_INET6:
509         sin6 = (struct sockaddr_in6 *) addr->sockaddr;
510         ngx\_memcpy(sin6->sin6_addr.s6_addr, inaddr6.s6_addr, 16);
511         break;
512     #endif
513
514     default: /* AF_INET */
515         sin = (struct sockaddr_in *) addr->sockaddr;
516         sin->sin_addr.s_addr = inaddr;
517         break;
518     }
519
520     return NGX\_OK;
521 }
522
523
524 ngx\_int\_t
525 ngx\_parse\_url(ngx\_pool\_t *pool, ngx\_url\_t *u)
526 {
527     u_char *p;
528
529     p = u->url.data;
530
531     if (ngx\_strncasecmp(p, (u_char *) "unix:", 5) == 0) {
532         return ngx\_parse\_unix\_domain\_url(pool, u);
533     }
534
535     if (p[0] == '[') {
536         return ngx\_parse\_inet6\_url(pool, u);
537     }
538
539     return ngx\_parse\_inet\_url(pool, u);
540 }
541
542
543 static ngx\_int\_t
544 ngx\_parse\_unix\_domain\_url(ngx\_pool\_t *pool, ngx\_url\_t *u)
545 {
546     #if (NGX_HAVE_UNIX_DOMAIN)
547     u_char          *path, *uri, *last;
548     size_t          len;
549     struct sockaddr_un *saun;
550
551     len = u->url.len;
552     path = u->url.data;
553
554     path += 5;
555     len -= 5;
556
557     if (u->uri_part) {
558
559         last = path + len;
560         uri = ngx\_strlchr(path, last, ':');
561
562         if (uri) {
563             len = uri - path;
564             uri++;
565             u->uri.len = last - uri;

```

```

566         u->uri.data = uri;
567     }
568 }
569
570 if (len == 0) {
571     u->err = "no path in the unix domain socket";
572     return NGX\_ERROR;
573 }
574
575 u->host.len = len++;
576 u->host.data = path;
577
578 if (len > sizeof(saun->sun_path)) {
579     u->err = "too long path in the unix domain socket";
580     return NGX\_ERROR;
581 }
582
583 u->socklen = sizeof(struct sockaddr_un);
584 saun = (struct sockaddr_un *) &u->sockaddr;
585 saun->sun_family = AF_UNIX;
586 (void) ngx\_cpystirn((u_char *) saun->sun_path, path, len);
587
588 u->addrs = ngx\_palloc(pool, sizeof(ngx\_addr\_t));
589 if (u->addrs == NULL) {
590     return NGX\_ERROR;
591 }
592
593 saun = ngx\_palloc(pool, sizeof(struct sockaddr_un));
594 if (saun == NULL) {
595     return NGX\_ERROR;
596 }
597
598 u->family = AF_UNIX;
599 u->naddrs = 1;
600
601 saun->sun_family = AF_UNIX;
602 (void) ngx\_cpystirn((u_char *) saun->sun_path, path, len);
603
604 u->addrs[0].sockaddr = (struct sockaddr *) saun;
605 u->addrs[0].socklen = sizeof(struct sockaddr_un);
606 u->addrs[0].name.len = len + 4;
607 u->addrs[0].name.data = u->url.data;
608
609 return NGX\_OK;
610
611 #else
612
613     u->err = "the unix domain sockets are not supported on this platform";
614
615     return NGX\_ERROR;
616
617 #endif
618 }
619
620
621 static ngx\_int\_t
622 ngx\_parse\_inet\_url(ngx\_pool\_t *pool, ngx\_url\_t *u)
623 {
624     u_char          *p, *host, *port, *last, *uri, *args;
625     size_t          len;
626     ngx\_int\_t      n;
627     struct sockaddr_in *sin;
628 #if (NGX_HAVE_INET6)
629     struct sockaddr_in6 *sin6;
630 #endif
631
632     u->socklen = sizeof(struct sockaddr_in);
633     sin = (struct sockaddr_in *) &u->sockaddr;
634     sin->sin_family = AF_INET;
635
636     u->family = AF_INET;
637
638     host = u->url.data;
639
640     last = host + u->url.len;
641

```

```

642 port = ngx_strlchr(host, last, ':');
643
644 uri = ngx_strlchr(host, last, '/');
645
646 args = ngx_strlchr(host, last, '?');
647
648 if (args) {
649     if (uri == NULL || args < uri) {
650         uri = args;
651     }
652 }
653
654 if (uri) {
655     if (u->listen || !u->uri_part) {
656         u->err = "invalid host";
657         return NGX_ERROR;
658     }
659
660     u->uri.len = last - uri;
661     u->uri.data = uri;
662
663     last = uri;
664
665     if (uri < port) {
666         port = NULL;
667     }
668 }
669
670 if (port) {
671     port++;
672
673     len = last - port;
674
675     n = ngx_atoi(port, len);
676
677     if (n < 1 || n > 65535) {
678         u->err = "invalid port";
679         return NGX_ERROR;
680     }
681
682     u->port = (in_port_t) n;
683     sin->sin_port = htons((in_port_t) n);
684
685     u->port_text.len = len;
686     u->port_text.data = port;
687
688     last = port - 1;
689
690 } else {
691     if (uri == NULL) {
692
693         if (u->listen) {
694
695             /* test value as port only */
696
697             n = ngx_atoi(host, last - host);
698
699             if (n != NGX_ERROR) {
700
701                 if (n < 1 || n > 65535) {
702                     u->err = "invalid port";
703                     return NGX_ERROR;
704                 }
705
706                 u->port = (in_port_t) n;
707                 sin->sin_port = htons((in_port_t) n);
708
709                 u->port_text.len = last - host;
710                 u->port_text.data = host;
711
712                 u->wildcard = 1;
713
714                 return NGX_OK;
715             }
716         }
717     }

```

```

718     u->no_port = 1;
719     u->port = u->default_port;
720     sin->sin_port = htons(u->default_port);
721 }
722
723 len = last - host;
724
725 if (len == 0) {
726     u->err = "no host";
727     return NGX\_ERROR;
728 }
729
730 u->host.len = len;
731 u->host.data = host;
732
733 if (u->listen && len == 1 && *host == '*') {
734     sin->sin_addr.s_addr = INADDR_ANY;
735     u->wildcard = 1;
736     return NGX\_OK;
737 }
738
739 sin->sin_addr.s_addr = ngx\_inet\_addr(host, len);
740
741 if (sin->sin_addr.s_addr != INADDR\_NONE) {
742     if (sin->sin_addr.s_addr == INADDR\_ANY) {
743         u->wildcard = 1;
744     }
745
746     u->naddrs = 1;
747
748     u->addrs = ngx\_pcalloc(pool, sizeof(ngx\_addr\_t));
749     if (u->addrs == NULL) {
750         return NGX\_ERROR;
751     }
752
753     sin = ngx\_pcalloc(pool, sizeof(struct sockaddr_in));
754     if (sin == NULL) {
755         return NGX\_ERROR;
756     }
757
758     ngx\_memcpy(sin, u->sockaddr, sizeof(struct sockaddr_in));
759
760     u->addrs[0].sockaddr = (struct sockaddr *) sin;
761     u->addrs[0].socklen = sizeof(struct sockaddr_in);
762
763     p = ngx\_pnalloc(pool, u->host.len + sizeof(":65535") - 1);
764     if (p == NULL) {
765         return NGX\_ERROR;
766     }
767
768     u->addrs[0].name.len = ngx\_sprintf(p, "%V:%d",
769                                     &u->host, u->port) - p;
770     u->addrs[0].name.data = p;
771
772     return NGX\_OK;
773 }
774
775 if (u->no_resolve) {
776     return NGX\_OK;
777 }
778
779 if (ngx\_inet\_resolve\_host(pool, u) != NGX\_OK) {
780     return NGX\_ERROR;
781 }
782
783 u->family = u->addrs[0].sockaddr->sa_family;
784 u->socklen = u->addrs[0].socklen;
785 ngx\_memcpy(u->sockaddr, u->addrs[0].sockaddr, u->addrs[0].socklen);
786
787 switch (u->family) {
788
789 #if (NGX\_HAVE\_INET6)
790 case AF_INET6:
791     sin6 = (struct sockaddr_in6 *) &u->sockaddr;

```



```

794         if (IN6_IS_ADDR_UNSPECIFIED(&sin6->sin6_addr)) {
795             u->wildcard = 1;
796         }
797     }
798
799     break;
800 #endif
801
802     default: /* AF_INET */
803         sin = (struct sockaddr_in *) &u->sockaddr;
804
805         if (sin->sin_addr.s_addr == INADDR_ANY) {
806             u->wildcard = 1;
807         }
808
809         break;
810     }
811
812     return NGX_OK;
813 }
814
815
816 static ngx_int_t
817 ngx_parse_inet6_url(ngx_pool_t *pool, ngx_url_t *u)
818 {
819     #if (NGX_HAVE_INET6)
820     u_char          *p, *host, *port, *last, *uri;
821     size_t          len;
822     ngx_int_t       n;
823     struct sockaddr_in6 *sin6;
824
825     u->socklen = sizeof(struct sockaddr_in6);
826     sin6 = (struct sockaddr_in6 *) &u->sockaddr;
827     sin6->sin6_family = AF_INET6;
828
829     host = u->url.data + 1;
830
831     last = u->url.data + u->url.len;
832
833     p = ngx_strlchr(host, last, ']');
834
835     if (p == NULL) {
836         u->err = "invalid host";
837         return NGX_ERROR;
838     }
839
840     if (last - p) {
841
842         port = p + 1;
843
844         uri = ngx_strlchr(port, last, '/');
845
846         if (uri) {
847             if (u->listen || !u->uri_part) {
848                 u->err = "invalid host";
849                 return NGX_ERROR;
850             }
851
852             u->uri.len = last - uri;
853             u->uri.data = uri;
854
855             last = uri;
856         }
857
858         if (*port == ':') {
859             port++;
860
861             len = last - port;
862
863             n = ngx_atoi(port, len);
864
865             if (n < 1 || n > 65535) {
866                 u->err = "invalid port";
867                 return NGX_ERROR;
868             }
869

```

```

870         u->port = (in_port_t) n;
871         sin6->sin6_port = htons((in_port_t) n);
872
873         u->port_text.len = len;
874         u->port_text.data = port;
875
876     } else {
877         u->no_port = 1;
878         u->port = u->default_port;
879         sin6->sin6_port = htons(u->default_port);
880     }
881 }
882
883 len = p - host;
884
885 if (len == 0) {
886     u->err = "no host";
887     return NGX_ERROR;
888 }
889
890 u->host.len = len + 2;
891 u->host.data = host - 1;
892
893 if (ngx_inet6_addr(host, len, sin6->sin6_addr.s6_addr) != NGX_OK) {
894     u->err = "invalid IPv6 address";
895     return NGX_ERROR;
896 }
897
898 if (IN6_IS_ADDR_UNSPECIFIED(&sin6->sin6_addr)) {
899     u->wildcard = 1;
900 }
901
902 u->family = AF_INET6;
903 u->naddrs = 1;
904
905 u->addrs = ngx_palloc(pool, sizeof(ngx_addr_t));
906 if (u->addrs == NULL) {
907     return NGX_ERROR;
908 }
909
910 sin6 = ngx_palloc(pool, sizeof(struct sockaddr_in6));
911 if (sin6 == NULL) {
912     return NGX_ERROR;
913 }
914
915 ngx_memcpy(sin6, u->sockaddr, sizeof(struct sockaddr_in6));
916
917 u->addrs[0].sockaddr = (struct sockaddr *) sin6;
918 u->addrs[0].socklen = sizeof(struct sockaddr_in6);
919
920 p = ngx_palloc(pool, u->host.len + sizeof(":65535") - 1);
921 if (p == NULL) {
922     return NGX_ERROR;
923 }
924
925 u->addrs[0].name.len = ngx_sprintf(p, "%v:%d",
926                                     &u->host, u->port) - p;
927 u->addrs[0].name.data = p;
928
929 return NGX_OK;
930
931 #else
932
933     u->err = "the INET6 sockets are not supported on this platform";
934
935     return NGX_ERROR;
936
937 #endif
938 }
939
940
941 #if (NGX_HAVE_GETADDRINFO && NGX_HAVE_INET6)
942
943 ngx_int_t
944 ngx_inet_resolve_host(ngx_pool_t *pool, ngx_url_t *u)
945 {

```

```

946     u_char          *p, *host;
947     size_t         len;
948     in_port_t      port;
949     ngx_uint_t     i;
950     struct addrinfo hints, *res, *rp;
951     struct sockaddr_in *sin;
952     struct sockaddr_in6 *sin6;
953
954     port = htons(u->port);
955
956     host = ngx_alloc(u->host.len + 1, pool->log);
957     if (host == NULL) {
958         return NGX_ERROR;
959     }
960
961     (void) ngx_cpymstrn(host, u->host.data, u->host.len + 1);
962
963     ngx_memzero(&hints, sizeof(struct addrinfo));
964     hints.ai_family = AF_UNSPEC;
965     hints.ai_socktype = SOCK_STREAM;
966     #ifdef AI_ADDRCONFIG
967     hints.ai_flags = AI_ADDRCONFIG;
968     #endif
969
970     if (getaddrinfo((char *) host, NULL, &hints, &res) != 0) {
971         u->err = "host not found";
972         ngx_free(host);
973         return NGX_ERROR;
974     }
975
976     ngx_free(host);
977
978     for (i = 0, rp = res; rp != NULL; rp = rp->ai_next) {
979
980         switch (rp->ai_family) {
981
982             case AF_INET:
983             case AF_INET6:
984                 break;
985
986             default:
987                 continue;
988         }
989
990         i++;
991     }
992
993     if (i == 0) {
994         u->err = "host not found";
995         goto failed;
996     }
997
998     /* MP: ngx_shared_palloc() */
999
1000     u->addrs = ngx_pcalloc(pool, i * sizeof(ngx_addr_t));
1001     if (u->addrs == NULL) {
1002         goto failed;
1003     }
1004
1005     u->naddrs = i;
1006
1007     i = 0;
1008
1009     /* AF_INET addresses first */
1010
1011     for (rp = res; rp != NULL; rp = rp->ai_next) {
1012
1013         if (rp->ai_family != AF_INET) {
1014             continue;
1015         }
1016
1017         sin = ngx_pcalloc(pool, rp->ai_addrlen);
1018         if (sin == NULL) {
1019             goto failed;
1020         }
1021

```

```

1022     ngx_memcpy(sin, rp->ai_addr, rp->ai_addrlen);
1023
1024     sin->sin_port = port;
1025
1026     u->addrs[i].sockaddr = (struct sockaddr *) sin;
1027     u->addrs[i].socklen = rp->ai_addrlen;
1028
1029     len = NGX_INET_ADDRSTRLEN + sizeof(":65535") - 1;
1030
1031     p = ngx_pnalloc(pool, len);
1032     if (p == NULL) {
1033         goto failed;
1034     }
1035
1036     len = ngx_sock_ntop((struct sockaddr *) sin, rp->ai_addrlen, p, len, 1);
1037
1038     u->addrs[i].name.len = len;
1039     u->addrs[i].name.data = p;
1040
1041     i++;
1042 }
1043
1044 for (rp = res; rp != NULL; rp = rp->ai_next) {
1045
1046     if (rp->ai_family != AF_INET6) {
1047         continue;
1048     }
1049
1050     sin6 = ngx_pcalloc(pool, rp->ai_addrlen);
1051     if (sin6 == NULL) {
1052         goto failed;
1053     }
1054
1055     ngx_memcpy(sin6, rp->ai_addr, rp->ai_addrlen);
1056
1057     sin6->sin6_port = port;
1058
1059     u->addrs[i].sockaddr = (struct sockaddr *) sin6;
1060     u->addrs[i].socklen = rp->ai_addrlen;
1061
1062     len = NGX_INET6_ADDRSTRLEN + sizeof("[:65535]") - 1;
1063
1064     p = ngx_pnalloc(pool, len);
1065     if (p == NULL) {
1066         goto failed;
1067     }
1068
1069     len = ngx_sock_ntop((struct sockaddr *) sin6, rp->ai_addrlen, p,
1070                        len, 1);
1071
1072     u->addrs[i].name.len = len;
1073     u->addrs[i].name.data = p;
1074
1075     i++;
1076 }
1077
1078 freeaddrinfo(res);
1079 return NGX_OK;
1080
1081 failed:
1082
1083 freeaddrinfo(res);
1084 return NGX_ERROR;
1085 }
1086
1087 #else /* !NGX_HAVE_GETADDRINFO || !NGX_HAVE_INET6 */
1088
1089 ngx_int_t
1090 ngx_inet_resolve_host(ngx_pool_t *pool, ngx_url_t *u)
1091 {
1092     u_char          *p, *host;
1093     size_t          len;
1094     in_port_t       port;
1095     in_addr_t       in_addr;
1096     ngx_uint_t      i;
1097     struct hostent  *h;

```

```

1098 struct sockaddr_in *sin;
1099
1100 /* AF_INET only */
1101
1102 port = htons(u->port);
1103
1104 in_addr = ngx_inet_addr(u->host.data, u->host.len);
1105
1106 if (in_addr == INADDR_NONE) {
1107     host = ngx_alloc(u->host.len + 1, pool->log);
1108     if (host == NULL) {
1109         return NGX_ERROR;
1110     }
1111
1112     (void) ngx_cpystn(host, u->host.data, u->host.len + 1);
1113
1114     h = gethostbyname((char *) host);
1115
1116     ngx_free(host);
1117
1118     if (h == NULL || h->h_addr_list[0] == NULL) {
1119         u->err = "host not found";
1120         return NGX_ERROR;
1121     }
1122
1123     for (i = 0; h->h_addr_list[i] != NULL; i++) { /* void */
1124
1125         /* MP: ngx_shared_palloc() */
1126
1127         u->addrs = ngx_palloc(pool, i * sizeof(ngx_addr_t));
1128         if (u->addrs == NULL) {
1129             return NGX_ERROR;
1130         }
1131
1132         u->naddrs = i;
1133
1134         for (i = 0; i < u->naddrs; i++) {
1135
1136             sin = ngx_palloc(pool, sizeof(struct sockaddr_in));
1137             if (sin == NULL) {
1138                 return NGX_ERROR;
1139             }
1140
1141             sin->sin_family = AF_INET;
1142             sin->sin_port = port;
1143             sin->sin_addr.s_addr = *(in_addr_t *) (h->h_addr_list[i]);
1144
1145             u->addrs[i].sockaddr = (struct sockaddr *) sin;
1146             u->addrs[i].socklen = sizeof(struct sockaddr_in);
1147
1148             len = NGX_INET_ADDRSTRLEN + sizeof(":65535") - 1;
1149
1150             p = ngx_palloc(pool, len);
1151             if (p == NULL) {
1152                 return NGX_ERROR;
1153             }
1154
1155             len = ngx_sock_ntop((struct sockaddr *) sin,
1156                                sizeof(struct sockaddr_in), p, len, 1);
1157
1158             u->addrs[i].name.len = len;
1159             u->addrs[i].name.data = p;
1160         }
1161     } else {
1162
1163         /* MP: ngx_shared_palloc() */
1164
1165         u->addrs = ngx_palloc(pool, sizeof(ngx_addr_t));
1166         if (u->addrs == NULL) {
1167             return NGX_ERROR;
1168         }
1169
1170
1171         sin = ngx_palloc(pool, sizeof(struct sockaddr_in));
1172         if (sin == NULL) {
1173             return NGX_ERROR;

```

```

1174     }
1175
1176     u->naddrs = 1;
1177
1178     sin->sin_family = AF_INET;
1179     sin->sin_port = port;
1180     sin->sin_addr.s_addr = in_addr;
1181
1182     u->addrs[0].sockaddr = (struct sockaddr *) sin;
1183     u->addrs[0].socklen = sizeof(struct sockaddr_in);
1184
1185     p = ngx_pnalloc(pool, u->host.len + sizeof(":65535") - 1);
1186     if (p == NULL) {
1187         return NGX_ERROR;
1188     }
1189
1190     u->addrs[0].name.len = ngx_sprintf(p, "%V:%d",
1191                                     &u->host, ntohs(port)) - p;
1192     u->addrs[0].name.data = p;
1193 }
1194
1195 return NGX_OK;
1196 }
1197
1198 #endif /* NGX_HAVE_GETADDRINFO && NGX_HAVE_INET6 */
1199
1200
1201 ngx_int_t
1202 ngx_cmp_sockaddr(struct sockaddr *sa1, socklen_t slen1,
1203                 struct sockaddr *sa2, socklen_t slen2, ngx_uint_t cmp_port)
1204 {
1205     struct sockaddr_in  *sin1, *sin2;
1206     #if (NGX_HAVE_INET6)
1207     struct sockaddr_in6 *sin61, *sin62;
1208     #endif
1209     #if (NGX_HAVE_UNIX_DOMAIN)
1210     struct sockaddr_un  *saun1, *saun2;
1211     #endif
1212
1213     if (sa1->sa_family != sa2->sa_family) {
1214         return NGX_DECLINED;
1215     }
1216
1217     switch (sa1->sa_family) {
1218
1219     #if (NGX_HAVE_INET6)
1220     case AF_INET6:
1221
1222         sin61 = (struct sockaddr_in6 *) sa1;
1223         sin62 = (struct sockaddr_in6 *) sa2;
1224
1225         if (cmp_port && sin61->sin6_port != sin62->sin6_port) {
1226             return NGX_DECLINED;
1227         }
1228
1229         if (ngx_memcmp(&sin61->sin6_addr, &sin62->sin6_addr, 16) != 0) {
1230             return NGX_DECLINED;
1231         }
1232
1233         break;
1234     #endif
1235
1236     #if (NGX_HAVE_UNIX_DOMAIN)
1237     case AF_UNIX:
1238
1239         /* TODO length */
1240
1241         saun1 = (struct sockaddr_un *) sa1;
1242         saun2 = (struct sockaddr_un *) sa2;
1243
1244         if (ngx_memcmp(&saun1->sun_path, &saun2->sun_path,
1245                     sizeof(saun1->sun_path))
1246             != 0)
1247         {
1248             return NGX_DECLINED;
1249         }

```

```
1250         break;
1251     #endif
1252
1253     default: /* AF_INET */
1254
1255         sin1 = (struct sockaddr_in *) sa1;
1256         sin2 = (struct sockaddr_in *) sa2;
1257
1258         if (cmp_port && sin1->sin_port != sin2->sin_port) {
1259             return NGX\_DECLINED;
1260         }
1261
1262         if (sin1->sin_addr.s_addr != sin2->sin_addr.s_addr) {
1263             return NGX\_DECLINED;
1264         }
1265
1266         break;
1267     }
1268
1269     return NGX\_OK;
1270 }
1271 }
```

[One Level Up](#)

[Top Level](#)

# src/http/nginx\_http\_variables.h - nginx-1.7.10

## Data types defined

- [ngx\\_http\\_get\\_variable\\_pt](#)
- [ngx\\_http\\_map\\_regex\\_t](#)
- [ngx\\_http\\_map\\_t](#)
- [ngx\\_http\\_regex\\_t](#)
- [ngx\\_http\\_regex\\_variable\\_t](#)
- [ngx\\_http\\_set\\_variable\\_pt](#)
- [ngx\\_http\\_variable\\_s](#)
- [ngx\\_http\\_variable\\_t](#)
- [ngx\\_http\\_variable\\_value\\_t](#)

## Macros defined

- [NGX\\_HTTP\\_VAR\\_CHANGEABLE](#)
- [NGX\\_HTTP\\_VAR\\_INDEXED](#)
- [NGX\\_HTTP\\_VAR\\_NOCACHEABLE](#)
- [NGX\\_HTTP\\_VAR\\_NOHASH](#)
- [\\_NGX\\_HTTP\\_VARIABLES\\_H\\_INCLUDED](#)
- [ngx\\_http\\_variable](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_HTTP_VARIABLES_H_INCLUDED
9 #define _NGX_HTTP_VARIABLES_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14 #include <ngx_http.h>
15
16
17 typedef ngx_variable_value_t ngx_http_variable_value_t;
18
19 #define ngx_http_variable(v)    { sizeof(v) - 1, 1, 0, 0, 0, (u_char *) v }
20
21 typedef struct ngx_http_variable_s ngx_http_variable_t;
22
23 typedef void (*ngx_http_set_variable_pt) (ngx_http_request_t *r,
24     ngx_http_variable_value_t *v, uintptr_t data);
25 typedef ngx_int_t (*ngx_http_get_variable_pt) (ngx_http_request_t *r,
26     ngx_http_variable_value_t *v, uintptr_t data);
27
```



```

28
29 #define NGX_HTTP_VAR_CHANGEABLE 1
30 #define NGX_HTTP_VAR_NOCACHEABLE 2
31 #define NGX_HTTP_VAR_INDEXED 4
32 #define NGX_HTTP_VAR_NOHASH 8
33
34
35 struct ngx_http_variable_s {
36     ngx_str_t name; /* must be first to build the hash */
37     ngx_http_set_variable_pt set_handler;
38     ngx_http_get_variable_pt get_handler;
39     u_intptr_t data;
40     ngx_uint_t flags;
41     ngx_uint_t index;
42 };
43
44
45 ngx_http_variable_t *ngx_http_add_variable(ngx_conf_t *cf, ngx_str_t *name,
46     ngx_uint_t flags);
47 ngx_int_t ngx_http_get_variable_index(ngx_conf_t *cf, ngx_str_t *name);
48 ngx_http_variable_value_t *ngx_http_get_indexed_variable(ngx_http_request_t *r,
49     ngx_uint_t index);
50 ngx_http_variable_value_t *ngx_http_get_flushed_variable(ngx_http_request_t *r,
51     ngx_uint_t index);
52
53 ngx_http_variable_value_t *ngx_http_get_variable(ngx_http_request_t *r,
54     ngx_str_t *name, ngx_uint_t key);
55
56 ngx_int_t ngx_http_variable_unknown_header(ngx_http_variable_value_t *v,
57     ngx_str_t *var, ngx_list_part_t *part, size_t prefix);
58
59 #if (NGX_PCRE)
60
61 typedef struct {
62     ngx_uint_t capture;
63     ngx_int_t index;
64 } ngx_http_regex_variable_t;
65
66
67 typedef struct {
68     ngx_regex_t *regex;
69     ngx_uint_t ncaptures;
70     ngx_http_regex_variable_t *variables;
71     ngx_uint_t nvariables;
72     ngx_str_t name;
73 } ngx_http_regex_t;
74
75 typedef struct {
76     ngx_http_regex_t *regex;
77     void *value;
78 } ngx_http_map_regex_t;
79
80
81
82 ngx_http_regex_t *ngx_http_regex_compile(ngx_conf_t *cf,
83     ngx_regex_compile_t *rc);
84 ngx_int_t ngx_http_regex_exec(ngx_http_request_t *r, ngx_http_regex_t *re,
85     ngx_str_t *s);
86
87 #endif
88
89
90 typedef struct {
91     ngx_hash_combined_t hash;
92 #if (NGX_PCRE)
93     ngx_http_map_regex_t *regex;
94     ngx_uint_t nregex;
95 #endif
96 } ngx_http_map_t;
97
98
99 void *ngx_http_map_find(ngx_http_request_t *r, ngx_http_map_t *map,
100     ngx_str_t *match);
101
102
103

```

```
104 ngx\_int\_t ngx\_http\_variables\_add\_core\_vars\(ngx\_conf\_t \*cf\);
105 ngx\_int\_t ngx\_http\_variables\_init\_vars\(ngx\_conf\_t \*cf\);
106
107
108 extern ngx\_http\_variable\_value\_t ngx\_http\_variable\_null\_value;
109 extern ngx\_http\_variable\_value\_t ngx\_http\_variable\_true\_value;
110
111
112 #endif /* \_NGX\_HTTP\_VARIABLES\_H\_INCLUDED */
```

[One Level Up](#)

[Top Level](#)

## src/http/nginx\_http\_variables.c - nginx-1.7.10

### Global variables defined

- [ngx\\_http\\_core\\_variables](#)
- [ngx\\_http\\_variable\\_null\\_value](#)
- [ngx\\_http\\_variable\\_true\\_value](#)

### Functions defined

- [ngx\\_http\\_add\\_variable](#)
- [ngx\\_http\\_get\\_flushed\\_variable](#)
- [ngx\\_http\\_get\\_indexed\\_variable](#)
- [ngx\\_http\\_get\\_variable](#)
- [ngx\\_http\\_get\\_variable\\_index](#)
- [ngx\\_http\\_map\\_find](#)
- [ngx\\_http\\_regex\\_compile](#)
- [ngx\\_http\\_regex\\_exec](#)
- [ngx\\_http\\_variable\\_argument](#)
- [ngx\\_http\\_variable\\_binary\\_remote\\_addr](#)
- [ngx\\_http\\_variable\\_body\\_bytes\\_sent](#)
- [ngx\\_http\\_variable\\_bytes\\_sent](#)
- [ngx\\_http\\_variable\\_connection](#)
- [ngx\\_http\\_variable\\_connection\\_requests](#)
- [ngx\\_http\\_variable\\_content\\_length](#)
- [ngx\\_http\\_variable\\_cookie](#)
- [ngx\\_http\\_variable\\_cookies](#)
- [ngx\\_http\\_variable\\_document\\_root](#)
- [ngx\\_http\\_variable\\_header](#)
- [ngx\\_http\\_variable\\_headers](#)
- [ngx\\_http\\_variable\\_headers\\_internal](#)
- [ngx\\_http\\_variable\\_host](#)
- [ngx\\_http\\_variable\\_hostname](#)
- [ngx\\_http\\_variable\\_https](#)
- [ngx\\_http\\_variable\\_is\\_args](#)

- [ngx http variable msec](#)
- [ngx http variable nginx version](#)
- [ngx http variable not found](#)
- [ngx http variable pid](#)
- [ngx http variable pipe](#)
- [ngx http variable proxy protocol addr](#)
- [ngx http variable realpath root](#)
- [ngx http variable remote addr](#)
- [ngx http variable remote port](#)
- [ngx http variable remote user](#)
- [ngx http variable request](#)
- [ngx http variable request body](#)
- [ngx http variable request body file](#)
- [ngx http variable request completion](#)
- [ngx http variable request filename](#)
- [ngx http variable request get size](#)
- [ngx http variable request length](#)
- [ngx http variable request line](#)
- [ngx http variable request method](#)
- [ngx http variable request set size](#)
- [ngx http variable request time](#)
- [ngx http variable scheme](#)
- [ngx http variable sent connection](#)
- [ngx http variable sent content length](#)
- [ngx http variable sent content type](#)
- [ngx http variable sent keep alive](#)
- [ngx http variable sent last modified](#)
- [ngx http variable sent location](#)
- [ngx http variable sent transfer encoding](#)
- [ngx http variable server addr](#)
- [ngx http variable server name](#)
- [ngx http variable server port](#)
- [ngx http variable set args](#)

- [ngx\\_http\\_variable\\_status](#)
- [ngx\\_http\\_variable\\_tcpinfo](#)
- [ngx\\_http\\_variable\\_time\\_iso8601](#)
- [ngx\\_http\\_variable\\_time\\_local](#)
- [ngx\\_http\\_variable\\_unknown\\_header](#)
- [ngx\\_http\\_variable\\_unknown\\_header\\_in](#)
- [ngx\\_http\\_variable\\_unknown\\_header\\_out](#)
- [ngx\\_http\\_variables\\_add\\_core\\_vars](#)
- [ngx\\_http\\_variables\\_init\\_vars](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11 #include <nginx.h>
12
13
14 static ngx_int_t ngx_http_variable_request(ngx_http_request_t *r,
15     ngx_http_variable_value_t *v, uintptr_t data);
16 #if 0
17 static void ngx_http_variable_request_set(ngx_http_request_t *r,
18     ngx_http_variable_value_t *v, uintptr_t data);
19 #endif
20 static ngx_int_t ngx_http_variable_request_get_size(ngx_http_request_t *r,
21     ngx_http_variable_value_t *v, uintptr_t data);
22 static void ngx_http_variable_request_set_size(ngx_http_request_t *r,
23     ngx_http_variable_value_t *v, uintptr_t data);
24 static ngx_int_t ngx_http_variable_header(ngx_http_request_t *r,
25     ngx_http_variable_value_t *v, uintptr_t data);
26
27 static ngx_int_t ngx_http_variable_cookies(ngx_http_request_t *r,
28     ngx_http_variable_value_t *v, uintptr_t data);
29 static ngx_int_t ngx_http_variable_headers(ngx_http_request_t *r,
30     ngx_http_variable_value_t *v, uintptr_t data);
31 static ngx_int_t ngx_http_variable_headers_internal(ngx_http_request_t *r,
32     ngx_http_variable_value_t *v, uintptr_t data, u_char sep);
33
34 static ngx_int_t ngx_http_variable_unknown_header_in(ngx_http_request_t *r,
35     ngx_http_variable_value_t *v, uintptr_t data);
36 static ngx_int_t ngx_http_variable_unknown_header_out(ngx_http_request_t *r,
37     ngx_http_variable_value_t *v, uintptr_t data);
38 static ngx_int_t ngx_http_variable_request_line(ngx_http_request_t *r,
39     ngx_http_variable_value_t *v, uintptr_t data);
40 static ngx_int_t ngx_http_variable_cookie(ngx_http_request_t *r,
41     ngx_http_variable_value_t *v, uintptr_t data);
42 static ngx_int_t ngx_http_variable_argument(ngx_http_request_t *r,
43     ngx_http_variable_value_t *v, uintptr_t data);
44 #if (NGX_HAVE_TCP_INFO)
45 static ngx_int_t ngx_http_variable_tcpinfo(ngx_http_request_t *r,
46     ngx_http_variable_value_t *v, uintptr_t data);
47 #endif
48
49 static ngx_int_t ngx_http_variable_content_length(ngx_http_request_t *r,
50     ngx_http_variable_value_t *v, uintptr_t data);
51 static ngx_int_t ngx_http_variable_host(ngx_http_request_t *r,
52     ngx_http_variable_value_t *v, uintptr_t data);

```

```
53 static ngx_int_t ngx_http_variable_binary_remote_addr(ngx_http_request_t *r,
54     ngx_http_variable_value_t *v, uintptr_t data);
55 static ngx_int_t ngx_http_variable_remote_addr(ngx_http_request_t *r,
56     ngx_http_variable_value_t *v, uintptr_t data);
57 static ngx_int_t ngx_http_variable_remote_port(ngx_http_request_t *r,
58     ngx_http_variable_value_t *v, uintptr_t data);
59 static ngx_int_t ngx_http_variable_proxy_protocol_addr(ngx_http_request_t *r,
60     ngx_http_variable_value_t *v, uintptr_t data);
61 static ngx_int_t ngx_http_variable_server_addr(ngx_http_request_t *r,
62     ngx_http_variable_value_t *v, uintptr_t data);
63 static ngx_int_t ngx_http_variable_server_port(ngx_http_request_t *r,
64     ngx_http_variable_value_t *v, uintptr_t data);
65 static ngx_int_t ngx_http_variable_scheme(ngx_http_request_t *r,
66     ngx_http_variable_value_t *v, uintptr_t data);
67 static ngx_int_t ngx_http_variable_https(ngx_http_request_t *r,
68     ngx_http_variable_value_t *v, uintptr_t data);
69 static void ngx_http_variable_set_args(ngx_http_request_t *r,
70     ngx_http_variable_value_t *v, uintptr_t data);
71 static ngx_int_t ngx_http_variable_is_args(ngx_http_request_t *r,
72     ngx_http_variable_value_t *v, uintptr_t data);
73 static ngx_int_t ngx_http_variable_document_root(ngx_http_request_t *r,
74     ngx_http_variable_value_t *v, uintptr_t data);
75 static ngx_int_t ngx_http_variable_realpath_root(ngx_http_request_t *r,
76     ngx_http_variable_value_t *v, uintptr_t data);
77 static ngx_int_t ngx_http_variable_request_filename(ngx_http_request_t *r,
78     ngx_http_variable_value_t *v, uintptr_t data);
79 static ngx_int_t ngx_http_variable_server_name(ngx_http_request_t *r,
80     ngx_http_variable_value_t *v, uintptr_t data);
81 static ngx_int_t ngx_http_variable_request_method(ngx_http_request_t *r,
82     ngx_http_variable_value_t *v, uintptr_t data);
83 static ngx_int_t ngx_http_variable_remote_user(ngx_http_request_t *r,
84     ngx_http_variable_value_t *v, uintptr_t data);
85 static ngx_int_t ngx_http_variable_bytes_sent(ngx_http_request_t *r,
86     ngx_http_variable_value_t *v, uintptr_t data);
87 static ngx_int_t ngx_http_variable_body_bytes_sent(ngx_http_request_t *r,
88     ngx_http_variable_value_t *v, uintptr_t data);
89 static ngx_int_t ngx_http_variable_pipe(ngx_http_request_t *r,
90     ngx_http_variable_value_t *v, uintptr_t data);
91 static ngx_int_t ngx_http_variable_request_completion(ngx_http_request_t *r,
92     ngx_http_variable_value_t *v, uintptr_t data);
93 static ngx_int_t ngx_http_variable_request_body(ngx_http_request_t *r,
94     ngx_http_variable_value_t *v, uintptr_t data);
95 static ngx_int_t ngx_http_variable_request_body_file(ngx_http_request_t *r,
96     ngx_http_variable_value_t *v, uintptr_t data);
97 static ngx_int_t ngx_http_variable_request_length(ngx_http_request_t *r,
98     ngx_http_variable_value_t *v, uintptr_t data);
99 static ngx_int_t ngx_http_variable_request_time(ngx_http_request_t *r,
100     ngx_http_variable_value_t *v, uintptr_t data);
101 static ngx_int_t ngx_http_variable_status(ngx_http_request_t *r,
102     ngx_http_variable_value_t *v, uintptr_t data);
103
104 static ngx_int_t ngx_http_variable_sent_content_type(ngx_http_request_t *r,
105     ngx_http_variable_value_t *v, uintptr_t data);
106 static ngx_int_t ngx_http_variable_sent_content_length(ngx_http_request_t *r,
107     ngx_http_variable_value_t *v, uintptr_t data);
108 static ngx_int_t ngx_http_variable_sent_location(ngx_http_request_t *r,
109     ngx_http_variable_value_t *v, uintptr_t data);
110 static ngx_int_t ngx_http_variable_sent_last_modified(ngx_http_request_t *r,
111     ngx_http_variable_value_t *v, uintptr_t data);
112 static ngx_int_t ngx_http_variable_sent_connection(ngx_http_request_t *r,
113     ngx_http_variable_value_t *v, uintptr_t data);
114 static ngx_int_t ngx_http_variable_sent_keep_alive(ngx_http_request_t *r,
115     ngx_http_variable_value_t *v, uintptr_t data);
116 static ngx_int_t ngx_http_variable_sent_transfer_encoding(ngx_http_request_t *r,
117     ngx_http_variable_value_t *v, uintptr_t data);
118
119 static ngx_int_t ngx_http_variable_connection(ngx_http_request_t *r,
120     ngx_http_variable_value_t *v, uintptr_t data);
121 static ngx_int_t ngx_http_variable_connection_requests(ngx_http_request_t *r,
122     ngx_http_variable_value_t *v, uintptr_t data);
123
124 static ngx_int_t ngx_http_variable_nginx_version(ngx_http_request_t *r,
125     ngx_http_variable_value_t *v, uintptr_t data);
126 static ngx_int_t ngx_http_variable_hostname(ngx_http_request_t *r,
127     ngx_http_variable_value_t *v, uintptr_t data);
128 static ngx_int_t ngx_http_variable_pid(ngx_http_request_t *r,
```

```

129     ngx_http_variable_value_t *v, uintptr_t data);
130 static ngx_int_t ngx_http_variable_msec(ngx_http_request_t *r,
131     ngx_http_variable_value_t *v, uintptr_t data);
132 static ngx_int_t ngx_http_variable_time_iso8601(ngx_http_request_t *r,
133     ngx_http_variable_value_t *v, uintptr_t data);
134 static ngx_int_t ngx_http_variable_time_local(ngx_http_request_t *r,
135     ngx_http_variable_value_t *v, uintptr_t data);
136
137 /*
138 * TODO:
139 *     Apache CGI: AUTH_TYPE, PATH_INFO (null), PATH_TRANSLATED
140 *                REMOTE_HOST (null), REMOTE_IDENT (null),
141 *                SERVER_SOFTWARE
142 *
143 *     Apache SSI: DOCUMENT_NAME, LAST_MODIFIED, USER_NAME (file owner)
144 */
145
146 /*
147 * the $http_host, $http_user_agent, $http_referer, and $http_via
148 * variables may be handled by generic
149 * ngx_http_variable_unknown_header_in(), but for performance reasons
150 * they are handled using dedicated entries
151 */
152
153 static ngx_http_variable_t ngx_http_core_variables[] = {
154     { ngx_string("http_host"), NULL, ngx_http_variable_header,
155     offsetof(ngx_http_request_t, headers_in.host), 0, 0 },
156
157     { ngx_string("http_user_agent"), NULL, ngx_http_variable_header,
158     offsetof(ngx_http_request_t, headers_in.user_agent), 0, 0 },
159
160     { ngx_string("http_referer"), NULL, ngx_http_variable_header,
161     offsetof(ngx_http_request_t, headers_in.referer), 0, 0 },
162
163     #if (NGX_HTTP_GZIP)
164     { ngx_string("http_via"), NULL, ngx_http_variable_header,
165     offsetof(ngx_http_request_t, headers_in.via), 0, 0 },
166     #endif
167
168     #if (NGX_HTTP_X_FORWARDED_FOR)
169     { ngx_string("http_x_forwarded_for"), NULL, ngx_http_variable_headers,
170     offsetof(ngx_http_request_t, headers_in.x_forwarded_for), 0, 0 },
171     #endif
172
173     { ngx_string("http_cookie"), NULL, ngx_http_variable_cookies,
174     offsetof(ngx_http_request_t, headers_in.cookies), 0, 0 },
175
176     { ngx_string("content_length"), NULL, ngx_http_variable_content_length,
177     0, 0, 0 },
178
179     { ngx_string("content_type"), NULL, ngx_http_variable_header,
180     offsetof(ngx_http_request_t, headers_in.content_type), 0, 0 },
181
182     { ngx_string("host"), NULL, ngx_http_variable_host, 0, 0, 0 },
183
184     { ngx_string("binary_remote_addr"), NULL,
185     ngx_http_variable_binary_remote_addr, 0, 0, 0 },
186
187     { ngx_string("remote_addr"), NULL, ngx_http_variable_remote_addr, 0, 0, 0 },
188
189     { ngx_string("remote_port"), NULL, ngx_http_variable_remote_port, 0, 0, 0 },
190
191     { ngx_string("proxy_protocol_addr"), NULL,
192     ngx_http_variable_proxy_protocol_addr, 0, 0, 0 },
193
194     { ngx_string("server_addr"), NULL, ngx_http_variable_server_addr, 0, 0, 0 },
195
196     { ngx_string("server_port"), NULL, ngx_http_variable_server_port, 0, 0, 0 },
197
198     { ngx_string("server_protocol"), NULL, ngx_http_variable_request,
199     offsetof(ngx_http_request_t, http_protocol), 0, 0 },
200
201     { ngx_string("scheme"), NULL, ngx_http_variable_scheme, 0, 0, 0 },
202
203     { ngx_string("https"), NULL, ngx_http_variable_https, 0, 0, 0 },

```

```

205 { ngx_string("request_uri"), NULL, ngx_http_variable_request,
206     offsetof(ngx_http_request_t, unparsed_uri), 0, 0 },
207
208
209 { ngx_string("uri"), NULL, ngx_http_variable_request,
210     offsetof(ngx_http_request_t, uri),
211     NGX_HTTP_VAR_NOCACHEABLE, 0 },
212
213 { ngx_string("document_uri"), NULL, ngx_http_variable_request,
214     offsetof(ngx_http_request_t, uri),
215     NGX_HTTP_VAR_NOCACHEABLE, 0 },
216
217 { ngx_string("request"), NULL, ngx_http_variable_request_line, 0, 0, 0 },
218
219 { ngx_string("document_root"), NULL,
220     ngx_http_variable_document_root, 0, NGX_HTTP_VAR_NOCACHEABLE, 0 },
221
222 { ngx_string("realpath_root"), NULL,
223     ngx_http_variable_realpath_root, 0, NGX_HTTP_VAR_NOCACHEABLE, 0 },
224
225 { ngx_string("query_string"), NULL, ngx_http_variable_request,
226     offsetof(ngx_http_request_t, args),
227     NGX_HTTP_VAR_NOCACHEABLE, 0 },
228
229 { ngx_string("args"),
230     ngx_http_variable_set_args,
231     ngx_http_variable_request,
232     offsetof(ngx_http_request_t, args),
233     NGX_HTTP_VAR_CHANGEABLE|NGX_HTTP_VAR_NOCACHEABLE, 0 },
234
235 { ngx_string("is_args"), NULL, ngx_http_variable_is_args,
236     0, NGX_HTTP_VAR_NOCACHEABLE, 0 },
237
238 { ngx_string("request_filename"), NULL,
239     ngx_http_variable_request_filename, 0,
240     NGX_HTTP_VAR_NOCACHEABLE, 0 },
241
242 { ngx_string("server_name"), NULL, ngx_http_variable_server_name, 0, 0, 0 },
243
244 { ngx_string("request_method"), NULL,
245     ngx_http_variable_request_method, 0,
246     NGX_HTTP_VAR_NOCACHEABLE, 0 },
247
248 { ngx_string("remote_user"), NULL, ngx_http_variable_remote_user, 0, 0, 0 },
249
250 { ngx_string("bytes_sent"), NULL, ngx_http_variable_bytes_sent,
251     0, 0, 0 },
252
253 { ngx_string("body_bytes_sent"), NULL, ngx_http_variable_body_bytes_sent,
254     0, 0, 0 },
255
256 { ngx_string("pipe"), NULL, ngx_http_variable_pipe,
257     0, 0, 0 },
258
259 { ngx_string("request_completion"), NULL,
260     ngx_http_variable_request_completion,
261     0, 0, 0 },
262
263 { ngx_string("request_body"), NULL,
264     ngx_http_variable_request_body,
265     0, 0, 0 },
266
267 { ngx_string("request_body_file"), NULL,
268     ngx_http_variable_request_body_file,
269     0, 0, 0 },
270
271 { ngx_string("request_length"), NULL, ngx_http_variable_request_length,
272     0, NGX_HTTP_VAR_NOCACHEABLE, 0 },
273
274 { ngx_string("request_time"), NULL, ngx_http_variable_request_time,
275     0, NGX_HTTP_VAR_NOCACHEABLE, 0 },
276
277 { ngx_string("status"), NULL,
278     ngx_http_variable_status, 0,
279     NGX_HTTP_VAR_NOCACHEABLE, 0 },
280

```



```

281 { ngx_string("sent_http_content_type"), NULL,
282     ngx_http_variable_sent_content_type, 0, 0, 0 },
283
284 { ngx_string("sent_http_content_length"), NULL,
285     ngx_http_variable_sent_content_length, 0, 0, 0 },
286
287 { ngx_string("sent_http_location"), NULL,
288     ngx_http_variable_sent_location, 0, 0, 0 },
289
290 { ngx_string("sent_http_last_modified"), NULL,
291     ngx_http_variable_sent_last_modified, 0, 0, 0 },
292
293 { ngx_string("sent_http_connection"), NULL,
294     ngx_http_variable_sent_connection, 0, 0, 0 },
295
296 { ngx_string("sent_http_keep_alive"), NULL,
297     ngx_http_variable_sent_keep_alive, 0, 0, 0 },
298
299 { ngx_string("sent_http_transfer_encoding"), NULL,
300     ngx_http_variable_sent_transfer_encoding, 0, 0, 0 },
301
302 { ngx_string("sent_http_cache_control"), NULL, ngx_http_variable_headers,
303     offsetof(ngx_http_request_t, headers_out.cache_control), 0, 0 },
304
305 { ngx_string("limit_rate"), ngx_http_variable_request_set_size,
306     ngx_http_variable_request_get_size,
307     offsetof(ngx_http_request_t, limit_rate),
308     NGX_HTTP_VAR_CHANGEABLE|NGX_HTTP_VAR_NOCACHEABLE, 0 },
309
310 { ngx_string("connection"), NULL,
311     ngx_http_variable_connection, 0, 0, 0 },
312
313 { ngx_string("connection_requests"), NULL,
314     ngx_http_variable_connection_requests, 0, 0, 0 },
315
316 { ngx_string("nginx_version"), NULL, ngx_http_variable_nginx_version,
317     0, 0, 0 },
318
319 { ngx_string("hostname"), NULL, ngx_http_variable_hostname,
320     0, 0, 0 },
321
322 { ngx_string("pid"), NULL, ngx_http_variable_pid,
323     0, 0, 0 },
324
325 { ngx_string("msec"), NULL, ngx_http_variable_msec,
326     0, NGX_HTTP_VAR_NOCACHEABLE, 0 },
327
328 { ngx_string("time_iso8601"), NULL, ngx_http_variable_time_iso8601,
329     0, NGX_HTTP_VAR_NOCACHEABLE, 0 },
330
331 { ngx_string("time_local"), NULL, ngx_http_variable_time_local,
332     0, NGX_HTTP_VAR_NOCACHEABLE, 0 },
333
334 #if (NGX_HAVE_TCP_INFO)
335 { ngx_string("tcpinfo_rtt"), NULL, ngx_http_variable_tcpinfo,
336     0, NGX_HTTP_VAR_NOCACHEABLE, 0 },
337
338 { ngx_string("tcpinfo_rttvar"), NULL, ngx_http_variable_tcpinfo,
339     1, NGX_HTTP_VAR_NOCACHEABLE, 0 },
340
341 { ngx_string("tcpinfo_snd_cwnd"), NULL, ngx_http_variable_tcpinfo,
342     2, NGX_HTTP_VAR_NOCACHEABLE, 0 },
343
344 { ngx_string("tcpinfo_rcv_space"), NULL, ngx_http_variable_tcpinfo,
345     3, NGX_HTTP_VAR_NOCACHEABLE, 0 },
346 #endif
347
348 { ngx_null_string, NULL, NULL, 0, 0, 0 }
349 };
350
351 ngx_http_variable_value_t ngx_http_variable_null_value =
352     ngx_http_variable("");
353 ngx_http_variable_value_t ngx_http_variable_true_value =
354     ngx_http_variable("1");
355
356

```

```

357 ngx_http_variable_t *
358 ngx_http_add_variable(ngx_conf_t *cf, ngx_str_t *name, ngx_uint_t flags)
359 {
360     ngx_int_t          rc;
361     ngx_uint_t        i;
362     ngx_hash_key_t    *key;
363     ngx_http_variable_t *v;
364     ngx_http_core_main_conf_t *cmcf;
365
366     if (name->len == 0) {
367         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
368             "invalid variable name \"%s\"",
369             name->data);
370         return NULL;
371     }
372
373     cmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_core_module);
374
375     key = cmcf->variables_keys->keys.elts;
376     for (i = 0; i < cmcf->variables_keys->keys.nelts; i++) {
377         if (name->len != key[i].key.len
378             || ngx_strncasecmp(name->data, key[i].key.data, name->len) != 0)
379             {
380                 continue;
381             }
382
383             v = key[i].value;
384
385             if (!(v->flags & NGX_HTTP_VAR_CHANGEABLE)) {
386                 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
387                     "the duplicate \"%V\" variable", name);
388                 return NULL;
389             }
390
391             return v;
392     }
393
394     v = ngx_palloc(cf->pool, sizeof(ngx_http_variable_t));
395     if (v == NULL) {
396         return NULL;
397     }
398
399     v->name.len = name->len;
400     v->name.data = ngx_pnalloc(cf->pool, name->len);
401     if (v->name.data == NULL) {
402         return NULL;
403     }
404
405     ngx_strlow(v->name.data, name->data, name->len);
406
407     v->set_handler = NULL;
408     v->get_handler = NULL;
409     v->data = 0;
410     v->flags = flags;
411     v->index = 0;
412
413     rc = ngx_hash_add_key(cmcf->variables_keys, &v->name, v, 0);
414
415     if (rc == NGX_ERROR) {
416         return NULL;
417     }
418
419     if (rc == NGX_BUSY) {
420         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
421             "conflicting variable name \"%V\"", name);
422         return NULL;
423     }
424
425     return v;
426 }
427
428
429 ngx_int_t
430 ngx_http_get_variable_index(ngx_conf_t *cf, ngx_str_t *name)
431 {
432     ngx_uint_t          i;

```

```

433 ngx\_http\_variable\_t *v;
434 ngx\_http\_core\_main\_conf\_t *cmcf;
435
436 if (name->len == 0) {
437     ngx\_conf\_log\_error(NGX_LOG_EMERG, cf, 0,
438         "invalid variable name \"$\"");
439     return NGX_ERROR;
440 }
441
442 cmcf = ngx\_http\_conf\_get\_module\_main\_conf(cf, ngx\_http\_core\_module);
443
444 v = cmcf->variables.elts;
445
446 if (v == NULL) {
447     if (ngx\_array\_init(&cmcf->variables, cf->pool, 4,
448         sizeof(ngx\_http\_variable\_t))
449         != NGX_OK)
450     {
451         return NGX_ERROR;
452     }
453 } else {
454     for (i = 0; i < cmcf->variables.nelts; i++) {
455         if (name->len != v[i].name.len
456             || ngx\_strncasecmp(name->data, v[i].name.data, name->len) != 0)
457         {
458             continue;
459         }
460     }
461
462     return i;
463 }
464 }
465
466 v = ngx\_array\_push(&cmcf->variables);
467 if (v == NULL) {
468     return NGX_ERROR;
469 }
470
471 v->name.len = name->len;
472 v->name.data = ngx\_pnalloc(cf->pool, name->len);
473 if (v->name.data == NULL) {
474     return NGX_ERROR;
475 }
476
477 ngx\_strlow(v->name.data, name->data, name->len);
478
479 v->set_handler = NULL;
480 v->get_handler = NULL;
481 v->data = 0;
482 v->flags = 0;
483 v->index = cmcf->variables.nelts - 1;
484
485 return v->index;
486 }
487
488
489 ngx\_http\_variable\_value\_t *
490 ngx\_http\_get\_indexed\_variable(ngx\_http\_request\_t *r, ngx\_uint\_t index)
491 {
492     ngx\_http\_variable\_t *v;
493     ngx\_http\_core\_main\_conf\_t *cmcf;
494
495     cmcf = ngx\_http\_get\_module\_main\_conf(r, ngx\_http\_core\_module);
496
497     if (cmcf->variables.nelts <= index) {
498         ngx\_log\_error(NGX_LOG_ALERT, r->connection->log, 0,
499             "unknown variable index: %ui", index);
500         return NULL;
501     }
502
503     if (r->variables[index].not_found || r->variables[index].valid) {
504         return &r->variables[index];
505     }
506
507     v = cmcf->variables.elts;
508

```

```

509     if (v[index].get_handler(r, &r->variables[index], v[index].data)
510         == NGX\_OK)
511     {
512         if (v[index].flags & NGX\_HTTP\_VAR\_NOCACHEABLE) {
513             r->variables[index].no_cacheable = 1;
514         }
515
516         return &r->variables[index];
517     }
518
519     r->variables[index].valid = 0;
520     r->variables[index].not_found = 1;
521
522     return NULL;
523 }
524
525
526 ngx\_http\_variable\_value\_t *
527 ngx\_http\_get\_flushed\_variable(ngx\_http\_request\_t *r, ngx\_uint\_t index)
528 {
529     ngx\_http\_variable\_value\_t *v;
530
531     v = &r->variables[index];
532
533     if (v->valid || v->not_found) {
534         if (!v->no_cacheable) {
535             return v;
536         }
537
538         v->valid = 0;
539         v->not_found = 0;
540     }
541
542     return ngx\_http\_get\_indexed\_variable(r, index);
543 }
544
545
546 ngx\_http\_variable\_value\_t *
547 ngx\_http\_get\_variable(ngx\_http\_request\_t *r, ngx\_str\_t *name, ngx\_uint\_t key)
548 {
549     ngx\_http\_variable\_t *v;
550     ngx\_http\_variable\_value\_t *vv;
551     ngx\_http\_core\_main\_conf\_t *cmcf;
552
553     cmcf = ngx\_http\_get\_module\_main\_conf(r, ngx\_http\_core\_module);
554
555     v = ngx\_hash\_find(&cmcf->variables_hash, key, name->data, name->len);
556
557     if (v) {
558         if (v->flags & NGX\_HTTP\_VAR\_INDEXED) {
559             return ngx\_http\_get\_flushed\_variable(r, v->index);
560         }
561     } else {
562
563         vv = ngx\_palloc(r->pool, sizeof(ngx\_http\_variable\_value\_t));
564
565         if (vv && v->get_handler(r, vv, v->data) == NGX\_OK) {
566             return vv;
567         }
568
569         return NULL;
570     }
571 }
572
573 vv = ngx\_palloc(r->pool, sizeof(ngx\_http\_variable\_value\_t));
574 if (vv == NULL) {
575     return NULL;
576 }
577
578 if (ngx\_strncmp(name->data, "http_", 5) == 0) {
579
580     if (ngx\_http\_variable\_unknown\_header\_in(r, vv, (uintptr\_t) name)
581         == NGX\_OK)
582     {
583         return vv;
584     }

```

```

585     return NULL;
586 }
587
588
589 if (ngx_strncmp(name->data, "sent_http_", 10) == 0) {
590
591     if (ngx_http_variable_unknown_header_out(r, vv, (uintptr_t) name)
592         == NGX_OK)
593     {
594         return vv;
595     }
596
597     return NULL;
598 }
599
600 if (ngx_strncmp(name->data, "upstream_http_", 14) == 0) {
601
602     if (ngx_http_upstream_header_variable(r, vv, (uintptr_t) name)
603         == NGX_OK)
604     {
605         return vv;
606     }
607
608     return NULL;
609 }
610
611 if (ngx_strncmp(name->data, "cookie_", 7) == 0) {
612
613     if (ngx_http_variable_cookie(r, vv, (uintptr_t) name) == NGX_OK) {
614         return vv;
615     }
616
617     return NULL;
618 }
619
620 if (ngx_strncmp(name->data, "upstream_cookie_", 16) == 0) {
621
622     if (ngx_http_upstream_cookie_variable(r, vv, (uintptr_t) name)
623         == NGX_OK)
624     {
625         return vv;
626     }
627
628     return NULL;
629 }
630
631 if (ngx_strncmp(name->data, "arg_", 4) == 0) {
632
633     if (ngx_http_variable_argument(r, vv, (uintptr_t) name) == NGX_OK) {
634         return vv;
635     }
636
637     return NULL;
638 }
639
640 vv->not_found = 1;
641
642 return vv;
643 }
644
645
646 static ngx_int_t
647 ngx_http_variable_request(ngx_http_request_t *r, ngx_http_variable_value_t *v,
648     uintptr_t data)
649 {
650     ngx_str_t *s;
651
652     s = (ngx_str_t *) ((char *) r + data);
653
654     if (s->data) {
655         v->len = s->len;
656         v->valid = 1;
657         v->no_cacheable = 0;
658         v->not_found = 0;
659         v->data = s->data;
660

```

```

661     } else {
662         v->not_found = 1;
663     }
664
665     return NGX_OK;
666 }
667
668
669 #if 0
670
671 static void
672 ngx_http_variable_request_set(ngx_http_request_t *r,
673 ngx_http_variable_value_t *v, uintptr_t data)
674 {
675     ngx_str_t *s;
676
677     s = (ngx_str_t *) ((char *) r + data);
678
679     s->len = v->len;
680     s->data = v->data;
681 }
682
683 #endif
684
685
686 static ngx_int_t
687 ngx_http_variable_request_get_size(ngx_http_request_t *r,
688 ngx_http_variable_value_t *v, uintptr_t data)
689 {
690     size_t *sp;
691
692     sp = (size_t *) ((char *) r + data);
693
694     v->data = ngx_pnalloc(r->pool, NGX_SIZE_T_LEN);
695     if (v->data == NULL) {
696         return NGX_ERROR;
697     }
698
699     v->len = ngx_sprintf(v->data, "%uz", *sp) - v->data;
700     v->valid = 1;
701     v->no_cacheable = 0;
702     v->not_found = 0;
703
704     return NGX_OK;
705 }
706
707
708 static void
709 ngx_http_variable_request_set_size(ngx_http_request_t *r,
710 ngx_http_variable_value_t *v, uintptr_t data)
711 {
712     ssize_t s, *sp;
713     ngx_str_t val;
714
715     val.len = v->len;
716     val.data = v->data;
717
718     s = ngx_parse_size(&val);
719
720     if (s == NGX_ERROR) {
721         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
722             "invalid size \"%V\"", &val);
723         return;
724     }
725
726     sp = (ssize_t *) ((char *) r + data);
727
728     *sp = s;
729
730     return;
731 }
732
733
734 static ngx_int_t
735 ngx_http_variable_header(ngx_http_request_t *r, ngx_http_variable_value_t *v,
736     uintptr_t data)

```

```

737 {
738     ngx_table_elt_t *h;
739
740     h = *(ngx_table_elt_t **) ((char *) r + data);
741
742     if (h) {
743         v->len = h->value.len;
744         v->valid = 1;
745         v->no_cacheable = 0;
746         v->not_found = 0;
747         v->data = h->value.data;
748
749     } else {
750         v->not_found = 1;
751     }
752
753     return NGX_OK;
754 }
755
756 static ngx_int_t
757 ngx_http_variable_cookies(ngx_http_request_t *r,
758     ngx_http_variable_value_t *v, uintptr_t data)
759 {
760     return ngx_http_variable_headers_internal(r, v, data, ';');
761 }
762
763
764 static ngx_int_t
765 ngx_http_variable_headers(ngx_http_request_t *r,
766     ngx_http_variable_value_t *v, uintptr_t data)
767 {
768     return ngx_http_variable_headers_internal(r, v, data, ',');
769 }
770
771
772 static ngx_int_t
773 ngx_http_variable_headers_internal(ngx_http_request_t *r,
774     ngx_http_variable_value_t *v, uintptr_t data, u_char sep)
775 {
776     {
777         size_t          len;
778         u_char          *p, *end;
779         ngx_uint_t      i, n;
780         ngx_array_t     *a;
781         ngx_table_elt_t **h;
782
783         a = (ngx_array_t *) ((char *) r + data);
784
785         n = a->nelts;
786         h = a->elts;
787
788         len = 0;
789
790         for (i = 0; i < n; i++) {
791             if (h[i]->hash == 0) {
792                 continue;
793             }
794
795             len += h[i]->value.len + 2;
796         }
797
798         if (len == 0) {
799             v->not_found = 1;
800             return NGX_OK;
801         }
802     }
803
804     len -= 2;
805
806     v->valid = 1;
807     v->no_cacheable = 0;
808     v->not_found = 0;
809
810     if (n == 1) {
811         v->len = (*h)->value.len;
812         v->data = (*h)->value.data;

```

```

813     return NGX\_OK;
814 }
815
816 p = ngx\_pnalloc(r->pool, len);
817 if (p == NULL) {
818     return NGX\_ERROR;
819 }
820
821 v->len = len;
822 v->data = p;
823
824 end = p + len;
825
826 for (i = 0; /* void */ ; i++) {
827     if (h[i]->hash == 0) {
828         continue;
829     }
830
831     p = ngx\_copy(p, h[i]->value.data, h[i]->value.len);
832
833     if (p == end) {
834         break;
835     }
836
837     *p++ = sep; *p++ = ' ';
838 }
839
840 return NGX\_OK;
841 }
842
843
844
845
846 static ngx\_int\_t
847 ngx\_http\_variable\_unknown\_header\_in(ngx\_http\_request\_t *r,
848 ngx\_http\_variable\_value\_t *v, uintptr\_t data)
849 {
850     return ngx\_http\_variable\_unknown\_header(v, (ngx\_str\_t *) data,
851                                             &r->headers_in.headers.part,
852                                             sizeof("http") - 1);
853 }
854
855
856 static ngx\_int\_t
857 ngx\_http\_variable\_unknown\_header\_out(ngx\_http\_request\_t *r,
858 ngx\_http\_variable\_value\_t *v, uintptr\_t data)
859 {
860     return ngx\_http\_variable\_unknown\_header(v, (ngx\_str\_t *) data,
861                                             &r->headers_out.headers.part,
862                                             sizeof("sent_http") - 1);
863 }
864
865
866 ngx\_int\_t
867 ngx\_http\_variable\_unknown\_header(ngx\_http\_variable\_value\_t *v, ngx\_str\_t *var,
868 ngx\_list\_part\_t *part, size\_t prefix)
869 {
870     u\_char          ch;
871     ngx\_uint\_t     i, n;
872     ngx\_table\_elt\_t *header;
873
874     header = part->elts;
875
876     for (i = 0; /* void */ ; i++) {
877         if (i >= part->nelts) {
878             if (part->next == NULL) {
879                 break;
880             }
881
882             part = part->next;
883             header = part->elts;
884             i = 0;
885         }
886
887         if (header[i].hash == 0) {

```



```

889     continue;
890 }
891
892 for (n = 0; n + prefix < var->len && n < header[i].key.len; n++) {
893     ch = header[i].key.data[n];
894
895     if (ch >= 'A' && ch <= 'Z') {
896         ch |= 0x20;
897
898     } else if (ch == '-') {
899         ch = '_';
900     }
901
902     if (var->data[n + prefix] != ch) {
903         break;
904     }
905 }
906
907 if (n + prefix == var->len && n == header[i].key.len) {
908     v->len = header[i].value.len;
909     v->valid = 1;
910     v->no_cacheable = 0;
911     v->not_found = 0;
912     v->data = header[i].value.data;
913
914     return NGX_OK;
915 }
916 }
917
918 v->not_found = 1;
919
920 return NGX_OK;
921 }
922
923
924 static ngx_int_t
925 ngx_http_variable_request_line(ngx_http_request_t *r,
926     ngx_http_variable_value_t *v, uintptr_t data)
927 {
928     u_char *p, *s;
929
930     s = r->request_line.data;
931
932     if (s == NULL) {
933         s = r->request_start;
934
935         if (s == NULL) {
936             v->not_found = 1;
937             return NGX_OK;
938         }
939
940         for (p = s; p < r->header_in->last; p++) {
941             if (*p == CR || *p == LF) {
942                 break;
943             }
944         }
945
946         r->request_line.len = p - s;
947         r->request_line.data = s;
948     }
949
950     v->len = r->request_line.len;
951     v->valid = 1;
952     v->no_cacheable = 0;
953     v->not_found = 0;
954     v->data = s;
955
956     return NGX_OK;
957 }
958
959
960 static ngx_int_t
961 ngx_http_variable_cookie(ngx_http_request_t *r, ngx_http_variable_value_t *v,
962     uintptr_t data)
963 {
964     ngx_str_t *name = (ngx_str_t *) data;

```

```

965     ngx_str_t  cookie, s;
966
967
968     s.len = name->len - (sizeof("cookie_") - 1);
969     s.data = name->data + sizeof("cookie_") - 1;
970
971     if (ngx_http_parse_multi_header_lines(&r->headers_in.cookies, &s, &cookie)
972         == NGX_DECLINED)
973     {
974         v->not_found = 1;
975         return NGX_OK;
976     }
977
978     v->len = cookie.len;
979     v->valid = 1;
980     v->no_cacheable = 0;
981     v->not_found = 0;
982     v->data = cookie.data;
983
984     return NGX_OK;
985 }
986
987
988 static ngx_int_t
989 ngx_http_variable_argument(ngx_http_request_t *r, ngx_http_variable_value_t *v,
990     uintptr_t data)
991 {
992     ngx_str_t *name = (ngx_str_t *) data;
993
994     u_char    *arg;
995     size_t    len;
996     ngx_str_t  value;
997
998     len = name->len - (sizeof("arg_") - 1);
999     arg = name->data + sizeof("arg_") - 1;
1000
1001     if (ngx_http_arg(r, arg, len, &value) != NGX_OK) {
1002         v->not_found = 1;
1003         return NGX_OK;
1004     }
1005
1006     v->data = value.data;
1007     v->len = value.len;
1008     v->valid = 1;
1009     v->no_cacheable = 0;
1010     v->not_found = 0;
1011
1012     return NGX_OK;
1013 }
1014
1015
1016 #if (NGX_HAVE_TCP_INFO)
1017
1018 static ngx_int_t
1019 ngx_http_variable_tcpinfo(ngx_http_request_t *r, ngx_http_variable_value_t *v,
1020     uintptr_t data)
1021 {
1022     struct tcp_info  ti;
1023     socklen_t       len;
1024     uint32_t        value;
1025
1026     len = sizeof(struct tcp_info);
1027     if (getsockopt(r->connection->fd, IPPROTO_TCP, TCP_INFO, &ti, &len) == -1) {
1028         v->not_found = 1;
1029         return NGX_OK;
1030     }
1031
1032     v->data = ngx_pnalloc(r->pool, NGX_INT32_LEN);
1033     if (v->data == NULL) {
1034         return NGX_ERROR;
1035     }
1036
1037     switch (data) {
1038     case 0:
1039         value = ti.tcpi_rtt;
1040         break;

```

```

1041
1042     case 1:
1043         value = ti.tcpi_rttvar;
1044         break;
1045
1046     case 2:
1047         value = ti.tcpi_snd_cwnd;
1048         break;
1049
1050     case 3:
1051         value = ti.tcpi_rcv_space;
1052         break;
1053
1054     /* suppress warning */
1055     default:
1056         value = 0;
1057         break;
1058 }
1059
1060 v->len = ngx_sprintf(v->data, "%uD", value) - v->data;
1061 v->valid = 1;
1062 v->no_cacheable = 0;
1063 v->not_found = 0;
1064
1065     return NGX_OK;
1066 }
1067
1068 #endif
1069
1070
1071 static ngx_int_t
1072 ngx_http_variable_content_length(ngx_http_request_t *r,
1073     ngx_http_variable_value_t *v, uintptr_t data)
1074 {
1075     u_char *p;
1076
1077     if (r->headers_in.content_length) {
1078         v->len = r->headers_in.content_length->value.len;
1079         v->data = r->headers_in.content_length->value.data;
1080         v->valid = 1;
1081         v->no_cacheable = 0;
1082         v->not_found = 0;
1083
1084     } else if (r->headers_in.content_length_n >= 0) {
1085         p = ngx_pnalloc(r->pool, NGX_OFF_T_LEN);
1086         if (p == NULL) {
1087             return NGX_ERROR;
1088         }
1089
1090         v->len = ngx_sprintf(p, "%O", r->headers_in.content_length_n) - p;
1091         v->data = p;
1092         v->valid = 1;
1093         v->no_cacheable = 0;
1094         v->not_found = 0;
1095
1096     } else {
1097         v->not_found = 1;
1098     }
1099
1100     return NGX_OK;
1101 }
1102
1103
1104 static ngx_int_t
1105 ngx_http_variable_host(ngx_http_request_t *r, ngx_http_variable_value_t *v,
1106     uintptr_t data)
1107 {
1108     ngx_http_core_srv_conf_t *cscf;
1109
1110     if (r->headers_in.server.len) {
1111         v->len = r->headers_in.server.len;
1112         v->data = r->headers_in.server.data;
1113
1114     } else {
1115         cscf = ngx_http_get_module_srv_conf(r, ngx_http_core_module);
1116

```

```

1117     v->len = cscf->server_name.len;
1118     v->data = cscf->server_name.data;
1119 }
1120
1121 v->valid = 1;
1122 v->no_cacheable = 0;
1123 v->not_found = 0;
1124
1125 return NGX_OK;
1126 }
1127
1128
1129 static ngx_int_t
1130 ngx_http_variable_binary_remote_addr(ngx_http_request_t *r,
1131 ngx_http_variable_value_t *v, uintptr_t data)
1132 {
1133     struct sockaddr_in  *sin;
1134     #if (NGX_HAVE_INET6)
1135     struct sockaddr_in6 *sin6;
1136     #endif
1137
1138     switch (r->connection->sockaddr->sa_family) {
1139
1140     #if (NGX_HAVE_INET6)
1141     case AF_INET6:
1142         sin6 = (struct sockaddr_in6 *) r->connection->sockaddr;
1143
1144         v->len = sizeof(struct in6_addr);
1145         v->valid = 1;
1146         v->no_cacheable = 0;
1147         v->not_found = 0;
1148         v->data = sin6->sin6_addr.s6_addr;
1149
1150         break;
1151     #endif
1152
1153     default: /* AF_INET */
1154         sin = (struct sockaddr_in *) r->connection->sockaddr;
1155
1156         v->len = sizeof(in_addr_t);
1157         v->valid = 1;
1158         v->no_cacheable = 0;
1159         v->not_found = 0;
1160         v->data = (u_char *) &sin->sin_addr;
1161
1162         break;
1163     }
1164
1165     return NGX_OK;
1166 }
1167
1168
1169 static ngx_int_t
1170 ngx_http_variable_remote_addr(ngx_http_request_t *r,
1171 ngx_http_variable_value_t *v, uintptr_t data)
1172 {
1173     v->len = r->connection->addr_text.len;
1174     v->valid = 1;
1175     v->no_cacheable = 0;
1176     v->not_found = 0;
1177     v->data = r->connection->addr_text.data;
1178
1179     return NGX_OK;
1180 }
1181
1182
1183 static ngx_int_t
1184 ngx_http_variable_remote_port(ngx_http_request_t *r,
1185 ngx_http_variable_value_t *v, uintptr_t data)
1186 {
1187     ngx_uint_t      port;
1188     struct sockaddr_in  *sin;
1189     #if (NGX_HAVE_INET6)
1190     struct sockaddr_in6 *sin6;
1191     #endif
1192

```

```

1193     v->len = 0;
1194     v->valid = 1;
1195     v->no_cacheable = 0;
1196     v->not_found = 0;
1197
1198     v->data = ngx_pnalloc(r->pool, sizeof("65535") - 1);
1199     if (v->data == NULL) {
1200         return NGX_ERROR;
1201     }
1202
1203     switch (r->connection->sockaddr->sa_family) {
1204
1205 #if (NGX_HAVE_INET6)
1206     case AF_INET6:
1207         sin6 = (struct sockaddr_in6 *) r->connection->sockaddr;
1208         port = ntohs(sin6->sin6_port);
1209         break;
1210 #endif
1211
1212 #if (NGX_HAVE_UNIX_DOMAIN)
1213     case AF_UNIX:
1214         port = 0;
1215         break;
1216 #endif
1217
1218     default: /* AF_INET */
1219         sin = (struct sockaddr_in *) r->connection->sockaddr;
1220         port = ntohs(sin->sin_port);
1221         break;
1222     }
1223
1224     if (port > 0 && port < 65536) {
1225         v->len = ngx_sprintf(v->data, "%ui", port) - v->data;
1226     }
1227
1228     return NGX_OK;
1229 }
1230
1231
1232 static ngx_int_t
1233 ngx_http_variable_proxy_protocol_addr(ngx_http_request_t *r,
1234 ngx_http_variable_value_t *v, uintptr_t data)
1235 {
1236     v->len = r->connection->proxy_protocol_addr.len;
1237     v->valid = 1;
1238     v->no_cacheable = 0;
1239     v->not_found = 0;
1240     v->data = r->connection->proxy_protocol_addr.data;
1241
1242     return NGX_OK;
1243 }
1244
1245
1246 static ngx_int_t
1247 ngx_http_variable_server_addr(ngx_http_request_t *r,
1248 ngx_http_variable_value_t *v, uintptr_t data)
1249 {
1250     ngx_str_t s;
1251     u_char addr[NGX_SOCKADDR_STRLEN];
1252
1253     s.len = NGX_SOCKADDR_STRLEN;
1254     s.data = addr;
1255
1256     if (ngx_connection_local_sockaddr(r->connection, &s, 0) != NGX_OK) {
1257         return NGX_ERROR;
1258     }
1259
1260     s.data = ngx_pnalloc(r->pool, s.len);
1261     if (s.data == NULL) {
1262         return NGX_ERROR;
1263     }
1264
1265     ngx_memcpy(s.data, addr, s.len);
1266
1267     v->len = s.len;
1268     v->valid = 1;

```

```

1269     v->no_cacheable = 0;
1270     v->not_found = 0;
1271     v->data = s.data;
1272
1273     return NGX_OK;
1274 }
1275
1276
1277 static ngx_int_t
1278 ngx_http_variable_server_port(ngx_http_request_t *r,
1279     ngx_http_variable_value_t *v, uintptr_t data)
1280 {
1281     ngx_uint_t      port;
1282     struct sockaddr_in *sin;
1283     #if (NGX_HAVE_INET6)
1284     struct sockaddr_in6 *sin6;
1285     #endif
1286
1287     v->len = 0;
1288     v->valid = 1;
1289     v->no_cacheable = 0;
1290     v->not_found = 0;
1291
1292     if (ngx_connection_local_sockaddr(r->connection, NULL, 0) != NGX_OK) {
1293         return NGX_ERROR;
1294     }
1295
1296     v->data = ngx_pnalloc(r->pool, sizeof("65535") - 1);
1297     if (v->data == NULL) {
1298         return NGX_ERROR;
1299     }
1300
1301     switch (r->connection->local_sockaddr->sa_family) {
1302
1303     #if (NGX_HAVE_INET6)
1304     case AF_INET6:
1305         sin6 = (struct sockaddr_in6 *) r->connection->local_sockaddr;
1306         port = ntohs(sin6->sin6_port);
1307         break;
1308     #endif
1309
1310     #if (NGX_HAVE_UNIX_DOMAIN)
1311     case AF_UNIX:
1312         port = 0;
1313         break;
1314     #endif
1315
1316     default: /* AF_INET */
1317         sin = (struct sockaddr_in *) r->connection->local_sockaddr;
1318         port = ntohs(sin->sin_port);
1319         break;
1320     }
1321
1322     if (port > 0 && port < 65536) {
1323         v->len = ngx_sprintf(v->data, "%ui", port) - v->data;
1324     }
1325
1326     return NGX_OK;
1327 }
1328
1329
1330 static ngx_int_t
1331 ngx_http_variable_scheme(ngx_http_request_t *r,
1332     ngx_http_variable_value_t *v, uintptr_t data)
1333 {
1334     #if (NGX_HTTP_SSL)
1335
1336     if (r->connection->ssl) {
1337         v->len = sizeof("https") - 1;
1338         v->valid = 1;
1339         v->no_cacheable = 0;
1340         v->not_found = 0;
1341         v->data = (u_char *) "https";
1342
1343         return NGX_OK;
1344     }

```

```

1345
1346 #endif
1347
1348     v->len = sizeof("http") - 1;
1349     v->valid = 1;
1350     v->no_cacheable = 0;
1351     v->not_found = 0;
1352     v->data = (u_char *) "http";
1353
1354     return NGX_OK;
1355 }
1356
1357
1358 static ngx_int_t
1359 ngx_http_variable_https(ngx_http_request_t *r,
1360     ngx_http_variable_value_t *v, uintptr_t data)
1361 {
1362     #if (NGX_HTTP_SSL)
1363
1364         if (r->connection->ssl) {
1365             v->len = sizeof("on") - 1;
1366             v->valid = 1;
1367             v->no_cacheable = 0;
1368             v->not_found = 0;
1369             v->data = (u_char *) "on";
1370
1371             return NGX_OK;
1372         }
1373
1374     #endif
1375
1376     *v = ngx_http_variable_null_value;
1377
1378     return NGX_OK;
1379 }
1380
1381
1382 static void
1383 ngx_http_variable_set_args(ngx_http_request_t *r,
1384     ngx_http_variable_value_t *v, uintptr_t data)
1385 {
1386     r->args.len = v->len;
1387     r->args.data = v->data;
1388     r->valid_unparsed_uri = 0;
1389 }
1390
1391
1392 static ngx_int_t
1393 ngx_http_variable_is_args(ngx_http_request_t *r,
1394     ngx_http_variable_value_t *v, uintptr_t data)
1395 {
1396     v->valid = 1;
1397     v->no_cacheable = 0;
1398     v->not_found = 0;
1399
1400     if (r->args.len == 0) {
1401         v->len = 0;
1402         v->data = NULL;
1403         return NGX_OK;
1404     }
1405
1406     v->len = 1;
1407     v->data = (u_char *) "?";
1408
1409     return NGX_OK;
1410 }
1411
1412
1413 static ngx_int_t
1414 ngx_http_variable_document_root(ngx_http_request_t *r,
1415     ngx_http_variable_value_t *v, uintptr_t data)
1416 {
1417     ngx_str_t          path;
1418     ngx_http_core_loc_conf_t *clcf;
1419
1420     clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);

```

```

1421
1422     if (clcf->root_lengths == NULL) {
1423         v->len = clcf->root.len;
1424         v->valid = 1;
1425         v->no_cacheable = 0;
1426         v->not_found = 0;
1427         v->data = clcf->root.data;
1428
1429     } else {
1430         if (ngx\_http\_script\_run(r, &path, clcf->root_lengths->elts, 0,
1431             clcf->root_values->elts)
1432             == NULL)
1433         {
1434             return NGX\_ERROR;
1435         }
1436
1437         if (ngx\_get\_full\_name(r->pool, (ngx\_str\_t *) &ngx\_cycle->prefix, &path)
1438             != NGX\_OK)
1439         {
1440             return NGX\_ERROR;
1441         }
1442
1443         v->len = path.len;
1444         v->valid = 1;
1445         v->no_cacheable = 0;
1446         v->not_found = 0;
1447         v->data = path.data;
1448     }
1449
1450     return NGX\_OK;
1451 }
1452
1453
1454 static ngx\_int\_t
1455 ngx\_http\_variable\_realpath\_root(ngx\_http\_request\_t *r,
1456     ngx\_http\_variable\_value\_t *v, uintptr\_t data)
1457 {
1458     u\_char                *real;
1459     size\_t                len;
1460     ngx\_str\_t            path;
1461     ngx\_http\_core\_loc\_conf\_t *clcf;
1462     #if (NGX\_HAVE\_MAX\_PATH)
1463     u\_char                buffer[NGX\_MAX\_PATH];
1464 #endif
1465
1466     clcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_core\_module);
1467
1468     if (clcf->root_lengths == NULL) {
1469         path = clcf->root;
1470
1471     } else {
1472         if (ngx\_http\_script\_run(r, &path, clcf->root_lengths->elts, 1,
1473             clcf->root_values->elts)
1474             == NULL)
1475         {
1476             return NGX\_ERROR;
1477         }
1478
1479         path.data[path.len - 1] = '\\0';
1480
1481         if (ngx\_get\_full\_name(r->pool, (ngx\_str\_t *) &ngx\_cycle->prefix, &path)
1482             != NGX\_OK)
1483         {
1484             return NGX\_ERROR;
1485         }
1486     }
1487
1488     #if (NGX\_HAVE\_MAX\_PATH)
1489     real = buffer;
1490 #else
1491     real = NULL;
1492 #endif
1493
1494     real = ngx\_realpath(path.data, real);
1495
1496     if (real == NULL) {

```



```

1497     ngx_log_error(NGX_LOG_CRIT, r->connection->log, ngx_errno,
1498                 ngx_realpath_n " \"%s\" failed", path.data);
1499     return NGX_ERROR;
1500 }
1501
1502 len = ngx_strlen(real);
1503
1504 v->data = ngx_pnalloc(r->pool, len);
1505 if (v->data == NULL) {
1506 #if !(NGX_HAVE_MAX_PATH)
1507     ngx_free(real);
1508 #endif
1509     return NGX_ERROR;
1510 }
1511
1512 v->len = len;
1513 v->valid = 1;
1514 v->no_cacheable = 0;
1515 v->not_found = 0;
1516
1517 ngx_memcpy(v->data, real, len);
1518
1519 #if !(NGX_HAVE_MAX_PATH)
1520     ngx_free(real);
1521 #endif
1522
1523     return NGX_OK;
1524 }
1525
1526
1527 static ngx_int_t
1528 ngx_http_variable_request_filename(ngx_http_request_t *r,
1529 ngx_http_variable_value_t *v, uintptr_t data)
1530 {
1531     size_t    root;
1532     ngx_str_t path;
1533
1534     if (ngx_http_map_uri_to_path(r, &path, &root, 0) == NULL) {
1535         return NGX_ERROR;
1536     }
1537
1538     /* ngx_http_map_uri_to_path() allocates memory for terminating '\0' */
1539
1540     v->len = path.len - 1;
1541     v->valid = 1;
1542     v->no_cacheable = 0;
1543     v->not_found = 0;
1544     v->data = path.data;
1545
1546     return NGX_OK;
1547 }
1548
1549
1550 static ngx_int_t
1551 ngx_http_variable_server_name(ngx_http_request_t *r,
1552 ngx_http_variable_value_t *v, uintptr_t data)
1553 {
1554     ngx_http_core_srv_conf_t *cscf;
1555
1556     cscf = ngx_http_get_module_srv_conf(r, ngx_http_core_module);
1557
1558     v->len = cscf->server_name.len;
1559     v->valid = 1;
1560     v->no_cacheable = 0;
1561     v->not_found = 0;
1562     v->data = cscf->server_name.data;
1563
1564     return NGX_OK;
1565 }
1566
1567
1568 static ngx_int_t
1569 ngx_http_variable_request_method(ngx_http_request_t *r,
1570 ngx_http_variable_value_t *v, uintptr_t data)
1571 {
1572     if (r->main->method_name.data) {

```

```

1573     v->len = r->main->method_name.len;
1574     v->valid = 1;
1575     v->no_cacheable = 0;
1576     v->not_found = 0;
1577     v->data = r->main->method_name.data;
1578
1579 } else {
1580     v->not_found = 1;
1581 }
1582
1583 return NGX_OK;
1584 }
1585
1586
1587 static ngx_int_t
1588 ngx_http_variable_remote_user(ngx_http_request_t *r,
1589     ngx_http_variable_value_t *v, uintptr_t data)
1590 {
1591     ngx_int_t rc;
1592
1593     rc = ngx_http_auth_basic_user(r);
1594
1595     if (rc == NGX_DECLINED) {
1596         v->not_found = 1;
1597         return NGX_OK;
1598     }
1599
1600     if (rc == NGX_ERROR) {
1601         return NGX_ERROR;
1602     }
1603
1604     v->len = r->headers_in.user.len;
1605     v->valid = 1;
1606     v->no_cacheable = 0;
1607     v->not_found = 0;
1608     v->data = r->headers_in.user.data;
1609
1610     return NGX_OK;
1611 }
1612
1613
1614 static ngx_int_t
1615 ngx_http_variable_bytes_sent(ngx_http_request_t *r,
1616     ngx_http_variable_value_t *v, uintptr_t data)
1617 {
1618     u_char *p;
1619
1620     p = ngx_pnalloc(r->pool, NGX_OFF_T_LEN);
1621     if (p == NULL) {
1622         return NGX_ERROR;
1623     }
1624
1625     v->len = ngx_sprintf(p, "%O", r->connection->sent) - p;
1626     v->valid = 1;
1627     v->no_cacheable = 0;
1628     v->not_found = 0;
1629     v->data = p;
1630
1631     return NGX_OK;
1632 }
1633
1634
1635 static ngx_int_t
1636 ngx_http_variable_body_bytes_sent(ngx_http_request_t *r,
1637     ngx_http_variable_value_t *v, uintptr_t data)
1638 {
1639     off_t sent;
1640     u_char *p;
1641
1642     sent = r->connection->sent - r->header_size;
1643
1644     if (sent < 0) {
1645         sent = 0;
1646     }
1647
1648     p = ngx_pnalloc(r->pool, NGX_OFF_T_LEN);

```

```

1649     if (p == NULL) {
1650         return NGX\_ERROR;
1651     }
1652
1653     v->len = ngx\_sprintf(p, "%0", sent) - p;
1654     v->valid = 1;
1655     v->no_cacheable = 0;
1656     v->not_found = 0;
1657     v->data = p;
1658
1659     return NGX\_OK;
1660 }
1661
1662
1663 static ngx\_int\_t
1664 ngx\_http\_variable\_pipe(ngx\_http\_request\_t *r,
1665     ngx\_http\_variable\_value\_t *v, uintptr\_t data)
1666 {
1667     v->data = (u_char *) (r->pipeline ? "p" : ".");
1668     v->len = 1;
1669     v->valid = 1;
1670     v->no_cacheable = 0;
1671     v->not_found = 0;
1672
1673     return NGX\_OK;
1674 }
1675
1676
1677 static ngx\_int\_t
1678 ngx\_http\_variable\_status(ngx\_http\_request\_t *r,
1679     ngx\_http\_variable\_value\_t *v, uintptr\_t data)
1680 {
1681     ngx\_uint\_t status;
1682
1683     v->data = ngx\_pnalloc(r->pool, NGX\_INT\_T\_LEN);
1684     if (v->data == NULL) {
1685         return NGX\_ERROR;
1686     }
1687
1688     if (r->err_status) {
1689         status = r->err_status;
1690
1691     } else if (r->headers_out.status) {
1692         status = r->headers_out.status;
1693
1694     } else if (r->http_version == NGX\_HTTP\_VERSION\_9) {
1695         status = 9;
1696
1697     } else {
1698         status = 0;
1699     }
1700
1701     v->len = ngx\_sprintf(v->data, "%03ui", status) - v->data;
1702     v->valid = 1;
1703     v->no_cacheable = 0;
1704     v->not_found = 0;
1705
1706     return NGX\_OK;
1707 }
1708
1709
1710 static ngx\_int\_t
1711 ngx\_http\_variable\_sent\_content\_type(ngx\_http\_request\_t *r,
1712     ngx\_http\_variable\_value\_t *v, uintptr\_t data)
1713 {
1714     if (r->headers_out.content_type.len) {
1715         v->len = r->headers_out.content_type.len;
1716         v->valid = 1;
1717         v->no_cacheable = 0;
1718         v->not_found = 0;
1719         v->data = r->headers_out.content_type.data;
1720
1721     } else {
1722         v->not_found = 1;
1723     }
1724 }

```

```

1725     return NGX\_OK;
1726 }
1727
1728
1729 static ngx\_int\_t
1730 ngx\_http\_variable\_sent\_content\_length(ngx\_http\_request\_t *r,
1731     ngx\_http\_variable\_value\_t *v, uintptr\_t data)
1732 {
1733     u\_char *p;
1734
1735     if (r->headers_out.content_length) {
1736         v->len = r->headers_out.content_length->value.len;
1737         v->valid = 1;
1738         v->no_cacheable = 0;
1739         v->not_found = 0;
1740         v->data = r->headers_out.content_length->value.data;
1741
1742         return NGX\_OK;
1743     }
1744
1745     if (r->headers_out.content_length_n >= 0) {
1746         p = ngx\_pnalloc(r->pool, NGX\_OFF\_T\_LEN);
1747         if (p == NULL) {
1748             return NGX\_ERROR;
1749         }
1750
1751         v->len = ngx\_sprintf(p, "%0", r->headers_out.content_length_n) - p;
1752         v->valid = 1;
1753         v->no_cacheable = 0;
1754         v->not_found = 0;
1755         v->data = p;
1756
1757         return NGX\_OK;
1758     }
1759
1760     v->not_found = 1;
1761
1762     return NGX\_OK;
1763 }
1764
1765
1766 static ngx\_int\_t
1767 ngx\_http\_variable\_sent\_location(ngx\_http\_request\_t *r,
1768     ngx\_http\_variable\_value\_t *v, uintptr\_t data)
1769 {
1770     ngx\_str\_t name;
1771
1772     if (r->headers_out.location) {
1773         v->len = r->headers_out.location->value.len;
1774         v->valid = 1;
1775         v->no_cacheable = 0;
1776         v->not_found = 0;
1777         v->data = r->headers_out.location->value.data;
1778
1779         return NGX\_OK;
1780     }
1781
1782     ngx\_str\_set(&name, "sent_http_location");
1783
1784     return ngx\_http\_variable\_unknown\_header(v, &name,
1785         &r->headers_out.headers.part,
1786         sizeof("sent_http_") - 1);
1787 }
1788
1789
1790 static ngx\_int\_t
1791 ngx\_http\_variable\_sent\_last\_modified(ngx\_http\_request\_t *r,
1792     ngx\_http\_variable\_value\_t *v, uintptr\_t data)
1793 {
1794     u\_char *p;
1795
1796     if (r->headers_out.last_modified) {
1797         v->len = r->headers_out.last_modified->value.len;
1798         v->valid = 1;
1799         v->no_cacheable = 0;
1800         v->not_found = 0;

```

```

1801     v->data = r->headers_out.last_modified->value.data;
1802
1803     return NGX_OK;
1804 }
1805
1806 if (r->headers_out.last_modified_time >= 0) {
1807     p = ngx_pnalloc(r->pool, sizeof("Mon, 28 Sep 1970 06:00:00 GMT") - 1);
1808     if (p == NULL) {
1809         return NGX_ERROR;
1810     }
1811
1812     v->len = ngx_http_time(p, r->headers_out.last_modified_time) - p;
1813     v->valid = 1;
1814     v->no_cacheable = 0;
1815     v->not_found = 0;
1816     v->data = p;
1817
1818     return NGX_OK;
1819 }
1820
1821 v->not_found = 1;
1822
1823 return NGX_OK;
1824 }
1825
1826
1827 static ngx_int_t
1828 ngx_http_variable_sent_connection(ngx_http_request_t *r,
1829     ngx_http_variable_value_t *v, uintptr_t data)
1830 {
1831     size_t len;
1832     char *p;
1833
1834     if (r->headers_out.status == NGX_HTTP_SWITCHING_PROTOCOLS) {
1835         len = sizeof("upgrade") - 1;
1836         p = "upgrade";
1837
1838     } else if (r->keepalive) {
1839         len = sizeof("keep-alive") - 1;
1840         p = "keep-alive";
1841
1842     } else {
1843         len = sizeof("close") - 1;
1844         p = "close";
1845     }
1846
1847     v->len = len;
1848     v->valid = 1;
1849     v->no_cacheable = 0;
1850     v->not_found = 0;
1851     v->data = (u_char *) p;
1852
1853     return NGX_OK;
1854 }
1855
1856
1857 static ngx_int_t
1858 ngx_http_variable_sent_keep_alive(ngx_http_request_t *r,
1859     ngx_http_variable_value_t *v, uintptr_t data)
1860 {
1861     u_char *p;
1862     ngx_http_core_loc_conf_t *clcf;
1863
1864     if (r->keepalive) {
1865         clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
1866
1867         if (clcf->keepalive_header) {
1868
1869             p = ngx_pnalloc(r->pool, sizeof("timeout=") - 1 + NGX_TIME_T_LEN);
1870             if (p == NULL) {
1871                 return NGX_ERROR;
1872             }
1873
1874             v->len = ngx_sprintf(p, "timeout=%T", clcf->keepalive_header) - p;
1875             v->valid = 1;
1876             v->no_cacheable = 0;

```

```

1877         v->not_found = 0;
1878         v->data = p;
1879
1880         return NGX_OK;
1881     }
1882 }
1883
1884 v->not_found = 1;
1885
1886 return NGX_OK;
1887 }
1888
1889
1890 static ngx_int_t
1891 ngx_http_variable_sent_transfer_encoding(ngx_http_request_t *r,
1892     ngx_http_variable_value_t *v, uintptr_t data)
1893 {
1894     if (r->chunked) {
1895         v->len = sizeof("chunked") - 1;
1896         v->valid = 1;
1897         v->no_cacheable = 0;
1898         v->not_found = 0;
1899         v->data = (u_char *) "chunked";
1900
1901     } else {
1902         v->not_found = 1;
1903     }
1904
1905     return NGX_OK;
1906 }
1907
1908
1909 static ngx_int_t
1910 ngx_http_variable_request_completion(ngx_http_request_t *r,
1911     ngx_http_variable_value_t *v, uintptr_t data)
1912 {
1913     if (r->request_complete) {
1914         v->len = 2;
1915         v->valid = 1;
1916         v->no_cacheable = 0;
1917         v->not_found = 0;
1918         v->data = (u_char *) "OK";
1919
1920         return NGX_OK;
1921     }
1922
1923     v->len = 0;
1924     v->valid = 1;
1925     v->no_cacheable = 0;
1926     v->not_found = 0;
1927     v->data = (u_char *) "";
1928
1929     return NGX_OK;
1930 }
1931
1932
1933 static ngx_int_t
1934 ngx_http_variable_request_body(ngx_http_request_t *r,
1935     ngx_http_variable_value_t *v, uintptr_t data)
1936 {
1937     u_char      *p;
1938     size_t      len;
1939     ngx_buf_t   *buf;
1940     ngx_chain_t *cl;
1941
1942     if (r->request_body == NULL
1943         || r->request_body->bufs == NULL
1944         || r->request_body->temp_file)
1945     {
1946         v->not_found = 1;
1947
1948         return NGX_OK;
1949     }
1950
1951     cl = r->request_body->bufs;
1952     buf = cl->buf;

```

```

1953
1954     if (cl->next == NULL) {
1955         v->len = buf->last - buf->pos;
1956         v->valid = 1;
1957         v->no_cacheable = 0;
1958         v->not_found = 0;
1959         v->data = buf->pos;
1960
1961         return NGX_OK;
1962     }
1963
1964     len = buf->last - buf->pos;
1965     cl = cl->next;
1966
1967     for ( /* void */ ; cl; cl = cl->next) {
1968         buf = cl->buf;
1969         len += buf->last - buf->pos;
1970     }
1971
1972     p = ngx_pnalloc(r->pool, len);
1973     if (p == NULL) {
1974         return NGX_ERROR;
1975     }
1976
1977     v->data = p;
1978     cl = r->request_body->bufs;
1979
1980     for ( /* void */ ; cl; cl = cl->next) {
1981         buf = cl->buf;
1982         p = ngx_cpymem(p, buf->pos, buf->last - buf->pos);
1983     }
1984
1985     v->len = len;
1986     v->valid = 1;
1987     v->no_cacheable = 0;
1988     v->not_found = 0;
1989
1990     return NGX_OK;
1991 }
1992
1993
1994 static ngx_int_t
1995 ngx_http_variable_request_body_file(ngx_http_request_t *r,
1996     ngx_http_variable_value_t *v, uintptr_t data)
1997 {
1998     if (r->request_body == NULL || r->request_body->temp_file == NULL) {
1999         v->not_found = 1;
2000
2001         return NGX_OK;
2002     }
2003
2004     v->len = r->request_body->temp_file->file.name.len;
2005     v->valid = 1;
2006     v->no_cacheable = 0;
2007     v->not_found = 0;
2008     v->data = r->request_body->temp_file->file.name.data;
2009
2010     return NGX_OK;
2011 }
2012
2013
2014 static ngx_int_t
2015 ngx_http_variable_request_length(ngx_http_request_t *r,
2016     ngx_http_variable_value_t *v, uintptr_t data)
2017 {
2018     u_char *p;
2019
2020     p = ngx_pnalloc(r->pool, NGX_OFF_T_LEN);
2021     if (p == NULL) {
2022         return NGX_ERROR;
2023     }
2024
2025     v->len = ngx_sprintf(p, "%O", r->request_length) - p;
2026     v->valid = 1;
2027     v->no_cacheable = 0;
2028     v->not_found = 0;

```

```

2029     v->data = p;
2030
2031     return NGX_OK;
2032 }
2033
2034
2035 static ngx_int_t
2036 ngx_http_variable_request_time(ngx_http_request_t *r,
2037     ngx_http_variable_value_t *v, uintptr_t data)
2038 {
2039     u_char      *p;
2040     ngx_time_t  *tp;
2041     ngx_msec_int_t  ms;
2042
2043     p = ngx_pnalloc(r->pool, NGX_TIME_T_LEN + 4);
2044     if (p == NULL) {
2045         return NGX_ERROR;
2046     }
2047
2048     tp = ngx_timeofday();
2049
2050     ms = (ngx_msec_int_t)
2051         ((tp->sec - r->start_sec) * 1000 + (tp->msec - r->start_msec));
2052     ms = ngx_max(ms, 0);
2053
2054     v->len = ngx_sprintf(p, "%T.%03M", (time_t) ms / 1000, ms % 1000) - p;
2055     v->valid = 1;
2056     v->no_cacheable = 0;
2057     v->not_found = 0;
2058     v->data = p;
2059
2060     return NGX_OK;
2061 }
2062
2063
2064 static ngx_int_t
2065 ngx_http_variable_connection(ngx_http_request_t *r,
2066     ngx_http_variable_value_t *v, uintptr_t data)
2067 {
2068     u_char *p;
2069
2070     p = ngx_pnalloc(r->pool, NGX_ATOMIC_T_LEN);
2071     if (p == NULL) {
2072         return NGX_ERROR;
2073     }
2074
2075     v->len = ngx_sprintf(p, "%uA", r->connection->number) - p;
2076     v->valid = 1;
2077     v->no_cacheable = 0;
2078     v->not_found = 0;
2079     v->data = p;
2080
2081     return NGX_OK;
2082 }
2083
2084
2085 static ngx_int_t
2086 ngx_http_variable_connection_requests(ngx_http_request_t *r,
2087     ngx_http_variable_value_t *v, uintptr_t data)
2088 {
2089     u_char *p;
2090
2091     p = ngx_pnalloc(r->pool, NGX_INT_T_LEN);
2092     if (p == NULL) {
2093         return NGX_ERROR;
2094     }
2095
2096     v->len = ngx_sprintf(p, "%ui", r->connection->requests) - p;
2097     v->valid = 1;
2098     v->no_cacheable = 0;
2099     v->not_found = 0;
2100     v->data = p;
2101
2102     return NGX_OK;
2103 }
2104

```



```

2105 static ngx_int_t
2106 ngx_http_variable_nginx_version(ngx_http_request_t *r,
2107 ngx_http_variable_value_t *v, uintptr_t data)
2108 {
2109     v->len = sizeof(NGINX_VERSION) - 1;
2110     v->valid = 1;
2111     v->no_cacheable = 0;
2112     v->not_found = 0;
2113     v->data = (u_char *) NGINX_VERSION;
2114
2115     return NGX_OK;
2116 }
2117
2118
2119
2120 static ngx_int_t
2121 ngx_http_variable_hostname(ngx_http_request_t *r,
2122 ngx_http_variable_value_t *v, uintptr_t data)
2123 {
2124     v->len = ngx_cycle->hostname.len;
2125     v->valid = 1;
2126     v->no_cacheable = 0;
2127     v->not_found = 0;
2128     v->data = ngx_cycle->hostname.data;
2129
2130     return NGX_OK;
2131 }
2132
2133
2134 static ngx_int_t
2135 ngx_http_variable_pid(ngx_http_request_t *r,
2136 ngx_http_variable_value_t *v, uintptr_t data)
2137 {
2138     u_char *p;
2139
2140     p = ngx_pnalloc(r->pool, NGX_INT64_LEN);
2141     if (p == NULL) {
2142         return NGX_ERROR;
2143     }
2144
2145     v->len = ngx_sprintf(p, "%P", ngx_pid) - p;
2146     v->valid = 1;
2147     v->no_cacheable = 0;
2148     v->not_found = 0;
2149     v->data = p;
2150
2151     return NGX_OK;
2152 }
2153
2154
2155 static ngx_int_t
2156 ngx_http_variable_msec(ngx_http_request_t *r,
2157 ngx_http_variable_value_t *v, uintptr_t data)
2158 {
2159     u_char *p;
2160     ngx_time_t *tp;
2161
2162     p = ngx_pnalloc(r->pool, NGX_TIME_T_LEN + 4);
2163     if (p == NULL) {
2164         return NGX_ERROR;
2165     }
2166
2167     tp = ngx_timeofday();
2168
2169     v->len = ngx_sprintf(p, "%T.%03M", tp->sec, tp->msec) - p;
2170     v->valid = 1;
2171     v->no_cacheable = 0;
2172     v->not_found = 0;
2173     v->data = p;
2174
2175     return NGX_OK;
2176 }
2177
2178
2179 static ngx_int_t
2180 ngx_http_variable_time_iso8601(ngx_http_request_t *r,

```

```

2181     ngx\_http\_variable\_value\_t *v, uintptr\_t data)
2182 {
2183     u\_char *p;
2184
2185     p = ngx\_pnalloc(r->pool, ngx\_cached\_http\_log\_iso8601.len);
2186     if (p == NULL) {
2187         return NGX\_ERROR;
2188     }
2189
2190     ngx\_memcpy(p, ngx\_cached\_http\_log\_iso8601.data,
2191               ngx\_cached\_http\_log\_iso8601.len);
2192
2193     v->len = ngx\_cached\_http\_log\_iso8601.len;
2194     v->valid = 1;
2195     v->no_cacheable = 0;
2196     v->not_found = 0;
2197     v->data = p;
2198
2199     return NGX\_OK;
2200 }
2201
2202
2203 static ngx\_int\_t
2204 ngx\_http\_variable\_time\_local(ngx\_http\_request\_t *r,
2205 ngx\_http\_variable\_value\_t *v, uintptr\_t data)
2206 {
2207     u\_char *p;
2208
2209     p = ngx\_pnalloc(r->pool, ngx\_cached\_http\_log\_time.len);
2210     if (p == NULL) {
2211         return NGX\_ERROR;
2212     }
2213
2214     ngx\_memcpy(p, ngx\_cached\_http\_log\_time.data, ngx\_cached\_http\_log\_time.len);
2215
2216     v->len = ngx\_cached\_http\_log\_time.len;
2217     v->valid = 1;
2218     v->no_cacheable = 0;
2219     v->not_found = 0;
2220     v->data = p;
2221
2222     return NGX\_OK;
2223 }
2224
2225
2226 void *
2227 ngx\_http\_map\_find(ngx\_http\_request\_t *r, ngx\_http\_map\_t *map, ngx\_str\_t *match)
2228 {
2229     void *value;
2230     u\_char *low;
2231     size\_t len;
2232     ngx\_uint\_t key;
2233
2234     len = match->len;
2235
2236     if (len) {
2237         low = ngx\_pnalloc(r->pool, len);
2238         if (low == NULL) {
2239             return NULL;
2240         }
2241     }
2242     else {
2243         low = NULL;
2244     }
2245
2246     key = ngx\_hash\_strlow(low, match->data, len);
2247
2248     value = ngx\_hash\_find\_combined(&map->hash, key, low, len);
2249     if (value) {
2250         return value;
2251     }
2252
2253     #if (NGX\_PCRE)
2254
2255     if (len && map->nregex) {
2256         ngx\_int\_t n;

```

```

2257     ngx_uint_t          i;
2258     ngx_http_map_regex_t *reg;
2259
2260     reg = map->regex;
2261
2262     for (i = 0; i < map->nregex; i++) {
2263
2264         n = ngx_http_regex_exec(r, reg[i].regex, match);
2265
2266         if (n == NGX_OK) {
2267             return reg[i].value;
2268         }
2269
2270         if (n == NGX_DECLINED) {
2271             continue;
2272         }
2273
2274         /* NGX_ERROR */
2275
2276         return NULL;
2277     }
2278 }
2279
2280 #endif
2281
2282     return NULL;
2283 }
2284
2285
2286 #if (NGX_PCRE)
2287
2288 static ngx_int_t
2289 ngx_http_variable_not_found(ngx_http_request_t *r, ngx_http_variable_value_t *v,
2290     uintptr_t data)
2291 {
2292     v->not_found = 1;
2293     return NGX_OK;
2294 }
2295
2296
2297 ngx_http_regex_t *
2298 ngx_http_regex_compile(ngx_conf_t *cf, ngx_regex_compile_t *rc)
2299 {
2300     u_char          *p;
2301     size_t          size;
2302     ngx_str_t       name;
2303     ngx_uint_t      i, n;
2304     ngx_http_variable_t *v;
2305     ngx_http_regex_t *re;
2306     ngx_http_regex_variable_t *rv;
2307     ngx_http_core_main_conf_t *cmcf;
2308
2309     rc->pool = cf->pool;
2310
2311     if (ngx_regex_compile(rc) != NGX_OK) {
2312         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0, "%V", &rc->err);
2313         return NULL;
2314     }
2315
2316     re = ngx_palloc(cf->pool, sizeof(ngx_http_regex_t));
2317     if (re == NULL) {
2318         return NULL;
2319     }
2320
2321     re->regex = rc->regex;
2322     re->ncaptures = rc->captures;
2323     re->name = rc->pattern;
2324
2325     cmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_core_module);
2326     cmcf->ncaptures = ngx_max(cmcf->ncaptures, re->ncaptures);
2327
2328     n = (ngx_uint_t) rc->named_captures;
2329
2330     if (n == 0) {
2331         return re;
2332     }

```

```

2333     rv = ngx_palloc(rc->pool, n * sizeof(ngx_http_regex_variable_t));
2334     if (rv == NULL) {
2335         return NULL;
2336     }
2337
2338     re->variables = rv;
2339     re->nvariables = n;
2340
2341     size = rc->name_size;
2342     p = rc->names;
2343
2344     for (i = 0; i < n; i++) {
2345         rv[i].capture = 2 * ((p[0] << 8) + p[1]);
2346
2347         name.data = &p[2];
2348         name.len = ngx_strlen(name.data);
2349
2350         v = ngx_http_add_variable(cf, &name, NGX_HTTP_VAR_CHANGEABLE);
2351         if (v == NULL) {
2352             return NULL;
2353         }
2354
2355         rv[i].index = ngx_http_get_variable_index(cf, &name);
2356         if (rv[i].index == NGX_ERROR) {
2357             return NULL;
2358         }
2359
2360         v->get_handler = ngx_http_variable_not_found;
2361
2362         p += size;
2363     }
2364
2365     return re;
2366 }
2367
2368
2369
2370 ngx_int_t
2371 ngx_http_regex_exec(ngx_http_request_t *r, ngx_http_regex_t *re, ngx_str_t *s)
2372 {
2373     ngx_int_t          rc, index;
2374     ngx_uint_t         i, n, len;
2375     ngx_http_variable_value_t *vv;
2376     ngx_http_core_main_conf_t *cmcf;
2377
2378     cmcf = ngx_http_get_module_main_conf(r, ngx_http_core_module);
2379
2380     if (re->ncaptures) {
2381         len = cmcf->ncaptures;
2382
2383         if (r->captures == NULL) {
2384             r->captures = ngx_palloc(r->pool, len * sizeof(int));
2385             if (r->captures == NULL) {
2386                 return NGX_ERROR;
2387             }
2388         }
2389     }
2390
2391     else {
2392         len = 0;
2393     }
2394
2395     rc = ngx_regex_exec(re->regex, s, r->captures, len);
2396
2397     if (rc == NGX_REGEX_NO_MATCHED) {
2398         return NGX_DECLINED;
2399     }
2400
2401     if (rc < 0) {
2402         ngx_log_error(NGX_LOG_ALERT, r->connection->log, 0,
2403             ngx_regex_exec_n " failed: %i on \"%V\" using \"%V\"",
2404             rc, s, &re->name);
2405         return NGX_ERROR;
2406     }
2407
2408     for (i = 0; i < re->nvariables; i++) {

```

```

2409     n = re->variables[i].capture;
2410     index = re->variables[i].index;
2411     vv = &r->variables[index];
2412
2413     vv->len = r->captures[n + 1] - r->captures[n];
2414     vv->valid = 1;
2415     vv->no_cacheable = 0;
2416     vv->not_found = 0;
2417     vv->data = &s->data[r->captures[n]];
2418
2419     #if (NGX_DEBUG)
2420     {
2421         ngx_http_variable_t *v;
2422
2423         v = cmcf->variables.elts;
2424
2425         ngx_log_debug3(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
2426             "http regex set %V to \"%s\"",
2427             &v[index].name, vv->len, vv->data);
2428     }
2429 #endif
2430 }
2431
2432 r->ncaptures = rc * 2;
2433 r->captures_data = s->data;
2434
2435 return NGX_OK;
2436 }
2437
2438 #endif
2439
2440
2441 ngx_int_t
2442 ngx_http_variables_add_core_vars(ngx_conf_t *cf)
2443 {
2444     ngx_int_t rc;
2445     ngx_http_variable_t *cv, *v;
2446     ngx_http_core_main_conf_t *cmcf;
2447
2448     cmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_core_module);
2449
2450     cmcf->variables_keys = ngx_palloc(cf->temp_pool,
2451         sizeof(ngx_hash_keys_arrays_t));
2452     if (cmcf->variables_keys == NULL) {
2453         return NGX_ERROR;
2454     }
2455
2456     cmcf->variables_keys->pool = cf->pool;
2457     cmcf->variables_keys->temp_pool = cf->temp_pool;
2458
2459     if (ngx_hash_keys_array_init(cmcf->variables_keys, NGX_HASH_SMALL)
2460         != NGX_OK)
2461     {
2462         return NGX_ERROR;
2463     }
2464
2465     for (cv = ngx_http_core_variables; cv->name.len; cv++) {
2466         v = ngx_palloc(cf->pool, sizeof(ngx_http_variable_t));
2467         if (v == NULL) {
2468             return NGX_ERROR;
2469         }
2470
2471         *v = *cv;
2472
2473         rc = ngx_hash_add_key(cmcf->variables_keys, &v->name, v,
2474             NGX_HASH_READONLY_KEY);
2475
2476         if (rc == NGX_OK) {
2477             continue;
2478         }
2479
2480         if (rc == NGX_BUSY) {
2481             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
2482                 "conflicting variable name \"%V\"", &v->name);
2483         }
2484     }

```

```

2485     return NGX\_ERROR;
2486 }
2487
2488     return NGX\_OK;
2489 }
2490
2491
2492 ngx\_int\_t
2493 ngx\_http\_variables\_init\_vars(ngx\_conf\_t *cf)
2494 {
2495     ngx\_uint\_t          i, n;
2496     ngx\_hash\_key\_t     *key;
2497     ngx\_hash\_init\_t     hash;
2498     ngx\_http\_variable\_t *v, *av;
2499     ngx\_http\_core\_main\_conf\_t *cmcf;
2500
2501     /* set the handlers for the indexed http variables */
2502
2503     cmcf = ngx\_http\_conf\_get\_module\_main\_conf(cf, ngx\_http\_core\_module);
2504
2505     v = cmcf->variables.elts;
2506     key = cmcf->variables_keys->keys.elts;
2507
2508     for (i = 0; i < cmcf->variables.nelts; i++) {
2509
2510         for (n = 0; n < cmcf->variables_keys->keys.nelts; n++) {
2511
2512             av = key[n].value;
2513
2514             if (v[i].name.len == key[n].key.len
2515                 && ngx\_strncmp(v[i].name.data, key[n].key.data, v[i].name.len)
2516                     == 0)
2517             {
2518                 v[i].get_handler = av->get_handler;
2519                 v[i].data = av->data;
2520
2521                 av->flags |= NGX\_HTTP\_VAR\_INDEXED;
2522                 v[i].flags = av->flags;
2523
2524                 av->index = i;
2525
2526                 if (av->get_handler == NULL) {
2527                     break;
2528                 }
2529
2530                 goto next;
2531             }
2532         }
2533
2534         if (ngx\_strncmp(v[i].name.data, "http_", 5) == 0) {
2535             v[i].get_handler = ngx\_http\_variable\_unknown\_header\_in;
2536             v[i].data = (uintptr\_t) &v[i].name;
2537
2538             continue;
2539         }
2540
2541         if (ngx\_strncmp(v[i].name.data, "sent_http_", 10) == 0) {
2542             v[i].get_handler = ngx\_http\_variable\_unknown\_header\_out;
2543             v[i].data = (uintptr\_t) &v[i].name;
2544
2545             continue;
2546         }
2547
2548         if (ngx\_strncmp(v[i].name.data, "upstream_http_", 14) == 0) {
2549             v[i].get_handler = ngx\_http\_upstream\_header\_variable;
2550             v[i].data = (uintptr\_t) &v[i].name;
2551             v[i].flags = NGX\_HTTP\_VAR\_NOCACHEABLE;
2552
2553             continue;
2554         }
2555
2556         if (ngx\_strncmp(v[i].name.data, "cookie_", 7) == 0) {
2557             v[i].get_handler = ngx\_http\_variable\_cookie;
2558             v[i].data = (uintptr\_t) &v[i].name;
2559
2560             continue;

```

```

2561     }
2562
2563     if (ngx_strcmp(v[i].name.data, "upstream_cookie_", 16) == 0) {
2564         v[i].get_handler = ngx_http_upstream_cookie_variable;
2565         v[i].data = (uintptr_t) &v[i].name;
2566         v[i].flags = NGX_HTTP_VAR_NOCACHEABLE;
2567
2568         continue;
2569     }
2570
2571     if (ngx_strcmp(v[i].name.data, "arg_", 4) == 0) {
2572         v[i].get_handler = ngx_http_variable_argument;
2573         v[i].data = (uintptr_t) &v[i].name;
2574         v[i].flags = NGX_HTTP_VAR_NOCACHEABLE;
2575
2576         continue;
2577     }
2578
2579     ngx_log_error(NGX_LOG_EMERG, cf->log, 0,
2580                 "unknown \"%V\" variable", &v[i].name);
2581
2582     return NGX_ERROR;
2583
2584 next:
2585     continue;
2586 }
2587
2588
2589 for (n = 0; n < cmcf->variables_keys->keys.nelts; n++) {
2590     av = key[n].value;
2591
2592     if (av->flags & NGX_HTTP_VAR_NOHASH) {
2593         key[n].key.data = NULL;
2594     }
2595 }
2596
2597
2598 hash.hash = &cmcf->variables_hash;
2599 hash.key = ngx_hash_key;
2600 hash.max_size = cmcf->variables_hash_max_size;
2601 hash.bucket_size = cmcf->variables_hash_bucket_size;
2602 hash.name = "variables_hash";
2603 hash.pool = cf->pool;
2604 hash.temp_pool = NULL;
2605
2606 if (ngx_hash_init(&hash, cmcf->variables_keys->keys.elts,
2607                 cmcf->variables_keys->keys.nelts)
2608     != NGX_OK)
2609 {
2610     return NGX_ERROR;
2611 }
2612
2613 cmcf->variables_keys = NULL;
2614
2615 return NGX_OK;
2616 }

```

[One Level Up](#)

[Top Level](#)

## src/core/nginx.h - nginx-1.7.10

### Macros defined

- [NGINX\\_VAR](#)
- [NGINX\\_VER](#)
- [NGINX\\_VERSION](#)
- [NGINX\\_VER\\_BUILD](#)
- [NGINX\\_VER\\_BUILD](#)
- [NGX\\_OLDPID\\_EXT](#)
- [\\_NGINX\\_H\\_INCLUDED](#)
- [nginx\\_version](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGINX\_H\_INCLUDED
9 #define \_NGINX\_H\_INCLUDED
10
11
12 #define nginx_version          1007010
13 #define NGINX_VERSION         "1.7.10"
14 #define NGINX_VER             "nginx/" NGINX\_VERSION"
15
16 #ifdef NGX_BUILD
17 #define NGINX_VER_BUILD       NGINX\_VER " (" NGX_BUILD ")"
18 #else
19 #define NGINX_VER_BUILD       NGINX\_VER
20 #endif
21
22 #define NGINX_VAR              "NGINX"
23 #define NGX_OLDPID_EXT        ".oldbin"
24
25
26 #endif /* \_NGINX\_H\_INCLUDED */
```



## src/core/nginx\_conf\_file.c - nginx-1.7.10

### Global variables defined

- [argument\\_number](#)
- [ngx\\_conf\\_commands](#)
- [ngx\\_conf\\_module](#)

### Functions defined

- [ngx\\_conf\\_check\\_num\\_bounds](#)
- [ngx\\_conf\\_deprecated](#)
- [ngx\\_conf\\_flush\\_files](#)
- [ngx\\_conf\\_full\\_name](#)
- [ngx\\_conf\\_handler](#)
- [ngx\\_conf\\_include](#)
- [ngx\\_conf\\_log\\_error](#)
- [ngx\\_conf\\_open\\_file](#)
- [ngx\\_conf\\_param](#)
- [ngx\\_conf\\_parse](#)
- [ngx\\_conf\\_read\\_token](#)
- [ngx\\_conf\\_set\\_bitmask\\_slot](#)
- [ngx\\_conf\\_set\\_bufs\\_slot](#)
- [ngx\\_conf\\_set\\_enum\\_slot](#)
- [ngx\\_conf\\_set\\_flag\\_slot](#)
- [ngx\\_conf\\_set\\_keyval\\_slot](#)
- [ngx\\_conf\\_set\\_msec\\_slot](#)
- [ngx\\_conf\\_set\\_num\\_slot](#)
- [ngx\\_conf\\_set\\_off\\_slot](#)
- [ngx\\_conf\\_set\\_sec\\_slot](#)
- [ngx\\_conf\\_set\\_size\\_slot](#)
- [ngx\\_conf\\_set\\_str\\_array\\_slot](#)
- [ngx\\_conf\\_set\\_str\\_slot](#)

### Macros defined

- [NGX\\_CONF\\_BUFFER](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10
11 #define NGX_CONF_BUFFER 4096
12
13 static ngx_int_t ngx_conf_handler(ngx_conf_t *cf, ngx_int_t last);
14 static ngx_int_t ngx_conf_read_token(ngx_conf_t *cf);
15 static void ngx_conf_flush_files(ngx_cycle_t *cycle);
16
17
18 static ngx_command_t ngx_conf_commands[] = {
19
20     { ngx_string("include"),
21       NGX_ANY_CONF|NGX_CONF_TAKE1,
22       ngx_conf_include,
23       0,
24       0,
25       NULL },
26
27     ngx_null_command
28 };
29
30
31 ngx_module_t ngx_conf_module = {
32     NGX_MODULE_V1,
33     NULL, /* module context */
34     ngx_conf_commands, /* module directives */
35     NGX_CONF_MODULE, /* module type */
36     NULL, /* init master */
37     NULL, /* init module */
38     NULL, /* init process */
39     NULL, /* init thread */
40     NULL, /* exit thread */
41     ngx_conf_flush_files, /* exit process */
42     NULL, /* exit master */
43     NGX_MODULE_V1_PADDING
44 };
45
46
47 /* The eight fixed arguments */
48
49 static ngx_uint_t argument_number[] = {
50     NGX_CONF_NOARGS,
51     NGX_CONF_TAKE1,
52     NGX_CONF_TAKE2,
53     NGX_CONF_TAKE3,
54     NGX_CONF_TAKE4,
55     NGX_CONF_TAKE5,
56     NGX_CONF_TAKE6,
57     NGX_CONF_TAKE7
58 };
59
60
61 char *
62 ngx_conf_param(ngx_conf_t *cf)
63 {
64     char *rv;
65     ngx_str_t *param;
66     ngx_buf_t b;
67     ngx_conf_file_t conf_file;
68
69     param = &cf->cycle->conf_param;
70
71     if (param->len == 0) {
72         return NGX_CONF_OK;
73     }

```

```

74     ngx_memzero(&conf_file, sizeof(ngx_conf_file_t));
75
76
77     ngx_memzero(&b, sizeof(ngx_buf_t));
78
79     b.start = param->data;
80     b.pos = param->data;
81     b.last = param->data + param->len;
82     b.end = b.last;
83     b.temporary = 1;
84
85     conf_file.file.fd = NGX_INVALID_FILE;
86     conf_file.file.name.data = NULL;
87     conf_file.line = 0;
88
89     cf->conf_file = &conf_file;
90     cf->conf_file->buffer = &b;
91
92     rv = ngx_conf_parse(cf, NULL);
93
94     cf->conf_file = NULL;
95
96     return rv;
97 }
98
99
100 char *
101 ngx_conf_parse(ngx_conf_t *cf, ngx_str_t *filename)
102 {
103     char          *rv;
104     ngx_fd_t      fd;
105     ngx_int_t     rc;
106     ngx_buf_t     buf;
107     ngx_conf_file_t *prev, conf_file;
108     enum {
109         parse_file = 0,
110         parse_block,
111         parse_param
112     } type;
113
114     #if (NGX_SUPPRESS_WARN)
115         fd = NGX_INVALID_FILE;
116         prev = NULL;
117     #endif
118
119     if (filename) {
120
121         /* open configuration file */
122
123         fd = ngx_open_file(filename->data, NGX_FILE_RDONLY, NGX_FILE_OPEN, 0);
124         if (fd == NGX_INVALID_FILE) {
125             ngx_conf_log_error(NGX_LOG_EMERG, cf, ngx_errno,
126                 ngx_open_file_n "%s" failed",
127                 filename->data);
128             return NGX_CONF_ERROR;
129         }
130
131         prev = cf->conf_file;
132
133         cf->conf_file = &conf_file;
134
135         if (ngx_fd_info(fd, &cf->conf_file->file.info) == NGX_FILE_ERROR) {
136             ngx_log_error(NGX_LOG_EMERG, cf->log, ngx_errno,
137                 ngx_fd_info_n "%s" failed", filename->data);
138         }
139
140         cf->conf_file->buffer = &buf;
141
142         buf.start = ngx_alloc(NGX_CONF_BUFFER, cf->log);
143         if (buf.start == NULL) {
144             goto failed;
145         }
146
147         buf.pos = buf.start;
148         buf.last = buf.start;
149         buf.end = buf.last + NGX_CONF_BUFFER;

```

```

150     buf.temporary = 1;
151
152     cf->conf_file->file.fd = fd;
153     cf->conf_file->file.name.len = filename->len;
154     cf->conf_file->file.name.data = filename->data;
155     cf->conf_file->file.offset = 0;
156     cf->conf_file->file.log = cf->log;
157     cf->conf_file->line = 1;
158
159     type = parse_file;
160
161 } else if (cf->conf_file->file.fd != NGX_INVALID_FILE) {
162
163     type = parse_block;
164
165 } else {
166     type = parse_param;
167 }
168
169
170 for ( ;; ) {
171     rc = ngx_conf_read_token(cf);
172
173     /*
174     * ngx_conf_read_token() may return
175     *
176     * NGX_ERROR           there is error
177     * NGX_OK             the token terminated by ";" was found
178     * NGX_CONF_BLOCK_START the token terminated by "{" was found
179     * NGX_CONF_BLOCK_DONE the "}" was found
180     * NGX_CONF_FILE_DONE the configuration file is done
181     */
182
183     if (rc == NGX_ERROR) {
184         goto done;
185     }
186
187     if (rc == NGX_CONF_BLOCK_DONE) {
188
189         if (type != parse_block) {
190             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0, "unexpected \"}\"");
191             goto failed;
192         }
193
194         goto done;
195     }
196
197     if (rc == NGX_CONF_FILE_DONE) {
198
199         if (type == parse_block) {
200             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
201                 "unexpected end of file, expecting \"}\"");
202             goto failed;
203         }
204
205         goto done;
206     }
207
208     if (rc == NGX_CONF_BLOCK_START) {
209
210         if (type == parse_param) {
211             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
212                 "block directives are not supported "
213                 "in -g option");
214             goto failed;
215         }
216     }
217
218     /* rc == NGX_OK || rc == NGX_CONF_BLOCK_START */
219
220     if (cf->handler) {
221
222         /*
223         * the custom handler, i.e., that is used in the http's
224         * "types { ... }" directive
225         */

```

```

226     if (rc == NGX\_CONF\_BLOCK\_START) {
227         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0, "unexpected \"{\}\"");
228         goto failed;
229     }
230
231
232     rv = (*cf->handler)(cf, NULL, cf->handler_conf);
233     if (rv == NGX\_CONF\_OK) {
234         continue;
235     }
236
237     if (rv == NGX\_CONF\_ERROR) {
238         goto failed;
239     }
240
241     ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0, rv);
242
243     goto failed;
244 }
245
246 rc = ngx\_conf\_handler(cf, rc);
247
248 if (rc == NGX\_ERROR) {
249     goto failed;
250 }
251 }
252 }
253
254 failed:
255
256     rc = NGX\_ERROR;
257
258 done:
259
260 if (filename) {
261     if (cf->conf_file->buffer->start) {
262         ngx\_free(cf->conf_file->buffer->start);
263     }
264
265     if (ngx\_close\_file(fd) == NGX\_FILE\_ERROR) {
266         ngx\_log\_error(NGX\_LOG\_ALERT, cf->log, ngx\_errno,
267             ngx\_close\_file n " %s failed",
268             filename->data);
269         rc = NGX\_ERROR;
270     }
271
272     cf->conf_file = prev;
273 }
274
275 if (rc == NGX\_ERROR) {
276     return NGX\_CONF\_ERROR;
277 }
278
279 return NGX\_CONF\_OK;
280 }
281
282
283 static ngx\_int\_t
284 ngx\_conf\_handler(ngx\_conf\_t *cf, ngx\_int\_t last)
285 {
286     char          *rv;
287     void          *conf, **confp;
288     ngx\_uint\_t    i, found;
289     ngx\_str\_t     *name;
290     ngx\_command\_t *cmd;
291
292     name = cf->args->elts;
293
294     found = 0;
295
296     for (i = 0; ngx_modules[i]; i++) {
297
298         cmd = ngx_modules[i]->commands;
299         if (cmd == NULL) {
300             continue;
301         }

```

```

302
303 for ( /* void */ ; cmd->name.len; cmd++) {
304
305     if (name->len != cmd->name.len) {
306         continue;
307     }
308
309     if (ngx_strcmp(name->data, cmd->name.data) != 0) {
310         continue;
311     }
312
313     found = 1;
314
315     if (ngx_modules[i]->type != NGX_CONF_MODULE
316         && ngx_modules[i]->type != cf->module_type)
317     {
318         continue;
319     }
320
321     /* is the directive's location right ? */
322
323     if (!(cmd->type & cf->cmd_type)) {
324         continue;
325     }
326
327     if (!(cmd->type & NGX_CONF_BLOCK) && last != NGX_OK) {
328         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
329             "directive \"%s\" is not terminated by \";\",",
330             name->data);
331         return NGX_ERROR;
332     }
333
334     if ((cmd->type & NGX_CONF_BLOCK) && last != NGX_CONF_BLOCK_START) {
335         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
336             "directive \"%s\" has no opening \"{\"",
337             name->data);
338         return NGX_ERROR;
339     }
340
341     /* is the directive's argument count right ? */
342
343     if (!(cmd->type & NGX_CONF_ANY)) {
344
345         if (cmd->type & NGX_CONF_FLAG) {
346
347             if (cf->args->nelts != 2) {
348                 goto invalid;
349             }
350
351         } else if (cmd->type & NGX_CONF_1MORE) {
352
353             if (cf->args->nelts < 2) {
354                 goto invalid;
355             }
356
357         } else if (cmd->type & NGX_CONF_2MORE) {
358
359             if (cf->args->nelts < 3) {
360                 goto invalid;
361             }
362
363         } else if (cf->args->nelts > NGX_CONF_MAX_ARGS) {
364
365             goto invalid;
366
367         } else if (!(cmd->type & argument_number[cf->args->nelts - 1]))
368         {
369             goto invalid;
370         }
371     }
372
373     /* set up the directive's configuration context */
374
375     conf = NULL;
376
377     if (cmd->type & NGX_DIRECT_CONF) {

```

```

378         conf = ((void **) cf->ctx)[ngx_modules[i]->index];
379
380     } else if (cmd->type & NGX_MAIN_CONF) {
381         conf = &((void **) cf->ctx)[ngx_modules[i]->index];
382
383     } else if (cf->ctx) {
384         confp = *(void **) ((char *) cf->ctx + cmd->conf);
385
386         if (confp) {
387             conf = confp[ngx_modules[i]->ctx_index];
388         }
389     }
390
391     rv = cmd->set(cf, cmd, conf);
392
393     if (rv == NGX_CONF_OK) {
394         return NGX_OK;
395     }
396
397     if (rv == NGX_CONF_ERROR) {
398         return NGX_ERROR;
399     }
400
401     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
402                       "\"%s\" directive %s", name->data, rv);
403
404     return NGX_ERROR;
405 }
406
407
408 if (found) {
409     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
410                       "\"%s\" directive is not allowed here", name->data);
411
412     return NGX_ERROR;
413 }
414
415 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
416                   "unknown directive \"%s\"", name->data);
417
418 return NGX_ERROR;
419
420 invalid:
421
422 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
423                   "invalid number of arguments in \"%s\" directive",
424                   name->data);
425
426 return NGX_ERROR;
427 }
428
429
430 static ngx_int_t
431 ngx_conf_read_token(ngx_conf_t *cf)
432 {
433     u_char      *start, ch, *src, *dst;
434     off_t        file_size;
435     size_t       len;
436     ssize_t      n, size;
437     ngx_uint_t   found, need_space, last_space, sharp_comment, variable;
438     ngx_uint_t   quoted, s_quoted, d_quoted, start_line;
439     ngx_str_t    *word;
440     ngx_buf_t    *b;
441
442     found = 0;
443     need_space = 0;
444     last_space = 1;
445     sharp_comment = 0;
446     variable = 0;
447     quoted = 0;
448     s_quoted = 0;
449     d_quoted = 0;
450
451     cf->args->nelts = 0;
452     b = cf->conf_file->buffer;
453     start = b->pos;

```

```

454 start_line = cf->conf_file->line;
455
456 file_size = ngx_file_size(&cf->conf_file->file.info);
457
458 for ( ;; ) {
459     if (b->pos >= b->last) {
460
461         if (cf->conf_file->file.offset >= file_size) {
462
463             if (cf->args->nelts > 0 || !last_space) {
464
465                 if (cf->conf_file->file.fd == NGX_INVALID_FILE) {
466                     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
467                                     "unexpected end of parameter, "
468                                     "expecting \";\"");
469                     return NGX_ERROR;
470                 }
471
472                 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
473                                     "unexpected end of file, "
474                                     "expecting \";\" or \"}\"");
475                 return NGX_ERROR;
476             }
477
478             return NGX_CONF_FILE_DONE;
479         }
480
481         len = b->pos - start;
482
483         if (len == NGX_CONF_BUFFER) {
484             cf->conf_file->line = start_line;
485
486             if (d_quoted) {
487                 ch = '"';
488
489             } else if (s_quoted) {
490                 ch = '\\';
491
492             } else {
493                 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
494                                     "too long parameter \"%*s...\" started",
495                                     10, start);
496                 return NGX_ERROR;
497             }
498
499             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
500                                     "too long parameter, probably "
501                                     "missing terminating \"%c\" character", ch);
502             return NGX_ERROR;
503         }
504
505         if (len) {
506             ngx_memmove(b->start, start, len);
507         }
508
509         size = (ssize_t) (file_size - cf->conf_file->file.offset);
510
511         if (size > b->end - (b->start + len)) {
512             size = b->end - (b->start + len);
513         }
514
515         n = ngx_read_file(&cf->conf_file->file, b->start + len, size,
516                         cf->conf_file->file.offset);
517
518         if (n == NGX_ERROR) {
519             return NGX_ERROR;
520         }
521
522         if (n != size) {
523             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
524                                     ngx_read_file_n " returned "
525                                     "only %z bytes instead of %z",
526                                     n, size);
527             return NGX_ERROR;
528         }
529     }

```



```

530         b->pos = b->start + len;
531         b->last = b->pos + n;
532         start = b->start;
533     }
534 }
535
536 ch = *b->pos++;
537
538 if (ch == LF) {
539     cf->conf_file->line++;
540
541     if (sharp_comment) {
542         sharp_comment = 0;
543     }
544 }
545
546 if (sharp_comment) {
547     continue;
548 }
549
550 if (quoted) {
551     quoted = 0;
552     continue;
553 }
554
555 if (need_space) {
556     if (ch == ' ' || ch == '\t' || ch == CR || ch == LF) {
557         last_space = 1;
558         need_space = 0;
559         continue;
560     }
561
562     if (ch == ';') {
563         return NGX_OK;
564     }
565
566     if (ch == '{') {
567         return NGX_CONF_BLOCK_START;
568     }
569
570     if (ch == ')') {
571         last_space = 1;
572         need_space = 0;
573     }
574
575     } else {
576         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
577             "unexpected \"%c\"", ch);
578         return NGX_ERROR;
579     }
580 }
581
582 if (last_space) {
583     if (ch == ' ' || ch == '\t' || ch == CR || ch == LF) {
584         continue;
585     }
586
587     start = b->pos - 1;
588     start_line = cf->conf_file->line;
589
590     switch (ch) {
591     case ';':
592     case '{':
593         if (cf->args->nelts == 0) {
594             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
595                 "unexpected \"%c\"", ch);
596             return NGX_ERROR;
597         }
598
599         if (ch == '{') {
600             return NGX_CONF_BLOCK_START;
601         }
602
603         return NGX_OK;
604
605     case '}':

```

```

606     if (cf->args->nelts != 0) {
607         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
608             "unexpected \"}\");
609         return NGX_ERROR;
610     }
611
612     return NGX_CONF_BLOCK_DONE;
613
614     case '#':
615         sharp_comment = 1;
616         continue;
617
618     case '\\':
619         quoted = 1;
620         last_space = 0;
621         continue;
622
623     case '":
624         start++;
625         d_quoted = 1;
626         last_space = 0;
627         continue;
628
629     case '\':
630         start++;
631         s_quoted = 1;
632         last_space = 0;
633         continue;
634
635     default:
636         last_space = 0;
637     }
638
639 } else {
640     if (ch == '{' && variable) {
641         continue;
642     }
643
644     variable = 0;
645
646     if (ch == '\\') {
647         quoted = 1;
648         continue;
649     }
650
651     if (ch == '$') {
652         variable = 1;
653         continue;
654     }
655
656     if (d_quoted) {
657         if (ch == '"') {
658             d_quoted = 0;
659             need_space = 1;
660             found = 1;
661         }
662
663     } else if (s_quoted) {
664         if (ch == '\') {
665             s_quoted = 0;
666             need_space = 1;
667             found = 1;
668         }
669
670     } else if (ch == ' ' || ch == '\t' || ch == CR || ch == LF
671         || ch == ';' || ch == '{')
672     {
673         last_space = 1;
674         found = 1;
675     }
676
677     if (found) {
678         word = ngx_array_push(cf->args);
679         if (word == NULL) {
680             return NGX_ERROR;
681         }

```

```

682
683 word->data = ngx_pnalloc(cf->pool, b->pos - start + 1);
684 if (word->data == NULL) {
685     return NGX_ERROR;
686 }
687
688 for (dst = word->data, src = start, len = 0;
689     src < b->pos - 1;
690     len++)
691 {
692     if (*src == '\\') {
693         switch (src[1]) {
694             case '"':
695             case '\':
696             case '\\':
697                 src++;
698                 break;
699
700             case 't':
701                 *dst++ = '\t';
702                 src += 2;
703                 continue;
704
705             case 'r':
706                 *dst++ = '\r';
707                 src += 2;
708                 continue;
709
710             case 'n':
711                 *dst++ = '\n';
712                 src += 2;
713                 continue;
714         }
715     }
716     *dst++ = *src++;
717 }
718 *dst = '\0';
719 word->len = len;
720
721 if (ch == ';') {
722     return NGX_OK;
723 }
724
725 if (ch == '{') {
726     return NGX_CONF_BLOCK_START;
727 }
728
729 found = 0;
730 }
731 }
732 }
733 }
734 }
735
736
737 char *
738 ngx_conf_include(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
739 {
740     char *rv;
741     ngx_int_t n;
742     ngx_str_t *value, file, name;
743     ngx_glob_t gl;
744
745     value = cf->args->elts;
746     file = value[1];
747
748     ngx_log_debug1(NGX_LOG_DEBUG_CORE, cf->log, 0, "include %s", file.data);
749
750     if (ngx_conf_full_name(cf->cycle, &file, 1) != NGX_OK) {
751         return NGX_CONF_ERROR;
752     }
753
754     if (strpbrk((char *) file.data, "?[") == NULL) {
755
756         ngx_log_debug1(NGX_LOG_DEBUG_CORE, cf->log, 0, "include %s", file.data);
757

```

```

758     return ngx_conf_parse(cf, &file);
759 }
760
761 ngx_memzero(&gl, sizeof(ngx_glob_t));
762
763 gl.pattern = file.data;
764 gl.log = cf->log;
765 gl.test = 1;
766
767 if (ngx_open_glob(&gl) != NGX_OK) {
768     ngx_conf_log_error(NGX_LOG_EMERG, cf, ngx_errno,
769         ngx_open_glob_n "%s" failed", file.data);
770     return NGX_CONF_ERROR;
771 }
772
773 rv = NGX_CONF_OK;
774
775 for ( ;; ) {
776     n = ngx_read_glob(&gl, &name);
777
778     if (n != NGX_OK) {
779         break;
780     }
781
782     file.len = name.len++;
783     file.data = ngx_pstrdup(cf->pool, &name);
784     if (file.data == NULL) {
785         return NGX_CONF_ERROR;
786     }
787
788     ngx_log_debug1(NGX_LOG_DEBUG_CORE, cf->log, 0, "include %s", file.data);
789
790     rv = ngx_conf_parse(cf, &file);
791
792     if (rv != NGX_CONF_OK) {
793         break;
794     }
795 }
796
797 ngx_close_glob(&gl);
798
799 return rv;
800 }
801
802
803 ngx_int_t
804 ngx_conf_full_name(ngx_cycle_t *cycle, ngx_str_t *name, ngx_uint_t conf_prefix)
805 {
806     ngx_str_t *prefix;
807
808     prefix = conf_prefix ? &cycle->conf_prefix : &cycle->prefix;
809
810     return ngx_get_full_name(cycle->pool, prefix, name);
811 }
812
813
814 ngx_open_file_t *
815 ngx_conf_open_file(ngx_cycle_t *cycle, ngx_str_t *name)
816 {
817     ngx_str_t full;
818     ngx_uint_t i;
819     ngx_list_part_t *part;
820     ngx_open_file_t *file;
821
822     #if (NGX_SUPPRESS_WARN)
823     ngx_str_null(&full);
824     #endif
825
826     if (name->len) {
827         full = *name;
828
829         if (ngx_conf_full_name(cycle, &full, 0) != NGX_OK) {
830             return NULL;
831         }
832
833         part = &cycle->open_files.part;

```

```

834     file = part->elts;
835
836     for (i = 0; /* void */ ; i++) {
837
838         if (i >= part->nelts) {
839             if (part->next == NULL) {
840                 break;
841             }
842             part = part->next;
843             file = part->elts;
844             i = 0;
845         }
846
847         if (full.len != file[i].name.len) {
848             continue;
849         }
850
851         if (ngx_strcmp(full.data, file[i].name.data) == 0) {
852             return &file[i];
853         }
854     }
855 }
856
857 file = ngx_list_push(&cycle->open_files);
858 if (file == NULL) {
859     return NULL;
860 }
861
862 if (name->len) {
863     file->fd = NGX_INVALID_FILE;
864     file->name = full;
865 }
866 else {
867     file->fd = ngx_stderr;
868     file->name = *name;
869 }
870
871 file->flush = NULL;
872 file->data = NULL;
873
874 return file;
875 }
876
877
878 static void
879 ngx_conf_flush_files(ngx_cycle_t *cycle)
880 {
881     ngx_uint_t    i;
882     ngx_list_part_t *part;
883     ngx_open_file_t *file;
884
885     ngx_log_debug0(NGX_LOG_DEBUG_CORE, cycle->log, 0, "flush files");
886
887     part = &cycle->open_files.part;
888     file = part->elts;
889
890     for (i = 0; /* void */ ; i++) {
891
892         if (i >= part->nelts) {
893             if (part->next == NULL) {
894                 break;
895             }
896             part = part->next;
897             file = part->elts;
898             i = 0;
899         }
900
901         if (file[i].flush) {
902             file[i].flush(&file[i], cycle->log);
903         }
904     }
905 }
906
907
908 void ngx_cdecl
909 ngx_conf_log_error(ngx_uint_t level, ngx_conf_t *cf, ngx_err_t err,

```

```

910     const char *fmt, ...)
911 {
912     u_char  errstr[NGX_MAX_CONF_ERRSTR], *p, *last;
913     va_list args;
914
915     last = errstr + NGX_MAX_CONF_ERRSTR;
916
917     va_start(args, fmt);
918     p = ngx_vslprintf(errstr, last, fmt, args);
919     va_end(args);
920
921     if (err) {
922         p = ngx_log_errno(p, last, err);
923     }
924
925     if (cf->conf_file == NULL) {
926         ngx_log_error(level, cf->log, 0, "%s", p - errstr, errstr);
927         return;
928     }
929
930     if (cf->conf_file->file.fd == NGX_INVALID_FILE) {
931         ngx_log_error(level, cf->log, 0, "%s in command line",
932             p - errstr, errstr);
933         return;
934     }
935
936     ngx_log_error(level, cf->log, 0, "%s in %s:%ui",
937         p - errstr, errstr,
938         cf->conf_file->file.name.data, cf->conf_file->line);
939 }
940
941 char *
942 ngx_conf_set_flag_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
943 {
944     char *p = conf;
945
946     ngx_str_t    *value;
947     ngx_flag_t   *fp;
948     ngx_conf_post_t *post;
949
950     fp = (ngx_flag_t *) (p + cmd->offset);
951
952     if (*fp != NGX_CONF_UNSET) {
953         return "is duplicate";
954     }
955
956     value = cf->args->elts;
957
958     if (ngx_strcasecmp(value[1].data, (u_char *) "on") == 0) {
959         *fp = 1;
960     } else if (ngx_strcasecmp(value[1].data, (u_char *) "off") == 0) {
961         *fp = 0;
962     } else {
963         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
964             "invalid value \"%s\" in \"%s\" directive, "
965             "it must be \"on\" or \"off\"",
966             value[1].data, cmd->name.data);
967         return NGX_CONF_ERROR;
968     }
969
970     if (cmd->post) {
971         post = cmd->post;
972         return post->post_handler(cf, post, fp);
973     }
974
975     return NGX_CONF_OK;
976 }
977
978 char *
979 ngx_conf_set_str_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
980 {
981     char *p = conf;

```

```

986     ngx_str_t      *field, *value;
987     ngx_conf_post_t *post;
988
989     field = (ngx_str_t *) (p + cmd->offset);
990
991     if (field->data) {
992         return "is duplicate";
993     }
994
995     value = cf->args->elts;
996
997     *field = value[1];
998
999     if (cmd->post) {
1000         post = cmd->post;
1001         return post->post_handler(cf, post, field);
1002     }
1003
1004     return NGX_CONF_OK;
1005 }
1006
1007
1008
1009 char *
1010 ngx_conf_set_str_array_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
1011 {
1012     char *p = conf;
1013
1014     ngx_str_t      *value, *s;
1015     ngx_array_t     **a;
1016     ngx_conf_post_t *post;
1017
1018     a = (ngx_array_t **) (p + cmd->offset);
1019
1020     if (*a == NGX_CONF_UNSET_PTR) {
1021         *a = ngx_array_create(cf->pool, 4, sizeof(ngx_str_t));
1022         if (*a == NULL) {
1023             return NGX_CONF_ERROR;
1024         }
1025     }
1026
1027     s = ngx_array_push(*a);
1028     if (s == NULL) {
1029         return NGX_CONF_ERROR;
1030     }
1031
1032     value = cf->args->elts;
1033
1034     *s = value[1];
1035
1036     if (cmd->post) {
1037         post = cmd->post;
1038         return post->post_handler(cf, post, s);
1039     }
1040
1041     return NGX_CONF_OK;
1042 }
1043
1044
1045 char *
1046 ngx_conf_set_keyval_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
1047 {
1048     char *p = conf;
1049
1050     ngx_str_t      *value;
1051     ngx_array_t     **a;
1052     ngx_keyval_t    *kv;
1053     ngx_conf_post_t *post;
1054
1055     a = (ngx_array_t **) (p + cmd->offset);
1056
1057     if (*a == NULL) {
1058         *a = ngx_array_create(cf->pool, 4, sizeof(ngx_keyval_t));
1059         if (*a == NULL) {
1060             return NGX_CONF_ERROR;
1061         }

```

```

1062     }
1063
1064     kv = ngx_array_push(*a);
1065     if (kv == NULL) {
1066         return NGX_CONF_ERROR;
1067     }
1068
1069     value = cf->args->elts;
1070
1071     kv->key = value[1];
1072     kv->value = value[2];
1073
1074     if (cmd->post) {
1075         post = cmd->post;
1076         return post->post_handler(cf, post, kv);
1077     }
1078
1079     return NGX_CONF_OK;
1080 }
1081
1082
1083 char *
1084 ngx_conf_set_num_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
1085 {
1086     char *p = conf;
1087
1088     ngx_int_t *np;
1089     ngx_str_t *value;
1090     ngx_conf_post_t *post;
1091
1092
1093     np = (ngx_int_t *) (p + cmd->offset);
1094
1095     if (*np != NGX_CONF_UNSET) {
1096         return "is duplicate";
1097     }
1098
1099     value = cf->args->elts;
1100     *np = ngx_atoi(value[1].data, value[1].len);
1101     if (*np == NGX_ERROR) {
1102         return "invalid number";
1103     }
1104
1105     if (cmd->post) {
1106         post = cmd->post;
1107         return post->post_handler(cf, post, np);
1108     }
1109
1110     return NGX_CONF_OK;
1111 }
1112
1113
1114 char *
1115 ngx_conf_set_size_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
1116 {
1117     char *p = conf;
1118
1119     size_t *sp;
1120     ngx_str_t *value;
1121     ngx_conf_post_t *post;
1122
1123
1124     sp = (size_t *) (p + cmd->offset);
1125     if (*sp != NGX_CONF_UNSET_SIZE) {
1126         return "is duplicate";
1127     }
1128
1129     value = cf->args->elts;
1130
1131     *sp = ngx_parse_size(&value[1]);
1132     if (*sp == (size_t) NGX_ERROR) {
1133         return "invalid value";
1134     }
1135
1136     if (cmd->post) {
1137         post = cmd->post;

```



```

1138     return post->post_handler(cf, post, sp);
1139 }
1140
1141 return NGX_CONF_OK;
1142 }
1143
1144
1145 char *
1146 ngx_conf_set_off_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
1147 {
1148     char *p = conf;
1149
1150     off_t      *op;
1151     ngx_str_t   *value;
1152     ngx_conf_post_t *post;
1153
1154
1155     op = (off_t *) (p + cmd->offset);
1156     if (*op != NGX_CONF_UNSET) {
1157         return "is duplicate";
1158     }
1159
1160     value = cf->args->elts;
1161
1162     *op = ngx_parse_offset(&value[1]);
1163     if (*op == (off_t) NGX_ERROR) {
1164         return "invalid value";
1165     }
1166
1167     if (cmd->post) {
1168         post = cmd->post;
1169         return post->post_handler(cf, post, op);
1170     }
1171
1172     return NGX_CONF_OK;
1173 }
1174
1175
1176 char *
1177 ngx_conf_set_msec_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
1178 {
1179     char *p = conf;
1180
1181     ngx_msec_t *msp;
1182     ngx_str_t   *value;
1183     ngx_conf_post_t *post;
1184
1185
1186     msp = (ngx_msec_t *) (p + cmd->offset);
1187     if (*msp != NGX_CONF_UNSET_MSEC) {
1188         return "is duplicate";
1189     }
1190
1191     value = cf->args->elts;
1192
1193     *msp = ngx_parse_time(&value[1], 0);
1194     if (*msp == (ngx_msec_t) NGX_ERROR) {
1195         return "invalid value";
1196     }
1197
1198     if (cmd->post) {
1199         post = cmd->post;
1200         return post->post_handler(cf, post, msp);
1201     }
1202
1203     return NGX_CONF_OK;
1204 }
1205
1206
1207 char *
1208 ngx_conf_set_sec_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
1209 {
1210     char *p = conf;
1211
1212     time_t      *sp;
1213     ngx_str_t   *value;

```

```

1214     ngx_conf_post_t *post;
1215
1216
1217     sp = (time_t *) (p + cmd->offset);
1218     if (*sp != NGX_CONF_UNSET) {
1219         return "is duplicate";
1220     }
1221
1222     value = cf->args->elts;
1223
1224     *sp = ngx_parse_time(&value[1], 1);
1225     if (*sp == (time_t) NGX_ERROR) {
1226         return "invalid value";
1227     }
1228
1229     if (cmd->post) {
1230         post = cmd->post;
1231         return post->post_handler(cf, post, sp);
1232     }
1233
1234     return NGX_CONF_OK;
1235 }
1236
1237
1238 char *
1239 ngx_conf_set_bufs_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
1240 {
1241     char *p = conf;
1242
1243     ngx_str_t *value;
1244     ngx_bufs_t *bufs;
1245
1246
1247     bufs = (ngx_bufs_t *) (p + cmd->offset);
1248     if (bufs->num) {
1249         return "is duplicate";
1250     }
1251
1252     value = cf->args->elts;
1253
1254     bufs->num = ngx_atoi(value[1].data, value[1].len);
1255     if (bufs->num == NGX_ERROR || bufs->num == 0) {
1256         return "invalid value";
1257     }
1258
1259     bufs->size = ngx_parse_size(&value[2]);
1260     if (bufs->size == (size_t) NGX_ERROR || bufs->size == 0) {
1261         return "invalid value";
1262     }
1263
1264     return NGX_CONF_OK;
1265 }
1266
1267
1268 char *
1269 ngx_conf_set_enum_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
1270 {
1271     char *p = conf;
1272
1273     ngx_uint_t *np, i;
1274     ngx_str_t *value;
1275     ngx_conf_enum_t *e;
1276
1277     np = (ngx_uint_t *) (p + cmd->offset);
1278
1279     if (*np != NGX_CONF_UNSET_UINT) {
1280         return "is duplicate";
1281     }
1282
1283     value = cf->args->elts;
1284     e = cmd->post;
1285
1286     for (i = 0; e[i].name.len != 0; i++) {
1287         if (e[i].name.len != value[1].len
1288             || ngx_strcasecmp(e[i].name.data, value[1].data) != 0)
1289             {

```

```

1290     continue;
1291 }
1292
1293 *np = e[i].value;
1294
1295 return NGX_CONF_OK;
1296 }
1297
1298 ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
1299     "invalid value \"%s\"", value[1].data);
1300
1301 return NGX_CONF_ERROR;
1302 }
1303
1304
1305 char *
1306 ngx_conf_set_bitmask_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
1307 {
1308     char *p = conf;
1309
1310     ngx_uint_t *np, i, m;
1311     ngx_str_t *value;
1312     ngx_conf_bitmask_t *mask;
1313
1314
1315     np = (ngx_uint_t *) (p + cmd->offset);
1316     value = cf->args->elts;
1317     mask = cmd->post;
1318
1319     for (i = 1; i < cf->args->nelts; i++) {
1320         for (m = 0; mask[m].name.len != 0; m++) {
1321
1322             if (mask[m].name.len != value[i].len
1323                 || ngx_strcasecmp(mask[m].name.data, value[i].data) != 0)
1324             {
1325                 continue;
1326             }
1327
1328             if (*np & mask[m].mask) {
1329                 ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
1330                     "duplicate value \"%s\"", value[i].data);
1331             } else {
1332                 *np |= mask[m].mask;
1333             }
1334
1335             break;
1336         }
1337     }
1338
1339     if (mask[m].name.len == 0) {
1340         ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
1341             "invalid value \"%s\"", value[i].data);
1342     }
1343
1344     return NGX_CONF_ERROR;
1345 }
1346
1347 return NGX_CONF_OK;
1348 }
1349
1350
1351 #if 0
1352
1353 char *
1354 ngx_conf_unsupported(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
1355 {
1356     return "unsupported on this platform";
1357 }
1358
1359 #endif
1360
1361
1362 char *
1363 ngx_conf_deprecated(ngx_conf_t *cf, void *post, void *data)
1364 {
1365     ngx_conf_deprecated_t *d = post;

```

```

1366     ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
1367     "the \"%s\" directive is deprecated, "
1368     "use the \"%s\" directive instead",
1369     d->old_name, d->new_name);
1370
1371
1372     return NGX_CONF_OK;
1373 }
1374
1375
1376 char *
1377 ngx_conf_check_num_bounds(ngx_conf_t *cf, void *post, void *data)
1378 {
1379     ngx_conf_num_bounds_t *bounds = post;
1380     ngx_int_t *np = data;
1381
1382     if (bounds->high == -1) {
1383         if (*np >= bounds->low) {
1384             return NGX_CONF_OK;
1385         }
1386
1387         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1388             "value must be equal to or greater than %i",
1389             bounds->low);
1390
1391         return NGX_CONF_ERROR;
1392     }
1393
1394     if (*np >= bounds->low && *np <= bounds->high) {
1395         return NGX_CONF_OK;
1396     }
1397
1398     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1399         "value must be between %i and %i",
1400         bounds->low, bounds->high);
1401
1402     return NGX_CONF_ERROR;
1403 }

```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_buf.h - nginx-1.7.10

### Data types defined

- [ngx\\_buf\\_s](#)
- [ngx\\_buf\\_t](#)
- [ngx\\_buf\\_tag\\_t](#)
- [ngx\\_bufs\\_t](#)
- [ngx\\_chain\\_s](#)
- [ngx\\_chain\\_writer\\_ctx\\_t](#)
- [ngx\\_output\\_chain\\_aio\\_pt](#)
- [ngx\\_output\\_chain\\_ctx\\_s](#)
- [ngx\\_output\\_chain\\_ctx\\_t](#)
- [ngx\\_output\\_chain\\_filter\\_pt](#)

### Macros defined

- [NGX\\_CHAIN\\_ERROR](#)
- [\\_NGX\\_BUF\\_H\\_INCLUDED\\_](#)
- [ngx\\_alloc\\_buf](#)
- [ngx\\_buf\\_in\\_memory](#)
- [ngx\\_buf\\_in\\_memory\\_only](#)
- [ngx\\_buf\\_size](#)
- [ngx\\_buf\\_special](#)
- [ngx\\_buf\\_sync\\_only](#)
- [ngx\\_calloc\\_buf](#)
- [ngx\\_free\\_chain](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_BUF\_H\_INCLUDED\_
9 #define \_NGX\_BUF\_H\_INCLUDED\_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 typedef void * ngx\_buf\_tag\_t;
```

```

17
18 typedef struct ngx\_buf\_s ngx_buf_t;
19
20 struct ngx_buf_s {
21     u_char      *pos;
22     u_char      *last;
23     off_t       file_pos;
24     off_t       file_last;
25
26     u_char      *start;      /* start of buffer */
27     u_char      *end;        /* end of buffer */
28     ngx\_buf\_tag\_t tag;
29     ngx\_file\_t *file;
30     ngx\_buf\_t  *shadow;
31
32
33     /* the buf's content could be changed */
34     unsigned    temporary:1;
35
36     /*
37      * the buf's content is in a memory cache or in a read only memory
38      * and must not be changed
39      */
40     unsigned    memory:1;
41
42     /* the buf's content is mmap()ed and must not be changed */
43     unsigned    mmap:1;
44
45     unsigned    recycled:1;
46     unsigned    in_file:1;
47     unsigned    flush:1;
48     unsigned    sync:1;
49     unsigned    last_buf:1;
50     unsigned    last_in_chain:1;
51
52     unsigned    last_shadow:1;
53     unsigned    temp_file:1;
54
55     /* STUB */ int    num;
56 };
57
58
59 struct ngx_chain_s {
60     ngx\_buf\_t    *buf;
61     ngx\_chain\_t *next;
62 };
63
64
65 typedef struct {
66     ngx\_int\_t    num;
67     size_t       size;
68 } ngx_bufs_t;
69
70
71 typedef struct ngx\_output\_chain\_ctx\_s ngx_output_chain_ctx_t;
72
73 typedef ngx\_int\_t (*ngx_output_chain_filter_pt)(void *ctx, ngx\_chain\_t *in);
74
75 #if (NGX_HAVE_FILE_AIO)
76 typedef void (*ngx_output_chain_aio_pt)(ngx\_output\_chain\_ctx\_t *ctx,
77     ngx\_file\_t *file);
78 #endif
79
80 struct ngx_output_chain_ctx_s {
81     ngx\_buf\_t    *buf;
82     ngx\_chain\_t *in;
83     ngx\_chain\_t *free;
84     ngx\_chain\_t *busy;
85
86     unsigned    sendfile:1;
87     unsigned    directio:1;
88 #if (NGX_HAVE_ALIGNED_DIRECTIO)
89     unsigned    unaligned:1;
90 #endif
91     unsigned    need_in_memory:1;
92     unsigned    need_in_temp:1;

```

```

93 #if (NGX_HAVE_FILE_AIO)
94     unsigned                aio:1;
95
96     ngx_output_chain aio_pt    aio_handler;
97 #endif
98
99     off_t                    alignment;
100
101     ngx_pool_t                *pool;
102     ngx_int_t                 allocated;
103     ngx_bufs_t                bufs;
104     ngx_buf_tag_t            tag;
105
106     ngx_output_chain_filter_pt output_filter;
107     void                      *filter_ctx;
108 };
109
110
111 typedef struct {
112     ngx_chain_t              *out;
113     ngx_chain_t              **last;
114     ngx_connection_t        *connection;
115     ngx_pool_t               *pool;
116     off_t                    limit;
117 } ngx_chain_writer_ctx_t;
118
119
120 #define NGX_CHAIN_ERROR      (ngx_chain_t *) NGX_ERROR
121
122
123 #define ngx_buf_in_memory(b)    (b->temporary || b->memory || b->mmap)
124 #define ngx_buf_in_memory_only(b) (ngx_buf_in_memory(b) && !b->in_file)
125
126 #define ngx_buf_special(b)      \
127     ((b->flush || b->last_buf || b->sync) \
128     && !ngx_buf_in_memory(b) && !b->in_file) \
129
130 #define ngx_buf_sync_only(b)    \
131     (b->sync \
132     && !ngx_buf_in_memory(b) && !b->in_file && !b->flush && !b->last_buf) \
133
134 #define ngx_buf_size(b)        \
135     (ngx_buf_in_memory(b) ? (off_t) (b->last - b->pos): \
136     (b->file_last - b->file_pos)) \
137
138 ngx_buf_t *ngx_create_temp_buf(ngx_pool_t *pool, size_t size);
139 ngx_chain_t *ngx_create_chain_of_bufs(ngx_pool_t *pool, ngx_bufs_t *bufs);
140
141
142 #define ngx_alloc_buf(pool) ngx_palloc(pool, sizeof(ngx_buf_t))
143 #define ngx_calloc_buf(pool) ngx_pcalloc(pool, sizeof(ngx_buf_t))
144
145 ngx_chain_t *ngx_alloc_chain_link(ngx_pool_t *pool);
146 #define ngx_free_chain(pool, cl) \
147     cl->next = pool->chain; \
148     pool->chain = cl \
149
150
151
152 ngx_int_t ngx_output_chain(ngx_output_chain_ctx_t *ctx, ngx_chain_t *in);
153 ngx_int_t ngx_chain_writer(void *ctx, ngx_chain_t *in);
154
155 ngx_int_t ngx_chain_add_copy(ngx_pool_t *pool, ngx_chain_t **chain,
156     ngx_chain_t *in);
157 ngx_chain_t *ngx_chain_get_free_buf(ngx_pool_t *p, ngx_chain_t **free);
158 void ngx_chain_update_chains(ngx_pool_t *p, ngx_chain_t **free,
159     ngx_chain_t **busy, ngx_chain_t **out, ngx_buf_tag_t tag);
160
161 off_t ngx_chain_coalesce_file(ngx_chain_t **in, off_t limit);
162
163 ngx_chain_t *ngx_chain_update_sent(ngx_chain_t *in, off_t sent);
164
165 #endif /* NGX_BUF_H_INCLUDED */

```

## src/core/nginx\_buf.c - nginx-1.7.10

### Functions defined

- [ngx\\_alloc\\_chain\\_link](#)
- [ngx\\_chain\\_add\\_copy](#)
- [ngx\\_chain\\_coalesce\\_file](#)
- [ngx\\_chain\\_get\\_free\\_buf](#)
- [ngx\\_chain\\_update\\_chains](#)
- [ngx\\_chain\\_update\\_sent](#)
- [ngx\\_create\\_chain\\_of\\_bufs](#)
- [ngx\\_create\\_temp\\_buf](#)

### Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10
11
12  ngx_buf_t *
13  ngx_create_temp_buf(ngx_pool_t *pool, size_t size)
14  {
15      ngx_buf_t *b;
16
17      b = ngx_calloc_buf(pool);
18      if (b == NULL) {
19          return NULL;
20      }
21
22      b->start = ngx_palloc(pool, size);
23      if (b->start == NULL) {
24          return NULL;
25      }
26
27      /*
28       * set by ngx_calloc_buf():
29       *
30       *     b->file_pos = 0;
31       *     b->file_last = 0;
32       *     b->file = NULL;
33       *     b->shadow = NULL;
34       *     b->tag = 0;
35       *     and flags
36       */
37
38      b->pos = b->start;
39      b->last = b->start;
40      b->end = b->last + size;
41      b->temporary = 1;
42
43      return b;
44  }
45
46
47  ngx_chain_t *
```



```

48 ngx_alloc_chain_link(ngx_pool_t *pool)
49 {
50     ngx_chain_t *cl;
51
52     cl = pool->chain;
53
54     if (cl) {
55         pool->chain = cl->next;
56         return cl;
57     }
58
59     cl = ngx_palloc(pool, sizeof(ngx_chain_t));
60     if (cl == NULL) {
61         return NULL;
62     }
63
64     return cl;
65 }
66
67
68 ngx_chain_t *
69 ngx_create_chain_of_bufs(ngx_pool_t *pool, ngx_bufs_t *bufs)
70 {
71     u_char *p;
72     ngx_int_t i;
73     ngx_buf_t *b;
74     ngx_chain_t *chain, *cl, **ll;
75
76     p = ngx_palloc(pool, bufs->num * bufs->size);
77     if (p == NULL) {
78         return NULL;
79     }
80
81     ll = &chain;
82
83     for (i = 0; i < bufs->num; i++) {
84
85         b = ngx_calloc_buf(pool);
86         if (b == NULL) {
87             return NULL;
88         }
89
90         /*
91          * set by ngx_calloc_buf():
92          *
93          *     b->file_pos = 0;
94          *     b->file_last = 0;
95          *     b->file = NULL;
96          *     b->shadow = NULL;
97          *     b->tag = 0;
98          *     and flags
99          *
100          */
101
102         b->pos = p;
103         b->last = p;
104         b->temporary = 1;
105
106         b->start = p;
107         p += bufs->size;
108         b->end = p;
109
110         cl = ngx_alloc_chain_link(pool);
111         if (cl == NULL) {
112             return NULL;
113         }
114
115         cl->buf = b;
116         *ll = cl;
117         ll = &cl->next;
118     }
119
120     *ll = NULL;
121
122     return chain;
123 }

```

```

124
125
126 ngx_int_t
127 ngx_chain_add_copy(ngx_pool_t *pool, ngx_chain_t **chain, ngx_chain_t *in)
128 {
129     ngx_chain_t *cl, **ll;
130
131     ll = chain;
132
133     for (cl = *chain; cl; cl = cl->next) {
134         ll = &cl->next;
135     }
136
137     while (in) {
138         cl = ngx_alloc_chain_link(pool);
139         if (cl == NULL) {
140             return NGX_ERROR;
141         }
142
143         cl->buf = in->buf;
144         *ll = cl;
145         ll = &cl->next;
146         in = in->next;
147     }
148
149     *ll = NULL;
150
151     return NGX_OK;
152 }
153
154
155 ngx_chain_t *
156 ngx_chain_get_free_buf(ngx_pool_t *p, ngx_chain_t **free)
157 {
158     ngx_chain_t *cl;
159
160     if (*free) {
161         cl = *free;
162         *free = cl->next;
163         cl->next = NULL;
164         return cl;
165     }
166
167     cl = ngx_alloc_chain_link(p);
168     if (cl == NULL) {
169         return NULL;
170     }
171
172     cl->buf = ngx_calloc_buf(p);
173     if (cl->buf == NULL) {
174         return NULL;
175     }
176
177     cl->next = NULL;
178
179     return cl;
180 }
181
182
183 void
184 ngx_chain_update_chains(ngx_pool_t *p, ngx_chain_t **free, ngx_chain_t **busy,
185 ngx_chain_t **out, ngx_buf_tag_t tag)
186 {
187     ngx_chain_t *cl;
188
189     if (*busy == NULL) {
190         *busy = *out;
191     } else {
192         for (cl = *busy; cl->next; cl = cl->next) { /* void */ }
193
194         cl->next = *out;
195     }
196
197     *out = NULL;
198
199

```

```

200 while (*busy) {
201     cl = *busy;
202
203     if (ngx\_buf\_size(cl->buf) != 0) {
204         break;
205     }
206
207     if (cl->buf->tag != tag) {
208         *busy = cl->next;
209         ngx\_free\_chain(p, cl);
210         continue;
211     }
212
213     cl->buf->pos = cl->buf->start;
214     cl->buf->last = cl->buf->start;
215
216     *busy = cl->next;
217     cl->next = *free;
218     *free = cl;
219 }
220 }
221
222
223 off\_t
224 ngx\_chain\_coalesce\_file(ngx\_chain\_t **in, off\_t limit)
225 {
226     off\_t         total, size, aligned, fprev;
227     ngx\_fd\_t     fd;
228     ngx\_chain\_t *cl;
229
230     total = 0;
231
232     cl = *in;
233     fd = cl->buf->file->fd;
234
235     do {
236         size = cl->buf->file_last - cl->buf->file_pos;
237
238         if (size > limit - total) {
239             size = limit - total;
240
241             aligned = (cl->buf->file_pos + size + ngx\_pagesize - 1)
242                 & ~((off\_t) ngx\_pagesize - 1);
243
244             if (aligned <= cl->buf->file_last) {
245                 size = aligned - cl->buf->file_pos;
246             }
247         }
248
249         total += size;
250         fprev = cl->buf->file_pos + size;
251         cl = cl->next;
252
253     } while (cl
254             && cl->buf->in_file
255             && total < limit
256             && fd == cl->buf->file->fd
257             && fprev == cl->buf->file_pos);
258
259     *in = cl;
260
261     return total;
262 }
263
264
265 ngx\_chain\_t *
266 ngx\_chain\_update\_sent(ngx\_chain\_t *in, off\_t sent)
267 {
268     off\_t size;
269
270     for ( /* void */ ; in; in = in->next) {
271
272         if (ngx\_buf\_special(in->buf)) {
273             continue;
274         }
275

```

```
276     if (sent == 0) {
277         break;
278     }
279
280     size = ngx_buf_size(in->buf);
281
282     if (sent >= size) {
283         sent -= size;
284
285         if (ngx_buf_in_memory(in->buf)) {
286             in->buf->pos = in->buf->last;
287         }
288
289         if (in->buf->in_file) {
290             in->buf->file_pos = in->buf->file_last;
291         }
292
293         continue;
294     }
295
296     if (ngx_buf_in_memory(in->buf)) {
297         in->buf->pos += (size_t) sent;
298     }
299
300     if (in->buf->in_file) {
301         in->buf->file_pos += sent;
302     }
303
304     break;
305 }
306
307 return in;
308 }
```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_output\_chain.c - nginx-1.7.10

### Functions defined

- [ngx\\_chain\\_writer](#)
- [ngx\\_output\\_chain](#)
- [ngx\\_output\\_chain\\_add\\_copy](#)
- [ngx\\_output\\_chain\\_align\\_file\\_buf](#)
- [ngx\\_output\\_chain\\_as\\_is](#)
- [ngx\\_output\\_chain\\_copy\\_buf](#)
- [ngx\\_output\\_chain\\_get\\_buf](#)

### Macros defined

- [NGX\\_NONE](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 #if 0
14 #define NGX\_SENDFILE\_LIMIT 4096
15 #endif
16
17 /*
18  * When DIRECTIO is enabled FreeBSD, Solaris, and MacOSX read directly
19  * to an application memory from a device if parameters are aligned
20  * to device sector boundary (512 bytes). They fallback to usual read
21  * operation if the parameters are not aligned.
22  * Linux allows DIRECTIO only if the parameters are aligned to a filesystem
23  * sector boundary, otherwise it returns EINVAL. The sector size is
24  * usually 512 bytes, however, on XFS it may be 4096 bytes.
25  */
26
27 #define NGX\_NONE 1
28
29
30 static ngx\_inline ngx\_int\_t
31     ngx\_output\_chain\_as\_is(ngx\_output\_chain\_ctx\_t *ctx, ngx\_buf\_t *buf);
32 static ngx\_int\_t ngx\_output\_chain\_add\_copy(ngx\_pool\_t *pool,
33     ngx\_chain\_t **chain, ngx\_chain\_t *in);
34 static ngx\_int\_t ngx\_output\_chain\_align\_file\_buf(ngx\_output\_chain\_ctx\_t *ctx,
35     off_t bsize);
36 static ngx\_int\_t ngx\_output\_chain\_get\_buf(ngx\_output\_chain\_ctx\_t *ctx,
37     off_t bsize);
38 static ngx\_int\_t ngx\_output\_chain\_copy\_buf(ngx\_output\_chain\_ctx\_t *ctx);
39
40
41 ngx\_int\_t
42 ngx\_output\_chain(ngx\_output\_chain\_ctx\_t *ctx, ngx\_chain\_t *in)
43 {
```

```

44     off_t         bsize;
45     ngx_int_t     rc, last;
46     ngx_chain_t  *cl, *out, **last_out;
47
48     if (ctx->in == NULL && ctx->busy == NULL
49 #if (NGX_HAVE_FILE_AIO)
50         && !ctx->aio
51 #endif
52     )
53     {
54         /*
55          * the short path for the case when the ctx->in and ctx->busy chains
56          * are empty, the incoming chain is empty too or has the single buf
57          * that does not require the copy
58          */
59
60         if (in == NULL) {
61             return ctx->output_filter(ctx->filter_ctx, in);
62         }
63
64         if (in->next == NULL
65 #if (NGX_SENDFILE_LIMIT)
66             && !(in->buf->in_file && in->buf->file_last > NGX_SENDFILE_LIMIT)
67 #endif
68             && ngx_output_chain_as_is(ctx, in->buf))
69         {
70             return ctx->output_filter(ctx->filter_ctx, in);
71         }
72     }
73
74     /* add the incoming buf to the chain ctx->in */
75
76     if (in) {
77         if (ngx_output_chain_add_copy(ctx->pool, &ctx->in, in) == NGX_ERROR) {
78             return NGX_ERROR;
79         }
80     }
81
82     out = NULL;
83     last_out = &out;
84     last = NGX_NONE;
85
86     for ( ;; ) {
87
88 #if (NGX_HAVE_FILE_AIO)
89         if (ctx->aio) {
90             return NGX_AGAIN;
91         }
92 #endif
93
94         while (ctx->in) {
95
96             /*
97              * cycle while there are the ctx->in bufs
98              * and there are the free output bufs to copy in
99              */
100
101             bsize = ngx_buf_size(ctx->in->buf);
102
103             if (bsize == 0 && !ngx_buf_special(ctx->in->buf)) {
104
105                 ngx_log_error(NGX_LOG_ALERT, ctx->pool->log, 0,
106                     "zero size buf in output "
107                     "t:%d r:%d f:%d %p %p-%p %p %O-%O",
108                     ctx->in->buf->temporary,
109                     ctx->in->buf->recycled,
110                     ctx->in->buf->in_file,
111                     ctx->in->buf->start,
112                     ctx->in->buf->pos,
113                     ctx->in->buf->last,
114                     ctx->in->buf->file,
115                     ctx->in->buf->file_pos,
116                     ctx->in->buf->file_last);
117
118                 ngx_debug_point();
119

```

```

120     ctx->in = ctx->in->next;
121
122     continue;
123 }
124
125 if (ngx\_output\_chain\_as\_is(ctx, ctx->in->buf)) {
126
127     /* move the chain link to the output chain */
128
129     cl = ctx->in;
130     ctx->in = cl->next;
131
132     *last_out = cl;
133     last_out = &cl->next;
134     cl->next = NULL;
135
136     continue;
137 }
138
139 if (ctx->buf == NULL) {
140
141     rc = ngx\_output\_chain\_align\_file\_buf(ctx, bsize);
142
143     if (rc == NGX\_ERROR) {
144         return NGX\_ERROR;
145     }
146
147     if (rc != NGX\_OK) {
148
149         if (ctx->free) {
150
151             /* get the free buf */
152
153             cl = ctx->free;
154             ctx->buf = cl->buf;
155             ctx->free = cl->next;
156
157             ngx\_free\_chain(ctx->pool, cl);
158
159             } else if (out || ctx->allocated == ctx->bufs.num) {
160
161                 break;
162
163             } else if (ngx\_output\_chain\_get\_buf(ctx, bsize) != NGX\_OK) {
164                 return NGX\_ERROR;
165             }
166         }
167     }
168
169     rc = ngx\_output\_chain\_copy\_buf(ctx);
170
171     if (rc == NGX\_ERROR) {
172         return rc;
173     }
174
175     if (rc == NGX\_AGAIN) {
176         if (out) {
177             break;
178         }
179
180         return rc;
181     }
182
183     /* delete the completed buf from the ctx->in chain */
184
185     if (ngx\_buf\_size(ctx->in->buf) == 0) {
186         ctx->in = ctx->in->next;
187     }
188
189     cl = ngx\_alloc\_chain\_link(ctx->pool);
190     if (cl == NULL) {
191         return NGX\_ERROR;
192     }
193
194     cl->buf = ctx->buf;
195     cl->next = NULL;

```

```

196     *last_out = cl;
197     last_out = &cl->next;
198     ctx->buf = NULL;
199 }
200
201 if (out == NULL && last != NGX\_NONE) {
202
203     if (ctx->in) {
204         return NGX\_AGAIN;
205     }
206
207     return last;
208 }
209
210 last = ctx->output_filter(ctx->filter_ctx, out);
211
212 if (last == NGX\_ERROR || last == NGX\_DONE) {
213     return last;
214 }
215
216 ngx\_chain\_update\_chains(ctx->pool, &ctx->free, &ctx->busy, &out,
217                          ctx->tag);
218 last_out = &out;
219 }
220 }
221
222
223 static ngx\_inline ngx\_int\_t
224 ngx\_output\_chain\_as\_is(ngx\_output\_chain\_ctx\_t *ctx, ngx\_buf\_t *buf)
225 {
226     ngx\_uint\_t sendfile;
227
228     if (ngx\_buf\_special(buf)) {
229         return 1;
230     }
231
232     if (buf->in_file && buf->file->directio) {
233         return 0;
234     }
235
236     sendfile = ctx->sendfile;
237
238     #if \(NGX\_SENDFILE\_LIMIT\)
239
240     if (buf->in_file && buf->file_pos >= NGX\_SENDFILE\_LIMIT) {
241         sendfile = 0;
242     }
243
244     #endif
245
246     if (!sendfile) {
247
248         if (!ngx\_buf\_in\_memory(buf)) {
249             return 0;
250         }
251
252         buf->in_file = 0;
253     }
254
255     if (ctx->need_in_memory && !ngx\_buf\_in\_memory(buf)) {
256         return 0;
257     }
258
259     if (ctx->need_in_temp && (buf->memory || buf->mmap)) {
260         return 0;
261     }
262
263     return 1;
264 }
265
266
267 static ngx\_int\_t
268 ngx\_output\_chain\_add\_copy(ngx\_pool\_t *pool, ngx\_chain\_t **chain,
269 ngx\_chain\_t *in)
270 {
271     ngx\_chain\_t *cl, **ll;

```



```

272 #if (NGX_SENDFILE_LIMIT)
273     ngx_buf_t     *b, *buf;
274 #endif
275
276     ll = chain;
277
278     for (cl = *chain; cl; cl = cl->next) {
279         ll = &cl->next;
280     }
281
282     while (in) {
283
284         cl = ngx_alloc_chain_link(pool);
285         if (cl == NULL) {
286             return NGX_ERROR;
287         }
288
289 #if (NGX_SENDFILE_LIMIT)
290
291         buf = in->buf;
292
293         if (buf->in_file
294             && buf->file_pos < NGX_SENDFILE_LIMIT
295             && buf->file_last > NGX_SENDFILE_LIMIT)
296         {
297             /* split a file buf on two bufs by the sendfile limit */
298
299             b = ngx_calloc_buf(pool);
300             if (b == NULL) {
301                 return NGX_ERROR;
302             }
303
304             ngx_memcpy(b, buf, sizeof(ngx_buf_t));
305
306             if (ngx_buf_in_memory(buf)) {
307                 buf->pos += (ssize_t) (NGX_SENDFILE_LIMIT - buf->file_pos);
308                 b->last = buf->pos;
309             }
310
311             buf->file_pos = NGX_SENDFILE_LIMIT;
312             b->file_last = NGX_SENDFILE_LIMIT;
313
314             cl->buf = b;
315
316         } else {
317             cl->buf = buf;
318             in = in->next;
319         }
320
321 #else
322         cl->buf = in->buf;
323         in = in->next;
324
325 #endif
326
327         cl->next = NULL;
328         *ll = cl;
329         ll = &cl->next;
330     }
331
332     return NGX_OK;
333 }
334
335
336 static ngx_int_t
337 ngx_output_chain_align_file_buf(ngx_output_chain_ctx_t *ctx, off_t bsize)
338 {
339     size_t     size;
340     ngx_buf_t *in;
341
342     in = ctx->in->buf;
343
344     if (in->file == NULL || !in->file->directio) {
345         return NGX_DECLINED;
346     }
347

```

```

348     ctx->directio = 1;
349
350     size = (size_t) (in->file_pos - (in->file_pos & ~(ctx->alignment - 1)));
351
352     if (size == 0) {
353
354         if (bsize >= (off_t) ctx->bufs.size) {
355             return NGX\_DECLINED;
356         }
357
358         size = (size_t) bsize;
359
360     } else {
361         size = (size_t) ctx->alignment - size;
362
363         if ((off_t) size > bsize) {
364             size = (size_t) bsize;
365         }
366     }
367
368     ctx->buf = ngx\_create\_temp\_buf(ctx->pool, size);
369     if (ctx->buf == NULL) {
370         return NGX\_ERROR;
371     }
372
373     /*
374     * we do not set ctx->buf->tag, because we do not want
375     * to reuse the buf via ctx->free list
376     */
377
378     #if (NGX_HAVE_ALIGNED_DIRECTIO)
379     ctx->unaligned = 1;
380     #endif
381
382     return NGX\_OK;
383 }
384
385
386 static ngx\_int\_t
387 ngx\_output\_chain\_get\_buf(ngx\_output\_chain\_ctx\_t *ctx, off_t bsize)
388 {
389     size_t      size;
390     ngx\_buf\_t  *b, *in;
391     ngx\_uint\_t  recycled;
392
393     in = ctx->in->buf;
394     size = ctx->bufs.size;
395     recycled = 1;
396
397     if (in->last_in_chain) {
398
399         if (bsize < (off_t) size) {
400
401             /*
402             * allocate a small temp buf for a small last buf
403             * or its small last part
404             */
405
406             size = (size_t) bsize;
407             recycled = 0;
408
409         } else if (!ctx->directio
410                 && ctx->bufs.num == 1
411                 && (bsize < (off_t) (size + size / 4)))
412         {
413             /*
414             * allocate a temp buf that equals to a last buf,
415             * if there is no directio, the last buf size is lesser
416             * than 1.25 of bufs.size and the temp buf is single
417             */
418
419             size = (size_t) bsize;
420             recycled = 0;
421         }
422     }
423

```

```

424     b = ngx_calloc_buf(ctx->pool);
425     if (b == NULL) {
426         return NGX_ERROR;
427     }
428
429     if (ctx->directio) {
430
431         /*
432          * allocate block aligned to a disk sector size to enable
433          * userland buffer direct usage conjunctly with directio
434          */
435
436         b->start = ngx_pmemalign(ctx->pool, size, (size_t) ctx->alignment);
437         if (b->start == NULL) {
438             return NGX_ERROR;
439         }
440
441     } else {
442         b->start = ngx_palloc(ctx->pool, size);
443         if (b->start == NULL) {
444             return NGX_ERROR;
445         }
446     }
447
448     b->pos = b->start;
449     b->last = b->start;
450     b->end = b->last + size;
451     b->temporary = 1;
452     b->tag = ctx->tag;
453     b->recycled = recycled;
454
455     ctx->buf = b;
456     ctx->allocated++;
457
458     return NGX_OK;
459 }
460
461
462 static ngx_int_t
463 ngx_output_chain_copy_buf(ngx_output_chain_ctx_t *ctx)
464 {
465     off_t         size;
466     ssize_t       n;
467     ngx_buf_t     *src, *dst;
468     ngx_uint_t    sendfile;
469
470     src = ctx->in->buf;
471     dst = ctx->buf;
472
473     size = ngx_buf_size(src);
474     size = ngx_min(size, dst->end - dst->pos);
475
476     sendfile = ctx->sendfile & !ctx->directio;
477
478     #if (NGX_SENDFILE_LIMIT)
479
480     if (src->in_file && src->file_pos >= NGX_SENDFILE_LIMIT) {
481         sendfile = 0;
482     }
483
484     #endif
485
486     if (ngx_buf_in_memory(src)) {
487         ngx_memcpy(dst->pos, src->pos, (size_t) size);
488         src->pos += (size_t) size;
489         dst->last += (size_t) size;
490
491         if (src->in_file) {
492
493             if (sendfile) {
494                 dst->in_file = 1;
495                 dst->file = src->file;
496                 dst->file_pos = src->file_pos;
497                 dst->file_last = src->file_pos + size;
498             } else {

```

```

500         dst->in_file = 0;
501     }
502
503     src->file_pos += size;
504
505     } else {
506         dst->in_file = 0;
507     }
508
509     if (src->pos == src->last) {
510         dst->flush = src->flush;
511         dst->last_buf = src->last_buf;
512         dst->last_in_chain = src->last_in_chain;
513     }
514
515     } else {
516
517     #if (NGX_HAVE_ALIGNED_DIRECTIO)
518
519         if (ctx->unaligned) {
520             if (ngx_directio_off(src->file->fd) == NGX_FILE_ERROR) {
521                 ngx_log_error(NGX_LOG_ALERT, ctx->pool->log, ngx_errno,
522                     ngx_directio_off_n " \"%s\" failed",
523                     src->file->name.data);
524             }
525         }
526
527     #endif
528
529     #if (NGX_HAVE_FILE_AIO)
530
531         if (ctx->aio_handler) {
532             n = ngx_file_aio_read(src->file, dst->pos, (size_t) size,
533                 src->file_pos, ctx->pool);
534             if (n == NGX_AGAIN) {
535                 ctx->aio_handler(ctx, src->file);
536                 return NGX_AGAIN;
537             }
538
539         } else {
540             n = ngx_read_file(src->file, dst->pos, (size_t) size,
541                 src->file_pos);
542         }
543     #else
544
545         n = ngx_read_file(src->file, dst->pos, (size_t) size, src->file_pos);
546
547     #endif
548
549     #if (NGX_HAVE_ALIGNED_DIRECTIO)
550
551         if (ctx->unaligned) {
552             ngx_err_t err;
553
554             err = ngx_errno;
555
556             if (ngx_directio_on(src->file->fd) == NGX_FILE_ERROR) {
557                 ngx_log_error(NGX_LOG_ALERT, ctx->pool->log, ngx_errno,
558                     ngx_directio_on_n " \"%s\" failed",
559                     src->file->name.data);
560             }
561
562             ngx_set_errno(err);
563
564             ctx->unaligned = 0;
565         }
566
567     #endif
568
569     if (n == NGX_ERROR) {
570         return (ngx_int_t) n;
571     }
572
573     if (n != size) {
574         ngx_log_error(NGX_LOG_ALERT, ctx->pool->log, 0,
575             ngx_read_file_n " read only %z of %0 from \"%s\"",

```

```

576         n, size, src->file->name.data);
577     return NGX_ERROR;
578 }
579
580 dst->last += n;
581
582 if (sendfile) {
583     dst->in_file = 1;
584     dst->file = src->file;
585     dst->file_pos = src->file_pos;
586     dst->file_last = src->file_pos + n;
587
588 } else {
589     dst->in_file = 0;
590 }
591
592 src->file_pos += n;
593
594 if (src->file_pos == src->file_last) {
595     dst->flush = src->flush;
596     dst->last_buf = src->last_buf;
597     dst->last_in_chain = src->last_in_chain;
598 }
599 }
600
601 return NGX_OK;
602 }
603
604
605 ngx_int_t
606 ngx_chain_writer(void *data, ngx_chain_t *in)
607 {
608     ngx_chain_writer_ctx_t *ctx = data;
609
610     off_t          size;
611     ngx_chain_t  *cl;
612     ngx_connection_t *c;
613
614     c = ctx->connection;
615
616     for (size = 0; in; in = in->next) {
617
618 #if 1
619         if (ngx_buf_size(in->buf) == 0 && !ngx_buf_special(in->buf)) {
620             ngx_debug_point();
621         }
622 #endif
623
624         size += ngx_buf_size(in->buf);
625
626         ngx_log_debug2(NGX_LOG_DEBUG_CORE, c->log, 0,
627             "chain writer buf fl:%d s:%u0",
628             in->buf->flush, ngx_buf_size(in->buf));
629
630         cl = ngx_alloc_chain_link(ctx->pool);
631         if (cl == NULL) {
632             return NGX_ERROR;
633         }
634
635         cl->buf = in->buf;
636         cl->next = NULL;
637         *ctx->last = cl;
638         ctx->last = &cl->next;
639     }
640
641     ngx_log_debug1(NGX_LOG_DEBUG_CORE, c->log, 0,
642         "chain writer in: %p", ctx->out);
643
644     for (cl = ctx->out; cl; cl = cl->next) {
645
646 #if 1
647         if (ngx_buf_size(cl->buf) == 0 && !ngx_buf_special(cl->buf)) {
648             ngx_debug_point();
649         }
650
651 #endif

```

```
652     size += ngx\_buf\_size(cl->buf);
653 }
654
655 if (size == 0 && !c->buffered) {
656     return NGX\_OK;
657 }
658
659 ctx->out = c->send_chain(c, ctx->out, ctx->limit);
660
661 ngx\_log\_debug1(NGX\_LOG\_DEBUG\_CORE, c->log, 0,
662     "chain writer out: %p", ctx->out);
663
664 if (ctx->out == NGX\_CHAIN\_ERROR) {
665     return NGX\_ERROR;
666 }
667
668 if (ctx->out == NULL) {
669     ctx->last = &ctx->out;
670
671     if (!c->buffered) {
672         return NGX\_OK;
673     }
674 }
675
676 return NGX\_AGAIN;
677 }
678 }
```

[One Level Up](#)

[Top Level](#)

## src/http/nginx\_http\_parse.c - nginx-1.7.10

### Global variables defined

- [usual](#)

### Functions defined

- [ngx\\_http\\_arg](#)
- [ngx\\_http\\_parse\\_chunked](#)
- [ngx\\_http\\_parse\\_complex\\_uri](#)
- [ngx\\_http\\_parse\\_header\\_line](#)
- [ngx\\_http\\_parse\\_multi\\_header\\_lines](#)
- [ngx\\_http\\_parse\\_request\\_line](#)
- [ngx\\_http\\_parse\\_set\\_cookie\\_lines](#)
- [ngx\\_http\\_parse\\_status\\_line](#)
- [ngx\\_http\\_parse\\_unsafe\\_uri](#)
- [ngx\\_http\\_parse\\_uri](#)
- [ngx\\_http\\_split\\_args](#)

### Macros defined

- [ngx\\_str3Ocmp](#)
- [ngx\\_str3Ocmp](#)
- [ngx\\_str3\\_cmp](#)
- [ngx\\_str3\\_cmp](#)
- [ngx\\_str4cmp](#)
- [ngx\\_str4cmp](#)
- [ngx\\_str5cmp](#)
- [ngx\\_str5cmp](#)
- [ngx\\_str6cmp](#)
- [ngx\\_str6cmp](#)
- [ngx\\_str7\\_cmp](#)
- [ngx\\_str7\\_cmp](#)
- [ngx\\_str8cmp](#)
- [ngx\\_str8cmp](#)
- [ngx\\_str9cmp](#)

- [ngx\\_str9cmp](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 static uint32_t usual[] = {
14     0xffffdbfe, /* 1111 1111 1111 1111 1101 1011 1111 1110 */
15
16     /* ?>=< ;:98 7654 3210 /.-, +*)( '&%$ #"! */
17     0x7fff37d6, /* 0111 1111 1111 1111 0011 0111 1101 0110 */
18
19     /* _^]\ [ZYX WVUT SRQP ONML KJIH GFED CBA@ */
20 #if (NGX_WIN32)
21     0xffffffff, /* 1110 1111 1111 1111 1111 1111 1111 1111 */
22 #else
23     0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
24 #endif
25
26     /* ~}| {zyx wvut srqp onml kjih gfed cba` */
27     0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
28
29     0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
30     0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
31     0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
32     0xffffffff /* 1111 1111 1111 1111 1111 1111 1111 1111 */
33 };
34
35
36 #if (NGX_HAVE_LITTLE_ENDIAN && NGX_HAVE_NONALIGNED)
37
38 #define ngx_str3_cmp(m, c0, c1, c2, c3) \
39     *(uint32_t *) m == ((c3 << 24) | (c2 << 16) | (c1 << 8) | c0)
40
41 #define ngx_str30cmp(m, c0, c1, c2, c3) \
42     *(uint32_t *) m == ((c3 << 24) | (c2 << 16) | (c1 << 8) | c0)
43
44 #define ngx_str4cmp(m, c0, c1, c2, c3) \
45     *(uint32_t *) m == ((c3 << 24) | (c2 << 16) | (c1 << 8) | c0)
46
47 #define ngx_str5cmp(m, c0, c1, c2, c3, c4) \
48     *(uint32_t *) m == ((c3 << 24) | (c2 << 16) | (c1 << 8) | c0) \
49     && m[4] == c4
50
51 #define ngx_str6cmp(m, c0, c1, c2, c3, c4, c5) \
52     *(uint32_t *) m == ((c3 << 24) | (c2 << 16) | (c1 << 8) | c0) \
53     && ((uint32_t *) m)[1] & 0xffff == ((c5 << 8) | c4)
54
55 #define ngx_str7_cmp(m, c0, c1, c2, c3, c4, c5, c6, c7) \
56     *(uint32_t *) m == ((c3 << 24) | (c2 << 16) | (c1 << 8) | c0) \
57     && ((uint32_t *) m)[1] == ((c7 << 24) | (c6 << 16) | (c5 << 8) | c4)
58
59 #define ngx_str8cmp(m, c0, c1, c2, c3, c4, c5, c6, c7) \
60     *(uint32_t *) m == ((c3 << 24) | (c2 << 16) | (c1 << 8) | c0) \
61     && ((uint32_t *) m)[1] == ((c7 << 24) | (c6 << 16) | (c5 << 8) | c4)
62
63 #define ngx_str9cmp(m, c0, c1, c2, c3, c4, c5, c6, c7, c8) \
64     *(uint32_t *) m == ((c3 << 24) | (c2 << 16) | (c1 << 8) | c0) \
65     && ((uint32_t *) m)[1] == ((c7 << 24) | (c6 << 16) | (c5 << 8) | c4) \
66     && m[8] == c8
67
68 #else /* !(NGX_HAVE_LITTLE_ENDIAN && NGX_HAVE_NONALIGNED) */
69
70 #define ngx_str3_cmp(m, c0, c1, c2, c3) \

```



```

71     m[0] == c0 && m[1] == c1 && m[2] == c2
72
73 #define ngx_str30cmp(m, c0, c1, c2, c3) \
74     m[0] == c0 && m[2] == c2 && m[3] == c3
75
76 #define ngx_str4cmp(m, c0, c1, c2, c3) \
77     m[0] == c0 && m[1] == c1 && m[2] == c2 && m[3] == c3
78
79 #define ngx_str5cmp(m, c0, c1, c2, c3, c4) \
80     m[0] == c0 && m[1] == c1 && m[2] == c2 && m[3] == c3 && m[4] == c4
81
82 #define ngx_str6cmp(m, c0, c1, c2, c3, c4, c5) \
83     m[0] == c0 && m[1] == c1 && m[2] == c2 && m[3] == c3 \
84     && m[4] == c4 && m[5] == c5
85
86 #define ngx_str7_cmp(m, c0, c1, c2, c3, c4, c5, c6, c7) \
87     m[0] == c0 && m[1] == c1 && m[2] == c2 && m[3] == c3 \
88     && m[4] == c4 && m[5] == c5 && m[6] == c6
89
90 #define ngx_str8cmp(m, c0, c1, c2, c3, c4, c5, c6, c7) \
91     m[0] == c0 && m[1] == c1 && m[2] == c2 && m[3] == c3 \
92     && m[4] == c4 && m[5] == c5 && m[6] == c6 && m[7] == c7
93
94 #define ngx_str9cmp(m, c0, c1, c2, c3, c4, c5, c6, c7, c8) \
95     m[0] == c0 && m[1] == c1 && m[2] == c2 && m[3] == c3 \
96     && m[4] == c4 && m[5] == c5 && m[6] == c6 && m[7] == c7 && m[8] == c8
97
98 #endif
99
100
101 /* gcc, icc, msvc and others compile these switches as an jump table */
102
103 ngx_int_t
104 ngx_http_parse_request_line(ngx_http_request_t *r, ngx_buf_t *b)
105 {
106     u_char c, ch, *p, *m;
107     enum {
108         sw_start = 0,
109         sw_method,
110         sw_spaces_before_uri,
111         sw_schema,
112         sw_schema_slash,
113         sw_schema_slash_slash,
114         sw_host_start,
115         sw_host,
116         sw_host_end,
117         sw_host_ip_literal,
118         sw_port,
119         sw_host_http_09,
120         sw_after_slash_in_uri,
121         sw_check_uri,
122         sw_check_uri_http_09,
123         sw_uri,
124         sw_http_09,
125         sw_http_H,
126         sw_http_HT,
127         sw_http_HTTP,
128         sw_http_HTTP,
129         sw_first_major_digit,
130         sw_major_digit,
131         sw_first_minor_digit,
132         sw_minor_digit,
133         sw_spaces_after_digit,
134         sw_almost_done
135     } state;
136
137     state = r->state;
138
139     for (p = b->pos; p < b->last; p++) {
140         ch = *p;
141
142         switch (state) {
143
144             /* HTTP methods: GET, HEAD, POST */
145             case sw_start:
146                 r->request_start = p;

```

```

147
148     if (ch == CR || ch == LF) {
149         break;
150     }
151
152     if ((ch < 'A' || ch > 'Z') && ch != '_') {
153         return NGX\_HTTP\_PARSE\_INVALID\_METHOD;
154     }
155
156     state = sw_method;
157     break;
158
159 case sw_method:
160     if (ch == ' ') {
161         r->method_end = p - 1;
162         m = r->request_start;
163
164         switch (p - m) {
165
166         case 3:
167             if (ngx\_str3\_cmp(m, 'G', 'E', 'T', ' ')) {
168                 r->method = NGX\_HTTP\_GET;
169                 break;
170             }
171
172             if (ngx\_str3\_cmp(m, 'P', 'U', 'T', ' ')) {
173                 r->method = NGX\_HTTP\_PUT;
174                 break;
175             }
176
177             break;
178
179         case 4:
180             if (m[1] == 'O') {
181
182                 if (ngx\_str30cmp(m, 'P', 'O', 'S', 'T')) {
183                     r->method = NGX\_HTTP\_POST;
184                     break;
185                 }
186
187                 if (ngx\_str30cmp(m, 'C', 'O', 'P', 'Y')) {
188                     r->method = NGX\_HTTP\_COPY;
189                     break;
190                 }
191
192                 if (ngx\_str30cmp(m, 'M', 'O', 'V', 'E')) {
193                     r->method = NGX\_HTTP\_MOVE;
194                     break;
195                 }
196
197                 if (ngx\_str30cmp(m, 'L', 'O', 'C', 'K')) {
198                     r->method = NGX\_HTTP\_LOCK;
199                     break;
200                 }
201
202             } else {
203
204                 if (ngx\_str4cmp(m, 'H', 'E', 'A', 'D')) {
205                     r->method = NGX\_HTTP\_HEAD;
206                     break;
207                 }
208             }
209
210             break;
211
212         case 5:
213             if (ngx\_str5cmp(m, 'M', 'K', 'C', 'O', 'L')) {
214                 r->method = NGX\_HTTP\_MKCOL;
215                 break;
216             }
217
218             if (ngx\_str5cmp(m, 'P', 'A', 'T', 'C', 'H')) {
219                 r->method = NGX\_HTTP\_PATCH;
220                 break;
221             }
222

```

```

223     if (ngx_str5cmp(m, 'T', 'R', 'A', 'C', 'E')) {
224         r->method = NGX\_HTTP\_TRACE;
225         break;
226     }
227
228     break;
229
230 case 6:
231     if (ngx_str6cmp(m, 'D', 'E', 'L', 'E', 'T', 'E')) {
232         r->method = NGX\_HTTP\_DELETE;
233         break;
234     }
235
236     if (ngx_str6cmp(m, 'U', 'N', 'L', 'O', 'C', 'K')) {
237         r->method = NGX\_HTTP\_UNLOCK;
238         break;
239     }
240
241     break;
242
243 case 7:
244     if (ngx_str7_cmp(m, 'O', 'P', 'T', 'I', 'O', 'N', 'S', ' '))
245     {
246         r->method = NGX\_HTTP\_OPTIONS;
247     }
248
249     break;
250
251 case 8:
252     if (ngx_str8cmp(m, 'P', 'R', 'O', 'P', 'E', 'R', 'T', 'Y'))
253     {
254         r->method = NGX\_HTTP\_PROPFIND;
255     }
256
257     break;
258
259 case 9:
260     if (ngx_str9cmp(m,
261         'P', 'R', 'O', 'P', 'A', 'T', 'I', 'O', 'N'))
262     {
263         r->method = NGX\_HTTP\_PROPPATCH;
264     }
265
266     break;
267 }
268
269 state = sw_spaces_before_uri;
270 break;
271 }
272
273 if ((ch < 'A' || ch > 'Z') && ch != '_') {
274     return NGX\_HTTP\_PARSE\_INVALID\_METHOD;
275 }
276
277 break;
278
279 /* space* before URI */
280 case sw_spaces_before_uri:
281
282     if (ch == '/') {
283         r->uri_start = p;
284         state = sw_after_slash_in_uri;
285         break;
286     }
287
288     c = (u_char) (ch | 0x20);
289     if (c >= 'a' && c <= 'z') {
290         r->schema_start = p;
291         state = sw_schema;
292         break;
293     }
294
295     switch (ch) {
296     case ' ':
297         break;
298     default:

```

```

299         return NGX\_HTTP\_PARSE\_INVALID\_REQUEST;
300     }
301     break;
302
303 case sw_schema:
304
305     c = (u_char) (ch | 0x20);
306     if (c >= 'a' && c <= 'z') {
307         break;
308     }
309
310     switch (ch) {
311     case ':':
312         r->schema_end = p;
313         state = sw_schema_slash;
314         break;
315     default:
316         return NGX\_HTTP\_PARSE\_INVALID\_REQUEST;
317     }
318     break;
319
320 case sw_schema_slash:
321     switch (ch) {
322     case '/':
323         state = sw_schema_slash_slash;
324         break;
325     default:
326         return NGX\_HTTP\_PARSE\_INVALID\_REQUEST;
327     }
328     break;
329
330 case sw_schema_slash_slash:
331     switch (ch) {
332     case '/':
333         state = sw_host_start;
334         break;
335     default:
336         return NGX\_HTTP\_PARSE\_INVALID\_REQUEST;
337     }
338     break;
339
340 case sw_host_start:
341
342     r->host_start = p;
343
344     if (ch == '[') {
345         state = sw_host_ip_literal;
346         break;
347     }
348
349     state = sw_host;
350
351     /* fall through */
352
353 case sw_host:
354
355     c = (u_char) (ch | 0x20);
356     if (c >= 'a' && c <= 'z') {
357         break;
358     }
359
360     if ((ch >= '0' && ch <= '9') || ch == '.' || ch == '-') {
361         break;
362     }
363
364     /* fall through */
365
366 case sw_host_end:
367
368     r->host_end = p;
369
370     switch (ch) {
371     case ':':
372         state = sw_port;
373         break;
374     case '/':

```

```

375         r->uri_start = p;
376         state = sw_after_slash_in_uri;
377         break;
378     case ' ':
379         /*
380          * use single "/" from request line to preserve pointers,
381          * if request line will be copied to large client buffer
382          */
383         r->uri_start = r->schema_end + 1;
384         r->uri_end = r->schema_end + 2;
385         state = sw_host_http_09;
386         break;
387     default:
388         return NGX_HTTP_PARSE_INVALID_REQUEST;
389     }
390     break;
391
392 case sw_host_ip_literal:
393
394     if (ch >= '0' && ch <= '9') {
395         break;
396     }
397
398     c = (u_char) (ch | 0x20);
399     if (c >= 'a' && c <= 'z') {
400         break;
401     }
402
403     switch (ch) {
404     case ':':
405         break;
406     case ']':
407         state = sw_host_end;
408         break;
409     case '-':
410     case '.':
411     case '_':
412     case '~':
413         /* unreserved */
414         break;
415     case '!':
416     case '$':
417     case '&':
418     case '\\':
419     case '(':
420     case ')':
421     case '*':
422     case '+':
423     case ',':
424     case ';':
425     case '=':
426         /* sub-delims */
427         break;
428     default:
429         return NGX_HTTP_PARSE_INVALID_REQUEST;
430     }
431     break;
432
433 case sw_port:
434     if (ch >= '0' && ch <= '9') {
435         break;
436     }
437
438     switch (ch) {
439     case '/':
440         r->port_end = p;
441         r->uri_start = p;
442         state = sw_after_slash_in_uri;
443         break;
444     case ' ':
445         r->port_end = p;
446         /*
447          * use single "/" from request line to preserve pointers,
448          * if request line will be copied to large client buffer
449          */
450         r->uri_start = r->schema_end + 1;

```

```

451         r->uri_end = r->schema_end + 2;
452         state = sw_host_http_09;
453         break;
454     default:
455         return NGX_HTTP_PARSE_INVALID_REQUEST;
456     }
457     break;
458
459     /* space+ after "http://host[:port] " */
460     case sw_host_http_09:
461         switch (ch) {
462             case ' ':
463                 break;
464             case CR:
465                 r->http_minor = 9;
466                 state = sw_almost_done;
467                 break;
468             case LF:
469                 r->http_minor = 9;
470                 goto done;
471             case 'H':
472                 r->http_protocol.data = p;
473                 state = sw_http_H;
474                 break;
475             default:
476                 return NGX_HTTP_PARSE_INVALID_REQUEST;
477         }
478         break;
479
480     /* check "/.", "//", "%", and "\" (Win32) in URI */
481     case sw_after_slash_in_uri:
482
483         if (usual[ch >> 5] & (1 << (ch & 0x1f))) {
484             state = sw_check_uri;
485             break;
486         }
487
488         switch (ch) {
489             case ' ':
490                 r->uri_end = p;
491                 state = sw_check_uri_http_09;
492                 break;
493             case CR:
494                 r->uri_end = p;
495                 r->http_minor = 9;
496                 state = sw_almost_done;
497                 break;
498             case LF:
499                 r->uri_end = p;
500                 r->http_minor = 9;
501                 goto done;
502             case '.':
503                 r->complex_uri = 1;
504                 state = sw_uri;
505                 break;
506             case '%':
507                 r->quoted_uri = 1;
508                 state = sw_uri;
509                 break;
510             case '/':
511                 r->complex_uri = 1;
512                 state = sw_uri;
513                 break;
514             #if (NGX_WIN32)
515                 case '\\':
516                     r->complex_uri = 1;
517                     state = sw_uri;
518                     break;
519             #endif
520             case '?':
521                 r->args_start = p + 1;
522                 state = sw_uri;
523                 break;
524             case '#':
525                 r->complex_uri = 1;
526

```

```

527         state = sw_uri;
528         break;
529     case '+':
530         r->plus_in_uri = 1;
531         break;
532     case '\0':
533         return NGX\_HTTP\_PARSE\_INVALID\_REQUEST;
534     default:
535         state = sw_check_uri;
536         break;
537     }
538     break;
539
540     /* check "/", "%" and "\" (Win32) in URI */
541     case sw_check_uri:
542
543         if (usual[ch >> 5] & (1 << (ch & 0x1f))) {
544             break;
545         }
546
547         switch (ch) {
548             case '/':
549 #if (NGX_WIN32)
550                 if (r->uri_ext == p) {
551                     r->complex_uri = 1;
552                     state = sw_uri;
553                     break;
554                 }
555 #endif
556                 r->uri_ext = NULL;
557                 state = sw_after_slash_in_uri;
558                 break;
559             case '.':
560                 r->uri_ext = p + 1;
561                 break;
562             case ' ':
563                 r->uri_end = p;
564                 state = sw_check_uri_http_09;
565                 break;
566             case CR:
567                 r->uri_end = p;
568                 r->http_minor = 9;
569                 state = sw_almost_done;
570                 break;
571             case LF:
572                 r->uri_end = p;
573                 r->http_minor = 9;
574                 goto done;
575 #if (NGX_WIN32)
576             case '\\':
577                 r->complex_uri = 1;
578                 state = sw_after_slash_in_uri;
579                 break;
580 #endif
581             case '%':
582                 r->quoted_uri = 1;
583                 state = sw_uri;
584                 break;
585             case '?':
586                 r->args_start = p + 1;
587                 state = sw_uri;
588                 break;
589             case '#':
590                 r->complex_uri = 1;
591                 state = sw_uri;
592                 break;
593             case '+':
594                 r->plus_in_uri = 1;
595                 break;
596             case '\0':
597                 return NGX\_HTTP\_PARSE\_INVALID\_REQUEST;
598             }
599         break;
600
601     /* space+ after URI */
602     case sw_check_uri_http_09:

```

```

603     switch (ch) {
604     case ' ':
605         break;
606     case CR:
607         r->http_minor = 9;
608         state = sw_almost_done;
609         break;
610     case LF:
611         r->http_minor = 9;
612         goto done;
613     case 'H':
614         r->http_protocol.data = p;
615         state = sw_http_H;
616         break;
617     default:
618         r->space_in_uri = 1;
619         state = sw_check_uri;
620         p--;
621         break;
622     }
623     break;
624
625
626     /* URI */
627     case sw_uri:
628
629         if (usual[ch >> 5] & (1 << (ch & 0x1f))) {
630             break;
631         }
632
633         switch (ch) {
634         case ' ':
635             r->uri_end = p;
636             state = sw_http_09;
637             break;
638         case CR:
639             r->uri_end = p;
640             r->http_minor = 9;
641             state = sw_almost_done;
642             break;
643         case LF:
644             r->uri_end = p;
645             r->http_minor = 9;
646             goto done;
647         case '#':
648             r->complex_uri = 1;
649             break;
650         case '\0':
651             return NGX_HTTP_PARSE_INVALID_REQUEST;
652         }
653         break;
654
655     /* space+ after URI */
656     case sw_http_09:
657         switch (ch) {
658         case ' ':
659             break;
660         case CR:
661             r->http_minor = 9;
662             state = sw_almost_done;
663             break;
664         case LF:
665             r->http_minor = 9;
666             goto done;
667         case 'H':
668             r->http_protocol.data = p;
669             state = sw_http_H;
670             break;
671         default:
672             r->space_in_uri = 1;
673             state = sw_uri;
674             p--;
675             break;
676         }
677         break;
678

```



```

679     case sw_http_H:
680         switch (ch) {
681             case 'T':
682                 state = sw_http_HT;
683                 break;
684             default:
685                 return NGX\_HTTP\_PARSE\_INVALID\_REQUEST;
686         }
687         break;
688
689     case sw_http_HT:
690         switch (ch) {
691             case 'T':
692                 state = sw_http_HTT;
693                 break;
694             default:
695                 return NGX\_HTTP\_PARSE\_INVALID\_REQUEST;
696         }
697         break;
698
699     case sw_http_HTT:
700         switch (ch) {
701             case 'P':
702                 state = sw_http_HTTP;
703                 break;
704             default:
705                 return NGX\_HTTP\_PARSE\_INVALID\_REQUEST;
706         }
707         break;
708
709     case sw_http_HTTP:
710         switch (ch) {
711             case '/':
712                 state = sw_first_major_digit;
713                 break;
714             default:
715                 return NGX\_HTTP\_PARSE\_INVALID\_REQUEST;
716         }
717         break;
718
719     /* first digit of major HTTP version */
720     case sw_first_major_digit:
721         if (ch < '1' || ch > '9') {
722             return NGX\_HTTP\_PARSE\_INVALID\_REQUEST;
723         }
724
725         r->http_major = ch - '0';
726         state = sw_major_digit;
727         break;
728
729     /* major HTTP version or dot */
730     case sw_major_digit:
731         if (ch == '.') {
732             state = sw_first_minor_digit;
733             break;
734         }
735
736         if (ch < '0' || ch > '9') {
737             return NGX\_HTTP\_PARSE\_INVALID\_REQUEST;
738         }
739
740         r->http_major = r->http_major * 10 + ch - '0';
741         break;
742
743     /* first digit of minor HTTP version */
744     case sw_first_minor_digit:
745         if (ch < '0' || ch > '9') {
746             return NGX\_HTTP\_PARSE\_INVALID\_REQUEST;
747         }
748
749         r->http_minor = ch - '0';
750         state = sw_minor_digit;
751         break;
752
753     /* minor HTTP version or end of request line */
754     case sw_minor_digit:

```

```

755     if (ch == CR) {
756         state = sw_almost_done;
757         break;
758     }
759
760     if (ch == LF) {
761         goto done;
762     }
763
764     if (ch == ' ') {
765         state = sw_spaces_after_digit;
766         break;
767     }
768
769     if (ch < '0' || ch > '9') {
770         return NGX_HTTP_PARSE_INVALID_REQUEST;
771     }
772
773     r->http_minor = r->http_minor * 10 + ch - '0';
774     break;
775
776     case sw_spaces_after_digit:
777         switch (ch) {
778             case ' ':
779                 break;
780             case CR:
781                 state = sw_almost_done;
782                 break;
783             case LF:
784                 goto done;
785             default:
786                 return NGX_HTTP_PARSE_INVALID_REQUEST;
787         }
788         break;
789
790     /* end of request line */
791     case sw_almost_done:
792         r->request_end = p - 1;
793         switch (ch) {
794             case LF:
795                 goto done;
796             default:
797                 return NGX_HTTP_PARSE_INVALID_REQUEST;
798         }
799     }
800 }
801
802 b->pos = p;
803 r->state = state;
804
805 return NGX_AGAIN;
806
807 done:
808
809 b->pos = p + 1;
810
811 if (r->request_end == NULL) {
812     r->request_end = p;
813 }
814
815 r->http_version = r->http_major * 1000 + r->http_minor;
816 r->state = sw_start;
817
818 if (r->http_version == 9 && r->method != NGX_HTTP_GET) {
819     return NGX_HTTP_PARSE_INVALID_09_METHOD;
820 }
821
822 return NGX_OK;
823 }
824
825
826 ngx_int_t
827 ngx_http_parse_header_line(ngx_http_request_t *r, ngx_buf_t *b,
828     ngx_uint_t allow_underscores)
829 {
830     u_char    c, ch, *p;

```

```

831     ngx_uint_t hash, i;
832     enum {
833         sw_start = 0,
834         sw_name,
835         sw_space_before_value,
836         sw_value,
837         sw_space_after_value,
838         sw_ignore_line,
839         sw_almost_done,
840         sw_header_almost_done
841     } state;
842
843     /* the last '\0' is not needed because string is zero terminated */
844
845     static u_char lowercase[] =
846         "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"
847         "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"
848         "\0abcdefghijklmnopqrstuvwxyz\0\0\0\0\0"
849         "\0abcdefghijklmnopqrstuvwxyz\0\0\0\0\0"
850         "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"
851         "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"
852         "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"
853         "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0";
854
855     state = r->state;
856     hash = r->header_hash;
857     i = r->lowercase_index;
858
859     for (p = b->pos; p < b->last; p++) {
860         ch = *p;
861
862         switch (state) {
863
864             /* first char */
865             case sw_start:
866                 r->header_name_start = p;
867                 r->invalid_header = 0;
868
869                 switch (ch) {
870                     case CR:
871                         r->header_end = p;
872                         state = sw_header_almost_done;
873                         break;
874                     case LF:
875                         r->header_end = p;
876                         goto header_done;
877                     default:
878                         state = sw_name;
879
880                         c = lowercase[ch];
881
882                         if (c) {
883                             hash = ngx_hash(0, c);
884                             r->lowercase_header[0] = c;
885                             i = 1;
886                             break;
887                         }
888
889                         if (ch == '_') {
890                             if (allow_underscores) {
891                                 hash = ngx_hash(0, ch);
892                                 r->lowercase_header[0] = ch;
893                                 i = 1;
894
895                             } else {
896                                 r->invalid_header = 1;
897
898                             }
899
900                         } else {
901                             break;
902                         }
903
904                         if (ch == '\0') {
905                             return NGX_HTTP_PARSE_INVALID_HEADER;
906                         }
907
908                         r->invalid_header = 1;

```

```

907         break;
908
909     }
910     break;
911
912     /* header name */
913     case sw_name:
914         c = lowercase[ch];
915
916         if (c) {
917             hash = ngx_hash(hash, c);
918             r->lowercase_header[i++] = c;
919             i &= (NGX_HTTP_LC_HEADER_LEN - 1);
920             break;
921         }
922
923         if (ch == '_') {
924             if (allow_underscores) {
925                 hash = ngx_hash(hash, ch);
926                 r->lowercase_header[i++] = ch;
927                 i &= (NGX_HTTP_LC_HEADER_LEN - 1);
928             } else {
929                 r->invalid_header = 1;
930             }
931         }
932         break;
933     }
934
935     if (ch == ':') {
936         r->header_name_end = p;
937         state = sw_space_before_value;
938         break;
939     }
940
941     if (ch == CR) {
942         r->header_name_end = p;
943         r->header_start = p;
944         r->header_end = p;
945         state = sw_almost_done;
946         break;
947     }
948
949     if (ch == LF) {
950         r->header_name_end = p;
951         r->header_start = p;
952         r->header_end = p;
953         goto done;
954     }
955
956     /* IIS may send the duplicate "HTTP/1.1 ..." lines */
957     if (ch == '/')
958         && r->upstream
959         && p - r->header_name_start == 4
960         && ngx_strncmp(r->header_name_start, "HTTP", 4) == 0)
961     {
962         state = sw_ignore_line;
963         break;
964     }
965
966     if (ch == '\0') {
967         return NGX_HTTP_PARSE_INVALID_HEADER;
968     }
969
970     r->invalid_header = 1;
971
972     break;
973
974     /* space* before header value */
975     case sw_space_before_value:
976         switch (ch) {
977             case ' ':
978                 break;
979             case CR:
980                 r->header_start = p;

```

```

983         r->header_end = p;
984         state = sw_almost_done;
985         break;
986     case LF:
987         r->header_start = p;
988         r->header_end = p;
989         goto done;
990     case '\0':
991         return NGX\_HTTP\_PARSE\_INVALID\_HEADER;
992     default:
993         r->header_start = p;
994         state = sw_value;
995         break;
996 }
997 break;
998
999 /* header value */
1000 case sw_value:
1001     switch (ch) {
1002     case ' ':
1003         r->header_end = p;
1004         state = sw_space_after_value;
1005         break;
1006     case CR:
1007         r->header_end = p;
1008         state = sw_almost_done;
1009         break;
1010     case LF:
1011         r->header_end = p;
1012         goto done;
1013     case '\0':
1014         return NGX\_HTTP\_PARSE\_INVALID\_HEADER;
1015     }
1016     break;
1017
1018 /* space* before end of header line */
1019 case sw_space_after_value:
1020     switch (ch) {
1021     case ' ':
1022         break;
1023     case CR:
1024         state = sw_almost_done;
1025         break;
1026     case LF:
1027         goto done;
1028     case '\0':
1029         return NGX\_HTTP\_PARSE\_INVALID\_HEADER;
1030     default:
1031         state = sw_value;
1032         break;
1033     }
1034     break;
1035
1036 /* ignore header line */
1037 case sw_ignore_line:
1038     switch (ch) {
1039     case LF:
1040         state = sw_start;
1041         break;
1042     default:
1043         break;
1044     }
1045     break;
1046
1047 /* end of header line */
1048 case sw_almost_done:
1049     switch (ch) {
1050     case LF:
1051         goto done;
1052     case CR:
1053         break;
1054     default:
1055         return NGX\_HTTP\_PARSE\_INVALID\_HEADER;
1056     }
1057     break;
1058

```

```

1059     /* end of header */
1060     case sw_header_almost_done:
1061         switch (ch) {
1062             case LF:
1063                 goto header_done;
1064             default:
1065                 return NGX_HTTP_PARSE_INVALID_HEADER;
1066         }
1067     }
1068 }
1069
1070 b->pos = p;
1071 r->state = state;
1072 r->header_hash = hash;
1073 r->lowercase_index = i;
1074
1075 return NGX_AGAIN;
1076
1077 done:
1078
1079 b->pos = p + 1;
1080 r->state = sw_start;
1081 r->header_hash = hash;
1082 r->lowercase_index = i;
1083
1084 return NGX_OK;
1085
1086 header_done:
1087
1088 b->pos = p + 1;
1089 r->state = sw_start;
1090
1091 return NGX_HTTP_PARSE_HEADER_DONE;
1092 }
1093
1094
1095 ngx_int_t
1096 ngx_http_parse_uri(ngx_http_request_t *r)
1097 {
1098     u_char *p, ch;
1099     enum {
1100         sw_start = 0,
1101         sw_after_slash_in_uri,
1102         sw_check_uri,
1103         sw_uri
1104     } state;
1105
1106     state = sw_start;
1107
1108     for (p = r->uri_start; p != r->uri_end; p++) {
1109
1110         ch = *p;
1111
1112         switch (state) {
1113
1114             case sw_start:
1115
1116                 if (ch != '/') {
1117                     return NGX_ERROR;
1118                 }
1119
1120                 state = sw_after_slash_in_uri;
1121                 break;
1122
1123                 /* check "/*.", "///", "%", and "\" (Win32) in URI */
1124                 case sw_after_slash_in_uri:
1125
1126                     if (usual[ch >> 5] & (1 << (ch & 0x1f))) {
1127                         state = sw_check_uri;
1128                         break;
1129                     }
1130
1131                     switch (ch) {
1132                         case ' ':
1133                             r->space_in_uri = 1;
1134                             state = sw_check_uri;

```

```

1135         break;
1136     case '.':
1137         r->complex_uri = 1;
1138         state = sw_uri;
1139         break;
1140     case '%':
1141         r->quoted_uri = 1;
1142         state = sw_uri;
1143         break;
1144     case '/':
1145         r->complex_uri = 1;
1146         state = sw_uri;
1147         break;
1148     #if (NGX_WIN32)
1149         case '\\':
1150             r->complex_uri = 1;
1151             state = sw_uri;
1152             break;
1153     #endif
1154     case '?':
1155         r->args_start = p + 1;
1156         state = sw_uri;
1157         break;
1158     case '#':
1159         r->complex_uri = 1;
1160         state = sw_uri;
1161         break;
1162     case '+':
1163         r->plus_in_uri = 1;
1164         break;
1165     default:
1166         state = sw_check_uri;
1167         break;
1168     }
1169     break;
1170
1171     /* check "/", "%" and "\" (Win32) in URI */
1172     case sw_check_uri:
1173
1174         if (usual[ch >> 5] & (1 << (ch & 0x1f))) {
1175             break;
1176         }
1177
1178         switch (ch) {
1179             case '/':
1180                 #if (NGX_WIN32)
1181                     if (r->uri_ext == p) {
1182                         r->complex_uri = 1;
1183                         state = sw_uri;
1184                         break;
1185                     }
1186                 #endif
1187                 r->uri_ext = NULL;
1188                 state = sw_after_slash_in_uri;
1189                 break;
1190             case '.':
1191                 r->uri_ext = p + 1;
1192                 break;
1193             case ' ':
1194                 r->space_in_uri = 1;
1195                 break;
1196                 #if (NGX_WIN32)
1197                 case '\\':
1198                     r->complex_uri = 1;
1199                     state = sw_after_slash_in_uri;
1200                     break;
1201                 #endif
1202             case '%':
1203                 r->quoted_uri = 1;
1204                 state = sw_uri;
1205                 break;
1206             case '?':
1207                 r->args_start = p + 1;
1208                 state = sw_uri;
1209                 break;
1210             case '#':

```

```

1211         r->complex_uri = 1;
1212         state = sw_uri;
1213         break;
1214     case '+':
1215         r->plus_in_uri = 1;
1216         break;
1217     }
1218     break;
1219
1220     /* URI */
1221     case sw_uri:
1222
1223         if (usual[ch >> 5] & (1 << (ch & 0x1f))) {
1224             break;
1225         }
1226
1227         switch (ch) {
1228             case ' ':
1229                 r->space_in_uri = 1;
1230                 break;
1231             case '#':
1232                 r->complex_uri = 1;
1233                 break;
1234         }
1235         break;
1236     }
1237 }
1238
1239 return NGX_OK;
1240 }
1241
1242
1243 ngx_int_t
1244 ngx_http_parse_complex_uri(ngx_http_request_t *r, ngx_uint_t merge_slashes)
1245 {
1246     u_char c, ch, decoded, *p, *u;
1247     enum {
1248         sw_usual = 0,
1249         sw_slash,
1250         sw_dot,
1251         sw_dot_dot,
1252         sw_quoted,
1253         sw_quoted_second
1254     } state, quoted_state;
1255
1256     #if (NGX_SUPPRESS_WARN)
1257     decoded = '\0';
1258     quoted_state = sw_usual;
1259     #endif
1260
1261     state = sw_usual;
1262     p = r->uri_start;
1263     u = r->uri.data;
1264     r->uri_ext = NULL;
1265     r->args_start = NULL;
1266
1267     ch = *p++;
1268
1269     while (p <= r->uri_end) {
1270
1271         /*
1272          * we use "ch = *p++" inside the cycle, but this operation is safe,
1273          * because after the URI there is always at least one character:
1274          * the line feed
1275          */
1276
1277         ngx_log_debug3(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1278             "s:%d in:'%Xd:%c'", state, ch, ch);
1279
1280         switch (state) {
1281             case sw_usual:
1282
1283                 if (usual[ch >> 5] & (1 << (ch & 0x1f))) {
1284                     *u++ = ch;
1285                     ch = *p++;
1286

```



```

1287         break;
1288     }
1289
1290     switch (ch) {
1291 #if (NGX_WIN32)
1292     case '\\':
1293         if (u - 2 >= r->uri.data
1294             && *(u - 1) == '.' && *(u - 2) != '.')
1295             {
1296                 u--;
1297             }
1298
1299         r->uri_ext = NULL;
1300
1301         if (p == r->uri_start + r->uri.len) {
1302             /*
1303              * we omit the last "\" to cause redirect because
1304              * the browsers do not treat "\" as "/" in relative URL path
1305              */
1306
1307             break;
1308         }
1309
1310         state = sw_slash;
1311         *u++ = '/';
1312         break;
1313 #endif
1314     case '/':
1315 #if (NGX_WIN32)
1316         if (u - 2 >= r->uri.data
1317             && *(u - 1) == '.' && *(u - 2) != '.')
1318             {
1319                 u--;
1320             }
1321 #endif
1322 #endif
1323         r->uri_ext = NULL;
1324         state = sw_slash;
1325         *u++ = ch;
1326         break;
1327     case '%':
1328         quoted_state = state;
1329         state = sw_quoted;
1330         break;
1331     case '?':
1332         r->args_start = p;
1333         goto args;
1334     case '#':
1335         goto done;
1336     case '.':
1337         r->uri_ext = u + 1;
1338         *u++ = ch;
1339         break;
1340     case '+':
1341         r->plus_in_uri = 1;
1342         /* fall through */
1343     default:
1344         *u++ = ch;
1345         break;
1346     }
1347
1348     ch = *p++;
1349     break;
1350
1351 case sw_slash:
1352
1353     if (usual[ch >> 5] & (1 << (ch & 0x1f))) {
1354         state = sw_usual;
1355         *u++ = ch;
1356         ch = *p++;
1357         break;
1358     }
1359
1360     switch (ch) {
1361 #if (NGX_WIN32)
1362     case '\\':

```

```

1363         break;
1364 #endif
1365     case '/':
1366         if (!merge_slashes) {
1367             *u++ = ch;
1368         }
1369         break;
1370     case '.':
1371         state = sw_dot;
1372         *u++ = ch;
1373         break;
1374     case '%':
1375         quoted_state = state;
1376         state = sw_quoted;
1377         break;
1378     case '?':
1379         r->args_start = p;
1380         goto args;
1381     case '#':
1382         goto done;
1383     case '+':
1384         r->plus_in_uri = 1;
1385     default:
1386         state = sw_usual;
1387         *u++ = ch;
1388         break;
1389     }
1390
1391     ch = *p++;
1392     break;
1393
1394 case sw_dot:
1395
1396     if (usual[ch >> 5] & (1 << (ch & 0x1f))) {
1397         state = sw_usual;
1398         *u++ = ch;
1399         ch = *p++;
1400         break;
1401     }
1402
1403     switch (ch) {
1404 #if (NGX_WIN32)
1405     case '\\':
1406 #endif
1407     case '/':
1408         state = sw_slash;
1409         u--;
1410         break;
1411     case '.':
1412         state = sw_dot_dot;
1413         *u++ = ch;
1414         break;
1415     case '%':
1416         quoted_state = state;
1417         state = sw_quoted;
1418         break;
1419     case '?':
1420         r->args_start = p;
1421         goto args;
1422     case '#':
1423         goto done;
1424     case '+':
1425         r->plus_in_uri = 1;
1426     default:
1427         state = sw_usual;
1428         *u++ = ch;
1429         break;
1430     }
1431
1432     ch = *p++;
1433     break;
1434
1435 case sw_dot_dot:
1436
1437     if (usual[ch >> 5] & (1 << (ch & 0x1f))) {
1438         state = sw_usual;

```

```

1439         *u++ = ch;
1440         ch = *p++;
1441         break;
1442     }
1443
1444     switch (ch) {
1445 #if (NGX_WIN32)
1446     case '\\':
1447 #endif
1448     case '/':
1449         state = sw_slash;
1450         u -= 5;
1451         for ( ;; ) {
1452             if (u < r->uri.data) {
1453                 return NGX\_HTTP\_PARSE\_INVALID\_REQUEST;
1454             }
1455             if (*u == '/') {
1456                 u++;
1457                 break;
1458             }
1459             u--;
1460         }
1461         break;
1462     case '%':
1463         quoted_state = state;
1464         state = sw_quoted;
1465         break;
1466     case '?':
1467         r->args_start = p;
1468         goto args;
1469     case '#':
1470         goto done;
1471     case '+':
1472         r->plus_in_uri = 1;
1473     default:
1474         state = sw_usual;
1475         *u++ = ch;
1476         break;
1477     }
1478
1479     ch = *p++;
1480     break;
1481
1482 case sw_quoted:
1483     r->quoted_uri = 1;
1484
1485     if (ch >= '0' && ch <= '9') {
1486         decoded = (u_char) (ch - '0');
1487         state = sw_quoted_second;
1488         ch = *p++;
1489         break;
1490     }
1491
1492     c = (u_char) (ch | 0x20);
1493     if (c >= 'a' && c <= 'f') {
1494         decoded = (u_char) (c - 'a' + 10);
1495         state = sw_quoted_second;
1496         ch = *p++;
1497         break;
1498     }
1499
1500     return NGX\_HTTP\_PARSE\_INVALID\_REQUEST;
1501
1502 case sw_quoted_second:
1503     if (ch >= '0' && ch <= '9') {
1504         ch = (u_char) ((decoded << 4) + ch - '0');
1505
1506         if (ch == '%' || ch == '#') {
1507             state = sw_usual;
1508             *u++ = ch;
1509             ch = *p++;
1510             break;
1511         }
1512     } else if (ch == '\\0') {
1513         return NGX\_HTTP\_PARSE\_INVALID\_REQUEST;
1514     }

```

```

1515         state = quoted_state;
1516         break;
1517     }
1518 }
1519
1520 c = (u_char) (ch | 0x20);
1521 if (c >= 'a' && c <= 'f') {
1522     ch = (u_char) ((decoded << 4) + c - 'a' + 10);
1523
1524     if (ch == '?') {
1525         state = sw_usual;
1526         *u++ = ch;
1527         ch = *p++;
1528         break;
1529
1530     } else if (ch == '+') {
1531         r->plus_in_uri = 1;
1532     }
1533
1534     state = quoted_state;
1535     break;
1536 }
1537
1538 return NGX\_HTTP\_PARSE\_INVALID\_REQUEST;
1539 }
1540 }
1541
1542 done:
1543
1544     r->uri.len = u - r->uri.data;
1545
1546     if (r->uri_ext) {
1547         r->exten.len = u - r->uri_ext;
1548         r->exten.data = r->uri_ext;
1549     }
1550
1551     r->uri_ext = NULL;
1552
1553     return NGX\_OK;
1554
1555 args:
1556
1557     while (p < r->uri_end) {
1558         if (*p++ != '#') {
1559             continue;
1560         }
1561
1562         r->args.len = p - 1 - r->args_start;
1563         r->args.data = r->args_start;
1564         r->args_start = NULL;
1565
1566         break;
1567     }
1568
1569     r->uri.len = u - r->uri.data;
1570
1571     if (r->uri_ext) {
1572         r->exten.len = u - r->uri_ext;
1573         r->exten.data = r->uri_ext;
1574     }
1575
1576     r->uri_ext = NULL;
1577
1578     return NGX\_OK;
1579 }
1580
1581
1582 ngx\_int\_t
1583 ngx\_http\_parse\_status\_line(ngx\_http\_request\_t *r, ngx\_buf\_t *b,
1584 ngx\_http\_status\_t *status)
1585 {
1586     u_char ch;
1587     u_char *p;
1588     enum {
1589         sw_start = 0,
1590         sw_H,

```



```

1667         r->http_major = ch - '0';
1668         state = sw_major_digit;
1669         break;
1670
1671
1672     /* the major HTTP version or dot */
1673     case sw_major_digit:
1674         if (ch == '.') {
1675             state = sw_first_minor_digit;
1676             break;
1677         }
1678
1679         if (ch < '0' || ch > '9') {
1680             return NGX\_ERROR;
1681         }
1682
1683         r->http_major = r->http_major * 10 + ch - '0';
1684         break;
1685
1686     /* the first digit of minor HTTP version */
1687     case sw_first_minor_digit:
1688         if (ch < '0' || ch > '9') {
1689             return NGX\_ERROR;
1690         }
1691
1692         r->http_minor = ch - '0';
1693         state = sw_minor_digit;
1694         break;
1695
1696     /* the minor HTTP version or the end of the request line */
1697     case sw_minor_digit:
1698         if (ch == ' ') {
1699             state = sw_status;
1700             break;
1701         }
1702
1703         if (ch < '0' || ch > '9') {
1704             return NGX\_ERROR;
1705         }
1706
1707         r->http_minor = r->http_minor * 10 + ch - '0';
1708         break;
1709
1710     /* HTTP status code */
1711     case sw_status:
1712         if (ch == ' ') {
1713             break;
1714         }
1715
1716         if (ch < '0' || ch > '9') {
1717             return NGX\_ERROR;
1718         }
1719
1720         status->code = status->code * 10 + ch - '0';
1721
1722         if (++status->count == 3) {
1723             state = sw_space_after_status;
1724             status->start = p - 2;
1725         }
1726
1727         break;
1728
1729     /* space or end of line */
1730     case sw_space_after_status:
1731         switch (ch) {
1732             case ' ':
1733                 state = sw_status_text;
1734                 break;
1735             case '.': /* IIS may send 403.1, 403.2, etc */
1736                 state = sw_status_text;
1737                 break;
1738             case CR:
1739                 state = sw_almost_done;
1740                 break;
1741             case LF:
1742                 goto done;

```

```

1743         default:
1744             return NGX\_ERROR;
1745         }
1746         break;
1747
1748         /* any text until end of line */
1749         case sw_status_text:
1750             switch (ch) {
1751                 case CR:
1752                     state = sw_almost_done;
1753
1754                     break;
1755                 case LF:
1756                     goto done;
1757             }
1758             break;
1759
1760         /* end of status line */
1761         case sw_almost_done:
1762             status->end = p - 1;
1763             switch (ch) {
1764                 case LF:
1765                     goto done;
1766             default:
1767                 return NGX\_ERROR;
1768             }
1769     }
1770 }
1771
1772 b->pos = p;
1773 r->state = state;
1774
1775 return NGX\_AGAIN;
1776
1777 done:
1778
1779 b->pos = p + 1;
1780
1781 if (status->end == NULL) {
1782     status->end = p;
1783 }
1784
1785 status->http_version = r->http_major * 1000 + r->http_minor;
1786 r->state = sw_start;
1787
1788 return NGX\_OK;
1789 }
1790
1791
1792 ngx\_int\_t
1793 ngx\_http\_parse\_unsafe\_uri(ngx\_http\_request\_t *r, ngx\_str\_t *uri,
1794 ngx\_str\_t *args, ngx\_uint\_t *flags)
1795 {
1796     u_char    ch, *p, *src, *dst;
1797     size_t    len;
1798     ngx\_uint\_t quoted;
1799
1800     len = uri->len;
1801     p = uri->data;
1802     quoted = 0;
1803
1804     if (len == 0 || p[0] == '?') {
1805         goto unsafe;
1806     }
1807
1808     if (p[0] == '.' && len > 1 && p[1] == '.'
1809         && (len == 2 || ngx\_path\_separator(p[2])))
1810     {
1811         goto unsafe;
1812     }
1813
1814     for ( /* void */ ; len; len--) {
1815
1816         ch = *p++;
1817
1818         if (ch == '%') {

```

```

1819         quoted = 1;
1820         continue;
1821     }
1822
1823     if (usual[ch >> 5] & (1 << (ch & 0x1f))) {
1824         continue;
1825     }
1826
1827     if (ch == '?') {
1828         args->len = len - 1;
1829         args->data = p;
1830         uri->len -= len;
1831
1832         break;
1833     }
1834
1835     if (ch == '\0') {
1836         goto unsafe;
1837     }
1838
1839     if (ngx_path_separator(ch) && len > 2) {
1840
1841         /* detect "../" and "../" */
1842
1843         if (p[0] == '.' && p[1] == '.'
1844             && (len == 3 || ngx_path_separator(p[2])))
1845         {
1846             goto unsafe;
1847         }
1848     }
1849 }
1850
1851 if (quoted) {
1852     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1853         "escaped URI: \"%V\"", uri);
1854
1855     src = uri->data;
1856
1857     dst = ngx_pnalloc(r->pool, uri->len);
1858     if (dst == NULL) {
1859         return NGX_ERROR;
1860     }
1861
1862     uri->data = dst;
1863
1864     ngx_unescape_uri(&dst, &src, uri->len, 0);
1865
1866     uri->len = dst - uri->data;
1867
1868     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1869         "unescaped URI: \"%V\"", uri);
1870
1871     len = uri->len;
1872     p = uri->data;
1873
1874     if (p[0] == '.' && len > 1 && p[1] == '.'
1875         && (len == 2 || ngx_path_separator(p[2])))
1876     {
1877         goto unsafe;
1878     }
1879
1880     for ( /* void */ ; len; len-- ) {
1881
1882         ch = *p++;
1883
1884         if (ch == '\0') {
1885             goto unsafe;
1886         }
1887
1888         if (ngx_path_separator(ch) && len > 2) {
1889
1890             /* detect "../" and "../" */
1891
1892             if (p[0] == '.' && p[1] == '.'
1893                 && (len == 3 || ngx_path_separator(p[2])))
1894             {

```



```

1895         goto unsafe;
1896     }
1897 }
1898 }
1899 }
1900
1901 return NGX_OK;
1902
1903 unsafe:
1904
1905 if (*flags & NGX_HTTP_LOG_UNSAFE) {
1906     ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1907                 "unsafe URI \"%V\" was detected", uri);
1908 }
1909
1910 return NGX_ERROR;
1911 }
1912
1913
1914 ngx_int_t
1915 ngx_http_parse_multi_header_lines(ngx_array_t *headers, ngx_str_t *name,
1916 ngx_str_t *value)
1917 {
1918     ngx_uint_t i;
1919     u_char *start, *last, *end, ch;
1920     ngx_table_elt_t **h;
1921
1922     h = headers->elts;
1923
1924     for (i = 0; i < headers->nelts; i++) {
1925
1926         ngx_log_debug2(NGX_LOG_DEBUG_HTTP, headers->pool->log, 0,
1927                       "parse header: \"%V: %V\"", &h[i]->key, &h[i]->value);
1928
1929         if (name->len > h[i]->value.len) {
1930             continue;
1931         }
1932
1933         start = h[i]->value.data;
1934         end = h[i]->value.data + h[i]->value.len;
1935
1936         while (start < end) {
1937
1938             if (ngx_strncasecmp(start, name->data, name->len) != 0) {
1939                 goto skip;
1940             }
1941
1942             for (start += name->len; start < end && *start == ' '; start++) {
1943                 /* void */
1944             }
1945
1946             if (value == NULL) {
1947                 if (start == end || *start == ',') {
1948                     return i;
1949                 }
1950
1951                 goto skip;
1952             }
1953
1954             if (start == end || *start++ != '=') {
1955                 /* the invalid header value */
1956                 goto skip;
1957             }
1958
1959             while (start < end && *start == ' ') { start++; }
1960
1961             for (last = start; last < end && *last != ';'; last++) {
1962                 /* void */
1963             }
1964
1965             value->len = last - start;
1966             value->data = start;
1967
1968             return i;
1969
1970 skip:

```

```

1971         while (start < end) {
1972             ch = *start++;
1973             if (ch == ';' || ch == ',') {
1974                 break;
1975             }
1976         }
1977     }
1978
1979     while (start < end && *start == ' ') { start++; }
1980 }
1981 }
1982
1983 return NGX\_DECLINED;
1984 }
1985
1986
1987 ngx\_int\_t
1988 ngx\_http\_parse\_set\_cookie\_lines(ngx\_array\_t *headers, ngx\_str\_t *name,
1989 ngx\_str\_t *value)
1990 {
1991     ngx\_uint\_t i;
1992     u_char *start, *last, *end;
1993     ngx\_table\_elt\_t **h;
1994
1995     h = headers->elts;
1996
1997     for (i = 0; i < headers->nelts; i++) {
1998
1999         ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, headers->pool->log, 0,
2000             "parse header: \"%V: %V\"", &h[i]->key, &h[i]->value);
2001
2002         if (name->len >= h[i]->value.len) {
2003             continue;
2004         }
2005
2006         start = h[i]->value.data;
2007         end = h[i]->value.data + h[i]->value.len;
2008
2009         if (ngx\_strncasecmp(start, name->data, name->len) != 0) {
2010             continue;
2011         }
2012
2013         for (start += name->len; start < end && *start == ' '; start++) {
2014             /* void */
2015         }
2016
2017         if (start == end || *start++ != '=') {
2018             /* the invalid header value */
2019             continue;
2020         }
2021
2022         while (start < end && *start == ' ') { start++; }
2023
2024         for (last = start; last < end && *last != ';'; last++) {
2025             /* void */
2026         }
2027
2028         value->len = last - start;
2029         value->data = start;
2030
2031         return i;
2032     }
2033
2034     return NGX\_DECLINED;
2035 }
2036
2037
2038 ngx\_int\_t
2039 ngx\_http\_arg(ngx\_http\_request\_t *r, u_char *name, size_t len, ngx\_str\_t *value)
2040 {
2041     u_char *p, *last;
2042
2043     if (r->args.len == 0) {
2044         return NGX\_DECLINED;
2045     }
2046

```

```

2047     p = r->args.data;
2048     last = p + r->args.len;
2049
2050     for ( /* void */ ; p < last; p++) {
2051         /* we need '=' after name, so drop one char from last */
2052
2053         p = ngx_strlcasestrn(p, last - 1, name, len - 1);
2054
2055         if (p == NULL) {
2056             return NGX_DECLINED;
2057         }
2058
2059         if ((p == r->args.data || *(p - 1) == '&') && *(p + len) == '=') {
2060             value->data = p + len + 1;
2061
2062             p = ngx_strlchr(p, last, '&');
2063
2064             if (p == NULL) {
2065                 p = r->args.data + r->args.len;
2066             }
2067
2068             value->len = p - value->data;
2069
2070             return NGX_OK;
2071         }
2072     }
2073
2074     return NGX_DECLINED;
2075 }
2076
2077 void
2078 ngx_http_split_args(ngx_http_request_t *r, ngx_str_t *uri, ngx_str_t *args)
2079 {
2080     u_char *p, *last;
2081
2082     last = uri->data + uri->len;
2083
2084     p = ngx_strlchr(uri->data, last, '?');
2085
2086     if (p) {
2087         uri->len = p - uri->data;
2088         p++;
2089         args->len = last - p;
2090         args->data = p;
2091     } else {
2092         args->len = 0;
2093     }
2094 }
2095
2096 ngx_int_t
2097 ngx_http_parse_chunked(ngx_http_request_t *r, ngx_buf_t *b,
2098 ngx_http_chunked_t *ctx)
2099 {
2100     u_char *pos, ch, c;
2101     ngx_int_t rc;
2102     enum {
2103         sw_chunk_start = 0,
2104         sw_chunk_size,
2105         sw_chunk_extension,
2106         sw_chunk_extension_almost_done,
2107         sw_chunk_data,
2108         sw_after_data,
2109         sw_after_data_almost_done,
2110         sw_last_chunk_extension,
2111         sw_last_chunk_extension_almost_done,
2112         sw_trailer,
2113         sw_trailer_almost_done,
2114         sw_trailer_header,
2115         sw_trailer_header_almost_done
2116     } state;

```

```

2123 state = ctx->state;
2124
2125 if (state == sw_chunk_data && ctx->size == 0) {
2126     state = sw_after_data;
2127 }
2128
2129 rc = NGX\_AGAIN;
2130
2131 for (pos = b->pos; pos < b->last; pos++) {
2132
2133     ch = *pos;
2134
2135     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
2136                 "http chunked byte: %02Xd s:%d", ch, state);
2137
2138     switch (state) {
2139
2140     case sw_chunk_start:
2141         if (ch >= '0' && ch <= '9') {
2142             state = sw_chunk_size;
2143             ctx->size = ch - '0';
2144             break;
2145         }
2146
2147         c = (u_char) (ch | 0x20);
2148
2149         if (c >= 'a' && c <= 'f') {
2150             state = sw_chunk_size;
2151             ctx->size = c - 'a' + 10;
2152             break;
2153         }
2154
2155         goto invalid;
2156
2157     case sw_chunk_size:
2158         if (ch >= '0' && ch <= '9') {
2159             ctx->size = ctx->size * 16 + (ch - '0');
2160             break;
2161         }
2162
2163         c = (u_char) (ch | 0x20);
2164
2165         if (c >= 'a' && c <= 'f') {
2166             ctx->size = ctx->size * 16 + (c - 'a' + 10);
2167             break;
2168         }
2169
2170         if (ctx->size == 0) {
2171
2172             switch (ch) {
2173             case CR:
2174                 state = sw_last_chunk_extension_almost_done;
2175                 break;
2176             case LF:
2177                 state = sw_trailer;
2178                 break;
2179             case ';':
2180             case ' ':
2181             case '\t':
2182                 state = sw_last_chunk_extension;
2183                 break;
2184             default:
2185                 goto invalid;
2186             }
2187
2188             break;
2189         }
2190
2191         switch (ch) {
2192         case CR:
2193             state = sw_chunk_extension_almost_done;
2194             break;
2195         case LF:
2196             state = sw_chunk_data;
2197             break;
2198         case ';':

```

```

2199     case ' ':
2200     case '\t':
2201         state = sw_chunk_extension;
2202         break;
2203     default:
2204         goto invalid;
2205     }
2206
2207     break;
2208
2209 case sw_chunk_extension:
2210     switch (ch) {
2211     case CR:
2212         state = sw_chunk_extension_almost_done;
2213         break;
2214     case LF:
2215         state = sw_chunk_data;
2216     }
2217     break;
2218
2219 case sw_chunk_extension_almost_done:
2220     if (ch == LF) {
2221         state = sw_chunk_data;
2222         break;
2223     }
2224     goto invalid;
2225
2226 case sw_chunk_data:
2227     rc = NGX_OK;
2228     goto data;
2229
2230 case sw_after_data:
2231     switch (ch) {
2232     case CR:
2233         state = sw_after_data_almost_done;
2234         break;
2235     case LF:
2236         state = sw_chunk_start;
2237     }
2238     break;
2239
2240 case sw_after_data_almost_done:
2241     if (ch == LF) {
2242         state = sw_chunk_start;
2243         break;
2244     }
2245     goto invalid;
2246
2247 case sw_last_chunk_extension:
2248     switch (ch) {
2249     case CR:
2250         state = sw_last_chunk_extension_almost_done;
2251         break;
2252     case LF:
2253         state = sw_trailer;
2254     }
2255     break;
2256
2257 case sw_last_chunk_extension_almost_done:
2258     if (ch == LF) {
2259         state = sw_trailer;
2260         break;
2261     }
2262     goto invalid;
2263
2264 case sw_trailer:
2265     switch (ch) {
2266     case CR:
2267         state = sw_trailer_almost_done;
2268         break;
2269     case LF:
2270         goto done;
2271     default:
2272         state = sw_trailer_header;
2273     }
2274     break;

```

```

2275
2276     case sw_trailer_almost_done:
2277         if (ch == LF) {
2278             goto done;
2279         }
2280         goto invalid;
2281
2282     case sw_trailer_header:
2283         switch (ch) {
2284             case CR:
2285                 state = sw_trailer_header_almost_done;
2286                 break;
2287             case LF:
2288                 state = sw_trailer;
2289         }
2290         break;
2291
2292     case sw_trailer_header_almost_done:
2293         if (ch == LF) {
2294             state = sw_trailer;
2295             break;
2296         }
2297         goto invalid;
2298
2299     }
2300 }
2301
2302 data:
2303
2304     ctx->state = state;
2305     b->pos = pos;
2306
2307     switch (state) {
2308
2309     case sw_chunk_start:
2310         ctx->length = 3 /* "LF LF */;
2311         break;
2312     case sw_chunk_size:
2313         ctx->length = 1 /* LF */
2314             + (ctx->size ? ctx->size + 4 /* LF "0" LF LF */
2315                 : 1 /* LF */);
2316         break;
2317     case sw_chunk_extension:
2318     case sw_chunk_extension_almost_done:
2319         ctx->length = 1 /* LF */ + ctx->size + 4 /* LF "0" LF LF */;
2320         break;
2321     case sw_chunk_data:
2322         ctx->length = ctx->size + 4 /* LF "0" LF LF */;
2323         break;
2324     case sw_after_data:
2325     case sw_after_data_almost_done:
2326         ctx->length = 4 /* LF "0" LF LF */;
2327         break;
2328     case sw_last_chunk_extension:
2329     case sw_last_chunk_extension_almost_done:
2330         ctx->length = 2 /* LF LF */;
2331         break;
2332     case sw_trailer:
2333     case sw_trailer_almost_done:
2334         ctx->length = 1 /* LF */;
2335         break;
2336     case sw_trailer_header:
2337     case sw_trailer_header_almost_done:
2338         ctx->length = 2 /* LF LF */;
2339         break;
2340
2341     }
2342
2343     if (ctx->size < 0 || ctx->length < 0) {
2344         goto invalid;
2345     }
2346
2347     return rc;
2348
2349 done:
2350

```

```
2351     ctx->state = 0;
2352     b->pos = pos + 1;
2353
2354     return NGX\_DONE;
2355
2356 invalid:
2357
2358     return NGX\_ERROR;
2359 }
```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_md5.c - nginx-1.7.10

### Functions defined

- [ngx\\_md5\\_body](#)
- [ngx\\_md5\\_final](#)
- [ngx\\_md5\\_init](#)
- [ngx\\_md5\\_update](#)

### Macros defined

- [F](#)
- [G](#)
- [GET](#)
- [GET](#)
- [H](#)
- [I](#)
- [SET](#)
- [SET](#)
- [STEP](#)

### Source code

```
1
2 /*
3  * An internal implementation, based on Alexander Peslyak's
4  * public domain implementation:
5  * http://openwall.info/wiki/people/solar/software/public-domain-source-code/md5
6  * It is not expected to be optimal and is used only
7  * if no MD5 implementation was found in system.
8  */
9
10
11 #include <ngx_config.h>
12 #include <ngx_core.h>
13 #include <ngx_md5.h>
14
15
16 #if !(NGX_HAVE_MD5)
17
18 static const u_char *ngx_md5_body(ngx_md5_t *ctx, const u_char *data,
19     size_t size);
20
21
22 void
23 ngx_md5_init(ngx_md5_t *ctx)
24 {
25     ctx->a = 0x67452301;
26     ctx->b = 0xefcdab89;
27     ctx->c = 0x98badcfe;
28     ctx->d = 0x10325476;
29
30     ctx->bytes = 0;
31 }
32
```



```

33
34 void
35 ngx_md5_update(ngx_md5_t *ctx, const void *data, size_t size)
36 {
37     size_t used, free;
38
39     used = (size_t) (ctx->bytes & 0x3f);
40     ctx->bytes += size;
41
42     if (used) {
43         free = 64 - used;
44
45         if (size < free) {
46             ngx_memcpy(&ctx->buffer[used], data, size);
47             return;
48         }
49
50         ngx_memcpy(&ctx->buffer[used], data, free);
51         data = (u_char *) data + free;
52         size -= free;
53         (void) ngx_md5_body(ctx, ctx->buffer, 64);
54     }
55
56     if (size >= 64) {
57         data = ngx_md5_body(ctx, data, size & ~(size_t) 0x3f);
58         size &= 0x3f;
59     }
60
61     ngx_memcpy(ctx->buffer, data, size);
62 }
63
64
65 void
66 ngx_md5_final(u_char result[16], ngx_md5_t *ctx)
67 {
68     size_t used, free;
69
70     used = (size_t) (ctx->bytes & 0x3f);
71
72     ctx->buffer[used++] = 0x80;
73
74     free = 64 - used;
75
76     if (free < 8) {
77         ngx_memzero(&ctx->buffer[used], free);
78         (void) ngx_md5_body(ctx, ctx->buffer, 64);
79         used = 0;
80         free = 64;
81     }
82
83     ngx_memzero(&ctx->buffer[used], free - 8);
84
85     ctx->bytes <= 3;
86     ctx->buffer[56] = (u_char) ctx->bytes;
87     ctx->buffer[57] = (u_char) (ctx->bytes >> 8);
88     ctx->buffer[58] = (u_char) (ctx->bytes >> 16);
89     ctx->buffer[59] = (u_char) (ctx->bytes >> 24);
90     ctx->buffer[60] = (u_char) (ctx->bytes >> 32);
91     ctx->buffer[61] = (u_char) (ctx->bytes >> 40);
92     ctx->buffer[62] = (u_char) (ctx->bytes >> 48);
93     ctx->buffer[63] = (u_char) (ctx->bytes >> 56);
94
95     (void) ngx_md5_body(ctx, ctx->buffer, 64);
96
97     result[0] = (u_char) ctx->a;
98     result[1] = (u_char) (ctx->a >> 8);
99     result[2] = (u_char) (ctx->a >> 16);
100    result[3] = (u_char) (ctx->a >> 24);
101    result[4] = (u_char) ctx->b;
102    result[5] = (u_char) (ctx->b >> 8);
103    result[6] = (u_char) (ctx->b >> 16);
104    result[7] = (u_char) (ctx->b >> 24);
105    result[8] = (u_char) ctx->c;
106    result[9] = (u_char) (ctx->c >> 8);
107    result[10] = (u_char) (ctx->c >> 16);
108    result[11] = (u_char) (ctx->c >> 24);

```

```

1109     result[12] = (u_char) ctx->d;
1110     result[13] = (u_char) (ctx->d >> 8);
1111     result[14] = (u_char) (ctx->d >> 16);
1112     result[15] = (u_char) (ctx->d >> 24);
1113
1114     ngx_memzero(ctx, sizeof(*ctx));
1115 }
1116
1117
1118 /*
1119  * The basic MD5 functions.
1120  *
1121  * F and G are optimized compared to their RFC 1321 definitions for
1122  * architectures that lack an AND-NOT instruction, just like in
1123  * Colin Plumb's implementation.
1124  */
1125
1126 #define F(x, y, z) ((z) ^ ((x) & ((y) ^ (z))))
1127 #define G(x, y, z) ((y) ^ ((z) & ((x) ^ (y))))
1128 #define H(x, y, z) ((x) ^ (y) ^ (z))
1129 #define I(x, y, z) ((y) ^ ((x) | ~(z)))
1130
1131 /*
1132  * The MD5 transformation for all four rounds.
1133  */
1134
1135 #define STEP(f, a, b, c, d, x, t, s) \
1136     (a) += f((b), (c), (d)) + (x) + (t); \
1137     (a) = (((a) << (s)) | (((a) & 0xffffffff) >> (32 - (s)))); \
1138     (a) += (b)
1139
1140 /*
1141  * SET() reads 4 input bytes in little-endian byte order and stores them
1142  * in a properly aligned word in host byte order.
1143  *
1144  * The check for little-endian architectures that tolerate unaligned
1145  * memory accesses is just an optimization. Nothing will break if it
1146  * does not work.
1147  */
1148
1149 #if (NGX_HAVE_LITTLE_ENDIAN && NGX_HAVE_NONALIGNED)
1150
1151 #define SET(n)      (*(uint32_t *) &p[n * 4])
1152 #define GET(n)      (*(uint32_t *) &p[n * 4])
1153
1154 #else
1155
1156 #define SET(n) \
1157     (block[n] = \
1158     (uint32_t) p[n * 4] | \
1159     ((uint32_t) p[n * 4 + 1] << 8) | \
1160     ((uint32_t) p[n * 4 + 2] << 16) | \
1161     ((uint32_t) p[n * 4 + 3] << 24))
1162
1163 #define GET(n)      block[n]
1164
1165 #endif
1166
1167
1168 /*
1169  * This processes one or more 64-byte data blocks, but does not update
1170  * the bit counters. There are no alignment requirements.
1171  */
1172
1173 static const u_char *
1174 ngx_md5_body(ngx_md5_t *ctx, const u_char *data, size_t size)
1175 {
1176     uint32_t      a, b, c, d;
1177     uint32_t      saved_a, saved_b, saved_c, saved_d;
1178     const u_char  *p;
1179     #if !(NGX_HAVE_LITTLE_ENDIAN && NGX_HAVE_NONALIGNED)
1180     uint32_t      block[16];
1181     #endif
1182
1183     p = data;
1184

```

```

185 a = ctx->a;
186 b = ctx->b;
187 c = ctx->c;
188 d = ctx->d;
189
190 do {
191     saved_a = a;
192     saved_b = b;
193     saved_c = c;
194     saved_d = d;
195
196     /* Round 1 */
197
198     STEP(F, a, b, c, d, SET(0), 0xd76aa478, 7);
199     STEP(F, d, a, b, c, SET(1), 0xe8c7b756, 12);
200     STEP(F, c, d, a, b, SET(2), 0x242070db, 17);
201     STEP(F, b, c, d, a, SET(3), 0xc1bdceee, 22);
202     STEP(F, a, b, c, d, SET(4), 0xf57c0faf, 7);
203     STEP(F, d, a, b, c, SET(5), 0x4787c62a, 12);
204     STEP(F, c, d, a, b, SET(6), 0xa8304613, 17);
205     STEP(F, b, c, d, a, SET(7), 0xfd469501, 22);
206     STEP(F, a, b, c, d, SET(8), 0x698098d8, 7);
207     STEP(F, d, a, b, c, SET(9), 0x8b44f7af, 12);
208     STEP(F, c, d, a, b, SET(10), 0xffff5bb1, 17);
209     STEP(F, b, c, d, a, SET(11), 0x895cd7be, 22);
210     STEP(F, a, b, c, d, SET(12), 0x6b901122, 7);
211     STEP(F, d, a, b, c, SET(13), 0xfd987193, 12);
212     STEP(F, c, d, a, b, SET(14), 0xa679438e, 17);
213     STEP(F, b, c, d, a, SET(15), 0x49b40821, 22);
214
215     /* Round 2 */
216
217     STEP(G, a, b, c, d, GET(1), 0xf61e2562, 5);
218     STEP(G, d, a, b, c, GET(6), 0xc040b340, 9);
219     STEP(G, c, d, a, b, GET(11), 0x265e5a51, 14);
220     STEP(G, b, c, d, a, GET(0), 0xe9b6c7aa, 20);
221     STEP(G, a, b, c, d, GET(5), 0xd62f105d, 5);
222     STEP(G, d, a, b, c, GET(10), 0x02441453, 9);
223     STEP(G, c, d, a, b, GET(15), 0xd8a1e681, 14);
224     STEP(G, b, c, d, a, GET(4), 0xe7d3fbc8, 20);
225     STEP(G, a, b, c, d, GET(9), 0x21e1cde6, 5);
226     STEP(G, d, a, b, c, GET(14), 0xc33707d6, 9);
227     STEP(G, c, d, a, b, GET(3), 0xf4d50d87, 14);
228     STEP(G, b, c, d, a, GET(8), 0x455a14ed, 20);
229     STEP(G, a, b, c, d, GET(13), 0xa9e3e905, 5);
230     STEP(G, d, a, b, c, GET(2), 0xfcefa3f8, 9);
231     STEP(G, c, d, a, b, GET(7), 0x676f02d9, 14);
232     STEP(G, b, c, d, a, GET(12), 0x8d2a4c8a, 20);
233
234     /* Round 3 */
235
236     STEP(H, a, b, c, d, GET(5), 0xffffa3942, 4);
237     STEP(H, d, a, b, c, GET(8), 0x8771f681, 11);
238     STEP(H, c, d, a, b, GET(11), 0x6d9d6122, 16);
239     STEP(H, b, c, d, a, GET(14), 0xfde5380c, 23);
240     STEP(H, a, b, c, d, GET(1), 0xa4beea44, 4);
241     STEP(H, d, a, b, c, GET(4), 0x4bdecfa9, 11);
242     STEP(H, c, d, a, b, GET(7), 0xf6bb4b60, 16);
243     STEP(H, b, c, d, a, GET(10), 0xbebfb70, 23);
244     STEP(H, a, b, c, d, GET(13), 0x289b7ec6, 4);
245     STEP(H, d, a, b, c, GET(0), 0xeea127fa, 11);
246     STEP(H, c, d, a, b, GET(3), 0xd4ef3085, 16);
247     STEP(H, b, c, d, a, GET(6), 0x04881d05, 23);
248     STEP(H, a, b, c, d, GET(9), 0xd9d4d039, 4);
249     STEP(H, d, a, b, c, GET(12), 0xe6db99e5, 11);
250     STEP(H, c, d, a, b, GET(15), 0x1fa27cf8, 16);
251     STEP(H, b, c, d, a, GET(2), 0xc4ac5665, 23);
252
253     /* Round 4 */
254
255     STEP(I, a, b, c, d, GET(0), 0xf4292244, 6);
256     STEP(I, d, a, b, c, GET(7), 0x432aff97, 10);
257     STEP(I, c, d, a, b, GET(14), 0xab9423a7, 15);
258     STEP(I, b, c, d, a, GET(5), 0xfc93a039, 21);
259     STEP(I, a, b, c, d, GET(12), 0x655b59c3, 6);
260     STEP(I, d, a, b, c, GET(3), 0x8f0ccc92, 10);

```

```
261     STEP(I, c, d, a, b, GET(10), 0xffeff47d, 15);
262     STEP(I, b, c, d, a, GET(1), 0x85845dd1, 21);
263     STEP(I, a, b, c, d, GET(8), 0x6fa87e4f, 6);
264     STEP(I, d, a, b, c, GET(15), 0xfe2ce6e0, 10);
265     STEP(I, c, d, a, b, GET(6), 0xa3014314, 15);
266     STEP(I, b, c, d, a, GET(13), 0x4e0811a1, 21);
267     STEP(I, a, b, c, d, GET(4), 0xf7537e82, 6);
268     STEP(I, d, a, b, c, GET(11), 0xbd3af235, 10);
269     STEP(I, c, d, a, b, GET(2), 0x2ad7d2bb, 15);
270     STEP(I, b, c, d, a, GET(9), 0xeb86d391, 21);
271
272     a += saved_a;
273     b += saved_b;
274     c += saved_c;
275     d += saved_d;
276
277     p += 64;
278
279 } while (size -= 64);
280
281 ctx->a = a;
282 ctx->b = b;
283 ctx->c = c;
284 ctx->d = d;
285
286 return p;
287 }
288
289 #endif
```

[One Level Up](#)

[Top Level](#)

# src/core/nginx\_md5.h - nginx-1.7.10

## Data types defined

- [ngx\\_md5\\_t](#)
- [ngx\\_md5\\_t](#)

## Macros defined

- [\\_NGX\\_MD5\\_H\\_INCLUDED](#)
- [ngx\\_md5\\_final](#)
- [ngx\\_md5\\_final](#)
- [ngx\\_md5\\_init](#)
- [ngx\\_md5\\_init](#)
- [ngx\\_md5\\_update](#)
- [ngx\\_md5\\_update](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_MD5_H_INCLUDED_
9 #define _NGX_MD5_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 #if (NGX_HAVE_MD5)
17
18 #if (NGX_HAVE_OPENSSL_MD5_H)
19 #include <openssl/md5.h>
20 #else
21 #include <md5.h>
22 #endif
23
24
25 typedef MD5_CTX  ngx_md5_t;
26
27
28 #if (NGX_OPENSSL_MD5)
29
30 #define ngx_md5_init    MD5_Init
31 #define ngx_md5_update  MD5_Update
32 #define ngx_md5_final  MD5_Final
33
34 #else
35
36 #define ngx_md5_init    MD5Init
37 #define ngx_md5_update  MD5Update
38 #define ngx_md5_final  MD5Final
39
40 #endif
41
```

```
42
43 #else /* !NGX_HAVE_MD5 */
44
45
46 typedef struct {
47     uint64_t  bytes;
48     uint32_t  a, b, c, d;
49     u_char    buffer[64];
50 } ngx_md5_t;
51
52
53 void ngx_md5_init(ngx_md5_t *ctx);
54 void ngx_md5_update(ngx_md5_t *ctx, const void *data, size_t size);
55 void ngx_md5_final(u_char result[16], ngx_md5_t *ctx);
56
57
58 #endif
59
60 #endif /* NGX_MD5_H_INCLUDED */
```

[One Level Up](#)

[Top Level](#)

## src/http/nginx\_http\_spdy.c - nginx-1.7.10

### Global variables defined

- [ngx\\_http\\_spdy\\_dict](#)
- [ngx\\_http\\_spdy\\_request\\_headers](#)

### Data types defined

- [ngx\\_http\\_spdy\\_request\\_header\\_t](#)

### Functions defined

- [ngx\\_http\\_spdy\\_adjust\\_windows](#)
- [ngx\\_http\\_spdy\\_alloc\\_large\\_header\\_buffer](#)
- [ngx\\_http\\_spdy\\_close\\_stream](#)
- [ngx\\_http\\_spdy\\_close\\_stream\\_handler](#)
- [ngx\\_http\\_spdy\\_construct\\_request\\_line](#)
- [ngx\\_http\\_spdy\\_create\\_stream](#)
- [ngx\\_http\\_spdy\\_ctl\\_frame\\_handler](#)
- [ngx\\_http\\_spdy\\_finalize\\_connection](#)
- [ngx\\_http\\_spdy\\_get\\_ctl\\_frame](#)
- [ngx\\_http\\_spdy\\_get\\_stream\\_by\\_id](#)
- [ngx\\_http\\_spdy\\_handle\\_connection](#)
- [ngx\\_http\\_spdy\\_handle\\_connection\\_handler](#)
- [ngx\\_http\\_spdy\\_handle\\_request\\_header](#)
- [ngx\\_http\\_spdy\\_init](#)
- [ngx\\_http\\_spdy\\_init\\_request\\_body](#)
- [ngx\\_http\\_spdy\\_keepalive\\_handler](#)
- [ngx\\_http\\_spdy\\_parse\\_header](#)
- [ngx\\_http\\_spdy\\_parse\\_host](#)
- [ngx\\_http\\_spdy\\_parse\\_method](#)
- [ngx\\_http\\_spdy\\_parse\\_path](#)
- [ngx\\_http\\_spdy\\_parse\\_scheme](#)
- [ngx\\_http\\_spdy\\_parse\\_version](#)
- [ngx\\_http\\_spdy\\_pool\\_cleanup](#)
- [ngx\\_http\\_spdy\\_proxy\\_protocol](#)

- [ngx\\_http\\_spdy\\_read\\_handler](#)
- [ngx\\_http\\_spdy\\_read\\_request\\_body](#)
- [ngx\\_http\\_spdy\\_request\\_headers\\_init](#)
- [ngx\\_http\\_spdy\\_run\\_request](#)
- [ngx\\_http\\_spdy\\_send\\_output\\_queue](#)
- [ngx\\_http\\_spdy\\_send\\_rst\\_stream](#)
- [ngx\\_http\\_spdy\\_send\\_settings](#)
- [ngx\\_http\\_spdy\\_send\\_window\\_update](#)
- [ngx\\_http\\_spdy\\_settings\\_frame\\_handler](#)
- [ngx\\_http\\_spdy\\_state\\_complete](#)
- [ngx\\_http\\_spdy\\_state\\_data](#)
- [ngx\\_http\\_spdy\\_state\\_head](#)
- [ngx\\_http\\_spdy\\_state\\_headers](#)
- [ngx\\_http\\_spdy\\_state\\_headers\\_error](#)
- [ngx\\_http\\_spdy\\_state\\_headers\\_skip](#)
- [ngx\\_http\\_spdy\\_state\\_inflate\\_error](#)
- [ngx\\_http\\_spdy\\_state\\_internal\\_error](#)
- [ngx\\_http\\_spdy\\_state\\_ping](#)
- [ngx\\_http\\_spdy\\_state\\_protocol\\_error](#)
- [ngx\\_http\\_spdy\\_state\\_read\\_data](#)
- [ngx\\_http\\_spdy\\_state\\_rst\\_stream](#)
- [ngx\\_http\\_spdy\\_state\\_save](#)
- [ngx\\_http\\_spdy\\_state\\_settings](#)
- [ngx\\_http\\_spdy\\_state\\_skip](#)
- [ngx\\_http\\_spdy\\_state\\_syn\\_stream](#)
- [ngx\\_http\\_spdy\\_state\\_window\\_update](#)
- [ngx\\_http\\_spdy\\_terminate\\_stream](#)
- [ngx\\_http\\_spdy\\_write\\_handler](#)
- [ngx\\_http\\_spdy\\_zalloc](#)
- [ngx\\_http\\_spdy\\_zfree](#)

## Macros defined

- [NGX\\_SPDY\\_CANCEL](#)
- [NGX\\_SPDY\\_CONNECTION\\_WINDOW](#)



- [NGX\\_SPDY\\_CTL\\_FRAME\\_BUFFER\\_SIZE](#)
- [NGX\\_SPDY\\_FLOW\\_CONTROL\\_ERROR](#)
- [NGX\\_SPDY\\_FRAME\\_TOO\\_LARGE](#)
- [NGX\\_SPDY\\_INIT\\_STREAM\\_WINDOW](#)
- [NGX\\_SPDY\\_INTERNAL\\_ERROR](#)
- [NGX\\_SPDY\\_INVALID\\_STREAM](#)
- [NGX\\_SPDY\\_MAX\\_WINDOW](#)
- [NGX\\_SPDY\\_PROTOCOL\\_ERROR](#)
- [NGX\\_SPDY\\_REFUSED\\_STREAM](#)
- [NGX\\_SPDY\\_REQUEST\\_HEADERS](#)
- [NGX\\_SPDY\\_SETTINGS\\_FLAG\\_PERSIST](#)
- [NGX\\_SPDY\\_SETTINGS\\_FLAG\\_PERSISTED](#)
- [NGX\\_SPDY\\_SETTINGS\\_INIT\\_WINDOW](#)
- [NGX\\_SPDY\\_SETTINGS\\_MAX\\_STREAMS](#)
- [NGX\\_SPDY\\_SKIP\\_HEADERS\\_BUFFER\\_SIZE](#)
- [NGX\\_SPDY\\_STREAM\\_ALREADY\\_CLOSED](#)
- [NGX\\_SPDY\\_STREAM\\_IN\\_USE](#)
- [NGX\\_SPDY\\_STREAM\\_WINDOW](#)
- [NGX\\_SPDY\\_UNSUPPORTED\\_VERSION](#)
- [ngx\\_http\\_spdy\\_stream\\_index](#)
- [ngx\\_http\\_spdy\\_streams\\_index\\_size](#)
- [ngx\\_spdy\\_ctl\\_frame\\_check](#)
- [ngx\\_spdy\\_ctl\\_frame\\_type](#)
- [ngx\\_spdy\\_data\\_frame\\_check](#)
- [ngx\\_spdy\\_frame\\_flags](#)
- [ngx\\_spdy\\_frame\\_id](#)
- [ngx\\_spdy\\_frame\\_length](#)
- [ngx\\_spdy\\_frame\\_parse\\_delta](#)
- [ngx\\_spdy\\_frame\\_parse\\_sid](#)
- [ngx\\_spdy\\_frame\\_parse\\_uint16](#)
- [ngx\\_spdy\\_frame\\_parse\\_uint16](#)
- [ngx\\_spdy\\_frame\\_parse\\_uint32](#)
- [ngx\\_spdy\\_frame\\_parse\\_uint32](#)

- [ngx\\_str5cmp](#)
- [ngx\\_str5cmp](#)

## Source code

```

1
2 /*
3  * Copyright (C) Nginx, Inc.
4  * Copyright (C) Valentin V. Bartenev
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11 #include <ngx\_http\_spdy\_module.h>
12
13 #include <zlib.h>
14
15
16 #if (NGX_HAVE_LITTLE_ENDIAN && NGX_HAVE_NONALIGNED)
17
18 #define ngx_str5cmp(m, c0, c1, c2, c3, c4)           \
19     *(uint32_t *) m == (c3 << 24 | c2 << 16 | c1 << 8 | c0) \
20     && m[4] == c4
21
22 #else
23
24 #define ngx_str5cmp(m, c0, c1, c2, c3, c4)           \
25     m[0] == c0 && m[1] == c1 && m[2] == c2 && m[3] == c3 && m[4] == c4
26
27 #endif
28
29
30 #if (NGX_HAVE_NONALIGNED)
31
32 #define ngx_spdy_frame_parse_uint16(p) ntohs(*(uint16_t *) (p))
33 #define ngx_spdy_frame_parse_uint32(p) ntohl(*(uint32_t *) (p))
34
35 #else
36
37 #define ngx_spdy_frame_parse_uint16(p) ((p)[0] << 8 | (p)[1])
38 #define ngx_spdy_frame_parse_uint32(p)           \
39     ((p)[0] << 24 | (p)[1] << 16 | (p)[2] << 8 | (p)[3])
40
41 #endif
42
43 #define ngx_spdy_frame_parse_sid(p)           \
44     (ngx\_spdy\_frame\_parse\_uint32(p) & 0x7fffffff)
45 #define ngx_spdy_frame_parse_delta(p)           \
46     (ngx\_spdy\_frame\_parse\_uint32(p) & 0x7fffffff)
47
48
49 #define ngx_spdy_ctl_frame_check(h)           \
50     (((h) & 0xffff0000) == ngx\_spdy\_ctl\_frame\_head(0))
51 #define ngx_spdy_data_frame_check(h)           \
52     (!(h) & (uint32_t) NGX\_SPDY\_CTL\_BIT << 31))
53
54 #define ngx_spdy_ctl_frame_type(h) ((h) & 0x0000ffff)
55 #define ngx_spdy_frame_flags(p) ((p) >> 24)
56 #define ngx_spdy_frame_length(p) ((p) & 0x00ffffff)
57 #define ngx_spdy_frame_id(p) ((p) & 0x00ffffff)
58
59
60 #define NGX_SPDY_SKIP_HEADERS_BUFFER_SIZE 4096
61 #define NGX_SPDY_CTL_FRAME_BUFFER_SIZE 16
62
63 #define NGX_SPDY_PROTOCOL_ERROR 1
64 #define NGX_SPDY_INVALID_STREAM 2
65 #define NGX_SPDY_REFUSED_STREAM 3
66 #define NGX_SPDY_UNSUPPORTED_VERSION 4
67 #define NGX_SPDY_CANCEL 5
68 #define NGX_SPDY_INTERNAL_ERROR 6

```

```

69 #define NGX_SPDY_FLOW_CONTROL_ERROR 7
70 #define NGX_SPDY_STREAM_IN_USE 8
71 #define NGX_SPDY_STREAM_ALREADY_CLOSED 9
72 /* deprecated 10 */
73 #define NGX_SPDY_FRAME_TOO_LARGE 11
74
75 #define NGX_SPDY_SETTINGS_MAX_STREAMS 4
76 #define NGX_SPDY_SETTINGS_INIT_WINDOW 7
77
78 #define NGX_SPDY_SETTINGS_FLAG_PERSIST 0x01
79 #define NGX_SPDY_SETTINGS_FLAG_PERSISTED 0x02
80
81 #define NGX_SPDY_MAX_WINDOW NGX_MAX_INT32_VALUE
82 #define NGX_SPDY_CONNECTION_WINDOW 65536
83 #define NGX_SPDY_INIT_STREAM_WINDOW 65536
84 #define NGX_SPDY_STREAM_WINDOW NGX_SPDY_MAX_WINDOW
85
86 typedef struct {
87     ngx_uint_t hash;
88     u_char len;
89     u_char header[7];
90     ngx_int_t (*handler)(ngx_http_request_t *r);
91 } ngx_http_spdy_request_header_t;
92
93
94 static void ngx_http_spdy_read_handler(ngx_event_t *rev);
95 static void ngx_http_spdy_write_handler(ngx_event_t *wev);
96 static void ngx_http_spdy_handle_connection(ngx_http_spdy_connection_t *sc);
97
98 static u_char *ngx_http_spdy_proxy_protocol(ngx_http_spdy_connection_t *sc,
99     u_char *pos, u_char *end);
100 static u_char *ngx_http_spdy_state_head(ngx_http_spdy_connection_t *sc,
101     u_char *pos, u_char *end);
102 static u_char *ngx_http_spdy_state_syn_stream(ngx_http_spdy_connection_t *sc,
103     u_char *pos, u_char *end);
104 static u_char *ngx_http_spdy_state_headers(ngx_http_spdy_connection_t *sc,
105     u_char *pos, u_char *end);
106 static u_char *ngx_http_spdy_state_headers_skip(ngx_http_spdy_connection_t *sc,
107     u_char *pos, u_char *end);
108 static u_char *ngx_http_spdy_state_headers_error(ngx_http_spdy_connection_t *sc,
109     u_char *pos, u_char *end);
110 static u_char *ngx_http_spdy_state_window_update(ngx_http_spdy_connection_t *sc,
111     u_char *pos, u_char *end);
112 static u_char *ngx_http_spdy_state_data(ngx_http_spdy_connection_t *sc,
113     u_char *pos, u_char *end);
114 static u_char *ngx_http_spdy_state_read_data(ngx_http_spdy_connection_t *sc,
115     u_char *pos, u_char *end);
116 static u_char *ngx_http_spdy_state_rst_stream(ngx_http_spdy_connection_t *sc,
117     u_char *pos, u_char *end);
118 static u_char *ngx_http_spdy_state_ping(ngx_http_spdy_connection_t *sc,
119     u_char *pos, u_char *end);
120 static u_char *ngx_http_spdy_state_skip(ngx_http_spdy_connection_t *sc,
121     u_char *pos, u_char *end);
122 static u_char *ngx_http_spdy_state_settings(ngx_http_spdy_connection_t *sc,
123     u_char *pos, u_char *end);
124 static u_char *ngx_http_spdy_state_complete(ngx_http_spdy_connection_t *sc,
125     u_char *pos, u_char *end);
126 static u_char *ngx_http_spdy_state_save(ngx_http_spdy_connection_t *sc,
127     u_char *pos, u_char *end, ngx_http_spdy_handler_pt handler);
128
129 static u_char *ngx_http_spdy_state_inflate_error(
130     ngx_http_spdy_connection_t *sc, int rc);
131 static u_char *ngx_http_spdy_state_protocol_error(
132     ngx_http_spdy_connection_t *sc);
133 static u_char *ngx_http_spdy_state_internal_error(
134     ngx_http_spdy_connection_t *sc);
135
136 static ngx_int_t ngx_http_spdy_send_window_update(
137     ngx_http_spdy_connection_t *sc, ngx_uint_t sid, ngx_uint_t delta);
138 static ngx_int_t ngx_http_spdy_send_rst_stream(ngx_http_spdy_connection_t *sc,
139     ngx_uint_t sid, ngx_uint_t status, ngx_uint_t priority);
140 static ngx_int_t ngx_http_spdy_send_settings(ngx_http_spdy_connection_t *sc);
141 static ngx_int_t ngx_http_spdy_settings_frame_handler(
142     ngx_http_spdy_connection_t *sc, ngx_http_spdy_out_frame_t *frame);
143 static ngx_http_spdy_out_frame_t *ngx_http_spdy_get_ctl_frame(
144     ngx_http_spdy_connection_t *sc, size_t size, ngx_uint_t priority);

```

```

145 static ngx_int_t ngx_http_spdy_ctl_frame_handler(
146     ngx_http_spdy_connection_t *sc, ngx_http_spdy_out_frame_t *frame);
147
148 static ngx_http_spdy_stream_t *ngx_http_spdy_create_stream(
149     ngx_http_spdy_connection_t *sc, ngx_uint_t id, ngx_uint_t priority);
150 static ngx_http_spdy_stream_t *ngx_http_spdy_get_stream_by_id(
151     ngx_http_spdy_connection_t *sc, ngx_uint_t sid);
152 #define ngx_http_spdy_streams_index_size(sscf) (sscf->streams_index_mask + 1)
153 #define ngx_http_spdy_stream_index(sscf, sid) \
154     ((sid >> 1) & sscf->streams_index_mask)
155
156 static ngx_int_t ngx_http_spdy_parse_header(ngx_http_request_t *r);
157 static ngx_int_t ngx_http_spdy_alloc_large_header_buffer(ngx_http_request_t *r);
158
159 static ngx_int_t ngx_http_spdy_handle_request_header(ngx_http_request_t *r);
160 static ngx_int_t ngx_http_spdy_parse_method(ngx_http_request_t *r);
161 static ngx_int_t ngx_http_spdy_parse_scheme(ngx_http_request_t *r);
162 static ngx_int_t ngx_http_spdy_parse_host(ngx_http_request_t *r);
163 static ngx_int_t ngx_http_spdy_parse_path(ngx_http_request_t *r);
164 static ngx_int_t ngx_http_spdy_parse_version(ngx_http_request_t *r);
165
166 static ngx_int_t ngx_http_spdy_construct_request_line(ngx_http_request_t *r);
167 static void ngx_http_spdy_run_request(ngx_http_request_t *r);
168 static ngx_int_t ngx_http_spdy_init_request_body(ngx_http_request_t *r);
169
170 static ngx_int_t ngx_http_spdy_terminate_stream(ngx_http_spdy_connection_t *sc,
171     ngx_http_spdy_stream_t *stream, ngx_uint_t status);
172
173 static void ngx_http_spdy_close_stream_handler(ngx_event_t *ev);
174
175 static void ngx_http_spdy_handle_connection_handler(ngx_event_t *rev);
176 static void ngx_http_spdy_keepalive_handler(ngx_event_t *rev);
177 static void ngx_http_spdy_finalize_connection(ngx_http_spdy_connection_t *sc,
178     ngx_int_t rc);
179
180 static ngx_int_t ngx_http_spdy_adjust_windows(ngx_http_spdy_connection_t *sc,
181     ssize_t delta);
182
183 static void ngx_http_spdy_pool_cleanup(void *data);
184
185 static void *ngx_http_spdy_zalloc(void *opaque, u_int items, u_int size);
186 static void ngx_http_spdy_zfree(void *opaque, void *address);
187
188
189 static const u_char ngx_http_spdy_dict[] = {
190     0x00, 0x00, 0x00, 0x07, 0x6f, 0x70, 0x74, 0x69, /* - - - o p t i */
191     0x6f, 0x6e, 0x73, 0x00, 0x00, 0x00, 0x04, 0x68, /* o n s - - - h */
192     0x65, 0x61, 0x64, 0x00, 0x00, 0x00, 0x04, 0x70, /* e a d - - - p */
193     0x6f, 0x73, 0x74, 0x00, 0x00, 0x00, 0x03, 0x70, /* o s t - - - p */
194     0x75, 0x74, 0x00, 0x00, 0x00, 0x06, 0x64, 0x65, /* u t - - - d e */
195     0x6c, 0x65, 0x74, 0x65, 0x00, 0x00, 0x00, 0x05, /* l e t e - - - */
196     0x74, 0x72, 0x61, 0x63, 0x65, 0x00, 0x00, 0x00, /* t r a c e - - - */
197     0x06, 0x61, 0x63, 0x63, 0x65, 0x70, 0x74, 0x00, /* - a c c e p t - */
198     0x00, 0x00, 0x0e, 0x61, 0x63, 0x63, 0x65, 0x70, /* - - - a c c e p */
199     0x74, 0x2d, 0x63, 0x68, 0x61, 0x72, 0x73, 0x65, /* t - c h a r s e */
200     0x74, 0x00, 0x00, 0x00, 0x0f, 0x61, 0x63, 0x63, /* t - - - a c c */
201     0x65, 0x70, 0x74, 0x2d, 0x65, 0x6e, 0x63, 0x6f, /* e p t - e n c o */
202     0x64, 0x69, 0x6e, 0x67, 0x00, 0x00, 0x00, 0x0f, /* d i n g - - - */
203     0x61, 0x63, 0x63, 0x65, 0x70, 0x74, 0x2d, 0x6c, /* a c c e p t - l */
204     0x61, 0x6e, 0x67, 0x75, 0x61, 0x67, 0x65, 0x00, /* a n g u a g e - */
205     0x00, 0x00, 0x0d, 0x61, 0x63, 0x63, 0x65, 0x70, /* - - - a c c e p */
206     0x74, 0x2d, 0x72, 0x61, 0x6e, 0x67, 0x65, 0x73, /* t - r a n g e s */
207     0x00, 0x00, 0x00, 0x03, 0x61, 0x67, 0x65, 0x00, /* - - - a g e - */
208     0x00, 0x00, 0x05, 0x61, 0x6c, 0x6c, 0x6f, 0x77, /* - - - a l l o w */
209     0x00, 0x00, 0x00, 0x0d, 0x61, 0x75, 0x74, 0x68, /* - - - a u t h */
210     0x6f, 0x72, 0x69, 0x7a, 0x61, 0x74, 0x69, 0x6f, /* o r i z a t i o */
211     0x6e, 0x00, 0x00, 0x00, 0x0d, 0x63, 0x61, 0x63, /* n - - - c a c */
212     0x68, 0x65, 0x2d, 0x63, 0x6f, 0x6e, 0x74, 0x72, /* h e - c o n t r */
213     0x6f, 0x6c, 0x00, 0x00, 0x00, 0x0a, 0x63, 0x6f, /* o l - - - c o */
214     0x6e, 0x6e, 0x65, 0x63, 0x74, 0x69, 0x6f, 0x6e, /* n n e c t i o n */
215     0x00, 0x00, 0x00, 0x0c, 0x63, 0x6f, 0x6e, 0x74, /* - - - c o n t */
216     0x65, 0x6e, 0x74, 0x2d, 0x62, 0x61, 0x73, 0x65, /* e n t - b a s e */
217     0x00, 0x00, 0x00, 0x10, 0x63, 0x6f, 0x6e, 0x74, /* - - - c o n t */
218     0x65, 0x6e, 0x74, 0x2d, 0x65, 0x6e, 0x63, 0x6f, /* e n t - e n c o */
219     0x64, 0x69, 0x6e, 0x67, 0x00, 0x00, 0x00, 0x10, /* d i n g - - - */
220     0x63, 0x6f, 0x6e, 0x74, 0x65, 0x6e, 0x74, 0x2d, /* c o n t e n t - */

```

221 0x6c, 0x61, 0x6e, 0x67, 0x75, 0x61, 0x67, 0x65, /\* l a n g u a g e \*/  
222 0x00, 0x00, 0x00, 0x0e, 0x63, 0x6f, 0x6e, 0x74, /\* - - - c o n t \*/  
223 0x65, 0x6e, 0x74, 0x2d, 0x6c, 0x65, 0x6e, 0x67, /\* e n t - l e n g \*/  
224 0x74, 0x68, 0x00, 0x00, 0x00, 0x10, 0x63, 0x6f, /\* t h - - - c o \*/  
225 0x6e, 0x74, 0x65, 0x6e, 0x74, 0x2d, 0x6c, 0x6f, /\* n t e n t - l o \*/  
226 0x63, 0x61, 0x74, 0x69, 0x6f, 0x6e, 0x00, 0x00, /\* c a t i o n - - \*/  
227 0x00, 0x0b, 0x63, 0x6f, 0x6e, 0x74, 0x65, 0x6e, /\* - - c o n t e n \*/  
228 0x74, 0x2d, 0x6d, 0x64, 0x35, 0x00, 0x00, 0x00, /\* t - m d 5 - - - \*/  
229 0x0d, 0x63, 0x6f, 0x6e, 0x74, 0x65, 0x6e, 0x74, /\* - c o n t e n t \*/  
230 0x2d, 0x72, 0x61, 0x6e, 0x67, 0x65, 0x00, 0x00, /\* - r a n g e - - \*/  
231 0x00, 0x0c, 0x63, 0x6f, 0x6e, 0x74, 0x65, 0x6e, /\* - - c o n t e n \*/  
232 0x74, 0x2d, 0x74, 0x79, 0x70, 0x65, 0x00, 0x00, /\* t - t y p e - - \*/  
233 0x00, 0x04, 0x64, 0x61, 0x74, 0x65, 0x00, 0x00, /\* - - d a t e - - \*/  
234 0x00, 0x04, 0x65, 0x74, 0x61, 0x67, 0x00, 0x00, /\* - - e t a g - - \*/  
235 0x00, 0x06, 0x65, 0x78, 0x70, 0x65, 0x63, 0x74, /\* - - e x p e c t \*/  
236 0x00, 0x00, 0x00, 0x07, 0x65, 0x78, 0x70, 0x69, /\* - - - e x p i \*/  
237 0x72, 0x65, 0x73, 0x00, 0x00, 0x00, 0x04, 0x66, /\* r e s - - - - f \*/  
238 0x72, 0x6f, 0x6d, 0x00, 0x00, 0x00, 0x04, 0x68, /\* r o m - - - - h \*/  
239 0x6f, 0x73, 0x74, 0x00, 0x00, 0x00, 0x08, 0x69, /\* o s t - - - - i \*/  
240 0x66, 0x2d, 0x6d, 0x61, 0x74, 0x63, 0x68, 0x00, /\* f - m a t c h - \*/  
241 0x00, 0x00, 0x11, 0x69, 0x66, 0x2d, 0x6d, 0x6f, /\* - - - i f - m o \*/  
242 0x64, 0x69, 0x66, 0x69, 0x65, 0x64, 0x2d, 0x73, /\* d i f i e d - s \*/  
243 0x69, 0x6e, 0x63, 0x65, 0x00, 0x00, 0x00, 0x0d, /\* i n c e - - - - \*/  
244 0x69, 0x66, 0x2d, 0x6e, 0x6f, 0x6e, 0x65, 0x2d, /\* i f - n o n e - \*/  
245 0x6d, 0x61, 0x74, 0x63, 0x68, 0x00, 0x00, 0x00, /\* m a t c h - - - \*/  
246 0x08, 0x69, 0x66, 0x2d, 0x72, 0x61, 0x6e, 0x67, /\* - i f - r a n g \*/  
247 0x65, 0x00, 0x00, 0x00, 0x13, 0x69, 0x66, 0x2d, /\* e - - - - i f - \*/  
248 0x75, 0x6e, 0x6d, 0x6f, 0x64, 0x69, 0x66, 0x69, /\* u n m o d i f i \*/  
249 0x65, 0x64, 0x2d, 0x73, 0x69, 0x6e, 0x63, 0x65, /\* e d - s i n c e \*/  
250 0x00, 0x00, 0x00, 0x0d, 0x6c, 0x61, 0x73, 0x74, /\* - - - - l a s t \*/  
251 0x2d, 0x6d, 0x6f, 0x64, 0x69, 0x66, 0x69, 0x65, /\* - m o d i f i e \*/  
252 0x64, 0x00, 0x00, 0x00, 0x08, 0x6c, 0x6f, 0x63, /\* d - - - - l o c \*/  
253 0x61, 0x74, 0x69, 0x6f, 0x6e, 0x00, 0x00, 0x00, /\* a t i o n - - - \*/  
254 0x0c, 0x6d, 0x61, 0x78, 0x2d, 0x66, 0x6f, 0x72, /\* - m a x - f o r \*/  
255 0x77, 0x61, 0x72, 0x64, 0x73, 0x00, 0x00, 0x00, /\* w a r d s - - - \*/  
256 0x06, 0x70, 0x72, 0x61, 0x67, 0x6d, 0x61, 0x00, /\* - p r a g m a - \*/  
257 0x00, 0x00, 0x12, 0x70, 0x72, 0x6f, 0x78, 0x79, /\* - - - p r o x y \*/  
258 0x2d, 0x61, 0x75, 0x74, 0x68, 0x65, 0x6e, 0x74, /\* - a u t h e n t \*/  
259 0x69, 0x63, 0x61, 0x74, 0x65, 0x00, 0x00, 0x00, /\* i c a t e - - - \*/  
260 0x13, 0x70, 0x72, 0x6f, 0x78, 0x79, 0x2d, 0x61, /\* - p r o x y - a \*/  
261 0x75, 0x74, 0x68, 0x6f, 0x72, 0x69, 0x7a, 0x61, /\* u t h o r i z a \*/  
262 0x74, 0x69, 0x6f, 0x6e, 0x00, 0x00, 0x00, 0x05, /\* t i o n - - - - \*/  
263 0x72, 0x61, 0x6e, 0x67, 0x65, 0x00, 0x00, 0x00, /\* r a n g e - - - \*/  
264 0x07, 0x72, 0x65, 0x66, 0x65, 0x72, 0x65, 0x72, /\* - r e f e r e r \*/  
265 0x00, 0x00, 0x00, 0x0b, 0x72, 0x65, 0x74, 0x72, /\* - - - - r e t r \*/  
266 0x79, 0x2d, 0x61, 0x66, 0x74, 0x65, 0x72, 0x00, /\* y - a f t e r - \*/  
267 0x00, 0x00, 0x06, 0x73, 0x65, 0x72, 0x76, 0x65, /\* - - - s e r v e \*/  
268 0x72, 0x00, 0x00, 0x00, 0x02, 0x74, 0x65, 0x00, /\* r - - - - t e - \*/  
269 0x00, 0x00, 0x07, 0x74, 0x72, 0x61, 0x69, 0x6c, /\* - - - t r a i l \*/  
270 0x65, 0x72, 0x00, 0x00, 0x00, 0x11, 0x74, 0x72, /\* e r - - - - t r \*/  
271 0x61, 0x6e, 0x73, 0x66, 0x65, 0x72, 0x2d, 0x65, /\* a n s f e r - e \*/  
272 0x6e, 0x63, 0x6f, 0x64, 0x69, 0x6e, 0x67, 0x00, /\* n c o d i n g - \*/  
273 0x00, 0x00, 0x07, 0x75, 0x70, 0x67, 0x72, 0x61, /\* - - - u p g r a \*/  
274 0x64, 0x65, 0x00, 0x00, 0x00, 0x0a, 0x75, 0x73, /\* d e - - - - u s \*/  
275 0x65, 0x72, 0x2d, 0x61, 0x67, 0x65, 0x6e, 0x74, /\* e r - a g e n t \*/  
276 0x00, 0x00, 0x00, 0x04, 0x76, 0x61, 0x72, 0x79, /\* - - - - v a r y \*/  
277 0x00, 0x00, 0x00, 0x03, 0x76, 0x69, 0x61, 0x00, /\* - - - - v i a - \*/  
278 0x00, 0x00, 0x07, 0x77, 0x61, 0x72, 0x6e, 0x69, /\* - - - w a r n i \*/  
279 0x6e, 0x67, 0x00, 0x00, 0x00, 0x10, 0x77, 0x77, /\* n g - - - - w w \*/  
280 0x77, 0x2d, 0x61, 0x75, 0x74, 0x68, 0x65, 0x6e, /\* w - a u t h e n \*/  
281 0x74, 0x69, 0x63, 0x61, 0x74, 0x65, 0x00, 0x00, /\* t i c a t e - - \*/  
282 0x00, 0x06, 0x6d, 0x65, 0x74, 0x68, 0x6f, 0x64, /\* - - m e t h o d \*/  
283 0x00, 0x00, 0x00, 0x03, 0x67, 0x65, 0x74, 0x00, /\* - - - - g e t - \*/  
284 0x00, 0x00, 0x06, 0x73, 0x74, 0x61, 0x74, 0x75, /\* - - - s t a t u \*/  
285 0x73, 0x00, 0x00, 0x00, 0x06, 0x32, 0x30, 0x30, /\* s - - - - 2 0 0 \*/  
286 0x20, 0x4f, 0x4b, 0x00, 0x00, 0x00, 0x07, 0x76, /\* - O K - - - - v \*/  
287 0x65, 0x72, 0x73, 0x69, 0x6f, 0x6e, 0x00, 0x00, /\* e r s i o n - - \*/  
288 0x00, 0x08, 0x48, 0x54, 0x54, 0x50, 0x2f, 0x31, /\* - - H T T P - 1 \*/  
289 0x2e, 0x31, 0x00, 0x00, 0x00, 0x03, 0x75, 0x72, /\* - 1 - - - - u r \*/  
290 0x6c, 0x00, 0x00, 0x00, 0x06, 0x70, 0x75, 0x62, /\* l - - - - p u b \*/  
291 0x6c, 0x69, 0x63, 0x00, 0x00, 0x00, 0x0a, 0x73, /\* l i c - - - - s \*/  
292 0x65, 0x74, 0x2d, 0x63, 0x6f, 0x6f, 0x6b, 0x69, /\* e t - c o o k i \*/  
293 0x65, 0x00, 0x00, 0x00, 0x0a, 0x6b, 0x65, 0x65, /\* e - - - - k e e \*/  
294 0x70, 0x2d, 0x61, 0x6c, 0x69, 0x76, 0x65, 0x00, /\* p - a l i v e - \*/  
295 0x00, 0x00, 0x06, 0x6f, 0x72, 0x69, 0x67, 0x69, /\* - - - o r i g i \*/  
296 0x6e, 0x31, 0x30, 0x30, 0x31, 0x30, 0x31, 0x32, /\* n 1 0 0 1 0 1 2 \*/

```

297 0x30, 0x31, 0x32, 0x30, 0x32, 0x32, 0x30, 0x35, /* 0 1 2 0 2 2 0 5 */
298 0x32, 0x30, 0x36, 0x33, 0x30, 0x30, 0x33, 0x30, /* 2 0 6 3 0 0 3 0 */
299 0x32, 0x33, 0x30, 0x33, 0x33, 0x30, 0x34, 0x33, /* 2 3 0 3 3 0 4 3 */
300 0x30, 0x35, 0x33, 0x30, 0x36, 0x33, 0x30, 0x37, /* 0 5 3 0 6 3 0 7 */
301 0x34, 0x30, 0x32, 0x34, 0x30, 0x35, 0x34, 0x30, /* 4 0 2 4 0 5 4 0 */
302 0x36, 0x34, 0x30, 0x37, 0x34, 0x30, 0x38, 0x34, /* 6 4 0 7 4 0 8 4 */
303 0x30, 0x39, 0x34, 0x31, 0x30, 0x34, 0x31, 0x31, /* 0 9 4 1 0 4 1 1 */
304 0x34, 0x31, 0x32, 0x34, 0x31, 0x33, 0x34, 0x31, /* 4 1 2 4 1 3 4 1 */
305 0x34, 0x34, 0x31, 0x35, 0x34, 0x31, 0x36, 0x34, /* 4 4 1 5 4 1 6 4 */
306 0x31, 0x37, 0x35, 0x30, 0x32, 0x35, 0x30, 0x34, /* 1 7 5 0 2 5 0 4 */
307 0x35, 0x30, 0x35, 0x32, 0x30, 0x33, 0x20, 0x4e, /* 5 0 5 2 0 3 - N */
308 0x6f, 0x6e, 0x2d, 0x41, 0x75, 0x74, 0x68, 0x6f, /* o n - A u t h o */
309 0x72, 0x69, 0x74, 0x61, 0x74, 0x69, 0x76, 0x65, /* r i t a t i v e */
310 0x20, 0x49, 0x6e, 0x66, 0x6f, 0x72, 0x6d, 0x61, /* - I n f o r m a */
311 0x74, 0x69, 0x6f, 0x6e, 0x32, 0x30, 0x34, 0x20, /* t i o n 2 0 4 - */
312 0x4e, 0x6f, 0x20, 0x43, 0x6f, 0x6e, 0x74, 0x65, /* N o - C o n t e */
313 0x6e, 0x74, 0x33, 0x30, 0x31, 0x20, 0x4d, 0x6f, /* n t 3 0 1 - M o */
314 0x76, 0x65, 0x64, 0x20, 0x50, 0x65, 0x72, 0x6d, /* v e d - P e r m */
315 0x61, 0x6e, 0x65, 0x6e, 0x74, 0x6c, 0x79, 0x34, /* a n e n t l y 4 */
316 0x30, 0x30, 0x20, 0x42, 0x61, 0x64, 0x20, 0x52, /* 0 0 - B a d - R */
317 0x65, 0x71, 0x75, 0x65, 0x73, 0x74, 0x34, 0x30, /* e q u e s t 4 0 */
318 0x31, 0x20, 0x55, 0x6e, 0x61, 0x75, 0x74, 0x68, /* 1 - U n a u t h */
319 0x6f, 0x72, 0x69, 0x7a, 0x65, 0x64, 0x34, 0x30, /* o r i z e d 4 0 */
320 0x33, 0x20, 0x46, 0x6f, 0x72, 0x62, 0x69, 0x64, /* 3 - F o r b i d */
321 0x64, 0x65, 0x6e, 0x34, 0x30, 0x34, 0x20, 0x4e, /* d e n 4 0 4 - N */
322 0x6f, 0x74, 0x20, 0x46, 0x6f, 0x75, 0x6e, 0x64, /* o t - F o u n d */
323 0x35, 0x30, 0x30, 0x20, 0x49, 0x6e, 0x74, 0x65, /* 5 0 0 - I n t e */
324 0x72, 0x6e, 0x61, 0x6c, 0x20, 0x53, 0x65, 0x72, /* r n a l - S e r */
325 0x76, 0x65, 0x72, 0x20, 0x45, 0x72, 0x72, 0x6f, /* v e r - E r r o */
326 0x72, 0x35, 0x30, 0x31, 0x20, 0x4e, 0x6f, 0x74, /* r 5 0 1 - N o t */
327 0x20, 0x49, 0x6d, 0x70, 0x6c, 0x65, 0x6d, 0x65, /* - I m p l e m e */
328 0x6e, 0x74, 0x65, 0x64, 0x35, 0x30, 0x33, 0x20, /* n t e d 5 0 3 - */
329 0x53, 0x65, 0x72, 0x76, 0x69, 0x63, 0x65, 0x20, /* S e r v i c e - */
330 0x55, 0x6e, 0x61, 0x76, 0x61, 0x69, 0x6c, 0x61, /* U n a v a i l a */
331 0x62, 0x6c, 0x65, 0x4a, 0x61, 0x6e, 0x20, 0x46, /* b l e J a n - E */
332 0x65, 0x62, 0x20, 0x4d, 0x61, 0x72, 0x20, 0x41, /* e b - M a r - A */
333 0x70, 0x72, 0x20, 0x4d, 0x61, 0x79, 0x20, 0x4a, /* p r - M a y - J */
334 0x75, 0x6e, 0x20, 0x4a, 0x75, 0x6c, 0x20, 0x41, /* u n - J u l - A */
335 0x75, 0x67, 0x20, 0x53, 0x65, 0x70, 0x74, 0x20, /* u g - S e p t - */
336 0x4f, 0x63, 0x74, 0x20, 0x4e, 0x6f, 0x76, 0x20, /* O c t - N o v - */
337 0x44, 0x65, 0x63, 0x20, 0x30, 0x30, 0x3a, 0x30, /* D e c - 0 0 - 0 */
338 0x30, 0x3a, 0x30, 0x30, 0x20, 0x4d, 0x6f, 0x6e, /* 0 - 0 0 - M o n */
339 0x2c, 0x20, 0x54, 0x75, 0x65, 0x2c, 0x20, 0x57, /* - - T u e - - W */
340 0x65, 0x64, 0x2c, 0x20, 0x54, 0x68, 0x75, 0x2c, /* e d - - T h u - */
341 0x20, 0x46, 0x72, 0x69, 0x2c, 0x20, 0x53, 0x61, /* - F r i - - S a */
342 0x74, 0x2c, 0x20, 0x53, 0x75, 0x6e, 0x2c, 0x20, /* t - - S u n - - */
343 0x47, 0x4d, 0x54, 0x63, 0x68, 0x75, 0x6e, 0x6b, /* G M T c h u n k */
344 0x65, 0x64, 0x2c, 0x74, 0x65, 0x78, 0x74, 0x2f, /* e d - t e x t - */
345 0x68, 0x74, 0x6d, 0x6c, 0x2c, 0x69, 0x6d, 0x61, /* h t m l - i m a */
346 0x67, 0x65, 0x2f, 0x70, 0x6e, 0x67, 0x2c, 0x69, /* g e - p n g - i */
347 0x6d, 0x61, 0x67, 0x65, 0x2f, 0x6a, 0x70, 0x67, /* m a g e - j p g */
348 0x2c, 0x69, 0x6d, 0x61, 0x67, 0x65, 0x2f, 0x67, /* - i m a g e - g */
349 0x69, 0x66, 0x2c, 0x61, 0x70, 0x70, 0x6c, 0x69, /* i f - a p p l i */
350 0x63, 0x61, 0x74, 0x69, 0x6f, 0x6e, 0x2f, 0x78, /* c a t i o n - x */
351 0x6d, 0x6c, 0x2c, 0x61, 0x70, 0x70, 0x6c, 0x69, /* m l - a p p l i */
352 0x63, 0x61, 0x74, 0x69, 0x6f, 0x6e, 0x2f, 0x78, /* c a t i o n - x */
353 0x68, 0x74, 0x6d, 0x6c, 0x2b, 0x78, 0x6d, 0x6c, /* h t m l - x m l */
354 0x2c, 0x74, 0x65, 0x78, 0x74, 0x2f, 0x70, 0x6c, /* - t e x t - p l */
355 0x61, 0x69, 0x6e, 0x2c, 0x74, 0x65, 0x78, 0x74, /* a i n - t e x t */
356 0x2f, 0x6a, 0x61, 0x76, 0x61, 0x73, 0x63, 0x72, /* - j a v a s c r */
357 0x69, 0x70, 0x74, 0x2c, 0x70, 0x75, 0x62, 0x6c, /* i p t - p u b l */
358 0x69, 0x63, 0x70, 0x72, 0x69, 0x76, 0x61, 0x74, /* i c p r i v a t */
359 0x65, 0x6d, 0x61, 0x78, 0x2d, 0x61, 0x67, 0x65, /* e m a x - a g e */
360 0x3d, 0x67, 0x7a, 0x69, 0x70, 0x2c, 0x64, 0x65, /* - g z i p - d e */
361 0x66, 0x6c, 0x61, 0x74, 0x65, 0x2c, 0x73, 0x64, /* f l a t e - s d */
362 0x63, 0x68, 0x63, 0x68, 0x61, 0x72, 0x73, 0x65, /* c h c h a r s e */
363 0x74, 0x3d, 0x75, 0x74, 0x66, 0x2d, 0x38, 0x63, /* t - u t f - 8 c */
364 0x68, 0x61, 0x72, 0x73, 0x65, 0x74, 0x3d, 0x69, /* h a r s e t - i */
365 0x73, 0x6f, 0x2d, 0x38, 0x38, 0x35, 0x39, 0x2d, /* s o - 8 8 5 9 - */
366 0x31, 0x2c, 0x75, 0x74, 0x66, 0x2d, 0x2c, 0x2a, /* 1 - u t f - - - */
367 0x2c, 0x65, 0x6e, 0x71, 0x3d, 0x30, 0x2e /* - e n q - 0 - */

```

```

368 };
369
370
371 static ngx_http_spdy_request_header_t ngx_http_spdy_request_headers[] = {
372     { 0, 6, "method", ngx_http_spdy_parse_method },

```

```

373     { 0, 6, "scheme", ngx_http_spdy_parse_scheme },
374     { 0, 4, "host", ngx_http_spdy_parse_host },
375     { 0, 4, "path", ngx_http_spdy_parse_path },
376     { 0, 7, "version", ngx_http_spdy_parse_version },
377 };
378
379 #define NGX_SPDY_REQUEST_HEADERS \
380     (sizeof(ngx_http_spdy_request_headers) \
381      / sizeof(ngx_http_spdy_request_header_t))
382
383
384 void
385 ngx_http_spdy_init(ngx_event_t *rev)
386 {
387     int rc;
388     ngx_connection_t *c;
389     ngx_pool_cleanup_t *cfn;
390     ngx_http_connection_t *hc;
391     ngx_http_spdy_srv_conf_t *sscf;
392     ngx_http_spdy_main_conf_t *smcf;
393     ngx_http_spdy_connection_t *sc;
394
395     c = rev->data;
396     hc = c->data;
397
398     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, c->log, 0, "init spdy request");
399
400     c->log->action = "processing SPDY";
401
402     smcf = ngx_http_get_module_main_conf(hc->conf_ctx, ngx_http_spdy_module);
403
404     if (smcf->recv_buffer == NULL) {
405         smcf->recv_buffer = ngx_palloc(ngx_cycle->pool, smcf->recv_buffer_size);
406         if (smcf->recv_buffer == NULL) {
407             ngx_http_close_connection(c);
408             return;
409         }
410     }
411
412     sc = ngx_palloc(c->pool, sizeof(ngx_http_spdy_connection_t));
413     if (sc == NULL) {
414         ngx_http_close_connection(c);
415         return;
416     }
417
418     sc->connection = c;
419     sc->http_connection = hc;
420
421     sc->send_window = NGX_SPDY_CONNECTION_WINDOW;
422     sc->recv_window = NGX_SPDY_CONNECTION_WINDOW;
423
424     sc->init_window = NGX_SPDY_INIT_STREAM_WINDOW;
425
426     sc->handler = hc->proxy_protocol ? ngx_http_spdy_proxy_protocol
427                                     : ngx_http_spdy_state_head;
428
429     sc->zstream_in.zalloc = ngx_http_spdy_zalloc;
430     sc->zstream_in.zfree = ngx_http_spdy_zfree;
431     sc->zstream_in.opaque = sc;
432
433     rc = inflateInit(&sc->zstream_in);
434     if (rc != Z_OK) {
435         ngx_log_error(NGX_LOG_ALERT, c->log, 0,
436                     "inflateInit() failed: %d", rc);
437         ngx_http_close_connection(c);
438         return;
439     }
440
441     sc->zstream_out.zalloc = ngx_http_spdy_zalloc;
442     sc->zstream_out.zfree = ngx_http_spdy_zfree;
443     sc->zstream_out.opaque = sc;
444
445     sscf = ngx_http_get_module_srv_conf(hc->conf_ctx, ngx_http_spdy_module);
446
447     rc = deflateInit2(&sc->zstream_out, (int) sscf->headers_comp,
448                    Z_DEFLATED, 11, 4, Z_DEFAULT_STRATEGY);

```

```

449
450 if (rc != Z_OK) {
451     ngx_log_error(NGX_LOG_ALERT, c->log, 0,
452                 "deflateInit2() failed: %d", rc);
453     ngx_http_close_connection(c);
454     return;
455 }
456
457 rc = deflateSetDictionary(&sc->zstream_out, ngx_http_spdy_dict,
458                          sizeof(ngx_http_spdy_dict));
459 if (rc != Z_OK) {
460     ngx_log_error(NGX_LOG_ALERT, c->log, 0,
461                 "deflateSetDictionary() failed: %d", rc);
462     ngx_http_close_connection(c);
463     return;
464 }
465
466 sc->pool = ngx_create_pool(sscf->pool_size, sc->connection->log);
467 if (sc->pool == NULL) {
468     ngx_http_close_connection(c);
469     return;
470 }
471
472 cln = ngx_pool_cleanup_add(c->pool, sizeof(ngx_pool_cleanup_file_t));
473 if (cln == NULL) {
474     ngx_http_close_connection(c);
475     return;
476 }
477
478 cln->handler = ngx_http_spdy_pool_cleanup;
479 cln->data = sc;
480
481 sc->streams_index = ngx_palloc(sc->pool,
482                               ngx_http_spdy_streams_index_size(sscf)
483                               * sizeof(ngx_http_spdy_stream_t *));
484 if (sc->streams_index == NULL) {
485     ngx_http_close_connection(c);
486     return;
487 }
488
489 if (ngx_http_spdy_send_settings(sc) == NGX_ERROR) {
490     ngx_http_close_connection(c);
491     return;
492 }
493
494 if (ngx_http_spdy_send_window_update(sc, 0, NGX_SPDY_MAX_WINDOW
495                                     - sc->recv_window)
496     == NGX_ERROR)
497 {
498     ngx_http_close_connection(c);
499     return;
500 }
501
502 sc->recv_window = NGX_SPDY_MAX_WINDOW;
503
504 ngx_queue_init(&sc->waiting);
505 ngx_queue_init(&sc->posted);
506
507 c->data = sc;
508
509 rev->handler = ngx_http_spdy_read_handler;
510 c->write->handler = ngx_http_spdy_write_handler;
511
512 ngx_http_spdy_read_handler(rev);
513 }
514
515
516 static void
517 ngx_http_spdy_read_handler(ngx_event_t *rev)
518 {
519     u_char                *p, *end;
520     size_t                available;
521     ssize_t                n;
522     ngx_connection_t      *c;
523     ngx_http_spdy_main_conf_t *smcf;
524     ngx_http_spdy_connection_t *sc;

```



```

525
526 c = rev->data;
527 sc = c->data;
528
529 if (rev->timedout) {
530     ngx_log_error(NGX_LOG_INFO, c->log, NGX_ETIMEDOUT, "client timed out");
531     ngx_http_spdy_finalize_connection(sc, NGX_HTTP_REQUEST_TIME_OUT);
532     return;
533 }
534
535 ngx_log_debug0(NGX_LOG_DEBUG_HTTP, c->log, 0, "spdy read handler");
536
537 sc->blocked = 1;
538
539 smcf = ngx_http_get_module_main_conf(sc->http_connection->conf_ctx,
540     ngx_http_spdy_module);
541
542 available = smcf->recv_buffer_size - 2 * NGX_SPDY_STATE_BUFFER_SIZE;
543
544 do {
545     p = smcf->recv_buffer;
546
547     ngx_memcpy(p, sc->buffer, NGX_SPDY_STATE_BUFFER_SIZE);
548     end = p + sc->buffer_used;
549
550     n = c->recv(c, end, available);
551
552     if (n == NGX_AGAIN) {
553         break;
554     }
555
556     if (n == 0 && (sc->incomplete || sc->processing)) {
557         ngx_log_error(NGX_LOG_INFO, c->log, 0,
558             "client prematurely closed connection");
559     }
560
561     if (n == 0 || n == NGX_ERROR) {
562         ngx_http_spdy_finalize_connection(sc,
563             NGX_HTTP_CLIENT_CLOSED_REQUEST);
564         return;
565     }
566
567     end += n;
568
569     sc->buffer_used = 0;
570     sc->incomplete = 0;
571
572     do {
573         p = sc->handler(sc, p, end);
574
575         if (p == NULL) {
576             return;
577         }
578
579     } while (p != end);
580
581 } while (rev->ready);
582
583 if (ngx_handle_read_event(rev, 0) != NGX_OK) {
584     ngx_http_spdy_finalize_connection(sc, NGX_HTTP_INTERNAL_SERVER_ERROR);
585     return;
586 }
587
588 if (sc->last_out && ngx_http_spdy_send_output_queue(sc) == NGX_ERROR) {
589     ngx_http_spdy_finalize_connection(sc, NGX_HTTP_CLIENT_CLOSED_REQUEST);
590     return;
591 }
592
593 sc->blocked = 0;
594
595 if (sc->processing) {
596     if (rev->timer_set) {
597         ngx_del_timer(rev);
598     }
599     return;
600 }

```

```

601     ngx\_http\_spdy\_handle\_connection(sc);
602 }
603
604
605
606 static void
607 ngx\_http\_spdy\_write\_handler(ngx\_event\_t *wev)
608 {
609     ngx\_int\_t          rc;
610     ngx\_queue\_t       *q;
611     ngx\_connection\_t  *c;
612     ngx\_http\_spdy\_stream\_t *stream;
613     ngx\_http\_spdy\_connection\_t *sc;
614
615     c = wev->data;
616     sc = c->data;
617
618     if (wev->timedout) {
619         ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, c->log, 0,
620                     "spdy write event timed out");
621         ngx\_http\_spdy\_finalize\_connection(sc, NGX\_HTTP\_CLIENT\_CLOSED\_REQUEST);
622         return;
623     }
624
625     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, c->log, 0, "spdy write handler");
626
627     sc->blocked = 1;
628
629     rc = ngx\_http\_spdy\_send\_output\_queue(sc);
630
631     if (rc == NGX\_ERROR) {
632         ngx\_http\_spdy\_finalize\_connection(sc, NGX\_HTTP\_CLIENT\_CLOSED\_REQUEST);
633         return;
634     }
635
636     while (!ngx\_queue\_empty(&sc->posted)) {
637         q = ngx\_queue\_head(&sc->posted);
638
639         ngx\_queue\_remove(q);
640
641         stream = ngx\_queue\_data(q, ngx\_http\_spdy\_stream\_t, queue);
642
643         stream->handled = 0;
644
645         ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, c->log, 0,
646                     "run spdy stream %ui", stream->id);
647
648         wev = stream->request->connection->write;
649         wev->handler(wev);
650     }
651
652     sc->blocked = 0;
653
654     if (rc == NGX\_AGAIN) {
655         return;
656     }
657
658     ngx\_http\_spdy\_handle\_connection(sc);
659 }
660
661
662 ngx\_int\_t
663 ngx\_http\_spdy\_send\_output\_queue(ngx\_http\_spdy\_connection\_t *sc)
664 {
665     ngx\_chain\_t          *cl;
666     ngx\_event\_t         *wev;
667     ngx\_connection\_t    *c;
668     ngx\_http\_core\_loc\_conf\_t *clcf;
669     ngx\_http\_spdy\_out\_frame\_t *out, *frame, *fn;
670
671     c = sc->connection;
672
673     if (c->error) {
674         return NGX\_ERROR;
675     }
676

```

```

677 wev = c->write;
678
679 if (!wev->ready) {
680     return NGX_OK;
681 }
682
683 cl = NULL;
684 out = NULL;
685
686 for (frame = sc->last_out; frame; frame = fn) {
687     frame->last->next = cl;
688     cl = frame->first;
689
690     fn = frame->next;
691     frame->next = out;
692     out = frame;
693
694     ngx_log_debug5(NGX_LOG_DEBUG_HTTP, c->log, 0,
695         "spdy frame out: %p sid:%ui prio:%ui bl:%d len:%uz",
696         out, out->stream ? out->stream->id : 0, out->priority,
697         out->blocked, out->length);
698 }
699
700 cl = c->send_chain(c, cl, 0);
701
702 if (cl == NGX_CHAIN_ERROR) {
703     c->error = 1;
704
705     if (!sc->blocked) {
706         ngx_post_event(wev, &ngx_posted_events);
707     }
708
709     return NGX_ERROR;
710 }
711
712 clcf = ngx_http_get_module_loc_conf(sc->http_connection->conf_ctx,
713     ngx_http_core_module);
714
715 if (ngx_handle_write_event(wev, clcf->send_lowat) != NGX_OK) {
716     return NGX_ERROR; /* FIXME */
717 }
718
719 if (cl) {
720     ngx_add_timer(wev, clcf->send_timeout);
721 }
722 else {
723     if (wev->timer_set) {
724         ngx_del_timer(wev);
725     }
726 }
727
728 for ( /* void */ ; out; out = fn) {
729     fn = out->next;
730
731     if (out->handler(sc, out) != NGX_OK) {
732         out->blocked = 1;
733         out->priority = NGX_SPDY_HIGHEST_PRIORITY;
734         break;
735     }
736
737     ngx_log_debug4(NGX_LOG_DEBUG_HTTP, c->log, 0,
738         "spdy frame sent: %p sid:%ui bl:%d len:%uz",
739         out, out->stream ? out->stream->id : 0,
740         out->blocked, out->length);
741 }
742
743 frame = NULL;
744
745 for ( /* void */ ; out; out = fn) {
746     fn = out->next;
747     out->next = frame;
748     frame = out;
749 }
750
751 sc->last_out = frame;
752

```

```

753     return NGX\_OK;
754 }
755
756
757 static void
758 ngx\_http\_spdy\_handle\_connection(ngx\_http\_spdy\_connection\_t *sc)
759 {
760     ngx\_connection\_t *c;
761     ngx\_http\_spdy\_srv\_conf\_t *sscf;
762
763     if (sc->last_out || sc->processing) {
764         return;
765     }
766
767     c = sc->connection;
768
769     if (c->error) {
770         ngx\_http\_close\_connection(c);
771         return;
772     }
773
774     if (c->buffered) {
775         return;
776     }
777
778     sscf = ngx\_http\_get\_module\_srv\_conf(sc->http_connection->conf_ctx,
779                                         ngx\_http\_spdy\_module);
780
781     if (sc->incomplete) {
782         ngx\_add\_timer(c->read, sscf->recv_timeout);
783         return;
784     }
785
786     if (ngx\_terminate || ngx\_exiting) {
787         ngx\_http\_close\_connection(c);
788         return;
789     }
790
791     ngx\_destroy\_pool(sc->pool);
792
793     sc->pool = NULL;
794     sc->free_ctl_frames = NULL;
795     sc->free_fake_connections = NULL;
796
797     #if (NGX\_HTTP\_SSL)
798     if (c->ssl) {
799         ngx\_ssl\_free\_buffer(c);
800     }
801     #endif
802
803     c->destroyed = 1;
804     c->idle = 1;
805     ngx\_reusable\_connection(c, 1);
806
807     c->write->handler = ngx\_http\_empty\_handler;
808     c->read->handler = ngx\_http\_spdy\_keepalive\_handler;
809
810     if (c->write->timer_set) {
811         ngx\_del\_timer(c->write);
812     }
813
814     ngx\_add\_timer(c->read, sscf->keepalive_timeout);
815 }
816
817 static u_char *
818 ngx\_http\_spdy\_proxy\_protocol(ngx\_http\_spdy\_connection\_t *sc, u_char *pos,
819                               u_char *end)
820 {
821     ngx\_log\_t *log;
822
823     log = sc->connection->log;
824     log->action = "reading PROXY protocol";
825
826     pos = ngx\_proxy\_protocol\_parse(sc->connection, pos, end);
827
828     log->action = "processing SPDY";

```

```

829
830     if (pos == NULL) {
831         return ngx_http_spdy_state_protocol_error(sc);
832     }
833
834     return ngx_http_spdy_state_complete(sc, pos, end);
835 }
836
837
838 static u_char *
839 ngx_http_spdy_state_head(ngx_http_spdy_connection_t *sc, u_char *pos,
840     u_char *end)
841 {
842     uint32_t    head, flen;
843     ngx_uint_t  type;
844
845     if (end - pos < NGX_SPDY_FRAME_HEADER_SIZE) {
846         return ngx_http_spdy_state_save(sc, pos, end,
847             ngx_http_spdy_state_head);
848     }
849
850     head = ngx_spdy_frame_parse_uint32(pos);
851
852     pos += sizeof(uint32_t);
853
854     flen = ngx_spdy_frame_parse_uint32(pos);
855
856     sc->flags = ngx_spdy_frame_flags(flen);
857     sc->length = ngx_spdy_frame_length(flen);
858
859     pos += sizeof(uint32_t);
860
861     ngx_log_debug3(NGX_LOG_DEBUG_HTTP, sc->connection->log, 0,
862         "process spdy frame head:%08XD f:%Xd l:%uz",
863         head, sc->flags, sc->length);
864
865     if (ngx_spdy_ctl_frame_check(head)) {
866         type = ngx_spdy_ctl_frame_type(head);
867
868         switch (type) {
869
870             case NGX_SPDY_SYN_STREAM:
871                 return ngx_http_spdy_state_syn_stream(sc, pos, end);
872
873             case NGX_SPDY_SYN_REPLY:
874                 ngx_log_error(NGX_LOG_INFO, sc->connection->log, 0,
875                     "client sent unexpected SYN_REPLY frame");
876                 return ngx_http_spdy_state_protocol_error(sc);
877
878             case NGX_SPDY_RST_STREAM:
879                 return ngx_http_spdy_state_rst_stream(sc, pos, end);
880
881             case NGX_SPDY_SETTINGS:
882                 return ngx_http_spdy_state_settings(sc, pos, end);
883
884             case NGX_SPDY_PING:
885                 return ngx_http_spdy_state_ping(sc, pos, end);
886
887             case NGX_SPDY_GOAWAY:
888                 return ngx_http_spdy_state_skip(sc, pos, end); /* TODO */
889
890             case NGX_SPDY_HEADERS:
891                 ngx_log_error(NGX_LOG_INFO, sc->connection->log, 0,
892                     "client sent unexpected HEADERS frame");
893                 return ngx_http_spdy_state_protocol_error(sc);
894
895             case NGX_SPDY_WINDOW_UPDATE:
896                 return ngx_http_spdy_state_window_update(sc, pos, end);
897
898             default:
899                 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, sc->connection->log, 0,
900                     "spdy control frame with unknown type %ui", type);
901                 return ngx_http_spdy_state_skip(sc, pos, end);
902         }
903     }
904 }

```

```

905     if (ngx_spdy_data_frame_check(head)) {
906         sc->stream = ngx_http_spdy_get_stream_by_id(sc, head);
907         return ngx_http_spdy_state_data(sc, pos, end);
908     }
909
910     ngx_log_error(NGX_LOG_INFO, sc->connection->log, 0,
911                 "client sent invalid frame");
912
913     return ngx_http_spdy_state_protocol_error(sc);
914 }
915
916 static u_char *
917 ngx_http_spdy_state_syn_stream(ngx_http_spdy_connection_t *sc, u_char *pos,
918                               u_char *end)
919 {
920     ngx_uint_t      sid, prio;
921     ngx_http_spdy_stream_t *stream;
922     ngx_http_spdy_srv_conf_t *sscf;
923
924     if (end - pos < NGX_SPDY_SYN_STREAM_SIZE) {
925         return ngx_http_spdy_state_save(sc, pos, end,
926                                       ngx_http_spdy_state_syn_stream);
927     }
928
929     if (sc->length <= NGX_SPDY_SYN_STREAM_SIZE) {
930         ngx_log_error(NGX_LOG_INFO, sc->connection->log, 0,
931                     "client sent SYN_STREAM frame with incorrect length %uz",
932                     sc->length);
933
934         return ngx_http_spdy_state_protocol_error(sc);
935     }
936
937     sc->length -= NGX_SPDY_SYN_STREAM_SIZE;
938
939     sid = ngx_spdy_frame_parse_sid(pos);
940     prio = pos[8] >> 5;
941
942     pos += NGX_SPDY_SYN_STREAM_SIZE;
943
944     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, sc->connection->log, 0,
945                  "spdy SYN_STREAM frame sid:%ui prio:%ui", sid, prio);
946
947     if (sid % 2 == 0 || sid <= sc->last_sid) {
948         ngx_log_error(NGX_LOG_INFO, sc->connection->log, 0,
949                     "client sent SYN_STREAM frame "
950                     "with invalid Stream-ID %ui", sid);
951
952         stream = ngx_http_spdy_get_stream_by_id(sc, sid);
953
954         if (stream) {
955             if (ngx_http_spdy_terminate_stream(sc, stream,
956                                              NGX_SPDY_PROTOCOL_ERROR)
957                 != NGX_OK)
958             {
959                 return ngx_http_spdy_state_internal_error(sc);
960             }
961         }
962     }
963
964     return ngx_http_spdy_state_protocol_error(sc);
965 }
966
967 sc->last_sid = sid;
968
969 sscf = ngx_http_get_module_srv_conf(sc->http_connection->conf_ctx,
970                                     ngx_http_spdy_module);
971
972 if (sc->processing >= sscf->concurrent_streams) {
973     ngx_log_error(NGX_LOG_INFO, sc->connection->log, 0,
974                 "concurrent streams exceeded %ui", sc->processing);
975
976     if (ngx_http_spdy_send_rst_stream(sc, sid, NGX_SPDY_REFUSED_STREAM,
977                                     prio)
978         != NGX_OK)
979     {
980

```

```

981     return ngx_http_spdy_state_internal_error(sc);
982 }
983
984     return ngx_http_spdy_state_headers_skip(sc, pos, end);
985 }
986
987 stream = ngx_http_spdy_create_stream(sc, sid, prio);
988 if (stream == NULL) {
989     return ngx_http_spdy_state_internal_error(sc);
990 }
991
992 stream->in_closed = (sc->flags & NGX_SPDY_FLAG_FIN) ? 1 : 0;
993
994 stream->request->request_length = NGX_SPDY_FRAME_HEADER_SIZE
995     + NGX_SPDY_SYN_STREAM_SIZE
996     + sc->length;
997
998 sc->stream = stream;
999
1000     return ngx_http_spdy_state_headers(sc, pos, end);
1001 }
1002
1003
1004 static u_char *
1005 ngx_http_spdy_state_headers(ngx_http_spdy_connection_t *sc, u_char *pos,
1006     u_char *end)
1007 {
1008     int                z;
1009     size_t             size;
1010     ngx_buf_t         *buf;
1011     ngx_int_t         rc;
1012     ngx_http_request_t *r;
1013
1014     size = end - pos;
1015
1016     if (size == 0) {
1017         return ngx_http_spdy_state_save(sc, pos, end,
1018             ngx_http_spdy_state_headers);
1019     }
1020
1021     if (size > sc->length) {
1022         size = sc->length;
1023     }
1024
1025     r = sc->stream->request;
1026
1027     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1028         "process spdy header block %uz of %uz", size, sc->length);
1029
1030     buf = r->header_in;
1031
1032     sc->zstream_in.next_in = pos;
1033     sc->zstream_in.avail_in = size;
1034     sc->zstream_in.next_out = buf->last;
1035
1036     /* one byte is reserved for null-termination of the last header value */
1037     sc->zstream_in.avail_out = buf->end - buf->last - 1;
1038
1039     z = inflate(&sc->zstream_in, Z_NO_FLUSH);
1040
1041     if (z == Z_NEED_DICT) {
1042         z = inflateSetDictionary(&sc->zstream_in, ngx_http_spdy_dict,
1043             sizeof(ngx_http_spdy_dict));
1044
1045         if (z != Z_OK) {
1046             if (z == Z_DATA_ERROR) {
1047                 ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
1048                     "client sent SYN_STREAM frame with header "
1049                     "block encoded using wrong dictionary: %u1",
1050                     (u_long) sc->zstream_in.adler);
1051
1052                 return ngx_http_spdy_state_protocol_error(sc);
1053             }
1054
1055             ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1056                 "inflateSetDictionary() failed: %d", z);

```

```

1057     return ngx\_http\_spdy\_state\_internal\_error(sc);
1058 }
1059
1060
1061 ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
1062     "spdy inflateSetDictionary(): %d", z);
1063
1064 z = sc->zstream_in.avail_in ? inflate(&sc->zstream_in, Z_NO_FLUSH)
1065     : Z_OK;
1066 }
1067
1068 ngx\_log\_debug5(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
1069     "spdy inflate out: ni:%p no:%p ai:%ud ao:%ud rc:%d",
1070     sc->zstream_in.next_in, sc->zstream_in.next_out,
1071     sc->zstream_in.avail_in, sc->zstream_in.avail_out,
1072     z);
1073
1074 if (z != Z_OK) {
1075     return ngx\_http\_spdy\_state\_inflate\_error(sc, z);
1076 }
1077
1078 sc->length -= sc->zstream_in.next_in - pos;
1079 pos = sc->zstream_in.next_in;
1080
1081 buf->last = sc->zstream_in.next_out;
1082
1083 if (r->headers_in.headers.part.elts == NULL) {
1084
1085     if (buf->last - buf->pos < NGX\_SPDY\_NV\_NUM\_SIZE) {
1086
1087         if (sc->length == 0) {
1088             ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
1089                 "premature end of spdy header block");
1090
1091             return ngx\_http\_spdy\_state\_headers\_error(sc, pos, end);
1092         }
1093
1094         return ngx\_http\_spdy\_state\_save(sc, pos, end,
1095             ngx\_http\_spdy\_state\_headers);
1096     }
1097
1098     sc->entries = ngx\_spdy\_frame\_parse\_uint32(buf->pos);
1099
1100     buf->pos += NGX\_SPDY\_NV\_NUM\_SIZE;
1101
1102     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
1103         "spdy header block has %ui entries",
1104         sc->entries);
1105
1106     if (ngx\_list\_init(&r->headers_in.headers, r->pool, 20,
1107         sizeof(ngx\_table\_elt\_t))
1108         != NGX\_OK)
1109     {
1110         ngx\_http\_spdy\_close\_stream(sc->stream,
1111             NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);
1112         return ngx\_http\_spdy\_state\_headers\_skip(sc, pos, end);
1113     }
1114
1115     if (ngx\_array\_init(&r->headers_in.cookies, r->pool, 2,
1116         sizeof(ngx\_table\_elt\_t *))
1117         != NGX\_OK)
1118     {
1119         ngx\_http\_spdy\_close\_stream(sc->stream,
1120             NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);
1121         return ngx\_http\_spdy\_state\_headers\_skip(sc, pos, end);
1122     }
1123 }
1124
1125 while (sc->entries) {
1126
1127     rc = ngx\_http\_spdy\_parse\_header(r);
1128
1129     switch (rc) {
1130
1131     case NGX\_DONE:
1132         sc->entries--;

```



```

1133
1134 case NGX_OK:
1135     break;
1136
1137 case NGX_AGAIN:
1138
1139     if (sc->zstream_in.avail_in) {
1140
1141         rc = ngx_http_spdy_alloc_large_header_buffer(r);
1142
1143         if (rc == NGX_DECLINED) {
1144             ngx_http_finalize_request(r,
1145                                     NGX_HTTP_REQUEST_HEADER_TOO_LARGE);
1146             return ngx_http_spdy_state_headers_skip(sc, pos, end);
1147         }
1148
1149         if (rc != NGX_OK) {
1150             ngx_http_spdy_close_stream(sc->stream,
1151                                       NGX_HTTP_INTERNAL_SERVER_ERROR);
1152             return ngx_http_spdy_state_headers_skip(sc, pos, end);
1153         }
1154
1155         /* null-terminate the last processed header name or value */
1156         *buf->pos = '\0';
1157
1158         buf = r->header_in;
1159
1160         sc->zstream_in.next_out = buf->last;
1161
1162         /* one byte is reserved for null-termination */
1163         sc->zstream_in.avail_out = buf->end - buf->last - 1;
1164
1165         z = inflate(&sc->zstream_in, Z_NO_FLUSH);
1166
1167         ngx_log_debug5(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1168                     "spdy inflate out: ni:%p no:%p ai:%ud ao:%ud rc:%d",
1169                     sc->zstream_in.next_in, sc->zstream_in.next_out,
1170                     sc->zstream_in.avail_in, sc->zstream_in.avail_out,
1171                     z);
1172
1173         if (z != Z_OK) {
1174             return ngx_http_spdy_state_inflate_error(sc, z);
1175         }
1176
1177         sc->length -= sc->zstream_in.next_in - pos;
1178         pos = sc->zstream_in.next_in;
1179
1180         buf->last = sc->zstream_in.next_out;
1181
1182         continue;
1183     }
1184
1185     if (sc->length == 0) {
1186         ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1187                       "premature end of spdy header block");
1188
1189         return ngx_http_spdy_state_headers_error(sc, pos, end);
1190     }
1191
1192     return ngx_http_spdy_state_save(sc, pos, end,
1193                                    ngx_http_spdy_state_headers);
1194
1195 case NGX_HTTP_PARSE_INVALID_HEADER:
1196     ngx_http_finalize_request(r, NGX_HTTP_BAD_REQUEST);
1197     return ngx_http_spdy_state_headers_skip(sc, pos, end);
1198
1199 default: /* NGX_ERROR */
1200     return ngx_http_spdy_state_headers_error(sc, pos, end);
1201 }
1202
1203 /* a header line has been parsed successfully */
1204
1205 rc = ngx_http_spdy_handle_request_header(r);
1206
1207 if (rc != NGX_OK) {
1208     if (rc == NGX_HTTP_PARSE_INVALID_HEADER) {

```

```

1209         ngx_http_finalize_request(r, NGX_HTTP_BAD_REQUEST);
1210         return ngx_http_spdy_state_headers_skip(sc, pos, end);
1211     }
1212
1213     if (rc != NGX_ABORT) {
1214         ngx_http_spdy_close_stream(sc->stream,
1215                                     NGX_HTTP_INTERNAL_SERVER_ERROR);
1216     }
1217
1218     return ngx_http_spdy_state_headers_skip(sc, pos, end);
1219 }
1220 }
1221
1222 if (buf->pos != buf->last || sc->zstream_in.avail_in) {
1223     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1224                   "incorrect number of spdy header block entries");
1225
1226     return ngx_http_spdy_state_headers_error(sc, pos, end);
1227 }
1228
1229 if (sc->length) {
1230     return ngx_http_spdy_state_save(sc, pos, end,
1231                                     ngx_http_spdy_state_headers);
1232 }
1233
1234 /* null-terminate the last header value */
1235 *buf->pos = '\0';
1236
1237 ngx_http_spdy_run_request(r);
1238
1239 return ngx_http_spdy_state_complete(sc, pos, end);
1240 }
1241
1242
1243 static u_char *
1244 ngx_http_spdy_state_headers_skip(ngx_http_spdy_connection_t *sc, u_char *pos,
1245                                 u_char *end)
1246 {
1247     int    n;
1248     size_t size;
1249     u_char buffer[NGX_SPDY_SKIP_HEADERS_BUFFER_SIZE];
1250
1251     if (sc->length == 0) {
1252         return ngx_http_spdy_state_complete(sc, pos, end);
1253     }
1254
1255     size = end - pos;
1256
1257     if (size == 0) {
1258         return ngx_http_spdy_state_save(sc, pos, end,
1259                                         ngx_http_spdy_state_headers_skip);
1260     }
1261
1262     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, sc->connection->log, 0,
1263                   "spdy header block skip %uz of %uz", size, sc->length);
1264
1265     sc->zstream_in.next_in = pos;
1266     sc->zstream_in.avail_in = (size < sc->length) ? size : sc->length;
1267
1268     while (sc->zstream_in.avail_in) {
1269         sc->zstream_in.next_out = buffer;
1270         sc->zstream_in.avail_out = NGX_SPDY_SKIP_HEADERS_BUFFER_SIZE;
1271
1272         n = inflate(&sc->zstream_in, Z_NO_FLUSH);
1273
1274         ngx_log_debug5(NGX_LOG_DEBUG_HTTP, sc->connection->log, 0,
1275                       "spdy inflate out: ni:%p no:%p ai:%ud ao:%ud rc:%d",
1276                       sc->zstream_in.next_in, sc->zstream_in.next_out,
1277                       sc->zstream_in.avail_in, sc->zstream_in.avail_out,
1278                       n);
1279
1280         if (n != Z_OK) {
1281             return ngx_http_spdy_state_inflate_error(sc, n);
1282         }
1283     }
1284 }

```

```

1285     pos = sc->zstream_in.next_in;
1286
1287     if (size < sc->length) {
1288         sc->length -= size;
1289         return ngx\_http\_spdy\_state\_save(sc, pos, end,
1290                                     ngx\_http\_spdy\_state\_headers\_skip);
1291     }
1292
1293     return ngx\_http\_spdy\_state\_complete(sc, pos, end);
1294 }
1295
1296
1297 static u_char *
1298 ngx\_http\_spdy\_state\_headers\_error(ngx\_http\_spdy\_connection\_t *sc, u_char *pos,
1299 u_char *end)
1300 {
1301     ngx\_http\_spdy\_stream\_t *stream;
1302
1303     stream = sc->stream;
1304
1305     ngx\_log\_error(NGX\_LOG\_INFO, sc->connection->log, 0,
1306                 "client sent SYN_STREAM frame for stream %ui "
1307                 "with invalid header block", stream->id);
1308
1309     if (ngx\_http\_spdy\_send\_rst\_stream(sc, stream->id, NGX\_SPDY\_PROTOCOL\_ERROR,
1310                                     stream->priority)
1311         != NGX\_OK)
1312     {
1313         return ngx\_http\_spdy\_state\_internal\_error(sc);
1314     }
1315
1316     stream->out_closed = 1;
1317
1318     ngx\_http\_spdy\_close\_stream(stream, NGX\_HTTP\_BAD\_REQUEST);
1319
1320     return ngx\_http\_spdy\_state\_headers\_skip(sc, pos, end);
1321 }
1322
1323
1324 static u_char *
1325 ngx\_http\_spdy\_state\_window\_update(ngx\_http\_spdy\_connection\_t *sc, u_char *pos,
1326 u_char *end)
1327 {
1328     size_t          delta;
1329     ngx\_uint\_t      sid;
1330     ngx\_event\_t    *wev;
1331     ngx\_queue\_t    *q;
1332     ngx\_http\_spdy\_stream\_t *stream;
1333
1334     if (end - pos < NGX\_SPDY\_WINDOW\_UPDATE\_SIZE) {
1335         return ngx\_http\_spdy\_state\_save(sc, pos, end,
1336                                     ngx\_http\_spdy\_state\_window\_update);
1337     }
1338
1339     if (sc->length != NGX\_SPDY\_WINDOW\_UPDATE\_SIZE) {
1340         ngx\_log\_error(NGX\_LOG\_INFO, sc->connection->log, 0,
1341                     "client sent WINDOW_UPDATE frame "
1342                     "with incorrect length %uz", sc->length);
1343
1344         return ngx\_http\_spdy\_state\_protocol\_error(sc);
1345     }
1346
1347     sid = ngx\_spdy\_frame\_parse\_sid(pos);
1348
1349     pos += NGX\_SPDY\_SID\_SIZE;
1350
1351     delta = ngx\_spdy\_frame\_parse\_delta(pos);
1352
1353     pos += NGX\_SPDY\_DELTA\_SIZE;
1354
1355     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, sc->connection->log, 0,
1356                 "spdy WINDOW_UPDATE sid:%ui delta:%ui", sid, delta);
1357
1358     if (sid) {
1359         stream = ngx\_http\_spdy\_get\_stream\_by\_id(sc, sid);
1360

```

```

1361     if (stream == NULL) {
1362         ngx_log_debug0(NGX_LOG_DEBUG_HTTP, sc->connection->log, 0,
1363             "unknown spdy stream");
1364
1365         return ngx_http_spdy_state_complete(sc, pos, end);
1366     }
1367
1368     if (stream->send_window > (ssize_t) (NGX_SPDY_MAX_WINDOW - delta)) {
1369
1370         ngx_log_error(NGX_LOG_INFO, sc->connection->log, 0,
1371             "client violated flow control for stream %ui: "
1372             "received WINDOW_UPDATE frame with delta %uz "
1373             "not allowed for window %z",
1374             sid, delta, stream->send_window);
1375
1376         if (ngx_http_spdy_terminate_stream(sc, stream,
1377             NGX_SPDY_FLOW_CONTROL_ERROR)
1378             == NGX_ERROR)
1379         {
1380             return ngx_http_spdy_state_internal_error(sc);
1381         }
1382
1383         return ngx_http_spdy_state_complete(sc, pos, end);
1384     }
1385
1386     stream->send_window += delta;
1387
1388     if (stream->exhausted) {
1389         stream->exhausted = 0;
1390
1391         wev = stream->request->connection->write;
1392
1393         if (!wev->timer_set) {
1394             wev->delayed = 0;
1395             wev->handler(wev);
1396         }
1397     }
1398
1399 } else {
1400     sc->send_window += delta;
1401
1402     if (sc->send_window > NGX_SPDY_MAX_WINDOW) {
1403         ngx_log_error(NGX_LOG_INFO, sc->connection->log, 0,
1404             "client violated connection flow control: "
1405             "received WINDOW_UPDATE frame with delta %uz "
1406             "not allowed for window %uz",
1407             delta, sc->send_window);
1408
1409         return ngx_http_spdy_state_protocol_error(sc);
1410     }
1411
1412     while (!ngx_queue_empty(&sc->waiting)) {
1413         q = ngx_queue_head(&sc->waiting);
1414
1415         ngx_queue_remove(q);
1416
1417         stream = ngx_queue_data(q, ngx_http_spdy_stream_t, queue);
1418
1419         stream->handled = 0;
1420
1421         wev = stream->request->connection->write;
1422
1423         if (!wev->timer_set) {
1424             wev->delayed = 0;
1425             wev->handler(wev);
1426
1427             if (sc->send_window == 0) {
1428                 break;
1429             }
1430         }
1431     }
1432 }
1433
1434 return ngx_http_spdy_state_complete(sc, pos, end);
1435 }
1436

```

```

1437 static u_char *
1438 ngx_http_spdy_state_data(ngx_http_spdy_connection_t *sc, u_char *pos,
1439 u_char *end)
1440 {
1441     ngx_http_spdy_stream_t *stream;
1442
1443     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, sc->connection->log, 0,
1444 "spdy DATA frame");
1445
1446     if (sc->length > sc->recv_window) {
1447         ngx_log_error(NGX_LOG_INFO, sc->connection->log, 0,
1448 "client violated connection flow control: "
1449 "received DATA frame length %uz, available window %uz",
1450 sc->length, sc->recv_window);
1451
1452         return ngx_http_spdy_state_protocol_error(sc);
1453     }
1454
1455     sc->recv_window -= sc->length;
1456
1457     if (sc->recv_window < NGX_SPDY_MAX_WINDOW / 4) {
1458         if (ngx_http_spdy_send_window_update(sc, 0,
1459 NGX_SPDY_MAX_WINDOW
1460 - sc->recv_window)
1461 == NGX_ERROR)
1462         {
1463             return ngx_http_spdy_state_internal_error(sc);
1464         }
1465
1466         sc->recv_window = NGX_SPDY_MAX_WINDOW;
1467     }
1468
1469     stream = sc->stream;
1470
1471     if (stream == NULL) {
1472         ngx_log_debug0(NGX_LOG_DEBUG_HTTP, sc->connection->log, 0,
1473 "unknown spdy stream");
1474
1475         return ngx_http_spdy_state_skip(sc, pos, end);
1476     }
1477
1478     if (sc->length > stream->recv_window) {
1479         ngx_log_error(NGX_LOG_INFO, sc->connection->log, 0,
1480 "client violated flow control for stream %ui: "
1481 "received DATA frame length %uz, available window %uz",
1482 stream->id, sc->length, stream->recv_window);
1483
1484         if (ngx_http_spdy_terminate_stream(sc, stream,
1485 NGX_SPDY_FLOW_CONTROL_ERROR)
1486 == NGX_ERROR)
1487         {
1488             return ngx_http_spdy_state_internal_error(sc);
1489         }
1490
1491         return ngx_http_spdy_state_skip(sc, pos, end);
1492     }
1493
1494     stream->recv_window -= sc->length;
1495
1496     if (stream->recv_window < NGX_SPDY_STREAM_WINDOW / 4) {
1497         if (ngx_http_spdy_send_window_update(sc, stream->id,
1498 NGX_SPDY_STREAM_WINDOW
1499 - stream->recv_window)
1500 == NGX_ERROR)
1501         {
1502             return ngx_http_spdy_state_internal_error(sc);
1503         }
1504
1505         stream->recv_window = NGX_SPDY_STREAM_WINDOW;
1506     }
1507
1508     if (stream->in_closed) {
1509         ngx_log_error(NGX_LOG_INFO, sc->connection->log, 0,

```

```

1513         "client sent DATA frame for half-closed stream %ui",
1514         stream->id);
1515
1516     if (ngx_http_spdy_terminate_stream(sc, stream,
1517         NGX_SPDY_STREAM_ALREADY_CLOSED)
1518         == NGX_ERROR)
1519     {
1520         return ngx_http_spdy_state_internal_error(sc);
1521     }
1522
1523     return ngx_http_spdy_state_skip(sc, pos, end);
1524 }
1525
1526 return ngx_http_spdy_state_read_data(sc, pos, end);
1527 }
1528
1529
1530 static u_char *
1531 ngx_http_spdy_state_read_data(ngx_http_spdy_connection_t *sc, u_char *pos,
1532     u_char *end)
1533 {
1534     size_t          size;
1535     ssize_t         n;
1536     ngx_buf_t     *buf;
1537     ngx_int_t     rc;
1538     ngx_temp_file_t *tf;
1539     ngx_http_request_t *r;
1540     ngx_http_spdy_stream_t *stream;
1541     ngx_http_request_body_t *rb;
1542     ngx_http_core_loc_conf_t *clcf;
1543
1544     stream = sc->stream;
1545
1546     if (stream == NULL) {
1547         return ngx_http_spdy_state_skip(sc, pos, end);
1548     }
1549
1550     if (stream->skip_data) {
1551
1552         if (sc->flags & NGX_SPDY_FLAG_FIN) {
1553             stream->in_closed = 1;
1554         }
1555
1556         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, sc->connection->log, 0,
1557             "skipping spdy DATA frame, reason: %d",
1558             stream->skip_data);
1559
1560         return ngx_http_spdy_state_skip(sc, pos, end);
1561     }
1562
1563     size = end - pos;
1564
1565     if (size > sc->length) {
1566         size = sc->length;
1567     }
1568
1569     r = stream->request;
1570
1571     if (r->request_body == NULL
1572         && ngx_http_spdy_init_request_body(r) != NGX_OK)
1573     {
1574         stream->skip_data = NGX_SPDY_DATA_INTERNAL_ERROR;
1575         return ngx_http_spdy_state_skip(sc, pos, end);
1576     }
1577
1578     rb = r->request_body;
1579     tf = rb->temp_file;
1580     buf = rb->buf;
1581
1582     if (size) {
1583         rb->rest += size;
1584
1585         if (r->headers_in.content_length_n != -1
1586             && r->headers_in.content_length_n < rb->rest)
1587         {
1588             ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,

```

```

1589         "client intended to send body data "
1590         "larger than declared");
1591
1592     stream->skip_data = NGX\_SPDY\_DATA\_ERROR;
1593     goto error;
1594
1595 } else {
1596     clcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_core\_module);
1597
1598     if (clcf->client_max_body_size
1599         && clcf->client_max_body_size < rb->rest)
1600     {
1601         ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, 0,
1602             "client intended to send "
1603             "too large chunked body: %0 bytes", rb->rest);
1604
1605         stream->skip_data = NGX\_SPDY\_DATA\_ERROR;
1606         goto error;
1607     }
1608 }
1609
1610 sc->length -= size;
1611
1612 if (tf) {
1613     buf->start = pos;
1614     buf->pos = pos;
1615
1616     pos += size;
1617
1618     buf->end = pos;
1619     buf->last = pos;
1620
1621     n = ngx\_write\_chain\_to\_temp\_file(tf, rb->bufs);
1622
1623     /* TODO: n == 0 or not complete and level event */
1624
1625     if (n == NGX\_ERROR) {
1626         stream->skip_data = NGX\_SPDY\_DATA\_INTERNAL\_ERROR;
1627         goto error;
1628     }
1629
1630     tf->offset += n;
1631
1632 } else {
1633     buf->last = ngx\_cpymem(buf->last, pos, size);
1634     pos += size;
1635 }
1636
1637 r->request_length += size;
1638 }
1639
1640 if (sc->length) {
1641     return ngx\_http\_spdy\_state\_save(sc, pos, end,
1642         ngx\_http\_spdy\_state\_read\_data);
1643 }
1644
1645 if (sc->flags & NGX\_SPDY\_FLAG\_FIN) {
1646
1647     stream->in_closed = 1;
1648
1649     if (r->headers_in.content_length_n < 0) {
1650         r->headers_in.content_length_n = rb->rest;
1651
1652     } else if (r->headers_in.content_length_n != rb->rest) {
1653         ngx\_log\_error(NGX\_LOG\_INFO, r->connection->log, 0,
1654             "client prematurely closed stream: "
1655             "only %0 out of %0 bytes of request body received",
1656             rb->rest, r->headers_in.content_length_n);
1657
1658         stream->skip_data = NGX\_SPDY\_DATA\_ERROR;
1659         goto error;
1660     }
1661
1662 if (tf) {
1663     ngx\_memzero(buf, sizeof(ngx\_buf\_t));
1664

```

```

1665         buf->in_file = 1;
1666         buf->file_last = tf->file.offset;
1667         buf->file = &tf->file;
1668
1669         rb->buf = NULL;
1670     }
1671
1672     if (rb->post_handler) {
1673         r->read_event_handler = ngx\_http\_block\_reading;
1674         rb->post_handler(r);
1675     }
1676 }
1677
1678 return ngx\_http\_spdy\_state\_complete(sc, pos, end);
1679
1680 error:
1681
1682 if (rb->post_handler) {
1683
1684     if (stream->skip_data == NGX\_SPDY\_DATA\_ERROR) {
1685         rc = (r->headers_in.content_length_n == -1)
1686             ? NGX\_HTTP\_REQUEST\_ENTITY\_TOO\_LARGE
1687             : NGX\_HTTP\_BAD\_REQUEST;
1688
1689     } else {
1690         rc = NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
1691     }
1692
1693     ngx\_http\_finalize\_request(r, rc);
1694 }
1695
1696 return ngx\_http\_spdy\_state\_skip(sc, pos, end);
1697 }
1698
1699
1700 static u_char *
1701 ngx\_http\_spdy\_state\_rst\_stream(ngx\_http\_spdy\_connection\_t *sc, u_char *pos,
1702 u_char *end)
1703 {
1704     ngx\_uint\_t          sid, status;
1705     ngx\_event\_t        *ev;
1706     ngx\_connection\_t   *fc;
1707     ngx\_http\_spdy\_stream\_t *stream;
1708
1709     if (end - pos < NGX\_SPDY\_RST\_STREAM\_SIZE) {
1710         return ngx\_http\_spdy\_state\_save(sc, pos, end,
1711                                         ngx\_http\_spdy\_state\_rst\_stream);
1712     }
1713
1714     if (sc->length != NGX\_SPDY\_RST\_STREAM\_SIZE) {
1715         ngx\_log\_error(NGX\_LOG\_INFO, sc->connection->log, 0,
1716                     "client sent RST_STREAM frame with incorrect length %uz",
1717                     sc->length);
1718
1719         return ngx\_http\_spdy\_state\_protocol\_error(sc);
1720     }
1721
1722     sid = ngx\_spdy\_frame\_parse\_sid(pos);
1723
1724     pos += NGX\_SPDY\_SID\_SIZE;
1725
1726     status = ngx\_spdy\_frame\_parse\_uint32(pos);
1727
1728     pos += sizeof(uint32\_t);
1729
1730     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, sc->connection->log, 0,
1731                 "spdy RST_STREAM sid:%ui st:%ui", sid, status);
1732
1733     stream = ngx\_http\_spdy\_get\_stream\_by\_id(sc, sid);
1734
1735     if (stream == NULL) {
1736         ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, sc->connection->log, 0,
1737                       "unknown spdy stream");
1738
1739         return ngx\_http\_spdy\_state\_complete(sc, pos, end);
1740     }

```



```

1741 stream->in_closed = 1;
1742 stream->out_closed = 1;
1743
1744
1745 fc = stream->request->connection;
1746 fc->error = 1;
1747
1748 switch (status) {
1749
1750 case NGX_SPDY_CANCEL:
1751     ngx_log_error(NGX_LOG_INFO, fc->log, 0,
1752                 "client canceled stream %ui", sid);
1753     break;
1754
1755 case NGX_SPDY_INTERNAL_ERROR:
1756     ngx_log_error(NGX_LOG_INFO, fc->log, 0,
1757                 "client terminated stream %ui due to internal error",
1758                 sid);
1759     break;
1760
1761 default:
1762     ngx_log_error(NGX_LOG_INFO, fc->log, 0,
1763                 "client terminated stream %ui with status %ui",
1764                 sid, status);
1765     break;
1766 }
1767
1768 ev = fc->read;
1769 ev->handler(ev);
1770
1771 return ngx_http_spdy_state_complete(sc, pos, end);
1772 }
1773
1774
1775 static u_char *
1776 ngx_http_spdy_state_ping(ngx_http_spdy_connection_t *sc, u_char *pos,
1777                          u_char *end)
1778 {
1779     u_char          *p;
1780     ngx_buf_t       *buf;
1781     ngx_http_spdy_out_frame_t *frame;
1782
1783     if (end - pos < NGX_SPDY_PING_SIZE) {
1784         return ngx_http_spdy_state_save(sc, pos, end,
1785                                         ngx_http_spdy_state_ping);
1786     }
1787
1788     if (sc->length != NGX_SPDY_PING_SIZE) {
1789         ngx_log_error(NGX_LOG_INFO, sc->connection->log, 0,
1790                     "client sent PING frame with incorrect length %uz",
1791                     sc->length);
1792
1793         return ngx_http_spdy_state_protocol_error(sc);
1794     }
1795
1796     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, sc->connection->log, 0,
1797                  "spdy PING frame");
1798
1799     frame = ngx_http_spdy_get_ctl_frame(sc, NGX_SPDY_PING_SIZE,
1800                                       NGX_SPDY_HIGHEST_PRIORITY);
1801     if (frame == NULL) {
1802         return ngx_http_spdy_state_internal_error(sc);
1803     }
1804
1805     buf = frame->first->buf;
1806
1807     p = buf->pos;
1808
1809     p = ngx_spdy_frame_write_head(p, NGX_SPDY_PING);
1810     p = ngx_spdy_frame_write_flags_and_len(p, 0, NGX_SPDY_PING_SIZE);
1811
1812     p = ngx_cpymem(p, pos, NGX_SPDY_PING_SIZE);
1813
1814     buf->last = p;
1815
1816     ngx_http_spdy_queue_frame(sc, frame);

```

```

1817     pos += NGX\_SPDY\_PING\_SIZE;
1818
1819     return ngx\_http\_spdy\_state\_complete(sc, pos, end);
1820 }
1821
1822
1823
1824 static u_char *
1825 ngx\_http\_spdy\_state\_skip(ngx\_http\_spdy\_connection\_t *sc, u_char *pos,
1826     u_char *end)
1827 {
1828     size_t size;
1829
1830     size = end - pos;
1831
1832     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, sc->connection->log, 0,
1833         "spdy frame skip %uz of %uz", size, sc->length);
1834
1835     if (size < sc->length) {
1836         sc->length -= size;
1837         return ngx\_http\_spdy\_state\_save(sc, end, end,
1838             ngx\_http\_spdy\_state\_skip);
1839     }
1840
1841     return ngx\_http\_spdy\_state\_complete(sc, pos + sc->length, end);
1842 }
1843
1844
1845 static u_char *
1846 ngx\_http\_spdy\_state\_settings(ngx\_http\_spdy\_connection\_t *sc, u_char *pos,
1847     u_char *end)
1848 {
1849     ngx\_uint\_t fid, val;
1850
1851     if (sc->entries == 0) {
1852
1853         if (end - pos < NGX\_SPDY\_SETTINGS\_NUM\_SIZE) {
1854             return ngx\_http\_spdy\_state\_save(sc, pos, end,
1855                 ngx\_http\_spdy\_state\_settings);
1856         }
1857
1858         sc->entries = ngx\_spdy\_frame\_parse\_uint32(pos);
1859
1860         pos += NGX\_SPDY\_SETTINGS\_NUM\_SIZE;
1861         sc->length -= NGX\_SPDY\_SETTINGS\_NUM\_SIZE;
1862
1863         if (sc->length < sc->entries * NGX\_SPDY\_SETTINGS\_PAIR\_SIZE) {
1864             ngx\_log\_error(NGX\_LOG\_INFO, sc->connection->log, 0,
1865                 "client sent SETTINGS frame with incorrect "
1866                 "length %uz or number of entries %ui",
1867                 sc->length, sc->entries);
1868
1869             return ngx\_http\_spdy\_state\_protocol\_error(sc);
1870         }
1871
1872         ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, sc->connection->log, 0,
1873             "spdy SETTINGS frame has %ui entries", sc->entries);
1874     }
1875
1876     while (sc->entries) {
1877         if (end - pos < NGX\_SPDY\_SETTINGS\_PAIR\_SIZE) {
1878             return ngx\_http\_spdy\_state\_save(sc, pos, end,
1879                 ngx\_http\_spdy\_state\_settings);
1880         }
1881
1882         sc->entries--;
1883         sc->length -= NGX\_SPDY\_SETTINGS\_PAIR\_SIZE;
1884
1885         fid = ngx\_spdy\_frame\_parse\_uint32(pos);
1886
1887         pos += NGX\_SPDY\_SETTINGS\_FID\_SIZE;
1888
1889         val = ngx\_spdy\_frame\_parse\_uint32(pos);
1890
1891         pos += NGX\_SPDY\_SETTINGS\_VAL\_SIZE;
1892

```

```

1893     ngx_log_debug3(NGX_LOG_DEBUG_HTTP, sc->connection->log, 0,
1894         "spdy SETTINGS entry fl:%ui id:%ui val:%ui",
1895         ngx_spdy_frame_flags(fid), ngx_spdy_frame_id(fid), val);
1896
1897     if (ngx_spdy_frame_flags(fid) == NGX_SPDY_SETTINGS_FLAG_PERSISTED) {
1898         continue;
1899     }
1900
1901     switch (ngx_spdy_frame_id(fid)) {
1902
1903     case NGX_SPDY_SETTINGS_INIT_WINDOW:
1904
1905         if (val > NGX_SPDY_MAX_WINDOW) {
1906             ngx_log_error(NGX_LOG_INFO, sc->connection->log, 0,
1907                 "client sent SETTINGS frame with "
1908                 "incorrect INIT_WINDOW value: %ui", val);
1909
1910             return ngx_http_spdy_state_protocol_error(sc);
1911         }
1912
1913         if (ngx_http_spdy_adjust_windows(sc, val - sc->init_window)
1914             != NGX_OK)
1915         {
1916             return ngx_http_spdy_state_internal_error(sc);
1917         }
1918
1919         sc->init_window = val;
1920
1921         continue;
1922     }
1923 }
1924
1925 return ngx_http_spdy_state_complete(sc, pos, end);
1926 }
1927
1928
1929 static u_char *
1930 ngx_http_spdy_state_complete(ngx_http_spdy_connection_t *sc, u_char *pos,
1931     u_char *end)
1932 {
1933     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, sc->connection->log, 0,
1934         "spdy frame complete pos:%p end:%p", pos, end);
1935
1936     if (pos > end) {
1937         ngx_log_error(NGX_LOG_ALERT, sc->connection->log, 0,
1938             "receive buffer overrun");
1939
1940         ngx_debug_point();
1941         return ngx_http_spdy_state_internal_error(sc);
1942     }
1943
1944     sc->handler = ngx_http_spdy_state_head;
1945     sc->stream = NULL;
1946
1947     return pos;
1948 }
1949
1950
1951 static u_char *
1952 ngx_http_spdy_state_save(ngx_http_spdy_connection_t *sc,
1953     u_char *pos, u_char *end, ngx_http_spdy_handler_pt handler)
1954 {
1955     size_t size;
1956
1957     ngx_log_debug3(NGX_LOG_DEBUG_HTTP, sc->connection->log, 0,
1958         "spdy frame state save pos:%p end:%p handler:%p",
1959         pos, end, handler);
1960
1961     size = end - pos;
1962
1963     if (size > NGX_SPDY_STATE_BUFFER_SIZE) {
1964         ngx_log_error(NGX_LOG_ALERT, sc->connection->log, 0,
1965             "state buffer overflow: %uz bytes required", size);
1966
1967         ngx_debug_point();
1968         return ngx_http_spdy_state_internal_error(sc);

```

```

1969     }
1970
1971     ngx_memcpy(sc->buffer, pos, NGX\_SPDY\_STATE\_BUFFER\_SIZE);
1972
1973     sc->buffer_used = size;
1974     sc->handler = handler;
1975     sc->incomplete = 1;
1976
1977     return end;
1978 }
1979
1980
1981 static u_char *
1982 ngx_http_spdy_state_inflate_error(ngx\_http\_spdy\_connection\_t *sc, int rc)
1983 {
1984     if (rc == Z_DATA_ERROR || rc == Z_STREAM_END) {
1985         ngx_log_error(NGX\_LOG\_INFO, sc->connection->log, 0,
1986             "client sent SYN_STREAM frame with "
1987             "corrupted header block, inflate() failed: %d", rc);
1988
1989         return ngx\_http\_spdy\_state\_protocol\_error(sc);
1990     }
1991
1992     ngx_log_error(NGX\_LOG\_ERR, sc->connection->log, 0,
1993         "inflate() failed: %d", rc);
1994
1995     return ngx\_http\_spdy\_state\_internal\_error(sc);
1996 }
1997
1998
1999 static u_char *
2000 ngx_http_spdy_state_protocol_error(ngx\_http\_spdy\_connection\_t *sc)
2001 {
2002     ngx_log_debug0(NGX\_LOG\_DEBUG\_HTTP, sc->connection->log, 0,
2003         "spdy state protocol error");
2004
2005     if (sc->stream) {
2006         sc->stream->out_closed = 1;
2007         ngx\_http\_spdy\_close\_stream(sc->stream, NGX\_HTTP\_BAD\_REQUEST);
2008     }
2009
2010     ngx\_http\_spdy\_finalize\_connection(sc, NGX\_HTTP\_CLIENT\_CLOSED\_REQUEST);
2011
2012     return NULL;
2013 }
2014
2015
2016 static u_char *
2017 ngx_http_spdy_state_internal_error(ngx\_http\_spdy\_connection\_t *sc)
2018 {
2019     ngx_log_debug0(NGX\_LOG\_DEBUG\_HTTP, sc->connection->log, 0,
2020         "spdy state internal error");
2021
2022     if (sc->stream) {
2023         sc->stream->out_closed = 1;
2024         ngx\_http\_spdy\_close\_stream(sc->stream, NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);
2025     }
2026
2027     ngx\_http\_spdy\_finalize\_connection(sc, NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);
2028
2029     return NULL;
2030 }
2031
2032
2033 static ngx\_int\_t
2034 ngx_http_spdy_send_window_update(ngx\_http\_spdy\_connection\_t *sc, ngx\_uint\_t sid,
2035     ngx\_uint\_t delta)
2036 {
2037     u_char                *p;
2038     ngx\_buf\_t            *buf;
2039     ngx\_http\_spdy\_out\_frame\_t *frame;
2040
2041     ngx_log_debug2(NGX\_LOG\_DEBUG\_HTTP, sc->connection->log, 0,
2042         "spdy send WINDOW_UPDATE sid:%ui delta:%ui", sid, delta);
2043
2044     frame = ngx\_http\_spdy\_get\_ctl\_frame(sc, NGX\_SPDY\_WINDOW\_UPDATE\_SIZE,

```

```

2045                                     NGX_SPDY_HIGHEST_PRIORITY);
2046     if (frame == NULL) {
2047         return NGX_ERROR;
2048     }
2049
2050     buf = frame->first->buf;
2051
2052     p = buf->pos;
2053
2054     p = ngx_spdy_frame_write_head(p, NGX_SPDY_WINDOW_UPDATE);
2055     p = ngx_spdy_frame_write_flags_and_len(p, 0, NGX_SPDY_WINDOW_UPDATE_SIZE);
2056
2057     p = ngx_spdy_frame_write_sid(p, sid);
2058     p = ngx_spdy_frame_aligned_write_uint32(p, delta);
2059
2060     buf->last = p;
2061
2062     ngx_http_spdy_queue_frame(sc, frame);
2063
2064     return NGX_OK;
2065 }
2066
2067
2068 static ngx_int_t
2069 ngx_http_spdy_send_rst_stream(ngx_http_spdy_connection_t *sc, ngx_uint_t sid,
2070 ngx_uint_t status, ngx_uint_t priority)
2071 {
2072     u_char                *p;
2073     ngx_buf_t            *buf;
2074     ngx_http_spdy_out_frame_t *frame;
2075
2076     if (sc->connection->error) {
2077         return NGX_OK;
2078     }
2079
2080     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, sc->connection->log, 0,
2081         "spdy send RST_STREAM sid:%ui st:%ui", sid, status);
2082
2083     frame = ngx_http_spdy_get_ctl_frame(sc, NGX_SPDY_RST_STREAM_SIZE,
2084         priority);
2085     if (frame == NULL) {
2086         return NGX_ERROR;
2087     }
2088
2089     buf = frame->first->buf;
2090
2091     p = buf->pos;
2092
2093     p = ngx_spdy_frame_write_head(p, NGX_SPDY_RST_STREAM);
2094     p = ngx_spdy_frame_write_flags_and_len(p, 0, NGX_SPDY_RST_STREAM_SIZE);
2095
2096     p = ngx_spdy_frame_write_sid(p, sid);
2097     p = ngx_spdy_frame_aligned_write_uint32(p, status);
2098
2099     buf->last = p;
2100
2101     ngx_http_spdy_queue_frame(sc, frame);
2102
2103     return NGX_OK;
2104 }
2105
2106
2107 #if 0
2108 static ngx_int_t
2109 ngx_http_spdy_send_goaway(ngx_http_spdy_connection_t *sc)
2110 {
2111     u_char                *p;
2112     ngx_buf_t            *buf;
2113     ngx_http_spdy_out_frame_t *frame;
2114
2115     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, sc->connection->log, 0,
2116         "spdy send GOAWAY sid:%ui", sc->last_sid);
2117
2118     frame = ngx_http_spdy_get_ctl_frame(sc, NGX_SPDY_GOAWAY_SIZE,
2119         NGX_SPDY_HIGHEST_PRIORITY);
2120     if (frame == NULL) {

```

```

2121     return NGX_ERROR;
2122 }
2123
2124 buf = frame->first->buf;
2125
2126 p = buf->pos;
2127
2128 p = ngx_spdy_frame_write_head(p, NGX_SPDY_GOAWAY);
2129 p = ngx_spdy_frame_write_flags_and_len(p, 0, NGX_SPDY_GOAWAY_SIZE);
2130
2131 p = ngx_spdy_frame_write_sid(p, sc->last_sid);
2132
2133 buf->last = p;
2134
2135 ngx_http_spdy_queue_frame(sc, frame);
2136
2137 return NGX_OK;
2138 }
2139 #endif
2140
2141
2142 static ngx_int_t
2143 ngx_http_spdy_send_settings(ngx_http_spdy_connection_t *sc)
2144 {
2145     u_char                *p;
2146     ngx_buf_t             *buf;
2147     ngx_chain_t           *cl;
2148     ngx_http_spdy_srv_conf_t *sscf;
2149     ngx_http_spdy_out_frame_t *frame;
2150
2151     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, sc->connection->log, 0,
2152                 "spdy send SETTINGS frame");
2153
2154     frame = ngx_palloc(sc->pool, sizeof(ngx_http_spdy_out_frame_t));
2155     if (frame == NULL) {
2156         return NGX_ERROR;
2157     }
2158
2159     cl = ngx_alloc_chain_link(sc->pool);
2160     if (cl == NULL) {
2161         return NGX_ERROR;
2162     }
2163
2164     buf = ngx_create_temp_buf(sc->pool, NGX_SPDY_FRAME_HEADER_SIZE
2165                 + NGX_SPDY_SETTINGS_NUM_SIZE
2166                 + 2 * NGX_SPDY_SETTINGS_PAIR_SIZE);
2167     if (buf == NULL) {
2168         return NGX_ERROR;
2169     }
2170
2171     buf->last_buf = 1;
2172
2173     cl->buf = buf;
2174     cl->next = NULL;
2175
2176     frame->first = cl;
2177     frame->last = cl;
2178     frame->handler = ngx_http_spdy_settings_frame_handler;
2179     frame->stream = NULL;
2180     #if (NGX_DEBUG)
2181     frame->length = NGX_SPDY_SETTINGS_NUM_SIZE
2182                 + 2 * NGX_SPDY_SETTINGS_PAIR_SIZE;
2183     #endif
2184     frame->priority = NGX_SPDY_HIGHEST_PRIORITY;
2185     frame->blocked = 0;
2186
2187     p = buf->pos;
2188
2189     p = ngx_spdy_frame_write_head(p, NGX_SPDY_SETTINGS);
2190     p = ngx_spdy_frame_write_flags_and_len(p, NGX_SPDY_FLAG_CLEAR_SETTINGS,
2191                 NGX_SPDY_SETTINGS_NUM_SIZE
2192                 + 2 * NGX_SPDY_SETTINGS_PAIR_SIZE);
2193
2194     p = ngx_spdy_frame_aligned_write_uint32(p, 2);
2195
2196     sscf = ngx_http_get_module_srv_conf(sc->http_connection->conf_ctx,

```

```

2197         ngx_http_spdy_module);
2198
2199     p = ngx_spdy_frame_write_flags_and_id(p, 0, NGX_SPDY_SETTINGS_MAX_STREAMS);
2200     p = ngx_spdy_frame_aligned_write_uint32(p, sscf->concurrent_streams);
2201
2202     p = ngx_spdy_frame_write_flags_and_id(p, 0, NGX_SPDY_SETTINGS_INIT_WINDOW);
2203     p = ngx_spdy_frame_aligned_write_uint32(p, NGX_SPDY_STREAM_WINDOW);
2204
2205     buf->last = p;
2206
2207     ngx_http_spdy_queue_frame(sc, frame);
2208
2209     return NGX_OK;
2210 }
2211
2212
2213 ngx_int_t
2214 ngx_http_spdy_settings_frame_handler(ngx_http_spdy_connection_t *sc,
2215     ngx_http_spdy_out_frame_t *frame)
2216 {
2217     ngx_buf_t *buf;
2218
2219     buf = frame->first->buf;
2220
2221     if (buf->pos != buf->last) {
2222         return NGX_AGAIN;
2223     }
2224
2225     ngx_free_chain(sc->pool, frame->first);
2226
2227     return NGX_OK;
2228 }
2229
2230
2231 static ngx_http_spdy_out_frame_t *
2232 ngx_http_spdy_get_ctl_frame(ngx_http_spdy_connection_t *sc, size_t length,
2233     ngx_uint_t priority)
2234 {
2235     ngx_chain_t *cl;
2236     ngx_http_spdy_out_frame_t *frame;
2237
2238     frame = sc->free_ctl_frames;
2239
2240     if (frame) {
2241         sc->free_ctl_frames = frame->next;
2242
2243         cl = frame->first;
2244         cl->buf->pos = cl->buf->start;
2245
2246     } else {
2247         frame = ngx_palloc(sc->pool, sizeof(ngx_http_spdy_out_frame_t));
2248         if (frame == NULL) {
2249             return NULL;
2250         }
2251
2252         cl = ngx_alloc_chain_link(sc->pool);
2253         if (cl == NULL) {
2254             return NULL;
2255         }
2256
2257         cl->buf = ngx_create_temp_buf(sc->pool,
2258             NGX_SPDY_CTL_FRAME_BUFFER_SIZE);
2259         if (cl->buf == NULL) {
2260             return NULL;
2261         }
2262
2263         cl->buf->last_buf = 1;
2264
2265         frame->first = cl;
2266         frame->last = cl;
2267         frame->handler = ngx_http_spdy_ctl_frame_handler;
2268         frame->stream = NULL;
2269     }
2270
2271 #if (NGX_DEBUG)
2272     if (length > NGX_SPDY_CTL_FRAME_BUFFER_SIZE - NGX_SPDY_FRAME_HEADER_SIZE) {

```

```

2273     ngx_log_error(NGX_LOG_ALERT, sc->pool->log, 0,
2274                 "requested control frame is too large: %uz", length);
2275     return NULL;
2276 }
2277
2278     frame->length = length;
2279 #endif
2280
2281     frame->priority = priority;
2282     frame->blocked = 0;
2283
2284     return frame;
2285 }
2286
2287 static ngx_int_t
2288 ngx_http_spdy_ctl_frame_handler(ngx_http_spdy_connection_t *sc,
2289                                ngx_http_spdy_out_frame_t *frame)
2290 {
2291     ngx_buf_t *buf;
2292
2293     buf = frame->first->buf;
2294
2295     if (buf->pos != buf->last) {
2296         return NGX_AGAIN;
2297     }
2298
2299     frame->next = sc->free_ctl_frames;
2300     sc->free_ctl_frames = frame;
2301
2302     return NGX_OK;
2303 }
2304
2305
2306 static ngx_http_spdy_stream_t *
2307 ngx_http_spdy_create_stream(ngx_http_spdy_connection_t *sc, ngx_uint_t id,
2308                             ngx_uint_t priority)
2309 {
2310     ngx_log_t *log;
2311     ngx_uint_t index;
2312     ngx_event_t *rev, *wev;
2313     ngx_connection_t *fc;
2314     ngx_http_log_ctx_t *ctx;
2315     ngx_http_request_t *r;
2316     ngx_http_spdy_stream_t *stream;
2317     ngx_http_core_srv_conf_t *cscf;
2318     ngx_http_spdy_srv_conf_t *sscf;
2319
2320     fc = sc->free_fake_connections;
2321
2322     if (fc) {
2323         sc->free_fake_connections = fc->data;
2324
2325         rev = fc->read;
2326         wev = fc->write;
2327         log = fc->log;
2328         ctx = log->data;
2329
2330     } else {
2331         fc = ngx_palloc(sc->pool, sizeof(ngx_connection_t));
2332         if (fc == NULL) {
2333             return NULL;
2334         }
2335
2336         rev = ngx_palloc(sc->pool, sizeof(ngx_event_t));
2337         if (rev == NULL) {
2338             return NULL;
2339         }
2340
2341         wev = ngx_palloc(sc->pool, sizeof(ngx_event_t));
2342         if (wev == NULL) {
2343             return NULL;
2344         }
2345
2346         log = ngx_palloc(sc->pool, sizeof(ngx_log_t));
2347         if (log == NULL) {

```



```

2349         return NULL;
2350     }
2351
2352     ctx = ngx_palloc(sc->pool, sizeof(ngx_http_log_ctx_t));
2353     if (ctx == NULL) {
2354         return NULL;
2355     }
2356
2357     ctx->connection = fc;
2358     ctx->request = NULL;
2359 }
2360
2361 ngx_memcpy(log, sc->connection->log, sizeof(ngx_log_t));
2362
2363 log->data = ctx;
2364
2365 ngx_memzero(rev, sizeof(ngx_event_t));
2366
2367 rev->data = fc;
2368 rev->ready = 1;
2369 rev->handler = ngx_http_spdy_close_stream_handler;
2370 rev->log = log;
2371
2372 ngx_memcpy(wev, rev, sizeof(ngx_event_t));
2373
2374 wev->write = 1;
2375
2376 ngx_memcpy(fc, sc->connection, sizeof(ngx_connection_t));
2377
2378 fc->data = sc->http_connection;
2379 fc->read = rev;
2380 fc->write = wev;
2381 fc->sent = 0;
2382 fc->log = log;
2383 fc->buffered = 0;
2384 fc->sndlowat = 1;
2385 fc->tcp_nodelay = NGX_TCP_NODELAY_DISABLED;
2386
2387 r = ngx_http_create_request(fc);
2388 if (r == NULL) {
2389     return NULL;
2390 }
2391
2392 r->valid_location = 1;
2393
2394 fc->data = r;
2395 sc->connection->requests++;
2396
2397 cscf = ngx_http_get_module_srv_conf(r, ngx_http_core_module);
2398
2399 r->header_in = ngx_create_temp_buf(r->pool,
2400                                   cscf->client_header_buffer_size);
2401 if (r->header_in == NULL) {
2402     ngx_http_free_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
2403     return NULL;
2404 }
2405
2406 r->headers_in.connection_type = NGX_HTTP_CONNECTION_CLOSE;
2407
2408 stream = ngx_palloc(r->pool, sizeof(ngx_http_spdy_stream_t));
2409 if (stream == NULL) {
2410     ngx_http_free_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
2411     return NULL;
2412 }
2413
2414 r->spdy_stream = stream;
2415
2416 stream->id = id;
2417 stream->request = r;
2418 stream->connection = sc;
2419
2420 stream->send_window = sc->init_window;
2421 stream->recv_window = NGX_SPDY_STREAM_WINDOW;
2422
2423 stream->priority = priority;
2424

```

```

2425     sscf = ngx_http_get_module_srv_conf(r, ngx_http_spdy_module);
2426
2427     index = ngx_http_spdy_stream_index(sscf, id);
2428
2429     stream->index = sc->streams_index[index];
2430     sc->streams_index[index] = stream;
2431
2432     sc->processing++;
2433
2434     return stream;
2435 }
2436
2437
2438 static ngx_http_spdy_stream_t *
2439 ngx_http_spdy_get_stream_by_id(ngx_http_spdy_connection_t *sc,
2440     ngx_uint_t sid)
2441 {
2442     ngx_http_spdy_stream_t *stream;
2443     ngx_http_spdy_srv_conf_t *sscf;
2444
2445     sscf = ngx_http_get_module_srv_conf(sc->http_connection->conf_ctx,
2446         ngx_http_spdy_module);
2447
2448     stream = sc->streams_index[ngx_http_spdy_stream_index(sscf, sid)];
2449
2450     while (stream) {
2451         if (stream->id == sid) {
2452             return stream;
2453         }
2454
2455         stream = stream->index;
2456     }
2457
2458     return NULL;
2459 }
2460
2461
2462 static ngx_int_t
2463 ngx_http_spdy_parse_header(ngx_http_request_t *r)
2464 {
2465     u_char *p, *end, ch;
2466     ngx_uint_t hash;
2467     ngx_http_core_srv_conf_t *cscf;
2468
2469     enum {
2470         sw_name_len = 0,
2471         sw_name,
2472         sw_value_len,
2473         sw_value
2474     } state;
2475
2476     state = r->state;
2477
2478     p = r->header_in->pos;
2479     end = r->header_in->last;
2480
2481     switch (state) {
2482
2483     case sw_name_len:
2484
2485         if (end - p < NGX_SPDY_NV_NLEN_SIZE) {
2486             return NGX_AGAIN;
2487         }
2488
2489         r->lowercase_index = ngx_spdy_frame_parse_uint32(p);
2490
2491         if (r->lowercase_index == 0) {
2492             return NGX_ERROR;
2493         }
2494
2495         /* null-terminate the previous header value */
2496         *p = '\0';
2497
2498         p += NGX_SPDY_NV_NLEN_SIZE;
2499
2500         r->invalid_header = 0;

```

```

2501     state = sw_name;
2502
2503
2504     /* fall through */
2505
2506 case sw_name:
2507
2508     if ((ngx_uint_t) (end - p) < r->lowercase_index) {
2509         break;
2510     }
2511
2512     cscf = ngx_http_get_module_srv_conf(r, ngx_http_core_module);
2513
2514     r->header_name_start = p;
2515     r->header_name_end = p + r->lowercase_index;
2516
2517     if (p[0] == ':') {
2518         p++;
2519     }
2520
2521     hash = 0;
2522
2523     for ( /* void */ ; p != r->header_name_end; p++) {
2524
2525         ch = *p;
2526
2527         hash = ngx_hash(hash, ch);
2528
2529         if ((ch >= 'a' && ch <= 'z')
2530             || (ch == '-')
2531             || (ch >= '0' && ch <= '9')
2532             || (ch == '_' && cscf->underscores_in_headers))
2533         {
2534             continue;
2535         }
2536
2537         switch (ch) {
2538             case '\0':
2539             case LF:
2540             case CR:
2541             case ':':
2542                 ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
2543                     "client sent invalid header name: \"%s\"",
2544                     r->lowercase_index, r->header_name_start);
2545
2546                 return NGX_HTTP_PARSE_INVALID_HEADER;
2547             }
2548
2549             if (ch >= 'A' && ch <= 'Z') {
2550                 return NGX_ERROR;
2551             }
2552
2553             r->invalid_header = 1;
2554         }
2555
2556         r->header_hash = hash;
2557
2558         state = sw_value_len;
2559
2560         /* fall through */
2561
2562 case sw_value_len:
2563
2564     if (end - p < NGX_SPDY_NV_VLEN_SIZE) {
2565         break;
2566     }
2567
2568     r->lowercase_index = ngx_spdy_frame_parse_uint32(p);
2569
2570     /* null-terminate header name */
2571     *p = '\0';
2572
2573     p += NGX_SPDY_NV_VLEN_SIZE;
2574
2575     state = sw_value;
2576

```

```

2577     /* fall through */
2578
2579 case sw_value:
2580
2581     if ((ngx_uint_t) (end - p) < r->lowcase_index) {
2582         break;
2583     }
2584
2585     r->header_start = p;
2586
2587     while (r->lowcase_index--) {
2588         ch = *p;
2589
2590         if (ch == '\0') {
2591
2592             if (p == r->header_start) {
2593                 return NGX_ERROR;
2594             }
2595
2596             r->header_end = p;
2597             r->header_in->pos = p + 1;
2598
2599             r->state = sw_value;
2600
2601             return NGX_OK;
2602         }
2603
2604         if (ch == CR || ch == LF) {
2605             ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
2606                 "client sent header \"%s\" with "
2607                 "invalid value: \"%s\\%c...\"",
2608                 r->header_name_end - r->header_name_start,
2609                 r->header_name_start,
2610                 p - r->header_start,
2611                 r->header_start,
2612                 ch == CR ? 'r' : 'n');
2613
2614             return NGX_HTTP_PARSE_INVALID_HEADER;
2615         }
2616
2617         p++;
2618     }
2619
2620     r->header_end = p;
2621     r->header_in->pos = p;
2622
2623     r->state = 0;
2624
2625     return NGX_DONE;
2626 }
2627
2628 r->header_in->pos = p;
2629 r->state = state;
2630
2631 return NGX_AGAIN;
2632 }
2633
2634
2635 static ngx_int_t
2636 ngx_http_spdy_alloc_large_header_buffer(ngx_http_request_t *r)
2637 {
2638     u_char          *old, *new, *p;
2639     size_t          rest;
2640     ngx_buf_t       *buf;
2641     ngx_http_spdy_stream_t *stream;
2642     ngx_http_core_srv_conf_t *cscf;
2643
2644     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
2645         "spdy alloc large header buffer");
2646
2647     stream = r->spdy_stream;
2648
2649     cscf = ngx_http_get_module_srv_conf(r, ngx_http_core_module);
2650
2651     if (stream->header_buffers
2652         == (ngx_uint_t) cscf->large_client_header_buffers.num)

```

```

2653 {
2654     ngx\_log\_error(NGX_LOG_INFO, r->connection->log, 0,
2655         "client sent too large request");
2656
2657     return NGX\_DECLINED;
2658 }
2659
2660 rest = r->header_in->last - r->header_in->pos;
2661
2662 /*
2663  * One more byte is needed for null-termination
2664  * and another one for further progress.
2665  */
2666 if (rest > cscf->large_client_header_buffers.size - 2) {
2667     p = r->header_in->pos;
2668
2669     if (rest > NGX\_MAX\_ERROR\_STR - 300) {
2670         rest = NGX\_MAX\_ERROR\_STR - 300;
2671     }
2672
2673     ngx\_log\_error(NGX_LOG_INFO, r->connection->log, 0,
2674         "client sent too long header name or value: \"%*s...\"",
2675         rest, p);
2676
2677     return NGX\_DECLINED;
2678 }
2679
2680 buf = ngx\_create\_temp\_buf(r->pool, cscf->large_client_header_buffers.size);
2681 if (buf == NULL) {
2682     return NGX\_ERROR;
2683 }
2684
2685 ngx\_log\_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
2686     "spdy large header alloc: %p %uz",
2687     buf->pos, buf->end - buf->last);
2688
2689 old = r->header_in->pos;
2690 new = buf->pos;
2691
2692 if (rest) {
2693     buf->last = ngx\_cpymem(new, old, rest);
2694 }
2695
2696 r->header_in = buf;
2697
2698 stream->header_buffers++;
2699
2700 return NGX\_OK;
2701 }
2702
2703
2704 static ngx\_int\_t
2705 ngx\_http\_spdy\_handle\_request\_header(ngx\_http\_request\_t *r)
2706 {
2707     ngx\_uint\_t i;
2708     ngx\_table\_elt\_t *h;
2709     ngx\_http\_core\_srv\_conf\_t *cscf;
2710     ngx\_http\_spdy\_request\_header\_t *sh;
2711
2712     if (r->invalid_header) {
2713         cscf = ngx\_http\_get\_module\_srv\_conf(r, ngx\_http\_core\_module);
2714
2715         if (cscf->ignore_invalid_headers) {
2716             ngx\_log\_error(NGX_LOG_INFO, r->connection->log, 0,
2717                 "client sent invalid header: \"%*s\"",
2718                 r->header_end - r->header_name_start,
2719                 r->header_name_start);
2720             return NGX\_OK;
2721         }
2722     }
2723
2724
2725     if (r->header_name_start[0] == ':') {
2726         r->header_name_start++;
2727
2728         for (i = 0; i < NGX\_SPDY\_REQUEST\_HEADERS; i++) {

```

```

2729     sh = &ngx_http_spdy_request_headers[i];
2730
2731     if (sh->hash != r->header_hash
2732         || sh->len != r->header_name_end - r->header_name_start
2733         || ngx_strncmp(sh->header, r->header_name_start, sh->len) != 0)
2734     {
2735         continue;
2736     }
2737
2738     return sh->handler(r);
2739 }
2740
2741 ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
2742             "client sent invalid header name: \"%s\"",
2743             r->header_end - r->header_name_start,
2744             r->header_name_start);
2745
2746     return NGX_HTTP_PARSE_INVALID_HEADER;
2747 }
2748
2749 h = ngx_list_push(&r->headers_in.headers);
2750 if (h == NULL) {
2751     return NGX_ERROR;
2752 }
2753
2754 h->hash = r->header_hash;
2755
2756 h->key.len = r->header_name_end - r->header_name_start;
2757 h->key.data = r->header_name_start;
2758
2759 h->value.len = r->header_end - r->header_start;
2760 h->value.data = r->header_start;
2761
2762 h->lowercase_key = h->key.data;
2763
2764     return NGX_OK;
2765 }
2766
2767
2768 void
2769 ngx_http_spdy_request_headers_init(void)
2770 {
2771     ngx_uint_t          i;
2772     ngx_http_spdy_request_header_t *h;
2773
2774     for (i = 0; i < NGX_SPDY_REQUEST_HEADERS; i++) {
2775         h = &ngx_http_spdy_request_headers[i];
2776         h->hash = ngx_hash_key(h->header, h->len);
2777     }
2778 }
2779
2780
2781 static ngx_int_t
2782 ngx_http_spdy_parse_method(ngx_http_request_t *r)
2783 {
2784     size_t          k, len;
2785     ngx_uint_t      n;
2786     const u_char    *p, *m;
2787
2788     /*
2789      * This array takes less than 256 sequential bytes,
2790      * and if typical CPU cache line size is 64 bytes,
2791      * it is prefetched for 4 load operations.
2792      */
2793     static const struct {
2794         u_char          len;
2795         const u_char    method[11];
2796         uint32_t        value;
2797     } tests[] = {
2798         { 3, "GET",          NGX_HTTP_GET },
2799         { 4, "POST",        NGX_HTTP_POST },
2800         { 4, "HEAD",        NGX_HTTP_HEAD },
2801         { 7, "OPTIONS",     NGX_HTTP_OPTIONS },
2802         { 8, "PROPFIND",    NGX_HTTP_PROPFIND },
2803         { 3, "PUT",         NGX_HTTP_PUT },
2804         { 5, "MKCOL",       NGX_HTTP_MKCOL },

```

```

2805     { 6, "DELETE",    NGX\_HTTP\_DELETE },
2806     { 4, "COPY",      NGX\_HTTP\_COPY },
2807     { 4, "MOVE",      NGX\_HTTP\_MOVE },
2808     { 9, "PROPPATCH", NGX\_HTTP\_PROPPATCH },
2809     { 4, "LOCK",      NGX\_HTTP\_LOCK },
2810     { 6, "UNLOCK",    NGX\_HTTP\_UNLOCK },
2811     { 5, "PATCH",    NGX\_HTTP\_PATCH },
2812     { 5, "TRACE",     NGX\_HTTP\_TRACE }
2813 }, *test;
2814
2815 if (r->method_name.len) {
2816     ngx\_log\_error(NGX\_LOG\_INFO, r->connection->log, 0,
2817         "client sent duplicate :method header");
2818
2819     return NGX\_HTTP\_PARSE\_INVALID\_HEADER;
2820 }
2821
2822 len = r->header_end - r->header_start;
2823
2824 r->method_name.len = len;
2825 r->method_name.data = r->header_start;
2826
2827 test = tests;
2828 n = sizeof(tests) / sizeof(tests[0]);
2829
2830 do {
2831     if (len == test->len) {
2832         p = r->method_name.data;
2833         m = test->method;
2834         k = len;
2835
2836         do {
2837             if (*p++ != *m++) {
2838                 goto next;
2839             }
2840         } while (--k);
2841
2842         r->method = test->value;
2843         return NGX\_OK;
2844     }
2845
2846 next:
2847     test++;
2848
2849 } while (--n);
2850
2851 p = r->method_name.data;
2852
2853 do {
2854     if ((*p < 'A' || *p > 'Z') && *p != '_') {
2855         ngx\_log\_error(NGX\_LOG\_INFO, r->connection->log, 0,
2856             "client sent invalid method: \"%V\"",
2857             &r->method_name);
2858
2859         return NGX\_HTTP\_PARSE\_INVALID\_HEADER;
2860     }
2861
2862     p++;
2863
2864 } while (--len);
2865
2866 return NGX\_OK;
2867 }
2868
2869
2870 static ngx\_int\_t
2871 ngx\_http\_spdy\_parse\_scheme(ngx\_http\_request\_t *r)
2872 {
2873     if (r->schema_start) {
2874         ngx\_log\_error(NGX\_LOG\_INFO, r->connection->log, 0,
2875             "client sent duplicate :schema header");
2876
2877         return NGX\_HTTP\_PARSE\_INVALID\_HEADER;
2878     }
2879
2880     r->schema_start = r->header_start;

```

```

2881     r->schema_end = r->header_end;
2882
2883     return NGX_OK;
2884 }
2885
2886
2887 static ngx_int_t
2888 ngx_http_spdy_parse_host(ngx_http_request_t *r)
2889 {
2890     ngx_table_elt_t *h;
2891
2892     if (r->headers_in.host) {
2893         ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
2894             "client sent duplicate :host header");
2895
2896         return NGX_HTTP_PARSE_INVALID_HEADER;
2897     }
2898
2899     h = ngx_list_push(&r->headers_in.headers);
2900     if (h == NULL) {
2901         return NGX_ERROR;
2902     }
2903
2904     r->headers_in.host = h;
2905
2906     h->hash = r->header_hash;
2907
2908     h->key.len = r->header_name_end - r->header_name_start;
2909     h->key.data = r->header_name_start;
2910
2911     h->value.len = r->header_end - r->header_start;
2912     h->value.data = r->header_start;
2913
2914     h->lowercase_key = h->key.data;
2915
2916     return NGX_OK;
2917 }
2918
2919
2920 static ngx_int_t
2921 ngx_http_spdy_parse_path(ngx_http_request_t *r)
2922 {
2923     if (r->unparsed_uri.len) {
2924         ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
2925             "client sent duplicate :path header");
2926
2927         return NGX_HTTP_PARSE_INVALID_HEADER;
2928     }
2929
2930     r->uri_start = r->header_start;
2931     r->uri_end = r->header_end;
2932
2933     if (ngx_http_parse_uri(r) != NGX_OK) {
2934         ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
2935             "client sent invalid URI: \"%*s\"",
2936             r->uri_end - r->uri_start, r->uri_start);
2937
2938         return NGX_HTTP_PARSE_INVALID_HEADER;
2939     }
2940
2941     if (ngx_http_process_request_uri(r) != NGX_OK) {
2942         /*
2943          * request has been finalized already
2944          * in ngx_http_process_request_uri()
2945          */
2946         return NGX_ABORT;
2947     }
2948
2949     return NGX_OK;
2950 }
2951
2952
2953 static ngx_int_t
2954 ngx_http_spdy_parse_version(ngx_http_request_t *r)
2955 {
2956     u_char *p, ch;

```



```

2957
2958 if (r->http_protocol.len) {
2959     ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
2960         "client sent duplicate :version header");
2961
2962     return NGX_HTTP_PARSE_INVALID_HEADER;
2963 }
2964
2965 p = r->header_start;
2966
2967 if (r->header_end - p < 8 || !(ngx_str5cmp(p, 'H', 'T', 'T', 'P', '/')) {
2968     goto invalid;
2969 }
2970
2971 ch = *(p + 5);
2972
2973 if (ch < '1' || ch > '9') {
2974     goto invalid;
2975 }
2976
2977 r->http_major = ch - '0';
2978
2979 for (p += 6; p != r->header_end - 2; p++) {
2980
2981     ch = *p;
2982
2983     if (ch == '.') {
2984         break;
2985     }
2986
2987     if (ch < '0' || ch > '9') {
2988         goto invalid;
2989     }
2990
2991     r->http_major = r->http_major * 10 + ch - '0';
2992 }
2993
2994 if (*p != '.') {
2995     goto invalid;
2996 }
2997
2998 ch = *(p + 1);
2999
3000 if (ch < '0' || ch > '9') {
3001     goto invalid;
3002 }
3003
3004 r->http_minor = ch - '0';
3005
3006 for (p += 2; p != r->header_end; p++) {
3007
3008     ch = *p;
3009
3010     if (ch < '0' || ch > '9') {
3011         goto invalid;
3012     }
3013
3014     r->http_minor = r->http_minor * 10 + ch - '0';
3015 }
3016
3017 r->http_protocol.len = r->header_end - r->header_start;
3018 r->http_protocol.data = r->header_start;
3019 r->http_version = r->http_major * 1000 + r->http_minor;
3020
3021 return NGX_OK;
3022
3023 invalid:
3024
3025 ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
3026     "client sent invalid http version: \"%s\"",
3027     r->header_end - r->header_start, r->header_start);
3028
3029 return NGX_HTTP_PARSE_INVALID_HEADER;
3030 }
3031
3032

```

```

3033 static ngx_int_t
3034 ngx_http_spdy_construct_request_line(ngx_http_request_t *r)
3035 {
3036     u_char *p;
3037
3038     if (r->method_name.len == 0
3039         || r->unparsed_uri.len == 0
3040         || r->http_protocol.len == 0)
3041     {
3042         ngx_http_finalize_request(r, NGX_HTTP_BAD_REQUEST);
3043         return NGX_ERROR;
3044     }
3045
3046     r->request_line.len = r->method_name.len + 1
3047                         + r->unparsed_uri.len + 1
3048                         + r->http_protocol.len;
3049
3050     p = ngx_pnalloc(r->pool, r->request_line.len + 1);
3051     if (p == NULL) {
3052         ngx_http_spdy_close_stream(r->spdy_stream,
3053                                     NGX_HTTP_INTERNAL_SERVER_ERROR);
3054         return NGX_ERROR;
3055     }
3056
3057     r->request_line.data = p;
3058
3059     p = ngx_cpymem(p, r->method_name.data, r->method_name.len);
3060
3061     *p++ = ' ';
3062
3063     p = ngx_cpymem(p, r->unparsed_uri.data, r->unparsed_uri.len);
3064
3065     *p++ = ' ';
3066
3067     ngx_memcpy(p, r->http_protocol.data, r->http_protocol.len + 1);
3068
3069     /* some modules expect the space character after method name */
3070     r->method_name.data = r->request_line.data;
3071
3072     return NGX_OK;
3073 }
3074
3075
3076 static void
3077 ngx_http_spdy_run_request(ngx_http_request_t *r)
3078 {
3079     ngx_uint_t          i;
3080     ngx_list_part_t    *part;
3081     ngx_table_elt_t    *h;
3082     ngx_http_header_t  *hh;
3083     ngx_http_core_main_conf_t *cmcf;
3084
3085     if (ngx_http_spdy_construct_request_line(r) != NGX_OK) {
3086         return;
3087     }
3088
3089     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
3090                  "spdy http request line: \"%V\"", &r->request_line);
3091
3092     cmcf = ngx_http_get_module_main_conf(r, ngx_http_core_module);
3093
3094     part = &r->headers_in.headers.part;
3095     h = part->elts;
3096
3097     for (i = 0 ;; i++) {
3098
3099         if (i >= part->nelts) {
3100             if (part->next == NULL) {
3101                 break;
3102             }
3103
3104             part = part->next;
3105             h = part->elts;
3106             i = 0;
3107         }
3108

```

```

3109     hh = ngx_hash_find(&cmcf->headers_in_hash, h[i].hash,
3110                       h[i].lowercase_key, h[i].key.len);
3111
3112     if (hh && hh->handler(r, &h[i], hh->offset) != NGX_OK) {
3113         return;
3114     }
3115
3116     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
3117                  "spdy http header: \"%V: %V\"", &h[i].key, &h[i].value);
3118 }
3119
3120 r->http_state = NGX_HTTP_PROCESS_REQUEST_STATE;
3121
3122 if (ngx_http_process_request_header(r) != NGX_OK) {
3123     return;
3124 }
3125
3126 if (r->headers_in.content_length_n > 0 && r->spdy_stream->in_closed) {
3127     ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
3128                  "client prematurely closed stream");
3129
3130     r->spdy_stream->skip_data = NGX_SPDY_DATA_ERROR;
3131
3132     ngx_http_finalize_request(r, NGX_HTTP_BAD_REQUEST);
3133     return;
3134 }
3135
3136 ngx_http_process_request(r);
3137 }
3138
3139
3140 static ngx_int_t
3141 ngx_http_spdy_init_request_body(ngx_http_request_t *r)
3142 {
3143     ngx_buf_t          *buf;
3144     ngx_temp_file_t    *tf;
3145     ngx_http_request_body_t *rb;
3146     ngx_http_core_loc_conf_t *clcf;
3147
3148     rb = ngx_palloc(r->pool, sizeof(ngx_http_request_body_t));
3149     if (rb == NULL) {
3150         return NGX_ERROR;
3151     }
3152
3153     r->request_body = rb;
3154
3155     if (r->spdy_stream->in_closed) {
3156         return NGX_OK;
3157     }
3158
3159     rb->rest = r->headers_in.content_length_n;
3160
3161     clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
3162
3163     if (r->request_body_in_file_only
3164         || rb->rest > (off_t) clcf->client_body_buffer_size
3165         || rb->rest < 0)
3166     {
3167         tf = ngx_palloc(r->pool, sizeof(ngx_temp_file_t));
3168         if (tf == NULL) {
3169             return NGX_ERROR;
3170         }
3171
3172         tf->file.fd = NGX_INVALID_FILE;
3173         tf->file.log = r->connection->log;
3174         tf->path = clcf->client_body_temp_path;
3175         tf->pool = r->pool;
3176         tf->warn = "a client request body is buffered to a temporary file";
3177         tf->log_level = r->request_body_file_log_level;
3178         tf->persistent = r->request_body_in_persistent_file;
3179         tf->clean = r->request_body_in_clean_file;
3180
3181         if (r->request_body_file_group_access) {
3182             tf->access = 0660;
3183         }
3184     }

```

```

3185     rb->temp_file = tf;
3186
3187     if (r->spdy_stream->in_closed
3188         && ngx\_create\_temp\_file(&tf->file, tf->path, tf->pool,
3189                               tf->persistent, tf->clean, tf->access)
3190             != NGX\_OK)
3191     {
3192         return NGX\_ERROR;
3193     }
3194
3195     buf = ngx\_calloc\_buf(r->pool);
3196     if (buf == NULL) {
3197         return NGX\_ERROR;
3198     }
3199
3200 } else {
3201
3202     if (rb->rest == 0) {
3203         return NGX\_OK;
3204     }
3205
3206     buf = ngx\_create\_temp\_buf(r->pool, (size_t) rb->rest);
3207     if (buf == NULL) {
3208         return NGX\_ERROR;
3209     }
3210 }
3211
3212 rb->buf = buf;
3213
3214 rb->bufs = ngx\_alloc\_chain\_link(r->pool);
3215 if (rb->bufs == NULL) {
3216     return NGX\_ERROR;
3217 }
3218
3219 rb->bufs->buf = buf;
3220 rb->bufs->next = NULL;
3221
3222 rb->rest = 0;
3223
3224 return NGX\_OK;
3225 }
3226
3227
3228 ngx\_int\_t
3229 ngx\_http\_spdy\_read\_request\_body(ngx\_http\_request\_t *r,
3230 ngx\_http\_client\_body\_handler\_pt post_handler)
3231 {
3232     ngx\_http\_spdy\_stream\_t *stream;
3233
3234     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
3235                   "spdy read request body");
3236
3237     stream = r->spdy_stream;
3238
3239     switch (stream->skip_data) {
3240
3241     case NGX\_SPDY\_DATA\_DISCARD:
3242         post_handler(r);
3243         return NGX\_OK;
3244
3245     case NGX\_SPDY\_DATA\_ERROR:
3246         if (r->headers_in.content_length_n == -1) {
3247             return NGX\_HTTP\_REQUEST\_ENTITY\_TOO\_LARGE;
3248         } else {
3249             return NGX\_HTTP\_BAD\_REQUEST;
3250         }
3251
3252     case NGX\_SPDY\_DATA\_INTERNAL\_ERROR:
3253         return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
3254     }
3255
3256     if (!r->request_body && ngx\_http\_spdy\_init\_request\_body(r) != NGX\_OK) {
3257         stream->skip_data = NGX\_SPDY\_DATA\_INTERNAL\_ERROR;
3258         return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
3259     }
3260

```

```

3261     if (stream->in_closed) {
3262         post_handler(r);
3263         return NGX\_OK;
3264     }
3265
3266     r->request_body->post_handler = post_handler;
3267
3268     r->read_event_handler = ngx\_http\_test\_reading;
3269     r->write_event_handler = ngx\_http\_request\_empty\_handler;
3270
3271     return NGX\_AGAIN;
3272 }
3273
3274
3275 static ngx\_int\_t
3276 ngx\_http\_spdy\_terminate\_stream(ngx\_http\_spdy\_connection\_t *sc,
3277 ngx\_http\_spdy\_stream\_t *stream, ngx\_uint\_t status)
3278 {
3279     ngx\_event\_t *rev;
3280     ngx\_connection\_t *fc;
3281
3282     if (ngx\_http\_spdy\_send\_rst\_stream(sc, stream->id, status,
3283 NGX\_SPDY\_HIGHEST\_PRIORITY)
3284 == NGX\_ERROR)
3285     {
3286         return NGX\_ERROR;
3287     }
3288
3289     stream->out_closed = 1;
3290
3291     fc = stream->request->connection;
3292     fc->error = 1;
3293
3294     rev = fc->read;
3295     rev->handler(rev);
3296
3297     return NGX\_OK;
3298 }
3299
3300
3301 static void
3302 ngx\_http\_spdy\_close\_stream\_handler(ngx\_event\_t *ev)
3303 {
3304     ngx\_connection\_t *fc;
3305     ngx\_http\_request\_t *r;
3306
3307     fc = ev->data;
3308     r = fc->data;
3309
3310     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
3311 "spdy close stream handler");
3312
3313     ngx\_http\_spdy\_close\_stream(r->spdy_stream, 0);
3314 }
3315
3316
3317 void
3318 ngx\_http\_spdy\_close\_stream(ngx\_http\_spdy\_stream\_t *stream, ngx\_int\_t rc)
3319 {
3320     int tcp_nodelay;
3321     ngx\_event\_t *ev;
3322     ngx\_connection\_t *c, *fc;
3323     ngx\_http\_core\_loc\_conf\_t *clcf;
3324     ngx\_http\_spdy\_stream\_t **index, *s;
3325     ngx\_http\_spdy\_srv\_conf\_t *sscf;
3326     ngx\_http\_spdy\_connection\_t *sc;
3327
3328     sc = stream->connection;
3329
3330     ngx\_log\_debug3(NGX\_LOG\_DEBUG\_HTTP, sc->connection->log, 0,
3331 "spdy close stream %ui, queued %ui, processing %ui",
3332 stream->id, stream->queued, sc->processing);
3333
3334     fc = stream->request->connection;
3335
3336     if (stream->queued) {

```

```

3337     fc->write->handler = ngx\_http\_spdy\_close\_stream\_handler;
3338     return;
3339 }
3340
3341 if (!stream->out_closed) {
3342     if (ngx\_http\_spdy\_send\_rst\_stream(sc, stream->id,
3343                                     NGX\_SPDY\_INTERNAL\_ERROR,
3344                                     stream->priority)
3345         != NGX\_OK)
3346     {
3347         sc->connection->error = 1;
3348     }
3349
3350 } else {
3351     c = sc->connection;
3352
3353     if (c->tcp_nopush == NGX\_TCP\_NOPUSH\_SET) {
3354         if (ngx\_tcp\_push(c->fd) == -1) {
3355             ngx\_connection\_error(c, ngx\_socket\_errno,
3356                                 ngx\_tcp\_push\_n " failed");
3357             c->error = 1;
3358             tcp_nodelay = 0;
3359
3360         } else {
3361             c->tcp_nopush = NGX\_TCP\_NOPUSH\_UNSET;
3362             tcp_nodelay = ngx\_tcp\_nodelay\_and\_tcp\_nopush ? 1 : 0;
3363         }
3364
3365     } else {
3366         tcp_nodelay = 1;
3367     }
3368
3369     clcf = ngx\_http\_get\_module\_loc\_conf(stream->request,
3370                                         ngx\_http\_core\_module);
3371
3372     if (tcp_nodelay
3373         && clcf->tcp_nodelay
3374         && c->tcp_nodelay == NGX\_TCP\_NODELAY\_UNSET)
3375     {
3376         ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, c->log, 0, "tcp_nodelay");
3377
3378         if (setsockopt(c->fd, IPPROTO_TCP, TCP_NODELAY,
3379                       (const void *) &tcp_nodelay, sizeof(int))
3380             == -1)
3381         {
3382             #if (NGX_SOLARIS)
3383                 /* Solaris returns EINVAL if a socket has been shut down */
3384                 c->log_error = NGX\_ERROR\_IGNORE\_EINVAL;
3385             #endif
3386
3387             ngx\_connection\_error(c, ngx\_socket\_errno,
3388                                 "setsockopt(TCP_NODELAY) failed");
3389
3390             c->log_error = NGX\_ERROR\_INFO;
3391             c->error = 1;
3392
3393         } else {
3394             c->tcp_nodelay = NGX\_TCP\_NODELAY\_SET;
3395         }
3396     }
3397 }
3398
3399 if (sc->stream == stream) {
3400     sc->stream = NULL;
3401 }
3402
3403 sscf = ngx\_http\_get\_module\_srv\_conf(sc->http_connection->conf_ctx,
3404                                     ngx\_http\_spdy\_module);
3405
3406 index = sc->streams_index + ngx\_http\_spdy\_stream\_index(sscf, stream->id);
3407
3408 for ( ;; ) {
3409     s = *index;
3410
3411     if (s == NULL) {
3412         break;

```

```

3413     }
3414
3415     if (s == stream) {
3416         *index = s->index;
3417         break;
3418     }
3419
3420     index = &s->index;
3421 }
3422
3423 ngx\_http\_free\_request(stream->request, rc);
3424
3425 ev = fc->read;
3426
3427 if (ev->active || ev->disabled) {
3428     ngx\_log\_error(NGX_LOG_ALERT, sc->connection->log, 0,
3429         "fake read event was activated");
3430 }
3431
3432 if (ev->timer_set) {
3433     ngx\_del\_timer(ev);
3434 }
3435
3436 if (ev->posted) {
3437     ngx\_delete\_posted\_event(ev);
3438 }
3439
3440 ev = fc->write;
3441
3442 if (ev->active || ev->disabled) {
3443     ngx\_log\_error(NGX_LOG_ALERT, sc->connection->log, 0,
3444         "fake write event was activated");
3445 }
3446
3447 if (ev->timer_set) {
3448     ngx\_del\_timer(ev);
3449 }
3450
3451 if (ev->posted) {
3452     ngx\_delete\_posted\_event(ev);
3453 }
3454
3455 fc->data = sc->free_fake_connections;
3456 sc->free_fake_connections = fc;
3457
3458 sc->processing--;
3459
3460 if (sc->processing || sc->blocked) {
3461     return;
3462 }
3463
3464 ev = sc->connection->read;
3465
3466 ev->handler = ngx\_http\_spdy\_handle\_connection\_handler;
3467 ngx\_post\_event(ev, &ngx\_posted\_events);
3468 }
3469
3470
3471 static void
3472 ngx\_http\_spdy\_handle\_connection\_handler(ngx\_event\_t *rev)
3473 {
3474     ngx\_connection\_t *c;
3475
3476     rev->handler = ngx\_http\_spdy\_read\_handler;
3477
3478     if (rev->ready) {
3479         ngx\_http\_spdy\_read\_handler(rev);
3480         return;
3481     }
3482
3483     c = rev->data;
3484
3485     ngx\_http\_spdy\_handle\_connection(c->data);
3486 }
3487
3488

```

```

3489 static void
3490 ngx_http_spdy_keepalive_handler(ngx_event_t *rev)
3491 {
3492     ngx_connection_t          *c;
3493     ngx_http_spdy_srv_conf_t  *sscf;
3494     ngx_http_spdy_connection_t *sc;
3495
3496     c = rev->data;
3497
3498     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, c->log, 0, "spdy keepalive handler");
3499
3500     if (rev->timedout || c->close) {
3501         ngx_http_close_connection(c);
3502         return;
3503     }
3504
3505     #if (NGX_HAVE_KQUEUE)
3506
3507     if (ngx_event_flags & NGX_USE_KQUEUE_EVENT) {
3508         if (rev->pending_eof) {
3509             c->log->handler = NULL;
3510             ngx_log_error(NGX_LOG_INFO, c->log, rev->kq_errno,
3511                 "kevent() reported that client %V closed "
3512                 "keepalive connection", &c->addr_text);
3513         }
3514         #if (NGX_HTTP_SSL)
3515         if (c->ssl) {
3516             c->ssl->no_send_shutdown = 1;
3517         }
3518         #endif
3519         ngx_http_close_connection(c);
3520         return;
3521     }
3522
3523     #endif
3524
3525     c->destroyed = 0;
3526     c->idle = 0;
3527     ngx_reusable_connection(c, 0);
3528
3529     sc = c->data;
3530
3531     sscf = ngx_http_get_module_srv_conf(sc->http_connection->conf_ctx,
3532         ngx_http_spdy_module);
3533
3534     sc->pool = ngx_create_pool(sscf->pool_size, sc->connection->log);
3535     if (sc->pool == NULL) {
3536         ngx_http_close_connection(c);
3537         return;
3538     }
3539
3540     sc->streams_index = ngx_palloc(sc->pool,
3541         ngx_http_spdy_streams_index_size(sscf)
3542         * sizeof(ngx_http_spdy_stream_t *));
3543     if (sc->streams_index == NULL) {
3544         ngx_http_close_connection(c);
3545         return;
3546     }
3547
3548     c->write->handler = ngx_http_spdy_write_handler;
3549
3550     rev->handler = ngx_http_spdy_read_handler;
3551     ngx_http_spdy_read_handler(rev);
3552 }
3553
3554
3555 static void
3556 ngx_http_spdy_finalize_connection(ngx_http_spdy_connection_t *sc,
3557     ngx_int_t rc)
3558 {
3559     ngx_uint_t          i, size;
3560     ngx_event_t        *ev;
3561     ngx_connection_t   *c, *fc;
3562     ngx_http_request_t *r;
3563     ngx_http_spdy_stream_t *stream;
3564     ngx_http_spdy_srv_conf_t *sscf;

```



```

3565
3566     c = sc->connection;
3567
3568     if (!sc->processing) {
3569         ngx\_http\_close\_connection(c);
3570         return;
3571     }
3572
3573     c->error = 1;
3574     c->read->handler = ngx\_http\_empty\_handler;
3575     c->write->handler = ngx\_http\_empty\_handler;
3576
3577     sc->last_out = NULL;
3578
3579     sc->blocked = 1;
3580
3581     sscf = ngx\_http\_get\_module\_srv\_conf(sc->http_connection->conf_ctx,
3582                                         ngx\_http\_spdy\_module);
3583
3584     size = ngx\_http\_spdy\_streams\_index\_size(sscf);
3585
3586     for (i = 0; i < size; i++) {
3587         stream = sc->streams_index[i];
3588
3589         while (stream) {
3590             stream->handled = 0;
3591
3592             r = stream->request;
3593             fc = r->connection;
3594
3595             fc->error = 1;
3596
3597             if (stream->queued) {
3598                 stream->queued = 0;
3599
3600                 ev = fc->write;
3601                 ev->delayed = 0;
3602
3603             } else {
3604                 ev = fc->read;
3605             }
3606
3607             stream = stream->index;
3608
3609             ev->eof = 1;
3610             ev->handler(ev);
3611         }
3612     }
3613
3614     sc->blocked = 0;
3615
3616     if (sc->processing) {
3617         return;
3618     }
3619
3620     ngx\_http\_close\_connection(c);
3621 }
3622
3623
3624 static ngx\_int\_t
3625 ngx\_http\_spdy\_adjust\_windows(ngx\_http\_spdy\_connection\_t *sc, ssize\_t delta)
3626 {
3627     ngx\_uint\_t          i, size;
3628     ngx\_event\_t        *wev;
3629     ngx\_http\_spdy\_stream\_t *stream, *sn;
3630     ngx\_http\_spdy\_srv\_conf\_t *sscf;
3631
3632     sscf = ngx\_http\_get\_module\_srv\_conf(sc->http_connection->conf_ctx,
3633                                         ngx\_http\_spdy\_module);
3634
3635     size = ngx\_http\_spdy\_streams\_index\_size(sscf);
3636
3637     for (i = 0; i < size; i++) {
3638
3639         for (stream = sc->streams_index[i]; stream; stream = sn) {
3640             sn = stream->index;

```

```

3641     if (delta > 0
3642         && stream->send_window
3643             > (ssize_t) (NGX_SPDY_MAX_WINDOW - delta))
3644     {
3645         if (ngx_http_spdy_terminate_stream(sc, stream,
3646             NGX_SPDY_FLOW_CONTROL_ERROR)
3647             == NGX_ERROR)
3648         {
3649             return NGX_ERROR;
3650         }
3651
3652         continue;
3653     }
3654
3655     stream->send_window += delta;
3656
3657     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, sc->connection->log, 0,
3658         "spdy:%ui adjust window:%z",
3659         stream->id, stream->send_window);
3660
3661     if (stream->send_window > 0 && stream->exhausted) {
3662         stream->exhausted = 0;
3663
3664         wev = stream->request->connection->write;
3665
3666         if (!wev->timer_set) {
3667             wev->delayed = 0;
3668             wev->handler(wev);
3669         }
3670     }
3671 }
3672 }
3673 }
3674
3675 return NGX_OK;
3676 }
3677
3678
3679 static void
3680 ngx_http_spdy_pool_cleanup(void *data)
3681 {
3682     ngx_http_spdy_connection_t *sc = data;
3683
3684     if (sc->pool) {
3685         ngx_destroy_pool(sc->pool);
3686     }
3687 }
3688
3689
3690 static void *
3691 ngx_http_spdy_zalloc(void *opaque, u_int items, u_int size)
3692 {
3693     ngx_http_spdy_connection_t *sc = opaque;
3694
3695     return ngx_palloc(sc->connection->pool, items * size);
3696 }
3697
3698
3699 static void
3700 ngx_http_spdy_zfree(void *opaque, void *address)
3701 {
3702     #if 0
3703         ngx_http_spdy_connection_t *sc = opaque;
3704
3705         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, sc->connection->log, 0,
3706             "spdy zfree: %p", address);
3707     #endif
3708 }

```

[One Level Up](#)

[Top Level](#)

## src/http/nginx\_http\_spdy\_module.c - nginx-1.7.10

### Global variables defined

- [ngx\\_http\\_spdy\\_chunk\\_size\\_post](#)
- [ngx\\_http\\_spdy\\_commands](#)
- [ngx\\_http\\_spdy\\_headers\\_comp\\_bounds](#)
- [ngx\\_http\\_spdy\\_module](#)
- [ngx\\_http\\_spdy\\_module\\_ctx](#)
- [ngx\\_http\\_spdy\\_pool\\_size\\_post](#)
- [ngx\\_http\\_spdy\\_recv\\_buffer\\_size\\_post](#)
- [ngx\\_http\\_spdy\\_streams\\_index\\_mask\\_post](#)
- [ngx\\_http\\_spdy\\_vars](#)

### Functions defined

- [ngx\\_http\\_spdy\\_add\\_variables](#)
- [ngx\\_http\\_spdy\\_chunk\\_size](#)
- [ngx\\_http\\_spdy\\_create\\_loc\\_conf](#)
- [ngx\\_http\\_spdy\\_create\\_main\\_conf](#)
- [ngx\\_http\\_spdy\\_create\\_srv\\_conf](#)
- [ngx\\_http\\_spdy\\_init\\_main\\_conf](#)
- [ngx\\_http\\_spdy\\_merge\\_loc\\_conf](#)
- [ngx\\_http\\_spdy\\_merge\\_srv\\_conf](#)
- [ngx\\_http\\_spdy\\_module\\_init](#)
- [ngx\\_http\\_spdy\\_pool\\_size](#)
- [ngx\\_http\\_spdy\\_recv\\_buffer\\_size](#)
- [ngx\\_http\\_spdy\\_request\\_priority\\_variable](#)
- [ngx\\_http\\_spdy\\_streams\\_index\\_mask](#)
- [ngx\\_http\\_spdy\\_variable](#)

### Source code

```
1
2 /*
3  * Copyright (C) Nginx, Inc.
4  * Copyright (C) Valentin V. Bartenev
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
```

```

10 #include <ngx_http.h>
11 #include <ngx_http_spdy_module.h>
12
13
14 static ngx_int_t ngx_http_spdy_add_variables(ngx_conf_t *cf);
15
16 static ngx_int_t ngx_http_spdy_variable(ngx_http_request_t *r,
17     ngx_http_variable_value_t *v, uintptr_t data);
18 static ngx_int_t ngx_http_spdy_request_priority_variable(ngx_http_request_t *r,
19     ngx_http_variable_value_t *v, uintptr_t data);
20
21 static ngx_int_t ngx_http_spdy_module_init(ngx_cycle_t *cycle);
22
23 static void *ngx_http_spdy_create_main_conf(ngx_conf_t *cf);
24 static char *ngx_http_spdy_init_main_conf(ngx_conf_t *cf, void *conf);
25 static void *ngx_http_spdy_create_srv_conf(ngx_conf_t *cf);
26 static char *ngx_http_spdy_merge_srv_conf(ngx_conf_t *cf, void *parent,
27     void *child);
28 static void *ngx_http_spdy_create_loc_conf(ngx_conf_t *cf);
29 static char *ngx_http_spdy_merge_loc_conf(ngx_conf_t *cf, void *parent,
30     void *child);
31
32 static char *ngx_http_spdy_recv_buffer_size(ngx_conf_t *cf, void *post,
33     void *data);
34 static char *ngx_http_spdy_pool_size(ngx_conf_t *cf, void *post, void *data);
35 static char *ngx_http_spdy_streams_index_mask(ngx_conf_t *cf, void *post,
36     void *data);
37 static char *ngx_http_spdy_chunk_size(ngx_conf_t *cf, void *post, void *data);
38
39
40 static ngx_conf_num_bounds_t ngx_http_spdy_headers_comp_bounds = {
41     ngx_conf_check_num_bounds, 0, 9
42 };
43
44 static ngx_conf_post_t ngx_http_spdy_recv_buffer_size_post =
45     { ngx_http_spdy_recv_buffer_size };
46 static ngx_conf_post_t ngx_http_spdy_pool_size_post =
47     { ngx_http_spdy_pool_size };
48 static ngx_conf_post_t ngx_http_spdy_streams_index_mask_post =
49     { ngx_http_spdy_streams_index_mask };
50 static ngx_conf_post_t ngx_http_spdy_chunk_size_post =
51     { ngx_http_spdy_chunk_size };
52
53
54 static ngx_command_t ngx_http_spdy_commands[] = {
55
56     { ngx_string("spdy_recv_buffer_size"),
57         NGX_HTTP_MAIN_CONF|NGX_CONF TAKE1,
58         ngx_conf_set_size_slot,
59         NGX_HTTP_MAIN_CONF OFFSET,
60         offsetof(ngx_http_spdy_main_conf_t, recv_buffer_size),
61         &ngx_http_spdy_recv_buffer_size_post },
62
63     { ngx_string("spdy_pool_size"),
64         NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_CONF TAKE1,
65         ngx_conf_set_size_slot,
66         NGX_HTTP_SRV_CONF OFFSET,
67         offsetof(ngx_http_spdy_srv_conf_t, pool_size),
68         &ngx_http_spdy_pool_size_post },
69
70     { ngx_string("spdy_max_concurrent_streams"),
71         NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_CONF TAKE1,
72         ngx_conf_set_num_slot,
73         NGX_HTTP_SRV_CONF OFFSET,
74         offsetof(ngx_http_spdy_srv_conf_t, concurrent_streams),
75         NULL },
76
77     { ngx_string("spdy_streams_index_size"),
78         NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_CONF TAKE1,
79         ngx_conf_set_num_slot,
80         NGX_HTTP_SRV_CONF OFFSET,
81         offsetof(ngx_http_spdy_srv_conf_t, streams_index_mask),
82         &ngx_http_spdy_streams_index_mask_post },
83
84     { ngx_string("spdy_recv_timeout"),
85         NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_CONF TAKE1,

```

```

86     ngx_conf_set_msec_slot,
87     NGX_HTTP_SRV_CONF_OFFSET,
88     offsetof(ngx_http_spdy_srv_conf_t, recv_timeout),
89     NULL },
90
91     { ngx_string("spdy_keepalive_timeout"),
92       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_CONF TAKE1,
93       ngx_conf_set_msec_slot,
94       NGX_HTTP_SRV_CONF_OFFSET,
95       offsetof(ngx_http_spdy_srv_conf_t, keepalive_timeout),
96       NULL },
97
98     { ngx_string("spdy_headers_comp"),
99       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_CONF TAKE1,
100      ngx_conf_set_num_slot,
101      NGX_HTTP_SRV_CONF_OFFSET,
102      offsetof(ngx_http_spdy_srv_conf_t, headers_comp),
103      &ngx_http_spdy_headers_comp_bounds },
104
105     { ngx_string("spdy_chunk_size"),
106       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF TAKE1,
107       ngx_conf_set_size_slot,
108       NGX_HTTP_LOC_CONF_OFFSET,
109       offsetof(ngx_http_spdy_loc_conf_t, chunk_size),
110       &ngx_http_spdy_chunk_size_post },
111
112     ngx_null_command
113 };
114
115
116 static ngx_http_module_t ngx_http_spdy_module_ctx = {
117     ngx_http_spdy_add_variables,      /* preconfiguration */
118     NULL,                             /* postconfiguration */
119
120     ngx_http_spdy_create_main_conf,   /* create main configuration */
121     ngx_http_spdy_init_main_conf,    /* init main configuration */
122
123     ngx_http_spdy_create_srv_conf,    /* create server configuration */
124     ngx_http_spdy_merge_srv_conf,    /* merge server configuration */
125
126     ngx_http_spdy_create_loc_conf,    /* create location configuration */
127     ngx_http_spdy_merge_loc_conf     /* merge location configuration */
128 };
129
130
131 ngx_module_t ngx_http_spdy_module = {
132     NGX_MODULE_V1,
133     &ngx_http_spdy_module_ctx,      /* module context */
134     ngx_http_spdy_commands,         /* module directives */
135     NGX_HTTP_MODULE,                /* module type */
136     NULL,                            /* init master */
137     ngx_http_spdy_module_init,      /* init module */
138     NULL,                             /* init process */
139     NULL,                             /* init thread */
140     NULL,                             /* exit thread */
141     NULL,                             /* exit process */
142     NULL,                             /* exit master */
143     NGX_MODULE_V1_PADDING
144 };
145
146
147 static ngx_http_variable_t ngx_http_spdy_vars[] = {
148
149     { ngx_string("spdy"), NULL,
150       ngx_http_spdy_variable, 0, 0, 0 },
151
152     { ngx_string("spdy_request_priority"), NULL,
153       ngx_http_spdy_request_priority_variable, 0, 0, 0 },
154
155     { ngx_null_string, NULL, NULL, 0, 0, 0 }
156 };
157
158
159 static ngx_int_t
160 ngx_http_spdy_add_variables(ngx_conf_t *cf)
161 {

```

```

162     ngx_http_variable_t *var, *v;
163
164     for (v = ngx_http_spdy_vars; v->name.len; v++) {
165         var = ngx_http_add_variable(cf, &v->name, v->flags);
166         if (var == NULL) {
167             return NGX_ERROR;
168         }
169
170         var->get_handler = v->get_handler;
171         var->data = v->data;
172     }
173
174     return NGX_OK;
175 }
176
177
178 static ngx_int_t
179 ngx_http_spdy_variable(ngx_http_request_t *r,
180     ngx_http_variable_value_t *v, uintptr_t data)
181 {
182     if (r->spdy_stream) {
183         v->len = sizeof("3.1") - 1;
184         v->valid = 1;
185         v->no_cacheable = 0;
186         v->not_found = 0;
187         v->data = (u_char *) "3.1";
188
189         return NGX_OK;
190     }
191
192     *v = ngx_http_variable_null_value;
193
194     return NGX_OK;
195 }
196
197
198 static ngx_int_t
199 ngx_http_spdy_request_priority_variable(ngx_http_request_t *r,
200     ngx_http_variable_value_t *v, uintptr_t data)
201 {
202     if (r->spdy_stream) {
203         v->len = 1;
204         v->valid = 1;
205         v->no_cacheable = 0;
206         v->not_found = 0;
207
208         v->data = ngx_pnalloc(r->pool, 1);
209         if (v->data == NULL) {
210             return NGX_ERROR;
211         }
212
213         v->data[0] = '0' + (u_char) r->spdy_stream->priority;
214
215         return NGX_OK;
216     }
217
218     *v = ngx_http_variable_null_value;
219
220     return NGX_OK;
221 }
222
223
224 static ngx_int_t
225 ngx_http_spdy_module_init(ngx_cycle_t *cycle)
226 {
227     ngx_http_spdy_request_headers_init();
228
229     return NGX_OK;
230 }
231
232
233 static void *
234 ngx_http_spdy_create_main_conf(ngx_conf_t *cf)
235 {
236     ngx_http_spdy_main_conf_t *smcf;
237

```

```

238     smcf = ngx_palloc(cf->pool, sizeof(ngx_http_spdy_main_conf_t));
239     if (smcf == NULL) {
240         return NULL;
241     }
242
243     smcf->recv_buffer_size = NGX_CONF_UNSET_SIZE;
244
245     return smcf;
246 }
247
248
249 static char *
250 ngx_http_spdy_init_main_conf(ngx_conf_t *cf, void *conf)
251 {
252     ngx_http_spdy_main_conf_t *smcf = conf;
253
254     ngx_conf_init_size_value(smcf->recv_buffer_size, 256 * 1024);
255
256     return NGX_CONF_OK;
257 }
258
259
260 static void *
261 ngx_http_spdy_create_srv_conf(ngx_conf_t *cf)
262 {
263     ngx_http_spdy_srv_conf_t *sscf;
264
265     sscf = ngx_palloc(cf->pool, sizeof(ngx_http_spdy_srv_conf_t));
266     if (sscf == NULL) {
267         return NULL;
268     }
269
270     sscf->pool_size = NGX_CONF_UNSET_SIZE;
271
272     sscf->concurrent_streams = NGX_CONF_UNSET_UINT;
273     sscf->streams_index_mask = NGX_CONF_UNSET_UINT;
274
275     sscf->recv_timeout = NGX_CONF_UNSET_MSEC;
276     sscf->keepalive_timeout = NGX_CONF_UNSET_MSEC;
277
278     sscf->headers_comp = NGX_CONF_UNSET;
279
280     return sscf;
281 }
282
283
284 static char *
285 ngx_http_spdy_merge_srv_conf(ngx_conf_t *cf, void *parent, void *child)
286 {
287     ngx_http_spdy_srv_conf_t *prev = parent;
288     ngx_http_spdy_srv_conf_t *conf = child;
289
290     ngx_conf_merge_size_value(conf->pool_size, prev->pool_size, 4096);
291
292     ngx_conf_merge_uint_value(conf->concurrent_streams,
293                               prev->concurrent_streams, 100);
294
295     ngx_conf_merge_uint_value(conf->streams_index_mask,
296                               prev->streams_index_mask, 32 - 1);
297
298     ngx_conf_merge_msec_value(conf->recv_timeout,
299                               prev->recv_timeout, 30000);
300     ngx_conf_merge_msec_value(conf->keepalive_timeout,
301                               prev->keepalive_timeout, 180000);
302
303     ngx_conf_merge_value(conf->headers_comp, prev->headers_comp, 0);
304
305     return NGX_CONF_OK;
306 }
307
308
309 static void *
310 ngx_http_spdy_create_loc_conf(ngx_conf_t *cf)
311 {
312     ngx_http_spdy_loc_conf_t *slcf;
313

```

```

314     slcf = ngx_palloc(cf->pool, sizeof(ngx_http_spdy_loc_conf_t));
315     if (slcf == NULL) {
316         return NULL;
317     }
318
319     slcf->chunk_size = NGX_CONF_UNSET_SIZE;
320
321     return slcf;
322 }
323
324
325 static char *
326 ngx_http_spdy_merge_loc_conf(ngx_conf_t *cf, void *parent, void *child)
327 {
328     ngx_http_spdy_loc_conf_t *prev = parent;
329     ngx_http_spdy_loc_conf_t *conf = child;
330
331     ngx_conf_merge_size_value(conf->chunk_size, prev->chunk_size, 8 * 1024);
332
333     return NGX_CONF_OK;
334 }
335
336
337 static char *
338 ngx_http_spdy_recv_buffer_size(ngx_conf_t *cf, void *post, void *data)
339 {
340     size_t *sp = data;
341
342     if (*sp <= 2 * NGX_SPDY_STATE_BUFFER_SIZE) {
343         return "value is too small";
344     }
345
346     return NGX_CONF_OK;
347 }
348
349
350 static char *
351 ngx_http_spdy_pool_size(ngx_conf_t *cf, void *post, void *data)
352 {
353     size_t *sp = data;
354
355     if (*sp < NGX_MIN_POOL_SIZE) {
356         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
357             "the pool size must be no less than %uz",
358             NGX_MIN_POOL_SIZE);
359         return NGX_CONF_ERROR;
360     }
361
362     if (*sp % NGX_POOL_ALIGNMENT) {
363         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
364             "the pool size must be a multiple of %uz",
365             NGX_POOL_ALIGNMENT);
366         return NGX_CONF_ERROR;
367     }
368
369     return NGX_CONF_OK;
370 }
371
372
373 static char *
374 ngx_http_spdy_streams_index_mask(ngx_conf_t *cf, void *post, void *data)
375 {
376     ngx_uint_t *np = data;
377
378     ngx_uint_t mask;
379
380     mask = *np - 1;
381
382     if (*np == 0 || (*np & mask)) {
383         return "must be a power of two";
384     }
385
386     *np = mask;
387
388     return NGX_CONF_OK;
389 }

```



```
390
391
392 static char *
393 ngx_http_spdy_chunk_size(ngx_conf_t *cf, void *post, void *data)
394 {
395     size_t *sp = data;
396
397     if (*sp == 0) {
398         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
399             "the spdy chunk size cannot be zero");
400         return NGX_CONF_ERROR;
401     }
402
403     if (*sp > NGX_SPDY_MAX_FRAME_SIZE) {
404         *sp = NGX_SPDY_MAX_FRAME_SIZE;
405     }
406
407     return NGX_CONF_OK;
408 }
```

[One Level Up](#)

[Top Level](#)

# src/http/nginx\_http\_config.h - nginx-1.7.10

## Data types defined

- [ngx\\_http\\_conf\\_ctx\\_t](#)
- [ngx\\_http\\_module\\_t](#)

## Macros defined

- [NGX\\_HTTP\\_LIF\\_CONF](#)
- [NGX\\_HTTP\\_LMT\\_CONF](#)
- [NGX\\_HTTP\\_LOC\\_CONF](#)
- [NGX\\_HTTP\\_LOC\\_CONF\\_OFFSET](#)
- [NGX\\_HTTP\\_MAIN\\_CONF](#)
- [NGX\\_HTTP\\_MAIN\\_CONF\\_OFFSET](#)
- [NGX\\_HTTP\\_MODULE](#)
- [NGX\\_HTTP\\_SIF\\_CONF](#)
- [NGX\\_HTTP\\_SRV\\_CONF](#)
- [NGX\\_HTTP\\_SRV\\_CONF\\_OFFSET](#)
- [NGX\\_HTTP\\_UPS\\_CONF](#)
- [\\_NGX\\_HTTP\\_CONFIG\\_H\\_INCLUDED](#)
- [ngx\\_http\\_conf\\_get\\_module\\_loc\\_conf](#)
- [ngx\\_http\\_conf\\_get\\_module\\_main\\_conf](#)
- [ngx\\_http\\_conf\\_get\\_module\\_srv\\_conf](#)
- [ngx\\_http\\_cycle\\_get\\_module\\_main\\_conf](#)
- [ngx\\_http\\_get\\_module\\_loc\\_conf](#)
- [ngx\\_http\\_get\\_module\\_main\\_conf](#)
- [ngx\\_http\\_get\\_module\\_srv\\_conf](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_HTTP\_CONFIG\_H\_INCLUDED
9 #define \_NGX\_HTTP\_CONFIG\_H\_INCLUDED
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14 #include <ngx_http.h>
```

```

15
16
17 typedef struct {
18     void    **main_conf;
19     void    **srv_conf;
20     void    **loc_conf;
21 } ngx_http_conf_ctx_t;
22
23
24 typedef struct {
25     ngx_int_t  (*preconfiguration)(ngx_conf_t *cf);
26     ngx_int_t  (*postconfiguration)(ngx_conf_t *cf);
27
28     void    (*create_main_conf)(ngx_conf_t *cf);
29     char    (*init_main_conf)(ngx_conf_t *cf, void *conf);
30
31     void    (*create_srv_conf)(ngx_conf_t *cf);
32     char    (*merge_srv_conf)(ngx_conf_t *cf, void *prev, void *conf);
33
34     void    (*create_loc_conf)(ngx_conf_t *cf);
35     char    (*merge_loc_conf)(ngx_conf_t *cf, void *prev, void *conf);
36 } ngx_http_module_t;
37
38
39 #define  NGX_HTTP_MODULE          0x50545448    /* "HTTP" */
40
41 #define  NGX_HTTP_MAIN_CONF      0x02000000
42 #define  NGX_HTTP_SRV_CONF      0x04000000
43 #define  NGX_HTTP_LOC_CONF      0x08000000
44 #define  NGX_HTTP_UPS_CONF      0x10000000
45 #define  NGX_HTTP_SIF_CONF      0x20000000
46 #define  NGX_HTTP_LIF_CONF      0x40000000
47 #define  NGX_HTTP_LMT_CONF      0x80000000
48
49
50 #define  NGX_HTTP_MAIN_CONF_OFFSET  offsetof(ngx_http_conf_ctx_t, main_conf)
51 #define  NGX_HTTP_SRV_CONF_OFFSET  offsetof(ngx_http_conf_ctx_t, srv_conf)
52 #define  NGX_HTTP_LOC_CONF_OFFSET  offsetof(ngx_http_conf_ctx_t, loc_conf)
53
54
55 #define  ngx_http_get_module_main_conf(r, module)          \
56     (r)->main_conf[module.ctx_index]
57 #define  ngx_http_get_module_srv_conf(r, module)          \
58     (r)->srv_conf[module.ctx_index]
59 #define  ngx_http_get_module_loc_conf(r, module)          \
60     (r)->loc_conf[module.ctx_index]
61
62 #define  ngx_http_conf_get_module_main_conf(cf, module)   \
63     ((ngx_http_conf_ctx_t *) cf->ctx)->main_conf[module.ctx_index]
64 #define  ngx_http_conf_get_module_srv_conf(cf, module)   \
65     ((ngx_http_conf_ctx_t *) cf->ctx)->srv_conf[module.ctx_index]
66 #define  ngx_http_conf_get_module_loc_conf(cf, module)   \
67     ((ngx_http_conf_ctx_t *) cf->ctx)->loc_conf[module.ctx_index]
68
69 #define  ngx_http_cycle_get_module_main_conf(cycle, module) \
70     (cycle->conf_ctx[ngx_http_module.index] ?              \
71     ((ngx_http_conf_ctx_t *) cycle->conf_ctx[ngx_http_module.index]) \
72     ->main_conf[module.ctx_index]:                          \
73     NULL)
74
75 #endif /* NGX_HTTP_CONFIG_H_INCLUDED */

```

[One Level Up](#)

[Top Level](#)

# src/http/nginx\_http\_spdy\_module.h - nginx-1.7.10

## Data types defined

- [ngx\\_http\\_spdy\\_loc\\_conf\\_t](#)
- [ngx\\_http\\_spdy\\_main\\_conf\\_t](#)
- [ngx\\_http\\_spdy\\_srv\\_conf\\_t](#)

## Macros defined

- [\\_NGX\\_HTTP\\_SPDY\\_MODULE\\_H\\_INCLUDED](#)

## Source code

```
1
2 /*
3  * Copyright (C) Nginx, Inc.
4  * Copyright (C) Valentin V. Bartenev
5  */
6
7
8 #ifndef _NGX_HTTP_SPDY_MODULE_H_INCLUDED
9 #define _NGX_HTTP_SPDY_MODULE_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14 #include <ngx_http.h>
15
16
17 typedef struct {
18     size_t                recv_buffer_size;
19     u_char                *recv_buffer;
20 } ngx_http_spdy_main_conf_t;
21
22
23 typedef struct {
24     size_t                pool_size;
25     ngx_uint_t            concurrent_streams;
26     ngx_uint_t            streams_index_mask;
27     ngx_msec_t            recv_timeout;
28     ngx_msec_t            keepalive_timeout;
29     ngx_int_t             headers_comp;
30 } ngx_http_spdy_srv_conf_t;
31
32
33 typedef struct {
34     size_t                chunk_size;
35 } ngx_http_spdy_loc_conf_t;
36
37
38 extern ngx_module_t  ngx_http_spdy_module;
39
40
41 #endif /* _NGX_HTTP_SPDY_MODULE_H_INCLUDED */
```

## src/http/nginx\_http\_spdy.h - nginx-1.7.10

### Data types defined

- [ngx\\_http\\_spdy\\_connection\\_s](#)
- [ngx\\_http\\_spdy\\_connection\\_t](#)
- [ngx\\_http\\_spdy\\_handler\\_pt](#)
- [ngx\\_http\\_spdy\\_out\\_frame\\_s](#)
- [ngx\\_http\\_spdy\\_out\\_frame\\_t](#)
- [ngx\\_http\\_spdy\\_stream\\_s](#)

### Functions defined

- [ngx\\_http\\_spdy\\_queue\\_blocked\\_frame](#)
- [ngx\\_http\\_spdy\\_queue\\_frame](#)

### Macros defined

- [NGX\\_SPDY\\_CTL\\_BIT](#)
- [NGX\\_SPDY\\_DATA\\_DISCARD](#)
- [NGX\\_SPDY\\_DATA\\_ERROR](#)
- [NGX\\_SPDY\\_DATA\\_INTERNAL\\_ERROR](#)
- [NGX\\_SPDY\\_DELTA\\_SIZE](#)
- [NGX\\_SPDY\\_FLAG\\_CLEAR\\_SETTINGS](#)
- [NGX\\_SPDY\\_FLAG\\_FIN](#)
- [NGX\\_SPDY\\_FLAG\\_UNIDIRECTIONAL](#)
- [NGX\\_SPDY\\_FRAME\\_HEADER\\_SIZE](#)
- [NGX\\_SPDY\\_GOAWAY](#)
- [NGX\\_SPDY\\_GOAWAY\\_SIZE](#)
- [NGX\\_SPDY\\_HEADERS](#)
- [NGX\\_SPDY\\_HIGHEST\\_PRIORITY](#)
- [NGX\\_SPDY\\_LOWEST\\_PRIORITY](#)
- [NGX\\_SPDY\\_MAX\\_FRAME\\_SIZE](#)
- [NGX\\_SPDY\\_NPN\\_ADVERTISE](#)
- [NGX\\_SPDY\\_NPN\\_NEGOTIATED](#)
- [NGX\\_SPDY\\_NV\\_NLEN\\_SIZE](#)
- [NGX\\_SPDY\\_NV\\_NUM\\_SIZE](#)

- [NGX\\_SPDY\\_NV\\_VLEN\\_SIZE](#)
- [NGX\\_SPDY\\_PING](#)
- [NGX\\_SPDY\\_PING\\_SIZE](#)
- [NGX\\_SPDY\\_RST\\_STREAM](#)
- [NGX\\_SPDY\\_RST\\_STREAM\\_SIZE](#)
- [NGX\\_SPDY\\_SETTINGS](#)
- [NGX\\_SPDY\\_SETTINGS\\_FID\\_SIZE](#)
- [NGX\\_SPDY\\_SETTINGS\\_NUM\\_SIZE](#)
- [NGX\\_SPDY\\_SETTINGS\\_PAIR\\_SIZE](#)
- [NGX\\_SPDY\\_SETTINGS\\_VAL\\_SIZE](#)
- [NGX\\_SPDY\\_SID\\_SIZE](#)
- [NGX\\_SPDY\\_STATE\\_BUFFER\\_SIZE](#)
- [NGX\\_SPDY\\_SYN\\_REPLY](#)
- [NGX\\_SPDY\\_SYN\\_REPLY\\_SIZE](#)
- [NGX\\_SPDY\\_SYN\\_STREAM](#)
- [NGX\\_SPDY\\_SYN\\_STREAM\\_SIZE](#)
- [NGX\\_SPDY\\_VERSION](#)
- [NGX\\_SPDY\\_WINDOW\\_UPDATE](#)
- [NGX\\_SPDY\\_WINDOW\\_UPDATE\\_SIZE](#)
- [\\_NGX\\_HTTP\\_SPDY\\_H\\_INCLUDED](#)
- [ngx\\_spdy\\_ctl\\_frame\\_head](#)
- [ngx\\_spdy\\_frame\\_aligned\\_write\\_uint16](#)
- [ngx\\_spdy\\_frame\\_aligned\\_write\\_uint32](#)
- [ngx\\_spdy\\_frame\\_write\\_flags\\_and\\_id](#)
- [ngx\\_spdy\\_frame\\_write\\_flags\\_and\\_len](#)
- [ngx\\_spdy\\_frame\\_write\\_head](#)
- [ngx\\_spdy\\_frame\\_write\\_sid](#)
- [ngx\\_spdy\\_frame\\_write\\_uint16](#)
- [ngx\\_spdy\\_frame\\_write\\_uint16](#)
- [ngx\\_spdy\\_frame\\_write\\_uint32](#)
- [ngx\\_spdy\\_frame\\_write\\_uint32](#)
- [ngx\\_spdy\\_frame\\_write\\_window](#)

```

1  /*
2  * Copyright (C) Nginx, Inc.
3  * Copyright (C) Valentin V. Bartenev
4  */
5
6
7  #ifndef _NGX_HTTP_SPDY_H_INCLUDED_
8  #define _NGX_HTTP_SPDY_H_INCLUDED_
9
10
11 #include <ngx_config.h>
12 #include <ngx_core.h>
13 #include <ngx_http.h>
14
15 #include <zlib.h>
16
17
18 #define NGX_SPDY_VERSION          3
19
20 #define NGX_SPDY_NPN_ADVERTISE   "\x08spdy/3.1"
21 #define NGX_SPDY_NPN_NEGOTIATED "spdy/3.1"
22
23 #define NGX_SPDY_STATE_BUFFER_SIZE 16
24
25 #define NGX_SPDY_CTL_BIT         1
26
27 #define NGX_SPDY_SYN_STREAM      1
28 #define NGX_SPDY_SYN_REPLY      2
29 #define NGX_SPDY_RST_STREAM     3
30 #define NGX_SPDY_SETTINGS      4
31 #define NGX_SPDY_PING           6
32 #define NGX_SPDY_GOAWAY        7
33 #define NGX_SPDY_HEADERS        8
34 #define NGX_SPDY_WINDOW_UPDATE  9
35
36 #define NGX_SPDY_FRAME_HEADER_SIZE 8
37
38 #define NGX_SPDY_SID_SIZE        4
39 #define NGX_SPDY_DELTA_SIZE      4
40
41 #define NGX_SPDY_SYN_STREAM_SIZE 10
42 #define NGX_SPDY_SYN_REPLY_SIZE  4
43 #define NGX_SPDY_RST_STREAM_SIZE  8
44 #define NGX_SPDY_PING_SIZE        4
45 #define NGX_SPDY_GOAWAY_SIZE      8
46 #define NGX_SPDY_WINDOW_UPDATE_SIZE 8
47 #define NGX_SPDY_NV_NUM_SIZE      4
48 #define NGX_SPDY_NV_NLEN_SIZE     4
49 #define NGX_SPDY_NV_VLEN_SIZE     4
50 #define NGX_SPDY_SETTINGS_NUM_SIZE 4
51 #define NGX_SPDY_SETTINGS_FID_SIZE 4
52 #define NGX_SPDY_SETTINGS_VAL_SIZE 4
53
54 #define NGX_SPDY_SETTINGS_PAIR_SIZE \
55     (NGX_SPDY_SETTINGS_FID_SIZE + NGX_SPDY_SETTINGS_VAL_SIZE)
56
57 #define NGX_SPDY_HIGHEST_PRIORITY 0
58 #define NGX_SPDY_LOWEST_PRIORITY  7
59
60 #define NGX_SPDY_FLAG_FIN          0x01
61 #define NGX_SPDY_FLAG_UNIDIRECTIONAL 0x02
62 #define NGX_SPDY_FLAG_CLEAR_SETTINGS 0x01
63
64 #define NGX_SPDY_MAX_FRAME_SIZE    ((1 << 24) - 1)
65
66 #define NGX_SPDY_DATA_DISCARD      1
67 #define NGX_SPDY_DATA_ERROR        2
68 #define NGX_SPDY_DATA_INTERNAL_ERROR 3
69
70
71 typedef struct ngx_http_spdy_connection_s    ngx_http_spdy_connection_t;
72 typedef struct ngx_http_spdy_out_frame_s    ngx_http_spdy_out_frame_t;
73
74
75 typedef u_char *(*ngx_http_spdy_handler_pt) (ngx_http_spdy_connection_t *sc,
76     u_char *pos, u_char *end);

```

```

77
78 struct ngx_http_spdy_connection_s {
79     ngx_connection_t      *connection;
80     ngx_http_connection_t *http_connection;
81
82     ngx_uint_t            processing;
83
84     size_t                send_window;
85     size_t                recv_window;
86     size_t                init_window;
87
88     ngx_queue_t           waiting;
89
90     u_char                buffer[NGX_SPDY_STATE_BUFFER_SIZE];
91     size_t                buffer_used;
92     ngx_http_spdy_handler_pt handler;
93
94     z_stream              zstream_in;
95     z_stream              zstream_out;
96
97     ngx_pool_t            *pool;
98
99     ngx_http_spdy_out_frame_t *free_ctl_frames;
100    ngx_connection_t      *free_fake_connections;
101
102    ngx_http_spdy_stream_t **streams_index;
103
104    ngx_http_spdy_out_frame_t *last_out;
105
106    ngx_queue_t            posted;
107
108    ngx_http_spdy_stream_t *stream;
109
110    ngx_uint_t             entries;
111    size_t                 length;
112    u_char                 flags;
113
114    ngx_uint_t             last_sid;
115
116    unsigned               blocked:1;
117    unsigned               incomplete:1;
118 };
119
120
121 struct ngx_http_spdy_stream_s {
122     ngx_uint_t            id;
123     ngx_http_request_t    *request;
124     ngx_http_spdy_connection_t *connection;
125     ngx_http_spdy_stream_t *index;
126
127     ngx_uint_t            header_buffers;
128     ngx_uint_t            queued;
129
130     /*
131      * A change to SETTINGS_INITIAL_WINDOW_SIZE could cause the
132      * send_window to become negative, hence it's signed.
133      */
134     ssize_t               send_window;
135     size_t                recv_window;
136
137     ngx_http_spdy_out_frame_t *free_frames;
138     ngx_chain_t           *free_data_headers;
139     ngx_chain_t           *free_bufs;
140
141     ngx_queue_t           queue;
142
143     unsigned               priority:3;
144     unsigned               handled:1;
145     unsigned               blocked:1;
146     unsigned               exhausted:1;
147     unsigned               in_closed:1;
148     unsigned               out_closed:1;
149     unsigned               skip_data:2;
150 };
151
152

```



```

153 struct ngx_http_spdy_out_frame_s {
154     ngx_http_spdy_out_frame_t    *next;
155     ngx_chain_t                   *first;
156     ngx_chain_t                   *last;
157     ngx_int_t                      (*handler)(ngx_http_spdy_connection_t *sc,
158                                               ngx_http_spdy_out_frame_t *frame);
159
160     ngx_http_spdy_stream_t         *stream;
161     size_t                          length;
162
163     ngx_uint_t                     priority;
164     unsigned                        blocked:1;
165     unsigned                        fin:1;
166 };
167
168
169 static ngx_inline void
170 ngx_http_spdy_queue_frame(ngx_http_spdy_connection_t *sc,
171                          ngx_http_spdy_out_frame_t *frame)
172 {
173     ngx_http_spdy_out_frame_t **out;
174
175     for (out = &sc->last_out; *out; out = &(*out)->next)
176     {
177         /*
178          * NB: higher values represent lower priorities.
179          */
180         if (frame->priority >= (*out)->priority) {
181             break;
182         }
183     }
184
185     frame->next = *out;
186     *out = frame;
187 }
188
189
190 static ngx_inline void
191 ngx_http_spdy_queue_blocked_frame(ngx_http_spdy_connection_t *sc,
192                                  ngx_http_spdy_out_frame_t *frame)
193 {
194     ngx_http_spdy_out_frame_t **out;
195
196     for (out = &sc->last_out; *out; out = &(*out)->next)
197     {
198         if ((*out)->blocked) {
199             break;
200         }
201     }
202
203     frame->next = *out;
204     *out = frame;
205 }
206
207
208 void ngx_http_spdy_init(ngx_event_t *rev);
209 void ngx_http_spdy_request_headers_init(void);
210
211 ngx_int_t ngx_http_spdy_read_request_body(ngx_http_request_t *r,
212     ngx_http_client_body_handler_pt post_handler);
213
214 void ngx_http_spdy_close_stream(ngx_http_spdy_stream_t *stream, ngx_int_t rc);
215
216 ngx_int_t ngx_http_spdy_send_output_queue(ngx_http_spdy_connection_t *sc);
217
218
219 #define ngx_spdy_frame_aligned_write_uint16(p, s) \
220     (*(uint16_t *) (p) = htons((uint16_t) (s)), (p) + sizeof(uint16_t))
221
222 #define ngx_spdy_frame_aligned_write_uint32(p, s) \
223     (*(uint32_t *) (p) = htonl((uint32_t) (s)), (p) + sizeof(uint32_t))
224
225 #if (NGX_HAVE_NONALIGNED)
226
227 #define ngx_spdy_frame_write_uint16 ngx_spdy_frame_aligned_write_uint16
228 #define ngx_spdy_frame_write_uint32 ngx_spdy_frame_aligned_write_uint32

```

```

229
230 #else
231
232 #define ngx_spdy_frame_write_uint16(p, s) \
233     ((p)[0] = (u_char) ((s) >> 8), \
234     (p)[1] = (u_char) (s), \
235     (p) + sizeof(uint16_t))
236
237 #define ngx_spdy_frame_write_uint32(p, s) \
238     ((p)[0] = (u_char) ((s) >> 24), \
239     (p)[1] = (u_char) ((s) >> 16), \
240     (p)[2] = (u_char) ((s) >> 8), \
241     (p)[3] = (u_char) (s), \
242     (p) + sizeof(uint32_t))
243
244 #endif
245
246
247 #define ngx_spdy_ctl_frame_head(t) \
248     ((uint32_t) NGX_SPDY_CTL_BIT << 31 | NGX_SPDY_VERSION << 16 | (t))
249
250 #define ngx_spdy_frame_write_head(p, t) \
251     ngx_spdy_frame_aligned_write_uint32(p, ngx_spdy_ctl_frame_head(t))
252
253 #define ngx_spdy_frame_write_flags_and_len(p, f, l) \
254     ngx_spdy_frame_aligned_write_uint32(p, (f) << 24 | (l))
255 #define ngx_spdy_frame_write_flags_and_id(p, f, i) \
256     ngx_spdy_frame_aligned_write_uint32(p, (f) << 24 | (i))
257
258 #define ngx_spdy_frame_write_sid ngx_spdy_frame_aligned_write_uint32
259 #define ngx_spdy_frame_write_window ngx_spdy_frame_aligned_write_uint32
260
261 #endif /* NGX_HTTP_SPDY_H_INCLUDED */

```

[One Level Up](#)

[Top Level](#)

## src/http/nginx\_http\_request.c - nginx-1.7.10

### Global variables defined

- [ngx\\_http\\_client\\_errors](#)
- [ngx\\_http\\_headers\\_in](#)

### Functions defined

- [ngx\\_http\\_alloc\\_large\\_header\\_buffer](#)
- [ngx\\_http\\_block\\_reading](#)
- [ngx\\_http\\_close\\_connection](#)
- [ngx\\_http\\_close\\_request](#)
- [ngx\\_http\\_create\\_request](#)
- [ngx\\_http\\_empty\\_handler](#)
- [ngx\\_http\\_finalize\\_connection](#)
- [ngx\\_http\\_finalize\\_request](#)
- [ngx\\_http\\_find\\_virtual\\_server](#)
- [ngx\\_http\\_free\\_request](#)
- [ngx\\_http\\_init\\_connection](#)
- [ngx\\_http\\_keepalive\\_handler](#)
- [ngx\\_http\\_lingering\\_close\\_handler](#)
- [ngx\\_http\\_log\\_error](#)
- [ngx\\_http\\_log\\_error\\_handler](#)
- [ngx\\_http\\_log\\_request](#)
- [ngx\\_http\\_post\\_action](#)
- [ngx\\_http\\_post\\_request](#)
- [ngx\\_http\\_process\\_connection](#)
- [ngx\\_http\\_process\\_header\\_line](#)
- [ngx\\_http\\_process\\_host](#)
- [ngx\\_http\\_process\\_multi\\_header\\_lines](#)
- [ngx\\_http\\_process\\_request](#)
- [ngx\\_http\\_process\\_request\\_header](#)
- [ngx\\_http\\_process\\_request\\_headers](#)
- [ngx\\_http\\_process\\_request\\_line](#)

- [ngx http process request uri](#)
- [ngx http process unique header line](#)
- [ngx http process user agent](#)
- [ngx http read request header](#)
- [ngx http request empty handler](#)
- [ngx http request finalizer](#)
- [ngx http request handler](#)
- [ngx http run posted requests](#)
- [ngx http send special](#)
- [ngx http set keepalive](#)
- [ngx http set lingering close](#)
- [ngx http set virtual server](#)
- [ngx http set write handler](#)
- [ngx http ssl handshake](#)
- [ngx http ssl handshake handler](#)
- [ngx http ssl servername](#)
- [ngx http terminate handler](#)
- [ngx http terminate request](#)
- [ngx http test reading](#)
- [ngx http validate host](#)
- [ngx http wait request handler](#)
- [ngx http writer](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 static void ngx_http_wait_request_handler(ngx_event_t *ev);
14 static void ngx_http_process_request_line(ngx_event_t *rev);
15 static void ngx_http_process_request_headers(ngx_event_t *rev);
16 static ssize_t ngx_http_read_request_header(ngx_http_request_t *r);
17 static ngx_int_t ngx_http_alloc_large_header_buffer(ngx_http_request_t *r,
18     ngx_uint_t request_line);
19
20 static ngx_int_t ngx_http_process_header_line(ngx_http_request_t *r,
21     ngx_table_elt_t *h, ngx_uint_t offset);
22 static ngx_int_t ngx_http_process_unique_header_line(ngx_http_request_t *r,

```

```

23     ngx_table_elt_t *h, ngx_uint_t offset);
24 static ngx_int_t ngx_http_process_multi_header_lines(ngx_http_request_t *r,
25     ngx_table_elt_t *h, ngx_uint_t offset);
26 static ngx_int_t ngx_http_process_host(ngx_http_request_t *r,
27     ngx_table_elt_t *h, ngx_uint_t offset);
28 static ngx_int_t ngx_http_process_connection(ngx_http_request_t *r,
29     ngx_table_elt_t *h, ngx_uint_t offset);
30 static ngx_int_t ngx_http_process_user_agent(ngx_http_request_t *r,
31     ngx_table_elt_t *h, ngx_uint_t offset);
32
33 static ngx_int_t ngx_http_validate_host(ngx_str_t *host, ngx_pool_t *pool,
34     ngx_uint_t alloc);
35 static ngx_int_t ngx_http_set_virtual_server(ngx_http_request_t *r,
36     ngx_str_t *host);
37 static ngx_int_t ngx_http_find_virtual_server(ngx_connection_t *c,
38     ngx_http_virtual_names_t *virtual_names, ngx_str_t *host,
39     ngx_http_request_t *r, ngx_http_core_srv_conf_t **cscfp);
40
41 static void ngx_http_request_handler(ngx_event_t *ev);
42 static void ngx_http_terminate_request(ngx_http_request_t *r, ngx_int_t rc);
43 static void ngx_http_terminate_handler(ngx_http_request_t *r);
44 static void ngx_http_finalize_connection(ngx_http_request_t *r);
45 static ngx_int_t ngx_http_set_write_handler(ngx_http_request_t *r);
46 static void ngx_http_writer(ngx_http_request_t *r);
47 static void ngx_http_request_finalizer(ngx_http_request_t *r);
48
49 static void ngx_http_set_keepalive(ngx_http_request_t *r);
50 static void ngx_http_keepalive_handler(ngx_event_t *ev);
51 static void ngx_http_set_lingering_close(ngx_http_request_t *r);
52 static void ngx_http_lingering_close_handler(ngx_event_t *ev);
53 static ngx_int_t ngx_http_post_action(ngx_http_request_t *r);
54 static void ngx_http_close_request(ngx_http_request_t *r, ngx_int_t error);
55 static void ngx_http_log_request(ngx_http_request_t *r);
56
57 static u_char *ngx_http_log_error(ngx_log_t *log, u_char *buf, size_t len);
58 static u_char *ngx_http_log_error_handler(ngx_http_request_t *r,
59     ngx_http_request_t *sr, u_char *buf, size_t len);
60
61 #if (NGX_HTTP_SSL)
62 static void ngx_http_ssl_handshake(ngx_event_t *rev);
63 static void ngx_http_ssl_handshake_handler(ngx_connection_t *c);
64 #endif
65
66
67 static char *ngx_http_client_errors[] = {
68
69     /* NGX_HTTP_PARSE_INVALID_METHOD */
70     "client sent invalid method",
71
72     /* NGX_HTTP_PARSE_INVALID_REQUEST */
73     "client sent invalid request",
74
75     /* NGX_HTTP_PARSE_INVALID_09_METHOD */
76     "client sent invalid method in HTTP/0.9 request"
77 };
78
79
80 ngx_http_header_t ngx_http_headers_in[] = {
81     { ngx_string("Host"), offsetof(ngx_http_headers_in_t, host),
82       ngx_http_process_host },
83
84     { ngx_string("Connection"), offsetof(ngx_http_headers_in_t, connection),
85       ngx_http_process_connection },
86
87     { ngx_string("If-Modified-Since"),
88       offsetof(ngx_http_headers_in_t, if_modified_since),
89       ngx_http_process_unique_header_line },
90
91     { ngx_string("If-Unmodified-Since"),
92       offsetof(ngx_http_headers_in_t, if_unmodified_since),
93       ngx_http_process_unique_header_line },
94
95     { ngx_string("If-Match"),
96       offsetof(ngx_http_headers_in_t, if_match),
97       ngx_http_process_unique_header_line },
98

```

```

99     { ngx_string("If-None-Match"),
100         offsetof(ngx_http_headers_in_t, if_none_match),
101         ngx_http_process_unique_header_line },
102
103     { ngx_string("User-Agent"), offsetof(ngx_http_headers_in_t, user_agent),
104         ngx_http_process_user_agent },
105
106     { ngx_string("Referer"), offsetof(ngx_http_headers_in_t, referer),
107         ngx_http_process_header_line },
108
109     { ngx_string("Content-Length"),
110         offsetof(ngx_http_headers_in_t, content_length),
111         ngx_http_process_unique_header_line },
112
113     { ngx_string("Content-Type"),
114         offsetof(ngx_http_headers_in_t, content_type),
115         ngx_http_process_header_line },
116
117     { ngx_string("Range"), offsetof(ngx_http_headers_in_t, range),
118         ngx_http_process_header_line },
119
120     { ngx_string("If-Range"),
121         offsetof(ngx_http_headers_in_t, if_range),
122         ngx_http_process_unique_header_line },
123
124     { ngx_string("Transfer-Encoding"),
125         offsetof(ngx_http_headers_in_t, transfer_encoding),
126         ngx_http_process_header_line },
127
128     { ngx_string("Expect"),
129         offsetof(ngx_http_headers_in_t, expect),
130         ngx_http_process_unique_header_line },
131
132     { ngx_string("Upgrade"),
133         offsetof(ngx_http_headers_in_t, upgrade),
134         ngx_http_process_header_line },
135
136     #if (NGX_HTTP_GZIP)
137     { ngx_string("Accept-Encoding"),
138         offsetof(ngx_http_headers_in_t, accept_encoding),
139         ngx_http_process_header_line },
140
141     { ngx_string("Via"), offsetof(ngx_http_headers_in_t, via),
142         ngx_http_process_header_line },
143     #endif
144
145     { ngx_string("Authorization"),
146         offsetof(ngx_http_headers_in_t, authorization),
147         ngx_http_process_unique_header_line },
148
149     { ngx_string("Keep-Alive"), offsetof(ngx_http_headers_in_t, keep_alive),
150         ngx_http_process_header_line },
151
152     #if (NGX_HTTP_X_FORWARDED_FOR)
153     { ngx_string("X-Forwarded-For"),
154         offsetof(ngx_http_headers_in_t, x_forwarded_for),
155         ngx_http_process_multi_header_lines },
156     #endif
157
158     #if (NGX_HTTP_REALIP)
159     { ngx_string("X-Real-IP"),
160         offsetof(ngx_http_headers_in_t, x_real_ip),
161         ngx_http_process_header_line },
162     #endif
163
164     #if (NGX_HTTP_HEADERS)
165     { ngx_string("Accept"), offsetof(ngx_http_headers_in_t, accept),
166         ngx_http_process_header_line },
167
168     { ngx_string("Accept-Language"),
169         offsetof(ngx_http_headers_in_t, accept_language),
170         ngx_http_process_header_line },
171     #endif
172
173     #if (NGX_HTTP_DAV)
174     { ngx_string("Depth"), offsetof(ngx_http_headers_in_t, depth),

```

```

175         ngx_http_process_header_line },
176
177     { ngx_string("Destination"), offsetof(ngx_http_headers_in_t, destination),
178       ngx_http_process_header_line },
179
180     { ngx_string("Overwrite"), offsetof(ngx_http_headers_in_t, overwrite),
181       ngx_http_process_header_line },
182
183     { ngx_string("Date"), offsetof(ngx_http_headers_in_t, date),
184       ngx_http_process_header_line },
185 #endif
186
187     { ngx_string("Cookie"), offsetof(ngx_http_headers_in_t, cookies),
188       ngx_http_process_multi_header_lines },
189
190     { ngx_null_string, 0, NULL }
191 };
192
193
194 void
195 ngx_http_init_connection(ngx_connection_t *c)
196 {
197     ngx_uint_t          i;
198     ngx_event_t        *rev;
199     struct sockaddr_in  *sin;
200     ngx_http_port_t    *port;
201     ngx_http_in_addr_t  *addr;
202     ngx_http_log_ctx_t *ctx;
203     ngx_http_connection_t *hc;
204 #if (NGX_HAVE_INET6)
205     struct sockaddr_in6 *sin6;
206     ngx_http_in6_addr_t *addr6;
207 #endif
208
209     hc = ngx_palloc(c->pool, sizeof(ngx_http_connection_t));
210     if (hc == NULL) {
211         ngx_http_close_connection(c);
212         return;
213     }
214
215     c->data = hc;
216
217     /* find the server configuration for the address:port */
218
219     port = c->listening->servers;
220
221     if (port->naddrs > 1) {
222
223         /*
224          * there are several addresses on this port and one of them
225          * is an "*:port" wildcard so getsockname() in ngx_http_server_addr()
226          * is required to determine a server address
227          */
228
229         if (ngx_connection_local_sockaddr(c, NULL, 0) != NGX_OK) {
230             ngx_http_close_connection(c);
231             return;
232         }
233
234         switch (c->local_sockaddr->sa_family) {
235
236 #if (NGX_HAVE_INET6)
237             case AF_INET6:
238                 sin6 = (struct sockaddr_in6 *) c->local_sockaddr;
239
240                 addr6 = port->addrs;
241
242                 /* the last address is "*" */
243
244                 for (i = 0; i < port->naddrs - 1; i++) {
245                     if (ngx_memcmp(&addr6[i].addr6, &sin6->sin6_addr, 16) == 0) {
246                         break;
247                     }
248                 }
249
250                 hc->addr_conf = &addr6[i].conf;

```

```

251         break;
252     #endif
253
254     default: /* AF_INET */
255         sin = (struct sockaddr_in *) c->local_sockaddr;
256
257         addr = port->addrs;
258
259         /* the last address is "" */
260
261         for (i = 0; i < port->naddrs - 1; i++) {
262             if (addr[i].addr == sin->sin_addr.s_addr) {
263                 break;
264             }
265         }
266
267         hc->addr_conf = &addr[i].conf;
268
269         break;
270     }
271 } else {
272
273     switch (c->local_sockaddr->sa_family) {
274
275     #if (NGX_HAVE_INET6)
276     case AF_INET6:
277         addr6 = port->addrs;
278         hc->addr_conf = &addr6[0].conf;
279         break;
280     #endif
281
282     default: /* AF_INET */
283         addr = port->addrs;
284         hc->addr_conf = &addr[0].conf;
285         break;
286     }
287 }
288
289 /* the default server configuration for the address:port */
290 hc->conf_ctx = hc->addr_conf->default_server->ctx;
291
292 ctx = ngx_palloc(c->pool, sizeof(ngx_http_log_ctx_t));
293 if (ctx == NULL) {
294     ngx_http_close_connection(c);
295     return;
296 }
297
298 ctx->connection = c;
299 ctx->request = NULL;
300 ctx->current_request = NULL;
301
302 c->log->connection = c->number;
303 c->log->handler = ngx_http_log_error;
304 c->log->data = ctx;
305 c->log->action = "waiting for request";
306
307 c->log_error = NGX_ERROR_INFO;
308
309 rev = c->read;
310 rev->handler = ngx_http_wait_request_handler;
311 c->write->handler = ngx_http_empty_handler;
312
313 #if (NGX_HTTP_SPDY)
314 if (hc->addr_conf->spdy) {
315     rev->handler = ngx_http_spdy_init;
316 }
317 #endif
318
319 #if (NGX_HTTP_SSL)
320 {
321     ngx_http_ssl_srv_conf_t *sscf;
322
323     sscf = ngx_http_get_module_srv_conf(hc->conf_ctx, ngx_http_ssl_module);
324 }
325
326

```



```

327 if (sscf->enable || hc->addr_conf->ssl) {
328
329     c->log->action = "SSL handshaking";
330
331     if (hc->addr_conf->ssl && sscf->ssl.ctx == NULL) {
332         ngx_log_error(NGX_LOG_ERR, c->log, 0,
333             "no \"ssl_certificate\" is defined "
334             "in server listening on SSL port");
335         ngx_http_close_connection(c);
336         return;
337     }
338
339     hc->ssl = 1;
340
341     rev->handler = ngx_http_ssl_handshake;
342 }
343 }
344 #endif
345
346 if (hc->addr_conf->proxy_protocol) {
347     hc->proxy_protocol = 1;
348     c->log->action = "reading PROXY protocol";
349 }
350
351 if (rev->ready) {
352     /* the deferred accept(), rtsig, aio, iocp */
353
354     if (ngx_use_accept_mutex) {
355         ngx_post_event(rev, &ngx_posted_events);
356         return;
357     }
358
359     rev->handler(rev);
360     return;
361 }
362
363 ngx_add_timer(rev, c->listening->post_accept_timeout);
364 ngx_reusable_connection(c, 1);
365
366 if (ngx_handle_read_event(rev, 0) != NGX_OK) {
367     ngx_http_close_connection(c);
368     return;
369 }
370 }
371
372
373 static void
374 ngx_http_wait_request_handler(ngx_event_t *rev)
375 {
376     u_char          *p;
377     size_t          size;
378     ssize_t         n;
379     ngx_buf_t       *b;
380     ngx_connection_t *c;
381     ngx_http_connection_t *hc;
382     ngx_http_core_srv_conf_t *cscf;
383
384     c = rev->data;
385
386     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, c->log, 0, "http wait request handler");
387
388     if (rev->timedout) {
389         ngx_log_error(NGX_LOG_INFO, c->log, NGX_ETIMEDOUT, "client timed out");
390         ngx_http_close_connection(c);
391         return;
392     }
393
394     if (c->close) {
395         ngx_http_close_connection(c);
396         return;
397     }
398
399     hc = c->data;
400     cscf = ngx_http_get_module_srv_conf(hc->conf_ctx, ngx_http_core_module);
401
402     size = cscf->client_header_buffer_size;

```

```

403     b = c->buffer;
404
405     if (b == NULL) {
406         b = ngx_create_temp_buf(c->pool, size);
407         if (b == NULL) {
408             ngx_http_close_connection(c);
409             return;
410         }
411
412         c->buffer = b;
413
414     } else if (b->start == NULL) {
415
416         b->start = ngx_palloc(c->pool, size);
417         if (b->start == NULL) {
418             ngx_http_close_connection(c);
419             return;
420         }
421
422         b->pos = b->start;
423         b->last = b->start;
424         b->end = b->last + size;
425     }
426
427     n = c->recv(c, b->last, size);
428
429     if (n == NGX_AGAIN) {
430
431         if (!rev->timer_set) {
432             ngx_add_timer(rev, c->listening->post_accept_timeout);
433             ngx_reusable_connection(c, 1);
434         }
435
436         if (ngx_handle_read_event(rev, 0) != NGX_OK) {
437             ngx_http_close_connection(c);
438             return;
439         }
440
441         /*
442          * We are trying to not hold c->buffer's memory for an idle connection.
443          */
444
445         if (ngx_pfree(c->pool, b->start) == NGX_OK) {
446             b->start = NULL;
447         }
448
449         return;
450     }
451
452     if (n == NGX_ERROR) {
453         ngx_http_close_connection(c);
454         return;
455     }
456
457     if (n == 0) {
458         ngx_log_error(NGX_LOG_INFO, c->log, 0,
459             "client closed connection");
460         ngx_http_close_connection(c);
461         return;
462     }
463
464     b->last += n;
465
466     if (hc->proxy_protocol) {
467         hc->proxy_protocol = 0;
468
469         p = ngx_proxy_protocol_parse(c, b->pos, b->last);
470
471         if (p == NULL) {
472             ngx_http_close_connection(c);
473             return;
474         }
475
476         b->pos = p;
477
478

```

```

479     if (b->pos == b->last) {
480         c->log->action = "waiting for request";
481         b->pos = b->start;
482         b->last = b->start;
483         ngx\_post\_event(rev, &ngx\_posted\_events);
484         return;
485     }
486 }
487
488 c->log->action = "reading client request line";
489
490 ngx\_reusable\_connection(c, 0);
491
492 c->data = ngx\_http\_create\_request(c);
493 if (c->data == NULL) {
494     ngx\_http\_close\_connection(c);
495     return;
496 }
497
498 rev->handler = ngx\_http\_process\_request\_line;
499 ngx\_http\_process\_request\_line(rev);
500 }
501
502
503 ngx\_http\_request\_t *
504 ngx\_http\_create\_request(ngx\_connection\_t *c)
505 {
506     ngx\_pool\_t          *pool;
507     ngx\_time\_t         *tp;
508     ngx\_http\_request\_t *r;
509     ngx\_http\_log\_ctx\_t *ctx;
510     ngx\_http\_connection\_t *hc;
511     ngx\_http\_core\_srv\_conf\_t *cscf;
512     ngx\_http\_core\_loc\_conf\_t *clcf;
513     ngx\_http\_core\_main\_conf\_t *cmcf;
514
515     c->requests++;
516
517     hc = c->data;
518
519     cscf = ngx\_http\_get\_module\_srv\_conf(hc->conf_ctx, ngx\_http\_core\_module);
520
521     pool = ngx\_create\_pool(cscf->request_pool_size, c->log);
522     if (pool == NULL) {
523         return NULL;
524     }
525
526     r = ngx\_palloc(pool, sizeof(ngx\_http\_request\_t));
527     if (r == NULL) {
528         ngx\_destroy\_pool(pool);
529         return NULL;
530     }
531
532     r->pool = pool;
533
534     r->http_connection = hc;
535     r->signature = NGX\_HTTP\_MODULE;
536     r->connection = c;
537
538     r->main_conf = hc->conf_ctx->main_conf;
539     r->srv_conf = hc->conf_ctx->srv_conf;
540     r->loc_conf = hc->conf_ctx->loc_conf;
541
542     r->read_event_handler = ngx\_http\_block\_reading;
543
544     clcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_core\_module);
545
546     ngx\_http\_set\_connection\_log(r->connection, clcf->error_log);
547
548     r->header_in = hc->nbusy ? hc->busy[0] : c->buffer;
549
550     if (ngx\_list\_init(&r->headers_out.headers, r->pool, 20,
551         sizeof(ngx\_table\_elt\_t))
552         != NGX\_OK)
553     {
554         ngx\_destroy\_pool(r->pool);

```

```

555     return NULL;
556 }
557
558 r->ctx = ngx_palloc(r->pool, sizeof(void *) * ngx_http_max_module);
559 if (r->ctx == NULL) {
560     ngx_destroy_pool(r->pool);
561     return NULL;
562 }
563
564 cmcf = ngx_http_get_module_main_conf(r, ngx_http_core_module);
565
566 r->variables = ngx_palloc(r->pool, cmcf->variables.nelts
567     * sizeof(ngx_http_variable_value_t));
568 if (r->variables == NULL) {
569     ngx_destroy_pool(r->pool);
570     return NULL;
571 }
572
573 #if (NGX_HTTP_SSL)
574     if (c->ssl) {
575         r->main_filter_need_in_memory = 1;
576     }
577 #endif
578
579     r->main = r;
580     r->count = 1;
581
582     tp = ngx_timeofday();
583     r->start_sec = tp->sec;
584     r->start_msec = tp->msec;
585
586     r->method = NGX_HTTP_UNKNOWN;
587     r->http_version = NGX_HTTP_VERSION_10;
588
589     r->headers_in.content_length_n = -1;
590     r->headers_in.keep_alive_n = -1;
591     r->headers_out.content_length_n = -1;
592     r->headers_out.last_modified_time = -1;
593
594     r->uri_changes = NGX_HTTP_MAX_URI_CHANGES + 1;
595     r->subrequests = NGX_HTTP_MAX_SUBREQUESTS + 1;
596
597     r->http_state = NGX_HTTP_READING_REQUEST_STATE;
598
599     ctx = c->log->data;
600     ctx->request = r;
601     ctx->current_request = r;
602     r->log_handler = ngx_http_log_error_handler;
603
604 #if (NGX_STAT_STUB)
605     (void) ngx_atomic_fetch_add(ngx_stat_reading, 1);
606     r->stat_reading = 1;
607     (void) ngx_atomic_fetch_add(ngx_stat_requests, 1);
608 #endif
609
610     return r;
611 }
612
613
614 #if (NGX_HTTP_SSL)
615
616 static void
617 ngx_http_ssl_handshake(ngx_event_t *rev)
618 {
619     u_char                *p, buf[NGX_PROXY_PROTOCOL_MAX_HEADER + 1];
620     size_t                size;
621     ssize_t               n;
622     ngx_err_t             err;
623     ngx_int_t             rc;
624     ngx_connection_t      *c;
625     ngx_http_connection_t *hc;
626     ngx_http_ssl_srv_conf_t *sscf;
627
628     c = rev->data;
629     hc = c->data;
630

```

```

631 ngx_log_debug0(NGX_LOG_DEBUG_HTTP, rev->log, 0,
632     "http check ssl handshake");
633
634 if (rev->timedout) {
635     ngx_log_error(NGX_LOG_INFO, c->log, NGX_ETIMEDOUT, "client timed out");
636     ngx_http_close_connection(c);
637     return;
638 }
639
640 if (c->close) {
641     ngx_http_close_connection(c);
642     return;
643 }
644
645 size = hc->proxy_protocol ? sizeof(buf) : 1;
646
647 n = recv(c->fd, (char *) buf, size, MSG_PEEK);
648
649 err = ngx_socket_errno;
650
651 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, rev->log, 0, "http recv(): %d", n);
652
653 if (n == -1) {
654     if (err == NGX_EAGAIN) {
655
656         if (!rev->timer_set) {
657             ngx_add_timer(rev, c->listening->post_accept_timeout);
658             ngx_reusable_connection(c, 1);
659         }
660
661         if (ngx_handle_read_event(rev, 0) != NGX_OK) {
662             ngx_http_close_connection(c);
663         }
664
665         return;
666     }
667
668     ngx_connection_error(c, err, "recv() failed");
669     ngx_http_close_connection(c);
670
671     return;
672 }
673
674 if (hc->proxy_protocol) {
675     hc->proxy_protocol = 0;
676
677     p = ngx_proxy_protocol_parse(c, buf, buf + n);
678
679     if (p == NULL) {
680         ngx_http_close_connection(c);
681         return;
682     }
683
684     size = p - buf;
685
686     if (c->recv(c, buf, size) != (ssize_t) size) {
687         ngx_http_close_connection(c);
688         return;
689     }
690
691     c->log->action = "SSL handshaking";
692
693     if (n == (ssize_t) size) {
694         ngx_post_event(rev, &ngx_posted_events);
695         return;
696     }
697
698     n = 1;
699     buf[0] = *p;
700 }
701
702 if (n == 1) {
703     if (buf[0] & 0x80 /* SSLv2 */ || buf[0] == 0x16 /* SSLv3/TLSv1 */) {
704         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, rev->log, 0,
705             "https ssl handshake: 0x%02Xd", buf[0]);
706     }

```

```

707     sscf = ngx_http_get_module_srv_conf(hc->conf_ctx,
708                                         ngx_http_ssl_module);
709
710     if (ngx_ssl_create_connection(&sscf->ssl, c, NGX_SSL_BUFFER)
711         != NGX_OK)
712     {
713         ngx_http_close_connection(c);
714         return;
715     }
716
717     rc = ngx_ssl_handshake(c);
718
719     if (rc == NGX_AGAIN) {
720
721         if (!rev->timer_set) {
722             ngx_add_timer(rev, c->listening->post_accept_timeout);
723         }
724
725         ngx_reusable_connection(c, 0);
726
727         c->ssl->handler = ngx_http_ssl_handshake_handler;
728         return;
729     }
730
731     ngx_http_ssl_handshake_handler(c);
732
733     return;
734 }
735
736 ngx_log_debug0(NGX_LOG_DEBUG_HTTP, rev->log, 0, "plain http");
737
738 c->log->action = "waiting for request";
739
740 rev->handler = ngx_http_wait_request_handler;
741 ngx_http_wait_request_handler(rev);
742
743 return;
744 }
745
746 ngx_log_error(NGX_LOG_INFO, c->log, 0, "client closed connection");
747 ngx_http_close_connection(c);
748 }
749
750
751 static void
752 ngx_http_ssl_handshake_handler(ngx_connection_t *c)
753 {
754     if (c->ssl->handshaked) {
755
756         /*
757          * The majority of browsers do not send the "close notify" alert.
758          * Among them are MSIE, old Mozilla, Netscape 4, Konqueror,
759          * and Links. And what is more, MSIE ignores the server's alert.
760          *
761          * Opera and recent Mozilla send the alert.
762          */
763
764         c->ssl->no_wait_shutdown = 1;
765
766         #if (NGX_HTTP_SPDY
767             && (defined TLSEXT_TYPE_application_layer_protocol_negotiation
768                 || defined TLSEXT_TYPE_next_proto_neg))
769             {
770                 unsigned int    len;
771                 const unsigned char *data;
772                 static const ngx_str_t spdy = ngx_string(NGX_SPDY_NPN_NEGOTIATED);
773
774                 #ifndef TLSEXT_TYPE_application_layer_protocol_negotiation
775                     SSL_get0_alpn_selected(c->ssl->connection, &data, &len);
776
777                 #endif
778                 #ifndef TLSEXT_TYPE_next_proto_neg
779                     if (len == 0) {
780                         SSL_get0_next_proto_negotiated(c->ssl->connection, &data, &len);
781                     }
782                 #endif
783             }
784         #endif

```

```

783 #else /* TLSEXT_TYPE_next_proto_neg */
784     SSL_get0_next_proto_negotiated(c->ssl->connection, &data, &len);
785 #endif
786
787     if (len == spdy.len && ngx_strncmp(data, spdy.data, spdy.len) == 0) {
788         ngx_http_spdy_init(c->read);
789         return;
790     }
791 }
792 #endif
793
794     c->log->action = "waiting for request";
795
796     c->read->handler = ngx_http_wait_request_handler;
797     /* STUB: epoll edge */ c->write->handler = ngx_http_empty_handler;
798
799     ngx_reusable_connection(c, 1);
800
801     ngx_http_wait_request_handler(c->read);
802
803     return;
804 }
805
806 if (c->read->timedout) {
807     ngx_log_error(NGX_LOG_INFO, c->log, NGX_ETIMEDOUT, "client timed out");
808 }
809
810 ngx_http_close_connection(c);
811 }
812
813 #ifdef SSL_CTRL_SET_TLSEXT_HOSTNAME
814 int
815 ngx_http_ssl_servername(ngx_ssl_conn_t *ssl_conn, int *ad, void *arg)
816 {
817     ngx_str_t          host;
818     const char         *servername;
819     ngx_connection_t   *c;
820     ngx_http_connection_t *hc;
821     ngx_http_ssl_srv_conf_t *sscf;
822     ngx_http_core_loc_conf_t *clcf;
823     ngx_http_core_srv_conf_t *cscf;
824
825     servername = SSL_get_servername(ssl_conn, TLSEXT_NAMETYPE_host_name);
826
827     if (servername == NULL) {
828         return SSL_TLSEXT_ERR_NOACK;
829     }
830 }
831
832 c = ngx_ssl_get_connection(ssl_conn);
833
834 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, c->log, 0,
835     "SSL server name: \"%s\"", servername);
836
837 host.len = ngx_strlen(servername);
838
839 if (host.len == 0) {
840     return SSL_TLSEXT_ERR_NOACK;
841 }
842
843 host.data = (u_char *) servername;
844
845 if (ngx_http_validate_host(&host, c->pool, 1) != NGX_OK) {
846     return SSL_TLSEXT_ERR_NOACK;
847 }
848
849 hc = c->data;
850
851 if (ngx_http_find_virtual_server(c, hc->addr_conf->virtual_names, &host,
852     NULL, &cscf)
853     != NGX_OK)
854 {
855     return SSL_TLSEXT_ERR_NOACK;
856 }
857
858 hc->ssl_servername = ngx_palloc(c->pool, sizeof(ngx_str_t));

```

```

859     if (hc->ssl_servername == NULL) {
860         return SSL_TLSEXT_ERR_NOACK;
861     }
862
863     *hc->ssl_servername = host;
864
865     hc->conf_ctx = cscf->ctx;
866
867     clcf = ngx\_http\_get\_module\_loc\_conf(hc->conf_ctx, ngx\_http\_core\_module);
868
869     ngx\_http\_set\_connection\_log(c, clcf->error_log);
870
871     sscf = ngx\_http\_get\_module\_srv\_conf(hc->conf_ctx, ngx\_http\_ssl\_module);
872
873     if (sscf->ssl.ctx) {
874         SSL_set_SSL_CTX(ssl_conn, sscf->ssl.ctx);
875
876         /*
877          * SSL_set_SSL_CTX() only changes certs as of 1.0.0d
878          * adjust other things we care about
879          */
880
881         SSL_set_verify(ssl_conn, SSL_CTX_get_verify_mode(sscf->ssl.ctx),
882                       SSL_CTX_get_verify_callback(sscf->ssl.ctx));
883
884         SSL_set_verify_depth(ssl_conn, SSL_CTX_get_verify_depth(sscf->ssl.ctx));
885
886     #ifdef SSL_CTRL_CLEAR_OPTIONS
887         /* only in 0.9.8m+ */
888         SSL_clear_options(ssl_conn, SSL_get_options(ssl_conn) &
889                          ~SSL_CTX_get_options(sscf->ssl.ctx));
890     #endif
891
892         SSL_set_options(ssl_conn, SSL_CTX_get_options(sscf->ssl.ctx));
893     }
894
895     return SSL_TLSEXT_ERR_OK;
896 }
897
898 #endif
899
900 #endif
901
902
903 static void
904 ngx\_http\_process\_request\_line(ngx\_event\_t *rev)
905 {
906     ssize_t          n;
907     ngx\_int\_t       rc, rv;
908     ngx\_str\_t       host;
909     ngx\_connection\_t *c;
910     ngx\_http\_request\_t *r;
911
912     c = rev->data;
913     r = c->data;
914
915     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, rev->log, 0,
916                  "http process request line");
917
918     if (rev->timedout) {
919         ngx\_log\_error(NGX\_LOG\_INFO, c->log, NGX\_ETIMEDOUT, "client timed out");
920         c->timedout = 1;
921         ngx\_http\_close\_request(r, NGX\_HTTP\_REQUEST\_TIME\_OUT);
922         return;
923     }
924
925     rc = NGX\_AGAIN;
926
927     for ( ;; ) {
928
929         if (rc == NGX\_AGAIN) {
930             n = ngx\_http\_read\_request\_header(r);
931
932             if (n == NGX\_AGAIN || n == NGX\_ERROR) {
933                 return;
934             }

```



```

935     }
936
937     rc = ngx_http_parse_request_line(r, r->header_in);
938
939     if (rc == NGX_OK) {
940         /* the request line has been parsed successfully */
941
942         r->request_line.len = r->request_end - r->request_start;
943         r->request_line.data = r->request_start;
944         r->request_length = r->header_in->pos - r->request_start;
945
946         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, c->log, 0,
947             "http request line: \"%V\"", &r->request_line);
948
949         r->method_name.len = r->method_end - r->request_start + 1;
950         r->method_name.data = r->request_line.data;
951
952         if (r->http_protocol.data) {
953             r->http_protocol.len = r->request_end - r->http_protocol.data;
954         }
955
956         if (ngx_http_process_request_uri(r) != NGX_OK) {
957             return;
958         }
959
960         if (r->host_start && r->host_end) {
961
962             host.len = r->host_end - r->host_start;
963             host.data = r->host_start;
964
965             rc = ngx_http_validate_host(&host, r->pool, 0);
966
967             if (rc == NGX_DECLINED) {
968                 ngx_log_error(NGX_LOG_INFO, c->log, 0,
969                     "client sent invalid host in request line");
970                 ngx_http_finalize_request(r, NGX_HTTP_BAD_REQUEST);
971                 return;
972             }
973
974             if (rc == NGX_ERROR) {
975                 ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
976                 return;
977             }
978
979             if (ngx_http_set_virtual_server(r, &host) == NGX_ERROR) {
980                 return;
981             }
982
983             r->headers_in.server = host;
984         }
985
986         if (r->http_version < NGX_HTTP_VERSION_10) {
987
988             if (r->headers_in.server.len == 0
989                 && ngx_http_set_virtual_server(r, &r->headers_in.server)
990                 == NGX_ERROR)
991             {
992                 return;
993             }
994
995             ngx_http_process_request(r);
996             return;
997         }
998
999
1000
1001         if (ngx_list_init(&r->headers_in.headers, r->pool, 20,
1002             sizeof(ngx_table_elt_t))
1003             != NGX_OK)
1004         {
1005             ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
1006             return;
1007         }
1008
1009         c->log->action = "reading client request headers";
1010

```

```

1011     rev->handler = ngx_http_process_request_headers;
1012     ngx_http_process_request_headers(rev);
1013
1014     return;
1015 }
1016
1017 if (rc != NGX_AGAIN) {
1018
1019     /* there was error while a request line parsing */
1020
1021     ngx_log_error(NGX_LOG_INFO, c->log, 0,
1022                 ngx_http_client_errors[rc - NGX_HTTP_CLIENT_ERROR]);
1023     ngx_http_finalize_request(r, NGX_HTTP_BAD_REQUEST);
1024     return;
1025 }
1026
1027 /* NGX_AGAIN: a request line parsing is still incomplete */
1028
1029 if (r->header_in->pos == r->header_in->end) {
1030
1031     rv = ngx_http_alloc_large_header_buffer(r, 1);
1032
1033     if (rv == NGX_ERROR) {
1034         ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
1035         return;
1036     }
1037
1038     if (rv == NGX_DECLINED) {
1039         r->request_line.len = r->header_in->end - r->request_start;
1040         r->request_line.data = r->request_start;
1041
1042         ngx_log_error(NGX_LOG_INFO, c->log, 0,
1043                     "client sent too long URI");
1044         ngx_http_finalize_request(r, NGX_HTTP_REQUEST_URI_TOO_LARGE);
1045         return;
1046     }
1047 }
1048 }
1049 }
1050
1051 ngx_int_t
1052 ngx_http_process_request_uri(ngx_http_request_t *r)
1053 {
1054     ngx_http_core_srv_conf_t *cscf;
1055
1056     if (r->args_start) {
1057         r->uri.len = r->args_start - 1 - r->uri_start;
1058     } else {
1059         r->uri.len = r->uri_end - r->uri_start;
1060     }
1061
1062     if (r->complex_uri || r->quoted_uri) {
1063
1064         r->uri.data = ngx_pnalloc(r->pool, r->uri.len + 1);
1065         if (r->uri.data == NULL) {
1066             ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
1067             return NGX_ERROR;
1068         }
1069     }
1070
1071     cscf = ngx_http_get_module_srv_conf(r, ngx_http_core_module);
1072
1073     if (ngx_http_parse_complex_uri(r, cscf->merge_slashes) != NGX_OK) {
1074         r->uri.len = 0;
1075
1076         ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
1077                     "client sent invalid request");
1078         ngx_http_finalize_request(r, NGX_HTTP_BAD_REQUEST);
1079         return NGX_ERROR;
1080     }
1081
1082 } else {
1083     r->uri.data = r->uri_start;
1084 }
1085
1086 r->unparsed_uri.len = r->uri_end - r->uri_start;

```

```

1087 r->unparsed_uri.data = r->uri_start;
1088
1089 r->valid_unparsed_uri = r->space_in_uri ? 0 : 1;
1090
1091 if (r->uri_ext) {
1092     if (r->args_start) {
1093         r->exten.len = r->args_start - 1 - r->uri_ext;
1094     } else {
1095         r->exten.len = r->uri_end - r->uri_ext;
1096     }
1097
1098     r->exten.data = r->uri_ext;
1099 }
1100
1101 if (r->args_start && r->uri_end > r->args_start) {
1102     r->args.len = r->uri_end - r->args_start;
1103     r->args.data = r->args_start;
1104 }
1105
1106 #if (NGX_WIN32)
1107 {
1108     u_char *p, *last;
1109
1110     p = r->uri.data;
1111     last = r->uri.data + r->uri.len;
1112
1113     while (p < last) {
1114
1115         if (*p++ == ':') {
1116
1117             /*
1118              * this check covers "::$data", "::$index_allocation" and
1119              * "::$i30:$index_allocation"
1120              */
1121
1122             if (p < last && *p == '$') {
1123                 ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
1124                     "client sent unsafe win32 URI");
1125                 ngx_http_finalize_request(r, NGX_HTTP_BAD_REQUEST);
1126                 return NGX_ERROR;
1127             }
1128         }
1129     }
1130
1131     p = r->uri.data + r->uri.len - 1;
1132
1133     while (p > r->uri.data) {
1134
1135         if (*p == ' ') {
1136             p--;
1137             continue;
1138         }
1139
1140         if (*p == '.') {
1141             p--;
1142             continue;
1143         }
1144
1145         break;
1146     }
1147
1148     if (p != r->uri.data + r->uri.len - 1) {
1149         r->uri.len = p + 1 - r->uri.data;
1150         ngx_http_set_exten(r);
1151     }
1152 }
1153 }
1154 #endif
1155
1156 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1157     "http uri: \"%V\"", &r->uri);
1158
1159 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1160     "http args: \"%V\"", &r->args);
1161
1162 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,

```

```

1163         "http exten: \"%V\"", &r->exten);
1164
1165     return NGX_OK;
1166 }
1167
1168
1169 static void
1170 ngx_http_process_request_headers(ngx_event_t *rev)
1171 {
1172     u_char          *p;
1173     size_t          len;
1174     ssize_t         n;
1175     ngx_int_t       rc, rv;
1176     ngx_table_elt_t *h;
1177     ngx_connection_t *c;
1178     ngx_http_header_t *hh;
1179     ngx_http_request_t *r;
1180     ngx_http_core_srv_conf_t *cscf;
1181     ngx_http_core_main_conf_t *cmcf;
1182
1183     c = rev->data;
1184     r = c->data;
1185
1186     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, rev->log, 0,
1187                  "http process request header line");
1188
1189     if (rev->timedout) {
1190         ngx_log_error(NGX_LOG_INFO, c->log, NGX_ETIMEDOUT, "client timed out");
1191         c->timedout = 1;
1192         ngx_http_close_request(r, NGX_HTTP_REQUEST_TIME_OUT);
1193         return;
1194     }
1195
1196     cmcf = ngx_http_get_module_main_conf(r, ngx_http_core_module);
1197
1198     rc = NGX_AGAIN;
1199
1200     for ( ;; ) {
1201
1202         if (rc == NGX_AGAIN) {
1203
1204             if (r->header_in->pos == r->header_in->end) {
1205
1206                 rv = ngx_http_alloc_large_header_buffer(r, 0);
1207
1208                 if (rv == NGX_ERROR) {
1209                     ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
1210                     return;
1211                 }
1212
1213                 if (rv == NGX_DECLINED) {
1214                     p = r->header_name_start;
1215
1216                     r->lingering_close = 1;
1217
1218                     if (p == NULL) {
1219                         ngx_log_error(NGX_LOG_INFO, c->log, 0,
1220                                      "client sent too large request");
1221                         ngx_http_finalize_request(r,
1222                                                  NGX_HTTP_REQUEST_HEADER_TOO_LARGE);
1223                         return;
1224                     }
1225
1226                     len = r->header_in->end - p;
1227
1228                     if (len > NGX_MAX_ERROR_STR - 300) {
1229                         len = NGX_MAX_ERROR_STR - 300;
1230                     }
1231
1232                     ngx_log_error(NGX_LOG_INFO, c->log, 0,
1233                                  "client sent too long header line: \"%*s...\"",
1234                                  len, r->header_name_start);
1235
1236                     ngx_http_finalize_request(r,
1237                                              NGX_HTTP_REQUEST_HEADER_TOO_LARGE);
1238                     return;

```

```

1239     }
1240 }
1241
1242 n = ngx_http_read_request_header(r);
1243
1244 if (n == NGX_AGAIN || n == NGX_ERROR) {
1245     return;
1246 }
1247 }
1248
1249 /* the host header could change the server configuration context */
1250 cscf = ngx_http_get_module_srv_conf(r, ngx_http_core_module);
1251
1252 rc = ngx_http_parse_header_line(r, r->header_in,
1253                                cscf->underscores_in_headers);
1254
1255 if (rc == NGX_OK) {
1256
1257     r->request_length += r->header_in->pos - r->header_name_start;
1258
1259     if (r->invalid_header && cscf->ignore_invalid_headers) {
1260
1261         /* there was error while a header line parsing */
1262
1263         ngx_log_error(NGX_LOG_INFO, c->log, 0,
1264                     "client sent invalid header line: \"%s\"",
1265                     r->header_end - r->header_name_start,
1266                     r->header_name_start);
1267
1268         continue;
1269     }
1270
1271     /* a header line has been parsed successfully */
1272
1273     h = ngx_list_push(&r->headers_in.headers);
1274     if (h == NULL) {
1275         ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
1276         return;
1277     }
1278
1279     h->hash = r->header_hash;
1280
1281     h->key.len = r->header_name_end - r->header_name_start;
1282     h->key.data = r->header_name_start;
1283     h->key.data[h->key.len] = '\0';
1284
1285     h->value.len = r->header_end - r->header_start;
1286     h->value.data = r->header_start;
1287     h->value.data[h->value.len] = '\0';
1288
1289     h->lowercase_key = ngx_pnalloc(r->pool, h->key.len);
1290     if (h->lowercase_key == NULL) {
1291         ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
1292         return;
1293     }
1294
1295     if (h->key.len == r->lowercase_index) {
1296         ngx_memcpy(h->lowercase_key, r->lowercase_header, h->key.len);
1297     } else {
1298         ngx_strlow(h->lowercase_key, h->key.data, h->key.len);
1299     }
1300
1301     hh = ngx_hash_find(&cscf->headers_in_hash, h->hash,
1302                      h->lowercase_key, h->key.len);
1303
1304     if (hh && hh->handler(r, h, hh->offset) != NGX_OK) {
1305         return;
1306     }
1307
1308     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1309                  "http header: \"%V: %V\"",
1310                  &h->key, &h->value);
1311
1312     continue;
1313 }
1314

```

```

1315     if (rc == NGX\_HTTP\_PARSE\_HEADER\_DONE) {
1316
1317         /* a whole header has been parsed successfully */
1318
1319         ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
1320             "http header done");
1321
1322         r->request_length += r->header_in->pos - r->header_name_start;
1323
1324         r->http_state = NGX\_HTTP\_PROCESS\_REQUEST\_STATE;
1325
1326         rc = ngx\_http\_process\_request\_header(r);
1327
1328         if (rc != NGX\_OK) {
1329             return;
1330         }
1331
1332         ngx\_http\_process\_request(r);
1333
1334         return;
1335     }
1336
1337     if (rc == NGX\_AGAIN) {
1338
1339         /* a header line parsing is still not complete */
1340
1341         continue;
1342     }
1343
1344     /* rc == NGX\_HTTP\_PARSE\_INVALID\_HEADER: "\r" is not followed by "\n" */
1345
1346     ngx\_log\_error(NGX\_LOG\_INFO, c->log, 0,
1347         "client sent invalid header line: \"%s\r...\"",
1348         r->header_end - r->header_name_start,
1349         r->header_name_start);
1350     ngx\_http\_finalize\_request(r, NGX\_HTTP\_BAD\_REQUEST);
1351     return;
1352 }
1353 }
1354
1355
1356 static ssize_t
1357 ngx\_http\_read\_request\_header(ngx\_http\_request\_t *r)
1358 {
1359     ssize_t            n;
1360     ngx\_event\_t      *rev;
1361     ngx\_connection\_t *c;
1362     ngx\_http\_core\_srv\_conf\_t *cscf;
1363
1364     c = r->connection;
1365     rev = c->read;
1366
1367     n = r->header_in->last - r->header_in->pos;
1368
1369     if (n > 0) {
1370         return n;
1371     }
1372
1373     if (rev->ready) {
1374         n = c->recv(c, r->header_in->last,
1375             r->header_in->end - r->header_in->last);
1376     } else {
1377         n = NGX\_AGAIN;
1378     }
1379
1380     if (n == NGX\_AGAIN) {
1381         if (!rev->timer_set) {
1382             cscf = ngx\_http\_get\_module\_srv\_conf(r, ngx\_http\_core\_module);
1383             ngx\_add\_timer(rev, cscf->client_header_timeout);
1384         }
1385
1386         if (ngx\_handle\_read\_event(rev, 0) != NGX\_OK) {
1387             ngx\_http\_close\_request(r, NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);
1388             return NGX\_ERROR;
1389         }
1390     }

```

```

1391     return NGX\_AGAIN;
1392 }
1393
1394 if (n == 0) {
1395     ngx\_log\_error(NGX\_LOG\_INFO, c->log, 0,
1396                 "client prematurely closed connection");
1397 }
1398
1399 if (n == 0 || n == NGX\_ERROR) {
1400     c->error = 1;
1401     c->log->action = "reading client request headers";
1402
1403     ngx\_http\_finalize\_request(r, NGX\_HTTP\_BAD\_REQUEST);
1404     return NGX\_ERROR;
1405 }
1406
1407 r->header_in->last += n;
1408
1409 return n;
1410 }
1411
1412
1413 static ngx\_int\_t
1414 ngx\_http\_alloc\_large\_header\_buffer(ngx\_http\_request\_t *r,
1415 ngx\_uint\_t request_line)
1416 {
1417     u\_char                *old, *new;
1418     ngx\_buf\_t            *b;
1419     ngx\_http\_connection\_t *hc;
1420     ngx\_http\_core\_srv\_conf\_t *cscf;
1421
1422     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
1423                 "http alloc large header buffer");
1424
1425     if (request_line && r->state == 0) {
1426
1427         /* the client fills up the buffer with "\r\n" */
1428
1429         r->header_in->pos = r->header_in->start;
1430         r->header_in->last = r->header_in->start;
1431
1432         return NGX\_OK;
1433     }
1434
1435     old = request_line ? r->request_start : r->header_name_start;
1436
1437     cscf = ngx\_http\_get\_module\_srv\_conf(r, ngx\_http\_core\_module);
1438
1439     if (r->state != 0
1440         && (size\_t) (r->header_in->pos - old)
1441             >= cscf->large_client_header_buffers.size)
1442     {
1443         return NGX\_DECLINED;
1444     }
1445
1446     hc = r->http_connection;
1447
1448     if (hc->nfree) {
1449         b = hc->free[--hc->nfree];
1450
1451         ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
1452                     "http large header free: %p %uz",
1453                     b->pos, b->end - b->last);
1454     } else if (hc->nbusy < cscf->large_client_header_buffers.num) {
1455
1456         if (hc->busy == NULL) {
1457             hc->busy = ngx\_palloc(r->connection->pool,
1458                                 cscf->large_client_header_buffers.num * sizeof(ngx\_buf\_t *));
1459             if (hc->busy == NULL) {
1460                 return NGX\_ERROR;
1461             }
1462         }
1463     }
1464
1465     b = ngx\_create\_temp\_buf(r->connection->pool,
1466                             cscf->large_client_header_buffers.size);

```

```

1467     if (b == NULL) {
1468         return NGX_ERROR;
1469     }
1470
1471     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1472                 "http large header alloc: %p %uz",
1473                 b->pos, b->end - b->last);
1474
1475 } else {
1476     return NGX_DECLINED;
1477 }
1478
1479 hc->busy[hc->nbusy++] = b;
1480
1481 if (r->state == 0) {
1482     /*
1483      * r->state == 0 means that a header line was parsed successfully
1484      * and we do not need to copy incomplete header line and
1485      * to relocate the parser header pointers
1486      */
1487
1488     r->header_in = b;
1489
1490     return NGX_OK;
1491 }
1492
1493 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1494               "http large header copy: %d", r->header_in->pos - old);
1495
1496 new = b->start;
1497
1498 ngx_memcpy(new, old, r->header_in->pos - old);
1499
1500 b->pos = new + (r->header_in->pos - old);
1501 b->last = new + (r->header_in->pos - old);
1502
1503 if (request_line) {
1504     r->request_start = new;
1505
1506     if (r->request_end) {
1507         r->request_end = new + (r->request_end - old);
1508     }
1509
1510     r->method_end = new + (r->method_end - old);
1511
1512     r->uri_start = new + (r->uri_start - old);
1513     r->uri_end = new + (r->uri_end - old);
1514
1515     if (r->schema_start) {
1516         r->schema_start = new + (r->schema_start - old);
1517         r->schema_end = new + (r->schema_end - old);
1518     }
1519
1520     if (r->host_start) {
1521         r->host_start = new + (r->host_start - old);
1522         if (r->host_end) {
1523             r->host_end = new + (r->host_end - old);
1524         }
1525     }
1526
1527     if (r->port_start) {
1528         r->port_start = new + (r->port_start - old);
1529         r->port_end = new + (r->port_end - old);
1530     }
1531
1532     if (r->uri_ext) {
1533         r->uri_ext = new + (r->uri_ext - old);
1534     }
1535
1536     if (r->args_start) {
1537         r->args_start = new + (r->args_start - old);
1538     }
1539
1540     if (r->http_protocol.data) {
1541         r->http_protocol.data = new + (r->http_protocol.data - old);
1542     }

```



```

1543     } else {
1544     }
1545     r->header_name_start = new;
1546     r->header_name_end = new + (r->header_name_end - old);
1547     r->header_start = new + (r->header_start - old);
1548     r->header_end = new + (r->header_end - old);
1549 }
1550
1551 r->header_in = b;
1552
1553 return NGX_OK;
1554 }
1555
1556
1557 static ngx_int_t
1558 ngx_http_process_header_line(ngx_http_request_t *r, ngx_table_elt_t *h,
1559     ngx_uint_t offset)
1560 {
1561     ngx_table_elt_t **ph;
1562
1563     ph = (ngx_table_elt_t **) ((char *) &r->headers_in + offset);
1564
1565     if (*ph == NULL) {
1566         *ph = h;
1567     }
1568
1569     return NGX_OK;
1570 }
1571
1572
1573 static ngx_int_t
1574 ngx_http_process_unique_header_line(ngx_http_request_t *r, ngx_table_elt_t *h,
1575     ngx_uint_t offset)
1576 {
1577     ngx_table_elt_t **ph;
1578
1579     ph = (ngx_table_elt_t **) ((char *) &r->headers_in + offset);
1580
1581     if (*ph == NULL) {
1582         *ph = h;
1583         return NGX_OK;
1584     }
1585
1586     ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
1587         "client sent duplicate header line: \"%V: %V\", "
1588         "previous value: \"%V: %V\"",
1589         &h->key, &h->value, &(*ph)->key, &(*ph)->value);
1590
1591     ngx_http_finalize_request(r, NGX_HTTP_BAD_REQUEST);
1592
1593     return NGX_ERROR;
1594 }
1595
1596
1597 static ngx_int_t
1598 ngx_http_process_host(ngx_http_request_t *r, ngx_table_elt_t *h,
1599     ngx_uint_t offset)
1600 {
1601     ngx_int_t rc;
1602     ngx_str_t host;
1603
1604     if (r->headers_in.host == NULL) {
1605         r->headers_in.host = h;
1606     }
1607
1608     host = h->value;
1609
1610     rc = ngx_http_validate_host(&host, r->pool, 0);
1611
1612     if (rc == NGX_DECLINED) {
1613         ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
1614             "client sent invalid host header");
1615         ngx_http_finalize_request(r, NGX_HTTP_BAD_REQUEST);
1616         return NGX_ERROR;
1617     }
1618 }

```

```

1619     if (rc == NGX\_ERROR) {
1620         ngx\_http\_close\_request(r, NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);
1621         return NGX\_ERROR;
1622     }
1623
1624     if (r->headers_in.server.len) {
1625         return NGX\_OK;
1626     }
1627
1628     if (ngx\_http\_set\_virtual\_server(r, &host) == NGX\_ERROR) {
1629         return NGX\_ERROR;
1630     }
1631
1632     r->headers_in.server = host;
1633
1634     return NGX\_OK;
1635 }
1636
1637
1638 static ngx\_int\_t
1639 ngx\_http\_process\_connection(ngx\_http\_request\_t *r, ngx\_table\_elt\_t *h,
1640 ngx\_uint\_t offset)
1641 {
1642     if (ngx\_strcasestrn(h->value.data, "close", 5 - 1)) {
1643         r->headers_in.connection_type = NGX\_HTTP\_CONNECTION\_CLOSE;
1644     } else if (ngx\_strcasestrn(h->value.data, "keep-alive", 10 - 1)) {
1645         r->headers_in.connection_type = NGX\_HTTP\_CONNECTION\_KEEP\_ALIVE;
1646     }
1647
1648     return NGX\_OK;
1649 }
1650
1651
1652
1653 static ngx\_int\_t
1654 ngx\_http\_process\_user\_agent(ngx\_http\_request\_t *r, ngx\_table\_elt\_t *h,
1655 ngx\_uint\_t offset)
1656 {
1657     u_char *user_agent, *msie;
1658
1659     if (r->headers_in.user_agent) {
1660         return NGX\_OK;
1661     }
1662
1663     r->headers_in.user_agent = h;
1664
1665     /* check some widespread browsers while the header is in CPU cache */
1666
1667     user_agent = h->value.data;
1668
1669     msie = ngx\_strstrn(user_agent, "MSIE ", 5 - 1);
1670
1671     if (msie && msie + 7 < user_agent + h->value.len) {
1672
1673         r->headers_in.msie = 1;
1674
1675         if (msie[6] == '.') {
1676
1677             switch (msie[5]) {
1678                 case '4':
1679                 case '5':
1680                     r->headers_in.msie6 = 1;
1681                     break;
1682                 case '6':
1683                     if (ngx\_strstrn(msie + 8, "SV1", 3 - 1) == NULL) {
1684                         r->headers_in.msie6 = 1;
1685                     }
1686                     break;
1687             }
1688         }
1689     }
1690
1691     #if 0
1692     /* MSIE ignores the SSL "close notify" alert */
1693     if (c->ssl) {
1694         c->ssl->no_send_shutdown = 1;
1695     }

```

```

1695 #endif
1696 }
1697
1698 if (ngx_strstr(user_agent, "Opera", 5 - 1)) {
1699     r->headers_in.opera = 1;
1700     r->headers_in.msie = 0;
1701     r->headers_in.msie6 = 0;
1702 }
1703
1704 if (!r->headers_in.msie && !r->headers_in.opera) {
1705
1706     if (ngx_strstr(user_agent, "Gecko/", 6 - 1)) {
1707         r->headers_in.gecko = 1;
1708
1709     } else if (ngx_strstr(user_agent, "Chrome/", 7 - 1)) {
1710         r->headers_in.chrome = 1;
1711
1712     } else if (ngx_strstr(user_agent, "Safari/", 7 - 1)
1713         && ngx_strstr(user_agent, "Mac OS X", 8 - 1))
1714     {
1715         r->headers_in.safari = 1;
1716
1717     } else if (ngx_strstr(user_agent, "Konqueror", 9 - 1)) {
1718         r->headers_in.konqueror = 1;
1719     }
1720 }
1721
1722 return NGX_OK;
1723 }
1724
1725
1726 static ngx_int_t
1727 ngx_http_process_multi_header_lines(ngx_http_request_t *r, ngx_table_elt_t *h,
1728     ngx_uint_t offset)
1729 {
1730     ngx_array_t    *headers;
1731     ngx_table_elt_t **ph;
1732
1733     headers = (ngx_array_t *) ((char *) &r->headers_in + offset);
1734
1735     if (headers->elts == NULL) {
1736         if (ngx_array_init(headers, r->pool, 1, sizeof(ngx_table_elt_t *))
1737             != NGX_OK)
1738         {
1739             ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
1740             return NGX_ERROR;
1741         }
1742     }
1743
1744     ph = ngx_array_push(headers);
1745     if (ph == NULL) {
1746         ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
1747         return NGX_ERROR;
1748     }
1749
1750     *ph = h;
1751     return NGX_OK;
1752 }
1753
1754
1755 ngx_int_t
1756 ngx_http_process_request_header(ngx_http_request_t *r)
1757 {
1758     if (r->headers_in.server.len == 0
1759         && ngx_http_set_virtual_server(r, &r->headers_in.server)
1760             == NGX_ERROR)
1761     {
1762         return NGX_ERROR;
1763     }
1764
1765     if (r->headers_in.host == NULL && r->http_version > NGX_HTTP_VERSION_10) {
1766         ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
1767             "client sent HTTP/1.1 request without \"Host\" header");
1768         ngx_http_finalize_request(r, NGX_HTTP_BAD_REQUEST);
1769         return NGX_ERROR;
1770     }

```

```

1771
1772 if (r->headers_in.content_length) {
1773     r->headers_in.content_length_n =
1774         ngx_atoof(r->headers_in.content_length->value.data,
1775                 r->headers_in.content_length->value.len);
1776
1777     if (r->headers_in.content_length_n == NGX_ERROR) {
1778         ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
1779                     "client sent invalid \"Content-Length\" header");
1780         ngx_http_finalize_request(r, NGX_HTTP_BAD_REQUEST);
1781         return NGX_ERROR;
1782     }
1783 }
1784
1785 if (r->method & NGX_HTTP_TRACE) {
1786     ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
1787                 "client sent TRACE method");
1788     ngx_http_finalize_request(r, NGX_HTTP_NOT_ALLOWED);
1789     return NGX_ERROR;
1790 }
1791
1792 if (r->headers_in.transfer_encoding) {
1793     if (r->headers_in.transfer_encoding->value.len == 7
1794         && ngx_strncasecmp(r->headers_in.transfer_encoding->value.data,
1795                          (u_char *) "chunked", 7) == 0)
1796     {
1797         r->headers_in.content_length = NULL;
1798         r->headers_in.content_length_n = -1;
1799         r->headers_in.chunked = 1;
1800
1801     } else if (r->headers_in.transfer_encoding->value.len != 8
1802               || ngx_strncasecmp(r->headers_in.transfer_encoding->value.data,
1803                                  (u_char *) "identity", 8) != 0)
1804     {
1805         ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
1806                     "client sent unknown \"Transfer-Encoding\": \"%V\"",
1807                     &r->headers_in.transfer_encoding->value);
1808         ngx_http_finalize_request(r, NGX_HTTP_NOT_IMPLEMENTED);
1809         return NGX_ERROR;
1810     }
1811 }
1812
1813 if (r->headers_in.connection_type == NGX_HTTP_CONNECTION_KEEP_ALIVE) {
1814     if (r->headers_in.keep_alive) {
1815         r->headers_in.keep_alive_n =
1816             ngx_atotm(r->headers_in.keep_alive->value.data,
1817                     r->headers_in.keep_alive->value.len);
1818     }
1819 }
1820
1821 return NGX_OK;
1822 }
1823
1824
1825 void
1826 ngx_http_process_request(ngx_http_request_t *r)
1827 {
1828     ngx_connection_t *c;
1829
1830     c = r->connection;
1831
1832     #if (NGX_HTTP_SSL)
1833
1834     if (r->http_connection->ssl) {
1835         long rc;
1836         X509 *cert;
1837         ngx_http_ssl_srv_conf_t *sscf;
1838
1839         if (c->ssl == NULL) {
1840             ngx_log_error(NGX_LOG_INFO, c->log, 0,
1841                         "client sent plain HTTP request to HTTPS port");
1842             ngx_http_finalize_request(r, NGX_HTTP_TO_HTTPS);
1843             return;
1844         }
1845
1846         sscf = ngx_http_get_module_srv_conf(r, ngx_http_ssl_module);

```

```

1847
1848     if (sscf->verify) {
1849         rc = SSL_get_verify_result(c->ssl->connection);
1850
1851         if (rc != X509_V_OK
1852             && (sscf->verify != 3 || !ngx_ssl_verify_error_optional(rc)))
1853             {
1854                 ngx_log_error(NGX_LOG_INFO, c->log, 0,
1855                     "client SSL certificate verify error: (%l:%s)",
1856                     rc, X509_verify_cert_error_string(rc));
1857
1858                 ngx_ssl_remove_cached_session(sscf->ssl.ctx,
1859                     (SSL_get0_session(c->ssl->connection)));
1860
1861                 ngx_http_finalize_request(r, NGX_HTTPS_CERT_ERROR);
1862                 return;
1863             }
1864
1865         if (sscf->verify == 1) {
1866             cert = SSL_get_peer_certificate(c->ssl->connection);
1867
1868             if (cert == NULL) {
1869                 ngx_log_error(NGX_LOG_INFO, c->log, 0,
1870                     "client sent no required SSL certificate");
1871
1872                 ngx_ssl_remove_cached_session(sscf->ssl.ctx,
1873                     (SSL_get0_session(c->ssl->connection)));
1874
1875                 ngx_http_finalize_request(r, NGX_HTTPS_NO_CERT);
1876                 return;
1877             }
1878
1879             X509_free(cert);
1880         }
1881     }
1882 }
1883
1884 #endif
1885
1886     if (c->read->timer_set) {
1887         ngx_del_timer(c->read);
1888     }
1889
1890 #if (NGX_STAT_STUB)
1891     (void) ngx_atomic_fetch_add(ngx_stat_reading, -1);
1892     r->stat_reading = 0;
1893     (void) ngx_atomic_fetch_add(ngx_stat_writing, 1);
1894     r->stat_writing = 1;
1895 #endif
1896
1897     c->read->handler = ngx_http_request_handler;
1898     c->write->handler = ngx_http_request_handler;
1899     r->read_event_handler = ngx_http_block_reading;
1900
1901     ngx_http_handler(r);
1902
1903     ngx_http_run_posted_requests(c);
1904 }
1905
1906
1907 static ngx_int_t
1908 ngx_http_validate_host(ngx_str_t *host, ngx_pool_t *pool, ngx_uint_t alloc)
1909 {
1910     u_char *h, ch;
1911     size_t i, dot_pos, host_len;
1912
1913     enum {
1914         sw_usual = 0,
1915         sw_literal,
1916         sw_rest
1917     } state;
1918
1919     dot_pos = host->len;
1920     host_len = host->len;
1921
1922     h = host->data;

```

```

1923
1924     state = sw_usual;
1925
1926     for (i = 0; i < host->len; i++) {
1927         ch = h[i];
1928
1929         switch (ch) {
1930
1931             case '.':
1932                 if (dot_pos == i - 1) {
1933                     return NGX\_DECLINED;
1934                 }
1935                 dot_pos = i;
1936                 break;
1937
1938             case ':':
1939                 if (state == sw_usual) {
1940                     host_len = i;
1941                     state = sw_rest;
1942                 }
1943                 break;
1944
1945             case '[':
1946                 if (i == 0) {
1947                     state = sw_literal;
1948                 }
1949                 break;
1950
1951             case ']':
1952                 if (state == sw_literal) {
1953                     host_len = i + 1;
1954                     state = sw_rest;
1955                 }
1956                 break;
1957
1958             case '\\0':
1959                 return NGX\_DECLINED;
1960
1961             default:
1962
1963                 if (ngx\_path\_separator(ch)) {
1964                     return NGX\_DECLINED;
1965                 }
1966
1967                 if (ch >= 'A' && ch <= 'Z') {
1968                     alloc = 1;
1969                 }
1970
1971                 break;
1972         }
1973     }
1974
1975     if (dot_pos == host_len - 1) {
1976         host_len--;
1977     }
1978
1979     if (host_len == 0) {
1980         return NGX\_DECLINED;
1981     }
1982
1983     if (alloc) {
1984         host->data = ngx\_pnalloc(pool, host_len);
1985         if (host->data == NULL) {
1986             return NGX\_ERROR;
1987         }
1988
1989         ngx\_strlow(host->data, h, host_len);
1990     }
1991
1992     host->len = host_len;
1993
1994     return NGX\_OK;
1995 }
1996
1997
1998 static ngx\_int\_t

```

```

1999 ngx_http_set_virtual_server(ngx_http_request_t *r, ngx_str_t *host)
2000 {
2001     ngx_int_t          rc;
2002     ngx_http_connection_t *hc;
2003     ngx_http_core_loc_conf_t *clcf;
2004     ngx_http_core_srv_conf_t *cscf;
2005
2006     #if (NGX_SUPPRESS_WARN)
2007         cscf = NULL;
2008     #endif
2009
2010     hc = r->http_connection;
2011
2012     #if (NGX_HTTP_SSL && defined SSL_CTRL_SET_TLSEXT_HOSTNAME)
2013
2014     if (hc->ssl_servername) {
2015         if (hc->ssl_servername->len == host->len
2016             && ngx_strncmp(hc->ssl_servername->data,
2017                            host->data, host->len) == 0)
2018             {
2019             #if (NGX_PCRE)
2020                 if (hc->ssl_servername_regex
2021                     && ngx_http_regex_exec(r, hc->ssl_servername_regex,
2022                                             hc->ssl_servername) != NGX_OK)
2023                     {
2024                         ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
2025                         return NGX_ERROR;
2026                     }
2027             #endif
2028                 return NGX_OK;
2029             }
2030         }
2031
2032     #endif
2033
2034     rc = ngx_http_find_virtual_server(r->connection,
2035                                     hc->addr_conf->virtual_names,
2036                                     host, r, &cscf);
2037
2038     if (rc == NGX_ERROR) {
2039         ngx_http_close_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
2040         return NGX_ERROR;
2041     }
2042
2043     #if (NGX_HTTP_SSL && defined SSL_CTRL_SET_TLSEXT_HOSTNAME)
2044
2045     if (hc->ssl_servername) {
2046         ngx_http_ssl_srv_conf_t *sscf;
2047
2048         if (rc == NGX_DECLINED) {
2049             cscf = hc->addr_conf->default_server;
2050             rc = NGX_OK;
2051         }
2052
2053         sscf = ngx_http_get_module_srv_conf(cscf->ctx, ngx_http_ssl_module);
2054
2055         if (sscf->verify) {
2056             ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
2057                          "client attempted to request the server name "
2058                          "different from that one was negotiated");
2059             ngx_http_finalize_request(r, NGX_HTTP_BAD_REQUEST);
2060             return NGX_ERROR;
2061         }
2062     }
2063
2064     #endif
2065
2066     if (rc == NGX_DECLINED) {
2067         return NGX_OK;
2068     }
2069
2070     r->srv_conf = cscf->ctx->srv_conf;
2071     r->loc_conf = cscf->ctx->loc_conf;
2072
2073     clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
2074

```

```

2075     ngx_http_set_connection_log(r->connection, clcf->error_log);
2076
2077     return NGX_OK;
2078 }
2079
2080
2081 static ngx_int_t
2082 ngx_http_find_virtual_server(ngx_connection_t *c,
2083     ngx_http_virtual_names_t *virtual_names, ngx_str_t *host,
2084     ngx_http_request_t *r, ngx_http_core_srv_conf_t **cscfp)
2085 {
2086     ngx_http_core_srv_conf_t *cscf;
2087
2088     if (virtual_names == NULL) {
2089         return NGX_DECLINED;
2090     }
2091
2092     cscf = ngx_hash_find_combined(&virtual_names->names,
2093         ngx_hash_key(host->data, host->len),
2094         host->data, host->len);
2095
2096     if (cscf) {
2097         *cscfp = cscf;
2098         return NGX_OK;
2099     }
2100
2101     #if (NGX_PCRE)
2102
2103     if (host->len && virtual_names->nregex) {
2104         ngx_int_t         n;
2105         ngx_uint_t        i;
2106         ngx_http_server_name_t *sn;
2107
2108         sn = virtual_names->regex;
2109
2110         #if (NGX_HTTP_SSL && defined SSL_CTRL_SET_TLSEXT_HOSTNAME)
2111
2112         if (r == NULL) {
2113             ngx_http_connection_t *hc;
2114
2115             for (i = 0; i < virtual_names->nregex; i++) {
2116
2117                 n = ngx_regex_exec(sn[i].regex->regex, host, NULL, 0);
2118
2119                 if (n == NGX_REGEX_NO_MATCHED) {
2120                     continue;
2121                 }
2122
2123                 if (n >= 0) {
2124                     hc = c->data;
2125                     hc->ssl_servername_regex = sn[i].regex;
2126
2127                     *cscfp = sn[i].server;
2128                     return NGX_OK;
2129                 }
2130
2131                 ngx_log_error(NGX_LOG_ALERT, c->log, 0,
2132                     ngx_regex_exec_n " failed: %i "
2133                     "on \"%V\" using \"%V\"",
2134                     n, host, &sn[i].regex->name);
2135
2136                 return NGX_ERROR;
2137             }
2138
2139             return NGX_DECLINED;
2140         }
2141
2142         #endif /* NGX_HTTP_SSL && defined SSL_CTRL_SET_TLSEXT_HOSTNAME */
2143
2144         for (i = 0; i < virtual_names->nregex; i++) {
2145
2146             n = ngx_http_regex_exec(r, sn[i].regex, host);
2147
2148             if (n == NGX_DECLINED) {
2149                 continue;
2150             }

```



```

2151         if (n == NGX\_OK) {
2152             *cscfp = sn[i].server;
2153             return NGX\_OK;
2154         }
2155     }
2156
2157     return NGX\_ERROR;
2158 }
2159 }
2160
2161 #endif /* NGX\_PCRE */
2162
2163     return NGX\_DECLINED;
2164 }
2165
2166
2167 static void
2168 ngx_http_request_handler(ngx\_event\_t *ev)
2169 {
2170     ngx\_connection\_t *c;
2171     ngx\_http\_request\_t *r;
2172
2173     c = ev->data;
2174     r = c->data;
2175
2176     ngx\_http\_set\_log\_request(c->log, r);
2177
2178     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, c->log, 0,
2179                 "http run request: \"%V?%V\"", &r->uri, &r->args);
2180
2181     if (ev->write) {
2182         r->write_event_handler(r);
2183     } else {
2184         r->read_event_handler(r);
2185     }
2186 }
2187
2188 ngx\_http\_run\_posted\_requests(c);
2189 }
2190
2191
2192 void
2193 ngx_http_run_posted_requests(ngx\_connection\_t *c)
2194 {
2195     ngx\_http\_request\_t *r;
2196     ngx\_http\_posted\_request\_t *pr;
2197
2198     for ( ;; ) {
2199
2200         if (c->destroyed) {
2201             return;
2202         }
2203
2204         r = c->data;
2205         pr = r->main->posted_requests;
2206
2207         if (pr == NULL) {
2208             return;
2209         }
2210
2211         r->main->posted_requests = pr->next;
2212
2213         r = pr->request;
2214
2215         ngx\_http\_set\_log\_request(c->log, r);
2216
2217         ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, c->log, 0,
2218                 "http posted request: \"%V?%V\"", &r->uri, &r->args);
2219
2220         r->write_event_handler(r);
2221     }
2222 }
2223
2224
2225 ngx\_int\_t
2226 ngx_http_post_request(ngx\_http\_request\_t *r, ngx\_http\_posted\_request\_t *pr)

```

```

2227 {
2228     ngx\_http\_posted\_request\_t **p;
2229
2230     if (pr == NULL) {
2231         pr = ngx\_palloc(r->pool, sizeof(ngx\_http\_posted\_request\_t));
2232         if (pr == NULL) {
2233             return NGX\_ERROR;
2234         }
2235     }
2236
2237     pr->request = r;
2238     pr->next = NULL;
2239
2240     for (p = &r->main->posted_requests; *p; p = &(*p)->next) { /* void */ }
2241
2242     *p = pr;
2243
2244     return NGX\_OK;
2245 }
2246
2247 void
2248 ngx\_http\_finalize\_request(ngx\_http\_request\_t *r, ngx\_int\_t rc)
2249 {
2250     ngx\_connection\_t *c;
2251     ngx\_http\_request\_t *pr;
2252     ngx\_http\_core\_loc\_conf\_t *clcf;
2253
2254     c = r->connection;
2255
2256     ngx\_log\_debug5(NGX\_LOG\_DEBUG\_HTTP, c->log, 0,
2257         "http finalize request: %d, \"%V?%V\" a:%d, c:%d",
2258         rc, &r->uri, &r->args, r == c->data, r->main->count);
2259
2260     if (rc == NGX\_DONE) {
2261         ngx\_http\_finalize\_connection(r);
2262         return;
2263     }
2264
2265     if (rc == NGX\_OK && r->filter_finalize) {
2266         c->error = 1;
2267     }
2268
2269     if (rc == NGX\_DECLINED) {
2270         r->content_handler = NULL;
2271         r->write_event_handler = ngx\_http\_core\_run\_phases;
2272         ngx\_http\_core\_run\_phases(r);
2273         return;
2274     }
2275
2276     if (r != r->main && r->post_subrequest) {
2277         rc = r->post_subrequest->handler(r, r->post_subrequest->data, rc);
2278     }
2279
2280     if (rc == NGX\_ERROR
2281         || rc == NGX\_HTTP\_REQUEST\_TIME\_OUT
2282         || rc == NGX\_HTTP\_CLIENT\_CLOSED\_REQUEST
2283         || c->error)
2284     {
2285         if (ngx\_http\_post\_action(r) == NGX\_OK) {
2286             return;
2287         }
2288
2289         if (r->main->blocked) {
2290             r->write_event_handler = ngx\_http\_request\_finalizer;
2291         }
2292
2293         ngx\_http\_terminate\_request(r, rc);
2294         return;
2295     }
2296
2297     if (rc >= NGX\_HTTP\_SPECIAL\_RESPONSE
2298         || rc == NGX\_HTTP\_CREATED
2299         || rc == NGX\_HTTP\_NO\_CONTENT)
2300     {
2301         if (rc == NGX\_HTTP\_CLOSE) {

```

```

2303     ngx_http_terminate_request(r, rc);
2304     return;
2305 }
2306
2307 if (r == r->main) {
2308     if (c->read->timer_set) {
2309         ngx_del_timer(c->read);
2310     }
2311
2312     if (c->write->timer_set) {
2313         ngx_del_timer(c->write);
2314     }
2315 }
2316
2317 c->read->handler = ngx_http_request_handler;
2318 c->write->handler = ngx_http_request_handler;
2319
2320 ngx_http_finalize_request(r, ngx_http_special_response_handler(r, rc));
2321 return;
2322 }
2323
2324 if (r != r->main) {
2325
2326     if (r->buffered || r->postponed) {
2327
2328         if (ngx_http_set_write_handler(r) != NGX_OK) {
2329             ngx_http_terminate_request(r, 0);
2330         }
2331
2332         return;
2333     }
2334
2335     pr = r->parent;
2336
2337     if (r == c->data) {
2338
2339         r->main->count--;
2340         r->main->subrequests++;
2341
2342         if (!r->logged) {
2343
2344             clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
2345
2346             if (clcf->log_subrequest) {
2347                 ngx_http_log_request(r);
2348             }
2349
2350             r->logged = 1;
2351
2352         } else {
2353             ngx_log_error(NGX_LOG_ALERT, c->log, 0,
2354                 "subrequest: \"%V?%V\" logged again",
2355                 &r->uri, &r->args);
2356         }
2357
2358         r->done = 1;
2359
2360         if (pr->postponed && pr->postponed->request == r) {
2361             pr->postponed = pr->postponed->next;
2362         }
2363
2364         c->data = pr;
2365
2366     } else {
2367
2368         ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
2369             "http finalize non-active request: \"%V?%V\"",
2370             &r->uri, &r->args);
2371
2372         r->write_event_handler = ngx_http_request_finalizer;
2373
2374         if (r->waited) {
2375             r->done = 1;
2376         }
2377     }
2378

```

```

2379     if (ngx\_http\_post\_request(pr, NULL) != NGX\_OK) {
2380         r->main->count++;
2381         ngx\_http\_terminate\_request(r, 0);
2382         return;
2383     }
2384
2385     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, c->log, 0,
2386                 "http wake parent request: \"%V?%V\"",
2387                 &pr->uri, &pr->args);
2388
2389     return;
2390 }
2391
2392 if (r->buffered || c->buffered || r->postponed || r->blocked) {
2393
2394     if (ngx\_http\_set\_write\_handler(r) != NGX\_OK) {
2395         ngx\_http\_terminate\_request(r, 0);
2396     }
2397
2398     return;
2399 }
2400
2401 if (r != c->data) {
2402     ngx\_log\_error(NGX\_LOG\_ALERT, c->log, 0,
2403                 "http finalize non-active request: \"%V?%V\"",
2404                 &r->uri, &r->args);
2405     return;
2406 }
2407
2408 r->done = 1;
2409 r->write_event_handler = ngx\_http\_request\_empty\_handler;
2410
2411 if (!r->post_action) {
2412     r->request_complete = 1;
2413 }
2414
2415 if (ngx\_http\_post\_action(r) == NGX\_OK) {
2416     return;
2417 }
2418
2419 if (c->read->timer_set) {
2420     ngx\_del\_timer(c->read);
2421 }
2422
2423 if (c->write->timer_set) {
2424     c->write->delayed = 0;
2425     ngx\_del\_timer(c->write);
2426 }
2427
2428 if (c->read->eof) {
2429     ngx\_http\_close\_request(r, 0);
2430     return;
2431 }
2432
2433 ngx\_http\_finalize\_connection(r);
2434 }
2435
2436
2437 static void
2438 ngx\_http\_terminate\_request(ngx\_http\_request\_t *r, ngx\_int\_t rc)
2439 {
2440     ngx\_http\_cleanup\_t *cIn;
2441     ngx\_http\_request\_t *mr;
2442     ngx\_http\_ephemeral\_t *e;
2443
2444     mr = r->main;
2445
2446     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
2447                 "http terminate request count:%d", mr->count);
2448
2449     if (rc > 0 && (mr->headers_out.status == 0 || mr->connection->sent == 0)) {
2450         mr->headers_out.status = rc;
2451     }
2452
2453     cIn = mr->cleanup;
2454     mr->cleanup = NULL;

```

```

2455
2456 while (cIn) {
2457     if (cIn->handler) {
2458         cIn->handler(cIn->data);
2459     }
2460
2461     cIn = cIn->next;
2462 }
2463
2464 ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
2465     "http terminate cleanup count:%d blk:%d",
2466     mr->count, mr->blocked);
2467
2468 if (mr->write_event_handler) {
2469
2470     if (mr->blocked) {
2471         return;
2472     }
2473
2474     e = ngx\_http\_ephemeral(mr);
2475     mr->posted_requests = NULL;
2476     mr->write_event_handler = ngx\_http\_terminate\_handler;
2477     (void) ngx\_http\_post\_request(mr, &e->terminal_posted_request);
2478     return;
2479 }
2480
2481 ngx\_http\_close\_request(mr, rc);
2482 }
2483
2484
2485 static void
2486 ngx\_http\_terminate\_handler(ngx\_http\_request\_t *r)
2487 {
2488     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
2489         "http terminate handler count:%d", r->count);
2490
2491     r->count = 1;
2492
2493     ngx\_http\_close\_request(r, 0);
2494 }
2495
2496
2497 static void
2498 ngx\_http\_finalize\_connection(ngx\_http\_request\_t *r)
2499 {
2500     ngx\_http\_core\_loc\_conf\_t *clcf;
2501
2502     #if (NGX\_HTTP\_SPDY)
2503     if (r->spdy_stream) {
2504         ngx\_http\_close\_request(r, 0);
2505         return;
2506     }
2507     #endif
2508
2509     clcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_core\_module);
2510
2511     if (r->main->count != 1) {
2512
2513         if (r->discard_body) {
2514             r->read_event_handler = ngx\_http\_discarded\_request\_body\_handler;
2515             ngx\_add\_timer(r->connection->read, clcf->lingering_timeout);
2516
2517             if (r->lingering_time == 0) {
2518                 r->lingering_time = ngx\_time()
2519                     + (time_t) (clcf->lingering_time / 1000);
2520             }
2521         }
2522
2523         ngx\_http\_close\_request(r, 0);
2524         return;
2525     }
2526
2527     if (!ngx\_terminate
2528         && !ngx\_exiting
2529         && r->keepalive
2530         && clcf->keepalive_timeout > 0)

```

```

2531     {
2532         ngx\_http\_set\_keepalive(r);
2533         return;
2534     }
2535
2536     if (clcf->lingering_close == NGX\_HTTP\_LINGERING\_ALWAYS
2537         || (clcf->lingering_close == NGX\_HTTP\_LINGERING\_ON
2538             && (r->lingering_close
2539                 || r->header_in->pos < r->header_in->last
2540                 || r->connection->read->ready)))
2541     {
2542         ngx\_http\_set\_lingering\_close(r);
2543         return;
2544     }
2545
2546     ngx\_http\_close\_request(r, 0);
2547 }
2548
2549
2550 static ngx\_int\_t
2551 ngx\_http\_set\_write\_handler(ngx\_http\_request\_t *r)
2552 {
2553     ngx\_event\_t          *wev;
2554     ngx\_http\_core\_loc\_conf\_t *clcf;
2555
2556     r->http_state = NGX\_HTTP\_WRITING\_REQUEST\_STATE;
2557
2558     r->read_event_handler = r->discard_body ?
2559         ngx\_http\_discarded\_request\_body\_handler:
2560         ngx\_http\_test\_reading;
2561     r->write_event_handler = ngx\_http\_writer;
2562
2563     #if (NGX\_HTTP\_SPDY)
2564         if (r->spdy_stream) {
2565             return NGX\_OK;
2566         }
2567     #endif
2568
2569     wev = r->connection->write;
2570
2571     if (wev->ready && wev->delayed) {
2572         return NGX\_OK;
2573     }
2574
2575     clcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_core\_module);
2576     if (!wev->delayed) {
2577         ngx\_add\_timer(wev, clcf->send_timeout);
2578     }
2579
2580     if (ngx\_handle\_write\_event(wev, clcf->send_lowat) != NGX\_OK) {
2581         ngx\_http\_close\_request(r, 0);
2582         return NGX\_ERROR;
2583     }
2584
2585     return NGX\_OK;
2586 }
2587
2588
2589 static void
2590 ngx\_http\_writer(ngx\_http\_request\_t *r)
2591 {
2592     int          rc;
2593     ngx\_event\_t  *wev;
2594     ngx\_connection\_t *c;
2595     ngx\_http\_core\_loc\_conf\_t *clcf;
2596
2597     c = r->connection;
2598     wev = c->write;
2599
2600     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, wev->log, 0,
2601                 "http writer handler: \"%V?%V\"", &r->uri, &r->args);
2602
2603     clcf = ngx\_http\_get\_module\_loc\_conf(r->main, ngx\_http\_core\_module);
2604
2605     if (wev->timedout) {
2606         if (!wev->delayed) {

```

```

2607     ngx_log_error(NGX_LOG_INFO, c->log, NGX_ETIMEDOUT,
2608                 "client timed out");
2609     c->timedout = 1;
2610
2611     ngx_http_finalize_request(r, NGX_HTTP_REQUEST_TIME_OUT);
2612     return;
2613 }
2614
2615 wev->timedout = 0;
2616 wev->delayed = 0;
2617
2618 if (!wev->ready) {
2619     ngx_add_timer(wev, clcf->send_timeout);
2620
2621     if (ngx_handle_write_event(wev, clcf->send_lowat) != NGX_OK) {
2622         ngx_http_close_request(r, 0);
2623     }
2624
2625     return;
2626 }
2627
2628 }
2629
2630 if (wev->delayed || r->aio) {
2631     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, wev->log, 0,
2632                  "http writer delayed");
2633
2634     if (ngx_handle_write_event(wev, clcf->send_lowat) != NGX_OK) {
2635         ngx_http_close_request(r, 0);
2636     }
2637
2638     return;
2639 }
2640
2641 rc = ngx_http_output_filter(r, NULL);
2642
2643 ngx_log_debug3(NGX_LOG_DEBUG_HTTP, c->log, 0,
2644               "http writer output filter: %d, \"%V?%V\"",
2645               rc, &r->uri, &r->args);
2646
2647 if (rc == NGX_ERROR) {
2648     ngx_http_finalize_request(r, rc);
2649     return;
2650 }
2651
2652 if (r->buffered || r->postponed || (r == r->main && c->buffered)) {
2653
2654 #if (NGX_HTTP_SPDY)
2655     if (r->spdy_stream) {
2656         return;
2657     }
2658 #endif
2659
2660     if (!wev->delayed) {
2661         ngx_add_timer(wev, clcf->send_timeout);
2662     }
2663
2664     if (ngx_handle_write_event(wev, clcf->send_lowat) != NGX_OK) {
2665         ngx_http_close_request(r, 0);
2666     }
2667
2668     return;
2669 }
2670
2671 ngx_log_debug2(NGX_LOG_DEBUG_HTTP, wev->log, 0,
2672               "http writer done: \"%V?%V\"", &r->uri, &r->args);
2673
2674 r->write_event_handler = ngx_http_request_empty_handler;
2675
2676 ngx_http_finalize_request(r, rc);
2677 }
2678
2679
2680 static void
2681 ngx_http_request_finalizer(ngx_http_request_t *r)
2682 {

```

```

2683     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
2684                 "http finalizer done: \"%V?%V\"", &r->uri, &r->args);
2685
2686     ngx_http_finalize_request(r, 0);
2687 }
2688
2689
2690 void
2691 ngx_http_block_reading(ngx_http_request_t *r)
2692 {
2693     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
2694                 "http reading blocked");
2695
2696     /* aio does not call this handler */
2697
2698     if ((ngx_event_flags & NGX_USE_LEVEL_EVENT)
2699         && r->connection->read->active)
2700     {
2701         if (ngx_del_event(r->connection->read, NGX_READ_EVENT, 0) != NGX_OK) {
2702             ngx_http_close_request(r, 0);
2703         }
2704     }
2705 }
2706
2707
2708 void
2709 ngx_http_test_reading(ngx_http_request_t *r)
2710 {
2711     int             n;
2712     char            buf[1];
2713     ngx_err_t       err;
2714     ngx_event_t     *rev;
2715     ngx_connection_t *c;
2716
2717     c = r->connection;
2718     rev = c->read;
2719
2720     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, c->log, 0, "http test reading");
2721
2722     #if (NGX_HTTP_SPDY)
2723
2724     if (r->spdy_stream) {
2725         if (c->error) {
2726             err = 0;
2727             goto closed;
2728         }
2729
2730         return;
2731     }
2732
2733     #endif
2734
2735     #if (NGX_HAVE_KQUEUE)
2736
2737     if (ngx_event_flags & NGX_USE_KQUEUE_EVENT) {
2738
2739         if (!rev->pending_eof) {
2740             return;
2741         }
2742
2743         rev->eof = 1;
2744         c->error = 1;
2745         err = rev->kq_errno;
2746
2747         goto closed;
2748     }
2749
2750     #endif
2751
2752     #if (NGX_HAVE_EPOLLRDHUP)
2753
2754     if ((ngx_event_flags & NGX_USE_EPOLL_EVENT) && rev->pending_eof) {
2755         socklen_t len;
2756
2757         rev->eof = 1;
2758         c->error = 1;

```



```

2759     err = 0;
2760     len = sizeof(ngx\_err\_t);
2761
2762     /*
2763      * BSDs and Linux return 0 and set a pending error in err
2764      * Solaris returns -1 and sets errno
2765      */
2766
2767     if (getsockopt(c->fd, SOL_SOCKET, SO_ERROR, (void *) &err, &len)
2768         == -1)
2769     {
2770         err = ngx\_socket\_errno;
2771     }
2772
2773     goto closed;
2774 }
2775
2776 #endif
2777
2778     n = recv(c->fd, buf, 1, MSG_PEEK);
2779
2780     if (n == 0) {
2781         rev->eof = 1;
2782         c->error = 1;
2783         err = 0;
2784
2785         goto closed;
2786     }
2787
2788     } else if (n == -1) {
2789         err = ngx\_socket\_errno;
2790
2791         if (err != NGX\_EAGAIN) {
2792             rev->eof = 1;
2793             c->error = 1;
2794
2795             goto closed;
2796         }
2797     }
2798
2799     /* aio does not call this handler */
2800
2801     if ((ngx\_event\_flags & NGX\_USE\_LEVEL\_EVENT) && rev->active) {
2802
2803         if (ngx\_del\_event(rev, NGX\_READ\_EVENT, 0) != NGX\_OK) {
2804             ngx\_http\_close\_request(r, 0);
2805         }
2806     }
2807
2808     return;
2809
2810 closed:
2811
2812     if (err) {
2813         rev->error = 1;
2814     }
2815
2816     ngx\_log\_error(NGX\_LOG\_INFO, c->log, err,
2817                 "client prematurely closed connection");
2818
2819     ngx\_http\_finalize\_request(r, NGX\_HTTP\_CLIENT\_CLOSED\_REQUEST);
2820 }
2821
2822
2823 static void
2824 ngx\_http\_set\_keepalive(ngx\_http\_request\_t *r)
2825 {
2826     int                tcp_nodelay;
2827     ngx\_int\_t         i;
2828     ngx\_buf\_t         *b, *f;
2829     ngx\_event\_t       *rev, *wev;
2830     ngx\_connection\_t *c;
2831     ngx\_http\_connection\_t *hc;
2832     ngx\_http\_core\_srv\_conf\_t *cscf;
2833     ngx\_http\_core\_loc\_conf\_t *clcf;
2834

```

```

2835     c = r->connection;
2836     rev = c->read;
2837
2838     clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
2839
2840     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, c->log, 0, "set http keepalive handler");
2841
2842     if (r->discard_body) {
2843         r->write_event_handler = ngx_http_request_empty_handler;
2844         r->lingering_time = ngx_time() + (time_t) (clcf->lingering_time / 1000);
2845         ngx_add_timer(rev, clcf->lingering_timeout);
2846         return;
2847     }
2848
2849     c->log->action = "closing request";
2850
2851     hc = r->http_connection;
2852     b = r->header_in;
2853
2854     if (b->pos < b->last) {
2855         /* the pipelined request */
2856
2857         if (b != c->buffer) {
2858             /*
2859              * If the large header buffers were allocated while the previous
2860              * request processing then we do not use c->buffer for
2861              * the pipelined request (see ngx_http_create_request()).
2862              *
2863              * Now we would move the large header buffers to the free list.
2864              */
2865
2866             cscf = ngx_http_get_module_srv_conf(r, ngx_http_core_module);
2867
2868             if (hc->free == NULL) {
2869                 hc->free = ngx_palloc(c->pool,
2870                     cscf->large_client_header_buffers.num * sizeof(ngx_buf_t *));
2871
2872                 if (hc->free == NULL) {
2873                     ngx_http_close_request(r, 0);
2874                     return;
2875                 }
2876             }
2877
2878             for (i = 0; i < hc->nbusy - 1; i++) {
2879                 f = hc->busy[i];
2880                 hc->free[hc->nfree++] = f;
2881                 f->pos = f->start;
2882                 f->last = f->start;
2883             }
2884
2885             hc->busy[0] = b;
2886             hc->nbusy = 1;
2887         }
2888     }
2889
2890     /* guard against recursive call from ngx_http_finalize_connection() */
2891     r->keepalive = 0;
2892
2893     ngx_http_free_request(r, 0);
2894
2895     c->data = hc;
2896
2897     if (ngx_handle_read_event(rev, 0) != NGX_OK) {
2898         ngx_http_close_connection(c);
2899         return;
2900     }
2901
2902     wev = c->write;
2903     wev->handler = ngx_http_empty_handler;
2904
2905     if (b->pos < b->last) {
2906         ngx_log_debug0(NGX_LOG_DEBUG_HTTP, c->log, 0, "pipelined request");
2907     }

```

```

2911     c->log->action = "reading client pipelined request line";
2912
2913     r = ngx_http_create_request(c);
2914     if (r == NULL) {
2915         ngx_http_close_connection(c);
2916         return;
2917     }
2918
2919     r->pipeline = 1;
2920
2921     c->data = r;
2922
2923     c->sent = 0;
2924     c->destroyed = 0;
2925
2926     if (rev->timer_set) {
2927         ngx_del_timer(rev);
2928     }
2929
2930     rev->handler = ngx_http_process_request_line;
2931     ngx_post_event(rev, &ngx_posted_events);
2932     return;
2933 }
2934
2935 /*
2936  * To keep a memory footprint as small as possible for an idle keepalive
2937  * connection we try to free c->buffer's memory if it was allocated outside
2938  * the c->pool. The large header buffers are always allocated outside the
2939  * c->pool and are freed too.
2940  */
2941
2942 b = c->buffer;
2943
2944 if (ngx_pfree(c->pool, b->start) == NGX_OK) {
2945     /*
2946      * the special note for ngx_http_keepalive_handler() that
2947      * c->buffer's memory was freed
2948      */
2949
2950     b->pos = NULL;
2951
2952 } else {
2953     b->pos = b->start;
2954     b->last = b->start;
2955 }
2956
2957 ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0, "hc free: %p %d",
2958             hc->free, hc->nfree);
2959
2960 if (hc->free) {
2961     for (i = 0; i < hc->nfree; i++) {
2962         ngx_pfree(c->pool, hc->free[i]->start);
2963         hc->free[i] = NULL;
2964     }
2965
2966     hc->nfree = 0;
2967 }
2968
2969 ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0, "hc busy: %p %d",
2970             hc->busy, hc->nbusy);
2971
2972 if (hc->busy) {
2973     for (i = 0; i < hc->nbusy; i++) {
2974         ngx_pfree(c->pool, hc->busy[i]->start);
2975         hc->busy[i] = NULL;
2976     }
2977
2978     hc->nbusy = 0;
2979 }
2980
2981 #if (NGX_HTTP_SSL)
2982     if (c->ssl) {
2983         ngx_ssl_free_buffer(c);
2984     }
2985 #endif
2986

```

```

2987
2988     rev->handler = ngx\_http\_keepalive\_handler;
2989
2990     if (wev->active && (ngx\_event\_flags & NGX\_USE\_LEVEL\_EVENT)) {
2991         if (ngx\_del\_event(wev, NGX\_WRITE\_EVENT, 0) != NGX\_OK) {
2992             ngx\_http\_close\_connection(c);
2993             return;
2994         }
2995     }
2996
2997     c->log->action = "keepalive";
2998
2999     if (c->tcp_nopush == NGX\_TCP\_NOPUSH\_SET) {
3000         if (ngx\_tcp\_push(c->fd) == -1) {
3001             ngx\_connection\_error(c, ngx\_socket\_errno, ngx\_tcp\_push\_n " failed");
3002             ngx\_http\_close\_connection(c);
3003             return;
3004         }
3005
3006         c->tcp_nopush = NGX\_TCP\_NOPUSH\_UNSET;
3007         tcp_nodelay = ngx\_tcp\_nodelay\_and\_tcp\_nopush ? 1 : 0;
3008
3009     } else {
3010         tcp_nodelay = 1;
3011     }
3012
3013     if (tcp_nodelay
3014         && clcf->tcp_nodelay
3015         && c->tcp_nodelay == NGX\_TCP\_NODELAY\_UNSET)
3016     {
3017         ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, c->log, 0, "tcp_nodelay");
3018
3019         if (setsockopt(c->fd, IPPROTO_TCP, TCP_NODELAY,
3020                       (const void *) &tcp_nodelay, sizeof(int))
3021             == -1)
3022         {
3023             #if (NGX\_SOLARIS)
3024                 /* Solaris returns EINVAL if a socket has been shut down */
3025                 c->log_error = NGX\_ERROR\_IGNORE\_EINVAL;
3026             #endif
3027
3028             ngx\_connection\_error(c, ngx\_socket\_errno,
3029                                 "setsockopt(TCP_NODELAY) failed");
3030
3031             c->log_error = NGX\_ERROR\_INFO;
3032             ngx\_http\_close\_connection(c);
3033             return;
3034         }
3035
3036         c->tcp_nodelay = NGX\_TCP\_NODELAY\_SET;
3037     }
3038
3039     #if 0
3040         /* if ngx\_http\_request\_t was freed then we need some other place */
3041         r->http_state = NGX\_HTTP\_KEEPALIVE\_STATE;
3042     #endif
3043
3044     c->idle = 1;
3045     ngx\_reusable\_connection(c, 1);
3046
3047     ngx\_add\_timer(rev, clcf->keepalive_timeout);
3048
3049     if (rev->ready) {
3050         ngx\_post\_event(rev, &ngx\_posted\_events);
3051     }
3052 }
3053
3054
3055 static void
3056 ngx\_http\_keepalive\_handler(ngx\_event\_t *rev)
3057 {
3058     size\_t          size;
3059     ssize\_t        n;
3060     ngx\_buf\_t      *b;
3061     ngx\_connection\_t *c;
3062

```

```

3063     c = rev->data;
3064
3065     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, c->log, 0, "http keepalive handler");
3066
3067     if (rev->timedout || c->close) {
3068         ngx\_http\_close\_connection(c);
3069         return;
3070     }
3071
3072     #if (NGX_HAVE_KQUEUE)
3073
3074     if (ngx\_event\_flags & NGX\_USE\_KQUEUE\_EVENT) {
3075         if (rev->pending_eof) {
3076             c->log->handler = NULL;
3077             ngx\_log\_error(NGX\_LOG\_INFO, c->log, rev->kq_errno,
3078                 "kevent() reported that client %V closed "
3079                 "keepalive connection", &c->addr_text);
3080         #if (NGX_HTTP_SSL)
3081             if (c->ssl) {
3082                 c->ssl->no_send_shutdown = 1;
3083             }
3084         #endif
3085         ngx\_http\_close\_connection(c);
3086         return;
3087     }
3088 }
3089
3090 #endif
3091
3092     b = c->buffer;
3093     size = b->end - b->start;
3094
3095     if (b->pos == NULL) {
3096
3097         /*
3098          * The c->buffer's memory was freed by ngx\_http\_set\_keepalive().
3099          * However, the c->buffer->start and c->buffer->end were not changed
3100          * to keep the buffer size.
3101          */
3102
3103         b->pos = ngx\_palloc(c->pool, size);
3104         if (b->pos == NULL) {
3105             ngx\_http\_close\_connection(c);
3106             return;
3107         }
3108
3109         b->start = b->pos;
3110         b->last = b->pos;
3111         b->end = b->pos + size;
3112     }
3113
3114     /*
3115      * MSIE closes a keepalive connection with RST flag
3116      * so we ignore ECONNRESET here.
3117      */
3118
3119     c->log_error = NGX_ERROR_IGNORE_ECONNRESET;
3120     ngx\_set\_socket\_errno(0);
3121
3122     n = c->recv(c, b->last, size);
3123     c->log_error = NGX_ERROR_INFO;
3124
3125     if (n == NGX\_AGAIN) {
3126         if (ngx\_handle\_read\_event(rev, 0) != NGX\_OK) {
3127             ngx\_http\_close\_connection(c);
3128             return;
3129         }
3130
3131         /*
3132          * Like ngx\_http\_set\_keepalive() we are trying to not hold
3133          * c->buffer's memory for a keepalive connection.
3134          */
3135
3136         if (ngx\_pfree(c->pool, b->start) == NGX\_OK) {
3137
3138             /*

```

```

3139         * the special note that c->buffer's memory was freed
3140         */
3141
3142         b->pos = NULL;
3143     }
3144
3145     return;
3146 }
3147
3148 if (n == NGX_ERROR) {
3149     ngx_http_close_connection(c);
3150     return;
3151 }
3152
3153 c->log->handler = NULL;
3154
3155 if (n == 0) {
3156     ngx_log_error(NGX_LOG_INFO, c->log, ngx_socket_errno,
3157                 "client %V closed keepalive connection", &c->addr_text);
3158     ngx_http_close_connection(c);
3159     return;
3160 }
3161
3162 b->last += n;
3163
3164 c->log->handler = ngx_http_log_error;
3165 c->log->action = "reading client request line";
3166
3167 c->idle = 0;
3168 ngx_reusable_connection(c, 0);
3169
3170 c->data = ngx_http_create_request(c);
3171 if (c->data == NULL) {
3172     ngx_http_close_connection(c);
3173     return;
3174 }
3175
3176 c->sent = 0;
3177 c->destroyed = 0;
3178
3179 ngx_del_timer(rev);
3180
3181 rev->handler = ngx_http_process_request_line;
3182 ngx_http_process_request_line(rev);
3183 }
3184
3185
3186 static void
3187 ngx_http_set_lingering_close(ngx_http_request_t *r)
3188 {
3189     ngx_event_t          *rev, *wev;
3190     ngx_connection_t     *c;
3191     ngx_http_core_loc_conf_t *clcf;
3192
3193     c = r->connection;
3194
3195     clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
3196
3197     rev = c->read;
3198     rev->handler = ngx_http_lingering_close_handler;
3199
3200     r->lingering_time = ngx_time() + (time_t) (clcf->lingering_time / 1000);
3201     ngx_add_timer(rev, clcf->lingering_timeout);
3202
3203     if (ngx_handle_read_event(rev, 0) != NGX_OK) {
3204         ngx_http_close_request(r, 0);
3205         return;
3206     }
3207
3208     wev = c->write;
3209     wev->handler = ngx_http_empty_handler;
3210
3211     if (wev->active && (ngx_event_flags & NGX_USE_LEVEL_EVENT)) {
3212         if (ngx_del_event(wev, NGX_WRITE_EVENT, 0) != NGX_OK) {
3213             ngx_http_close_request(r, 0);
3214             return;

```

```

3215     }
3216 }
3217
3218 if (ngx\_shutdown\_socket(c->fd, NGX\_WRITE\_SHUTDOWN) == -1) {
3219     ngx\_connection\_error(c, ngx\_socket\_errno,
3220         ngx\_shutdown\_socket\_n " failed");
3221     ngx\_http\_close\_request(r, 0);
3222     return;
3223 }
3224
3225 if (rev->ready) {
3226     ngx\_http\_lingering\_close\_handler(rev);
3227 }
3228 }
3229
3230
3231 static void
3232 ngx\_http\_lingering\_close\_handler(ngx\_event\_t *rev)
3233 {
3234     ssize\_t                n;
3235     ngx\_msec\_t            timer;
3236     ngx\_connection\_t     *c;
3237     ngx\_http\_request\_t   *r;
3238     ngx\_http\_core\_loc\_conf\_t *clcf;
3239     u\_char                buffer[NGX\_HTTP\_LINGERING\_BUFFER\_SIZE];
3240
3241     c = rev->data;
3242     r = c->data;
3243
3244     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, c->log, 0,
3245         "http lingering close handler");
3246
3247     if (rev->timedout) {
3248         ngx\_http\_close\_request(r, 0);
3249         return;
3250     }
3251
3252     timer = (ngx\_msec\_t) r->lingering_time - (ngx\_msec\_t) ngx\_time();
3253     if ((ngx\_msec\_int\_t) timer <= 0) {
3254         ngx\_http\_close\_request(r, 0);
3255         return;
3256     }
3257
3258     do {
3259         n = c->recv(c, buffer, NGX\_HTTP\_LINGERING\_BUFFER\_SIZE);
3260
3261         ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, c->log, 0, "lingering read: %d", n);
3262
3263         if (n == NGX\_ERROR || n == 0) {
3264             ngx\_http\_close\_request(r, 0);
3265             return;
3266         }
3267
3268     } while (rev->ready);
3269
3270     if (ngx\_handle\_read\_event(rev, 0) != NGX\_OK) {
3271         ngx\_http\_close\_request(r, 0);
3272         return;
3273     }
3274
3275     clcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_core\_module);
3276
3277     timer *= 1000;
3278
3279     if (timer > clcf->lingering_timeout) {
3280         timer = clcf->lingering_timeout;
3281     }
3282
3283     ngx\_add\_timer(rev, timer);
3284 }
3285
3286
3287 void
3288 ngx\_http\_empty\_handler(ngx\_event\_t *wev)
3289 {
3290     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, wev->log, 0, "http empty handler");

```

```

3291     return;
3292 }
3293 }
3294
3295
3296 void
3297 ngx_http_request_empty_handler(ngx_http_request_t *r)
3298 {
3299     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
3300                 "http request empty handler");
3301
3302     return;
3303 }
3304
3305
3306 ngx_int_t
3307 ngx_http_send_special(ngx_http_request_t *r, ngx_uint_t flags)
3308 {
3309     ngx_buf_t     *b;
3310     ngx_chain_t   out;
3311
3312     b = ngx_calloc_buf(r->pool);
3313     if (b == NULL) {
3314         return NGX_ERROR;
3315     }
3316
3317     if (flags & NGX_HTTP_LAST) {
3318
3319         if (r == r->main && !r->post_action) {
3320             b->last_buf = 1;
3321
3322         } else {
3323             b->sync = 1;
3324             b->last_in_chain = 1;
3325         }
3326     }
3327
3328     if (flags & NGX_HTTP_FLUSH) {
3329         b->flush = 1;
3330     }
3331
3332     out.buf = b;
3333     out.next = NULL;
3334
3335     return ngx_http_output_filter(r, &out);
3336 }
3337
3338
3339 static ngx_int_t
3340 ngx_http_post_action(ngx_http_request_t *r)
3341 {
3342     ngx_http_core_loc_conf_t *clcf;
3343
3344     clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
3345
3346     if (clcf->post_action.data == NULL) {
3347         return NGX_DECLINED;
3348     }
3349
3350     if (r->post_action && r->uri_changes == 0) {
3351         return NGX_DECLINED;
3352     }
3353
3354     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
3355                 "post action: \"%V\"", &clcf->post_action);
3356
3357     r->main->count--;
3358
3359     r->http_version = NGX_HTTP_VERSION_9;
3360     r->header_only = 1;
3361     r->post_action = 1;
3362
3363     r->read_event_handler = ngx_http_block_reading;
3364
3365     if (clcf->post_action.data[0] == '/') {
3366         ngx_http_internal_redirect(r, &clcf->post_action, NULL);

```



```

3367     } else {
3368         ngx_http_named_location(r, &clcf->post_action);
3369     }
3370 }
3371
3372     return NGX_OK;
3373 }
3374
3375 static void
3376 ngx_http_close_request(ngx_http_request_t *r, ngx_int_t rc)
3377 {
3378     ngx_connection_t *c;
3379
3380     r = r->main;
3381     c = r->connection;
3382
3383     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
3384                 "http request count:%d blk:%d", r->count, r->blocked);
3385
3386     if (r->count == 0) {
3387         ngx_log_error(NGX_LOG_ALERT, c->log, 0, "http request count is zero");
3388     }
3389
3390     r->count--;
3391
3392     if (r->count || r->blocked) {
3393         return;
3394     }
3395 }
3396
3397 #if (NGX_HTTP_SPDY)
3398     if (r->spdy_stream) {
3399         ngx_http_spdy_close_stream(r->spdy_stream, rc);
3400         return;
3401     }
3402 #endif
3403
3404     ngx_http_free_request(r, rc);
3405     ngx_http_close_connection(c);
3406 }
3407
3408 void
3409 ngx_http_free_request(ngx_http_request_t *r, ngx_int_t rc)
3410 {
3411     ngx_log_t *log;
3412     ngx_pool_t *pool;
3413     struct linger linger;
3414     ngx_http_cleanup_t *cln;
3415     ngx_http_log_ctx_t *ctx;
3416     ngx_http_core_loc_conf_t *clcf;
3417
3418     log = r->connection->log;
3419
3420     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, log, 0, "http close request");
3421
3422     if (r->pool == NULL) {
3423         ngx_log_error(NGX_LOG_ALERT, log, 0, "http request already closed");
3424         return;
3425     }
3426 }
3427
3428     cln = r->cleanup;
3429     r->cleanup = NULL;
3430
3431     while (cln) {
3432         if (cln->handler) {
3433             cln->handler(cln->data);
3434         }
3435
3436         cln = cln->next;
3437     }
3438
3439 #if (NGX_STAT_STUB)
3440     if (r->stat_reading) {
3441         (void) ngx_atomic_fetch_add(ngx_stat_reading, -1);

```

```

3443     }
3444
3445     if (r->stat_writing) {
3446         (void) ngx\_atomic\_fetch\_add(ngx\_stat\_writing, -1);
3447     }
3448
3449 #endif
3450
3451     if (rc > 0 && (r->headers_out.status == 0 || r->connection->sent == 0)) {
3452         r->headers_out.status = rc;
3453     }
3454
3455     log->action = "logging request";
3456
3457     ngx\_http\_log\_request(r);
3458
3459     log->action = "closing request";
3460
3461     if (r->connection->timedout) {
3462         clcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_core\_module);
3463
3464         if (clcf->reset_timedout_connection) {
3465             linger.l_onoff = 1;
3466             linger.l_linger = 0;
3467
3468             if (setsockopt(r->connection->fd, SOL_SOCKET, SO_LINGER,
3469                 (const void *) &linger, sizeof(struct linger)) == -1)
3470             {
3471                 ngx\_log\_error(NGX\_LOG\_ALERT, log, ngx\_socket\_errno,
3472                     "setsockopt(SO_LINGER) failed");
3473             }
3474         }
3475     }
3476
3477     /* the various request strings were allocated from r->pool */
3478     ctx = log->data;
3479     ctx->request = NULL;
3480
3481     r->request_line.len = 0;
3482
3483     r->connection->destroyed = 1;
3484
3485     /*
3486      * Setting r->pool to NULL will increase probability to catch double close
3487      * of request since the request object is allocated from its own pool.
3488     */
3489
3490     pool = r->pool;
3491     r->pool = NULL;
3492
3493     ngx\_destroy\_pool(pool);
3494 }
3495
3496
3497 static void
3498 ngx\_http\_log\_request(ngx\_http\_request\_t *r)
3499 {
3500     ngx\_uint\_t          i, n;
3501     ngx\_http\_handler\_pt *log_handler;
3502     ngx\_http\_core\_main\_conf\_t *cmcf;
3503
3504     cmcf = ngx\_http\_get\_module\_main\_conf(r, ngx\_http\_core\_module);
3505
3506     log_handler = cmcf->phases[NGX\_HTTP\_LOG\_PHASE].handlers.elts;
3507     n = cmcf->phases[NGX\_HTTP\_LOG\_PHASE].handlers.nelts;
3508
3509     for (i = 0; i < n; i++) {
3510         log_handler[i](r);
3511     }
3512 }
3513
3514
3515 void
3516 ngx\_http\_close\_connection(ngx\_connection\_t *c)
3517 {
3518     ngx\_pool\_t *pool;

```

```

3519     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, c->log, 0,
3520                 "close http connection: %d", c->fd);
3521
3522
3523 #if (NGX\_HTTP\_SSL)
3524
3525     if (c->ssl) {
3526         if (ngx\_ssl\_shutdown(c) == NGX\_AGAIN) {
3527             c->ssl->handler = ngx\_http\_close\_connection;
3528             return;
3529         }
3530     }
3531
3532 #endif
3533
3534 #if (NGX\_STAT\_STUB)
3535     (void) ngx\_atomic\_fetch\_add(ngx\_stat\_active, -1);
3536 #endif
3537
3538     c->destroyed = 1;
3539
3540     pool = c->pool;
3541
3542     ngx\_close\_connection(c);
3543
3544     ngx\_destroy\_pool(pool);
3545 }
3546
3547
3548 static u_char *
3549 ngx\_http\_log\_error(ngx\_log\_t *log, u_char *buf, size_t len)
3550 {
3551     u_char          *p;
3552     ngx\_http\_request\_t *r;
3553     ngx\_http\_log\_ctx\_t *ctx;
3554
3555     if (log->action) {
3556         p = ngx\_snprintf(buf, len, " while %s", log->action);
3557         len -= p - buf;
3558         buf = p;
3559     }
3560
3561     ctx = log->data;
3562
3563     p = ngx\_snprintf(buf, len, ", client: %V", &ctx->connection->addr_text);
3564     len -= p - buf;
3565
3566     r = ctx->request;
3567
3568     if (r) {
3569         return r->log_handler(r, ctx->current_request, p, len);
3570     } else {
3571         p = ngx\_snprintf(p, len, ", server: %V",
3572                         &ctx->connection->listening->addr_text);
3573     }
3574
3575     return p;
3576 }
3577
3578
3579
3580 static u_char *
3581 ngx\_http\_log\_error\_handler(ngx\_http\_request\_t *r, ngx\_http\_request\_t *sr,
3582 u_char *buf, size_t len)
3583 {
3584     char          *uri_separator;
3585     u_char        *p;
3586     ngx\_http\_upstream\_t *u;
3587     ngx\_http\_core\_srv\_conf\_t *cscf;
3588
3589     cscf = ngx\_http\_get\_module\_srv\_conf(r, ngx\_http\_core\_module);
3590
3591     p = ngx\_snprintf(buf, len, ", server: %V", &cscf->server_name);
3592     len -= p - buf;
3593     buf = p;
3594

```

```

3595     if (r->request_line.data == NULL && r->request_start) {
3596         for (p = r->request_start; p < r->header_in->last; p++) {
3597             if (*p == CR || *p == LF) {
3598                 break;
3599             }
3600         }
3601
3602         r->request_line.len = p - r->request_start;
3603         r->request_line.data = r->request_start;
3604     }
3605
3606     if (r->request_line.len) {
3607         p = ngx_snprintf(buf, len, " request: \"%V\"", &r->request_line);
3608         len -= p - buf;
3609         buf = p;
3610     }
3611
3612     if (r != sr) {
3613         p = ngx_snprintf(buf, len, " subrequest: \"%V\"", &sr->uri);
3614         len -= p - buf;
3615         buf = p;
3616     }
3617
3618     u = sr->upstream;
3619
3620     if (u && u->peer.name) {
3621
3622         uri_separator = "";
3623
3624         #if (NGX_HAVE_UNIX_DOMAIN)
3625             if (u->peer.sockaddr && u->peer.sockaddr->sa_family == AF_UNIX) {
3626                 uri_separator = ".";
3627             }
3628         #endif
3629
3630         p = ngx_snprintf(buf, len, " upstream: \"%V%V%S%V\"",
3631                         &u->schema, u->peer.name,
3632                         uri_separator, &u->uri);
3633         len -= p - buf;
3634         buf = p;
3635     }
3636
3637     if (r->headers_in.host) {
3638         p = ngx_snprintf(buf, len, " host: \"%V\"",
3639                         &r->headers_in.host->value);
3640         len -= p - buf;
3641         buf = p;
3642     }
3643
3644     if (r->headers_in.referer) {
3645         p = ngx_snprintf(buf, len, " referer: \"%V\"",
3646                         &r->headers_in.referer->value);
3647         buf = p;
3648     }
3649
3650     return buf;
3651 }

```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_ssl\_module.h - nginx-1.7.10

## Data types defined

- [ngx\\_http\\_ssl\\_srv\\_conf\\_t](#)

## Macros defined

- [\\_NGX\\_HTTP\\_SSL\\_H\\_INCLUDED](#)

## Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #ifndef \_NGX\_HTTP\_SSL\_H\_INCLUDED
9  #define \_NGX\_HTTP\_SSL\_H\_INCLUDED
10
11
12  #include <ngx_config.h>
13  #include <ngx_core.h>
14  #include <ngx_http.h>
15
16
17  typedef struct {
18      ngx\_flag\_t                enable;
19
20      ngx\_ssl\_t                ssl;
21
22      ngx\_flag\_t                prefer_server_ciphers;
23
24      ngx\_uint\_t                protocols;
25
26      ngx\_uint\_t                verify;
27      ngx\_uint\_t                verify_depth;
28
29      size_t                  buffer_size;
30
31      ssize_t                 builtin_session_cache;
32
33      time_t                  session_timeout;
34
35      ngx\_str\_t                 certificate;
36      ngx\_str\_t                 certificate_key;
37      ngx\_str\_t                 dhparam;
38      ngx\_str\_t                 ecdh_curve;
39      ngx\_str\_t                 client_certificate;
40      ngx\_str\_t                 trusted_certificate;
41      ngx\_str\_t                 crl;
42
43      ngx\_str\_t                 ciphers;
44
45      ngx\_array\_t               *passwords;
46
47      ngx\_shm\_zone\_t           *shm_zone;
48
49      ngx\_flag\_t                session_tickets;
50      ngx\_array\_t               *session_ticket_keys;
51
52      ngx\_flag\_t                stapling;
53      ngx\_flag\_t                stapling_verify;
54      ngx\_str\_t                 stapling_file;
55      ngx\_str\_t                 stapling_responder;
56
57      u_char                  *file;
```

```
58     ngx\_uint\_t                line;
59 } ngx\_http\_ssl\_srv\_conf\_t;
60
61
62 extern ngx\_module\_t ngx\_http\_ssl\_module;
63
64
65 #endif /* \_NGX\_HTTP\_SSL\_H\_INCLUDED */
```

[One Level Up](#)

[Top Level](#)

## src/event/nginx\_event\_openssl.h - nginx-1.7.10

### Data types defined

- [ngx\\_ssl\\_connection\\_t](#)
- [ngx\\_ssl\\_sess\\_id\\_s](#)
- [ngx\\_ssl\\_sess\\_id\\_t](#)
- [ngx\\_ssl\\_session\\_cache\\_t](#)
- [ngx\\_ssl\\_session\\_ticket\\_key\\_t](#)
- [ngx\\_ssl\\_t](#)

### Macros defined

- [NGX\\_SSL\\_BUFFER](#)
- [NGX\\_SSL\\_BUFSIZE](#)
- [NGX\\_SSL\\_CLIENT](#)
- [NGX\\_SSL\\_DFLT\\_BUILTIN\\_SCACHE](#)
- [NGX\\_SSL\\_MAX\\_SESSION\\_SIZE](#)
- [NGX\\_SSL\\_NAME](#)
- [NGX\\_SSL\\_NONE\\_SCACHE](#)
- [NGX\\_SSL\\_NO\\_BUILTIN\\_SCACHE](#)
- [NGX\\_SSL\\_NO\\_SCACHE](#)
- [NGX\\_SSL\\_SSLv2](#)
- [NGX\\_SSL\\_SSLv3](#)
- [NGX\\_SSL\\_TLSv1](#)
- [NGX\\_SSL\\_TLSv1\\_1](#)
- [NGX\\_SSL\\_TLSv1\\_2](#)
- [\\_NGX\\_EVENT\\_OPENSSL\\_H\\_INCLUDED](#)
- [ngx\\_ssl\\_conn\\_t](#)
- [ngx\\_ssl\\_free\\_session](#)
- [ngx\\_ssl\\_get\\_connection](#)
- [ngx\\_ssl\\_get\\_server\\_conf](#)
- [ngx\\_ssl\\_get\\_session](#)
- [ngx\\_ssl\\_session\\_t](#)
- [ngx\\_ssl\\_verify\\_error\\_optional](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef NGX_EVENT_OPENSSL_H_INCLUDED
9 #define NGX_EVENT_OPENSSL_H_INCLUDED
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15 #include <openssl/ssl.h>
16 #include <openssl/err.h>
17 #include <openssl/bn.h>
18 #include <openssl/conf.h>
19 #include <openssl/crypto.h>
20 #include <openssl/dh.h>
21 #ifndef OPENSSL_NO_ENGINE
22 #include <openssl/engine.h>
23 #endif
24 #include <openssl/evp.h>
25 #ifndef OPENSSL_NO_OCSP
26 #include <openssl/ocsp.h>
27 #endif
28 #include <openssl/rand.h>
29 #include <openssl/rsa.h>
30 #include <openssl/x509.h>
31 #include <openssl/x509v3.h>
32
33 #define NGX_SSL_NAME      "OpenSSL"
34
35
36 #define ngx_ssl_session_t      SSL_SESSION
37 #define ngx_ssl_conn_t        SSL
38
39
40 typedef struct {
41     SSL_CTX          *ctx;
42     ngx_log_t        *log;
43     size_t           buffer_size;
44 } ngx_ssl_t;
45
46
47 typedef struct {
48     ngx_ssl_conn_t    *connection;
49
50     ngx_int_t         last;
51     ngx_buf_t         *buf;
52     size_t           buffer_size;
53
54     ngx_connection_handler_pt handler;
55
56     ngx_event_handler_pt saved_read_handler;
57     ngx_event_handler_pt saved_write_handler;
58
59     unsigned         handshaked:1;
60     unsigned         renegotiation:1;
61     unsigned         buffer:1;
62     unsigned         no_wait_shutdown:1;
63     unsigned         no_send_shutdown:1;
64     unsigned         handshake_buffer_set:1;
65 } ngx_ssl_connection_t;
66
67
68 #define NGX_SSL_NO_SCACHE      -2
69 #define NGX_SSL_NONE_SCACHE   -3
70 #define NGX_SSL_NO_BUILTIN_SCACHE -4
71 #define NGX_SSL_DFLT_BUILTIN_SCACHE -5
72
73
```



```

74 #define NGX_SSL_MAX_SESSION_SIZE 4096
75
76 typedef struct ngx_ssl_sess_id_s ngx_ssl_sess_id_t;
77
78 struct ngx_ssl_sess_id_s {
79     ngx_rbtree_node_t    node;
80     u_char               *id;
81     size_t               len;
82     u_char               *session;
83     ngx_queue_t         queue;
84     time_t              expire;
85 #if (NGX_PTR_SIZE == 8)
86     void                *stub;
87     u_char               sess_id[32];
88 #endif
89 };
90
91
92 typedef struct {
93     ngx_rbtree_t         session_rbtrees;
94     ngx_rbtree_node_t    sentinel;
95     ngx_queue_t         expire_queue;
96 } ngx_ssl_session_cache_t;
97
98
99 #ifdef SSL_CTRL_SET_TLSEXT_TICKET_KEY_CB
100
101 typedef struct {
102     u_char               name[16];
103     u_char               aes_key[16];
104     u_char               hmac_key[16];
105 } ngx_ssl_session_ticket_key_t;
106
107 #endif
108
109
110 #define NGX_SSL_SSLV2    0x0002
111 #define NGX_SSL_SSLV3    0x0004
112 #define NGX_SSL_TLSV1    0x0008
113 #define NGX_SSL_TLSV1_1  0x0010
114 #define NGX_SSL_TLSV1_2  0x0020
115
116
117 #define NGX_SSL_BUFFER    1
118 #define NGX_SSL_CLIENT    2
119
120 #define NGX_SSL_BUFSIZE  16384
121
122
123 ngx_int_t ngx_ssl_init(ngx_log_t *log);
124 ngx_int_t ngx_ssl_create(ngx_ssl_t *ssl, ngx_uint_t protocols, void *data);
125 ngx_int_t ngx_ssl_certificate(ngx_conf_t *cf, ngx_ssl_t *ssl,
126     ngx_str_t *cert, ngx_str_t *key, ngx_array_t *passwords);
127 ngx_int_t ngx_ssl_client_certificate(ngx_conf_t *cf, ngx_ssl_t *ssl,
128     ngx_str_t *cert, ngx_int_t depth);
129 ngx_int_t ngx_ssl_trusted_certificate(ngx_conf_t *cf, ngx_ssl_t *ssl,
130     ngx_str_t *cert, ngx_int_t depth);
131 ngx_int_t ngx_ssl_crl(ngx_conf_t *cf, ngx_ssl_t *ssl, ngx_str_t *crl);
132 ngx_int_t ngx_ssl_stapling(ngx_conf_t *cf, ngx_ssl_t *ssl,
133     ngx_str_t *file, ngx_str_t *responder, ngx_uint_t verify);
134 ngx_int_t ngx_ssl_stapling_resolver(ngx_conf_t *cf, ngx_ssl_t *ssl,
135     ngx_resolver_t *resolver, ngx_msec_t resolver_timeout);
136 RSA *ngx_ssl_rsa512_key_callback(ngx_ssl_conn_t *ssl_conn, int is_export,
137     int key_length);
138 ngx_array_t *ngx_ssl_read_password_file(ngx_conf_t *cf, ngx_str_t *file);
139 ngx_int_t ngx_ssl_dhparam(ngx_conf_t *cf, ngx_ssl_t *ssl, ngx_str_t *file);
140 ngx_int_t ngx_ssl_ecdh_curve(ngx_conf_t *cf, ngx_ssl_t *ssl, ngx_str_t *name);
141 ngx_int_t ngx_ssl_session_cache(ngx_ssl_t *ssl, ngx_str_t *sess_ctx,
142     ssize_t builtin_session_cache, ngx_shm_zone_t *shm_zone, time_t timeout);
143 ngx_int_t ngx_ssl_session_ticket_keys(ngx_conf_t *cf, ngx_ssl_t *ssl,
144     ngx_array_t *paths);
145 ngx_int_t ngx_ssl_session_cache_init(ngx_shm_zone_t *shm_zone, void *data);
146 ngx_int_t ngx_ssl_create_connection(ngx_ssl_t *ssl, ngx_connection_t *c,
147     ngx_uint_t flags);
148
149 void ngx_ssl_remove_cached_session(SSL_CTX *ssl, ngx_ssl_session_t *sess);

```

```

150 ngx_int_t ngx_ssl_set_session(ngx_connection_t *c, ngx_ssl_session_t *session);
151 #define ngx_ssl_get_session(c)      SSL_get1_session(c->ssl->connection)
152 #define ngx_ssl_free_session      SSL_SESSION_free
153 #define ngx_ssl_get_connection(ssl_conn)          \
154     SSL_get_ex_data(ssl_conn, ngx_ssl_connection_index)
155 #define ngx_ssl_get_server_conf(ssl_ctx)         \
156     SSL_CTX_get_ex_data(ssl_ctx, ngx_ssl_server_conf_index)
157
158 #define ngx_ssl_verify_error_optional(n)          \
159     (n == X509_V_ERR_DEPTH_ZERO_SELF_SIGNED_CERT \
160      || n == X509_V_ERR_SELF_SIGNED_CERT_IN_CHAIN \
161      || n == X509_V_ERR_UNABLE_TO_GET_ISSUER_CERT_LOCALLY \
162      || n == X509_V_ERR_CERT_UNTRUSTED \
163      || n == X509_V_ERR_UNABLE_TO_VERIFY_LEAF_SIGNATURE)
164
165 ngx_int_t ngx_ssl_check_host(ngx_connection_t *c, ngx_str_t *name);
166
167
168 ngx_int_t ngx_ssl_get_protocol(ngx_connection_t *c, ngx_pool_t *pool,
169     ngx_str_t *s);
170 ngx_int_t ngx_ssl_get_cipher_name(ngx_connection_t *c, ngx_pool_t *pool,
171     ngx_str_t *s);
172 ngx_int_t ngx_ssl_get_session_id(ngx_connection_t *c, ngx_pool_t *pool,
173     ngx_str_t *s);
174 ngx_int_t ngx_ssl_get_session_reused(ngx_connection_t *c, ngx_pool_t *pool,
175     ngx_str_t *s);
176 ngx_int_t ngx_ssl_get_server_name(ngx_connection_t *c, ngx_pool_t *pool,
177     ngx_str_t *s);
178 ngx_int_t ngx_ssl_get_raw_certificate(ngx_connection_t *c, ngx_pool_t *pool,
179     ngx_str_t *s);
180 ngx_int_t ngx_ssl_get_certificate(ngx_connection_t *c, ngx_pool_t *pool,
181     ngx_str_t *s);
182 ngx_int_t ngx_ssl_get_subject_dn(ngx_connection_t *c, ngx_pool_t *pool,
183     ngx_str_t *s);
184 ngx_int_t ngx_ssl_get_issuer_dn(ngx_connection_t *c, ngx_pool_t *pool,
185     ngx_str_t *s);
186 ngx_int_t ngx_ssl_get_serial_number(ngx_connection_t *c, ngx_pool_t *pool,
187     ngx_str_t *s);
188 ngx_int_t ngx_ssl_get_fingerprint(ngx_connection_t *c, ngx_pool_t *pool,
189     ngx_str_t *s);
190 ngx_int_t ngx_ssl_get_client_verify(ngx_connection_t *c, ngx_pool_t *pool,
191     ngx_str_t *s);
192
193
194 ngx_int_t ngx_ssl_handshake(ngx_connection_t *c);
195 ssize_t ngx_ssl_recv(ngx_connection_t *c, u_char *buf, size_t size);
196 ssize_t ngx_ssl_write(ngx_connection_t *c, u_char *data, size_t size);
197 ssize_t ngx_ssl_recv_chain(ngx_connection_t *c, ngx_chain_t *cl, off_t limit);
198 ngx_chain_t *ngx_ssl_send_chain(ngx_connection_t *c, ngx_chain_t *in,
199     off_t limit);
200 void ngx_ssl_free_buffer(ngx_connection_t *c);
201 ngx_int_t ngx_ssl_shutdown(ngx_connection_t *c);
202 void ngx_cdecl ngx_ssl_error(ngx_uint_t level, ngx_log_t *log, ngx_err_t err,
203     char *fmt, ...);
204 void ngx_ssl_cleanup_ctx(void *data);
205
206
207 extern int ngx_ssl_connection_index;
208 extern int ngx_ssl_server_conf_index;
209 extern int ngx_ssl_session_cache_index;
210 extern int ngx_ssl_session_ticket_keys_index;
211 extern int ngx_ssl_certificate_index;
212 extern int ngx_ssl_stapling_index;
213
214
215 #endif /* NGX_EVENT_OPENSSL_H_INCLUDED */

```

[One Level Up](#)

[Top Level](#)

## src/event/nginx\_event\_openssl.c - nginx-1.7.10

### Global variables defined

- [ngx\\_openssl\\_commands](#)
- [ngx\\_openssl\\_module](#)
- [ngx\\_openssl\\_module\\_ctx](#)
- [ngx\\_ssl\\_certificate\\_index](#)
- [ngx\\_ssl\\_connection\\_index](#)
- [ngx\\_ssl\\_server\\_conf\\_index](#)
- [ngx\\_ssl\\_session\\_cache\\_index](#)
- [ngx\\_ssl\\_session\\_ticket\\_keys\\_index](#)
- [ngx\\_ssl\\_stapling\\_index](#)

### Data types defined

- [ngx\\_openssl\\_conf\\_t](#)

### Functions defined

- [ngx\\_openssl\\_create\\_conf](#)
- [ngx\\_openssl\\_engine](#)
- [ngx\\_openssl\\_exit](#)
- [ngx\\_ssl\\_certificate](#)
- [ngx\\_ssl\\_check\\_host](#)
- [ngx\\_ssl\\_check\\_name](#)
- [ngx\\_ssl\\_cleanup\\_ctx](#)
- [ngx\\_ssl\\_clear\\_error](#)
- [ngx\\_ssl\\_client\\_certificate](#)
- [ngx\\_ssl\\_connection\\_error](#)
- [ngx\\_ssl\\_create](#)
- [ngx\\_ssl\\_create\\_connection](#)
- [ngx\\_ssl\\_crl](#)
- [ngx\\_ssl\\_dhparam](#)
- [ngx\\_ssl\\_ecdh\\_curve](#)
- [ngx\\_ssl\\_error](#)
- [ngx\\_ssl\\_expire\\_sessions](#)

- [ngx\\_ssl\\_free\\_buffer](#)
- [ngx\\_ssl\\_get\\_cached\\_session](#)
- [ngx\\_ssl\\_get\\_certificate](#)
- [ngx\\_ssl\\_get\\_cipher\\_name](#)
- [ngx\\_ssl\\_get\\_client\\_verify](#)
- [ngx\\_ssl\\_get\\_fingerprint](#)
- [ngx\\_ssl\\_get\\_issuer\\_dn](#)
- [ngx\\_ssl\\_get\\_protocol](#)
- [ngx\\_ssl\\_get\\_raw\\_certificate](#)
- [ngx\\_ssl\\_get\\_serial\\_number](#)
- [ngx\\_ssl\\_get\\_server\\_name](#)
- [ngx\\_ssl\\_get\\_session\\_id](#)
- [ngx\\_ssl\\_get\\_session\\_reused](#)
- [ngx\\_ssl\\_get\\_subject\\_dn](#)
- [ngx\\_ssl\\_handle\\_recv](#)
- [ngx\\_ssl\\_handshake](#)
- [ngx\\_ssl\\_handshake\\_handler](#)
- [ngx\\_ssl\\_info\\_callback](#)
- [ngx\\_ssl\\_init](#)
- [ngx\\_ssl\\_new\\_session](#)
- [ngx\\_ssl\\_password\\_callback](#)
- [ngx\\_ssl\\_passwords\\_cleanup](#)
- [ngx\\_ssl\\_read\\_handler](#)
- [ngx\\_ssl\\_read\\_password\\_file](#)
- [ngx\\_ssl\\_recv](#)
- [ngx\\_ssl\\_recv\\_chain](#)
- [ngx\\_ssl\\_remove\\_cached\\_session](#)
- [ngx\\_ssl\\_remove\\_session](#)
- [ngx\\_ssl\\_rsa512\\_key\\_callback](#)
- [ngx\\_ssl\\_send\\_chain](#)
- [ngx\\_ssl\\_session\\_cache](#)
- [ngx\\_ssl\\_session\\_cache\\_init](#)
- [ngx\\_ssl\\_session\\_id\\_context](#)

- [ngx\\_ssl\\_session\\_rbtrees\\_insert\\_value](#)
- [ngx\\_ssl\\_session\\_ticket\\_key\\_callback](#)
- [ngx\\_ssl\\_session\\_ticket\\_keys](#)
- [ngx\\_ssl\\_session\\_ticket\\_keys](#)
- [ngx\\_ssl\\_set\\_session](#)
- [ngx\\_ssl\\_shutdown](#)
- [ngx\\_ssl\\_shutdown\\_handler](#)
- [ngx\\_ssl\\_trusted\\_certificate](#)
- [ngx\\_ssl\\_verify\\_callback](#)
- [ngx\\_ssl\\_write](#)
- [ngx\\_ssl\\_write\\_handler](#)

## Macros defined

- [NGX\\_SSL\\_PASSWORD\\_BUFFER\\_SIZE](#)
- [ngx\\_ssl\\_session\\_ticket\\_md](#)
- [ngx\\_ssl\\_session\\_ticket\\_md](#)

## Source code

```

1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 #define NGX_SSL_PASSWORD_BUFFER_SIZE 4096
14
15
16 typedef struct {
17     ngx_uint_t engine; /* unsigned engine:1; */
18 } ngx_openssl_conf_t;
19
20
21 static int ngx_ssl_password_callback(char *buf, int size, int rwflag,
22     void *userdata);
23 static int ngx_ssl_verify_callback(int ok, X509_STORE_CTX *x509_store);
24 static void ngx_ssl_info_callback(const ngx_ssl_conn_t *ssl_conn, int where,
25     int ret);
26 static void ngx_ssl_passwords_cleanup(void *data);
27 static void ngx_ssl_handshake_handler(ngx_event_t *ev);
28 static ngx_int_t ngx_ssl_handle_recv(ngx_connection_t *c, int n);
29 static void ngx_ssl_write_handler(ngx_event_t *wev);
30 static void ngx_ssl_read_handler(ngx_event_t *rev);
31 static void ngx_ssl_shutdown_handler(ngx_event_t *ev);
32 static void ngx_ssl_connection_error(ngx_connection_t *c, int sslerr,
33     ngx_err_t err, char *text);
34 static void ngx_ssl_clear_error(ngx_log_t *log);
35
36 static ngx_int_t ngx_ssl_session_id_context(ngx_ssl_t *ssl,
37     ngx_str_t *sess_ctx);

```

```

38 ngx_int_t ngx_ssl_session_cache_init(ngx_shm_zone_t *shm_zone, void *data);
39 static int ngx_ssl_new_session(ngx_ssl_conn_t *ssl_conn,
40     ngx_ssl_session_t *sess);
41 static ngx_ssl_session_t *ngx_ssl_get_cached_session(ngx_ssl_conn_t *ssl_conn,
42     u_char *id, int len, int *copy);
43 static void ngx_ssl_remove_session(SSL_CTX *ssl, ngx_ssl_session_t *sess);
44 static void ngx_ssl_expire_sessions(ngx_ssl_session_cache_t *cache,
45     ngx_slab_pool_t *shpool, ngx_uint_t n);
46 static void ngx_ssl_session_rbtrees_insert_value(ngx_rbtrees_node_t *temp,
47     ngx_rbtrees_node_t *node, ngx_rbtrees_node_t *sentinel);
48
49 #ifdef SSL_CTRL_SET_TLSEXT_TICKET_KEY_CB
50 static int ngx_ssl_session_ticket_key_callback(ngx_ssl_conn_t *ssl_conn,
51     unsigned char *name, unsigned char *iv, EVP_CIPHER_CTX *ectx,
52     HMAC_CTX *hctx, int enc);
53 #endif
54
55 #if (OPENSSL_VERSION_NUMBER < 0x10002002L || defined LIBRESSL_VERSION_NUMBER)
56 static ngx_int_t ngx_ssl_check_name(ngx_str_t *name, ASN1_STRING *str);
57 #endif
58
59 static void *ngx_openssl_create_conf(ngx_cycle_t *cycle);
60 static char *ngx_openssl_engine(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
61 static void ngx_openssl_exit(ngx_cycle_t *cycle);
62
63
64 static ngx_command_t ngx_openssl_commands[] = {
65     { ngx_string("ssl_engine"),
66         NGX_MAIN_CONF|NGX_DIRECT_CONF|NGX_CONF_TAKE1,
67         ngx_openssl_engine,
68         0,
69         0,
70         NULL },
71     ngx_null_command
72 };
73
74 };
75
76
77 static ngx_core_module_t ngx_openssl_module_ctx = {
78     ngx_string("openssl"),
79     ngx_openssl_create_conf,
80     NULL
81 };
82
83
84 ngx_module_t ngx_openssl_module = {
85     NGX_MODULE_V1,
86     &ngx_openssl_module_ctx,          /* module context */
87     ngx_openssl_commands,            /* module directives */
88     NGX_CORE_MODULE,                 /* module type */
89     NULL,                             /* init master */
90     NULL,                             /* init module */
91     NULL,                             /* init process */
92     NULL,                             /* init thread */
93     NULL,                             /* exit thread */
94     NULL,                             /* exit process */
95     ngx_openssl_exit,                /* exit master */
96     NGX_MODULE_V1_PADDING
97 };
98
99
100 int ngx_ssl_connection_index;
101 int ngx_ssl_server_conf_index;
102 int ngx_ssl_session_cache_index;
103 int ngx_ssl_session_ticket_keys_index;
104 int ngx_ssl_certificate_index;
105 int ngx_ssl_stapling_index;
106
107
108 ngx_int_t
109 ngx_ssl_init(ngx_log_t *log)
110 {
111     #ifndef OPENSSL_IS_BORINGSSL
112         OPENSSL_config(NULL);
113     #endif

```

```

114     SSL_library_init();
115     SSL_load_error_strings();
116
117     OpenSSL_add_all_algorithms();
118
119
120     #if OPENSSSL_VERSION_NUMBER >= 0x0090800fL
121     #ifndef SSL_OP_NO_COMPRESSION
122     {
123     /*
124     * Disable gzip compression in OpenSSL prior to 1.0.0 version,
125     * this saves about 522K per connection.
126     */
127     int n;
128     STACK_OF(SSL_COMP) *ssl_comp_methods;
129
130     ssl_comp_methods = SSL_COMP_get_compression_methods();
131     n = sk_SSL_COMP_num(ssl_comp_methods);
132
133     while (n--) {
134         (void) sk_SSL_COMP_pop(ssl_comp_methods);
135     }
136     }
137     #endif
138     #endif
139
140     ngx_ssl_connection_index = SSL_get_ex_new_index(0, NULL, NULL, NULL, NULL);
141
142     if (ngx_ssl_connection_index == -1) {
143         ngx_ssl_error(NGX_LOG_ALERT, log, 0, "SSL_get_ex_new_index() failed");
144         return NGX_ERROR;
145     }
146
147     ngx_ssl_server_conf_index = SSL_CTX_get_ex_new_index(0, NULL, NULL, NULL,
148                                                         NULL);
149     if (ngx_ssl_server_conf_index == -1) {
150         ngx_ssl_error(NGX_LOG_ALERT, log, 0,
151                     "SSL_CTX_get_ex_new_index() failed");
152         return NGX_ERROR;
153     }
154
155     ngx_ssl_session_cache_index = SSL_CTX_get_ex_new_index(0, NULL, NULL, NULL,
156                                                         NULL);
157     if (ngx_ssl_session_cache_index == -1) {
158         ngx_ssl_error(NGX_LOG_ALERT, log, 0,
159                     "SSL_CTX_get_ex_new_index() failed");
160         return NGX_ERROR;
161     }
162
163     ngx_ssl_session_ticket_keys_index = SSL_CTX_get_ex_new_index(0, NULL, NULL,
164                                                                 NULL, NULL);
165     if (ngx_ssl_session_ticket_keys_index == -1) {
166         ngx_ssl_error(NGX_LOG_ALERT, log, 0,
167                     "SSL_CTX_get_ex_new_index() failed");
168         return NGX_ERROR;
169     }
170
171     ngx_ssl_certificate_index = SSL_CTX_get_ex_new_index(0, NULL, NULL, NULL,
172                                                         NULL);
173     if (ngx_ssl_certificate_index == -1) {
174         ngx_ssl_error(NGX_LOG_ALERT, log, 0,
175                     "SSL_CTX_get_ex_new_index() failed");
176         return NGX_ERROR;
177     }
178
179     ngx_ssl_stapling_index = SSL_CTX_get_ex_new_index(0, NULL, NULL, NULL,
180                                                         NULL);
181     if (ngx_ssl_stapling_index == -1) {
182         ngx_ssl_error(NGX_LOG_ALERT, log, 0,
183                     "SSL_CTX_get_ex_new_index() failed");
184         return NGX_ERROR;
185     }
186
187     return NGX_OK;
188 }
189

```

```

190 ngx_int_t
191 ngx_ssl_create(ngx_ssl_t *ssl, ngx_uint_t protocols, void *data)
192 {
193     ssl->ctx = SSL_CTX_new(SSLv23_method());
194
195     if (ssl->ctx == NULL) {
196         ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0, "SSL_CTX_new() failed");
197         return NGX_ERROR;
198     }
199
200     if (SSL_CTX_set_ex_data(ssl->ctx, ngx_ssl_server_conf_index, data) == 0) {
201         ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
202             "SSL_CTX_set_ex_data() failed");
203         return NGX_ERROR;
204     }
205
206     ssl->buffer_size = NGX_SSL_BUFSIZE;
207
208     /* client side options */
209
210 #ifdef SSL_OP_MICROSOFT_SESS_ID_BUG
211     SSL_CTX_set_options(ssl->ctx, SSL_OP_MICROSOFT_SESS_ID_BUG);
212 #endif
213
214 #ifdef SSL_OP_NETSCAPE_CHALLENGE_BUG
215     SSL_CTX_set_options(ssl->ctx, SSL_OP_NETSCAPE_CHALLENGE_BUG);
216 #endif
217
218     /* server side options */
219
220 #ifdef SSL_OP_SSLREF2_REUSE_CERT_TYPE_BUG
221     SSL_CTX_set_options(ssl->ctx, SSL_OP_SSLREF2_REUSE_CERT_TYPE_BUG);
222 #endif
223
224 #ifdef SSL_OP_MICROSOFT_BIG_SSLV3_BUFFER
225     SSL_CTX_set_options(ssl->ctx, SSL_OP_MICROSOFT_BIG_SSLV3_BUFFER);
226 #endif
227
228 #ifdef SSL_OP_MSIE_SSLV2_RSA_PADDING
229     /* this option allow a potential SSL 2.0 rollback (CAN-2005-2969) */
230     SSL_CTX_set_options(ssl->ctx, SSL_OP_MSIE_SSLV2_RSA_PADDING);
231 #endif
232
233 #ifdef SSL_OP_SSLEAY_080_CLIENT_DH_BUG
234     SSL_CTX_set_options(ssl->ctx, SSL_OP_SSLEAY_080_CLIENT_DH_BUG);
235 #endif
236
237 #ifdef SSL_OP_TLS_D5_BUG
238     SSL_CTX_set_options(ssl->ctx, SSL_OP_TLS_D5_BUG);
239 #endif
240
241 #ifdef SSL_OP_TLS_BLOCK_PADDING_BUG
242     SSL_CTX_set_options(ssl->ctx, SSL_OP_TLS_BLOCK_PADDING_BUG);
243 #endif
244
245 #ifdef SSL_OP_DONT_INSERT_EMPTY_FRAGMENTS
246     SSL_CTX_set_options(ssl->ctx, SSL_OP_DONT_INSERT_EMPTY_FRAGMENTS);
247 #endif
248
249     SSL_CTX_set_options(ssl->ctx, SSL_OP_SINGLE_DH_USE);
250
251     if (!(protocols & NGX_SSL_SSLV2)) {
252         SSL_CTX_set_options(ssl->ctx, SSL_OP_NO_SSLV2);
253     }
254     if (!(protocols & NGX_SSL_SSLV3)) {
255         SSL_CTX_set_options(ssl->ctx, SSL_OP_NO_SSLV3);
256     }
257     if (!(protocols & NGX_SSL_TLSv1)) {
258         SSL_CTX_set_options(ssl->ctx, SSL_OP_NO_TLSv1);
259     }
260 #ifdef SSL_OP_NO_TLSv1_1
261     if (!(protocols & NGX_SSL_TLSv1_1)) {
262         SSL_CTX_set_options(ssl->ctx, SSL_OP_NO_TLSv1_1);
263     }
264 #endif
265 }

```



```

266 #ifndef SSL_OP_NO_TLSv1_2
267     if (!(protocols & NGX\_SSL\_TLSv1\_2)) {
268         SSL_CTX_set_options(ssl->ctx, SSL_OP_NO_TLSv1_2);
269     }
270 #endif
271
272 #ifndef SSL_OP_NO_COMPRESSION
273     SSL_CTX_set_options(ssl->ctx, SSL_OP_NO_COMPRESSION);
274 #endif
275
276 #ifndef SSL_MODE_RELEASE_BUFFERS
277     SSL_CTX_set_mode(ssl->ctx, SSL_MODE_RELEASE_BUFFERS);
278 #endif
279
280     SSL_CTX_set_read_ahead(ssl->ctx, 1);
281
282     SSL_CTX_set_info_callback(ssl->ctx, ngx\_ssl\_info\_callback);
283
284     return NGX\_OK;
285 }
286
287
288 ngx\_int\_t
289 ngx\_ssl\_certificate(ngx\_conf\_t *cf, ngx\_ssl\_t *ssl, ngx\_str\_t *cert,
290 ngx\_str\_t *key, ngx\_array\_t *passwords)
291 {
292     BIO          *bio;
293     X509         *x509;
294     u_long       n;
295     ngx\_str\_t    *pwd;
296     ngx\_uint\_t   tries;
297
298     if (ngx\_conf\_full\_name(cf->cycle, cert, 1) != NGX\_OK) {
299         return NGX\_ERROR;
300     }
301
302     /*
303      * we can't use SSL_CTX_use_certificate_chain_file() as it doesn't
304      * allow to access certificate later from SSL_CTX, so we reimplement
305      * it here
306      */
307
308     bio = BIO_new_file((char *) cert->data, "r");
309     if (bio == NULL) {
310         ngx\_ssl\_error(NGX\_LOG\_EMERG, ssl->log, 0,
311             "BIO_new_file(\"%s\") failed", cert->data);
312         return NGX\_ERROR;
313     }
314
315     x509 = PEM_read_bio_X509_AUX(bio, NULL, NULL, NULL);
316     if (x509 == NULL) {
317         ngx\_ssl\_error(NGX\_LOG\_EMERG, ssl->log, 0,
318             "PEM_read_bio_X509_AUX(\"%s\") failed", cert->data);
319         BIO_free(bio);
320         return NGX\_ERROR;
321     }
322
323     if (SSL_CTX_use_certificate(ssl->ctx, x509) == 0) {
324         ngx\_ssl\_error(NGX\_LOG\_EMERG, ssl->log, 0,
325             "SSL_CTX_use_certificate(\"%s\") failed", cert->data);
326         X509_free(x509);
327         BIO_free(bio);
328         return NGX\_ERROR;
329     }
330
331     if (SSL_CTX_set_ex_data(ssl->ctx, ngx\_ssl\_certificate\_index, x509)
332         == 0)
333     {
334         ngx\_ssl\_error(NGX\_LOG\_EMERG, ssl->log, 0,
335             "SSL_CTX_set_ex_data() failed");
336         X509_free(x509);
337         BIO_free(bio);
338         return NGX\_ERROR;
339     }
340
341     X509_free(x509);

```

```

342
343 /* read rest of the chain */
344
345 for ( ;; ) {
346
347     x509 = PEM_read_bio_X509(bio, NULL, NULL, NULL);
348     if (x509 == NULL) {
349         n = ERR_peek_last_error();
350
351         if (ERR_GET_LIB(n) == ERR_LIB_PEM
352             && ERR_GET_REASON(n) == PEM_R_NO_START_LINE)
353         {
354             /* end of file */
355             ERR_clear_error();
356             break;
357         }
358
359         /* some real error */
360
361         ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
362                     "PEM_read_bio_X509(\"%s\") failed", cert->data);
363         BIO_free(bio);
364         return NGX_ERROR;
365     }
366
367     if (SSL_CTX_add_extra_chain_cert(ssl->ctx, x509) == 0) {
368         ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
369                     "SSL_CTX_add_extra_chain_cert(\"%s\") failed",
370                     cert->data);
371         X509_free(x509);
372         BIO_free(bio);
373         return NGX_ERROR;
374     }
375 }
376
377 BIO_free(bio);
378
379 if (ngx_strncmp(key->data, "engine:", sizeof("engine:") - 1) == 0) {
380
381 #ifndef OPENSSSL_NO_ENGINE
382
383     u_char      *p, *last;
384     ENGINE      *engine;
385     EVP_PKEY    *pkey;
386
387     p = key->data + sizeof("engine:") - 1;
388     last = (u_char *) ngx_strchr(p, ':');
389
390     if (last == NULL) {
391         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
392                             "invalid syntax in \"%V\"", key);
393         return NGX_ERROR;
394     }
395
396     *last = '\0';
397
398     engine = ENGINE_by_id((char *) p);
399
400     if (engine == NULL) {
401         ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
402                     "ENGINE_by_id(\"%s\") failed", p);
403         return NGX_ERROR;
404     }
405
406     *last++ = ':';
407
408     pkey = ENGINE_load_private_key(engine, (char *) last, 0, 0);
409
410     if (pkey == NULL) {
411         ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
412                     "ENGINE_load_private_key(\"%s\") failed", last);
413         ENGINE_free(engine);
414         return NGX_ERROR;
415     }
416
417     ENGINE_free(engine);

```

```

418     if (SSL_CTX_use_PrivateKey(ssl->ctx, pkey) == 0) {
419         ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
420             "SSL_CTX_use_PrivateKey(\"%s\") failed", last);
421         EVP_PKEY_free(pkey);
422         return NGX_ERROR;
423     }
424
425     EVP_PKEY_free(pkey);
426
427     return NGX_OK;
428
429 #else
430
431     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
432         "loading \"engine:...\" certificate keys "
433         "is not supported");
434     return NGX_ERROR;
435
436 #endif
437 }
438
439 if (ngx_conf_full_name(cf->cycle, key, 1) != NGX_OK) {
440     return NGX_ERROR;
441 }
442
443 if (passwords) {
444     tries = passwords->nelts;
445     pwd = passwords->elts;
446
447     SSL_CTX_set_default_passwd_cb(ssl->ctx, ngx_ssl_password_callback);
448     SSL_CTX_set_default_passwd_cb_userdata(ssl->ctx, pwd);
449
450 } else {
451     tries = 1;
452 #if (NGX_SUPPRESS_WARN)
453     pwd = NULL;
454 #endif
455 #endif
456 }
457
458 for ( ;; ) {
459     if (SSL_CTX_use_PrivateKey_file(ssl->ctx, (char *) key->data,
460         SSL_FILETYPE_PEM)
461         != 0)
462     {
463         break;
464     }
465
466     if (--tries) {
467         ERR_clear_error();
468         SSL_CTX_set_default_passwd_cb_userdata(ssl->ctx, ++pwd);
469         continue;
470     }
471
472     ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
473         "SSL_CTX_use_PrivateKey_file(\"%s\") failed", key->data);
474     return NGX_ERROR;
475 }
476
477 SSL_CTX_set_default_passwd_cb(ssl->ctx, NULL);
478
479 return NGX_OK;
480 }
481
482
483
484 static int
485 ngx_ssl_password_callback(char *buf, int size, int rwflag, void *userdata)
486 {
487     ngx_str_t *pwd = userdata;
488
489     if (rwflag) {
490         ngx_log_error(NGX_LOG_ALERT, ngx_cycle->log, 0,
491             "ngx_ssl_password_callback() is called for encryption");
492         return 0;
493     }

```

```

494     if (pwd->len > (size_t) size) {
495         ngx_log_error(NGX_LOG_ERR, ngx_cycle->log, 0,
496             "password is truncated to %d bytes", size);
497     } else {
498         size = pwd->len;
499     }
500 }
501
502 ngx_memcpy(buf, pwd->data, size);
503
504 return size;
505 }
506
507
508 ngx_int_t
509 ngx_ssl_client_certificate(ngx_conf_t *cf, ngx_ssl_t *ssl, ngx_str_t *cert,
510     ngx_int_t depth)
511 {
512     STACK_OF(X509_NAME) *list;
513
514     SSL_CTX_set_verify(ssl->ctx, SSL_VERIFY_PEER, ngx_ssl_verify_callback);
515
516     SSL_CTX_set_verify_depth(ssl->ctx, depth);
517
518     if (cert->len == 0) {
519         return NGX_OK;
520     }
521
522     if (ngx_conf_full_name(cf->cycle, cert, 1) != NGX_OK) {
523         return NGX_ERROR;
524     }
525
526     if (SSL_CTX_load_verify_locations(ssl->ctx, (char *) cert->data, NULL)
527         == 0)
528     {
529         ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
530             "SSL_CTX_load_verify_locations(\"%s\") failed",
531             cert->data);
532         return NGX_ERROR;
533     }
534
535     /*
536      * SSL_CTX_load_verify_locations() may leave errors in the error queue
537      * while returning success
538      */
539
540     ERR_clear_error();
541
542     list = SSL_load_client_CA_file((char *) cert->data);
543
544     if (list == NULL) {
545         ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
546             "SSL_load_client_CA_file(\"%s\") failed", cert->data);
547         return NGX_ERROR;
548     }
549
550     /*
551      * before 0.9.7h and 0.9.8 SSL_load_client_CA_file()
552      * always leaved an error in the error queue
553      */
554
555     ERR_clear_error();
556
557     SSL_CTX_set_client_CA_list(ssl->ctx, list);
558
559     return NGX_OK;
560 }
561
562
563 ngx_int_t
564 ngx_ssl_trusted_certificate(ngx_conf_t *cf, ngx_ssl_t *ssl, ngx_str_t *cert,
565     ngx_int_t depth)
566 {
567     SSL_CTX_set_verify_depth(ssl->ctx, depth);
568
569     if (cert->len == 0) {

```

```

570     return NGX_OK;
571 }
572
573 if (ngx_conf_full_name(cf->cycle, cert, 1) != NGX_OK) {
574     return NGX_ERROR;
575 }
576
577 if (SSL_CTX_load_verify_locations(ssl->ctx, (char *) cert->data, NULL)
578     == 0)
579 {
580     ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
581                 "SSL_CTX_load_verify_locations(\"%s\") failed",
582                 cert->data);
583     return NGX_ERROR;
584 }
585
586 /*
587  * SSL_CTX_load_verify_locations() may leave errors in the error queue
588  * while returning success
589  */
590
591 ERR_clear_error();
592
593 return NGX_OK;
594 }
595
596
597 ngx_int_t
598 ngx_ssl_crl(ngx_conf_t *cf, ngx_ssl_t *ssl, ngx_str_t *crl)
599 {
600     X509_STORE *store;
601     X509_LOOKUP *lookup;
602
603     if (crl->len == 0) {
604         return NGX_OK;
605     }
606
607     if (ngx_conf_full_name(cf->cycle, crl, 1) != NGX_OK) {
608         return NGX_ERROR;
609     }
610
611     store = SSL_CTX_get_cert_store(ssl->ctx);
612
613     if (store == NULL) {
614         ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
615                     "SSL_CTX_get_cert_store() failed");
616         return NGX_ERROR;
617     }
618
619     lookup = X509_STORE_add_lookup(store, X509_LOOKUP_file());
620
621     if (lookup == NULL) {
622         ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
623                     "X509_STORE_add_lookup() failed");
624         return NGX_ERROR;
625     }
626
627     if (X509_LOOKUP_load_file(lookup, (char *) crl->data, X509_FILETYPE_PEM)
628         == 0)
629     {
630         ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
631                     "X509_LOOKUP_load_file(\"%s\") failed", crl->data);
632         return NGX_ERROR;
633     }
634
635     X509_STORE_set_flags(store,
636                         X509_V_FLAG_CRL_CHECK|X509_V_FLAG_CRL_CHECK_ALL);
637
638     return NGX_OK;
639 }
640
641
642 static int
643 ngx_ssl_verify_callback(int ok, X509_STORE_CTX *x509_store)
644 {
645     #if (NGX_DEBUG)

```

```

646 char          *subject, *issuer;
647 int           err, depth;
648 X509          *cert;
649 X509_NAME     *sname, *iname;
650 ngx\_connection\_t *c;
651 ngx\_ssl\_conn\_t *ssl_conn;
652
653 ssl_conn = X509_STORE_CTX_get_ex_data(x509_store,
654                                     SSL_get_ex_data_X509_STORE_CTX_idx());
655
656 c = ngx\_ssl\_get\_connection(ssl_conn);
657
658 cert = X509_STORE_CTX_get_current_cert(x509_store);
659 err = X509_STORE_CTX_get_error(x509_store);
660 depth = X509_STORE_CTX_get_error_depth(x509_store);
661
662 sname = X509_get_subject_name(cert);
663 subject = sname ? X509_NAME_oneline(sname, NULL, 0) : "(none)";
664
665 iname = X509_get_issuer_name(cert);
666 issuer = iname ? X509_NAME_oneline(iname, NULL, 0) : "(none)";
667
668 ngx\_log\_debug5(NGX\_LOG\_DEBUG\_EVENT, c->log, 0,
669              "verify:%d, error:%d, depth:%d, "
670              "subject:\"%s\", issuer:\"%s\"",
671              ok, err, depth, subject, issuer);
672
673 if (sname) {
674     OPENSSL_free(subject);
675 }
676
677 if (iname) {
678     OPENSSL_free(issuer);
679 }
680 #endif
681
682 return 1;
683 }
684
685
686 static void
687 ngx\_ssl\_info\_callback(const ngx\_ssl\_conn\_t *ssl_conn, int where, int ret)
688 {
689     BIO          *rbio, *wbio;
690     ngx\_connection\_t *c;
691
692     if (where & SSL_CB_HANDSHAKE_START) {
693         c = ngx\_ssl\_get\_connection((ngx\_ssl\_conn\_t *) ssl_conn);
694
695         if (c->ssl->handshaked) {
696             c->ssl->renegotiation = 1;
697             ngx\_log\_debug0(NGX\_LOG\_DEBUG\_EVENT, c->log, 0, "SSL renegotiation");
698         }
699     }
700
701     if ((where & SSL_CB_ACCEPT_LOOP) == SSL_CB_ACCEPT_LOOP) {
702         c = ngx\_ssl\_get\_connection((ngx\_ssl\_conn\_t *) ssl_conn);
703
704         if (!c->ssl->handshake_buffer_set) {
705             /*
706              * By default OpenSSL uses 4k buffer during a handshake,
707              * which is too low for long certificate chains and might
708              * result in extra round-trips.
709              *
710              * To adjust a buffer size we detect that buffering was added
711              * to write side of the connection by comparing rbio and wbio.
712              * If they are different, we assume that it's due to buffering
713              * added to wbio, and set buffer size.
714              */
715
716             rbio = SSL_get_rbio((ngx\_ssl\_conn\_t *) ssl_conn);
717             wbio = SSL_get_wbio((ngx\_ssl\_conn\_t *) ssl_conn);
718
719             if (rbio != wbio) {
720                 (void) BIO_set_write_buffer_size(wbio, NGX\_SSL\_BUFSIZE);
721                 c->ssl->handshake_buffer_set = 1;

```

```

722     }
723 }
724 }
725 }
726
727
728 RSA *
729 ngx_ssl_rsa512_key_callback(ngx_ssl_conn_t *ssl_conn, int is_export,
730 int key_length)
731 {
732     static RSA *key;
733
734     if (key_length != 512) {
735         return NULL;
736     }
737
738 #ifndef OPENSSSL_NO_DEPRECATED
739
740     if (key == NULL) {
741         key = RSA_generate_key(512, RSA_F4, NULL, NULL);
742     }
743
744 #endif
745
746     return key;
747 }
748
749
750 ngx_array_t *
751 ngx_ssl_read_password_file(ngx_conf_t *cf, ngx_str_t *file)
752 {
753     u_char          *p, *last, *end;
754     size_t          len;
755     ssize_t         n;
756     ngx_fd_t        fd;
757     ngx_str_t       *pwd;
758     ngx_array_t     *passwords;
759     ngx_pool_cleanup_t *cIn;
760     u_char          buf[NGX_SSL_PASSWORD_BUFFER_SIZE];
761
762     if (ngx_conf_full_name(cf->cycle, file, 1) != NGX_OK) {
763         return NULL;
764     }
765
766     cIn = ngx_pool_cleanup_add(cf->temp_pool, 0);
767     passwords = ngx_array_create(cf->temp_pool, 4, sizeof(ngx_str_t));
768
769     if (cIn == NULL || passwords == NULL) {
770         return NULL;
771     }
772
773     cIn->handler = ngx_ssl_passwords_cleanup;
774     cIn->data = passwords;
775
776     fd = ngx_open_file(file->data, NGX_FILE_RDONLY, NGX_FILE_OPEN, 0);
777     if (fd == NGX_INVALID_FILE) {
778         ngx_conf_log_error(NGX_LOG_EMERG, cf, ngx_errno,
779             ngx_open_file_n "%s" failed", file->data);
780         return NULL;
781     }
782
783     len = 0;
784     last = buf;
785
786     do {
787         n = ngx_read_fd(fd, last, NGX_SSL_PASSWORD_BUFFER_SIZE - len);
788
789         if (n == -1) {
790             ngx_conf_log_error(NGX_LOG_EMERG, cf, ngx_errno,
791                 ngx_read_fd_n "%s" failed", file->data);
792             passwords = NULL;
793             goto cleanup;
794         }
795
796         end = last + n;
797

```

```

798     if (len && n == 0) {
799         *end++ = LF;
800     }
801
802     p = buf;
803
804     for ( ;; ) {
805         last = ngx_strlchr(last, end, LF);
806
807         if (last == NULL) {
808             break;
809         }
810
811         len = last++ - p;
812
813         if (len && p[len - 1] == CR) {
814             len--;
815         }
816
817         if (len) {
818             pwd = ngx_array_push(passwords);
819             if (pwd == NULL) {
820                 passwords = NULL;
821                 goto cleanup;
822             }
823
824             pwd->len = len;
825             pwd->data = ngx_pnalloc(cf->temp_pool, len);
826
827             if (pwd->data == NULL) {
828                 passwords->nelts--;
829                 passwords = NULL;
830                 goto cleanup;
831             }
832
833             ngx_memcpy(pwd->data, p, len);
834         }
835
836         p = last;
837     }
838
839     len = end - p;
840
841     if (len == NGX_SSL_PASSWORD_BUFFER_SIZE) {
842         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
843             "too long line in \"%s\"", file->data);
844         passwords = NULL;
845         goto cleanup;
846     }
847
848     ngx_memmove(buf, p, len);
849     last = buf + len;
850
851 } while (n != 0);
852
853 if (passwords->nelts == 0) {
854     pwd = ngx_array_push(passwords);
855     if (pwd == NULL) {
856         passwords = NULL;
857         goto cleanup;
858     }
859
860     ngx_memzero(pwd, sizeof(ngx_str_t));
861 }
862
863 cleanup:
864
865 if (ngx_close_file(fd) == NGX_FILE_ERROR) {
866     ngx_conf_log_error(NGX_LOG_ALERT, cf, ngx_errno,
867         ngx_close_file_n " \"%s\" failed", file->data);
868 }
869
870 ngx_memzero(buf, NGX_SSL_PASSWORD_BUFFER_SIZE);
871
872 return passwords;
873 }

```



```

874
875
876 static void
877 ngx_ssl_passwords_cleanup(void *data)
878 {
879     ngx_array_t *passwords = data;
880
881     ngx_str_t   *pwd;
882     ngx_uint_t   i;
883
884     pwd = passwords->elts;
885
886     for (i = 0; i < passwords->nelts; i++) {
887         ngx_memzero(pwd[i].data, pwd[i].len);
888     }
889 }
890
891
892 ngx_int_t
893 ngx_ssl_dhparam(ngx_conf_t *cf, ngx_ssl_t *ssl, ngx_str_t *file)
894 {
895     DH   *dh;
896     BIO  *bio;
897
898     /*
899     * -----BEGIN DH PARAMETERS-----
900     * MIGHAoGBALu8LcrYRnSQfEP89YDpz9vZWKP1aLQtSwju10sPs1BmbAMCducQgAxc
901     * y7qokiYUxb7spwWl/fHSh6K8BJvmd4Bg6RqSp1fjBI9osHb302zI8pu134HcLKcI
902     * 70ZicMyaUDXYzs7vnqAnSm0rHlj6/UmI0PZdFGdX2gcd8EXP4WubAgEC
903     * -----END DH PARAMETERS-----
904     */
905
906     static unsigned char dh1024_p[] = {
907         0xBB, 0xBC, 0x2D, 0xCA, 0xD8, 0x46, 0x74, 0x90, 0x7C, 0x43, 0xFC, 0xF5,
908         0x80, 0xE9, 0xCF, 0xDB, 0xD9, 0x58, 0xA3, 0xF5, 0x68, 0xB4, 0x2D, 0x4B,
909         0x08, 0xEE, 0xD4, 0xEB, 0x0F, 0xB3, 0x50, 0x4C, 0x6C, 0x03, 0x02, 0x76,
910         0xE7, 0x10, 0x80, 0x0C, 0x5C, 0xCB, 0xBA, 0xA8, 0x92, 0x26, 0x14, 0xC5,
911         0xBE, 0xEC, 0xA5, 0x65, 0xA5, 0xFD, 0xF1, 0xD2, 0x87, 0xA2, 0xBC, 0x04,
912         0x9B, 0xE6, 0x77, 0x80, 0x60, 0xE9, 0x1A, 0x92, 0xA7, 0x57, 0xE3, 0x04,
913         0x8F, 0x68, 0xB0, 0x76, 0xF7, 0xD3, 0x6C, 0xC8, 0xF2, 0x9B, 0xA5, 0xDF,
914         0x81, 0xDC, 0x2C, 0xA7, 0x25, 0xEC, 0xE6, 0x62, 0x70, 0xCC, 0x9A, 0x50,
915         0x35, 0xD8, 0xCE, 0xCE, 0xEF, 0x9E, 0xA0, 0x27, 0x4A, 0x63, 0xAB, 0x1E,
916         0x58, 0xFA, 0xFD, 0x49, 0x88, 0xD0, 0xF6, 0x5D, 0x14, 0x67, 0x57, 0xDA,
917         0x07, 0x1D, 0xF0, 0x45, 0xCF, 0xE1, 0x6B, 0x9B
918     };
919
920     static unsigned char dh1024_g[] = { 0x02 };
921
922
923     if (file->len == 0) {
924
925         dh = DH_new();
926         if (dh == NULL) {
927             ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0, "DH_new() failed");
928             return NGX_ERROR;
929         }
930
931         dh->p = BN_bin2bn(dh1024_p, sizeof(dh1024_p), NULL);
932         dh->g = BN_bin2bn(dh1024_g, sizeof(dh1024_g), NULL);
933
934         if (dh->p == NULL || dh->g == NULL) {
935             ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0, "BN_bin2bn() failed");
936             DH_free(dh);
937             return NGX_ERROR;
938         }
939
940         SSL_CTX_set_tmp_dh(ssl->ctx, dh);
941
942         DH_free(dh);
943
944         return NGX_OK;
945     }
946
947     if (ngx_conf_full_name(cf->cycle, file, 1) != NGX_OK) {
948         return NGX_ERROR;
949     }

```

```

950 bio = BIO_new_file((char *) file->data, "r");
951 if (bio == NULL) {
952     ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
953                 "BIO_new_file(\"%s\") failed", file->data);
954     return NGX_ERROR;
955 }
956
957 dh = PEM_read_bio_DHparams(bio, NULL, NULL, NULL);
958 if (dh == NULL) {
959     ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
960                 "PEM_read_bio_DHparams(\"%s\") failed", file->data);
961     BIO_free(bio);
962     return NGX_ERROR;
963 }
964
965 SSL_CTX_set_tmp_dh(ssl->ctx, dh);
966
967 DH_free(dh);
968 BIO_free(bio);
969
970 return NGX_OK;
971 }
972
973
974
975 ngx_int_t
976 ngx_ssl_ecdh_curve(ngx_conf_t *cf, ngx_ssl_t *ssl, ngx_str_t *name)
977 {
978     #if OPENSSSL_VERSION_NUMBER >= 0x0090800fL
979     #ifndef OPENSSSL_NO_ECDH
980         int     nid;
981         EC_KEY *ecdh;
982
983         /*
984          * Elliptic-Curve Diffie-Hellman parameters are either "named curves"
985          * from RFC 4492 section 5.1.1, or explicitly described curves over
986          * binary fields. OpenSSL only supports the "named curves", which provide
987          * maximum interoperability.
988          */
989
990         nid = OBJ_sn2nid((const char *) name->data);
991         if (nid == 0) {
992             ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
993                         "Unknown curve name \"%s\"", name->data);
994             return NGX_ERROR;
995         }
996
997         ecdh = EC_KEY_new_by_curve_name(nid);
998         if (ecdh == NULL) {
999             ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
1000                        "Unable to create curve \"%s\"", name->data);
1001             return NGX_ERROR;
1002         }
1003
1004         SSL_CTX_set_options(ssl->ctx, SSL_OP_SINGLE_ECDH_USE);
1005
1006         SSL_CTX_set_tmp_ecdh(ssl->ctx, ecdh);
1007
1008         EC_KEY_free(ecdh);
1009     #endif
1010 #endif
1011
1012     return NGX_OK;
1013 }
1014
1015
1016 ngx_int_t
1017 ngx_ssl_create_connection(ngx_ssl_t *ssl, ngx_connection_t *c, ngx_uint_t flags)
1018 {
1019     ngx_ssl_connection_t *sc;
1020
1021     sc = ngx_palloc(c->pool, sizeof(ngx_ssl_connection_t));
1022     if (sc == NULL) {
1023         return NGX_ERROR;
1024     }
1025

```

```

1026     sc->buffer = ((flags & NGX\_SSL\_BUFFER) != 0);
1027     sc->buffer_size = ssl->buffer_size;
1028
1029     sc->connection = SSL_new(ssl->ctx);
1030
1031     if (sc->connection == NULL) {
1032         ngx\_ssl\_error(NGX\_LOG\_ALERT, c->log, 0, "SSL_new() failed");
1033         return NGX\_ERROR;
1034     }
1035
1036     if (SSL_set_fd(sc->connection, c->fd) == 0) {
1037         ngx\_ssl\_error(NGX\_LOG\_ALERT, c->log, 0, "SSL_set_fd() failed");
1038         return NGX\_ERROR;
1039     }
1040
1041     if (flags & NGX\_SSL\_CLIENT) {
1042         SSL_set_connect_state(sc->connection);
1043     }
1044     else {
1045         SSL_set_accept_state(sc->connection);
1046     }
1047
1048     if (SSL_set_ex_data(sc->connection, ngx\_ssl\_connection\_index, c) == 0) {
1049         ngx\_ssl\_error(NGX\_LOG\_ALERT, c->log, 0, "SSL_set_ex_data() failed");
1050         return NGX\_ERROR;
1051     }
1052
1053     c->ssl = sc;
1054
1055     return NGX\_OK;
1056 }
1057
1058
1059 ngx\_int\_t
1060 ngx\_ssl\_set\_session(ngx\_connection\_t *c, ngx\_ssl\_session\_t *session)
1061 {
1062     if (session) {
1063         if (SSL_set_session(c->ssl->connection, session) == 0) {
1064             ngx\_ssl\_error(NGX\_LOG\_ALERT, c->log, 0, "SSL_set_session() failed");
1065             return NGX\_ERROR;
1066         }
1067     }
1068
1069     return NGX\_OK;
1070 }
1071
1072
1073 ngx\_int\_t
1074 ngx\_ssl\_handshake(ngx\_connection\_t *c)
1075 {
1076     int          n, sslerr;
1077     ngx\_err\_t   err;
1078
1079     ngx\_ssl\_clear\_error(c->log);
1080
1081     n = SSL_do_handshake(c->ssl->connection);
1082
1083     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_EVENT, c->log, 0, "SSL_do_handshake: %d", n);
1084
1085     if (n == 1) {
1086
1087         if (ngx\_handle\_read\_event(c->read, 0) != NGX\_OK) {
1088             return NGX\_ERROR;
1089         }
1090
1091         if (ngx\_handle\_write\_event(c->write, 0) != NGX\_OK) {
1092             return NGX\_ERROR;
1093         }
1094
1095     #if (NGX\_DEBUG)
1096     {
1097         char          buf[129], *s, *d;
1098     #if OPENSSL\_VERSION\_NUMBER >= 0x10000000L
1099         const
1100     #endif
1101         SSL\_CIPHER *cipher;

```

```

1102 cipher = SSL_get_current_cipher(c->ssl->connection);
1103
1104
1105 if (cipher) {
1106     SSL_CIPHER_description(cipher, &buf[1], 128);
1107
1108     for (s = &buf[1], d = buf; *s; s++) {
1109         if (*s == ' ' && *d == ' ') {
1110             continue;
1111         }
1112
1113         if (*s == LF || *s == CR) {
1114             continue;
1115         }
1116
1117         *++d = *s;
1118     }
1119
1120     if (*d != ' ') {
1121         d++;
1122     }
1123
1124     *d = '\\0';
1125
1126     ngx_log_debug2(NGX_LOG_DEBUG_EVENT, c->log, 0,
1127                 "SSL: %s, cipher: \"%s\"",
1128                 SSL_get_version(c->ssl->connection), &buf[1]);
1129
1130     if (SSL_session_reused(c->ssl->connection)) {
1131         ngx_log_debug0(NGX_LOG_DEBUG_EVENT, c->log, 0,
1132                     "SSL reused session");
1133     }
1134
1135 } else {
1136     ngx_log_debug0(NGX_LOG_DEBUG_EVENT, c->log, 0,
1137                 "SSL no shared ciphers");
1138 }
1139 }
1140 #endif
1141
1142 c->ssl->handshaked = 1;
1143
1144 c->recv = ngx_ssl_recv;
1145 c->send = ngx_ssl_write;
1146 c->recv_chain = ngx_ssl_recv_chain;
1147 c->send_chain = ngx_ssl_send_chain;
1148
1149 #ifndef SSL3_FLAGS_NO_RENEGOTIATE_CIPHERS
1150
1151     /* initial handshake done, disable renegotiation (CVE-2009-3555) */
1152     if (c->ssl->connection->s3) {
1153         c->ssl->connection->s3->flags |= SSL3_FLAGS_NO_RENEGOTIATE_CIPHERS;
1154     }
1155
1156 #endif
1157
1158     return NGX_OK;
1159 }
1160
1161 sslerr = SSL_get_error(c->ssl->connection, n);
1162
1163 ngx_log_debug1(NGX_LOG_DEBUG_EVENT, c->log, 0, "SSL_get_error: %d", sslerr);
1164
1165 if (sslerr == SSL_ERROR_WANT_READ) {
1166     c->read->ready = 0;
1167     c->read->handler = ngx_ssl_handshake_handler;
1168     c->write->handler = ngx_ssl_handshake_handler;
1169
1170     if (ngx_handle_read_event(c->read, 0) != NGX_OK) {
1171         return NGX_ERROR;
1172     }
1173
1174     if (ngx_handle_write_event(c->write, 0) != NGX_OK) {
1175         return NGX_ERROR;
1176     }
1177

```

```

1178     return NGX\_AGAIN;
1179 }
1180
1181 if (sslerr == SSL_ERROR_WANT_WRITE) {
1182     c->write->ready = 0;
1183     c->read->handler = ngx\_ssl\_handshake\_handler;
1184     c->write->handler = ngx\_ssl\_handshake\_handler;
1185
1186     if (ngx\_handle\_read\_event(c->read, 0) != NGX\_OK) {
1187         return NGX\_ERROR;
1188     }
1189
1190     if (ngx\_handle\_write\_event(c->write, 0) != NGX\_OK) {
1191         return NGX\_ERROR;
1192     }
1193
1194     return NGX\_AGAIN;
1195 }
1196
1197 err = (sslerr == SSL_ERROR_SYSCALL) ? ngx\_errno : 0;
1198
1199 c->ssl->no_wait_shutdown = 1;
1200 c->ssl->no_send_shutdown = 1;
1201 c->read->eof = 1;
1202
1203 if (sslerr == SSL_ERROR_ZERO_RETURN || ERR_peek_error() == 0) {
1204     ngx\_connection\_error(c, err,
1205         "peer closed connection in SSL handshake");
1206
1207     return NGX\_ERROR;
1208 }
1209
1210 c->read->error = 1;
1211
1212 ngx\_ssl\_connection\_error(c, sslerr, err, "SSL_do_handshake() failed");
1213
1214 return NGX\_ERROR;
1215 }
1216
1217
1218 static void
1219 ngx\_ssl\_handshake\_handler(ngx\_event\_t *ev)
1220 {
1221     ngx\_connection\_t *c;
1222
1223     c = ev->data;
1224
1225     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_EVENT, c->log, 0,
1226         "SSL handshake handler: %d", ev->write);
1227
1228     if (ev->timedout) {
1229         c->ssl->handler(c);
1230         return;
1231     }
1232
1233     if (ngx\_ssl\_handshake(c) == NGX\_AGAIN) {
1234         return;
1235     }
1236
1237     c->ssl->handler(c);
1238 }
1239
1240
1241 ssize\_t
1242 ngx\_ssl\_recv\_chain(ngx\_connection\_t *c, ngx\_chain\_t *cl, off\_t limit)
1243 {
1244     u\_char *last;
1245     ssize\_t n, bytes, size;
1246     ngx\_buf\_t *b;
1247
1248     bytes = 0;
1249
1250     b = cl->buf;
1251     last = b->last;
1252
1253     for ( ;; ) {

```

```

1254     size = b->end - last;
1255
1256     if (limit) {
1257         if (bytes >= limit) {
1258             return bytes;
1259         }
1260
1261         if (bytes + size > limit) {
1262             size = (ssize_t) (limit - bytes);
1263         }
1264     }
1265
1266     n = ngx_ssl_recv(c, last, size);
1267
1268     if (n > 0) {
1269         last += n;
1270         bytes += n;
1271
1272         if (last == b->end) {
1273             cl = cl->next;
1274
1275             if (cl == NULL) {
1276                 return bytes;
1277             }
1278
1279             b = cl->buf;
1280             last = b->last;
1281         }
1282
1283         continue;
1284     }
1285
1286     if (bytes) {
1287
1288         if (n == 0 || n == NGX_ERROR) {
1289             c->read->ready = 1;
1290         }
1291
1292         return bytes;
1293     }
1294
1295     return n;
1296 }
1297 }
1298
1299
1300 ssize_t
1301 ngx_ssl_recv(ngx_connection_t *c, u_char *buf, size_t size)
1302 {
1303     int n, bytes;
1304
1305     if (c->ssl->last == NGX_ERROR) {
1306         c->read->error = 1;
1307         return NGX_ERROR;
1308     }
1309
1310     if (c->ssl->last == NGX_DONE) {
1311         c->read->ready = 0;
1312         c->read->eof = 1;
1313         return 0;
1314     }
1315
1316     bytes = 0;
1317
1318     ngx_ssl_clear_error(c->log);
1319
1320     /*
1321     * SSL_read() may return data in parts, so try to read
1322     * until SSL_read() would return no data
1323     */
1324
1325     for ( ;; ) {
1326
1327         n = SSL_read(c->ssl->connection, buf, size);
1328
1329         ngx_log_debug1(NGX_LOG_DEBUG_EVENT, c->log, 0, "SSL_read: %d", n);

```

```

1330     if (n > 0) {
1331         bytes += n;
1332     }
1333
1334     c->ssl->last = ngx_ssl_handle_recv(c, n);
1335
1336     if (c->ssl->last == NGX_OK) {
1337
1338         size -= n;
1339
1340         if (size == 0) {
1341             c->read->ready = 1;
1342             return bytes;
1343         }
1344
1345         buf += n;
1346
1347         continue;
1348     }
1349
1350     if (bytes) {
1351         if (c->ssl->last != NGX_AGAIN) {
1352             c->read->ready = 1;
1353         }
1354
1355         return bytes;
1356     }
1357
1358     switch (c->ssl->last) {
1359     case NGX_DONE:
1360         c->read->ready = 0;
1361         c->read->eof = 1;
1362         return 0;
1363
1364     case NGX_ERROR:
1365         c->read->error = 1;
1366
1367         /* fall through */
1368
1369     case NGX_AGAIN:
1370         return c->ssl->last;
1371     }
1372 }
1373
1374
1375
1376
1377
1378 static ngx_int_t
1379 ngx_ssl_handle_recv(ngx_connection_t *c, int n)
1380 {
1381     int         sslerr;
1382     ngx_err_t  err;
1383
1384     if (c->ssl->renegotiation) {
1385         /*
1386          * disable renegotiation (CVE-2009-3555):
1387          * OpenSSL (at least up to 0.9.8l) does not handle disabled
1388          * renegotiation gracefully, so drop connection here
1389          */
1390
1391         ngx_log_error(NGX_LOG_NOTICE, c->log, 0, "SSL renegotiation disabled");
1392
1393         while (ERR_peek_error()) {
1394             ngx_ssl_error(NGX_LOG_DEBUG, c->log, 0,
1395                 "ignoring stale global SSL error");
1396         }
1397
1398         ERR_clear_error();
1399
1400         c->ssl->no_wait_shutdown = 1;
1401         c->ssl->no_send_shutdown = 1;
1402
1403         return NGX_ERROR;
1404     }
1405

```

```

1406     if (n > 0) {
1407
1408         if (c->ssl->saved_write_handler) {
1409
1410             c->write->handler = c->ssl->saved_write_handler;
1411             c->ssl->saved_write_handler = NULL;
1412             c->write->ready = 1;
1413
1414             if (ngx_handle_write_event(c->write, 0) != NGX_OK) {
1415                 return NGX_ERROR;
1416             }
1417
1418             ngx_post_event(c->write, &ngx_posted_events);
1419         }
1420
1421         return NGX_OK;
1422     }
1423
1424     sslerr = SSL_get_error(c->ssl->connection, n);
1425
1426     err = (sslerr == SSL_ERROR_SYSCALL) ? ngx_errno : 0;
1427
1428     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, c->log, 0, "SSL_get_error: %d", sslerr);
1429
1430     if (sslerr == SSL_ERROR_WANT_READ) {
1431         c->read->ready = 0;
1432         return NGX_AGAIN;
1433     }
1434
1435     if (sslerr == SSL_ERROR_WANT_WRITE) {
1436
1437         ngx_log_error(NGX_LOG_INFO, c->log, 0,
1438             "peer started SSL renegotiation");
1439
1440         c->write->ready = 0;
1441
1442         if (ngx_handle_write_event(c->write, 0) != NGX_OK) {
1443             return NGX_ERROR;
1444         }
1445
1446         /*
1447          * we do not set the timer because there is already the read event timer
1448          */
1449
1450         if (c->ssl->saved_write_handler == NULL) {
1451             c->ssl->saved_write_handler = c->write->handler;
1452             c->write->handler = ngx_ssl_write_handler;
1453         }
1454
1455         return NGX_AGAIN;
1456     }
1457
1458     c->ssl->no_wait_shutdown = 1;
1459     c->ssl->no_send_shutdown = 1;
1460
1461     if (sslerr == SSL_ERROR_ZERO_RETURN || ERR_peek_error() == 0) {
1462         ngx_log_debug0(NGX_LOG_DEBUG_EVENT, c->log, 0,
1463             "peer shutdown SSL cleanly");
1464         return NGX_DONE;
1465     }
1466
1467     ngx_ssl_connection_error(c, sslerr, err, "SSL_read() failed");
1468
1469     return NGX_ERROR;
1470 }
1471
1472
1473 static void
1474 ngx_ssl_write_handler(ngx_event_t *wev)
1475 {
1476     ngx_connection_t *c;
1477
1478     c = wev->data;
1479
1480     c->read->handler(c->read);
1481 }

```



```

1482
1483
1484 /*
1485  * OpenSSL has no SSL_wrotev() so we copy several bufs into our 16K buffer
1486  * before the SSL_write() call to decrease a SSL overhead.
1487  *
1488  * Besides for protocols such as HTTP it is possible to always buffer
1489  * the output to decrease a SSL overhead some more.
1490  */
1491
1492 ngx_chain_t *
1493 ngx_ssl_send_chain(ngx_connection_t *c, ngx_chain_t *in, off_t limit)
1494 {
1495     int n;
1496     ngx_uint_t flush;
1497     ssize_t send, size;
1498     ngx_buf_t *buf;
1499
1500     if (!c->ssl->buffer) {
1501
1502         while (in) {
1503             if (ngx_buf_special(in->buf)) {
1504                 in = in->next;
1505                 continue;
1506             }
1507
1508             n = ngx_ssl_write(c, in->buf->pos, in->buf->last - in->buf->pos);
1509
1510             if (n == NGX_ERROR) {
1511                 return NGX_CHAIN_ERROR;
1512             }
1513
1514             if (n == NGX_AGAIN) {
1515                 return in;
1516             }
1517
1518             in->buf->pos += n;
1519             c->sent += n;
1520
1521             if (in->buf->pos == in->buf->last) {
1522                 in = in->next;
1523             }
1524         }
1525
1526         return in;
1527     }
1528
1529
1530     /* the maximum limit size is the maximum int32_t value - the page size */
1531
1532     if (limit == 0 || limit > (off_t) (NGX_MAX_INT32_VALUE - ngx_pagesize)) {
1533         limit = NGX_MAX_INT32_VALUE - ngx_pagesize;
1534     }
1535
1536     buf = c->ssl->buf;
1537
1538     if (buf == NULL) {
1539         buf = ngx_create_temp_buf(c->pool, c->ssl->buffer_size);
1540         if (buf == NULL) {
1541             return NGX_CHAIN_ERROR;
1542         }
1543
1544         c->ssl->buf = buf;
1545     }
1546
1547     if (buf->start == NULL) {
1548         buf->start = ngx_palloc(c->pool, c->ssl->buffer_size);
1549         if (buf->start == NULL) {
1550             return NGX_CHAIN_ERROR;
1551         }
1552
1553         buf->pos = buf->start;
1554         buf->last = buf->start;
1555         buf->end = buf->start + c->ssl->buffer_size;
1556     }
1557

```

```

1558 send = buf->last - buf->pos;
1559 flush = (in == NULL) ? 1 : buf->flush;
1560
1561 for ( ;; ) {
1562
1563     while (in && buf->last < buf->end && send < limit) {
1564         if (in->buf->last_buf || in->buf->flush) {
1565             flush = 1;
1566         }
1567
1568         if (ngx_buf_special(in->buf)) {
1569             in = in->next;
1570             continue;
1571         }
1572
1573         size = in->buf->last - in->buf->pos;
1574
1575         if (size > buf->end - buf->last) {
1576             size = buf->end - buf->last;
1577         }
1578
1579         if (send + size > limit) {
1580             size = (ssize_t) (limit - send);
1581         }
1582
1583         ngx_log_debug1(NGX_LOG_DEBUG_EVENT, c->log, 0,
1584             "SSL buf copy: %d", size);
1585
1586         ngx_memcpy(buf->last, in->buf->pos, size);
1587
1588         buf->last += size;
1589         in->buf->pos += size;
1590         send += size;
1591
1592         if (in->buf->pos == in->buf->last) {
1593             in = in->next;
1594         }
1595     }
1596
1597     if (!flush && send < limit && buf->last < buf->end) {
1598         break;
1599     }
1600
1601     size = buf->last - buf->pos;
1602
1603     if (size == 0) {
1604         buf->flush = 0;
1605         c->buffered &= ~NGX_SSL_BUFFERED;
1606         return in;
1607     }
1608
1609     n = ngx_ssl_write(c, buf->pos, size);
1610
1611     if (n == NGX_ERROR) {
1612         return NGX_CHAIN_ERROR;
1613     }
1614
1615     if (n == NGX_AGAIN) {
1616         break;
1617     }
1618
1619     buf->pos += n;
1620     c->sent += n;
1621
1622     if (n < size) {
1623         break;
1624     }
1625
1626     flush = 0;
1627
1628     buf->pos = buf->start;
1629     buf->last = buf->start;
1630
1631     if (in == NULL || send == limit) {
1632         break;
1633     }

```

```

1634     }
1635
1636     buf->flush = flush;
1637
1638     if (buf->pos < buf->last) {
1639         c->buffered |= NGX\_SSL\_BUFFERED;
1640
1641     } else {
1642         c->buffered &= ~NGX\_SSL\_BUFFERED;
1643     }
1644
1645     return in;
1646 }
1647
1648
1649 ssize\_t
1650 ngx\_ssl\_write(ngx\_connection\_t *c, u\_char *data, size\_t size)
1651 {
1652     int n, sslerr;
1653     ngx\_err\_t err;
1654
1655     ngx\_ssl\_clear\_error(c->log);
1656
1657     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_EVENT, c->log, 0, "SSL to write: %d", size);
1658
1659     n = SSL_write(c->ssl->connection, data, size);
1660
1661     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_EVENT, c->log, 0, "SSL_write: %d", n);
1662
1663     if (n > 0) {
1664
1665         if (c->ssl->saved_read_handler) {
1666
1667             c->read->handler = c->ssl->saved_read_handler;
1668             c->ssl->saved_read_handler = NULL;
1669             c->read->ready = 1;
1670
1671             if (ngx\_handle\_read\_event(c->read, 0) != NGX\_OK) {
1672                 return NGX\_ERROR;
1673             }
1674
1675             ngx\_post\_event(c->read, &ngx\_posted\_events);
1676         }
1677
1678         return n;
1679     }
1680
1681     sslerr = SSL_get_error(c->ssl->connection, n);
1682
1683     err = (sslerr == SSL_ERROR_SYSCALL) ? ngx\_errno : 0;
1684
1685     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_EVENT, c->log, 0, "SSL_get_error: %d", sslerr);
1686
1687     if (sslerr == SSL_ERROR_WANT_WRITE) {
1688         c->write->ready = 0;
1689         return NGX\_AGAIN;
1690     }
1691
1692     if (sslerr == SSL_ERROR_WANT_READ) {
1693
1694         ngx\_log\_error(NGX\_LOG\_INFO, c->log, 0,
1695             "peer started SSL renegotiation");
1696
1697         c->read->ready = 0;
1698
1699         if (ngx\_handle\_read\_event(c->read, 0) != NGX\_OK) {
1700             return NGX\_ERROR;
1701         }
1702
1703         /*
1704          * we do not set the timer because there is already
1705          * the write event timer
1706          */
1707
1708         if (c->ssl->saved_read_handler == NULL) {
1709             c->ssl->saved_read_handler = c->read->handler;

```

```

1710     c->read->handler = ngx_ssl_read_handler;
1711 }
1712
1713     return NGX_AGAIN;
1714 }
1715
1716 c->ssl->no_wait_shutdown = 1;
1717 c->ssl->no_send_shutdown = 1;
1718 c->write->error = 1;
1719
1720 ngx_ssl_connection_error(c, sslerr, err, "SSL_write() failed");
1721
1722 return NGX_ERROR;
1723 }
1724
1725
1726 static void
1727 ngx_ssl_read_handler(ngx_event_t *rev)
1728 {
1729     ngx_connection_t *c;
1730
1731     c = rev->data;
1732
1733     c->write->handler(c->write);
1734 }
1735
1736
1737 void
1738 ngx_ssl_free_buffer(ngx_connection_t *c)
1739 {
1740     if (c->ssl->buf && c->ssl->buf->start) {
1741         if (ngx_pfree(c->pool, c->ssl->buf->start) == NGX_OK) {
1742             c->ssl->buf->start = NULL;
1743         }
1744     }
1745 }
1746
1747
1748 ngx_int_t
1749 ngx_ssl_shutdown(ngx_connection_t *c)
1750 {
1751     int n, sslerr, mode;
1752     ngx_err_t err;
1753
1754     if (c->timedout) {
1755         mode = SSL_RECEIVED_SHUTDOWN|SSL_SENT_SHUTDOWN;
1756         SSL_set_quiet_shutdown(c->ssl->connection, 1);
1757
1758     } else {
1759         mode = SSL_get_shutdown(c->ssl->connection);
1760
1761         if (c->ssl->no_wait_shutdown) {
1762             mode |= SSL_RECEIVED_SHUTDOWN;
1763         }
1764
1765         if (c->ssl->no_send_shutdown) {
1766             mode |= SSL_SENT_SHUTDOWN;
1767         }
1768
1769         if (c->ssl->no_wait_shutdown && c->ssl->no_send_shutdown) {
1770             SSL_set_quiet_shutdown(c->ssl->connection, 1);
1771         }
1772     }
1773
1774     SSL_set_shutdown(c->ssl->connection, mode);
1775
1776     ngx_ssl_clear_error(c->log);
1777
1778     n = SSL_shutdown(c->ssl->connection);
1779
1780     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, c->log, 0, "SSL_shutdown: %d", n);
1781
1782     sslerr = 0;
1783
1784     /* SSL_shutdown() never returns -1, on error it returns 0 */
1785

```

```

1786     if (n != 1 && ERR_peek_error()) {
1787         sslerr = SSL_get_error(c->ssl->connection, n);
1788
1789         ngx\_log\_debug1(NGX\_LOG\_DEBUG\_EVENT, c->log, 0,
1790                 "SSL_get_error: %d", sslerr);
1791     }
1792
1793     if (n == 1 || sslerr == 0 || sslerr == SSL_ERROR_ZERO_RETURN) {
1794         SSL_free(c->ssl->connection);
1795         c->ssl = NULL;
1796
1797         return NGX\_OK;
1798     }
1799
1800     if (sslerr == SSL_ERROR_WANT_READ || sslerr == SSL_ERROR_WANT_WRITE) {
1801         c->read->handler = ngx\_ssl\_shutdown\_handler;
1802         c->write->handler = ngx\_ssl\_shutdown\_handler;
1803
1804         if (ngx\_handle\_read\_event(c->read, 0) != NGX\_OK) {
1805             return NGX\_ERROR;
1806         }
1807
1808         if (ngx\_handle\_write\_event(c->write, 0) != NGX\_OK) {
1809             return NGX\_ERROR;
1810         }
1811
1812         if (sslerr == SSL_ERROR_WANT_READ) {
1813             ngx\_add\_timer(c->read, 30000);
1814         }
1815
1816         return NGX\_AGAIN;
1817     }
1818
1819     err = (sslerr == SSL_ERROR_SYSCALL) ? ngx\_errno : 0;
1820
1821     ngx\_ssl\_connection\_error(c, sslerr, err, "SSL_shutdown() failed");
1822
1823     SSL_free(c->ssl->connection);
1824     c->ssl = NULL;
1825
1826     return NGX\_ERROR;
1827 }
1828
1829
1830 static void
1831 ngx\_ssl\_shutdown\_handler(ngx\_event\_t *ev)
1832 {
1833     ngx\_connection\_t *c;
1834     ngx\_connection\_handler\_pt handler;
1835
1836     c = ev->data;
1837     handler = c->ssl->handler;
1838
1839     if (ev->timedout) {
1840         c->timedout = 1;
1841     }
1842
1843     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_EVENT, ev->log, 0, "SSL shutdown handler");
1844
1845     if (ngx\_ssl\_shutdown(c) == NGX\_AGAIN) {
1846         return;
1847     }
1848
1849     handler(c);
1850 }
1851
1852
1853 static void
1854 ngx\_ssl\_connection\_error(ngx\_connection\_t *c, int sslerr, ngx\_err\_t err,
1855                          char *text)
1856 {
1857     int n;
1858     ngx\_uint\_t level;
1859
1860     level = NGX\_LOG\_CRIT;
1861

```

```

1862     if (sslerr == SSL_ERROR_SYSCALL) {
1863
1864         if (err == NGX\_ECONNRESET
1865             || err == NGX\_EPIPE
1866             || err == NGX\_ENOTCONN
1867             || err == NGX\_ETIMEDOUT
1868             || err == NGX\_ECONNREFUSED
1869             || err == NGX\_ENETDOWN
1870             || err == NGX\_ENETUNREACH
1871             || err == NGX\_EHOSTDOWN
1872             || err == NGX\_EHOSTUNREACH)
1873         {
1874             switch (c->log_error) {
1875
1876                 case NGX_ERROR_IGNORE_ECONNRESET:
1877                 case NGX_ERROR_INFO:
1878                     level = NGX\_LOG\_INFO;
1879                     break;
1880
1881                 case NGX_ERROR_ERR:
1882                     level = NGX\_LOG\_ERR;
1883                     break;
1884
1885                 default:
1886                     break;
1887             }
1888         }
1889
1890     } else if (sslerr == SSL_ERROR_SSL) {
1891
1892         n = ERR_GET_REASON(ERR_peek_error());
1893
1894         /* handshake failures */
1895         if (n == SSL_R_BAD_CHANGE_CIPHER_SPEC /* 103 */
1896             || n == SSL_R_BLOCK_CIPHER_PAD_IS_WRONG /* 129 */
1897             || n == SSL_R_DIGEST_CHECK_FAILED /* 149 */
1898             || n == SSL_R_ERROR_IN_RECEIVED_CIPHER_LIST /* 151 */
1899             || n == SSL_R_EXCESSIVE_MESSAGE_SIZE /* 152 */
1900             || n == SSL_R_LENGTH_MISMATCH /* 159 */
1901             || n == SSL_R_NO_CIPHERS_PASSED /* 182 */
1902             || n == SSL_R_NO_CIPHERS_SPECIFIED /* 183 */
1903             || n == SSL_R_NO_COMPRESSION_SPECIFIED /* 187 */
1904             || n == SSL_R_NO_SHARED_CIPHER /* 193 */
1905             || n == SSL_R_RECORD_LENGTH_MISMATCH /* 213 */
1906         #ifdef SSL_R_PARSE_TLSEXT
1907             || n == SSL_R_PARSE_TLSEXT /* 227 */
1908         #endif
1909             || n == SSL_R_UNEXPECTED_MESSAGE /* 244 */
1910             || n == SSL_R_UNEXPECTED_RECORD /* 245 */
1911             || n == SSL_R_UNKNOWN_ALERT_TYPE /* 246 */
1912             || n == SSL_R_UNKNOWN_PROTOCOL /* 252 */
1913             || n == SSL_R_WRONG_VERSION_NUMBER /* 267 */
1914             || n == SSL_R_DECRYPTION_FAILED_OR_BAD_RECORD_MAC /* 281 */
1915         #ifdef SSL_R_RENEGOTIATE_EXT_TOO_LONG
1916             || n == SSL_R_RENEGOTIATE_EXT_TOO_LONG /* 335 */
1917             || n == SSL_R_RENEGOTIATION_ENCODING_ERR /* 336 */
1918             || n == SSL_R_RENEGOTIATION_MISMATCH /* 337 */
1919         #endif
1920         #ifdef SSL_R_UNSAFE_LEGACY_RENEGOTIATION_DISABLED
1921             || n == SSL_R_UNSAFE_LEGACY_RENEGOTIATION_DISABLED /* 338 */
1922         #endif
1923         #ifdef SSL_R_SCSV_RECEIVED_WHEN_RENEGOTIATING
1924             || n == SSL_R_SCSV_RECEIVED_WHEN_RENEGOTIATING /* 345 */
1925         #endif
1926         #ifdef SSL_R_INAPPROPRIATE_FALLBACK
1927             || n == SSL_R_INAPPROPRIATE_FALLBACK /* 373 */
1928         #endif
1929             || n == 1000 /* SSL_R_SSLV3_ALERT_CLOSE_NOTIFY */
1930             || n == SSL_R_SSLV3_ALERT_UNEXPECTED_MESSAGE /* 1010 */
1931             || n == SSL_R_SSLV3_ALERT_BAD_RECORD_MAC /* 1020 */
1932             || n == SSL_R_TLSV1_ALERT_DECRYPTION_FAILED /* 1021 */
1933             || n == SSL_R_TLSV1_ALERT_RECORD_OVERFLOW /* 1022 */
1934             || n == SSL_R_SSLV3_ALERT_DECOMPRESSION_FAILURE /* 1030 */
1935             || n == SSL_R_SSLV3_ALERT_HANDSHAKE_FAILURE /* 1040 */
1936             || n == SSL_R_SSLV3_ALERT_NO_CERTIFICATE /* 1041 */
1937             || n == SSL_R_SSLV3_ALERT_BAD_CERTIFICATE /* 1042 */

```

```

1938     || n == SSL_R_SSLV3_ALERT_UNSUPPORTED_CERTIFICATE      /* 1043 */
1939     || n == SSL_R_SSLV3_ALERT_CERTIFICATE_REVOKED          /* 1044 */
1940     || n == SSL_R_SSLV3_ALERT_CERTIFICATE_EXPIRED          /* 1045 */
1941     || n == SSL_R_SSLV3_ALERT_CERTIFICATE_UNKNOWN          /* 1046 */
1942     || n == SSL_R_SSLV3_ALERT_ILLEGAL_PARAMETER            /* 1047 */
1943     || n == SSL_R_TLSV1_ALERT_UNKNOWN_CA                    /* 1048 */
1944     || n == SSL_R_TLSV1_ALERT_ACCESS_DENIED                 /* 1049 */
1945     || n == SSL_R_TLSV1_ALERT_DECODE_ERROR                 /* 1050 */
1946     || n == SSL_R_TLSV1_ALERT_DECRYPT_ERROR                 /* 1051 */
1947     || n == SSL_R_TLSV1_ALERT_EXPORT_RESTRICTION           /* 1060 */
1948     || n == SSL_R_TLSV1_ALERT_PROTOCOL_VERSION             /* 1070 */
1949     || n == SSL_R_TLSV1_ALERT_INSUFFICIENT_SECURITY        /* 1071 */
1950     || n == SSL_R_TLSV1_ALERT_INTERNAL_ERROR               /* 1080 */
1951     || n == SSL_R_TLSV1_ALERT_USER_CANCELLED               /* 1090 */
1952     || n == SSL_R_TLSV1_ALERT_NO_RENEGOTIATION             /* 1100 */
1953 {
1954     switch (c->log_error) {
1955
1956     case NGX_ERROR_IGNORE_ECONNRESET:
1957     case NGX_ERROR_INFO:
1958         level = NGX\_LOG\_INFO;
1959         break;
1960
1961     case NGX_ERROR_ERR:
1962         level = NGX\_LOG\_ERR;
1963         break;
1964
1965     default:
1966         break;
1967     }
1968 }
1969 }
1970
1971 ngx\_ssl\_error(level, c->log, err, text);
1972 }
1973
1974
1975 static void
1976 ngx\_ssl\_clear\_error(ngx\_log\_t *log)
1977 {
1978     while (ERR_peek_error()) {
1979         ngx\_ssl\_error(NGX\_LOG\_ALERT, log, 0, "ignoring stale global SSL error");
1980     }
1981
1982     ERR_clear_error();
1983 }
1984
1985
1986 void ngx\_cdecl
1987 ngx\_ssl\_error(ngx\_uint\_t level, ngx\_log\_t *log, ngx\_err\_t err, char *fmt, ...)
1988 {
1989     int         flags;
1990     u_long      n;
1991     va_list     args;
1992     u_char      *p, *last;
1993     u_char      errstr[NGX\_MAX\_CONF\_ERRSTR];
1994     const char  *data;
1995
1996     last = errstr + NGX\_MAX\_CONF\_ERRSTR;
1997
1998     va_start(args, fmt);
1999     p = ngx\_vslprintf(errstr, last - 1, fmt, args);
2000     va_end(args);
2001
2002     p = ngx\_cpystn(p, (u_char *) " (SSL:", last - p);
2003
2004     for ( ;; ) {
2005
2006         n = ERR_peek_error_line_data(NULL, NULL, &data, &flags);
2007
2008         if (n == 0) {
2009             break;
2010         }
2011
2012         if (p >= last) {
2013             goto next;

```

```

2014     }
2015
2016     *p++ = ' ';
2017
2018     ERR_error_string_n(n, (char *) p, last - p);
2019
2020     while (p < last && *p) {
2021         p++;
2022     }
2023
2024     if (p < last && *data && (flags & ERR_TXT_STRING)) {
2025         *p++ = ':';
2026         p = ngx_cpystn(p, (u_char *) data, last - p);
2027     }
2028
2029     next:
2030
2031     (void) ERR_get_error();
2032 }
2033
2034 ngx_log_error(level, log, err, "%s)", errstr);
2035 }
2036
2037
2038 ngx_int_t
2039 ngx_ssl_session_cache(ngx_ssl_t *ssl, ngx_str_t *sess_ctx,
2040     ssize_t builtin_session_cache, ngx_shm_zone_t *shm_zone, time_t timeout)
2041 {
2042     long cache_mode;
2043
2044     SSL_CTX_set_timeout(ssl->ctx, (long) timeout);
2045
2046     if (ngx_ssl_session_id_context(ssl, sess_ctx) != NGX_OK) {
2047         return NGX_ERROR;
2048     }
2049
2050     if (builtin_session_cache == NGX_SSL_NO_SCACHE) {
2051         SSL_CTX_set_session_cache_mode(ssl->ctx, SSL_SESS_CACHE_OFF);
2052         return NGX_OK;
2053     }
2054
2055     if (builtin_session_cache == NGX_SSL_NONE_SCACHE) {
2056
2057         /*
2058          * If the server explicitly says that it does not support
2059          * session reuse (see SSL_SESS_CACHE_OFF above), then
2060          * Outlook Express fails to upload a sent email to
2061          * the Sent Items folder on the IMAP server via a separate IMAP
2062          * connection in the background. Therefore we have a special
2063          * mode (SSL_SESS_CACHE_SERVER|SSL_SESS_CACHE_NO_INTERNAL_STORE)
2064          * where the server pretends that it supports session reuse,
2065          * but it does not actually store any session.
2066          */
2067
2068         SSL_CTX_set_session_cache_mode(ssl->ctx,
2069             SSL_SESS_CACHE_SERVER
2070             |SSL_SESS_CACHE_NO_AUTO_CLEAR
2071             |SSL_SESS_CACHE_NO_INTERNAL_STORE);
2072
2073         SSL_CTX_sess_set_cache_size(ssl->ctx, 1);
2074
2075         return NGX_OK;
2076     }
2077
2078     cache_mode = SSL_SESS_CACHE_SERVER;
2079
2080     if (shm_zone && builtin_session_cache == NGX_SSL_NO_BUILTIN_SCACHE) {
2081         cache_mode |= SSL_SESS_CACHE_NO_INTERNAL;
2082     }
2083
2084     SSL_CTX_set_session_cache_mode(ssl->ctx, cache_mode);
2085
2086     if (builtin_session_cache != NGX_SSL_NO_BUILTIN_SCACHE) {
2087
2088         if (builtin_session_cache != NGX_SSL_DFLT_BUILTIN_SCACHE) {
2089             SSL_CTX_sess_set_cache_size(ssl->ctx, builtin_session_cache);

```



```

2090     }
2091 }
2092
2093 if (shm_zone) {
2094     SSL_CTX_sess_set_new_cb(ssl->ctx, ngx\_ssl\_new\_session);
2095     SSL_CTX_sess_set_get_cb(ssl->ctx, ngx\_ssl\_get\_cached\_session);
2096     SSL_CTX_sess_set_remove_cb(ssl->ctx, ngx\_ssl\_remove\_session);
2097
2098     if (SSL_CTX_set_ex_data(ssl->ctx, ngx\_ssl\_session\_cache\_index, shm_zone)
2099         == 0)
2100     {
2101         ngx\_ssl\_error(NGX\_LOG\_EMERG, ssl->log, 0,
2102             "SSL_CTX_set_ex_data() failed");
2103         return NGX\_ERROR;
2104     }
2105 }
2106
2107 return NGX\_OK;
2108 }
2109
2110
2111 static ngx\_int\_t
2112 ngx\_ssl\_session\_id\_context(ngx\_ssl\_t *ssl, ngx\_str\_t *sess_ctx)
2113 {
2114     int                n, i;
2115     X509               *cert;
2116     X509_NAME         *name;
2117     EVP_MD_CTX        md;
2118     unsigned int       len;
2119     STACK_OF(X509_NAME) *list;
2120     u_char             buf[EVP_MAX_MD_SIZE];
2121
2122     /*
2123      * Session ID context is set based on the string provided,
2124      * the server certificate, and the client CA list.
2125      */
2126
2127     EVP_MD_CTX_init(&md);
2128
2129     if (EVP_DigestInit_ex(&md, EVP_sha1(), NULL) == 0) {
2130         ngx\_ssl\_error(NGX\_LOG\_EMERG, ssl->log, 0,
2131             "EVP_DigestInit_ex() failed");
2132         goto failed;
2133     }
2134
2135     if (EVP_DigestUpdate(&md, sess_ctx->data, sess_ctx->len) == 0) {
2136         ngx\_ssl\_error(NGX\_LOG\_EMERG, ssl->log, 0,
2137             "EVP_DigestUpdate() failed");
2138         goto failed;
2139     }
2140
2141     cert = SSL_CTX_get_ex_data(ssl->ctx, ngx\_ssl\_certificate\_index);
2142
2143     if (X509_digest(cert, EVP_sha1(), buf, &len) == 0) {
2144         ngx\_ssl\_error(NGX\_LOG\_EMERG, ssl->log, 0,
2145             "X509_digest() failed");
2146         goto failed;
2147     }
2148
2149     if (EVP_DigestUpdate(&md, buf, len) == 0) {
2150         ngx\_ssl\_error(NGX\_LOG\_EMERG, ssl->log, 0,
2151             "EVP_DigestUpdate() failed");
2152         goto failed;
2153     }
2154
2155     list = SSL_CTX_get_client_CA_list(ssl->ctx);
2156
2157     if (list != NULL) {
2158         n = sk_X509_NAME_num(list);
2159
2160         for (i = 0; i < n; i++) {
2161             name = sk_X509_NAME_value(list, i);
2162
2163             if (X509_NAME_digest(name, EVP_sha1(), buf, &len) == 0) {
2164                 ngx\_ssl\_error(NGX\_LOG\_EMERG, ssl->log, 0,
2165                     "X509_NAME_digest() failed");

```

```

2166         goto failed;
2167     }
2168
2169     if (EVP_DigestUpdate(&md, buf, len) == 0) {
2170         ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
2171                     "EVP_DigestUpdate() failed");
2172         goto failed;
2173     }
2174 }
2175 }
2176
2177 if (EVP_DigestFinal_ex(&md, buf, &len) == 0) {
2178     ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
2179                 "EVP_DigestUpdate() failed");
2180     goto failed;
2181 }
2182
2183 EVP_MD_CTX_cleanup(&md);
2184
2185 if (SSL_CTX_set_session_id_context(ssl->ctx, buf, len) == 0) {
2186     ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
2187                 "SSL_CTX_set_session_id_context() failed");
2188     return NGX_ERROR;
2189 }
2190
2191 return NGX_OK;
2192
2193 failed:
2194
2195     EVP_MD_CTX_cleanup(&md);
2196
2197     return NGX_ERROR;
2198 }
2199
2200
2201 ngx_int_t
2202 ngx_ssl_session_cache_init(ngx_shm_zone_t *shm_zone, void *data)
2203 {
2204     size_t          len;
2205     ngx_slab_pool_t *shpool;
2206     ngx_ssl_session_cache_t *cache;
2207
2208     if (data) {
2209         shm_zone->data = data;
2210         return NGX_OK;
2211     }
2212
2213     shpool = (ngx_slab_pool_t *) shm_zone->shm.addr;
2214
2215     if (shm_zone->shm.exists) {
2216         shm_zone->data = shpool->data;
2217         return NGX_OK;
2218     }
2219
2220     cache = ngx_slab_alloc(shpool, sizeof(ngx_ssl_session_cache_t));
2221     if (cache == NULL) {
2222         return NGX_ERROR;
2223     }
2224
2225     shpool->data = cache;
2226     shm_zone->data = cache;
2227
2228     ngx_rbtree_init(&cache->session_rbtree, &cache->sentinel,
2229                   ngx_ssl_session_rbtree_insert_value);
2230
2231     ngx_queue_init(&cache->expire_queue);
2232
2233     len = sizeof(" in SSL session shared cache \\\"\\") + shm_zone->shm.name.len;
2234
2235     shpool->log_ctx = ngx_slab_alloc(shpool, len);
2236     if (shpool->log_ctx == NULL) {
2237         return NGX_ERROR;
2238     }
2239
2240     ngx_sprintf(shpool->log_ctx, " in SSL session shared cache \\%V\\%Z",
2241                &shm_zone->shm.name);

```

```

2242     shpool->log_nomem = 0;
2243
2244     return NGX_OK;
2245 }
2246
2247
2248
2249 /*
2250  * The length of the session id is 16 bytes for SSLv2 sessions and
2251  * between 1 and 32 bytes for SSLv3/TLSv1, typically 32 bytes.
2252  * It seems that the typical length of the external ASN1 representation
2253  * of a session is 118 or 119 bytes for SSLv3/TLSv1.
2254  *
2255  * Thus on 32-bit platforms we allocate separately an rbtree node,
2256  * a session id, and an ASN1 representation, they take accordingly
2257  * 64, 32, and 128 bytes.
2258  *
2259  * On 64-bit platforms we allocate separately an rbtree node + session_id,
2260  * and an ASN1 representation, they take accordingly 128 and 128 bytes.
2261  *
2262  * OpenSSL's i2d_SSL_SESSION() and d2i_SSL_SESSION are slow,
2263  * so they are outside the code locked by shared pool mutex
2264  */
2265
2266 static int
2267 ngx_ssl_new_session(ngx_ssl_conn_t *ssl_conn, ngx_ssl_session_t *sess)
2268 {
2269     int len;
2270     u_char *p, *id, *cached_sess, *session_id;
2271     uint32_t hash;
2272     SSL_CTX *ssl_ctx;
2273     unsigned int session_id_length;
2274     ngx_shm_zone_t *shm_zone;
2275     ngx_connection_t *c;
2276     ngx_slab_pool_t *shpool;
2277     ngx_ssl_sess_id_t *sess_id;
2278     ngx_ssl_session_cache_t *cache;
2279     u_char buf[NGX_SSL_MAX_SESSION_SIZE];
2280
2281     len = i2d_SSL_SESSION(sess, NULL);
2282
2283     /* do not cache too big session */
2284
2285     if (len > (int) NGX_SSL_MAX_SESSION_SIZE) {
2286         return 0;
2287     }
2288
2289     p = buf;
2290     i2d_SSL_SESSION(sess, &p);
2291
2292     c = ngx_ssl_get_connection(ssl_conn);
2293
2294     ssl_ctx = SSL_get_SSL_CTX(ssl_conn);
2295     shm_zone = SSL_CTX_get_ex_data(ssl_ctx, ngx_ssl_session_cache_index);
2296
2297     cache = shm_zone->data;
2298     shpool = (ngx_slab_pool_t *) shm_zone->shm.addr;
2299
2300     ngx_shmtx_lock(&shpool->mutex);
2301
2302     /* drop one or two expired sessions */
2303     ngx_ssl_expire_sessions(cache, shpool, 1);
2304
2305     cached_sess = ngx_slab_alloc_locked(shpool, len);
2306
2307     if (cached_sess == NULL) {
2308
2309         /* drop the oldest non-expired session and try once more */
2310
2311         ngx_ssl_expire_sessions(cache, shpool, 0);
2312
2313         cached_sess = ngx_slab_alloc_locked(shpool, len);
2314
2315         if (cached_sess == NULL) {
2316             sess_id = NULL;
2317             goto failed;

```

```

2318     }
2319 }
2320
2321 sess_id = ngx_slab_alloc_locked(shpool, sizeof(ngx_ssl_sess_id_t));
2322
2323 if (sess_id == NULL) {
2324     /* drop the oldest non-expired session and try once more */
2325
2326     ngx_ssl_expire_sessions(cache, shpool, 0);
2327
2328     sess_id = ngx_slab_alloc_locked(shpool, sizeof(ngx_ssl_sess_id_t));
2329
2330     if (sess_id == NULL) {
2331         goto failed;
2332     }
2333 }
2334 }
2335
2336 #if OPENSSSL_VERSION_NUMBER >= 0x0090800fL
2337     session_id = (u_char *) SSL_SESSION_get_id(sess, &session_id_length);
2338
2339 #else
2340     session_id = sess->session_id;
2341     session_id_length = sess->session_id_length;
2342
2343 #endif
2344
2345 #if (NGX_PTR_SIZE == 8)
2346     id = sess_id->sess_id;
2347
2348 #else
2349     id = ngx_slab_alloc_locked(shpool, session_id_length);
2350
2351     if (id == NULL) {
2352         /* drop the oldest non-expired session and try once more */
2353
2354         ngx_ssl_expire_sessions(cache, shpool, 0);
2355
2356         id = ngx_slab_alloc_locked(shpool, session_id_length);
2357
2358         if (id == NULL) {
2359             goto failed;
2360         }
2361     }
2362 #endif
2363
2364 ngx_memcpy(cached_sess, buf, len);
2365
2366 ngx_memcpy(id, session_id, session_id_length);
2367
2368 hash = ngx_crc32_short(session_id, session_id_length);
2369
2370 ngx_log_debug3(NGX_LOG_DEBUG_EVENT, c->log, 0,
2371     "ssl new session: %08XD:%ud:%d",
2372     hash, session_id_length, len);
2373
2374 sess_id->node.key = hash;
2375 sess_id->node.data = (u_char) session_id_length;
2376 sess_id->id = id;
2377 sess_id->len = len;
2378 sess_id->session = cached_sess;
2379
2380 sess_id->expire = ngx_time() + SSL_CTX_get_timeout(ssl_ctx);
2381
2382 ngx_queue_insert_head(&cache->expire_queue, &sess_id->queue);
2383
2384 ngx_rbtree_insert(&cache->session_rbtree, &sess_id->node);
2385
2386 ngx_shmtx_unlock(&shpool->mutex);
2387
2388
2389
2390
2391
2392
2393

```

```

2394     return 0;
2395
2396 failed:
2397
2398     if (cached_sess) {
2399         ngx_slab_free_locked(shpool, cached_sess);
2400     }
2401
2402     if (sess_id) {
2403         ngx_slab_free_locked(shpool, sess_id);
2404     }
2405
2406     ngx_shmtx_unlock(&shpool->mutex);
2407
2408     ngx_log_error(NGX_LOG_ALERT, c->log, 0,
2409         "could not allocate new session%s", shpool->log_ctx);
2410
2411     return 0;
2412 }
2413
2414
2415 static ngx_ssl_session_t *
2416 ngx_ssl_get_cached_session(ngx_ssl_conn_t *ssl_conn, u_char *id, int len,
2417     int *copy)
2418 {
2419     #if OPENSSSL_VERSION_NUMBER >= 0x0090707fL
2420     const
2421     #endif
2422     u_char                *p;
2423     uint32_t              hash;
2424     ngx_int_t             rc;
2425     ngx_shm_zone_t        *shm_zone;
2426     ngx_slab_pool_t        *shpool;
2427     ngx_rbtree_node_t     *node, *sentinel;
2428     ngx_ssl_session_t     *sess;
2429     ngx_ssl_sess_id_t     *sess_id;
2430     ngx_ssl_session_cache_t *cache;
2431     u_char                buf[NGX_SSL_MAX_SESSION_SIZE];
2432     #if (NGX_DEBUG)
2433     ngx_connection_t      *c;
2434     #endif
2435
2436     hash = ngx_crc32_short(id, (size_t) len);
2437     *copy = 0;
2438
2439     #if (NGX_DEBUG)
2440     c = ngx_ssl_get_connection(ssl_conn);
2441
2442     ngx_log_debug2(NGX_LOG_DEBUG_EVENT, c->log, 0,
2443         "ssl get session: %08XD:%d", hash, len);
2444     #endif
2445
2446     shm_zone = SSL_CTX_get_ex_data(SSL_get_SSL_CTX(ssl_conn),
2447         ngx_ssl_session_cache_index);
2448
2449     cache = shm_zone->data;
2450
2451     sess = NULL;
2452
2453     shpool = (ngx_slab_pool_t *) shm_zone->shm.addr;
2454
2455     ngx_shmtx_lock(&shpool->mutex);
2456
2457     node = cache->session_rbtree.root;
2458     sentinel = cache->session_rbtree.sentinel;
2459
2460     while (node != sentinel) {
2461
2462         if (hash < node->key) {
2463             node = node->left;
2464             continue;
2465         }
2466
2467         if (hash > node->key) {
2468             node = node->right;
2469             continue;

```

```

2470     }
2471
2472     /* hash == node->key */
2473
2474     sess_id = (ngx_ssl_sess_id_t *) node;
2475
2476     rc = ngx_memn2cmp(id, sess_id->id, (size_t) len, (size_t) node->data);
2477
2478     if (rc == 0) {
2479
2480         if (sess_id->expire > ngx_time()) {
2481             ngx_memcpy(buf, sess_id->session, sess_id->len);
2482
2483             ngx_shmtx_unlock(&shpool->mutex);
2484
2485             p = buf;
2486             sess = d2i_SSL_SESSION(NULL, &p, sess_id->len);
2487
2488             return sess;
2489         }
2490
2491         ngx_queue_remove(&sess_id->queue);
2492
2493         ngx_rbtrees_delete(&cache->session_rbtrees, node);
2494
2495         ngx_slab_free_locked(shpool, sess_id->session);
2496 #if (NGX_PTR_SIZE == 4)
2497         ngx_slab_free_locked(shpool, sess_id->id);
2498 #endif
2499         ngx_slab_free_locked(shpool, sess_id);
2500
2501         sess = NULL;
2502
2503         goto done;
2504     }
2505
2506     node = (rc < 0) ? node->left : node->right;
2507 }
2508
2509 done:
2510
2511     ngx_shmtx_unlock(&shpool->mutex);
2512
2513     return sess;
2514 }
2515
2516
2517 void
2518 ngx_ssl_remove_cached_session(SSL_CTX *ssl, ngx_ssl_session_t *sess)
2519 {
2520     SSL_CTX_remove_session(ssl, sess);
2521
2522     ngx_ssl_remove_session(ssl, sess);
2523 }
2524
2525
2526 static void
2527 ngx_ssl_remove_session(SSL_CTX *ssl, ngx_ssl_session_t *sess)
2528 {
2529     u_char                *id;
2530     uint32_t              hash;
2531     ngx_int_t             rc;
2532     unsigned int          len;
2533     ngx_shm_zone_t        *shm_zone;
2534     ngx_slab_pool_t        *shpool;
2535     ngx_rbtrees_node_t    *node, *sentinel;
2536     ngx_ssl_sess_id_t     *sess_id;
2537     ngx_ssl_session_cache_t *cache;
2538
2539     shm_zone = SSL_CTX_get_ex_data(ssl, ngx_ssl_session_cache_index);
2540
2541     if (shm_zone == NULL) {
2542         return;
2543     }
2544
2545     cache = shm_zone->data;

```

```

2546
2547 #if OPENSSSL_VERSION_NUMBER >= 0x0090800fL
2548
2549     id = (u_char *) SSL_SESSION_get_id(sess, &len);
2550
2551 #else
2552
2553     id = sess->session_id;
2554     len = sess->session_id_length;
2555
2556 #endif
2557
2558     hash = ngx_crc32_short(id, len);
2559
2560     ngx_log_debug2(NGX_LOG_DEBUG_EVENT, ngx_cycle->log, 0,
2561                  "ssl remove session: %08XD:%ud", hash, len);
2562
2563     shpool = (ngx_slab_pool_t *) shm_zone->shm.addr;
2564
2565     ngx_shmtx_lock(&shpool->mutex);
2566
2567     node = cache->session_rbtrees.root;
2568     sentinel = cache->session_rbtrees.sentinel;
2569
2570     while (node != sentinel) {
2571
2572         if (hash < node->key) {
2573             node = node->left;
2574             continue;
2575         }
2576
2577         if (hash > node->key) {
2578             node = node->right;
2579             continue;
2580         }
2581
2582         /* hash == node->key */
2583
2584         sess_id = (ngx_ssl_sess_id_t *) node;
2585
2586         rc = ngx_memn2cmp(id, sess_id->id, len, (size_t) node->data);
2587
2588         if (rc == 0) {
2589
2590             ngx_queue_remove(&sess_id->queue);
2591
2592             ngx_rbtrees_delete(&cache->session_rbtrees, node);
2593
2594             ngx_slab_free_locked(shpool, sess_id->session);
2595 #if (NGX_PTR_SIZE == 4)
2596             ngx_slab_free_locked(shpool, sess_id->id);
2597 #endif
2598             ngx_slab_free_locked(shpool, sess_id);
2599
2600             goto done;
2601         }
2602
2603         node = (rc < 0) ? node->left : node->right;
2604     }
2605
2606 done:
2607
2608     ngx_shmtx_unlock(&shpool->mutex);
2609 }
2610
2611
2612 static void
2613 ngx_ssl_expire_sessions(ngx_ssl_session_cache_t *cache,
2614                        ngx_slab_pool_t *shpool, ngx_uint_t n)
2615 {
2616     time_t          now;
2617     ngx_queue_t     *q;
2618     ngx_ssl_sess_id_t *sess_id;
2619
2620     now = ngx_time();
2621

```

```

2622 while (n < 3) {
2623
2624     if (ngx_queue_empty(&cache->expire_queue)) {
2625         return;
2626     }
2627
2628     q = ngx_queue_last(&cache->expire_queue);
2629
2630     sess_id = ngx_queue_data(q, ngx_ssl_sess_id_t, queue);
2631
2632     if (n++ != 0 && sess_id->expire > now) {
2633         return;
2634     }
2635
2636     ngx_queue_remove(q);
2637
2638     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, ngx_cycle->log, 0,
2639         "expire session: %08Xi", sess_id->node.key);
2640
2641     ngx_rbtree_delete(&cache->session_rbtree, &sess_id->node);
2642
2643     ngx_slab_free_locked(shpool, sess_id->session);
2644 #if (NGX_PTR_SIZE == 4)
2645     ngx_slab_free_locked(shpool, sess_id->id);
2646 #endif
2647     ngx_slab_free_locked(shpool, sess_id);
2648 }
2649 }
2650
2651 static void
2652 ngx_ssl_session_rbtree_insert_value(ngx_rbtree_node_t *temp,
2653     ngx_rbtree_node_t *node, ngx_rbtree_node_t *sentinel)
2654 {
2655     ngx_rbtree_node_t **p;
2656     ngx_ssl_sess_id_t *sess_id, *sess_id_temp;
2657
2658     for ( ;; ) {
2659
2660         if (node->key < temp->key) {
2661             p = &temp->left;
2662
2663         } else if (node->key > temp->key) {
2664             p = &temp->right;
2665
2666         } else { /* node->key == temp->key */
2667
2668             sess_id = (ngx_ssl_sess_id_t *) node;
2669             sess_id_temp = (ngx_ssl_sess_id_t *) temp;
2670
2671             p = (ngx_memn2cmp(sess_id->id, sess_id_temp->id,
2672                 (size_t) node->data, (size_t) temp->data)
2673                 < 0) ? &temp->left : &temp->right;
2674         }
2675
2676         if (*p == sentinel) {
2677             break;
2678         }
2679
2680         temp = *p;
2681     }
2682
2683     *p = node;
2684     node->parent = temp;
2685     node->left = sentinel;
2686     node->right = sentinel;
2687     ngx_rbt_red(node);
2688 }
2689
2690 #ifdef SSL_CTRL_SET_TLSEXT_TICKET_KEY_CB
2691
2692 ngx_int_t
2693 ngx_ssl_session_ticket_keys(ngx_conf_t *cf, ngx_ssl_t *ssl, ngx_array_t *paths)

```



```

2698 {
2699     u_char                buf[48];
2700     ssize_t              n;
2701     ngx_str_t            *path;
2702     ngx_file_t           file;
2703     ngx_uint_t           i;
2704     ngx_array_t          *keys;
2705     ngx_file_info_t      fi;
2706     ngx_ssl_session_ticket_key_t *key;
2707
2708     if (paths == NULL) {
2709         return NGX_OK;
2710     }
2711
2712     keys = ngx_array_create(cf->pool, paths->nelts,
2713                             sizeof(ngx_ssl_session_ticket_key_t));
2714     if (keys == NULL) {
2715         return NGX_ERROR;
2716     }
2717
2718     path = paths->elts;
2719     for (i = 0; i < paths->nelts; i++) {
2720
2721         if (ngx_conf_full_name(cf->cycle, &path[i], 1) != NGX_OK) {
2722             return NGX_ERROR;
2723         }
2724
2725         ngx_memzero(&file, sizeof(ngx_file_t));
2726         file.name = path[i];
2727         file.log = cf->log;
2728
2729         file.fd = ngx_open_file(file.name.data, NGX_FILE_RDONLY, 0, 0);
2730         if (file.fd == NGX_INVALID_FILE) {
2731             ngx_conf_log_error(NGX_LOG_EMERG, cf, ngx_errno,
2732                                 ngx_open_file_n " \"%V\" failed", &file.name);
2733             return NGX_ERROR;
2734         }
2735
2736         if (ngx_fd_info(file.fd, &fi) == NGX_FILE_ERROR) {
2737             ngx_conf_log_error(NGX_LOG_CRIT, cf, ngx_errno,
2738                                 ngx_fd_info_n " \"%V\" failed", &file.name);
2739             goto failed;
2740         }
2741
2742         if (ngx_file_size(&fi) != 48) {
2743             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
2744                                 "\"%V\" must be 48 bytes", &file.name);
2745             goto failed;
2746         }
2747
2748         n = ngx_read_file(&file, buf, 48, 0);
2749
2750         if (n == NGX_ERROR) {
2751             ngx_conf_log_error(NGX_LOG_CRIT, cf, ngx_errno,
2752                                 ngx_read_file_n " \"%V\" failed", &file.name);
2753             goto failed;
2754         }
2755
2756         if (n != 48) {
2757             ngx_conf_log_error(NGX_LOG_CRIT, cf, 0,
2758                                 ngx_read_file_n " \"%V\" returned only "
2759                                 "%z bytes instead of 48", &file.name, n);
2760             goto failed;
2761         }
2762
2763         key = ngx_array_push(keys);
2764         if (key == NULL) {
2765             goto failed;
2766         }
2767
2768         ngx_memcpy(key->name, buf, 16);
2769         ngx_memcpy(key->aes_key, buf + 16, 16);
2770         ngx_memcpy(key->hmac_key, buf + 32, 16);
2771
2772         if (ngx_close_file(file.fd) == NGX_FILE_ERROR) {
2773             ngx_log_error(NGX_LOG_ALERT, cf->log, ngx_errno,

```

```

2774         ngx_close_file_n " \"%V\" failed", &file.name);
2775     }
2776 }
2777
2778 if (SSL_CTX_set_ex_data(ssl->ctx, ngx_ssl_session_ticket_keys_index, keys)
2779     == 0)
2780 {
2781     ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
2782                 "SSL_CTX_set_ex_data() failed");
2783     return NGX_ERROR;
2784 }
2785
2786 if (SSL_CTX_set_tlsext_ticket_key_cb(ssl->ctx,
2787     ngx_ssl_session_ticket_key_callback)
2788     == 0)
2789 {
2790     ngx_log_error(NGX_LOG_WARN, cf->log, 0,
2791                 "nginx was built with Session Tickets support, however, "
2792                 "now it is linked dynamically to an OpenSSL library "
2793                 "which has no tlsext support, therefore Session Tickets "
2794                 "are not available");
2795 }
2796
2797 return NGX_OK;
2798
2799 failed:
2800
2801 if (ngx_close_file(file.fd) == NGX_FILE_ERROR) {
2802     ngx_log_error(NGX_LOG_ALERT, cf->log, ngx_errno,
2803                 ngx_close_file_n " \"%V\" failed", &file.name);
2804 }
2805
2806 return NGX_ERROR;
2807 }
2808
2809
2810 #ifdef OPENSSL_NO_SHA256
2811 #define ngx_ssl_session_ticket_md EVP_sha1
2812 #else
2813 #define ngx_ssl_session_ticket_md EVP_sha256
2814 #endif
2815
2816
2817 static int
2818 ngx_ssl_session_ticket_key_callback(ngx_ssl_conn_t *ssl_conn,
2819     unsigned char *name, unsigned char *iv, EVP_CIPHER_CTX *ectx,
2820     HMAC_CTX *hctx, int enc)
2821 {
2822     SSL_CTX          *ssl_ctx;
2823     ngx_uint_t       i;
2824     ngx_array_t      *keys;
2825     ngx_ssl_session_ticket_key_t *key;
2826 #if (NGX_DEBUG)
2827     u_char           buf[32];
2828     ngx_connection_t *c;
2829 #endif
2830
2831     ssl_ctx = SSL_get_SSL_CTX(ssl_conn);
2832
2833     keys = SSL_CTX_get_ex_data(ssl_ctx, ngx_ssl_session_ticket_keys_index);
2834     if (keys == NULL) {
2835         return -1;
2836     }
2837
2838     key = keys->elts;
2839
2840 #if (NGX_DEBUG)
2841     c = ngx_ssl_get_connection(ssl_conn);
2842 #endif
2843
2844     if (enc == 1) {
2845         /* encrypt session ticket */
2846
2847         ngx_log_debug3(NGX_LOG_DEBUG_EVENT, c->log, 0,
2848                     "ssl session ticket encrypt, key: \"%s\" (%s session)",
2849                     ngx_hex_dump(buf, key[0].name, 16) - buf, buf,

```

```

2850         SSL_session_reused(ssl_conn) ? "reused" : "new");
2851
2852     RAND_pseudo_bytes(iv, 16);
2853     EVP_EncryptInit_ex(ectx, EVP_aes_128_cbc(), NULL, key[0].aes_key, iv);
2854     HMAC_Init_ex(hctx, key[0].hmac_key, 16,
2855                 ngx_ssl_session_ticket_md(), NULL);
2856     ngx_memcpy(name, key[0].name, 16);
2857
2858     return 0;
2859
2860 } else {
2861     /* decrypt session ticket */
2862
2863     for (i = 0; i < keys->nelts; i++) {
2864         if (ngx_memcmp(name, key[i].name, 16) == 0) {
2865             goto found;
2866         }
2867     }
2868
2869     ngx_log_debug2(NGX_LOG_DEBUG_EVENT, c->log, 0,
2870                  "ssl session ticket decrypt, key: \"%s\" not found",
2871                  ngx_hex_dump(buf, name, 16) - buf, buf);
2872
2873     return 0;
2874
2875 found:
2876
2877     ngx_log_debug3(NGX_LOG_DEBUG_EVENT, c->log, 0,
2878                  "ssl session ticket decrypt, key: \"%s\" \"%s\"",
2879                  ngx_hex_dump(buf, key[i].name, 16) - buf, buf,
2880                  (i == 0) ? " (default)" : "");
2881
2882     HMAC_Init_ex(hctx, key[i].hmac_key, 16,
2883                 ngx_ssl_session_ticket_md(), NULL);
2884     EVP_DecryptInit_ex(ectx, EVP_aes_128_cbc(), NULL, key[i].aes_key, iv);
2885
2886     return (i == 0) ? 1 : 2 /* renew */;
2887 }
2888 }
2889
2890 #else
2891
2892 ngx_int_t
2893 ngx_ssl_session_ticket_keys(ngx_conf_t *cf, ngx_ssl_t *ssl, ngx_array_t *paths)
2894 {
2895     if (paths) {
2896         ngx_log_error(NGX_LOG_WARN, ssl->log, 0,
2897                      "\"ssl_session_ticket_keys\" ignored, not supported");
2898     }
2899
2900     return NGX_OK;
2901 }
2902
2903 #endif
2904
2905
2906 void
2907 ngx_ssl_cleanup_ctx(void *data)
2908 {
2909     ngx_ssl_t *ssl = data;
2910
2911     SSL_CTX_free(ssl->ctx);
2912 }
2913
2914
2915 ngx_int_t
2916 ngx_ssl_check_host(ngx_connection_t *c, ngx_str_t *name)
2917 {
2918     X509 *cert;
2919
2920     cert = SSL_get_peer_certificate(c->ssl->connection);
2921     if (cert == NULL) {
2922         return NGX_ERROR;
2923     }
2924
2925 #if (OPENSSL_VERSION_NUMBER >= 0x10002002L && !defined LIBRESSL_VERSION_NUMBER)

```

```

2926
2927 /* X509_check_host() is only available in OpenSSL 1.0.2+ */
2928
2929 if (name->len == 0) {
2930     goto failed;
2931 }
2932
2933 if (X509_check_host(cert, (char *) name->data, name->len, 0, NULL) != 1) {
2934     ngx_log_debug0(NGX_LOG_DEBUG_EVENT, c->log, 0,
2935                 "X509_check_host(): no match");
2936     goto failed;
2937 }
2938
2939 ngx_log_debug0(NGX_LOG_DEBUG_EVENT, c->log, 0,
2940                 "X509_check_host(): match");
2941
2942 goto found;
2943
2944 #else
2945 {
2946     int             n, i;
2947     X509_NAME      *sname;
2948     ASN1_STRING    *str;
2949     X509_NAME_ENTRY *entry;
2950     GENERAL_NAME   *altname;
2951     STACK_OF(GENERAL_NAME) *altnames;
2952
2953     /*
2954      * As per RFC6125 and RFC2818, we check subjectAltName extension,
2955      * and if it's not present - commonName in Subject is checked.
2956      */
2957
2958     altnames = X509_get_ext_d2i(cert, NID_subject_alt_name, NULL, NULL);
2959
2960     if (altnames) {
2961         n = sk_GENERAL_NAME_num(altnames);
2962
2963         for (i = 0; i < n; i++) {
2964             altname = sk_GENERAL_NAME_value(altnames, i);
2965
2966             if (altname->type != GEN_DNS) {
2967                 continue;
2968             }
2969
2970             str = altname->d.dNSName;
2971
2972             ngx_log_debug2(NGX_LOG_DEBUG_EVENT, c->log, 0,
2973                         "SSL subjectAltName: \"%s\"",
2974                         ASN1_STRING_length(str), ASN1_STRING_data(str));
2975
2976             if (ngx_ssl_check_name(name, str) == NGX_OK) {
2977                 ngx_log_debug0(NGX_LOG_DEBUG_EVENT, c->log, 0,
2978                             "SSL subjectAltName: match");
2979                 GENERAL_NAMES_free(altnames);
2980                 goto found;
2981             }
2982         }
2983
2984         ngx_log_debug0(NGX_LOG_DEBUG_EVENT, c->log, 0,
2985                     "SSL subjectAltName: no match");
2986
2987         GENERAL_NAMES_free(altnames);
2988         goto failed;
2989     }
2990
2991     /*
2992      * If there is no subjectAltName extension, check commonName
2993      * in Subject. While RFC2818 requires to only check "most specific"
2994      * CN, both Apache and OpenSSL check all CNs, and so do we.
2995      */
2996
2997     sname = X509_get_subject_name(cert);
2998
2999     if (sname == NULL) {
3000         goto failed;
3001     }

```

```

3002
3003     i = -1;
3004     for ( ;; ) {
3005         i = X509_NAME_get_index_by_NID(sname, NID_commonName, i);
3006
3007         if (i < 0) {
3008             break;
3009         }
3010
3011         entry = X509_NAME_get_entry(sname, i);
3012         str = X509_NAME_ENTRY_get_data(entry);
3013
3014         ngx_log_debug2(NGX_LOG_DEBUG_EVENT, c->log, 0,
3015             "SSL commonName: \"%s\"",
3016             ASN1_STRING_length(str), ASN1_STRING_data(str));
3017
3018         if (ngx_ssl_check_name(name, str) == NGX_OK) {
3019             ngx_log_debug0(NGX_LOG_DEBUG_EVENT, c->log, 0,
3020                 "SSL commonName: match");
3021             goto found;
3022         }
3023     }
3024
3025     ngx_log_debug0(NGX_LOG_DEBUG_EVENT, c->log, 0,
3026         "SSL commonName: no match");
3027 }
3028 #endif
3029
3030 failed:
3031
3032     X509_free(cert);
3033     return NGX_ERROR;
3034
3035 found:
3036
3037     X509_free(cert);
3038     return NGX_OK;
3039 }
3040
3041
3042 #if (OPENSSL_VERSION_NUMBER < 0x10002002L || defined LIBRESSL_VERSION_NUMBER)
3043
3044 static ngx_int_t
3045 ngx_ssl_check_name(ngx_str_t *name, ASN1_STRING *pattern)
3046 {
3047     u_char *s, *p, *end;
3048     size_t slen, plen;
3049
3050     s = name->data;
3051     slen = name->len;
3052
3053     p = ASN1_STRING_data(pattern);
3054     plen = ASN1_STRING_length(pattern);
3055
3056     if (slen == plen && ngx_strncasecmp(s, p, plen) == 0) {
3057         return NGX_OK;
3058     }
3059
3060     if (plen > 2 && p[0] == '*' && p[1] == '.') {
3061         plen -= 1;
3062         p += 1;
3063
3064         end = s + slen;
3065         s = ngx_strlchr(s, end, '.');
3066
3067         if (s == NULL) {
3068             return NGX_ERROR;
3069         }
3070
3071         slen = end - s;
3072
3073         if (slen == plen && ngx_strncasecmp(s, p, plen) == 0) {
3074             return NGX_OK;
3075         }
3076     }
3077 }

```

```

3078     return NGX\_ERROR;
3079 }
3080
3081 #endif
3082
3083
3084 ngx\_int\_t
3085 ngx\_ssl\_get\_protocol(ngx\_connection\_t *c, ngx\_pool\_t *pool, ngx\_str\_t *s)
3086 {
3087     s->data = (u_char *) SSL_get_version(c->ssl->connection);
3088     return NGX\_OK;
3089 }
3090
3091
3092 ngx\_int\_t
3093 ngx\_ssl\_get\_cipher\_name(ngx\_connection\_t *c, ngx\_pool\_t *pool, ngx\_str\_t *s)
3094 {
3095     s->data = (u_char *) SSL_get_cipher_name(c->ssl->connection);
3096     return NGX\_OK;
3097 }
3098
3099
3100 ngx\_int\_t
3101 ngx\_ssl\_get\_session\_id(ngx\_connection\_t *c, ngx\_pool\_t *pool, ngx\_str\_t *s)
3102 {
3103     u_char          *buf;
3104     SSL_SESSION     *sess;
3105     unsigned int    len;
3106
3107     sess = SSL_get0_session(c->ssl->connection);
3108     if (sess == NULL) {
3109         s->len = 0;
3110         return NGX\_OK;
3111     }
3112
3113     #if OPENSSL\_VERSION\_NUMBER >= 0x0090800fL
3114
3115     buf = (u_char *) SSL_SESSION_get_id(sess, &len);
3116
3117     #else
3118
3119     buf = sess->session_id;
3120     len = sess->session_id_length;
3121
3122     #endif
3123
3124     s->len = 2 * len;
3125     s->data = ngx\_pnalloc(pool, 2 * len);
3126     if (s->data == NULL) {
3127         return NGX\_ERROR;
3128     }
3129
3130     ngx\_hex\_dump(s->data, buf, len);
3131
3132     return NGX\_OK;
3133 }
3134
3135
3136 ngx\_int\_t
3137 ngx\_ssl\_get\_session\_reused(ngx\_connection\_t *c, ngx\_pool\_t *pool, ngx\_str\_t *s)
3138 {
3139     if (SSL_session_reused(c->ssl->connection)) {
3140         ngx\_str\_set(s, "r");
3141     } else {
3142         ngx\_str\_set(s, ".");
3143     }
3144
3145     return NGX\_OK;
3146 }
3147
3148
3149
3150 ngx\_int\_t
3151 ngx\_ssl\_get\_server\_name(ngx\_connection\_t *c, ngx\_pool\_t *pool, ngx\_str\_t *s)
3152 {
3153     #ifdef SSL\_CTRL\_SET\_TLSEXT\_HOSTNAME

```

```

3154     const char *servername;
3155
3156
3157     servername = SSL_get_servername(c->ssl->connection,
3158                                     TLSEXT_NAMETYPE_host_name);
3159     if (servername) {
3160         s->data = (u_char *) servername;
3161         s->len = ngx_strlen(servername);
3162         return NGX_OK;
3163     }
3164
3165 #endif
3166
3167     s->len = 0;
3168     return NGX_OK;
3169 }
3170
3171
3172 ngx_int_t
3173 ngx_ssl_get_raw_certificate(ngx_connection_t *c, ngx_pool_t *pool, ngx_str_t *s)
3174 {
3175     size_t len;
3176     BIO *bio;
3177     X509 *cert;
3178
3179     s->len = 0;
3180
3181     cert = SSL_get_peer_certificate(c->ssl->connection);
3182     if (cert == NULL) {
3183         return NGX_OK;
3184     }
3185
3186     bio = BIO_new(BIO_s_mem());
3187     if (bio == NULL) {
3188         ngx_ssl_error(NGX_LOG_ALERT, c->log, 0, "BIO_new() failed");
3189         X509_free(cert);
3190         return NGX_ERROR;
3191     }
3192
3193     if (PEM_write_bio_X509(bio, cert) == 0) {
3194         ngx_ssl_error(NGX_LOG_ALERT, c->log, 0, "PEM_write_bio_X509() failed");
3195         goto failed;
3196     }
3197
3198     len = BIO_pending(bio);
3199     s->len = len;
3200
3201     s->data = ngx_pnalloc(pool, len);
3202     if (s->data == NULL) {
3203         goto failed;
3204     }
3205
3206     BIO_read(bio, s->data, len);
3207
3208     BIO_free(bio);
3209     X509_free(cert);
3210
3211     return NGX_OK;
3212
3213 failed:
3214
3215     BIO_free(bio);
3216     X509_free(cert);
3217
3218     return NGX_ERROR;
3219 }
3220
3221
3222 ngx_int_t
3223 ngx_ssl_get_certificate(ngx_connection_t *c, ngx_pool_t *pool, ngx_str_t *s)
3224 {
3225     u_char *p;
3226     size_t len;
3227     ngx_uint_t i;
3228     ngx_str_t cert;
3229

```

```

3230     if (ngx_ssl_get_raw_certificate(c, pool, &cert) != NGX_OK) {
3231         return NGX_ERROR;
3232     }
3233
3234     if (cert.len == 0) {
3235         s->len = 0;
3236         return NGX_OK;
3237     }
3238
3239     len = cert.len - 1;
3240
3241     for (i = 0; i < cert.len - 1; i++) {
3242         if (cert.data[i] == LF) {
3243             len++;
3244         }
3245     }
3246
3247     s->len = len;
3248     s->data = ngx_pnalloc(pool, len);
3249     if (s->data == NULL) {
3250         return NGX_ERROR;
3251     }
3252
3253     p = s->data;
3254
3255     for (i = 0; i < cert.len - 1; i++) {
3256         *p++ = cert.data[i];
3257         if (cert.data[i] == LF) {
3258             *p++ = '\\t';
3259         }
3260     }
3261
3262     return NGX_OK;
3263 }
3264
3265
3266 ngx_int_t
3267 ngx_ssl_get_subject_dn(ngx_connection_t *c, ngx_pool_t *pool, ngx_str_t *s)
3268 {
3269     char      *p;
3270     size_t    len;
3271     X509      *cert;
3272     X509_NAME *name;
3273
3274     s->len = 0;
3275
3276     cert = SSL_get_peer_certificate(c->ssl->connection);
3277     if (cert == NULL) {
3278         return NGX_OK;
3279     }
3280
3281     name = X509_get_subject_name(cert);
3282     if (name == NULL) {
3283         X509_free(cert);
3284         return NGX_ERROR;
3285     }
3286
3287     p = X509_NAME_oneline(name, NULL, 0);
3288
3289     for (len = 0; p[len]; len++) { /* void */ }
3290
3291     s->len = len;
3292     s->data = ngx_pnalloc(pool, len);
3293     if (s->data == NULL) {
3294         OPENSSL_free(p);
3295         X509_free(cert);
3296         return NGX_ERROR;
3297     }
3298
3299     ngx_memcpy(s->data, p, len);
3300
3301     OPENSSL_free(p);
3302     X509_free(cert);
3303
3304     return NGX_OK;
3305 }

```



```

3306
3307
3308 ngx_int_t
3309 ngx_ssl_get_issuer_dn(ngx_connection_t *c, ngx_pool_t *pool, ngx_str_t *s)
3310 {
3311     char        *p;
3312     size_t      len;
3313     X509        *cert;
3314     X509_NAME   *name;
3315
3316     s->len = 0;
3317
3318     cert = SSL_get_peer_certificate(c->ssl->connection);
3319     if (cert == NULL) {
3320         return NGX_OK;
3321     }
3322
3323     name = X509_get_issuer_name(cert);
3324     if (name == NULL) {
3325         X509_free(cert);
3326         return NGX_ERROR;
3327     }
3328
3329     p = X509_NAME_oneline(name, NULL, 0);
3330
3331     for (len = 0; p[len]; len++) { /* void */ }
3332
3333     s->len = len;
3334     s->data = ngx_pnalloc(pool, len);
3335     if (s->data == NULL) {
3336         OPENSSL_free(p);
3337         X509_free(cert);
3338         return NGX_ERROR;
3339     }
3340
3341     ngx_memcpy(s->data, p, len);
3342
3343     OPENSSL_free(p);
3344     X509_free(cert);
3345
3346     return NGX_OK;
3347 }
3348
3349
3350 ngx_int_t
3351 ngx_ssl_get_serial_number(ngx_connection_t *c, ngx_pool_t *pool, ngx_str_t *s)
3352 {
3353     size_t      len;
3354     X509        *cert;
3355     BIO         *bio;
3356
3357     s->len = 0;
3358
3359     cert = SSL_get_peer_certificate(c->ssl->connection);
3360     if (cert == NULL) {
3361         return NGX_OK;
3362     }
3363
3364     bio = BIO_new(BIO_s_mem());
3365     if (bio == NULL) {
3366         X509_free(cert);
3367         return NGX_ERROR;
3368     }
3369
3370     i2a_ASN1_INTEGER(bio, X509_get_serialNumber(cert));
3371     len = BIO_pending(bio);
3372
3373     s->len = len;
3374     s->data = ngx_pnalloc(pool, len);
3375     if (s->data == NULL) {
3376         BIO_free(bio);
3377         X509_free(cert);
3378         return NGX_ERROR;
3379     }
3380
3381     BIO_read(bio, s->data, len);

```

```

3382     BIO_free(bio);
3383     X509_free(cert);
3384
3385     return NGX\_OK;
3386 }
3387
3388
3389 ngx\_int\_t
3390 ngx\_ssl\_get\_fingerprint(ngx\_connection\_t *c, ngx\_pool\_t *pool, ngx\_str\_t *s)
3391 {
3392     X509          *cert;
3393     unsigned int  len;
3394     u_char        buf[EVP_MAX_MD_SIZE];
3395
3396     s->len = 0;
3397
3398     cert = SSL_get_peer_certificate(c->ssl->connection);
3399     if (cert == NULL) {
3400         return NGX\_OK;
3401     }
3402
3403     if (!X509_digest(cert, EVP_sha1(), buf, &len)) {
3404         X509_free(cert);
3405         return NGX\_ERROR;
3406     }
3407
3408     s->len = 2 * len;
3409     s->data = ngx\_pnalloc(pool, 2 * len);
3410     if (s->data == NULL) {
3411         X509_free(cert);
3412         return NGX\_ERROR;
3413     }
3414
3415     ngx\_hex\_dump(s->data, buf, len);
3416
3417     X509_free(cert);
3418
3419     return NGX\_OK;
3420 }
3421
3422
3423 ngx\_int\_t
3424 ngx\_ssl\_get\_client\_verify(ngx\_connection\_t *c, ngx\_pool\_t *pool, ngx\_str\_t *s)
3425 {
3426     X509          *cert;
3427
3428     if (SSL_get_verify_result(c->ssl->connection) != X509_V_OK) {
3429         ngx\_str\_set(s, "FAILED");
3430         return NGX\_OK;
3431     }
3432
3433     cert = SSL_get_peer_certificate(c->ssl->connection);
3434
3435     if (cert) {
3436         ngx\_str\_set(s, "SUCCESS");
3437     } else {
3438         ngx\_str\_set(s, "NONE");
3439     }
3440
3441     X509_free(cert);
3442
3443     return NGX\_OK;
3444 }
3445
3446
3447
3448 static void *
3449 ngx\_openssl\_create\_conf(ngx\_cycle\_t *cycle)
3450 {
3451     ngx\_openssl\_conf\_t *oscf;
3452
3453     oscf = ngx\_pcalloc(cycle->pool, sizeof(ngx\_openssl\_conf\_t));
3454     if (oscf == NULL) {
3455         return NULL;
3456     }
3457

```

```

3458  /*
3459  * set by ngx\_palloc\(\):
3460  *
3461  *     oscf->engine = 0;
3462  */
3463
3464  return oscf;
3465 }
3466
3467 static char *
3468 ngx_openssl_engine(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
3469 {
3470 #ifndef OPENSSSL_NO_ENGINE
3471     ngx\_openssl\_conf\_t *oscf = conf;
3472
3473     ENGINE      *engine;
3474     ngx\_str\_t   *value;
3475
3476     if (oscf->engine) {
3477         return "is duplicate";
3478     }
3479
3480     oscf->engine = 1;
3481
3482     value = cf->args->elts;
3483
3484     engine = ENGINE_by_id((const char *) value[1].data);
3485
3486     if (engine == NULL) {
3487         ngx\_ssl\_error(NGX\_LOG\_WARN, cf->log, 0,
3488             "ENGINE_by_id(\"%V\") failed", &value[1]);
3489         return NGX\_CONF\_ERROR;
3490     }
3491
3492     if (ENGINE_set_default(engine, ENGINE_METHOD_ALL) == 0) {
3493         ngx\_ssl\_error(NGX\_LOG\_WARN, cf->log, 0,
3494             "ENGINE_set_default(\"%V\", ENGINE_METHOD_ALL) failed",
3495             &value[1]);
3496
3497         ENGINE_free(engine);
3498
3499         return NGX\_CONF\_ERROR;
3500     }
3501
3502     ENGINE_free(engine);
3503
3504     return NGX\_CONF\_OK;
3505 #else
3506     return "is not supported";
3507 #endif
3508 }
3509
3510 static void
3511 ngx_openssl_exit(ngx\_cycle\_t *cycle)
3512 {
3513     EVP_cleanup();
3514 #ifndef OPENSSSL_NO_ENGINE
3515     ENGINE_cleanup();
3516 #endif
3517 }

```

[One Level Up](#)

[Top Level](#)

# src/core/nginx\_slab.h - nginx-1.7.10

## Data types defined

- [ngx\\_slab\\_page\\_s](#)
- [ngx\\_slab\\_page\\_t](#)
- [ngx\\_slab\\_pool\\_t](#)

## Macros defined

- [\\_NGX\\_SLAB\\_H\\_INCLUDED](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_SLAB_H_INCLUDED
9 #define _NGX_SLAB_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 typedef struct ngx\_slab\_page\_s ngx\_slab\_page\_t;
17
18 struct ngx\_slab\_page\_s {
19     uintptr_t    slab;
20     ngx\_slab\_page\_t *next;
21     uintptr_t    prev;
22 };
23
24
25 typedef struct {
26     ngx\_shmtx\_sh\_t    lock;
27
28     size_t            min_size;
29     size_t            min_shift;
30
31     ngx\_slab\_page\_t *pages;
32     ngx\_slab\_page\_t *last;
33     ngx\_slab\_page\_t free;
34
35     u_char            *start;
36     u_char            *end;
37
38     ngx\_shmtx\_t        mutex;
39
40     u_char            *log_ctx;
41     u_char            zero;
42
43     unsigned          log_nomem:1;
44
45     void              *data;
46     void              *addr;
47 } ngx\_slab\_pool\_t;
48
49
50 void ngx\_slab\_init(ngx\_slab\_pool\_t *pool);
51 void *ngx\_slab\_alloc(ngx\_slab\_pool\_t *pool, size_t size);
52 void *ngx\_slab\_alloc\_locked(ngx\_slab\_pool\_t *pool, size_t size);
```

```
53 void *ngx_slab_alloc(ngx_slab_pool_t *pool, size_t size);
54 void *ngx_slab_alloc_locked(ngx_slab_pool_t *pool, size_t size);
55 void ngx_slab_free(ngx_slab_pool_t *pool, void *p);
56 void ngx_slab_free_locked(ngx_slab_pool_t *pool, void *p);
57
58
59 #endif /* NGX_SLAB_H_INCLUDED */
```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_shmtx.h - nginx-1.7.10

### Data types defined

- [ngx\\_shmtx\\_sh\\_t](#)
- [ngx\\_shmtx\\_t](#)

### Macros defined

- [\\_NGX\\_SHMTX\\_H\\_INCLUDED\\_](#)

### Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #ifndef _NGX_SHMTX_H_INCLUDED_
9  #define _NGX_SHMTX_H_INCLUDED_
10
11
12  #include <ngx_config.h>
13  #include <ngx_core.h>
14
15
16  typedef struct {
17      ngx_atomic_t  lock;
18  #if (NGX_HAVE_POSIX_SEM)
19      ngx_atomic_t  wait;
20  #endif
21  } ngx_shmtx_sh_t;
22
23
24  typedef struct {
25  #if (NGX_HAVE_ATOMIC_OPS)
26      ngx_atomic_t  *lock;
27  #if (NGX_HAVE_POSIX_SEM)
28      ngx_atomic_t  *wait;
29      ngx_uint_t     semaphore;
30      sem_t          sem;
31  #endif
32  #else
33      ngx_fd_t       fd;
34      u_char         *name;
35  #endif
36      ngx_uint_t     spin;
37  } ngx_shmtx_t;
38
39
40  ngx_int_t ngx_shmtx_create(ngx_shmtx_t *mtx, ngx_shmtx_sh_t *addr,
41      u_char *name);
42  void ngx_shmtx_destroy(ngx_shmtx_t *mtx);
43  ngx_uint_t ngx_shmtx_trylock(ngx_shmtx_t *mtx);
44  void ngx_shmtx_lock(ngx_shmtx_t *mtx);
45  void ngx_shmtx_unlock(ngx_shmtx_t *mtx);
46  ngx_uint_t ngx_shmtx_force_unlock(ngx_shmtx_t *mtx, ngx_pid_t pid);
47
48
49  #endif /* _NGX_SHMTX_H_INCLUDED_ */
```

## src/core/nginx\_shmtx.c - nginx-1.7.10

### Functions defined

- [ngx\\_shmtx\\_create](#)
- [ngx\\_shmtx\\_create](#)
- [ngx\\_shmtx\\_destroy](#)
- [ngx\\_shmtx\\_destroy](#)
- [ngx\\_shmtx\\_force\\_unlock](#)
- [ngx\\_shmtx\\_force\\_unlock](#)
- [ngx\\_shmtx\\_lock](#)
- [ngx\\_shmtx\\_lock](#)
- [ngx\\_shmtx\\_trylock](#)
- [ngx\\_shmtx\\_trylock](#)
- [ngx\\_shmtx\\_unlock](#)
- [ngx\\_shmtx\\_unlock](#)
- [ngx\\_shmtx\\_wakeup](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10
11
12 #if (NGX_HAVE_ATOMIC_OPS)
13
14
15 static void ngx_shmtx_wakeup(ngx_shmtx_t *mtx);
16
17
18 ngx_int_t
19 ngx_shmtx_create(ngx_shmtx_t *mtx, ngx_shmtx_sh_t *addr, u_char *name)
20 {
21     mtx->lock = &addr->lock;
22
23     if (mtx->spin == (ngx_uint_t) -1) {
24         return NGX_OK;
25     }
26
27     mtx->spin = 2048;
28
29 #if (NGX_HAVE_POSIX_SEM)
30
31     mtx->wait = &addr->wait;
32
33     if (sem_init(&mtx->sem, 1, 0) == -1) {
34         ngx_log_error(NGX_LOG_ALERT, ngx_cycle->log, ngx_errno,
35                     "sem_init() failed");
36     }
37 #endif
38 }
```

```

36     } else {
37         mtx->semaphore = 1;
38     }
39
40 #endif
41
42     return NGX_OK;
43 }
44
45
46 void
47 ngx_shmtx_destroy(ngx_shmtx_t *mtx)
48 {
49     #if (NGX_HAVE_POSIX_SEM)
50
51     if (mtx->semaphore) {
52         if (sem_destroy(&mtx->sem) == -1) {
53             ngx_log_error(NGX_LOG_ALERT, ngx_cycle->log, ngx_errno,
54                 "sem_destroy() failed");
55         }
56     }
57
58 #endif
59 }
60
61
62 ngx_uint_t
63 ngx_shmtx_trylock(ngx_shmtx_t *mtx)
64 {
65     return (*mtx->lock == 0 && ngx_atomic_cmp_set(mtx->lock, 0, ngx_pid));
66 }
67
68
69 void
70 ngx_shmtx_lock(ngx_shmtx_t *mtx)
71 {
72     ngx_uint_t         i, n;
73
74     ngx_log_debug0(NGX_LOG_DEBUG_CORE, ngx_cycle->log, 0, "shmtx lock");
75
76     for ( ;; ) {
77
78         if (*mtx->lock == 0 && ngx_atomic_cmp_set(mtx->lock, 0, ngx_pid)) {
79             return;
80         }
81
82         if (ngx_ncpu > 1) {
83
84             for (n = 1; n < mtx->spin; n <= 1) {
85
86                 for (i = 0; i < n; i++) {
87                     ngx_cpu_pause();
88                 }
89
90                 if (*mtx->lock == 0
91                     && ngx_atomic_cmp_set(mtx->lock, 0, ngx_pid))
92                 {
93                     return;
94                 }
95             }
96         }
97
98     #if (NGX_HAVE_POSIX_SEM)
99
100     if (mtx->semaphore) {
101         (void) ngx_atomic_fetch_add(mtx->wait, 1);
102
103         if (*mtx->lock == 0 && ngx_atomic_cmp_set(mtx->lock, 0, ngx_pid)) {
104             (void) ngx_atomic_fetch_add(mtx->wait, -1);
105             return;
106         }
107
108         ngx_log_debug1(NGX_LOG_DEBUG_CORE, ngx_cycle->log, 0,
109             "shmtx wait %uA", *mtx->wait);
110
111         while (sem_wait(&mtx->sem) == -1) {

```



```

112         ngx_err_t  err;
113
114         err = ngx_errno;
115
116         if (err != NGX_EINTR) {
117             ngx_log_error(NGX_LOG_ALERT, ngx_cycle->log, err,
118                 "sem_wait() failed while waiting on shmtx");
119             break;
120         }
121     }
122
123     ngx_log_debug0(NGX_LOG_DEBUG_CORE, ngx_cycle->log, 0,
124         "shmtx awoke");
125
126     continue;
127 }
128
129 #endif
130
131     ngx_sched_yield();
132 }
133 }
134
135
136 void
137 ngx_shmtx_unlock(ngx_shmtx_t *mtx)
138 {
139     if (mtx->spin != (ngx_uint_t) -1) {
140         ngx_log_debug0(NGX_LOG_DEBUG_CORE, ngx_cycle->log, 0, "shmtx unlock");
141     }
142
143     if (ngx_atomic_cmp_set(mtx->lock, ngx_pid, 0)) {
144         ngx_shmtx_wakeup(mtx);
145     }
146 }
147
148
149 ngx_uint_t
150 ngx_shmtx_force_unlock(ngx_shmtx_t *mtx, ngx_pid_t pid)
151 {
152     ngx_log_debug0(NGX_LOG_DEBUG_CORE, ngx_cycle->log, 0,
153         "shmtx forced unlock");
154
155     if (ngx_atomic_cmp_set(mtx->lock, pid, 0)) {
156         ngx_shmtx_wakeup(mtx);
157         return 1;
158     }
159
160     return 0;
161 }
162
163
164 static void
165 ngx_shmtx_wakeup(ngx_shmtx_t *mtx)
166 {
167     #if (NGX_HAVE_POSIX_SEM)
168         ngx_atomic_uint_t  wait;
169
170         if (!mtx->semaphore) {
171             return;
172         }
173
174         for ( ;; ) {
175             wait = *mtx->wait;
176
177             if ((ngx_atomic_int_t) wait <= 0) {
178                 return;
179             }
180
181             if (ngx_atomic_cmp_set(mtx->wait, wait, wait - 1)) {
182                 break;
183             }
184         }
185     }
186
187     ngx_log_debug1(NGX_LOG_DEBUG_CORE, ngx_cycle->log, 0,

```

```

188         "shmtx wake %uA", wait);
189
190     if (sem_post(&mtx->sem) == -1) {
191         ngx_log_error(NGX_LOG_ALERT, ngx_cycle->log, ngx_errno,
192             "sem_post() failed while wake shmtx");
193     }
194
195 #endif
196 }
197
198
199 #else
200
201
202 ngx_int_t
203 ngx_shmtx_create(ngx_shmtx_t *mtx, ngx_shmtx_sh_t *addr, u_char *name)
204 {
205     if (mtx->name) {
206
207         if (ngx_strcmp(name, mtx->name) == 0) {
208             mtx->name = name;
209             return NGX_OK;
210         }
211
212         ngx_shmtx_destroy(mtx);
213     }
214
215     mtx->fd = ngx_open_file(name, NGX_FILE_RDWR, NGX_FILE_CREATE_OR_OPEN,
216         NGX_FILE_DEFAULT_ACCESS);
217
218     if (mtx->fd == NGX_INVALID_FILE) {
219         ngx_log_error(NGX_LOG_EMERG, ngx_cycle->log, ngx_errno,
220             ngx_open_file_n " \"%s\" failed", name);
221         return NGX_ERROR;
222     }
223
224     if (ngx_delete_file(name) == NGX_FILE_ERROR) {
225         ngx_log_error(NGX_LOG_ALERT, ngx_cycle->log, ngx_errno,
226             ngx_delete_file_n " \"%s\" failed", name);
227     }
228
229     mtx->name = name;
230
231     return NGX_OK;
232 }
233
234
235 void
236 ngx_shmtx_destroy(ngx_shmtx_t *mtx)
237 {
238     if (ngx_close_file(mtx->fd) == NGX_FILE_ERROR) {
239         ngx_log_error(NGX_LOG_ALERT, ngx_cycle->log, ngx_errno,
240             ngx_close_file_n " \"%s\" failed", mtx->name);
241     }
242 }
243
244
245 ngx_uint_t
246 ngx_shmtx_trylock(ngx_shmtx_t *mtx)
247 {
248     ngx_err_t  err;
249
250     err = ngx_trylock_fd(mtx->fd);
251
252     if (err == 0) {
253         return 1;
254     }
255
256     if (err == NGX_EAGAIN) {
257         return 0;
258     }
259
260 #if __osf__ /* Tru64 UNIX */
261
262     if (err == NGX_EACCES) {
263         return 0;

```

```

264     }
265
266 #endif
267
268     ngx_log_abort(err, ngx_trylock_fd_n " %s failed", mtx->name);
269
270     return 0;
271 }
272
273
274 void
275 ngx_shmtx_lock(ngx_shmtx_t *mtx)
276 {
277     ngx_err_t  err;
278
279     err = ngx_lock_fd(mtx->fd);
280
281     if (err == 0) {
282         return;
283     }
284
285     ngx_log_abort(err, ngx_lock_fd_n " %s failed", mtx->name);
286 }
287
288
289 void
290 ngx_shmtx_unlock(ngx_shmtx_t *mtx)
291 {
292     ngx_err_t  err;
293
294     err = ngx_unlock_fd(mtx->fd);
295
296     if (err == 0) {
297         return;
298     }
299
300     ngx_log_abort(err, ngx_unlock_fd_n " %s failed", mtx->name);
301 }
302
303
304 ngx_uint_t
305 ngx_shmtx_force_unlock(ngx_shmtx_t *mtx, ngx_pid_t pid)
306 {
307     return 0;
308 }
309
310 #endif

```

[One Level Up](#)

[Top Level](#)

## src/os/unix/nginx\_process\_cycle.c - nginx-1.7.10

### Global variables defined

- [master\\_process](#)
- [ngx\\_cache\\_loader\\_ctx](#)
- [ngx\\_cache\\_manager\\_ctx](#)
- [ngx\\_change\\_binary](#)
- [ngx\\_daemonized](#)
- [ngx\\_debug\\_quit](#)
- [ngx\\_exit\\_cycle](#)
- [ngx\\_exit\\_log](#)
- [ngx\\_exit\\_log\\_file](#)
- [ngx\\_exiting](#)
- [ngx\\_inherited](#)
- [ngx\\_new\\_binary](#)
- [ngx\\_noaccept](#)
- [ngx\\_noaccepting](#)
- [ngx\\_pid](#)
- [ngx\\_process](#)
- [ngx\\_quit](#)
- [ngx\\_reap](#)
- [ngx\\_reconfigure](#)
- [ngx\\_reopen](#)
- [ngx\\_restart](#)
- [ngx\\_sigalrm](#)
- [ngx\\_sigio](#)
- [ngx\\_terminate](#)
- [ngx\\_threaded](#)
- [ngx\\_threads](#)
- [ngx\\_threads\\_n](#)

### Functions defined

- [ngx\\_cache\\_loader\\_process\\_handler](#)

- [ngx\\_cache\\_manager\\_process\\_cycle](#)
- [ngx\\_cache\\_manager\\_process\\_handler](#)
- [ngx\\_channel\\_handler](#)
- [ngx\\_master\\_process\\_cycle](#)
- [ngx\\_master\\_process\\_exit](#)
- [ngx\\_pass\\_open\\_channel](#)
- [ngx\\_reap\\_children](#)
- [ngx\\_signal\\_worker\\_processes](#)
- [ngx\\_single\\_process\\_cycle](#)
- [ngx\\_start\\_cache\\_manager\\_processes](#)
- [ngx\\_start\\_worker\\_processes](#)
- [ngx\\_wakeup\\_worker\\_threads](#)
- [ngx\\_worker\\_process\\_cycle](#)
- [ngx\\_worker\\_process\\_exit](#)
- [ngx\\_worker\\_process\\_init](#)
- [ngx\\_worker\\_thread\\_cycle](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11 #include <ngx_channel.h>
12
13
14 static void ngx_start_worker_processes(ngx_cycle_t *cycle, ngx_int_t n,
15     ngx_int_t type);
16 static void ngx_start_cache_manager_processes(ngx_cycle_t *cycle,
17     ngx_uint_t respawn);
18 static void ngx_pass_open_channel(ngx_cycle_t *cycle, ngx_channel_t *ch);
19 static void ngx_signal_worker_processes(ngx_cycle_t *cycle, int signo);
20 static void ngx_reap_children(ngx_cycle_t *cycle);
21 static void ngx_master_process_exit(ngx_cycle_t *cycle);
22 static void ngx_worker_process_cycle(ngx_cycle_t *cycle, void *data);
23 static void ngx_worker_process_init(ngx_cycle_t *cycle, ngx_int_t worker);
24 static void ngx_worker_process_exit(ngx_cycle_t *cycle);
25 static void ngx_channel_handler(ngx_event_t *ev);
26 #if (NGX_THREADS)
27 static void ngx_wakeup_worker_threads(ngx_cycle_t *cycle);
28 static ngx_thread_value_t ngx_worker_thread_cycle(void *data);
29 #endif
30 static void ngx_cache_manager_process_cycle(ngx_cycle_t *cycle, void *data);
31 static void ngx_cache_manager_process_handler(ngx_event_t *ev);
32 static void ngx_cache_loader_process_handler(ngx_event_t *ev);
33
34
35 ngx_uint_t     ngx_process;
36 ngx_pid_t     ngx_pid;

```

```

37 ngx_uint_t ngx_threaded;
38
39 sig_atomic_t ngx_reap;
40 sig_atomic_t ngx_sigio;
41 sig_atomic_t ngx_sigalrm;
42 sig_atomic_t ngx_terminate;
43 sig_atomic_t ngx_quit;
44 sig_atomic_t ngx_debug_quit;
45 ngx_uint_t ngx_exiting;
46 sig_atomic_t ngx_reconfigure;
47 sig_atomic_t ngx_reopen;
48
49 sig_atomic_t ngx_change_binary;
50 ngx_pid_t ngx_new_binary;
51 ngx_uint_t ngx_inherited;
52 ngx_uint_t ngx_daemonized;
53
54 sig_atomic_t ngx_noaccept;
55 ngx_uint_t ngx_noaccepting;
56 ngx_uint_t ngx_restart;
57
58
59 #if (NGX_THREADS)
60 volatile ngx_thread_t ngx_threads[NGX_MAX_THREADS];
61 ngx_int_t ngx_threads_n;
62 #endif
63
64
65 static u_char master_process[] = "master process";
66
67
68 static ngx_cache_manager_ctx_t ngx_cache_manager_ctx = {
69     ngx_cache_manager_process_handler, "cache manager process", 0
70 };
71
72 static ngx_cache_loader_ctx_t ngx_cache_loader_ctx = {
73     ngx_cache_loader_process_handler, "cache loader process", 60000
74 };
75
76
77 static ngx_cycle_t ngx_exit_cycle;
78 static ngx_log_t ngx_exit_log;
79 static ngx_open_file_t ngx_exit_log_file;
80
81
82 void
83 ngx_master_process_cycle(ngx_cycle_t *cycle)
84 {
85     char *title;
86     u_char *p;
87     size_t size;
88     ngx_int_t i;
89     ngx_uint_t n, sigio;
90     sigset_t set;
91     struct itimerval itv;
92     ngx_uint_t live;
93     ngx_msec_t delay;
94     ngx_listening_t *ls;
95     ngx_core_conf_t *ccf;
96
97     sigemptyset(&set);
98     sigaddset(&set, SIGCHLD);
99     sigaddset(&set, SIGALRM);
100    sigaddset(&set, SIGIO);
101    sigaddset(&set, SIGINT);
102    sigaddset(&set, ngx_signal_value(NGX_RECONFIGURE_SIGNAL));
103    sigaddset(&set, ngx_signal_value(NGX_REOPEN_SIGNAL));
104    sigaddset(&set, ngx_signal_value(NGX_NOACCEPT_SIGNAL));
105    sigaddset(&set, ngx_signal_value(NGX_TERMINATE_SIGNAL));
106    sigaddset(&set, ngx_signal_value(NGX_SHUTDOWN_SIGNAL));
107    sigaddset(&set, ngx_signal_value(NGX_CHANGEBIN_SIGNAL));
108
109    if (sigprocmask(SIG_BLOCK, &set, NULL) == -1) {
110        ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
111            "sigprocmask() failed");
112    }

```

```

113 sigemptyset(&set);
114
115
116
117 size = sizeof(master_process);
118
119 for (i = 0; i < ngx_argc; i++) {
120     size += ngx_strlen(ngx_argv[i]) + 1;
121 }
122
123 title = ngx_pnalloc(cycle->pool, size);
124 if (title == NULL) {
125     /* fatal */
126     exit(2);
127 }
128
129 p = ngx_cpymem(title, master_process, sizeof(master_process) - 1);
130 for (i = 0; i < ngx_argc; i++) {
131     *p++ = ' ';
132     p = ngx_cpystrn(p, (u_char *) ngx_argv[i], size);
133 }
134
135 ngx_setproctitle(title);
136
137
138 ccf = (ngx_core_conf_t *) ngx_get_conf(cycle->conf_ctx, ngx_core_module);
139
140 ngx_start_worker_processes(cycle, ccf->worker_processes,
141     NGX_PROCESS_RESPAWN);
142 ngx_start_cache_manager_processes(cycle, 0);
143
144 ngx_new_binary = 0;
145 delay = 0;
146 sigio = 0;
147 live = 1;
148
149 for ( ;; ) {
150     if (delay) {
151         if (ngx_sigalrm) {
152             sigio = 0;
153             delay *= 2;
154             ngx_sigalrm = 0;
155         }
156
157         ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
158             "termination cycle: %d", delay);
159
160         itv.it_interval.tv_sec = 0;
161         itv.it_interval.tv_usec = 0;
162         itv.it_value.tv_sec = delay / 1000;
163         itv.it_value.tv_usec = (delay % 1000) * 1000;
164
165         if (setitimer(ITIMER_REAL, &itv, NULL) == -1) {
166             ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
167                 "setitimer() failed");
168         }
169     }
170
171     ngx_log_debug0(NGX_LOG_DEBUG_EVENT, cycle->log, 0, "sigsuspend");
172
173     sigsuspend(&set);
174
175     ngx_time_update();
176
177     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
178         "wake up, sigio %i", sigio);
179
180     if (ngx_reap) {
181         ngx_reap = 0;
182         ngx_log_debug0(NGX_LOG_DEBUG_EVENT, cycle->log, 0, "reap children");
183
184         live = ngx_reap_children(cycle);
185     }
186
187     if (!live && (ngx_terminate || ngx_quit)) {
188         ngx_master_process_exit(cycle);

```

```

189     }
190
191     if (ngx_terminate) {
192         if (delay == 0) {
193             delay = 50;
194         }
195
196         if (sigio) {
197             sigio--;
198             continue;
199         }
200
201         sigio = ccf->worker_processes + 2 /* cache processes */;
202
203         if (delay > 1000) {
204             ngx_signal_worker_processes(cycle, SIGKILL);
205         } else {
206             ngx_signal_worker_processes(cycle,
207                                     ngx_signal_value(NGX_TERMINATE_SIGNAL));
208         }
209
210         continue;
211     }
212
213     if (ngx_quit) {
214         ngx_signal_worker_processes(cycle,
215                                   ngx_signal_value(NGX_SHUTDOWN_SIGNAL));
216
217         ls = cycle->listening.elts;
218         for (n = 0; n < cycle->listening.nelts; n++) {
219             if (ngx_close_socket(ls[n].fd) == -1) {
220                 ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_socket_errno,
221                             ngx_close_socket_n " %V failed",
222                             &ls[n].addr_text);
223             }
224         }
225         cycle->listening.nelts = 0;
226
227         continue;
228     }
229
230     if (ngx_reconfigure) {
231         ngx_reconfigure = 0;
232
233         if (ngx_new_binary) {
234             ngx_start_worker_processes(cycle, ccf->worker_processes,
235                                     NGX_PROCESS_RESPAWN);
236             ngx_start_cache_manager_processes(cycle, 0);
237             ngx_noaccepting = 0;
238
239             continue;
240         }
241
242         ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "reconfiguring");
243
244         cycle = ngx_init_cycle(cycle);
245         if (cycle == NULL) {
246             cycle = (ngx_cycle_t *) ngx_cycle;
247             continue;
248         }
249
250         ngx_cycle = cycle;
251         ccf = (ngx_core_conf_t *) ngx_get_conf(cycle->conf_ctx,
252                                               ngx_core_module);
253         ngx_start_worker_processes(cycle, ccf->worker_processes,
254                                 NGX_PROCESS_JUST_RESPAWN);
255         ngx_start_cache_manager_processes(cycle, 1);
256
257         /* allow new processes to start */
258         ngx_msleep(100);
259
260         live = 1;
261         ngx_signal_worker_processes(cycle,
262                                   ngx_signal_value(NGX_SHUTDOWN_SIGNAL));
263     }
264

```



```

265     if (ngx_restart) {
266         ngx_restart = 0;
267         ngx_start_worker_processes(cycle, ccf->worker_processes,
268                                 NGX_PROCESS_RESPAWN);
269         ngx_start_cache_manager_processes(cycle, 0);
270         live = 1;
271     }
272
273     if (ngx_reopen) {
274         ngx_reopen = 0;
275         ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "reopening logs");
276         ngx_reopen_files(cycle, ccf->user);
277         ngx_signal_worker_processes(cycle,
278                                 ngx_signal_value(NGX_REOPEN_SIGNAL));
279     }
280
281     if (ngx_change_binary) {
282         ngx_change_binary = 0;
283         ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "changing binary");
284         ngx_new_binary = ngx_exec_new_binary(cycle, ngx_argv);
285     }
286
287     if (ngx_noaccept) {
288         ngx_noaccept = 0;
289         ngx_noaccepting = 1;
290         ngx_signal_worker_processes(cycle,
291                                 ngx_signal_value(NGX_SHUTDOWN_SIGNAL));
292     }
293 }
294 }
295
296 void
297 ngx_single_process_cycle(ngx_cycle_t *cycle)
298 {
299     ngx_uint_t i;
300
301     if (ngx_set_environment(cycle, NULL) == NULL) {
302         /* fatal */
303         exit(2);
304     }
305
306     for (i = 0; ngx_modules[i]; i++) {
307         if (ngx_modules[i]->init_process) {
308             if (ngx_modules[i]->init_process(cycle) == NGX_ERROR) {
309                 /* fatal */
310                 exit(2);
311             }
312         }
313     }
314 }
315
316 for ( ;; ) {
317     ngx_log_debug0(NGX_LOG_DEBUG_EVENT, cycle->log, 0, "worker cycle");
318
319     ngx_process_events_and_timers(cycle);
320
321     if (ngx_terminate || ngx_quit) {
322
323         for (i = 0; ngx_modules[i]; i++) {
324             if (ngx_modules[i]->exit_process) {
325                 ngx_modules[i]->exit_process(cycle);
326             }
327         }
328
329         ngx_master_process_exit(cycle);
330     }
331
332     if (ngx_reconfigure) {
333         ngx_reconfigure = 0;
334         ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "reconfiguring");
335
336         cycle = ngx_init_cycle(cycle);
337         if (cycle == NULL) {
338             cycle = (ngx_cycle_t *) ngx_cycle;
339             continue;
340         }

```

```

341     ngx_cycle = cycle;
342 }
343 }
344
345 if (ngx_reopen) {
346     ngx_reopen = 0;
347     ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "reopening logs");
348     ngx_reopen_files(cycle, (ngx_uid_t) -1);
349 }
350 }
351 }
352
353
354 static void
355 ngx_start_worker_processes(ngx_cycle_t *cycle, ngx_int_t n, ngx_int_t type)
356 {
357     ngx_int_t    i;
358     ngx_channel_t ch;
359
360     ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "start worker processes");
361
362     ngx_memzero(&ch, sizeof(ngx_channel_t));
363
364     ch.command = NGX_CMD_OPEN_CHANNEL;
365
366     for (i = 0; i < n; i++) {
367
368         ngx_spawn_process(cycle, ngx_worker_process_cycle,
369             (void *) (intptr_t) i, "worker process", type);
370
371         ch.pid = ngx_processes[ngx_process_slot].pid;
372         ch.slot = ngx_process_slot;
373         ch.fd = ngx_processes[ngx_process_slot].channel[0];
374
375         ngx_pass_open_channel(cycle, &ch);
376     }
377 }
378
379
380 static void
381 ngx_start_cache_manager_processes(ngx_cycle_t *cycle, ngx_uint_t respawn)
382 {
383     ngx_uint_t    i, manager, loader;
384     ngx_path_t    **path;
385     ngx_channel_t ch;
386
387     manager = 0;
388     loader = 0;
389
390     path = ngx_cycle->paths.elts;
391     for (i = 0; i < ngx_cycle->paths.nelts; i++) {
392
393         if (path[i]->manager) {
394             manager = 1;
395         }
396
397         if (path[i]->loader) {
398             loader = 1;
399         }
400     }
401
402     if (manager == 0) {
403         return;
404     }
405
406     ngx_spawn_process(cycle, ngx_cache_manager_process_cycle,
407         &ngx_cache_manager_ctx, "cache manager process",
408         respawn ? NGX_PROCESS_JUST_RESPAWN : NGX_PROCESS_RESPAWN);
409
410     ngx_memzero(&ch, sizeof(ngx_channel_t));
411
412     ch.command = NGX_CMD_OPEN_CHANNEL;
413     ch.pid = ngx_processes[ngx_process_slot].pid;
414     ch.slot = ngx_process_slot;
415     ch.fd = ngx_processes[ngx_process_slot].channel[0];
416

```

```

417     ngx_pass_open_channel(cycle, &ch);
418
419     if (loader == 0) {
420         return;
421     }
422
423     ngx_spawn_process(cycle, ngx_cache_manager_process_cycle,
424                     &ngx_cache_loader_ctx, "cache loader process",
425                     respawn ? NGX_PROCESS_JUST_SPAWN : NGX_PROCESS_NORESPAWN);
426
427     ch.command = NGX_CMD_OPEN_CHANNEL;
428     ch.pid = ngx_processes[ngx_process_slot].pid;
429     ch.slot = ngx_process_slot;
430     ch.fd = ngx_processes[ngx_process_slot].channel[0];
431
432     ngx_pass_open_channel(cycle, &ch);
433 }
434
435
436 static void
437 ngx_pass_open_channel(ngx_cycle_t *cycle, ngx_channel_t *ch)
438 {
439     ngx_int_t i;
440
441     for (i = 0; i < ngx_last_process; i++) {
442
443         if (i == ngx_process_slot
444             || ngx_processes[i].pid == -1
445             || ngx_processes[i].channel[0] == -1)
446         {
447             continue;
448         }
449
450         ngx_log_debug6(NGX_LOG_DEBUG_CORE, cycle->log, 0,
451                      "pass channel s:%d pid:%P fd:%d to s:%i pid:%P fd:%d",
452                      ch->slot, ch->pid, ch->fd,
453                      i, ngx_processes[i].pid,
454                      ngx_processes[i].channel[0]);
455
456         /* TODO: NGX_Again */
457
458         ngx_write_channel(ngx_processes[i].channel[0],
459                          ch, sizeof(ngx_channel_t), cycle->log);
460     }
461 }
462
463
464 static void
465 ngx_signal_worker_processes(ngx_cycle_t *cycle, int signo)
466 {
467     ngx_int_t i;
468     ngx_err_t err;
469     ngx_channel_t ch;
470
471     ngx_memzero(&ch, sizeof(ngx_channel_t));
472
473     #if (NGX_BROKEN_SCM_RIGHTS)
474
475     ch.command = 0;
476
477     #else
478
479     switch (signo) {
480
481     case ngx_signal_value(NGX_SHUTDOWN_SIGNAL):
482         ch.command = NGX_CMD_QUIT;
483         break;
484
485     case ngx_signal_value(NGX_TERMINATE_SIGNAL):
486         ch.command = NGX_CMD_TERMINATE;
487         break;
488
489     case ngx_signal_value(NGX_REOPEN_SIGNAL):
490         ch.command = NGX_CMD_REOPEN;
491         break;
492

```

```

493     default:
494         ch.command = 0;
495     }
496
497 #endif
498
499     ch.fd = -1;
500
501
502     for (i = 0; i < ngx_last_process; i++) {
503
504         ngx_log_debug7(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
505             "child: %d %P e:%d t:%d d:%d r:%d j:%d",
506             i,
507             ngx_processes[i].pid,
508             ngx_processes[i].exiting,
509             ngx_processes[i].exited,
510             ngx_processes[i].detached,
511             ngx_processes[i].respawn,
512             ngx_processes[i].just_spawn);
513
514         if (ngx_processes[i].detached || ngx_processes[i].pid == -1) {
515             continue;
516         }
517
518         if (ngx_processes[i].just_spawn) {
519             ngx_processes[i].just_spawn = 0;
520             continue;
521         }
522
523         if (ngx_processes[i].exiting
524             && signo == ngx_signal_value(NGX_SHUTDOWN_SIGNAL))
525         {
526             continue;
527         }
528
529         if (ch.command) {
530             if (ngx_write_channel(ngx_processes[i].channel[0],
531                 &ch, sizeof(ngx_channel_t), cycle->log)
532                 == NGX_OK)
533             {
534                 if (signo != ngx_signal_value(NGX_REOPEN_SIGNAL)) {
535                     ngx_processes[i].exiting = 1;
536                 }
537
538                 continue;
539             }
540         }
541
542         ngx_log_debug2(NGX_LOG_DEBUG_CORE, cycle->log, 0,
543             "kill (%P, %d)", ngx_processes[i].pid, signo);
544
545         if (kill(ngx_processes[i].pid, signo) == -1) {
546             err = ngx_errno;
547             ngx_log_error(NGX_LOG_ALERT, cycle->log, err,
548                 "kill(%P, %d) failed", ngx_processes[i].pid, signo);
549
550             if (err == NGX_ESRCH) {
551                 ngx_processes[i].exited = 1;
552                 ngx_processes[i].exiting = 0;
553                 ngx_reap = 1;
554             }
555
556             continue;
557         }
558
559         if (signo != ngx_signal_value(NGX_REOPEN_SIGNAL)) {
560             ngx_processes[i].exiting = 1;
561         }
562     }
563 }
564
565
566 static ngx_uint_t
567 ngx_reap_children(ngx_cycle_t *cycle)
568 {

```

```

569 ngx_int_t i, n;
570 ngx_uint_t live;
571 ngx_channel_t ch;
572 ngx_core_conf_t *ccf;
573
574 ngx_memzero(&ch, sizeof(ngx_channel_t));
575
576 ch.command = NGX_CMD_CLOSE_CHANNEL;
577 ch.fd = -1;
578
579 live = 0;
580 for (i = 0; i < ngx_last_process; i++) {
581
582     ngx_log_debug7(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
583                 "child: %d %P e:%d t:%d d:%d r:%d j:%d",
584                 i,
585                 ngx_processes[i].pid,
586                 ngx_processes[i].exiting,
587                 ngx_processes[i].exited,
588                 ngx_processes[i].detached,
589                 ngx_processes[i].respawn,
590                 ngx_processes[i].just_spawn);
591
592     if (ngx_processes[i].pid == -1) {
593         continue;
594     }
595
596     if (ngx_processes[i].exited) {
597
598         if (!ngx_processes[i].detached) {
599             ngx_close_channel(ngx_processes[i].channel, cycle->log);
600
601             ngx_processes[i].channel[0] = -1;
602             ngx_processes[i].channel[1] = -1;
603
604             ch.pid = ngx_processes[i].pid;
605             ch.slot = i;
606
607             for (n = 0; n < ngx_last_process; n++) {
608                 if (ngx_processes[n].exited
609                     || ngx_processes[n].pid == -1
610                     || ngx_processes[n].channel[0] == -1)
611                 {
612                     continue;
613                 }
614
615                 ngx_log_debug3(NGX_LOG_DEBUG_CORE, cycle->log, 0,
616                             "pass close channel s:%i pid:%P to:%P",
617                             ch.slot, ch.pid, ngx_processes[n].pid);
618
619                 /* TODO: NGX_Again */
620
621                 ngx_write_channel(ngx_processes[n].channel[0],
622                                 &ch, sizeof(ngx_channel_t), cycle->log);
623             }
624         }
625
626         if (ngx_processes[i].respawn
627             && !ngx_processes[i].exiting
628             && !ngx_terminate
629             && !ngx_quit)
630         {
631             if (ngx_spawn_process(cycle, ngx_processes[i].proc,
632                                 ngx_processes[i].data,
633                                 ngx_processes[i].name, i)
634                 == NGX_INVALID_PID)
635             {
636                 ngx_log_error(NGX_LOG_ALERT, cycle->log, 0,
637                             "could not respawn %s",
638                             ngx_processes[i].name);
639                 continue;
640             }
641
642
643             ch.command = NGX_CMD_OPEN_CHANNEL;
644             ch.pid = ngx_processes[ngx_process_slot].pid;

```

```

645     ch.slot = ngx_process_slot;
646     ch.fd = ngx_processes[ngx_process_slot].channel[0];
647
648     ngx_pass_open_channel(cycle, &ch);
649
650     live = 1;
651
652     continue;
653 }
654
655 if (ngx_processes[i].pid == ngx_new_binary) {
656
657     ccf = (ngx_core_conf_t *) ngx_get_conf(cycle->conf_ctx,
658                                           ngx_core_module);
659
660     if (ngx_rename_file((char *) ccf->oldpid.data,
661                        (char *) ccf->pid.data)
662         == NGX_FILE_ERROR)
663     {
664         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
665                      ngx_rename_file_n " %s back to %s failed "
666                      "after the new binary process \"%s\" exited",
667                      ccf->oldpid.data, ccf->pid.data, ngx_argv[0]);
668     }
669
670     ngx_new_binary = 0;
671     if (ngx_noaccepting) {
672         ngx_restart = 1;
673         ngx_noaccepting = 0;
674     }
675 }
676
677 if (i == ngx_last_process - 1) {
678     ngx_last_process--;
679
680 } else {
681     ngx_processes[i].pid = -1;
682 }
683
684 } else if (ngx_processes[i].exiting || !ngx_processes[i].detached) {
685     live = 1;
686 }
687 }
688
689 return live;
690 }
691
692
693 static void
694 ngx_master_process_exit(ngx_cycle_t *cycle)
695 {
696     ngx_uint_t i;
697
698     ngx_delete_pidfile(cycle);
699
700     ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "exit");
701
702     for (i = 0; ngx_modules[i]; i++) {
703         if (ngx_modules[i]->exit_master) {
704             ngx_modules[i]->exit_master(cycle);
705         }
706     }
707
708     ngx_close_listening_sockets(cycle);
709
710     /*
711     * Copy ngx_cycle->log related data to the special static exit cycle,
712     * log, and log file structures enough to allow a signal handler to log.
713     * The handler may be called when standard ngx_cycle->log allocated from
714     * ngx_cycle->pool is already destroyed.
715     */
716
717     ngx_exit_log = *ngx_log_get_file_log(ngx_cycle->log);
718
719     ngx_exit_log.file.fd = ngx_exit_log.file->fd;

```

```

721     ngx_exit_log.file = &ngx_exit_log_file;
722     ngx_exit_log.next = NULL;
723     ngx_exit_log.writer = NULL;
724
725     ngx_exit_cycle.log = &ngx_exit_log;
726     ngx_exit_cycle.files = ngx_cycle->files;
727     ngx_exit_cycle.files_n = ngx_cycle->files_n;
728     ngx_cycle = &ngx_exit_cycle;
729
730     ngx_destroy_pool(cycle->pool);
731
732     exit(0);
733 }
734
735
736 static void
737 ngx_worker_process_cycle(ngx_cycle_t *cycle, void *data)
738 {
739     ngx_int_t worker = (intptr_t) data;
740
741     ngx_uint_t     i;
742     ngx_connection_t *c;
743
744     ngx_process = NGX_PROCESS_WORKER;
745
746     ngx_worker_process_init(cycle, worker);
747
748     ngx_setproctitle("worker process");
749
750 #if (NGX_THREADS)
751     {
752         ngx_int_t     n;
753         ngx_err_t     err;
754         ngx_core_conf_t *ccf;
755
756         ccf = (ngx_core_conf_t *) ngx_get_conf(cycle->conf_ctx, ngx_core_module);
757
758         if (ngx_threads_n) {
759             if (ngx_init_threads(ngx_threads_n, ccf->thread_stack_size, cycle)
760                 == NGX_ERROR)
761             {
762                 /* fatal */
763                 exit(2);
764             }
765
766             err = ngx_thread_key_create(&ngx_core_tls_key);
767             if (err != 0) {
768                 ngx_log_error(NGX_LOG_ALERT, cycle->log, err,
769                     ngx_thread_key_create_n " failed");
770                 /* fatal */
771                 exit(2);
772             }
773
774             for (n = 0; n < ngx_threads_n; n++) {
775
776                 ngx_threads[n].cv = ngx_cond_init(cycle->log);
777
778                 if (ngx_threads[n].cv == NULL) {
779                     /* fatal */
780                     exit(2);
781                 }
782
783                 if (ngx_create_thread((ngx_tid_t *) &ngx_threads[n].tid,
784                     ngx_worker_thread_cycle,
785                     (void *) &ngx_threads[n], cycle->log)
786                     != 0)
787                 {
788                     /* fatal */
789                     exit(2);
790                 }
791             }
792         }
793     }
794 #endif
795
796     for ( ;; ) {

```

```

797     if (ngx\_exiting) {
798
799         c = cycle->connections;
800
801         for (i = 0; i < cycle->connection_n; i++) {
802
803             /* THREAD: lock */
804
805             if (c[i].fd != -1 && c[i].idle) {
806                 c[i].close = 1;
807                 c[i].read->handler(c[i].read);
808             }
809         }
810
811         ngx\_event\_cancel\_timers();
812
813         if (ngx\_event\_timer\_rbtrees.root == ngx\_event\_timer\_rbtrees.sentinel)
814         {
815             ngx\_log\_error(NGX\_LOG\_NOTICE, cycle->log, 0, "exiting");
816
817             ngx\_worker\_process\_exit(cycle);
818         }
819     }
820
821     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_EVENT, cycle->log, 0, "worker cycle");
822
823     ngx\_process\_events\_and\_timers(cycle);
824
825     if (ngx\_terminate) {
826         ngx\_log\_error(NGX\_LOG\_NOTICE, cycle->log, 0, "exiting");
827
828         ngx\_worker\_process\_exit(cycle);
829     }
830
831     if (ngx\_quit) {
832         ngx\_quit = 0;
833         ngx\_log\_error(NGX\_LOG\_NOTICE, cycle->log, 0,
834             "gracefully shutting down");
835         ngx\_setproctitle("worker process is shutting down");
836
837         if (!ngx\_exiting) {
838             ngx\_close\_listening\_sockets(cycle);
839             ngx\_exiting = 1;
840         }
841     }
842
843     if (ngx\_reopen) {
844         ngx\_reopen = 0;
845         ngx\_log\_error(NGX\_LOG\_NOTICE, cycle->log, 0, "reopening logs");
846         ngx\_reopen\_files(cycle, -1);
847     }
848 }
849 }
850
851
852
853 static void
854 ngx\_worker\_process\_init(ngx\_cycle\_t *cycle, ngx\_int\_t worker)
855 {
856     sigset_t      set;
857     uint64_t      cpu_affinity;
858     ngx\_int\_t     n;
859     ngx\_uint\_t    i;
860     struct rlimit rlimit;
861     ngx\_core\_conf\_t *ccf;
862     ngx\_listening\_t *ls;
863
864     if (ngx\_set\_environment(cycle, NULL) == NULL) {
865         /* fatal */
866         exit(2);
867     }
868
869     ccf = (ngx\_core\_conf\_t *) ngx\_get\_conf(cycle->conf_ctx, ngx\_core\_module);
870
871     if (worker >= 0 && ccf->priority != 0) {
872         if (setpriority(PRIO_PROCESS, 0, ccf->priority) == -1) {

```



```

873     ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
874                 "setpriority(%d) failed", ccf->priority);
875 }
876 }
877
878 if (ccf->rlimit_nofile != NGX_CONF_UNSET) {
879     rlimt.rlim_cur = (rlim_t) ccf->rlimit_nofile;
880     rlimt.rlim_max = (rlim_t) ccf->rlimit_nofile;
881
882     if (setrlimit(RLIMIT_NOFILE, &rlimt) == -1) {
883         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
884                     "setrlimit(RLIMIT_NOFILE, %i) failed",
885                     ccf->rlimit_nofile);
886     }
887 }
888
889 if (ccf->rlimit_core != NGX_CONF_UNSET) {
890     rlimt.rlim_cur = (rlim_t) ccf->rlimit_core;
891     rlimt.rlim_max = (rlim_t) ccf->rlimit_core;
892
893     if (setrlimit(RLIMIT_CORE, &rlimt) == -1) {
894         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
895                     "setrlimit(RLIMIT_CORE, %0) failed",
896                     ccf->rlimit_core);
897     }
898 }
899
900 #ifndef RLIMIT_SIGPENDING
901     if (ccf->rlimit_sigpending != NGX_CONF_UNSET) {
902         rlimt.rlim_cur = (rlim_t) ccf->rlimit_sigpending;
903         rlimt.rlim_max = (rlim_t) ccf->rlimit_sigpending;
904
905         if (setrlimit(RLIMIT_SIGPENDING, &rlimt) == -1) {
906             ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
907                         "setrlimit(RLIMIT_SIGPENDING, %i) failed",
908                         ccf->rlimit_sigpending);
909         }
910     }
911 #endif
912
913 if (geteuid() == 0) {
914     if (setgid(ccf->group) == -1) {
915         ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
916                     "setgid(%d) failed", ccf->group);
917         /* fatal */
918         exit(2);
919     }
920
921     if (initgroups(ccf->username, ccf->group) == -1) {
922         ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
923                     "initgroups(%s, %d) failed",
924                     ccf->username, ccf->group);
925     }
926
927     if (setuid(ccf->user) == -1) {
928         ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
929                     "setuid(%d) failed", ccf->user);
930         /* fatal */
931         exit(2);
932     }
933 }
934
935 if (worker >= 0) {
936     cpu_affinity = ngx_get_cpu_affinity(worker);
937
938     if (cpu_affinity) {
939         ngx_setaffinity(cpu_affinity, cycle->log);
940     }
941 }
942
943 #if (NGX_HAVE_PR_SET_DUMPABLE)
944
945     /* allow coredump after setuid() in Linux 2.4.x */
946
947     if (prctl(PR_SET_DUMPABLE, 1, 0, 0, 0) == -1) {
948         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,

```

```

949         "prctl(PR_SET_DUMPABLE) failed");
950     }
951
952 #endif
953
954     if (ccf->working_directory.len) {
955         if (chdir((char *) ccf->working_directory.data) == -1) {
956             ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
957                 "chdir(\"%s\") failed", ccf->working_directory.data);
958             /* fatal */
959             exit(2);
960         }
961     }
962
963     sigemptyset(&set);
964
965     if (sigprocmask(SIG_SETMASK, &set, NULL) == -1) {
966         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
967             "sigprocmask() failed");
968     }
969
970     srandom((ngx_pid << 16) ^ ngx_time());
971
972     /*
973      * disable deleting previous events for the listening sockets because
974      * in the worker processes there are no events at all at this point
975      */
976     ls = cycle->listening.elts;
977     for (i = 0; i < cycle->listening.nelts; i++) {
978         ls[i].previous = NULL;
979     }
980
981     for (i = 0; ngx_modules[i]; i++) {
982         if (ngx_modules[i]->init_process) {
983             if (ngx_modules[i]->init_process(cycle) == NGX_ERROR) {
984                 /* fatal */
985                 exit(2);
986             }
987         }
988     }
989
990     for (n = 0; n < ngx_last_process; n++) {
991
992         if (ngx_processes[n].pid == -1) {
993             continue;
994         }
995
996         if (n == ngx_process_slot) {
997             continue;
998         }
999
1000         if (ngx_processes[n].channel[1] == -1) {
1001             continue;
1002         }
1003
1004         if (close(ngx_processes[n].channel[1]) == -1) {
1005             ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
1006                 "close() channel failed");
1007         }
1008     }
1009
1010     if (close(ngx_processes[ngx_process_slot].channel[0]) == -1) {
1011         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
1012             "close() channel failed");
1013     }
1014
1015 #if 0
1016     ngx_last_process = 0;
1017 #endif
1018
1019     if (ngx_add_channel_event(cycle, ngx_channel, NGX_READ_EVENT,
1020                             ngx_channel_handler)
1021         == NGX_ERROR)
1022     {
1023         /* fatal */
1024         exit(2);

```

```

1025     }
1026 }
1027
1028
1029 static void
1030 ngx_worker_process_exit(ngx_cycle_t *cycle)
1031 {
1032     ngx_uint_t     i;
1033     ngx_connection_t *c;
1034
1035     #if (NGX_THREADS)
1036         ngx_terminate = 1;
1037
1038         ngx_wakeup_worker_threads(cycle);
1039     #endif
1040
1041     for (i = 0; ngx_modules[i]; i++) {
1042         if (ngx_modules[i]->exit_process) {
1043             ngx_modules[i]->exit_process(cycle);
1044         }
1045     }
1046
1047     if (ngx_exiting) {
1048         c = cycle->connections;
1049         for (i = 0; i < cycle->connection_n; i++) {
1050             if (c[i].fd != -1
1051                 && c[i].read
1052                 && !c[i].read->accept
1053                 && !c[i].read->channel
1054                 && !c[i].read->resolver)
1055             {
1056                 ngx_log_error(NGX_LOG_ALERT, cycle->log, 0,
1057                     "**%uA open socket #%d left in connection %ui",
1058                     c[i].number, c[i].fd, i);
1059                 ngx_debug_quit = 1;
1060             }
1061         }
1062
1063         if (ngx_debug_quit) {
1064             ngx_log_error(NGX_LOG_ALERT, cycle->log, 0, "aborting");
1065             ngx_debug_point();
1066         }
1067     }
1068
1069     /*
1070     * Copy ngx_cycle->log related data to the special static exit cycle,
1071     * log, and log file structures enough to allow a signal handler to log.
1072     * The handler may be called when standard ngx_cycle->log allocated from
1073     * ngx_cycle->pool is already destroyed.
1074     */
1075
1076     ngx_exit_log = *ngx_log_get_file_log(ngx_cycle->log);
1077
1078     ngx_exit_log_file.fd = ngx_exit_log.file->fd;
1079     ngx_exit_log.file = &ngx_exit_log_file;
1080     ngx_exit_log.next = NULL;
1081     ngx_exit_log.writer = NULL;
1082
1083     ngx_exit_cycle.log = &ngx_exit_log;
1084     ngx_exit_cycle.files = ngx_cycle->files;
1085     ngx_exit_cycle.files_n = ngx_cycle->files_n;
1086     ngx_cycle = &ngx_exit_cycle;
1087
1088     ngx_destroy_pool(cycle->pool);
1089
1090     ngx_log_error(NGX_LOG_NOTICE, ngx_cycle->log, 0, "exit");
1091
1092     exit(0);
1093 }
1094
1095
1096 static void
1097 ngx_channel_handler(ngx_event_t *ev)
1098 {
1099     ngx_int_t     n;
1100     ngx_channel_t ch;

```

```

1101     ngx_connection_t *c;
1102
1103     if (ev->timedout) {
1104         ev->timedout = 0;
1105         return;
1106     }
1107
1108     c = ev->data;
1109
1110     ngx_log_debug0(NGX_LOG_DEBUG_CORE, ev->log, 0, "channel handler");
1111
1112     for ( ;; ) {
1113
1114         n = ngx_read_channel(c->fd, &ch, sizeof(ngx_channel_t), ev->log);
1115
1116         ngx_log_debug1(NGX_LOG_DEBUG_CORE, ev->log, 0, "channel: %i", n);
1117
1118         if (n == NGX_ERROR) {
1119
1120             if (ngx_event_flags & NGX_USE_EPOLL_EVENT) {
1121                 ngx_del_conn(c, 0);
1122             }
1123
1124             ngx_close_connection(c);
1125             return;
1126         }
1127
1128         if (ngx_event_flags & NGX_USE_EVENTPORT_EVENT) {
1129             if (ngx_add_event(ev, NGX_READ_EVENT, 0) == NGX_ERROR) {
1130                 return;
1131             }
1132         }
1133
1134         if (n == NGX_AGAIN) {
1135             return;
1136         }
1137
1138         ngx_log_debug1(NGX_LOG_DEBUG_CORE, ev->log, 0,
1139             "channel command: %d", ch.command);
1140
1141         switch (ch.command) {
1142
1143             case NGX_CMD_QUIT:
1144                 ngx_quit = 1;
1145                 break;
1146
1147             case NGX_CMD_TERMINATE:
1148                 ngx_terminate = 1;
1149                 break;
1150
1151             case NGX_CMD_REOPEN:
1152                 ngx_reopen = 1;
1153                 break;
1154
1155             case NGX_CMD_OPEN_CHANNEL:
1156
1157                 ngx_log_debug3(NGX_LOG_DEBUG_CORE, ev->log, 0,
1158                     "get channel s:%i pid:%P fd:%d",
1159                     ch.slot, ch.pid, ch.fd);
1160
1161                 ngx_processes[ch.slot].pid = ch.pid;
1162                 ngx_processes[ch.slot].channel[0] = ch.fd;
1163                 break;
1164
1165             case NGX_CMD_CLOSE_CHANNEL:
1166
1167                 ngx_log_debug4(NGX_LOG_DEBUG_CORE, ev->log, 0,
1168                     "close channel s:%i pid:%P our:%P fd:%d",
1169                     ch.slot, ch.pid, ngx_processes[ch.slot].pid,
1170                     ngx_processes[ch.slot].channel[0]);
1171
1172                 if (close(ngx_processes[ch.slot].channel[0]) == -1) {
1173                     ngx_log_error(NGX_LOG_ALERT, ev->log, ngx_errno,
1174                         "close() channel failed");
1175                 }
1176

```

```

1177         ngx_processes[ch.slot].channel[0] = -1;
1178         break;
1179     }
1180 }
1181 }
1182
1183
1184 #if (NGX_THREADS)
1185
1186 static void
1187 ngx_wakeup_worker_threads(ngx_cycle_t *cycle)
1188 {
1189     ngx_int_t    i;
1190     ngx_uint_t  live;
1191
1192     for ( ;; ) {
1193
1194         live = 0;
1195
1196         for (i = 0; i < ngx_threads_n; i++) {
1197             if (ngx_threads[i].state < NGX_THREAD_EXIT) {
1198                 if (ngx_cond_signal(ngx_threads[i].cv) == NGX_ERROR) {
1199                     ngx_threads[i].state = NGX_THREAD_DONE;
1200
1201                 } else {
1202                     live = 1;
1203                 }
1204             }
1205
1206             if (ngx_threads[i].state == NGX_THREAD_EXIT) {
1207                 ngx_thread_join(ngx_threads[i].tid, NULL);
1208                 ngx_threads[i].state = NGX_THREAD_DONE;
1209             }
1210         }
1211
1212         if (live == 0) {
1213             ngx_log_debug0(NGX_LOG_DEBUG_CORE, cycle->log, 0,
1214                 "all worker threads are joined");
1215
1216             /* STUB */
1217             ngx_done_events(cycle);
1218
1219             return;
1220         }
1221
1222         ngx_sched_yield();
1223     }
1224 }
1225
1226
1227 static ngx_thread_value_t
1228 ngx_worker_thread_cycle(void *data)
1229 {
1230     ngx_thread_t    *thr = data;
1231
1232     sigset_t        set;
1233     ngx_err_t       err;
1234     ngx_core_tls_t *tls;
1235     ngx_cycle_t     *cycle;
1236
1237     cycle = (ngx_cycle_t *) ngx_cycle;
1238
1239     sigemptyset(&set);
1240     sigaddset(&set, ngx_signal_value(NGX_RECONFIGURE_SIGNAL));
1241     sigaddset(&set, ngx_signal_value(NGX_REOPEN_SIGNAL));
1242     sigaddset(&set, ngx_signal_value(NGX_CHANGEBIN_SIGNAL));
1243
1244     err = ngx_thread_sigmask(SIG_BLOCK, &set, NULL);
1245     if (err) {
1246         ngx_log_error(NGX_LOG_ALERT, cycle->log, err,
1247             ngx_thread_sigmask_n " failed");
1248         return (ngx_thread_value_t) 1;
1249     }
1250
1251     ngx_log_debug1(NGX_LOG_DEBUG_CORE, cycle->log, 0,
1252         "thread " NGX_TID_T_FMT " started", ngx_thread_self());

```

```

1253     ngx_setthrtitle("worker thread");
1254
1255
1256     tls = ngx_alloc(sizeof(ngx_core_tls_t), cycle->log);
1257     if (tls == NULL) {
1258         return (ngx_thread_value_t) 1;
1259     }
1260
1261     err = ngx_thread_set_tls(ngx_core_tls_key, tls);
1262     if (err != 0) {
1263         ngx_log_error(NGX_LOG_ALERT, cycle->log, err,
1264             ngx_thread_set_tls_n " failed");
1265         return (ngx_thread_value_t) 1;
1266     }
1267
1268     for ( ;; ) {
1269         thr->state = NGX_THREAD_FREE;
1270
1271         #if 0
1272         if (ngx_cond_wait(thr->cv, ngx_posted_events_mutex) == NGX_ERROR) {
1273             return (ngx_thread_value_t) 1;
1274         }
1275         #endif
1276
1277         if (ngx_terminate) {
1278             thr->state = NGX_THREAD_EXIT;
1279
1280             ngx_log_debug1(NGX_LOG_DEBUG_CORE, cycle->log, 0,
1281                 "thread " NGX_TID_T_FMT " is done",
1282                 ngx_thread_self());
1283
1284             return (ngx_thread_value_t) 0;
1285         }
1286
1287         thr->state = NGX_THREAD_BUSY;
1288
1289         #if 0
1290         if (ngx_event_thread_process_posted(cycle) == NGX_ERROR) {
1291             return (ngx_thread_value_t) 1;
1292         }
1293
1294         if (ngx_event_thread_process_posted(cycle) == NGX_ERROR) {
1295             return (ngx_thread_value_t) 1;
1296         }
1297         #endif
1298
1299         if (ngx_process_changes) {
1300             if (ngx_process_changes(cycle, 1) == NGX_ERROR) {
1301                 return (ngx_thread_value_t) 1;
1302             }
1303         }
1304     }
1305 }
1306
1307 #endif
1308
1309
1310 static void
1311 ngx_cache_manager_process_cycle(ngx_cycle_t *cycle, void *data)
1312 {
1313     ngx_cache_manager_ctx_t *ctx = data;
1314
1315     void *ident[4];
1316     ngx_event_t ev;
1317
1318     /*
1319     * Set correct process type since closing listening Unix domain socket
1320     * in a master process also removes the Unix domain socket file.
1321     */
1322     ngx_process = NGX_PROCESS_HELPER;
1323
1324     ngx_close_listening_sockets(cycle);
1325
1326     /* Set a moderate number of connections for a helper process. */
1327     cycle->connection_n = 512;
1328

```

```

1329     ngx\_worker\_process\_init(cycle, -1);
1330
1331     ngx\_memzero(&ev, sizeof(ngx\_event\_t));
1332     ev.handler = ctx->handler;
1333     ev.data = ident;
1334     ev.log = cycle->log;
1335     ident[3] = (void *) -1;
1336
1337     ngx\_use\_accept\_mutex = 0;
1338
1339     ngx\_setproctitle(ctx->name);
1340
1341     ngx\_add\_timer(&ev, ctx->delay);
1342
1343     for ( ;; ) {
1344
1345         if (ngx\_terminate || ngx\_quit) {
1346             ngx\_log\_error(NGX\_LOG\_NOTICE, cycle->log, 0, "exiting");
1347             exit(0);
1348         }
1349
1350         if (ngx\_reopen) {
1351             ngx\_reopen = 0;
1352             ngx\_log\_error(NGX\_LOG\_NOTICE, cycle->log, 0, "reopening logs");
1353             ngx\_reopen\_files(cycle, -1);
1354         }
1355
1356         ngx\_process\_events\_and\_timers(cycle);
1357     }
1358 }
1359
1360
1361 static void
1362 ngx\_cache\_manager\_process\_handler(ngx\_event\_t *ev)
1363 {
1364     time\_t         next, n;
1365     ngx\_uint\_t     i;
1366     ngx\_path\_t    **path;
1367
1368     next = 60 * 60;
1369
1370     path = ngx\_cycle->paths.elts;
1371     for (i = 0; i < ngx\_cycle->paths.nelts; i++) {
1372
1373         if (path[i]->manager) {
1374             n = path[i]->manager(path[i]->data);
1375
1376             next = (n <= next) ? n : next;
1377
1378             ngx\_time\_update();
1379         }
1380     }
1381
1382     if (next == 0) {
1383         next = 1;
1384     }
1385
1386     ngx\_add\_timer(ev, next * 1000);
1387 }
1388
1389
1390 static void
1391 ngx\_cache\_loader\_process\_handler(ngx\_event\_t *ev)
1392 {
1393     ngx\_uint\_t     i;
1394     ngx\_path\_t    **path;
1395     ngx\_cycle\_t   *cycle;
1396
1397     cycle = (ngx\_cycle\_t *) ngx\_cycle;
1398
1399     path = cycle->paths.elts;
1400     for (i = 0; i < cycle->paths.nelts; i++) {
1401
1402         if (ngx\_terminate || ngx\_quit) {
1403             break;
1404         }

```

```
1405
1406     if (path[i]->loader) {
1407         path[i]->loader(path[i]->data);
1408         ngx\_time\_update\(\);
1409     }
1410 }
1411
1412 exit(0);
1413 }
```

[One Level Up](#)

[Top Level](#)



## src/os/unix/nginx\_process.c - nginx-1.7.10

### Global variables defined

- [ngx\\_argc](#)
- [ngx\\_argv](#)
- [ngx\\_channel](#)
- [ngx\\_last\\_process](#)
- [ngx\\_os\\_argv](#)
- [ngx\\_process\\_slot](#)
- [ngx\\_processes](#)
- [signals](#)

### Data types defined

- [ngx\\_signal\\_t](#)

### Functions defined

- [ngx\\_debug\\_point](#)
- [ngx\\_execute](#)
- [ngx\\_execute\\_proc](#)
- [ngx\\_init\\_signals](#)
- [ngx\\_os\\_signal\\_process](#)
- [ngx\\_process\\_get\\_status](#)
- [ngx\\_signal\\_handler](#)
- [ngx\\_spawn\\_process](#)
- [ngx\\_unlock\\_mutexes](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11 #include <ngx_channel.h>
12
13
14 typedef struct {
15     int     signo;
16     char   *signame;
17     char   *name;
```

```

18     void (*handler)(int signo);
19 } ngx_signal_t;
20
21
22
23 static void ngx_execute_proc(ngx_cycle_t *cycle, void *data);
24 static void ngx_signal_handler(int signo);
25 static void ngx_process_get_status(void);
26 static void ngx_unlock_mutexes(ngx_pid_t pid);
27
28
29 int         ngx_argc;
30 char        **ngx_argv;
31 char        **ngx_os_argv;
32
33 ngx_int_t    ngx_process_slot;
34 ngx_socket_t ngx_channel;
35 ngx_int_t    ngx_last_process;
36 ngx_process_t ngx_processes[NGX_MAX_PROCESSES];
37
38
39 ngx_signal_t signals[] = {
40     { ngx_signal_value(NGX_RECONFIGURE_SIGNAL),
41       "SIG" ngx_value(NGX_RECONFIGURE_SIGNAL),
42       "reload",
43       ngx_signal_handler },
44
45     { ngx_signal_value(NGX_REOPEN_SIGNAL),
46       "SIG" ngx_value(NGX_REOPEN_SIGNAL),
47       "reopen",
48       ngx_signal_handler },
49
50     { ngx_signal_value(NGX_NOACCEPT_SIGNAL),
51       "SIG" ngx_value(NGX_NOACCEPT_SIGNAL),
52       "",
53       ngx_signal_handler },
54
55     { ngx_signal_value(NGX_TERMINATE_SIGNAL),
56       "SIG" ngx_value(NGX_TERMINATE_SIGNAL),
57       "stop",
58       ngx_signal_handler },
59
60     { ngx_signal_value(NGX_SHUTDOWN_SIGNAL),
61       "SIG" ngx_value(NGX_SHUTDOWN_SIGNAL),
62       "quit",
63       ngx_signal_handler },
64
65     { ngx_signal_value(NGX_CHANGEBIN_SIGNAL),
66       "SIG" ngx_value(NGX_CHANGEBIN_SIGNAL),
67       "",
68       ngx_signal_handler },
69
70     { SIGALRM, "SIGALRM", "", ngx_signal_handler },
71
72     { SIGINT, "SIGINT", "", ngx_signal_handler },
73
74     { SIGIO, "SIGIO", "", ngx_signal_handler },
75
76     { SIGCHLD, "SIGCHLD", "", ngx_signal_handler },
77
78     { SIGSYS, "SIGSYS, SIG_IGN", "", SIG_IGN },
79
80     { SIGPIPE, "SIGPIPE, SIG_IGN", "", SIG_IGN },
81
82     { 0, NULL, "", NULL }
83 };
84
85
86 ngx_pid_t
87 ngx_spawn_process(ngx_cycle_t *cycle, ngx_spawn_proc_pt proc, void *data,
88                  char *name, ngx_int_t respawn)
89 {
90     u_long    on;
91     ngx_pid_t pid;
92     ngx_int_t s;
93

```

```

94  if (respawn >= 0) {
95      s = respawn;
96
97  } else {
98      for (s = 0; s < ngx_last_process; s++) {
99          if (ngx_processes[s].pid == -1) {
100             break;
101         }
102     }
103
104     if (s == NGX_MAX_PROCESSES) {
105         ngx_log_error(NGX_LOG_ALERT, cycle->log, 0,
106             "no more than %d processes can be spawned",
107             NGX_MAX_PROCESSES);
108         return NGX_INVALID_PID;
109     }
110 }
111
112
113 if (respawn != NGX_PROCESS_DETACHED) {
114
115     /* Solaris 9 still has no AF_LOCAL */
116
117     if (socketpair(AF_UNIX, SOCK_STREAM, 0, ngx_processes[s].channel) == -1)
118     {
119         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
120             "socketpair() failed while spawning \"%s\"", name);
121         return NGX_INVALID_PID;
122     }
123
124     ngx_log_debug2(NGX_LOG_DEBUG_CORE, cycle->log, 0,
125         "channel %d:%d",
126         ngx_processes[s].channel[0],
127         ngx_processes[s].channel[1]);
128
129     if (ngx_nonblocking(ngx_processes[s].channel[0]) == -1) {
130         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
131             ngx_nonblocking_n " failed while spawning \"%s\"",
132             name);
133         ngx_close_channel(ngx_processes[s].channel, cycle->log);
134         return NGX_INVALID_PID;
135     }
136
137     if (ngx_nonblocking(ngx_processes[s].channel[1]) == -1) {
138         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
139             ngx_nonblocking_n " failed while spawning \"%s\"",
140             name);
141         ngx_close_channel(ngx_processes[s].channel, cycle->log);
142         return NGX_INVALID_PID;
143     }
144
145     on = 1;
146     if (ioctl(ngx_processes[s].channel[0], FIOASYNC, &on) == -1) {
147         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
148             "ioctl(FIOASYNC) failed while spawning \"%s\"", name);
149         ngx_close_channel(ngx_processes[s].channel, cycle->log);
150         return NGX_INVALID_PID;
151     }
152
153     if (fcntl(ngx_processes[s].channel[0], F_SETOWN, ngx_pid) == -1) {
154         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
155             "fcntl(F_SETOWN) failed while spawning \"%s\"", name);
156         ngx_close_channel(ngx_processes[s].channel, cycle->log);
157         return NGX_INVALID_PID;
158     }
159
160     if (fcntl(ngx_processes[s].channel[0], F_SETFD, FD_CLOEXEC) == -1) {
161         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
162             "fcntl(FD_CLOEXEC) failed while spawning \"%s\"",
163             name);
164         ngx_close_channel(ngx_processes[s].channel, cycle->log);
165         return NGX_INVALID_PID;
166     }
167
168     if (fcntl(ngx_processes[s].channel[1], F_SETFD, FD_CLOEXEC) == -1) {
169         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,

```

```

170         "fcntl(FD_CLOEXEC) failed while spawning \"%s\"",
171         name);
172         ngx_close_channel(ngx_processes[s].channel, cycle->log);
173         return NGX_INVALID_PID;
174     }
175
176     ngx_channel = ngx_processes[s].channel[1];
177
178 } else {
179     ngx_processes[s].channel[0] = -1;
180     ngx_processes[s].channel[1] = -1;
181 }
182
183 ngx_process_slot = s;
184
185
186 pid = fork();
187
188 switch (pid) {
189
190 case -1:
191     ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
192                 "fork() failed while spawning \"%s\"", name);
193     ngx_close_channel(ngx_processes[s].channel, cycle->log);
194     return NGX_INVALID_PID;
195
196 case 0:
197     ngx_pid = ngx_getpid();
198     proc(cycle, data);
199     break;
200
201 default:
202     break;
203 }
204
205 ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "start %s %P", name, pid);
206
207 ngx_processes[s].pid = pid;
208 ngx_processes[s].exited = 0;
209
210 if (respawn >= 0) {
211     return pid;
212 }
213
214 ngx_processes[s].proc = proc;
215 ngx_processes[s].data = data;
216 ngx_processes[s].name = name;
217 ngx_processes[s].exiting = 0;
218
219 switch (respawn) {
220
221 case NGX_PROCESS_NORESPAWN:
222     ngx_processes[s].respawn = 0;
223     ngx_processes[s].just_spawn = 0;
224     ngx_processes[s].detached = 0;
225     break;
226
227 case NGX_PROCESS_JUST_SPAWN:
228     ngx_processes[s].respawn = 0;
229     ngx_processes[s].just_spawn = 1;
230     ngx_processes[s].detached = 0;
231     break;
232
233 case NGX_PROCESS_RESPAWN:
234     ngx_processes[s].respawn = 1;
235     ngx_processes[s].just_spawn = 0;
236     ngx_processes[s].detached = 0;
237     break;
238
239 case NGX_PROCESS_JUST_RESPAWN:
240     ngx_processes[s].respawn = 1;
241     ngx_processes[s].just_spawn = 1;
242     ngx_processes[s].detached = 0;
243     break;
244
245 case NGX_PROCESS_DETACHED:

```

```

246     ngx_processes[s].respawn = 0;
247     ngx_processes[s].just_spawn = 0;
248     ngx_processes[s].detached = 1;
249     break;
250 }
251
252 if (s == ngx_last_process) {
253     ngx_last_process++;
254 }
255
256 return pid;
257 }
258
259
260 ngx_pid_t
261 ngx_execute(ngx_cycle_t *cycle, ngx_exec_ctx_t *ctx)
262 {
263     return ngx_spawn_process(cycle, ngx_execute_proc, ctx, ctx->name,
264                             NGX_PROCESS_DETACHED);
265 }
266
267
268 static void
269 ngx_execute_proc(ngx_cycle_t *cycle, void *data)
270 {
271     ngx_exec_ctx_t *ctx = data;
272
273     if (execve(ctx->path, ctx->argv, ctx->envp) == -1) {
274         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
275                     "execve() failed while executing %s \"%s\"",
276                     ctx->name, ctx->path);
277     }
278
279     exit(1);
280 }
281
282
283 ngx_int_t
284 ngx_init_signals(ngx_log_t *log)
285 {
286     ngx_signal_t *sig;
287     struct sigaction sa;
288
289     for (sig = signals; sig->signo != 0; sig++) {
290         ngx_memzero(&sa, sizeof(struct sigaction));
291         sa.sa_handler = sig->handler;
292         sigemptyset(&sa.sa_mask);
293         if (sigaction(sig->signo, &sa, NULL) == -1) {
294 #if (NGX_VALGRIND)
295             ngx_log_error(NGX_LOG_ALERT, log, ngx_errno,
296                         "sigaction(%s) failed, ignored", sig->signame);
297 #else
298             ngx_log_error(NGX_LOG_EMERG, log, ngx_errno,
299                         "sigaction(%s) failed", sig->signame);
300             return NGX_ERROR;
301 #endif
302         }
303     }
304
305     return NGX_OK;
306 }
307
308
309 void
310 ngx_signal_handler(int signo)
311 {
312     char *action;
313     ngx_int_t ignore;
314     ngx_err_t err;
315     ngx_signal_t *sig;
316
317     ignore = 0;
318
319     err = ngx_errno;
320
321     for (sig = signals; sig->signo != 0; sig++) {

```

```

322     if (sig->signo == signo) {
323         break;
324     }
325 }
326
327 ngx_time sigsafe update();
328
329 action = "";
330
331 switch (ngx_process) {
332
333 case NGX_PROCESS_MASTER:
334 case NGX_PROCESS_SINGLE:
335     switch (signo) {
336
337     case ngx_signal_value(NGX_SHUTDOWN_SIGNAL):
338         ngx_quit = 1;
339         action = ", shutting down";
340         break;
341
342     case ngx_signal_value(NGX_TERMINATE_SIGNAL):
343     case SIGINT:
344         ngx_terminate = 1;
345         action = ", exiting";
346         break;
347
348     case ngx_signal_value(NGX_NOACCEPT_SIGNAL):
349         if (ngx_daemonized) {
350             ngx_noaccept = 1;
351             action = ", stop accepting connections";
352         }
353         break;
354
355     case ngx_signal_value(NGX_RECONFIGURE_SIGNAL):
356         ngx_reconfigure = 1;
357         action = ", reconfiguring";
358         break;
359
360     case ngx_signal_value(NGX_REOPEN_SIGNAL):
361         ngx_reopen = 1;
362         action = ", reopening logs";
363         break;
364
365     case ngx_signal_value(NGX_CHANGEBIN_SIGNAL):
366         if (getppid() > 1 || ngx_new_binary > 0) {
367
368             /*
369              * Ignore the signal in the new binary if its parent is
370              * not the init process, i.e. the old binary's process
371              * is still running. Or ignore the signal in the old binary's
372              * process if the new binary's process is already running.
373              */
374
375             action = ", ignoring";
376             ignore = 1;
377             break;
378         }
379
380         ngx_change_binary = 1;
381         action = ", changing binary";
382         break;
383
384     case SIGALRM:
385         ngx_sigalrm = 1;
386         break;
387
388     case SIGIO:
389         ngx_sigio = 1;
390         break;
391
392     case SIGCHLD:
393         ngx_reap = 1;
394         break;
395     }
396
397     break;

```

```

398
399 case NGX_PROCESS_WORKER:
400 case NGX_PROCESS_HELPER:
401     switch (signo) {
402
403         case ngx_signal_value(NGX_NOACCEPT_SIGNAL):
404             if (!ngx_daemonized) {
405                 break;
406             }
407             ngx_debug_quit = 1;
408         case ngx_signal_value(NGX_SHUTDOWN_SIGNAL):
409             ngx_quit = 1;
410             action = ", shutting down";
411             break;
412
413         case ngx_signal_value(NGX_TERMINATE_SIGNAL):
414         case SIGINT:
415             ngx_terminate = 1;
416             action = ", exiting";
417             break;
418
419         case ngx_signal_value(NGX_REOPEN_SIGNAL):
420             ngx_reopen = 1;
421             action = ", reopening logs";
422             break;
423
424         case ngx_signal_value(NGX_RECONFIGURE_SIGNAL):
425         case ngx_signal_value(NGX_CHANGEBIN_SIGNAL):
426         case SIGIO:
427             action = ", ignoring";
428             break;
429     }
430
431     break;
432 }
433
434 ngx_log_error(NGX_LOG_NOTICE, ngx_cycle->log, 0,
435     "signal %d (%s) received%s", signo, sig->signame, action);
436
437 if (ignore) {
438     ngx_log_error(NGX_LOG_CRIT, ngx_cycle->log, 0,
439         "the changing binary signal is ignored: "
440         "you should shutdown or terminate "
441         "before either old or new binary's process");
442 }
443
444 if (signo == SIGCHLD) {
445     ngx_process_get_status();
446 }
447
448 ngx_set_errno(err);
449 }
450
451
452 static void
453 ngx_process_get_status(void)
454 {
455     int             status;
456     char            *process;
457     ngx_pid_t      pid;
458     ngx_err_t      err;
459     ngx_int_t      i;
460     ngx_uint_t     one;
461
462     one = 0;
463
464     for ( ;; ) {
465         pid = waitpid(-1, &status, WNOHANG);
466
467         if (pid == 0) {
468             return;
469         }
470
471         if (pid == -1) {
472             err = ngx_errno;
473

```

```

474     if (err == NGX\_EINTR) {
475         continue;
476     }
477
478     if (err == NGX\_ECHILD && one) {
479         return;
480     }
481
482     /*
483      * Solaris always calls the signal handler for each exited process
484      * despite waitpid() may be already called for this process.
485      *
486      * When several processes exit at the same time FreeBSD may
487      * erroneously call the signal handler for exited process
488      * despite waitpid() may be already called for this process.
489      */
490
491     if (err == NGX\_ECHILD) {
492         ngx\_log\_error(NGX\_LOG\_INFO, ngx\_cycle->log, err,
493                 "waitpid() failed");
494         return;
495     }
496
497     ngx\_log\_error(NGX\_LOG\_ALERT, ngx\_cycle->log, err,
498                 "waitpid() failed");
499     return;
500 }
501
502
503 one = 1;
504 process = "unknown process";
505
506 for (i = 0; i < ngx\_last\_process; i++) {
507     if (ngx\_processes[i].pid == pid) {
508         ngx\_processes[i].status = status;
509         ngx\_processes[i].exited = 1;
510         process = ngx\_processes[i].name;
511         break;
512     }
513 }
514
515 if (WTERMSIG(status)) {
516 #ifdef WCOREDUMP
517     ngx\_log\_error(NGX\_LOG\_ALERT, ngx\_cycle->log, 0,
518                 "%s %P exited on signal %d%s",
519                 process, pid, WTERMSIG(status),
520                 WCOREDUMP(status) ? " (core dumped)" : "");
521 #else
522     ngx\_log\_error(NGX\_LOG\_ALERT, ngx\_cycle->log, 0,
523                 "%s %P exited on signal %d",
524                 process, pid, WTERMSIG(status));
525 #endif
526
527 } else {
528     ngx\_log\_error(NGX\_LOG\_NOTICE, ngx\_cycle->log, 0,
529                 "%s %P exited with code %d",
530                 process, pid, WEXITSTATUS(status));
531 }
532
533 if (WEXITSTATUS(status) == 2 && ngx\_processes[i].respawn) {
534     ngx\_log\_error(NGX\_LOG\_ALERT, ngx\_cycle->log, 0,
535                 "%s %P exited with fatal code %d "
536                 "and cannot be respawned",
537                 process, pid, WEXITSTATUS(status));
538     ngx\_processes[i].respawn = 0;
539 }
540
541 ngx\_unlock\_mutexes(pid);
542 }
543 }
544
545
546 static void
547 ngx\_unlock\_mutexes(ngx\_pid\_t pid)
548 {
549     ngx\_uint\_t     i;

```



```

550     ngx_shm_zone_t *shm_zone;
551     ngx_list_part_t *part;
552     ngx_slab_pool_t *sp;
553
554     /*
555      * unlock the accept mutex if the abnormally exited process
556      * held it
557      */
558
559     if (ngx_accept_mutex_ptr) {
560         (void) ngx_shmtx_force_unlock(&ngx_accept_mutex, pid);
561     }
562
563     /*
564      * unlock shared memory mutexes if held by the abnormally exited
565      * process
566      */
567
568     part = (ngx_list_part_t *) &ngx_cycle->shared_memory.part;
569     shm_zone = part->elts;
570
571     for (i = 0; /* void */ ; i++) {
572
573         if (i >= part->nelts) {
574             if (part->next == NULL) {
575                 break;
576             }
577             part = part->next;
578             shm_zone = part->elts;
579             i = 0;
580         }
581
582         sp = (ngx_slab_pool_t *) shm_zone[i].shm.addr;
583
584         if (ngx_shmtx_force_unlock(&sp->mutex, pid)) {
585             ngx_log_error(NGX_LOG_ALERT, ngx_cycle->log, 0,
586                 "shared memory zone \"%V\" was locked by %P",
587                 &shm_zone[i].shm.name, pid);
588         }
589     }
590 }
591
592
593 void
594 ngx_debug_point(void)
595 {
596     ngx_core_conf_t *ccf;
597
598     ccf = (ngx_core_conf_t *) ngx_get_conf(ngx_cycle->conf_ctx,
599         ngx_core_module);
600
601     switch (ccf->debug_points) {
602
603     case NGX_DEBUG_POINTS_STOP:
604         raise(SIGSTOP);
605         break;
606
607     case NGX_DEBUG_POINTS_ABORT:
608         ngx_abort();
609     }
610 }
611
612
613 ngx_int_t
614 ngx_os_signal_process(ngx_cycle_t *cycle, char *name, ngx_int_t pid)
615 {
616     ngx_signal_t *sig;
617
618     for (sig = signals; sig->signo != 0; sig++) {
619         if (ngx_strcmp(name, sig->name) == 0) {
620             if (kill(pid, sig->signo) != -1) {
621                 return 0;
622             }
623
624             ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
625                 "kill(%P, %d) failed", pid, sig->signo);

```

```
626     }  
627   }  
628  
629   return 1;  
630 }
```

[One Level Up](#)

[Top Level](#)

## src/os/unix/nginx\_process.h - nginx-1.7.10

### Data types defined

- [ngx\\_exec\\_ctx\\_t](#)
- [ngx\\_pid\\_t](#)
- [ngx\\_process\\_t](#)
- [ngx\\_spawn\\_proc\\_pt](#)

### Macros defined

- [NGX\\_INVALID\\_PID](#)
- [NGX\\_MAX\\_PROCESSES](#)
- [NGX\\_PROCESS\\_DETACHED](#)
- [NGX\\_PROCESS\\_JUST\\_RESPAWN](#)
- [NGX\\_PROCESS\\_JUST\\_SPAWN](#)
- [NGX\\_PROCESS\\_NORESPAWN](#)
- [NGX\\_PROCESS\\_RESPAWN](#)
- [\\_NGX\\_PROCESS\\_H\\_INCLUDED\\_](#)
- [ngx\\_getpid](#)
- [ngx\\_log\\_pid](#)
- [ngx\\_sched\\_yield](#)
- [ngx\\_sched\\_yield](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_PROCESS_H_INCLUDED_
9 #define _NGX_PROCESS_H_INCLUDED_
10
11
12 #include <ngx_setaffinity.h>
13 #include <ngx_setproctitle.h>
14
15
16 typedef pid_t      ngx_pid_t;
17
18 #define NGX_INVALID_PID  -1
19
20 typedef void (*ngx_spawn_proc_pt) (ngx_cycle_t *cycle, void *data);
21
22 typedef struct {
23     ngx_pid_t      pid;
24     int             status;
25     ngx_socket_t   channel[2];

```

```

26
27     ngx\_spawn\_proc\_pt    proc;
28     void                *data;
29     char                *name;
30
31     unsigned            respawn:1;
32     unsigned            just_spawn:1;
33     unsigned            detached:1;
34     unsigned            exiting:1;
35     unsigned            exited:1;
36 } ngx\_process\_t;
37
38
39 typedef struct {
40     char                *path;
41     char                *name;
42     char *const         *argv;
43     char *const         *envp;
44 } ngx\_exec\_ctx\_t;
45
46
47 #define NGX\_MAX\_PROCESSES    1024
48
49 #define NGX\_PROCESS\_NORESPAWN    -1
50 #define NGX\_PROCESS\_JUST\_SPAWN    -2
51 #define NGX\_PROCESS\_RESPAWN    -3
52 #define NGX\_PROCESS\_JUST\_RESPAWN    -4
53 #define NGX\_PROCESS\_DETACHED    -5
54
55
56 #define ngx\_getpid    getpid
57
58 #ifndef ngx\_log\_pid
59 #define ngx\_log\_pid    ngx\_pid
60 #endif
61
62
63 ngx\_pid\_t ngx\_spawn\_process(ngx\_cycle\_t *cycle,
64     ngx\_spawn\_proc\_pt proc, void *data, char *name, ngx\_int\_t respawn);
65 ngx\_pid\_t ngx\_execute(ngx\_cycle\_t *cycle, ngx\_exec\_ctx\_t *ctx);
66 ngx\_int\_t ngx\_init\_signals(ngx\_log\_t *log);
67 void ngx\_debug\_point(void);
68
69
70 #if (NGX_HAVE_SCHED_YIELD)
71 #define ngx\_sched\_yield()    sched_yield()
72 #else
73 #define ngx\_sched\_yield()    usleep(1)
74 #endif
75
76
77 extern int                ngx\_argc;
78 extern char               **ngx\_argv;
79 extern char               **ngx\_os\_argv;
80
81 extern ngx\_pid\_t          ngx\_pid;
82 extern ngx\_socket\_t      ngx\_channel;
83 extern ngx\_int\_t         ngx\_process\_slot;
84 extern ngx\_int\_t         ngx\_last\_process;
85 extern ngx\_process\_t    ngx\_processes[NGX\_MAX\_PROCESSES];
86
87
88 #endif /* \_NGX\_PROCESS\_H\_INCLUDED */

```

[One Level Up](#)

[Top Level](#)

# src/os/unix/nginx\_setaffinity.c - nginx-1.7.10

## Functions defined

- [ngx\\_setaffinity](#)
- [ngx\\_setaffinity](#)

## Source code

```
1
2  /*
3  * Copyright (C) Nginx, Inc.
4  */
5
6
7  #include <ngx_config.h>
8  #include <ngx_core.h>
9
10
11  #if (NGX_HAVE_CPUSET_SETAFFINITY)
12
13  #include <sys/cpuset.h>
14
15  void
16  ngx_setaffinity(uint64_t cpu_affinity, ngx_log_t *log)
17  {
18      cpuset_t  mask;
19      ngx_uint_t  i;
20
21      ngx_log_error(NGX_LOG_NOTICE, log, 0,
22                  "cpuset_setaffinity(0x%08X1)", cpu_affinity);
23
24      CPU_ZERO(&mask);
25      i = 0;
26      do {
27          if (cpu_affinity & 1) {
28              CPU_SET(i, &mask);
29          }
30          i++;
31          cpu_affinity >>= 1;
32      } while (cpu_affinity);
33
34      if (cpuset_setaffinity(CPU_LEVEL_WHICH, CPU_WHICH_PID, -1,
35                          sizeof(cpuset_t), &mask) == -1)
36      {
37          ngx_log_error(NGX_LOG_ALERT, log, ngx_errno,
38                      "cpuset_setaffinity() failed");
39      }
40  }
41
42  #elif (NGX_HAVE_SCHED_SETAFFINITY)
43
44  void
45  ngx_setaffinity(uint64_t cpu_affinity, ngx_log_t *log)
46  {
47      cpu_set_t  mask;
48      ngx_uint_t  i;
49
50      ngx_log_error(NGX_LOG_NOTICE, log, 0,
51                  "sched_setaffinity(0x%08X1)", cpu_affinity);
52
53      CPU_ZERO(&mask);
54      i = 0;
55      do {
56          if (cpu_affinity & 1) {
57              CPU_SET(i, &mask);
58          }
59          i++;
60          cpu_affinity >>= 1;
```

```
61     } while (cpu_affinity);
62
63     if (sched_setaffinity(0, sizeof(cpu_set_t), &mask) == -1) {
64         ngx_log_error(NGX_LOG_ALERT, log, ngx_errno,
65                     "sched_setaffinity() failed");
66     }
67 }
68
69 #endif
```

[One Level Up](#)

[Top Level](#)

# src/os/unix/nginx\_setproctitle.h - nginx-1.7.10

## Macros defined

- [NGX\\_SETPROCTITLE\\_PAD](#)
- [NGX\\_SETPROCTITLE\\_PAD](#)
- [NGX\\_SETPROCTITLE\\_USES\\_ENV](#)
- [NGX\\_SETPROCTITLE\\_USES\\_ENV](#)
- [\\_NGX\\_SETPROCTITLE\\_H\\_INCLUDED](#)
- [ngx\\_init\\_setproctitle](#)
- [ngx\\_init\\_setproctitle](#)
- [ngx\\_setproctitle](#)
- [ngx\\_setproctitle](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_SETPROCTITLE_H_INCLUDED_
9 #define _NGX_SETPROCTITLE_H_INCLUDED_
10
11
12 #if (NGX_HAVE_SETPROCTITLE)
13
14 /* FreeBSD, NetBSD, OpenBSD */
15
16 #define ngx_init_setproctitle(log) NGX_OK
17 #define ngx_setproctitle(title) setproctitle("%s", title)
18
19
20 #else /* !NGX_HAVE_SETPROCTITLE */
21
22 #if !defined _NGX_SETPROCTITLE_USES_ENV
23
24 #if (NGX_SOLARIS)
25
26 #define NGX_SETPROCTITLE_USES_ENV 1
27 #define NGX_SETPROCTITLE_PAD ' '
28
29 ngx_int_t ngx_init_setproctitle(ngx_log_t *log);
30 void ngx_setproctitle(char *title);
31
32 #elif (NGX_LINUX) || (NGX_DARWIN)
33
34 #define NGX_SETPROCTITLE_USES_ENV 1
35 #define NGX_SETPROCTITLE_PAD '\0'
36
37 ngx_int_t ngx_init_setproctitle(ngx_log_t *log);
38 void ngx_setproctitle(char *title);
39
40 #else
41
42 #define ngx_init_setproctitle(log) NGX_OK
43 #define ngx_setproctitle(title)
44
```

```
45 #endif /* OSes */
46
47 #endif /* NGX_SETPROCTITLE_USES_ENV */
48
49 #endif /* NGX_HAVE_SETPROCTITLE */
50
51
52 #endif /* _NGX_SETPROCTITLE_H_INCLUDED_ */
```

[One Level Up](#)

[Top Level](#)



## src/os/unix/nginx\_channel.c - nginx-1.7.10

### Functions defined

- [ngx\\_add\\_channel\\_event](#)
- [ngx\\_close\\_channel](#)
- [ngx\\_read\\_channel](#)
- [ngx\\_write\\_channel](#)

### Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_channel.h>
11
12
13 ngx_int_t
14 ngx_write_channel(ngx_socket_t s, ngx_channel_t *ch, size_t size,
15                 ngx_log_t *log)
16 {
17     ssize_t      n;
18     ngx_err_t    err;
19     struct iovec  iov[1];
20     struct msghdr msg;
21
22     #if (NGX_HAVE_MSGHDR_MSG_CONTROL)
23
24     union {
25         struct cmsghdr cm;
26         char            space[CMSG\_SPACE(sizeof(int))];
27     } cmsg;
28
29     if (ch->fd == -1) {
30         msg.msg_control = NULL;
31         msg.msg_controllen = 0;
32
33     } else {
34         msg.msg_control = (caddr_t) &cmsg;
35         msg.msg_controllen = sizeof(cmsg);
36
37         ngx\_memzero(&cmsg, sizeof(cmsg));
38
39         cmsg.cm.msg_len = CMSG\_LEN(sizeof(int));
40         cmsg.cm.msg_level = SOL_SOCKET;
41         cmsg.cm.msg_type = SCM_RIGHTS;
42
43         /*
44          * We have to use ngx\_memcpy() instead of simple
45          * *(int *) CMSG\_DATA(&cmsg.cm) = ch->fd;
46          * because some gcc 4.4 with -O2/3/s optimization issues the warning:
47          * dereferencing type-punned pointer will break strict-aliasing rules
48          *
49          * Fortunately, gcc with -O1 compiles this ngx\_memcpy()
50          * in the same simple assignment as in the code above
51          */
52
53         ngx\_memcpy(CMSG\_DATA(&cmsg.cm), &ch->fd, sizeof(int));
54     }
55
56     msg.msg_flags = 0;
```

```

57
58 #else
59
60     if (ch->fd == -1) {
61         msg.msg_accrighths = NULL;
62         msg.msg_accrighthslen = 0;
63
64     } else {
65         msg.msg_accrighths = (caddr_t) &ch->fd;
66         msg.msg_accrighthslen = sizeof(int);
67     }
68
69 #endif
70
71     iov[0].iov_base = (char *) ch;
72     iov[0].iov_len = size;
73
74     msg.msg_name = NULL;
75     msg.msg_namelen = 0;
76     msg.msg_iov = iov;
77     msg.msg_iovlen = 1;
78
79     n = sendmsg(s, &msg, 0);
80
81     if (n == -1) {
82         err = ngx_errno;
83         if (err == NGX_EAGAIN) {
84             return NGX_AGAIN;
85         }
86
87         ngx_log_error(NGX_LOG_ALERT, log, err, "sendmsg() failed");
88         return NGX_ERROR;
89     }
90
91     return NGX_OK;
92 }
93
94
95 ngx_int_t
96 ngx_read_channel(ngx_socket_t s, ngx_channel_t *ch, size_t size, ngx_log_t *log)
97 {
98     ssize_t          n;
99     ngx_err_t        err;
100    struct iovec      iov[1];
101    struct msghdr     msg;
102
103    #if (NGX_HAVE_MSGHDR_MSG_CONTROL)
104        union {
105            struct cmsghdr cm;
106            char            space[CMSG_SPACE(sizeof(int))];
107        } cmsg;
108    #else
109        int            fd;
110    #endif
111
112    iov[0].iov_base = (char *) ch;
113    iov[0].iov_len = size;
114
115    msg.msg_name = NULL;
116    msg.msg_namelen = 0;
117    msg.msg_iov = iov;
118    msg.msg_iovlen = 1;
119
120    #if (NGX_HAVE_MSGHDR_MSG_CONTROL)
121        msg.msg_control = (caddr_t) &cmsg;
122        msg.msg_controllen = sizeof(cmsg);
123    #else
124        msg.msg_accrighths = (caddr_t) &fd;
125        msg.msg_accrighthslen = sizeof(int);
126    #endif
127
128    n = recvmsg(s, &msg, 0);
129
130    if (n == -1) {
131        err = ngx_errno;
132        if (err == NGX_EAGAIN) {

```

```

133     return NGX_AGAIN;
134 }
135
136 ngx_log_error(NGX_LOG_ALERT, log, err, "recvmsg() failed");
137 return NGX_ERROR;
138 }
139
140 if (n == 0) {
141     ngx_log_debug0(NGX_LOG_DEBUG_CORE, log, 0, "recvmsg() returned zero");
142     return NGX_ERROR;
143 }
144
145 if ((size_t) n < sizeof(ngx_channel_t)) {
146     ngx_log_error(NGX_LOG_ALERT, log, 0,
147         "recvmsg() returned not enough data: %z", n);
148     return NGX_ERROR;
149 }
150
151 #if (NGX_HAVE_MSGHDR_MSG_CONTROL)
152
153     if (ch->command == NGX_CMD_OPEN_CHANNEL) {
154
155         if (cmsg.cm.cmsg_len < (socklen_t) CMSG_LEN(sizeof(int))) {
156             ngx_log_error(NGX_LOG_ALERT, log, 0,
157                 "recvmsg() returned too small ancillary data");
158             return NGX_ERROR;
159         }
160
161         if (cmsg.cm.cmsg_level != SOL_SOCKET || cmsg.cm.cmsg_type != SCM_RIGHTS)
162         {
163             ngx_log_error(NGX_LOG_ALERT, log, 0,
164                 "recvmsg() returned invalid ancillary data "
165                 "level %d or type %d",
166                 cmsg.cm.cmsg_level, cmsg.cm.cmsg_type);
167             return NGX_ERROR;
168         }
169
170         /* ch->fd = *(int *) CMSG_DATA(&cmsg.cm); */
171
172         ngx_memcpy(&ch->fd, CMSG_DATA(&cmsg.cm), sizeof(int));
173     }
174
175     if (msg.msg_flags & (MSG_TRUNC|MSG_CTRUNC)) {
176         ngx_log_error(NGX_LOG_ALERT, log, 0,
177             "recvmsg() truncated data");
178     }
179
180 #else
181
182     if (ch->command == NGX_CMD_OPEN_CHANNEL) {
183         if (msg.msg_accrighslen != sizeof(int)) {
184             ngx_log_error(NGX_LOG_ALERT, log, 0,
185                 "recvmsg() returned no ancillary data");
186             return NGX_ERROR;
187         }
188
189         ch->fd = fd;
190     }
191
192 #endif
193
194     return n;
195 }
196
197 ngx_int_t
198 ngx_add_channel_event(ngx_cycle_t *cycle, ngx_fd_t fd, ngx_int_t event,
199     ngx_event_handler_pt handler)
200 {
201     {
202         ngx_event_t *ev, *rev, *wev;
203         ngx_connection_t *c;
204
205         c = ngx_get_connection(fd, cycle->log);
206
207         if (c == NULL) {
208             return NGX_ERROR;

```

```

209     }
210
211     c->pool = cycle->pool;
212
213     rev = c->read;
214     wev = c->write;
215
216     rev->log = cycle->log;
217     wev->log = cycle->log;
218
219     rev->channel = 1;
220     wev->channel = 1;
221
222     ev = (event == NGX\_READ\_EVENT) ? rev : wev;
223
224     ev->handler = handler;
225
226     if (ngx\_add\_conn && (ngx\_event\_flags & NGX\_USE\_EPOLL\_EVENT) == 0) {
227         if (ngx\_add\_conn(c) == NGX\_ERROR) {
228             ngx\_free\_connection(c);
229             return NGX\_ERROR;
230         }
231     } else {
232         if (ngx\_add\_event(ev, event, 0) == NGX\_ERROR) {
233             ngx\_free\_connection(c);
234             return NGX\_ERROR;
235         }
236     }
237 }
238
239 return NGX\_OK;
240 }
241
242
243 void
244 ngx\_close\_channel(ngx\_fd\_t *fd, ngx\_log\_t *log)
245 {
246     if (close(fd[0]) == -1) {
247         ngx\_log\_error(NGX\_LOG\_ALERT, log, ngx\_errno, "close() channel failed");
248     }
249
250     if (close(fd[1]) == -1) {
251         ngx\_log\_error(NGX\_LOG\_ALERT, log, ngx\_errno, "close() channel failed");
252     }
253 }

```

[One Level Up](#)

[Top Level](#)

## src/os/unix/nginx\_channel.h - nginx-1.7.10

### Data types defined

- [ngx\\_channel\\_t](#)

### Macros defined

- [\\_NGX\\_CHANNEL\\_H\\_INCLUDED](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_CHANNEL\_H\_INCLUDED
9 #define \_NGX\_CHANNEL\_H\_INCLUDED
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14 #include <ngx_event.h>
15
16
17 typedef struct {
18     ngx\_uint\_t    command;
19     ngx\_pid\_t    pid;
20     ngx\_int\_t    slot;
21     ngx\_fd\_t    fd;
22 } ngx\_channel\_t;
23
24
25 ngx\_int\_t ngx_write_channel(ngx\_socket\_t s, ngx\_channel\_t *ch, size\_t size,
26     ngx\_log\_t *log);
27 ngx\_int\_t ngx_read_channel(ngx\_socket\_t s, ngx\_channel\_t *ch, size\_t size,
28     ngx\_log\_t *log);
29 ngx\_int\_t ngx_add_channel_event(ngx\_cycle\_t *cycle, ngx\_fd\_t fd,
30     ngx\_int\_t event, ngx\_event\_handler\_pt handler);
31 void ngx\_close\_channel(ngx\_fd\_t *fd, ngx\_log\_t *log);
32
33
34 #endif /* \_NGX\_CHANNEL\_H\_INCLUDED */
```

# src/os/unix/nginx\_process\_cycle.h - nginx-1.7.10

## Data types defined

- [ngx\\_cache\\_manager\\_ctx\\_t](#)

## Macros defined

- [NGX\\_CMD\\_CLOSE\\_CHANNEL](#)
- [NGX\\_CMD\\_OPEN\\_CHANNEL](#)
- [NGX\\_CMD\\_QUIT](#)
- [NGX\\_CMD\\_REOPEN](#)
- [NGX\\_CMD\\_TERMINATE](#)
- [NGX\\_PROCESS\\_HELPER](#)
- [NGX\\_PROCESS\\_MASTER](#)
- [NGX\\_PROCESS\\_SIGNALLER](#)
- [NGX\\_PROCESS\\_SINGLE](#)
- [NGX\\_PROCESS\\_WORKER](#)
- [\\_NGX\\_PROCESS\\_CYCLE\\_H\\_INCLUDED](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_PROCESS_CYCLE_H_INCLUDED_
9 #define _NGX_PROCESS_CYCLE_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 #define NGX_CMD_OPEN_CHANNEL    1
17 #define NGX_CMD_CLOSE_CHANNEL  2
18 #define NGX_CMD_QUIT           3
19 #define NGX_CMD_TERMINATE      4
20 #define NGX_CMD_REOPEN         5
21
22
23 #define NGX_PROCESS_SINGLE      0
24 #define NGX_PROCESS_MASTER      1
25 #define NGX_PROCESS_SIGNALLER  2
26 #define NGX_PROCESS_WORKER     3
27 #define NGX_PROCESS_HELPER     4
28
29
30 typedef struct {
31     ngx_event_handler_pt    handler;
32     char                    *name;
33     ngx_msec_t              delay;
34 } ngx_cache_manager_ctx_t;
```

```
35
36
37 void ngx\_master\_process\_cycle(ngx\_cycle\_t *cycle);
38 void ngx\_single\_process\_cycle(ngx\_cycle\_t *cycle);
39
40
41 extern ngx\_uint\_t ngx\_process;
42 extern ngx\_pid\_t ngx\_pid;
43 extern ngx\_pid\_t ngx\_new\_binary;
44 extern ngx\_uint\_t ngx\_inherited;
45 extern ngx\_uint\_t ngx\_daemonized;
46 extern ngx\_uint\_t ngx\_threaded;
47 extern ngx\_uint\_t ngx\_exiting;
48
49 extern sig\_atomic\_t ngx\_reap;
50 extern sig\_atomic\_t ngx\_sigio;
51 extern sig\_atomic\_t ngx\_sigalrm;
52 extern sig\_atomic\_t ngx\_quit;
53 extern sig\_atomic\_t ngx\_debug\_quit;
54 extern sig\_atomic\_t ngx\_terminate;
55 extern sig\_atomic\_t ngx\_noaccept;
56 extern sig\_atomic\_t ngx\_reconfigure;
57 extern sig\_atomic\_t ngx\_reopen;
58 extern sig\_atomic\_t ngx\_change\_binary;
59
60
61 #endif /* \_NGX\_PROCESS\_CYCLE\_H\_INCLUDED */
```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_file.h - nginx-1.7.10

### Data types defined

- [ngx\\_copy\\_file\\_t](#)
- [ngx\\_ext\\_rename\\_file\\_t](#)
- [ngx\\_file\\_s](#)
- [ngx\\_path\\_init\\_t](#)
- [ngx\\_path\\_loader\\_pt](#)
- [ngx\\_path\\_manager\\_pt](#)
- [ngx\\_path\\_t](#)
- [ngx\\_temp\\_file\\_t](#)
- [ngx\\_tree\\_ctx\\_s](#)
- [ngx\\_tree\\_ctx\\_t](#)
- [ngx\\_tree\\_handler\\_pt](#)
- [ngx\\_tree\\_init\\_handler\\_pt](#)

### Macros defined

- [NGX\\_MAX\\_PATH\\_LEVEL](#)
- [\\_NGX\\_FILE\\_H\\_INCLUDED](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_FILE_H_INCLUDED
9 #define _NGX_FILE_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 struct ngx_file_s {
17     ngx_fd_t          fd;
18     ngx_str_t         name;
19     ngx_file_info_t   info;
20
21     off_t             offset;
22     off_t             sys_offset;
23
24     ngx_log_t         *log;
25
26     #if (NGX_HAVE_FILE_AIO)
27     ngx_event_aio_t   *aio;
28 #endif
29
```



```

30     unsigned                valid_info:1;
31     unsigned                directio:1;
32 };
33
34
35 #define NGX_MAX_PATH_LEVEL 3
36
37
38 typedef time_t (*ngx_path_manager_pt) (void *data);
39 typedef void (*ngx_path_loader_pt) (void *data);
40
41
42 typedef struct {
43     ngx_str_t                name;
44     size_t                  len;
45     size_t                  level[3];
46
47     ngx_path_manager_pt     manager;
48     ngx_path_loader_pt     loader;
49     void                    *data;
50
51     u_char                  *conf_file;
52     ngx_uint_t              line;
53 } ngx_path_t;
54
55
56 typedef struct {
57     ngx_str_t                name;
58     size_t                  level[3];
59 } ngx_path_init_t;
60
61
62 typedef struct {
63     ngx_file_t               file;
64     off_t                   offset;
65     ngx_path_t               *path;
66     ngx_pool_t               *pool;
67     char                    *warn;
68
69     ngx_uint_t               access;
70
71     unsigned                 log_level:8;
72     unsigned                 persistent:1;
73     unsigned                 clean:1;
74 } ngx_temp_file_t;
75
76
77 typedef struct {
78     ngx_uint_t               access;
79     ngx_uint_t               path_access;
80     time_t                  time;
81     ngx_fd_t                 fd;
82
83     unsigned                 create_path:1;
84     unsigned                 delete_file:1;
85
86     ngx_log_t                *log;
87 } ngx_ext_rename_file_t;
88
89
90 typedef struct {
91     off_t                   size;
92     size_t                  buf_size;
93
94     ngx_uint_t               access;
95     time_t                  time;
96
97     ngx_log_t                *log;
98 } ngx_copy_file_t;
99
100
101 typedef struct ngx_tree_ctx_s ngx_tree_ctx_t;
102
103 typedef ngx_int_t (*ngx_tree_init_handler_pt) (void *ctx, void *prev);
104 typedef ngx_int_t (*ngx_tree_handler_pt) (ngx_tree_ctx_t *ctx, ngx_str_t *name);
105

```

```

106 struct ngx_tree_ctx_s {
107     off_t          size;
108     off_t          fs_size;
109     ngx_uint_t     access;
110     time_t         mtime;
111
112     ngx_tree_init_handler_pt  init_handler;
113     ngx_tree_handler_pt       file_handler;
114     ngx_tree_handler_pt       pre_tree_handler;
115     ngx_tree_handler_pt       post_tree_handler;
116     ngx_tree_handler_pt       spec_handler;
117
118     void          *data;
119     size_t        alloc;
120
121     ngx_log_t     *log;
122 };
123
124
125 ngx_int_t ngx_get_full_name(ngx_pool_t *pool, ngx_str_t *prefix,
126     ngx_str_t *name);
127
128 ssize_t ngx_write_chain_to_temp_file(ngx_temp_file_t *tf, ngx_chain_t *chain);
129 ngx_int_t ngx_create_temp_file(ngx_file_t *file, ngx_path_t *path,
130     ngx_pool_t *pool, ngx_uint_t persistent, ngx_uint_t clean,
131     ngx_uint_t access);
132 void ngx_create_hashed_filename(ngx_path_t *path, u_char *file, size_t len);
133 ngx_int_t ngx_create_path(ngx_file_t *file, ngx_path_t *path);
134 ngx_err_t ngx_create_full_path(u_char *dir, ngx_uint_t access);
135 ngx_int_t ngx_add_path(ngx_conf_t *cf, ngx_path_t **slot);
136 ngx_int_t ngx_create_paths(ngx_cycle_t *cycle, ngx_uid_t user);
137 ngx_int_t ngx_ext_rename_file(ngx_str_t *src, ngx_str_t *to,
138     ngx_ext_rename_file_t *ext);
139 ngx_int_t ngx_copy_file(u_char *from, u_char *to, ngx_copy_file_t *cf);
140 ngx_int_t ngx_walk_tree(ngx_tree_ctx_t *ctx, ngx_str_t *tree);
141
142 ngx_atomic_uint_t ngx_next_temp_number(ngx_uint_t collision);
143
144 char *ngx_conf_set_path_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
145 char *ngx_conf_merge_path_value(ngx_conf_t *cf, ngx_path_t **path,
146     ngx_path_t *prev, ngx_path_init_t *init);
147 char *ngx_conf_set_access_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
148
149
150 extern ngx_atomic_t      *ngx_temp_number;
151 extern ngx_atomic_int_t  ngx_random_number;
152
153
154 #endif /* NGX_FILE_H_INCLUDED */

```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_file.c - nginx-1.7.10

### Global variables defined

- [ngx\\_random\\_number](#)
- [ngx\\_temp\\_number](#)
- [temp\\_number](#)

### Functions defined

- [ngx\\_add\\_path](#)
- [ngx\\_conf\\_merge\\_path\\_value](#)
- [ngx\\_conf\\_set\\_access\\_slot](#)
- [ngx\\_conf\\_set\\_path\\_slot](#)
- [ngx\\_copy\\_file](#)
- [ngx\\_create\\_full\\_path](#)
- [ngx\\_create\\_hashed\\_filename](#)
- [ngx\\_create\\_path](#)
- [ngx\\_create\\_paths](#)
- [ngx\\_create\\_temp\\_file](#)
- [ngx\\_ext\\_rename\\_file](#)
- [ngx\\_get\\_full\\_name](#)
- [ngx\\_next\\_temp\\_number](#)
- [ngx\\_test\\_full\\_name](#)
- [ngx\\_walk\\_tree](#)
- [ngx\\_write\\_chain\\_to\\_temp\\_file](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10
11
12 static ngx_int_t ngx_test_full_name(ngx_str_t *name);
13
14
15 static ngx_atomic_t temp_number = 0;
16 ngx_atomic_t *ngx_temp_number = &temp_number;
17 ngx_atomic_int_t ngx_random_number = 123456;
18
```

```

19
20 ngx_int_t
21 ngx_get_full_name(ngx_pool_t *pool, ngx_str_t *prefix, ngx_str_t *name)
22 {
23     size_t    len;
24     u_char    *p, *n;
25     ngx_int_t rc;
26
27     rc = ngx_test_full_name(name);
28
29     if (rc == NGX_OK) {
30         return rc;
31     }
32
33     len = prefix->len;
34
35     #if (NGX_WIN32)
36
37     if (rc == 2) {
38         len = rc;
39     }
40
41     #endif
42
43     n = ngx_pnalloc(pool, len + name->len + 1);
44     if (n == NULL) {
45         return NGX_ERROR;
46     }
47
48     p = ngx_cpymem(n, prefix->data, len);
49     ngx_cpystren(p, name->data, name->len + 1);
50
51     name->len += len;
52     name->data = n;
53
54     return NGX_OK;
55 }
56
57
58 static ngx_int_t
59 ngx_test_full_name(ngx_str_t *name)
60 {
61     #if (NGX_WIN32)
62     u_char  c0, c1;
63
64     c0 = name->data[0];
65
66     if (name->len < 2) {
67         if (c0 == '/') {
68             return 2;
69         }
70
71         return NGX_DECLINED;
72     }
73
74     c1 = name->data[1];
75
76     if (c1 == ':') {
77         c0 |= 0x20;
78
79         if ((c0 >= 'a' && c0 <= 'z')) {
80             return NGX_OK;
81         }
82
83         return NGX_DECLINED;
84     }
85
86     if (c1 == '/') {
87         return NGX_OK;
88     }
89
90     if (c0 == '/') {
91         return 2;
92     }
93
94     return NGX_DECLINED;

```

```

95
96 #else
97
98     if (name->data[0] == '/') {
99         return NGX_OK;
100     }
101
102     return NGX_DECLINED;
103
104 #endif
105 }
106
107
108 ssize_t
109 ngx_write_chain_to_temp_file(ngx_temp_file_t *tf, ngx_chain_t *chain)
110 {
111     ngx_int_t rc;
112
113     if (tf->file.fd == NGX_INVALID_FILE) {
114         rc = ngx_create_temp_file(&tf->file, tf->path, tf->pool,
115                                     tf->persistent, tf->clean, tf->access);
116
117         if (rc != NGX_OK) {
118             return rc;
119         }
120
121         if (tf->log_level) {
122             ngx_log_error(tf->log_level, tf->file.log, 0, "%s %V",
123                             tf->warn, &tf->file.name);
124         }
125     }
126
127     return ngx_write_chain_to_file(&tf->file, chain, tf->offset, tf->pool);
128 }
129
130
131 ngx_int_t
132 ngx_create_temp_file(ngx_file_t *file, ngx_path_t *path, ngx_pool_t *pool,
133                     ngx_uint_t persistent, ngx_uint_t clean, ngx_uint_t access)
134 {
135     uint32_t n;
136     ngx_err_t err;
137     ngx_pool_cleanup_t *cIn;
138     ngx_pool_cleanup_file_t *cInF;
139
140     file->name.len = path->name.len + 1 + path->len + 10;
141
142     file->name.data = ngx_pnalloc(pool, file->name.len + 1);
143     if (file->name.data == NULL) {
144         return NGX_ERROR;
145     }
146
147     #if 0
148     for (i = 0; i < file->name.len; i++) {
149         file->name.data[i] = 'X';
150     }
151     #endif
152
153     ngx_memcpy(file->name.data, path->name.data, path->name.len);
154
155     n = (uint32_t) ngx_next_temp_number(0);
156
157     cIn = ngx_pool_cleanup_add(pool, sizeof(ngx_pool_cleanup_file_t));
158     if (cIn == NULL) {
159         return NGX_ERROR;
160     }
161
162     for ( ;; ) {
163         (void) ngx_sprintf(file->name.data + path->name.len + 1 + path->len,
164                             "%010uD%Z", n);
165
166         ngx_create_hashed_filename(path, file->name.data, file->name.len);
167
168         ngx_log_debug1(NGX_LOG_DEBUG_CORE, file->log, 0,
169                         "hashed path: %s", file->name.data);
170

```

```

171 file->fd = ngx_open_tempfile(file->name.data, persistent, access);
172
173 ngx_log_debug1(NGX_LOG_DEBUG_CORE, file->log, 0,
174             "temp fd:%d", file->fd);
175
176 if (file->fd != NGX_INVALID_FILE) {
177
178     cln->handler = clean ? ngx_pool_delete_file : ngx_pool_cleanup_file;
179     clnf = cln->data;
180
181     clnf->fd = file->fd;
182     clnf->name = file->name.data;
183     clnf->log = pool->log;
184
185     return NGX_OK;
186 }
187
188 err = ngx_errno;
189
190 if (err == NGX_EEXIST) {
191     n = (uint32_t) ngx_next_temp_number(1);
192     continue;
193 }
194
195 if ((path->level[0] == 0) || (err != NGX_ENOPATH)) {
196     ngx_log_error(NGX_LOG_CRIT, file->log, err,
197                 ngx_open_tempfile_n " \"%s\" failed",
198                 file->name.data);
199     return NGX_ERROR;
200 }
201
202 if (ngx_create_path(file, path) == NGX_ERROR) {
203     return NGX_ERROR;
204 }
205 }
206 }
207
208
209 void
210 ngx_create_hashed_filename(ngx_path_t *path, u_char *file, size_t len)
211 {
212     size_t    i, level;
213     ngx_uint_t n;
214
215     i = path->name.len + 1;
216
217     file[path->name.len + path->len] = '/';
218
219     for (n = 0; n < 3; n++) {
220         level = path->level[n];
221
222         if (level == 0) {
223             break;
224         }
225
226         len -= level;
227         file[i - 1] = '/';
228         ngx_memcpy(&file[i], &file[len], level);
229         i += level + 1;
230     }
231 }
232
233
234 ngx_int_t
235 ngx_create_path(ngx_file_t *file, ngx_path_t *path)
236 {
237     size_t    pos;
238     ngx_err_t err;
239     ngx_uint_t i;
240
241     pos = path->name.len;
242
243     for (i = 0; i < 3; i++) {
244         if (path->level[i] == 0) {
245             break;
246         }

```

```

247     pos += path->level[i] + 1;
248
249     file->name.data[pos] = '\0';
250
251     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_CORE, file->log, 0,
252                 "temp file: \"%s\"", file->name.data);
253
254
255     if (ngx\_create\_dir(file->name.data, 0700) == NGX\_FILE\_ERROR) {
256         err = ngx\_errno;
257         if (err != NGX\_EEXIST) {
258             ngx\_log\_error(NGX\_LOG\_CRIT, file->log, err,
259                          ngx\_create\_dir\_n " \"%s\" failed",
260                          file->name.data);
261             return NGX\_ERROR;
262         }
263     }
264
265     file->name.data[pos] = '/';
266 }
267
268 return NGX\_OK;
269 }
270
271
272 ngx\_err\_t
273 ngx\_create\_full\_path(u_char *dir, ngx\_uint\_t access)
274 {
275     u_char    *p, ch;
276     ngx\_err\_t  err;
277
278     err = 0;
279
280     #if (NGX\_WIN32)
281         p = dir + 3;
282     #else
283         p = dir + 1;
284     #endif
285
286     for ( /* void */ ; *p; p++) {
287         ch = *p;
288
289         if (ch != '/') {
290             continue;
291         }
292
293         *p = '\0';
294
295         if (ngx\_create\_dir(dir, access) == NGX\_FILE\_ERROR) {
296             err = ngx\_errno;
297
298             switch (err) {
299                 case NGX\_EEXIST:
300                     err = 0;
301                 case NGX\_EACCES:
302                     break;
303
304                 default:
305                     return err;
306             }
307         }
308
309         *p = '/';
310     }
311
312     return err;
313 }
314
315
316 ngx\_atomic\_uint\_t
317 ngx\_next\_temp\_number(ngx\_uint\_t collision)
318 {
319     ngx\_atomic\_uint\_t  n, add;
320
321     add = collision ? ngx\_random\_number : 1;
322

```

```

323     n = ngx_atomic_fetch_add(ngx_temp_number, add);
324
325     return n + add;
326 }
327
328
329 char *
330 ngx_conf_set_path_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
331 {
332     char *p = conf;
333
334     ssize_t    level;
335     ngx_str_t  *value;
336     ngx_uint_t  i, n;
337     ngx_path_t *path, **slot;
338
339     slot = (ngx_path_t **) (p + cmd->offset);
340
341     if (*slot) {
342         return "is duplicate";
343     }
344
345     path = ngx_palloc(cf->pool, sizeof(ngx_path_t));
346     if (path == NULL) {
347         return NGX_CONF_ERROR;
348     }
349
350     value = cf->args->elts;
351
352     path->name = value[1];
353
354     if (path->name.data[path->name.len - 1] == '/') {
355         path->name.len--;
356     }
357
358     if (ngx_conf_full_name(cf->cycle, &path->name, 0) != NGX_OK) {
359         return NULL;
360     }
361
362     path->conf_file = cf->conf_file->file.name.data;
363     path->line = cf->conf_file->line;
364
365     for (i = 0, n = 2; n < cf->args->nelts; i++, n++) {
366         level = ngx_atoi(value[n].data, value[n].len);
367         if (level == NGX_ERROR || level == 0) {
368             return "invalid value";
369         }
370
371         path->level[i] = level;
372         path->len += level + 1;
373     }
374
375     while (i < 3) {
376         path->level[i++] = 0;
377     }
378
379     *slot = path;
380
381     if (ngx_add_path(cf, slot) == NGX_ERROR) {
382         return NGX_CONF_ERROR;
383     }
384
385     return NGX_CONF_OK;
386 }
387
388
389 char *
390 ngx_conf_merge_path_value(ngx_conf_t *cf, ngx_path_t **path, ngx_path_t *prev,
391     ngx_path_init_t *init)
392 {
393     if (*path) {
394         return NGX_CONF_OK;
395     }
396
397     if (prev) {
398         *path = prev;

```



```

399     return NGX_CONF_OK;
400 }
401
402 *path = ngx_palloc(cf->pool, sizeof(ngx_path_t));
403 if (*path == NULL) {
404     return NGX_CONF_ERROR;
405 }
406
407 (*path)->name = init->name;
408
409 if (ngx_conf_full_name(cf->cycle, &(*path)->name, 0) != NGX_OK) {
410     return NGX_CONF_ERROR;
411 }
412
413 (*path)->level[0] = init->level[0];
414 (*path)->level[1] = init->level[1];
415 (*path)->level[2] = init->level[2];
416
417 (*path)->len = init->level[0] + (init->level[0] ? 1 : 0)
418             + init->level[1] + (init->level[1] ? 1 : 0)
419             + init->level[2] + (init->level[2] ? 1 : 0);
420
421 if (ngx_add_path(cf, path) != NGX_OK) {
422     return NGX_CONF_ERROR;
423 }
424
425 return NGX_CONF_OK;
426 }
427
428
429 char *
430 ngx_conf_set_access_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
431 {
432     char *confp = conf;
433
434     u_char    *p;
435     ngx_str_t  *value;
436     ngx_uint_t  i, right, shift, *access;
437
438     access = (ngx_uint_t *) (confp + cmd->offset);
439
440     if (*access != NGX_CONF_UNSET_UINT) {
441         return "is duplicate";
442     }
443
444     value = cf->args->elts;
445
446     *access = 0600;
447
448     for (i = 1; i < cf->args->nelts; i++) {
449         p = value[i].data;
450
451         if (ngx_strncmp(p, "user:", sizeof("user:") - 1) == 0) {
452             shift = 6;
453             p += sizeof("user:") - 1;
454
455         } else if (ngx_strncmp(p, "group:", sizeof("group:") - 1) == 0) {
456             shift = 3;
457             p += sizeof("group:") - 1;
458
459         } else if (ngx_strncmp(p, "all:", sizeof("all:") - 1) == 0) {
460             shift = 0;
461             p += sizeof("all:") - 1;
462
463         } else {
464             goto invalid;
465         }
466     }
467
468     if (ngx_strcmp(p, "rw") == 0) {
469         right = 6;
470
471     } else if (ngx_strcmp(p, "r") == 0) {
472         right = 4;
473
474     } else {

```

```

475     goto invalid;
476 }
477
478 *access |= right << shift;
479 }
480
481 return NGX_CONF_OK;
482
483 invalid:
484
485 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0, "invalid value \"%V\"", &value[i]);
486
487 return NGX_CONF_ERROR;
488 }
489
490
491 ngx_int_t
492 ngx_add_path(ngx_conf_t *cf, ngx_path_t **slot)
493 {
494     ngx_uint_t i, n;
495     ngx_path_t *path, **p;
496
497     path = *slot;
498
499     p = cf->cycle->paths.elts;
500     for (i = 0; i < cf->cycle->paths.nelts; i++) {
501         if (p[i]->name.len == path->name.len
502             && ngx_strcmp(p[i]->name.data, path->name.data) == 0)
503         {
504             if (p[i]->data != path->data) {
505                 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
506                     "the same path name \"%V\" "
507                     "used in %s:%ui and",
508                     &p[i]->name, p[i]->conf_file, p[i]->line);
509                 return NGX_ERROR;
510             }
511
512             for (n = 0; n < 3; n++) {
513                 if (p[i]->level[n] != path->level[n]) {
514                     if (path->conf_file == NULL) {
515                         if (p[i]->conf_file == NULL) {
516                             ngx_log_error(NGX_LOG_EMERG, cf->log, 0,
517                                 "the default path name \"%V\" has "
518                                 "the same name as another default path, "
519                                 "but the different levels, you need to "
520                                 "redefine one of them in http section",
521                                 &p[i]->name);
522                             return NGX_ERROR;
523                         }
524
525                         ngx_log_error(NGX_LOG_EMERG, cf->log, 0,
526                             "the path name \"%V\" in %s:%ui has "
527                             "the same name as default path, but "
528                             "the different levels, you need to "
529                             "define default path in http section",
530                             &p[i]->name, p[i]->conf_file, p[i]->line);
531                         return NGX_ERROR;
532                     }
533
534                     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
535                         "the same path name \"%V\" in %s:%ui "
536                         "has the different levels than",
537                         &p[i]->name, p[i]->conf_file, p[i]->line);
538                     return NGX_ERROR;
539                 }
540
541                 if (p[i]->level[n] == 0) {
542                     break;
543                 }
544             }
545
546             *slot = p[i];
547
548             return NGX_OK;
549         }
550     }

```

```

551     p = ngx_array_push(&cf->cycle->paths);
552     if (p == NULL) {
553         return NGX_ERROR;
554     }
555
556     *p = path;
557
558     return NGX_OK;
559 }
560
561
562
563 ngx_int_t
564 ngx_create_paths(ngx_cycle_t *cycle, ngx_uid_t user)
565 {
566     ngx_err_t      err;
567     ngx_uint_t     i;
568     ngx_path_t     **path;
569
570     path = cycle->paths.elts;
571     for (i = 0; i < cycle->paths.nelts; i++) {
572
573         if (ngx_create_dir(path[i]->name.data, 0700) == NGX_FILE_ERROR) {
574             err = ngx_errno;
575             if (err != NGX_EEXIST) {
576                 ngx_log_error(NGX_LOG_EMERG, cycle->log, err,
577                     ngx_create_dir_n " \"%s\" failed",
578                     path[i]->name.data);
579                 return NGX_ERROR;
580             }
581         }
582
583         if (user == (ngx_uid_t) NGX_CONF_UNSET_UINT) {
584             continue;
585         }
586
587         #if !(NGX_WIN32)
588         {
589             ngx_file_info_t  fi;
590
591             if (ngx_file_info((const char *) path[i]->name.data, &fi)
592                 == NGX_FILE_ERROR)
593             {
594                 ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
595                     ngx_file_info_n " \"%s\" failed", path[i]->name.data);
596                 return NGX_ERROR;
597             }
598
599             if (fi.st_uid != user) {
600                 if (chown((const char *) path[i]->name.data, user, -1) == -1) {
601                     ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
602                         "chown(\"%s\", %d) failed",
603                         path[i]->name.data, user);
604                     return NGX_ERROR;
605                 }
606             }
607
608             if ((fi.st_mode & (S_IRUSR|S_IWUSR|S_IXUSR))
609                 != (S_IRUSR|S_IWUSR|S_IXUSR))
610             {
611                 fi.st_mode |= (S_IRUSR|S_IWUSR|S_IXUSR);
612
613                 if (chmod((const char *) path[i]->name.data, fi.st_mode) == -1) {
614                     ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
615                         "chmod() \"%s\" failed", path[i]->name.data);
616                     return NGX_ERROR;
617                 }
618             }
619         }
620         #endif
621     }
622
623     return NGX_OK;
624 }
625
626

```

```

627 ngx_int_t
628 ngx_ext_rename_file(ngx_str_t *src, ngx_str_t *to, ngx_ext_rename_file_t *ext)
629 {
630     u_char          *name;
631     ngx_err_t       err;
632     ngx_copy_file_t cf;
633
634     #if !(NGX_WIN32)
635
636     if (ext->access) {
637         if (ngx_change_file_access(src->data, ext->access) == NGX_FILE_ERROR) {
638             ngx_log_error(NGX_LOG_CRIT, ext->log, ngx_errno,
639                 ngx_change_file_access_n " \"%s\" failed", src->data);
640             err = 0;
641             goto failed;
642         }
643     }
644
645     #endif
646
647     if (ext->time != -1) {
648         if (ngx_set_file_time(src->data, ext->fd, ext->time) != NGX_OK) {
649             ngx_log_error(NGX_LOG_CRIT, ext->log, ngx_errno,
650                 ngx_set_file_time_n " \"%s\" failed", src->data);
651             err = 0;
652             goto failed;
653         }
654     }
655
656     if (ngx_rename_file(src->data, to->data) != NGX_FILE_ERROR) {
657         return NGX_OK;
658     }
659
660     err = ngx_errno;
661
662     if (err == NGX_ENOPATH) {
663
664         if (!ext->create_path) {
665             goto failed;
666         }
667
668         err = ngx_create_full_path(to->data, ngx_dir_access(ext->path_access));
669
670         if (err) {
671             ngx_log_error(NGX_LOG_CRIT, ext->log, err,
672                 ngx_create_dir_n " \"%s\" failed", to->data);
673             err = 0;
674             goto failed;
675         }
676
677         if (ngx_rename_file(src->data, to->data) != NGX_FILE_ERROR) {
678             return NGX_OK;
679         }
680
681         err = ngx_errno;
682     }
683
684     #if (NGX_WIN32)
685
686     if (err == NGX_EEXIST) {
687         err = ngx_win32_rename_file(src, to, ext->log);
688
689         if (err == 0) {
690             return NGX_OK;
691         }
692     }
693
694     #endif
695
696     if (err == NGX_EXDEV) {
697
698         cf.size = -1;
699         cf.buf_size = 0;
700         cf.access = ext->access;
701         cf.time = ext->time;
702         cf.log = ext->log;

```

```

703     name = ngx_alloc(to->len + 1 + 10 + 1, ext->log);
704     if (name == NULL) {
705         return NGX_ERROR;
706     }
707
708
709     (void) ngx_sprintf(name, "%*s.%010uD%Z", to->len, to->data,
710                     (uint32_t) ngx_next_temp_number(0));
711
712     if (ngx_copy_file(src->data, name, &cf) == NGX_OK) {
713
714         if (ngx_rename_file(name, to->data) != NGX_FILE_ERROR) {
715             ngx_free(name);
716
717             if (ngx_delete_file(src->data) == NGX_FILE_ERROR) {
718                 ngx_log_error(NGX_LOG_CRIT, ext->log, ngx_errno,
719                             ngx_delete_file_n " \"%s\" failed",
720                             src->data);
721                 return NGX_ERROR;
722             }
723
724             return NGX_OK;
725         }
726
727         ngx_log_error(NGX_LOG_CRIT, ext->log, ngx_errno,
728                     ngx_rename_file_n " \"%s\" to \"%s\" failed",
729                     name, to->data);
730
731         if (ngx_delete_file(name) == NGX_FILE_ERROR) {
732             ngx_log_error(NGX_LOG_CRIT, ext->log, ngx_errno,
733                         ngx_delete_file_n " \"%s\" failed", name);
734         }
735     }
736
737     ngx_free(name);
738
739     err = 0;
740 }
741
742 failed:
743
744     if (ext->delete_file) {
745         if (ngx_delete_file(src->data) == NGX_FILE_ERROR) {
746             ngx_log_error(NGX_LOG_CRIT, ext->log, ngx_errno,
747                         ngx_delete_file_n " \"%s\" failed", src->data);
748         }
749     }
750
751     if (err) {
752         ngx_log_error(NGX_LOG_CRIT, ext->log, err,
753                     ngx_rename_file_n " \"%s\" to \"%s\" failed",
754                     src->data, to->data);
755     }
756
757     return NGX_ERROR;
758 }
759
760
761 ngx_int_t
762 ngx_copy_file(u_char *from, u_char *to, ngx_copy_file_t *cf)
763 {
764     char          *buf;
765     off_t         size;
766     size_t        len;
767     ssize_t       n;
768     ngx_fd_t      fd, nfd;
769     ngx_int_t     rc;
770     ngx_file_info_t fi;
771
772     rc = NGX_ERROR;
773     buf = NULL;
774     nfd = NGX_INVALID_FILE;
775
776     fd = ngx_open_file(from, NGX_FILE_RDONLY, NGX_FILE_OPEN, 0);

```

```

779 if (fd == NGX_INVALID_FILE) {
780     ngx_log_error(NGX_LOG_CRIT, cf->log, ngx_errno,
781                 ngx_open_file_n "\'%s\' failed", from);
782     goto failed;
783 }
784
785 if (cf->size != -1) {
786     size = cf->size;
787
788 } else {
789     if (ngx_fd_info(fd, &fi) == NGX_FILE_ERROR) {
790         ngx_log_error(NGX_LOG_ALERT, cf->log, ngx_errno,
791                     ngx_fd_info_n "\'%s\' failed", from);
792
793         goto failed;
794     }
795
796     size = ngx_file_size(&fi);
797 }
798
799 len = cf->buf_size ? cf->buf_size : 65536;
800
801 if ((off_t) len > size) {
802     len = (size_t) size;
803 }
804
805 buf = ngx_alloc(len, cf->log);
806 if (buf == NULL) {
807     goto failed;
808 }
809
810 nfd = ngx_open_file(to, NGX_FILE_WRONLY, NGX_FILE_CREATE_OR_OPEN,
811                   cf->access);
812
813 if (nfd == NGX_INVALID_FILE) {
814     ngx_log_error(NGX_LOG_CRIT, cf->log, ngx_errno,
815                 ngx_open_file_n "\'%s\' failed", to);
816     goto failed;
817 }
818
819 while (size > 0) {
820
821     if ((off_t) len > size) {
822         len = (size_t) size;
823     }
824
825     n = ngx_read_fd(fd, buf, len);
826
827     if (n == -1) {
828         ngx_log_error(NGX_LOG_ALERT, cf->log, ngx_errno,
829                     ngx_read_fd_n "\'%s\' failed", from);
830         goto failed;
831     }
832
833     if ((size_t) n != len) {
834         ngx_log_error(NGX_LOG_ALERT, cf->log, 0,
835                     ngx_read_fd_n " has read only %z of %uz from %s",
836                     n, size, from);
837         goto failed;
838     }
839
840     n = ngx_write_fd(nfd, buf, len);
841
842     if (n == -1) {
843         ngx_log_error(NGX_LOG_ALERT, cf->log, ngx_errno,
844                     ngx_write_fd_n "\'%s\' failed", to);
845         goto failed;
846     }
847
848     if ((size_t) n != len) {
849         ngx_log_error(NGX_LOG_ALERT, cf->log, 0,
850                     ngx_write_fd_n " has written only %z of %uz to %s",
851                     n, size, to);
852         goto failed;
853     }
854

```

```

855     size -= n;
856 }
857
858 if (cf->time != -1) {
859     if (ngx\_set\_file\_time(to, nfd, cf->time) != NGX\_OK) {
860         ngx\_log\_error(NGX\_LOG\_ALERT, cf->log, ngx\_errno,
861             ngx\_set\_file\_time\_n " \"%s\" failed", to);
862         goto failed;
863     }
864 }
865
866 rc = NGX\_OK;
867
868 failed:
869
870 if (nfd != NGX\_INVALID\_FILE) {
871     if (ngx\_close\_file(nfd) == NGX\_FILE\_ERROR) {
872         ngx\_log\_error(NGX\_LOG\_ALERT, cf->log, ngx\_errno,
873             ngx\_close\_file\_n " \"%s\" failed", to);
874     }
875 }
876
877 if (fd != NGX\_INVALID\_FILE) {
878     if (ngx\_close\_file(fd) == NGX\_FILE\_ERROR) {
879         ngx\_log\_error(NGX\_LOG\_ALERT, cf->log, ngx\_errno,
880             ngx\_close\_file\_n " \"%s\" failed", from);
881     }
882 }
883
884 if (buf) {
885     ngx\_free(buf);
886 }
887
888 return rc;
889 }
890
891
892 /*
893 * ctx->init\_handler\(\) - see ctx->alloc
894 * ctx->file\_handler\(\) - file handler
895 * ctx->pre\_tree\_handler\(\) - handler is called before entering directory
896 * ctx->post\_tree\_handler\(\) - handler is called after leaving directory
897 * ctx->spec\_handler\(\) - special (socket, FIFO, etc.) file handler
898 *
899 * ctx->data - some data structure, it may be the same on all levels, or
900 *   reallocated if ctx->alloc is nonzero
901 *
902 * ctx->alloc - a size of data structure that is allocated at every level
903 *   and is initialized by ctx->init\_handler\(\)
904 *
905 * ctx->log - a log
906 *
907 * on fatal (memory) error handler must return NGX\_ABORT to stop walking tree
908 */
909
910 ngx\_int\_t
911 ngx\_walk\_tree(ngx\_tree\_ctx\_t *ctx, ngx\_str\_t *tree)
912 {
913     void      *data, *prev;
914     u_char    *p, *name;
915     size_t    len;
916     ngx\_int\_t rc;
917     ngx\_err\_t err;
918     ngx\_str\_t file, buf;
919     ngx\_dir\_t dir;
920
921     ngx\_str\_null(&buf);
922
923     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_CORE, ctx->log, 0,
924         "walk tree \"%V\"", tree);
925
926     if (ngx\_open\_dir(tree, &dir) == NGX\_ERROR) {
927         ngx\_log\_error(NGX\_LOG\_CRIT, ctx->log, ngx\_errno,
928             ngx\_open\_dir\_n " \"%s\" failed", tree->data);
929         return NGX\_ERROR;
930     }
931 }

```

```

931 prev = ctx->data;
932
933
934 if (ctx->alloc) {
935     data = ngx_alloc(ctx->alloc, ctx->log);
936     if (data == NULL) {
937         goto failed;
938     }
939
940     if (ctx->init_handler(data, prev) == NGX_ABORT) {
941         goto failed;
942     }
943
944     ctx->data = data;
945
946 } else {
947     data = NULL;
948 }
949
950 for ( ;; ) {
951     ngx_set_errno(0);
952
953     if (ngx_read_dir(&dir) == NGX_ERROR) {
954         err = ngx_errno;
955
956         if (err == NGX_ENOMOREFILES) {
957             rc = NGX_OK;
958
959         } else {
960             ngx_log_error(NGX_LOG_CRIT, ctx->log, err,
961                 ngx_read_dir_n "%s\" failed", tree->data);
962             rc = NGX_ERROR;
963         }
964     }
965
966     goto done;
967 }
968
969 len = ngx_de_nameLen(&dir);
970 name = ngx_de_name(&dir);
971
972 ngx_log_debug2(NGX_LOG_DEBUG_CORE, ctx->log, 0,
973     "tree name %uz: \"%s\"", len, name);
974
975 if (len == 1 && name[0] == '.') {
976     continue;
977 }
978
979 if (len == 2 && name[0] == '.' && name[1] == '.') {
980     continue;
981 }
982
983 file.len = tree->len + 1 + len;
984
985 if (file.len + NGX_DIR_MASK_LEN > buf.len) {
986     if (buf.len) {
987         ngx_free(buf.data);
988     }
989
990     buf.len = tree->len + 1 + len + NGX_DIR_MASK_LEN;
991
992     buf.data = ngx_alloc(buf.len + 1, ctx->log);
993     if (buf.data == NULL) {
994         goto failed;
995     }
996 }
997
998
999 p = ngx_cpymem(buf.data, tree->data, tree->len);
1000 *p++ = '/';
1001 ngx_memcpy(p, name, len + 1);
1002
1003 file.data = buf.data;
1004
1005 ngx_log_debug1(NGX_LOG_DEBUG_CORE, ctx->log, 0,
1006     "tree path \"%s\"", file.data);

```



```

1007
1008     if (!dir.valid_info) {
1009         if (ngx\_de\_info(file.data, &dir) == NGX\_FILE\_ERROR) {
1010             ngx\_log\_error(NGX\_LOG\_CRIT, ctx->log, ngx\_errno,
1011                 ngx\_de\_info\_n " \"%s\" failed", file.data);
1012             continue;
1013         }
1014     }
1015
1016     if (ngx\_de\_is\_file(&dir)) {
1017
1018         ngx\_log\_debug1(NGX\_LOG\_DEBUG\_CORE, ctx->log, 0,
1019             "tree file \"%s\"", file.data);
1020
1021         ctx->size = ngx\_de\_size(&dir);
1022         ctx->fs_size = ngx\_de\_fs\_size(&dir);
1023         ctx->access = ngx\_de\_access(&dir);
1024         ctx->mtime = ngx\_de\_mtime(&dir);
1025
1026         if (ctx->file_handler(ctx, &file) == NGX\_ABORT) {
1027             goto failed;
1028         }
1029
1030     } else if (ngx\_de\_is\_dir(&dir)) {
1031
1032         ngx\_log\_debug1(NGX\_LOG\_DEBUG\_CORE, ctx->log, 0,
1033             "tree enter dir \"%s\"", file.data);
1034
1035         ctx->access = ngx\_de\_access(&dir);
1036         ctx->mtime = ngx\_de\_mtime(&dir);
1037
1038         rc = ctx->pre_tree_handler(ctx, &file);
1039
1040         if (rc == NGX\_ABORT) {
1041             goto failed;
1042         }
1043
1044         if (rc == NGX\_DECLINED) {
1045             ngx\_log\_debug1(NGX\_LOG\_DEBUG\_CORE, ctx->log, 0,
1046                 "tree skip dir \"%s\"", file.data);
1047             continue;
1048         }
1049
1050         if (ngx\_walk\_tree(ctx, &file) == NGX\_ABORT) {
1051             goto failed;
1052         }
1053
1054         ctx->access = ngx\_de\_access(&dir);
1055         ctx->mtime = ngx\_de\_mtime(&dir);
1056
1057         if (ctx->post_tree_handler(ctx, &file) == NGX\_ABORT) {
1058             goto failed;
1059         }
1060
1061     } else {
1062
1063         ngx\_log\_debug1(NGX\_LOG\_DEBUG\_CORE, ctx->log, 0,
1064             "tree special \"%s\"", file.data);
1065
1066         if (ctx->spec_handler(ctx, &file) == NGX\_ABORT) {
1067             goto failed;
1068         }
1069     }
1070 }
1071
1072 failed:
1073
1074     rc = NGX\_ABORT;
1075
1076 done:
1077
1078     if (buf.len) {
1079         ngx\_free(buf.data);
1080     }
1081
1082     if (data) {

```

```
1083     ngx_free(data);
1084     ctx->data = prev;
1085 }
1086
1087 if (ngx_close_dir(&dir) == NGX_ERROR) {
1088     ngx_log_error(NGX_LOG_CRIT, ctx->log, ngx_errno,
1089                 ngx_close_dir_n " \"%s\" failed", tree->data);
1090 }
1091
1092 return rc;
1093 }
```

[One Level Up](#)

[Top Level](#)

## src/event/nginx\_event\_timer.c - nginx-1.7.10

### Global variables defined

- [ngx\\_event\\_timer\\_rbtrees](#)
- [ngx\\_event\\_timer\\_sentinel](#)

### Functions defined

- [ngx\\_event\\_cancel\\_timers](#)
- [ngx\\_event\\_expire\\_timers](#)
- [ngx\\_event\\_find\\_timer](#)
- [ngx\\_event\\_timer\\_init](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 ngx_rbtrees_t          ngx_event_timer_rbtrees;
14 static ngx_rbtrees_t  ngx_event_timer_sentinel;
15
16 /*
17  * the event timer rbtrees may contain the duplicate keys, however,
18  * it should not be a problem, because we use the rbtrees to find
19  * a minimum timer value only
20  */
21
22 ngx_int_t
23 ngx_event_timer_init(ngx_log_t *log)
24 {
25     ngx_rbtrees_init(&ngx_event_timer_rbtrees, &ngx_event_timer_sentinel,
26                     ngx_rbtrees_insert_timer_value);
27
28     return NGX_OK;
29 }
30
31
32 ngx_msec_t
33 ngx_event_find_timer(void)
34 {
35     ngx_msec_int_t      timer;
36     ngx_rbtrees_node_t *node, *root, *sentinel;
37
38     if (ngx_event_timer_rbtrees.root == &ngx_event_timer_sentinel) {
39         return NGX_TIMER_INFINITE;
40     }
41
42     root = ngx_event_timer_rbtrees.root;
43     sentinel = ngx_event_timer_rbtrees.sentinel;
44
45     node = ngx_rbtrees_min(root, sentinel);
46
47     timer = (ngx_msec_int_t) (node->key - ngx_current_msec);
48 }
```

```

49     return (ngx_msec_t) (timer > 0 ? timer : 0);
50 }
51
52
53 void
54 ngx_event_expire_timers(void)
55 {
56     ngx_event_t      *ev;
57     ngx_rbtree_node_t *node, *root, *sentinel;
58
59     sentinel = ngx_event_timer_rbtree.sentinel;
60
61     for ( ;; ) {
62         root = ngx_event_timer_rbtree.root;
63
64         if (root == sentinel) {
65             return;
66         }
67
68         node = ngx_rbtree_min(root, sentinel);
69
70         /* node->key > ngx_current_time */
71
72         if ((ngx_msec_int_t) (node->key - ngx_current_msec) > 0) {
73             return;
74         }
75
76         ev = (ngx_event_t *) ((char *) node - offsetof(ngx_event_t, timer));
77
78         ngx_log_debug2(NGX_LOG_DEBUG_EVENT, ev->log, 0,
79             "event timer del: %d: %M",
80             ngx_event_ident(ev->data), ev->timer.key);
81
82         ngx_rbtree_delete(&ngx_event_timer_rbtree, &ev->timer);
83
84     #if (NGX_DEBUG)
85         ev->timer.left = NULL;
86         ev->timer.right = NULL;
87         ev->timer.parent = NULL;
88     #endif
89
90     ev->timer_set = 0;
91
92     ev->timedout = 1;
93
94     ev->handler(ev);
95 }
96 }
97
98
99 void
100 ngx_event_cancel_timers(void)
101 {
102     ngx_event_t      *ev;
103     ngx_rbtree_node_t *node, *root, *sentinel;
104
105     sentinel = ngx_event_timer_rbtree.sentinel;
106
107     for ( ;; ) {
108         root = ngx_event_timer_rbtree.root;
109
110         if (root == sentinel) {
111             return;
112         }
113
114         node = ngx_rbtree_min(root, sentinel);
115
116         ev = (ngx_event_t *) ((char *) node - offsetof(ngx_event_t, timer));
117
118         if (!ev->cancelable) {
119             return;
120         }
121
122         ngx_log_debug2(NGX_LOG_DEBUG_EVENT, ev->log, 0,
123             "event timer cancel: %d: %M",
124             ngx_event_ident(ev->data), ev->timer.key);

```

```
125         ngx\_rbtree\_delete(&ngx\_event\_timer\_rbtree, &ev->timer);
126
127
128     #if (NGX_DEBUG)
129         ev->timer.left = NULL;
130         ev->timer.right = NULL;
131         ev->timer.parent = NULL;
132     #endif
133
134         ev->timer_set = 0;
135
136         ev->handler(ev);
137     }
138 }
```

[One Level Up](#)

[Top Level](#)

# src/event/nginx\_event\_timer.h - nginx-1.7.10

## Functions defined

- [ngx\\_event\\_add\\_timer](#)
- [ngx\\_event\\_del\\_timer](#)

## Macros defined

- [NGX\\_TIMER\\_INFINITE](#)
- [NGX\\_TIMER\\_LAZY\\_DELAY](#)
- [\\_NGX\\_EVENT\\_TIMER\\_H\\_INCLUDED](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_EVENT_TIMER_H_INCLUDED_
9 #define _NGX_EVENT_TIMER_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14 #include <ngx_event.h>
15
16
17 #define NGX_TIMER_INFINITE  (ngx\_msec\_t) -1
18
19 #define NGX_TIMER_LAZY_DELAY  300
20
21
22 ngx\_int\_t ngx_event_timer_init(ngx\_log\_t *log);
23 ngx\_msec\_t ngx_event_find_timer(void);
24 void ngx_event_expire_timers(void);
25 void ngx_event_cancel_timers(void);
26
27
28 extern ngx\_rbtree\_t ngx_event_timer_rbtree;
29
30
31 static ngx\_inline void
32 ngx_event_del_timer(ngx\_event\_t *ev)
33 {
34     ngx\_log\_debug2(NGX_LOG_DEBUG_EVENT, ev->log, 0,
35                  "event timer del: %d: %M",
36                  ngx\_event\_ident(ev->data), ev->timer.key);
37
38     ngx\_rbtree\_delete(&ngx_event_timer_rbtree, &ev->timer);
39
40     #if (NGX_DEBUG)
41         ev->timer.left = NULL;
42         ev->timer.right = NULL;
43         ev->timer.parent = NULL;
44     #endif
45
46     ev->timer_set = 0;
47 }
48
49
50 static ngx\_inline void
```

```

51 ngx_event_add_timer(ngx_event_t *ev, ngx_msec_t timer)
52 {
53     ngx_msec_t    key;
54     ngx_msec_int_t diff;
55
56     key = ngx_current_msec + timer;
57
58     if (ev->timer_set) {
59
60         /*
61          * Use a previous timer value if difference between it and a new
62          * value is less than NGX_TIMER_LAZY_DELAY milliseconds: this allows
63          * to minimize the rbtree operations for fast connections.
64          */
65
66         diff = (ngx_msec_int_t) (key - ev->timer.key);
67
68         if (ngx_abs(diff) < NGX_TIMER_LAZY_DELAY) {
69             ngx_log_debug3(NGX_LOG_DEBUG_EVENT, ev->log, 0,
70                 "event timer: %d, old: %M, new: %M",
71                 ngx_event_ident(ev->data), ev->timer.key, key);
72             return;
73         }
74
75         ngx_del_timer(ev);
76     }
77
78     ev->timer.key = key;
79
80     ngx_log_debug3(NGX_LOG_DEBUG_EVENT, ev->log, 0,
81         "event timer add: %d: %M:%M",
82         ngx_event_ident(ev->data), timer, ev->timer.key);
83
84     ngx_rbtree_insert(&ngx_event_timer_rbtree, &ev->timer);
85
86     ev->timer_set = 1;
87 }
88
89
90 #endif /* _NGX_EVENT_TIMER_H_INCLUDED */

```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_times.h - nginx-1.7.10

### Data types defined

- [ngx\\_time\\_t](#)

### Macros defined

- [\\_NGX\\_TIMES\\_H\\_INCLUDED](#)
- [ngx\\_next\\_time\\_n](#)
- [ngx\\_time](#)
- [ngx\\_timeofday](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_TIMES_H_INCLUDED_
9 #define _NGX_TIMES_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 typedef struct {
17     time_t      sec;
18     ngx_uint_t  msec;
19     ngx_int_t   gmtoff;
20 } ngx_time_t;
21
22
23 void ngx_time_init(void);
24 void ngx_time_update(void);
25 void ngx_time_sigsafe_update(void);
26 u_char *ngx_http_time(u_char *buf, time_t t);
27 u_char *ngx_http_cookie_time(u_char *buf, time_t t);
28 void ngx_gmtime(time_t t, ngx_tm_t *tp);
29
30 time_t ngx_next_time(time_t when);
31 #define ngx_next_time_n      "mktime()"
32
33
34 extern volatile ngx_time_t *ngx_cached_time;
35
36 #define ngx_time()           ngx_cached_time->sec
37 #define ngx_timeofday()     (ngx_time_t *) ngx_cached_time
38
39 extern volatile ngx_str_t   ngx_cached_err_log_time;
40 extern volatile ngx_str_t   ngx_cached_http_time;
41 extern volatile ngx_str_t   ngx_cached_http_log_time;
42 extern volatile ngx_str_t   ngx_cached_http_log_iso8601;
43 extern volatile ngx_str_t   ngx_cached_syslog_time;
44
45 /*
46  * milliseconds elapsed since epoch and truncated to ngx_msec_t,
47  * used in event timers
48  */
49 extern volatile ngx_msec_t  ngx_current_msec;
50
```



51

52 #endif /\* NGX\_TIMES\_H\_INCLUDED \*/

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_slab.c - nginx-1.7.10

### Global variables defined

- [ngx\\_slab\\_exact\\_shift](#)
- [ngx\\_slab\\_exact\\_size](#)
- [ngx\\_slab\\_max\\_size](#)

### Functions defined

- [ngx\\_slab\\_alloc](#)
- [ngx\\_slab\\_alloc\\_locked](#)
- [ngx\\_slab\\_alloc\\_pages](#)
- [ngx\\_slab\\_calloc](#)
- [ngx\\_slab\\_calloc\\_locked](#)
- [ngx\\_slab\\_error](#)
- [ngx\\_slab\\_free](#)
- [ngx\\_slab\\_free\\_locked](#)
- [ngx\\_slab\\_free\\_pages](#)
- [ngx\\_slab\\_init](#)

### Macros defined

- [NGX\\_SLAB\\_BIG](#)
- [NGX\\_SLAB\\_BUSY](#)
- [NGX\\_SLAB\\_BUSY](#)
- [NGX\\_SLAB\\_EXACT](#)
- [NGX\\_SLAB\\_MAP\\_MASK](#)
- [NGX\\_SLAB\\_MAP\\_MASK](#)
- [NGX\\_SLAB\\_MAP\\_SHIFT](#)
- [NGX\\_SLAB\\_MAP\\_SHIFT](#)
- [NGX\\_SLAB\\_PAGE](#)
- [NGX\\_SLAB\\_PAGE\\_BUSY](#)
- [NGX\\_SLAB\\_PAGE\\_BUSY](#)
- [NGX\\_SLAB\\_PAGE\\_FREE](#)
- [NGX\\_SLAB\\_PAGE\\_FREE](#)
- [NGX\\_SLAB\\_PAGE\\_MASK](#)

- [NGX\\_SLAB\\_PAGE\\_START](#)
- [NGX\\_SLAB\\_PAGE\\_START](#)
- [NGX\\_SLAB\\_SHIFT\\_MASK](#)
- [NGX\\_SLAB\\_SHIFT\\_MASK](#)
- [NGX\\_SLAB\\_SMALL](#)
- [ngx\\_slab\\_junk](#)
- [ngx\\_slab\\_junk](#)
- [ngx\\_slab\\_junk](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7 #include <ngx_config.h>
8 #include <ngx_core.h>
9
10
11 #define NGX_SLAB_PAGE_MASK    3
12 #define NGX_SLAB_PAGE        0
13 #define NGX_SLAB_BIG          1
14 #define NGX_SLAB_EXACT        2
15 #define NGX_SLAB_SMALL        3
16
17 #if (NGX_PTR_SIZE == 4)
18
19 #define NGX_SLAB_PAGE_FREE    0
20 #define NGX_SLAB_PAGE_BUSY    0xffffffff
21 #define NGX_SLAB_PAGE_START   0x80000000
22
23 #define NGX_SLAB_SHIFT_MASK   0x0000000f
24 #define NGX_SLAB_MAP_MASK     0xffff0000
25 #define NGX_SLAB_MAP_SHIFT    16
26
27 #define NGX_SLAB_BUSY         0xffffffff
28
29 #else /* (NGX_PTR_SIZE == 8) */
30
31 #define NGX_SLAB_PAGE_FREE    0
32 #define NGX_SLAB_PAGE_BUSY    0xffffffffffffffff
33 #define NGX_SLAB_PAGE_START   0x8000000000000000
34
35 #define NGX_SLAB_SHIFT_MASK   0x000000000000000f
36 #define NGX_SLAB_MAP_MASK     0xffffffff00000000
37 #define NGX_SLAB_MAP_SHIFT    32
38
39 #define NGX_SLAB_BUSY         0xffffffffffffffff
40
41 #endif
42
43
44 #if (NGX_DEBUG_MALLOC)
45
46 #define ngx_slab_junk(p, size)    ngx_memset(p, 0xA5, size)
47
48 #elif (NGX_HAVE_DEBUG_MALLOC)
49
50 #define ngx_slab_junk(p, size)
51     if (ngx_debug_malloc)        ngx_memset(p, 0xA5, size)
52
53 #else
54

```

```

55 #define ngx_slab_junk(p, size)
56
57 #endif
58
59 static ngx_slab_page_t *ngx_slab_alloc_pages(ngx_slab_pool_t *pool,
60     ngx_uint_t pages);
61 static void ngx_slab_free_pages(ngx_slab_pool_t *pool, ngx_slab_page_t *page,
62     ngx_uint_t pages);
63 static void ngx_slab_error(ngx_slab_pool_t *pool, ngx_uint_t level,
64     char *text);
65
66
67 static ngx_uint_t ngx_slab_max_size;
68 static ngx_uint_t ngx_slab_exact_size;
69 static ngx_uint_t ngx_slab_exact_shift;
70
71
72 void
73 ngx_slab_init(ngx_slab_pool_t *pool)
74 {
75     u_char *p;
76     size_t size;
77     ngx_int_t m;
78     ngx_uint_t i, n, pages;
79     ngx_slab_page_t *slots;
80
81     /* STUB */
82     if (ngx_slab_max_size == 0) {
83         ngx_slab_max_size = ngx_pagesize / 2;
84         ngx_slab_exact_size = ngx_pagesize / (8 * sizeof(uintptr_t));
85         for (n = ngx_slab_exact_size; n >>= 1; ngx_slab_exact_shift++) {
86             /* void */
87         }
88     }
89     /**/
90
91     pool->min_size = 1 << pool->min_shift;
92
93     p = (u_char *) pool + sizeof(ngx_slab_pool_t);
94     size = pool->end - p;
95
96     ngx_slab_junk(p, size);
97
98     slots = (ngx_slab_page_t *) p;
99     n = ngx_pagesize_shift - pool->min_shift;
100
101     for (i = 0; i < n; i++) {
102         slots[i].slab = 0;
103         slots[i].next = &slots[i];
104         slots[i].prev = 0;
105     }
106
107     p += n * sizeof(ngx_slab_page_t);
108
109     pages = (ngx_uint_t) (size / (ngx_pagesize + sizeof(ngx_slab_page_t)));
110
111     ngx_memzero(p, pages * sizeof(ngx_slab_page_t));
112
113     pool->pages = (ngx_slab_page_t *) p;
114
115     pool->free.prev = 0;
116     pool->free.next = (ngx_slab_page_t *) p;
117
118     pool->pages->slab = pages;
119     pool->pages->next = &pool->free;
120     pool->pages->prev = (uintptr_t) &pool->free;
121
122     pool->start = (u_char *)
123         ngx_align_ptr((uintptr_t) p + pages * sizeof(ngx_slab_page_t),
124             ngx_pagesize);
125
126     m = pages - (pool->end - pool->start) / ngx_pagesize;
127     if (m > 0) {
128         pages -= m;
129         pool->pages->slab = pages;
130     }

```

```

131     pool->last = pool->pages + pages;
132
133     pool->log_nomem = 1;
134     pool->log_ctx = &pool->zero;
135     pool->zero = '\0';
136 }
137
138
139
140 void *
141 ngx_slab_alloc(ngx_slab_pool_t *pool, size_t size)
142 {
143     void *p;
144
145     ngx_shmtx_lock(&pool->mutex);
146
147     p = ngx_slab_alloc_locked(pool, size);
148
149     ngx_shmtx_unlock(&pool->mutex);
150
151     return p;
152 }
153
154
155 void *
156 ngx_slab_alloc_locked(ngx_slab_pool_t *pool, size_t size)
157 {
158     size_t      s;
159     uintptr_t   p, n, m, mask, *bitmap;
160     ngx_uint_t  i, slot, shift, map;
161     ngx_slab_page_t *page, *prev, *slots;
162
163     if (size > ngx_slab_max_size) {
164
165         ngx_log_debug1(NGX_LOG_DEBUG_ALLOC, ngx_cycle->log, 0,
166             "slab alloc: %uz", size);
167
168         page = ngx_slab_alloc_pages(pool, (size >> ngx_pagesize_shift)
169             + ((size % ngx_pagesize) ? 1 : 0));
170         if (page) {
171             p = (page - pool->pages) << ngx_pagesize_shift;
172             p += (uintptr_t) pool->start;
173
174         } else {
175             p = 0;
176         }
177
178         goto done;
179     }
180
181     if (size > pool->min_size) {
182         shift = 1;
183         for (s = size - 1; s >>= 1; shift++) { /* void */
184             slot = shift - pool->min_shift;
185
186         } else {
187             size = pool->min_size;
188             shift = pool->min_shift;
189             slot = 0;
190         }
191
192     ngx_log_debug2(NGX_LOG_DEBUG_ALLOC, ngx_cycle->log, 0,
193         "slab alloc: %uz slot: %ui", size, slot);
194
195     slots = (ngx_slab_page_t *) ((u_char *) pool + sizeof(ngx_slab_pool_t));
196     page = slots[slot].next;
197
198     if (page->next != page) {
199
200         if (shift < ngx_slab_exact_shift) {
201
202             do {
203                 p = (page - pool->pages) << ngx_pagesize_shift;
204                 bitmap = (uintptr_t *) (pool->start + p);
205
206                 map = (1 << (ngx_pagesize_shift - shift))

```

```

207         / (sizeof(uintptr_t) * 8);
208
209     for (n = 0; n < map; n++) {
210
211         if (bitmap[n] != NGX\_SLAB\_BUSY) {
212
213             for (m = 1, i = 0; m; m <<= 1, i++) {
214                 if ((bitmap[n] & m)) {
215                     continue;
216                 }
217
218                 bitmap[n] |= m;
219
220                 i = ((n * sizeof(uintptr_t) * 8) << shift)
221                     + (i << shift);
222
223                 if (bitmap[n] == NGX\_SLAB\_BUSY) {
224                     for (n = n + 1; n < map; n++) {
225                         if (bitmap[n] != NGX\_SLAB\_BUSY) {
226                             p = (uintptr_t) bitmap + i;
227
228                             goto done;
229                         }
230                     }
231
232                     prev = (ngx\_slab\_page\_t *)
233                         (page->prev & ~NGX\_SLAB\_PAGE\_MASK);
234                     prev->next = page->next;
235                     page->next->prev = page->prev;
236
237                     page->next = NULL;
238                     page->prev = NGX\_SLAB\_SMALL;
239                 }
240
241                 p = (uintptr_t) bitmap + i;
242
243                 goto done;
244             }
245         }
246     }
247
248     page = page->next;
249
250 } while (page);
251
252 } else if (shift == ngx\_slab\_exact\_shift) {
253
254     do {
255         if (page->slab != NGX\_SLAB\_BUSY) {
256
257             for (m = 1, i = 0; m; m <<= 1, i++) {
258                 if ((page->slab & m)) {
259                     continue;
260                 }
261
262                 page->slab |= m;
263
264                 if (page->slab == NGX\_SLAB\_BUSY) {
265                     prev = (ngx\_slab\_page\_t *)
266                         (page->prev & ~NGX\_SLAB\_PAGE\_MASK);
267                     prev->next = page->next;
268                     page->next->prev = page->prev;
269
270                     page->next = NULL;
271                     page->prev = NGX\_SLAB\_EXACT;
272                 }
273
274                 p = (page - pool->pages) << ngx\_pagesize\_shift;
275                 p += i << shift;
276                 p += (uintptr_t) pool->start;
277
278                 goto done;
279             }
280         }
281
282         page = page->next;

```

```

283     } while (page);
284
285
286 } else { /* shift > ngx_slab_exact_shift */
287
288     n = ngx_pagesize_shift - (page->slab & NGX_SLAB_SHIFT_MASK);
289     n = 1 << n;
290     n = ((uintptr_t) 1 << n) - 1;
291     mask = n << NGX_SLAB_MAP_SHIFT;
292
293     do {
294         if ((page->slab & NGX_SLAB_MAP_MASK) != mask) {
295
296             for (m = (uintptr_t) 1 << NGX_SLAB_MAP_SHIFT, i = 0;
297                 m & mask;
298                 m <<= 1, i++)
299             {
300                 if ((page->slab & m)) {
301                     continue;
302                 }
303
304                 page->slab |= m;
305
306                 if ((page->slab & NGX_SLAB_MAP_MASK) == mask) {
307                     prev = (ngx_slab_page_t *)
308                         (page->prev & ~NGX_SLAB_PAGE_MASK);
309                     prev->next = page->next;
310                     page->next->prev = page->prev;
311
312                     page->next = NULL;
313                     page->prev = NGX_SLAB_BIG;
314                 }
315
316                 p = (page - pool->pages) << ngx_pagesize_shift;
317                 p += i << shift;
318                 p += (uintptr_t) pool->start;
319
320                 goto done;
321             }
322         }
323
324         page = page->next;
325     } while (page);
326 }
327
328 }
329
330 page = ngx_slab_alloc_pages(pool, 1);
331
332 if (page) {
333     if (shift < ngx_slab_exact_shift) {
334         p = (page - pool->pages) << ngx_pagesize_shift;
335         bitmap = (uintptr_t *) (pool->start + p);
336
337         s = 1 << shift;
338         n = (1 << (ngx_pagesize_shift - shift)) / 8 / s;
339
340         if (n == 0) {
341             n = 1;
342         }
343
344         bitmap[0] = (2 << n) - 1;
345
346         map = (1 << (ngx_pagesize_shift - shift)) / (sizeof(uintptr_t) * 8);
347
348         for (i = 1; i < map; i++) {
349             bitmap[i] = 0;
350         }
351
352         page->slab = shift;
353         page->next = &slots[slot];
354         page->prev = (uintptr_t) &slots[slot] | NGX_SLAB_SMALL;
355
356         slots[slot].next = page;
357
358         p = ((page - pool->pages) << ngx_pagesize_shift) + s * n;

```

```

359         p += (uintptr_t) pool->start;
360
361         goto done;
362
363     } else if (shift == ngx\_slab\_exact\_shift) {
364
365         page->slab = 1;
366         page->next = &slots[slot];
367         page->prev = (uintptr_t) &slots[slot] | NGX\_SLAB\_EXACT;
368
369         slots[slot].next = page;
370
371         p = (page - pool->pages) << ngx\_pagesize\_shift;
372         p += (uintptr_t) pool->start;
373
374         goto done;
375
376     } else { /* shift > ngx_slab_exact_shift */
377
378         page->slab = ((uintptr_t) 1 << NGX\_SLAB\_MAP\_SHIFT) | shift;
379         page->next = &slots[slot];
380         page->prev = (uintptr_t) &slots[slot] | NGX\_SLAB\_BIG;
381
382         slots[slot].next = page;
383
384         p = (page - pool->pages) << ngx\_pagesize\_shift;
385         p += (uintptr_t) pool->start;
386
387         goto done;
388     }
389 }
390
391 p = 0;
392
393 done:
394
395     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_ALLOC, ngx\_cycle->log, 0, "slab alloc: %p", p);
396
397     return (void *) p;
398 }
399
400
401 void *
402 ngx\_slab\_calloc(ngx\_slab\_pool\_t *pool, size\_t size)
403 {
404     void *p;
405
406     ngx\_shmtx\_lock(&pool->mutex);
407
408     p = ngx\_slab\_calloc\_locked(pool, size);
409
410     ngx\_shmtx\_unlock(&pool->mutex);
411
412     return p;
413 }
414
415
416 void *
417 ngx\_slab\_calloc\_locked(ngx\_slab\_pool\_t *pool, size\_t size)
418 {
419     void *p;
420
421     p = ngx\_slab\_alloc\_locked(pool, size);
422     if (p) {
423         ngx\_memzero(p, size);
424     }
425
426     return p;
427 }
428
429
430 void
431 ngx\_slab\_free(ngx\_slab\_pool\_t *pool, void *p)
432 {
433     ngx\_shmtx\_lock(&pool->mutex);
434

```



```

435     ngx_slab_free_locked(pool, p);
436
437     ngx_shmtx_unlock(&pool->mutex);
438 }
439
440
441 void
442 ngx_slab_free_locked(ngx_slab_pool_t *pool, void *p)
443 {
444     size_t      size;
445     uintptr_t   slab, m, *bitmap;
446     ngx_uint_t  n, type, slot, shift, map;
447     ngx_slab_page_t *slots, *page;
448
449     ngx_log_debug1(NGX_LOG_DEBUG_ALLOC, ngx_cycle->log, 0, "slab free: %p", p);
450
451     if ((u_char *) p < pool->start || (u_char *) p > pool->end) {
452         ngx_slab_error(pool, NGX_LOG_ALERT, "ngx_slab_free(): outside of pool");
453         goto fail;
454     }
455
456     n = ((u_char *) p - pool->start) >> ngx_pagesize_shift;
457     page = &pool->pages[n];
458     slab = page->slab;
459     type = page->prev & NGX_SLAB_PAGE_MASK;
460
461     switch (type) {
462
463     case NGX_SLAB_SMALL:
464
465         shift = slab & NGX_SLAB_SHIFT_MASK;
466         size = 1 << shift;
467
468         if ((uintptr_t) p & (size - 1)) {
469             goto wrong_chunk;
470         }
471
472         n = ((uintptr_t) p & (ngx_pagesize - 1)) >> shift;
473         m = (uintptr_t) 1 << (n & (sizeof(uintptr_t) * 8 - 1));
474         n /= (sizeof(uintptr_t) * 8);
475         bitmap = (uintptr_t *)
476             ((uintptr_t) p & ~((uintptr_t) ngx_pagesize - 1));
477
478         if (bitmap[n] & m) {
479
480             if (page->next == NULL) {
481                 slots = (ngx_slab_page_t *)
482                     ((u_char *) pool + sizeof(ngx_slab_pool_t));
483                 slot = shift - pool->min_shift;
484
485                 page->next = slots[slot].next;
486                 slots[slot].next = page;
487
488                 page->prev = (uintptr_t) &slots[slot] | NGX_SLAB_SMALL;
489                 page->next->prev = (uintptr_t) page | NGX_SLAB_SMALL;
490             }
491
492             bitmap[n] &= ~m;
493
494             n = (1 << (ngx_pagesize_shift - shift)) / 8 / (1 << shift);
495
496             if (n == 0) {
497                 n = 1;
498             }
499
500             if (bitmap[0] & ~(((uintptr_t) 1 << n) - 1)) {
501                 goto done;
502             }
503
504             map = (1 << (ngx_pagesize_shift - shift)) / (sizeof(uintptr_t) * 8);
505
506             for (n = 1; n < map; n++) {
507                 if (bitmap[n]) {
508                     goto done;
509                 }
510             }
511

```

```

511         ngx_slab_free_pages(pool, page, 1);
512     }
513     goto done;
514 }
515
516 goto chunk_already_free;
517
518 case NGX_SLAB_EXACT:
519
520     m = (uintptr_t) 1 <<
521         (((uintptr_t) p & (ngx_pagesize - 1)) >> ngx_slab_exact_shift);
522     size = ngx_slab_exact_size;
523
524     if ((uintptr_t) p & (size - 1)) {
525         goto wrong_chunk;
526     }
527
528     if (slab & m) {
529         if (slab == NGX_SLAB_BUSY) {
530             slots = (ngx_slab_page_t *)
531                 ((u_char *) pool + sizeof(ngx_slab_pool_t));
532             slot = ngx_slab_exact_shift - pool->min_shift;
533
534             page->next = slots[slot].next;
535             slots[slot].next = page;
536
537             page->prev = (uintptr_t) &slots[slot] | NGX_SLAB_EXACT;
538             page->next->prev = (uintptr_t) page | NGX_SLAB_EXACT;
539         }
540
541         page->slab &= ~m;
542
543         if (page->slab) {
544             goto done;
545         }
546
547         ngx_slab_free_pages(pool, page, 1);
548
549         goto done;
550     }
551
552     goto chunk_already_free;
553
554 case NGX_SLAB_BIG:
555
556     shift = slab & NGX_SLAB_SHIFT_MASK;
557     size = 1 << shift;
558
559     if ((uintptr_t) p & (size - 1)) {
560         goto wrong_chunk;
561     }
562
563     m = (uintptr_t) 1 << (((uintptr_t) p & (ngx_pagesize - 1)) >> shift)
564         + NGX_SLAB_MAP_SHIFT;
565
566     if (slab & m) {
567
568         if (page->next == NULL) {
569             slots = (ngx_slab_page_t *)
570                 ((u_char *) pool + sizeof(ngx_slab_pool_t));
571             slot = shift - pool->min_shift;
572
573             page->next = slots[slot].next;
574             slots[slot].next = page;
575
576             page->prev = (uintptr_t) &slots[slot] | NGX_SLAB_BIG;
577             page->next->prev = (uintptr_t) page | NGX_SLAB_BIG;
578         }
579
580         page->slab &= ~m;
581
582         if (page->slab & NGX_SLAB_MAP_MASK) {
583             goto done;
584         }
585     }
586

```

```

587     ngx_slab_free_pages(pool, page, 1);
588
589     goto done;
590 }
591
592 goto chunk_already_free;
593
594 case NGX_SLAB_PAGE:
595
596     if ((uintptr_t) p & (ngx_pagesize - 1)) {
597         goto wrong_chunk;
598     }
599
600     if (slab == NGX_SLAB_PAGE_FREE) {
601         ngx_slab_error(pool, NGX_LOG_ALERT,
602             "ngx_slab_free(): page is already free");
603         goto fail;
604     }
605
606     if (slab == NGX_SLAB_PAGE_BUSY) {
607         ngx_slab_error(pool, NGX_LOG_ALERT,
608             "ngx_slab_free(): pointer to wrong page");
609         goto fail;
610     }
611
612     n = ((u_char *) p - pool->start) >> ngx_pagesize_shift;
613     size = slab & ~NGX_SLAB_PAGE_START;
614
615     ngx_slab_free_pages(pool, &pool->pages[n], size);
616
617     ngx_slab_junk(p, size << ngx_pagesize_shift);
618
619     return;
620 }
621
622 /* not reached */
623
624 return;
625
626 done:
627
628     ngx_slab_junk(p, size);
629
630     return;
631
632 wrong_chunk:
633
634     ngx_slab_error(pool, NGX_LOG_ALERT,
635         "ngx_slab_free(): pointer to wrong chunk");
636
637     goto fail;
638
639 chunk_already_free:
640
641     ngx_slab_error(pool, NGX_LOG_ALERT,
642         "ngx_slab_free(): chunk is already free");
643
644 fail:
645
646     return;
647 }
648
649
650 static ngx_slab_page_t *
651 ngx_slab_alloc_pages(ngx_slab_pool_t *pool, ngx_uint_t pages)
652 {
653     ngx_slab_page_t *page, *p;
654
655     for (page = pool->free.next; page != &pool->free; page = page->next) {
656
657         if (page->slab >= pages) {
658
659             if (page->slab > pages) {
660                 page[page->slab - 1].prev = (uintptr_t) &page[pages];
661
662                 page[pages].slab = page->slab - pages;

```

```

663         page[pages].next = page->next;
664         page[pages].prev = page->prev;
665
666         p = (ngx\_slab\_page\_t *) page->prev;
667         p->next = &page[pages];
668         page->next->prev = (uintptr\_t) &page[pages];
669
670     } else {
671         p = (ngx\_slab\_page\_t *) page->prev;
672         p->next = page->next;
673         page->next->prev = page->prev;
674     }
675
676     page->slab = pages | NGX\_SLAB\_PAGE\_START;
677     page->next = NULL;
678     page->prev = NGX\_SLAB\_PAGE;
679
680     if (--pages == 0) {
681         return page;
682     }
683
684     for (p = page + 1; pages; pages--) {
685         p->slab = NGX\_SLAB\_PAGE\_BUSY;
686         p->next = NULL;
687         p->prev = NGX\_SLAB\_PAGE;
688         p++;
689     }
690
691     return page;
692 }
693
694 if (pool->log_nomem) {
695     ngx\_slab\_error(pool, NGX\_LOG\_CRIT,
696         "ngx\_slab\_alloc() failed: no memory");
697 }
698
699 return NULL;
700 }
701
702
703
704 static void
705 ngx\_slab\_free\_pages(ngx\_slab\_pool\_t *pool, ngx\_slab\_page\_t *page,
706     ngx\_uint\_t pages)
707 {
708     ngx\_uint\_t type;
709     ngx\_slab\_page\_t *prev, *join;
710
711     page->slab = pages--;
712
713     if (pages) {
714         ngx\_memzero(&page[1], pages * sizeof(ngx\_slab\_page\_t));
715     }
716
717     if (page->next) {
718         prev = (ngx\_slab\_page\_t *) (page->prev & ~NGX\_SLAB\_PAGE\_MASK);
719         prev->next = page->next;
720         page->next->prev = page->prev;
721     }
722
723     join = page + page->slab;
724
725     if (join < pool->last) {
726         type = join->prev & NGX\_SLAB\_PAGE\_MASK;
727
728         if (type == NGX\_SLAB\_PAGE) {
729
730             if (join->next != NULL) {
731                 pages += join->slab;
732                 page->slab += join->slab;
733
734                 prev = (ngx\_slab\_page\_t *) (join->prev & ~NGX\_SLAB\_PAGE\_MASK);
735                 prev->next = join->next;
736                 join->next->prev = join->prev;
737
738                 join->slab = NGX\_SLAB\_PAGE\_FREE;

```

```

739         join->next = NULL;
740         join->prev = NGX\_SLAB\_PAGE;
741     }
742 }
743 }
744
745 if (page > pool->pages) {
746     join = page - 1;
747     type = join->prev & NGX\_SLAB\_PAGE\_MASK;
748
749     if (type == NGX\_SLAB\_PAGE) {
750
751         if (join->slab == NGX\_SLAB\_PAGE\_FREE) {
752             join = (ngx\_slab\_page\_t *) (join->prev & ~NGX\_SLAB\_PAGE\_MASK);
753         }
754
755         if (join->next != NULL) {
756             pages += join->slab;
757             join->slab += page->slab;
758
759             prev = (ngx\_slab\_page\_t *) (join->prev & ~NGX\_SLAB\_PAGE\_MASK);
760             prev->next = join->next;
761             join->next->prev = join->prev;
762
763             page->slab = NGX\_SLAB\_PAGE\_FREE;
764             page->next = NULL;
765             page->prev = NGX\_SLAB\_PAGE;
766
767             page = join;
768         }
769     }
770 }
771
772 if (pages) {
773     page[pages].prev = (uintptr\_t) page;
774 }
775
776 page->prev = (uintptr\_t) &pool->free;
777 page->next = pool->free.next;
778
779 page->next->prev = (uintptr\_t) page;
780
781 pool->free.next = page;
782 }
783
784
785 static void
786 ngx\_slab\_error(ngx\_slab\_pool\_t *pool, ngx\_uint\_t level, char *text)
787 {
788     ngx\_log\_error(level, ngx\_cycle->log, 0, "%s%s", text, pool->log_ctx);
789 }

```

[One Level Up](#)

[Top Level](#)

# src/os/unix/nginx\_darwin\_config.h - nginx-1.7.10

## Macros defined

- [IOV\\_MAX](#)
- [NGX\\_HAVE\\_CASELESS\\_FILESYSTEM](#)
- [NGX\\_HAVE\\_DEBUG\\_MALLOC](#)
- [NGX\\_HAVE\\_INHERITED\\_NONBLOCK](#)
- [NGX\\_HAVE\\_OS\\_SPECIFIC\\_INIT](#)
- [NGX\\_LISTEN\\_BACKLOG](#)
- [\\_NGX\\_DARWIN\\_CONFIG\\_H\\_INCLUDED](#)

## Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #ifndef \_NGX\_DARWIN\_CONFIG\_H\_INCLUDED
9  #define \_NGX\_DARWIN\_CONFIG\_H\_INCLUDED
10
11
12  #include <sys/types.h>
13  #include <sys/time.h>
14  #include <unistd.h>
15  #include <inttypes.h>
16  #include <stdarg.h>
17  #include <stddef.h>          /* offsetof() */
18  #include <stdio.h>
19  #include <stdlib.h>
20  #include <ctype.h>
21  #include <errno.h>
22  #include <string.h>
23  #include <signal.h>
24  #include <pwd.h>
25  #include <grp.h>
26  #include <dirent.h>
27  #include <glob.h>
28  #include <sys/mount.h>      /* statfs() */
29
30  #include <sys/filio.h>      /* FIONBIO */
31  #include <sys/ioctl.h>
32  #include <sys/uio.h>
33  #include <sys/stat.h>
34  #include <fcntl.h>
35
36  #include <sys/wait.h>
37  #include <sys/mman.h>
38  #include <sys/resource.h>
39  #include <sched.h>
40
41  #include <sys/socket.h>
42  #include <netinet/in.h>
43  #include <netinet/tcp.h>   /* TCP_NODELAY */
44  #include <arpa/inet.h>
45  #include <netdb.h>
46  #include <sys/un.h>
47
48  #include <sys/sysctl.h>
49  #include <xlocale.h>
```

```
50
51
52 #ifndef IOV_MAX
53 #define IOV_MAX 64
54 #endif
55
56
57 #include <ngx_auto_config.h>
58
59
60 #if (NGX_HAVE_POSIX_SEM)
61 #include <semaphore.h>
62 #endif
63
64
65 #if (NGX_HAVE_POLL)
66 #include <poll.h>
67 #endif
68
69
70 #if (NGX_HAVE_KQUEUE)
71 #include <sys/event.h>
72 #endif
73
74
75 #define NGX_LISTEN_BACKLOG -1
76
77
78 #ifndef NGX_HAVE_INHERITED_NONBLOCK
79 #define NGX_HAVE_INHERITED_NONBLOCK 1
80 #endif
81
82
83 #ifndef NGX_HAVE_CASELESS_FILESYSTEM
84 #define NGX_HAVE_CASELESS_FILESYSTEM 1
85 #endif
86
87
88 #define NGX_HAVE_OS_SPECIFIC_INIT 1
89 #define NGX_HAVE_DEBUG_MALLOC 1
90
91
92 extern char **environ;
93
94
95 #endif /* NGX_DARWIN_CONFIG_H_INCLUDED */
```

[One Level Up](#)

[Top Level](#)

# src/event/nginx\_event\_posted.h - nginx-1.7.10

## Macros defined

- [\\_NGX\\_EVENT\\_POSTED\\_H\\_INCLUDED](#)
- [ngx\\_delete\\_posted\\_event](#)
- [ngx\\_post\\_event](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_EVENT\_POSTED\_H\_INCLUDED
9 #define \_NGX\_EVENT\_POSTED\_H\_INCLUDED
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14 #include <ngx_event.h>
15
16
17 #define ngx_post_event(ev, q) \
18 \
19     if (!ev->posted) { \
20         ev->posted = 1; \
21         ngx\_queue\_insert\_tail(q, &ev->queue); \
22 \
23         ngx\_log\_debug1(NGX_LOG_DEBUG_CORE, ev->log, 0, "post event %p", ev); \
24 \
25     } else { \
26         ngx\_log\_debug1(NGX_LOG_DEBUG_CORE, ev->log, 0, \
27             "update posted event %p", ev); \
28     }
29
30
31 #define ngx_delete_posted_event(ev) \
32 \
33     ev->posted = 0; \
34     ngx\_queue\_remove(&ev->queue); \
35 \
36     ngx\_log\_debug1(NGX_LOG_DEBUG_CORE, ev->log, 0, \
37         "delete posted event %p", ev);
38
39
40
41 void ngx\_event\_process\_posted(ngx\_cycle\_t *cycle, ngx\_queue\_t *posted);
42
43
44 extern ngx\_queue\_t ngx\_posted\_accept\_events;
45 extern ngx\_queue\_t ngx\_posted\_events;
46
47
48 #endif /* \_NGX\_EVENT\_POSTED\_H\_INCLUDED */
```



## src/event/nginx\_event\_posted.c - nginx-1.7.10

### Global variables defined

- [ngx\\_posted\\_accept\\_events](#)
- [ngx\\_posted\\_events](#)

### Functions defined

- [ngx\\_event\\_process\\_posted](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 ngx_queue_t  ngx_posted_accept_events;
14 ngx_queue_t  ngx_posted_events;
15
16
17 void
18 ngx_event_process_posted(ngx_cycle_t *cycle, ngx_queue_t *posted)
19 {
20     ngx_queue_t  *q;
21     ngx_event_t  *ev;
22
23     while (!ngx_queue_empty(posted)) {
24
25         q = ngx_queue_head(posted);
26         ev = ngx_queue_data(q, ngx_event_t, queue);
27
28         ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
29                      "posted event %p", ev);
30
31         ngx_delete_posted_event(ev);
32
33         ev->handler(ev);
34     }
35 }
```

# src/core/nginx\_crc32.h - nginx-1.7.10

## Functions defined

- [ngx\\_crc32\\_long](#)
- [ngx\\_crc32\\_short](#)
- [ngx\\_crc32\\_update](#)

## Macros defined

- [\\_NGX\\_CRC32\\_H\\_INCLUDED](#)
- [ngx\\_crc32\\_final](#)
- [ngx\\_crc32\\_init](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_CRC32\_H\_INCLUDED
9 #define \_NGX\_CRC32\_H\_INCLUDED
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 extern uint32_t \*ngx\_crc32\_table\_short;
17 extern uint32_t ngx\_crc32\_table256[];
18
19
20 static ngx\_inline uint32_t
21 ngx_crc32_short(u_char *p, size_t len)
22 {
23     u_char    c;
24     uint32_t  crc;
25
26     crc = 0xffffffff;
27
28     while (len--) {
29         c = *p++;
30         crc = ngx\_crc32\_table\_short[(crc ^ (c & 0xf)) & 0xf] ^ (crc >> 4);
31         crc = ngx\_crc32\_table\_short[(crc ^ (c >> 4)) & 0xf] ^ (crc >> 4);
32     }
33
34     return crc ^ 0xffffffff;
35 }
36
37
38 static ngx\_inline uint32_t
39 ngx_crc32_long(u_char *p, size_t len)
40 {
41     uint32_t  crc;
42
43     crc = 0xffffffff;
44
45     while (len--) {
46         crc = ngx\_crc32\_table256[(crc ^ *p++) & 0xff] ^ (crc >> 8);
47     }
48 }
```

```
49     return crc ^ 0xffffffff;
50 }
51
52
53 #define ngx_crc32_init(crc) \
54     crc = 0xffffffff
55
56
57 static ngx_inline void
58 ngx_crc32_update(uint32_t *crc, u_char *p, size_t len)
59 {
60     uint32_t c;
61
62     c = *crc;
63
64     while (len--) {
65         c = ngx_crc32_table256[(c ^ *p++) & 0xff] ^ (c >> 8);
66     }
67
68     *crc = c;
69 }
70
71
72 #define ngx_crc32_final(crc) \
73     crc ^= 0xffffffff
74
75
76 ngx_int_t ngx_crc32_table_init(void);
77
78
79 #endif /* NGX_CRC32_H_INCLUDED */
```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_crc32.c - nginx-1.7.10

### Global variables defined

- [ngx\\_crc32\\_table16](#)
- [ngx\\_crc32\\_table256](#)
- [ngx\\_crc32\\_table\\_short](#)

### Functions defined

- [ngx\\_crc32\\_table\\_init](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10
11
12 /*
13  * The code and lookup tables are based on the algorithm
14  * described at http://www.w3.org/TR/PNG/
15  *
16  * The 256 element lookup table takes 1024 bytes, and it may be completely
17  * cached after processing about 30-60 bytes of data. So for short data
18  * we use the 16 element lookup table that takes only 64 bytes and align it
19  * to CPU cache line size. Of course, the small table adds code inside
20  * CRC32 loop, but the cache misses overhead is bigger than overhead of
21  * the additional code. For example, ngx\_crc32\_short\(\) of 16 bytes of data
22  * takes half as much CPU clocks than ngx\_crc32\_long\(\).
23  */
24
25
26 static uint32_t ngx_crc32_table16[] = {
27     0x00000000, 0x1db71064, 0x3b6e20c8, 0x26d930ac,
28     0x76dc4190, 0x6b6b51f4, 0x4db26158, 0x5005713c,
29     0xedb88320, 0xf00f9344, 0xd6d6a3e8, 0xcb61b38c,
30     0x9b64c2b0, 0x86d3d2d4, 0xa00ae278, 0xbdbdf21c
31 };
32
33
34 uint32_t ngx_crc32_table256[] = {
35     0x00000000, 0x77073096, 0xee0e612c, 0x990951ba,
36     0x076dc419, 0x706af48f, 0xe963a535, 0x9e6495a3,
37     0x0edb8832, 0x79dcb8a4, 0xe0d5e91e, 0x97d2d988,
38     0x09b64c2b, 0x7eb17cbd, 0xe7b82d07, 0x90bf1d91,
39     0x1db71064, 0x6ab020f2, 0xf3b97148, 0x84be41de,
40     0x1adad47d, 0x6ddde4eb, 0xfd4d4b551, 0x83d385c7,
41     0x136c9856, 0x646ba8c0, 0xfd62f97a, 0x8a65c9ec,
42     0x14015c4f, 0x63066cd9, 0xfa0f3d63, 0x8d080df5,
43     0x3b6e20c8, 0x4c69105e, 0xd56041e4, 0xa2677172,
44     0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b,
45     0x35b5a8fa, 0x42b2986c, 0xdbbbc9d6, 0xacbcf940,
46     0x32d86ce3, 0x45df5c75, 0xdcd60dcf, 0xabd13d59,
47     0x26d930ac, 0x51de003a, 0xc8d75180, 0xbfdb06116,
48     0x21b4f4b5, 0x56b3c423, 0xcfba9599, 0xb8bda50f,
49     0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924,
50     0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d,
51     0x76dc4190, 0x01db7106, 0x98d220bc, 0xefd5102a,
52     0x71b18589, 0x06b6b51f, 0x9fbfe4a5, 0xe8b8d433,
```

```

53     0x7807c9a2, 0x0f00f934, 0x9609a88e, 0xe10e9818,
54     0x7f6a0dbb, 0x086d3d2d, 0x91646c97, 0xe6635c01,
55     0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0xf262004e,
56     0x6c0695ed, 0x1b01a57b, 0x8208f4c1, 0xf50fc457,
57     0x65b0d9c6, 0x12b7e950, 0x8bbeb8ea, 0xfcb9887c,
58     0x62dd1ddf, 0x15da2d49, 0x8cd37cf3, 0xfbd44c65,
59     0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2,
60     0x4adfa541, 0x3dd895d7, 0xa4d1c46d, 0xd3d6f4fb,
61     0x4369e96a, 0x346ed9fc, 0xad678846, 0xda60b8d0,
62     0x44042d73, 0x33031de5, 0xaa0a4c5f, 0xdd0d7cc9,
63     0x5005713c, 0x270241aa, 0xbe0b1010, 0xc90c2086,
64     0x5768b525, 0x206f85b3, 0xb966d409, 0xce61e49f,
65     0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4,
66     0x59b33d17, 0x2eb40d81, 0xb7bd5c3b, 0xc0ba6cad,
67     0xedb88320, 0x9abfb3b6, 0x03b6e20c, 0x74b1d29a,
68     0xead54739, 0x9dd277af, 0x04db2615, 0x73dc1683,
69     0xe3630b12, 0x94643b84, 0x0d6d6a3e, 0x7a6a5aa8,
70     0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0x7d079eb1,
71     0xf00f9344, 0x8708a3d2, 0x1e01f268, 0x6906c2fe,
72     0xf762575d, 0x806567cb, 0x196c3671, 0x6e6b06e7,
73     0xfed41b76, 0x89d32be0, 0x10da7a5a, 0x67dd4acc,
74     0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5,
75     0xd6d6a3e8, 0xa1d1937e, 0x38d8c2c4, 0x4fdfff252,
76     0xd1bb67f1, 0xa6bc5767, 0x3fb506dd, 0x48b2364b,
77     0xd80d2bda, 0xaf0a1b4c, 0x36034af6, 0x41047a60,
78     0xdf60efc3, 0xa867df55, 0x316e8eef, 0x4669be79,
79     0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236,
80     0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f,
81     0xc5ba3bbe, 0xb2bd0b28, 0x2bb45a92, 0x5cb36a04,
82     0xc2d7ffa7, 0xb5d0cf31, 0x2cd99e8b, 0x5bdeae1d,
83     0x9b64c2b0, 0xec63f226, 0x756aa39c, 0x026d930a,
84     0x9c0906a9, 0xeb0e363f, 0x72076785, 0x05005713,
85     0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0x0cb61b38,
86     0x92d28e9b, 0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21,
87     0x86d3d2d4, 0xf1d4e242, 0x68ddb3f8, 0x1fda836e,
88     0x81be16cd, 0xf6b9265b, 0x6fb077e1, 0x18b74777,
89     0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c,
90     0x8f659eff, 0xf862ae69, 0x616bffd3, 0x166ccf45,
91     0xa00ae278, 0xd70dd2ee, 0x4e048354, 0x3903b3c2,
92     0xa7672661, 0xd06016f7, 0x4969474d, 0x3e6e77db,
93     0xaed16a4a, 0xd9d65adc, 0x40df0b66, 0x37d83bf0,
94     0xa9bcae53, 0xdeb99ec5, 0x47b2cf7f, 0x30b5ffe9,
95     0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6,
96     0xbad03605, 0xcdd70693, 0x54de5729, 0x23d967bf,
97     0xb3667a2e, 0xc4614ab8, 0x5d681b02, 0x2a6f2b94,
98     0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d
99 };
100
101
102 uint32_t *ngx_crc32_table_short = ngx_crc32_table16;
103
104
105 ngx_int_t
106 ngx_crc32_table_init(void)
107 {
108     void *p;
109
110     if (((uintptr_t) ngx_crc32_table_short
111         & ~(uintptr_t) ngx_cacheline_size - 1)
112         == (uintptr_t) ngx_crc32_table_short)
113     {
114         return NGX_OK;
115     }
116
117     p = ngx_alloc(16 * sizeof(uint32_t) + ngx_cacheline_size, ngx_cycle->log);
118     if (p == NULL) {
119         return NGX_ERROR;
120     }
121
122     p = ngx_align_ptr(p, ngx_cacheline_size);
123
124     ngx_memcpy(p, ngx_crc32_table16, 16 * sizeof(uint32_t));
125
126     ngx_crc32_table_short = p;
127
128     return NGX_OK;

```

[One Level Up](#)

[Top Level](#)

## src/event/nginx\_event\_openssl\_stapling.c - nginx-1.7.10

### Data types defined

- [ngx\\_ssl\\_ocsp\\_ctx\\_s](#)
- [ngx\\_ssl\\_ocsp\\_ctx\\_t](#)
- [ngx\\_ssl\\_stapling\\_t](#)

### Functions defined

- [ngx\\_ssl\\_certificate\\_status\\_callback](#)
- [ngx\\_ssl\\_ocsp\\_connect](#)
- [ngx\\_ssl\\_ocsp\\_create\\_request](#)
- [ngx\\_ssl\\_ocsp\\_done](#)
- [ngx\\_ssl\\_ocsp\\_dummy\\_handler](#)
- [ngx\\_ssl\\_ocsp\\_error](#)
- [ngx\\_ssl\\_ocsp\\_log\\_error](#)
- [ngx\\_ssl\\_ocsp\\_parse\\_header\\_line](#)
- [ngx\\_ssl\\_ocsp\\_parse\\_status\\_line](#)
- [ngx\\_ssl\\_ocsp\\_process\\_body](#)
- [ngx\\_ssl\\_ocsp\\_process\\_headers](#)
- [ngx\\_ssl\\_ocsp\\_process\\_status\\_line](#)
- [ngx\\_ssl\\_ocsp\\_read\\_handler](#)
- [ngx\\_ssl\\_ocsp\\_request](#)
- [ngx\\_ssl\\_ocsp\\_resolve\\_handler](#)
- [ngx\\_ssl\\_ocsp\\_start](#)
- [ngx\\_ssl\\_ocsp\\_write\\_handler](#)
- [ngx\\_ssl\\_stapling](#)
- [ngx\\_ssl\\_stapling](#)
- [ngx\\_ssl\\_stapling\\_cleanup](#)
- [ngx\\_ssl\\_stapling\\_file](#)
- [ngx\\_ssl\\_stapling\\_issuer](#)
- [ngx\\_ssl\\_stapling\\_ocsp\\_handler](#)
- [ngx\\_ssl\\_stapling\\_resolver](#)
- [ngx\\_ssl\\_stapling\\_resolver](#)

- [ngx\\_ssl\\_stapling\\_responder](#)
- [ngx\\_ssl\\_stapling\\_update](#)

## Source code

```

1
2  /*
3  * Copyright (C) Maxim Dounin
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_event.h>
11 #include <ngx_event_connect.h>
12
13
14 #if (!defined OPENSSSL_NO_OCSP && defined SSL_CTRL_SET_TLSEXT_STATUS_REQ_CB)
15
16
17 typedef struct {
18     ngx_str_t          staple;
19     ngx_msec_t        timeout;
20
21     ngx_resolver_t    *resolver;
22     ngx_msec_t        resolver_timeout;
23
24     ngx_addr_t        *addrs;
25     ngx_str_t         host;
26     ngx_str_t         uri;
27     in_port_t         port;
28
29     SSL_CTX           *ssl_ctx;
30
31     X509               *cert;
32     X509               *issuer;
33
34     time_t            valid;
35
36     unsigned          verify:1;
37     unsigned          loading:1;
38 } ngx_ssl_stapling_t;
39
40
41 typedef struct ngx_ssl_ocsp_ctx_s ngx_ssl_ocsp_ctx_t;
42
43 struct ngx_ssl_ocsp_ctx_s {
44     X509               *cert;
45     X509               *issuer;
46
47     ngx_uint_t        naddrs;
48
49     ngx_addr_t        *addrs;
50     ngx_str_t         host;
51     ngx_str_t         uri;
52     in_port_t         port;
53
54     ngx_resolver_t    *resolver;
55     ngx_msec_t        resolver_timeout;
56
57     ngx_msec_t        timeout;
58
59     void              (*handler)(ngx_ssl_ocsp_ctx_t *r);
60     void              *data;
61
62     ngx_buf_t         *request;
63     ngx_buf_t         *response;
64     ngx_peer_connection_t peer;
65
66     ngx_int_t         (*process)(ngx_ssl_ocsp_ctx_t *r);
67
68     ngx_uint_t        state;

```



```

69     ngx_uint_t      code;
70     ngx_uint_t      count;
71
72
73     ngx_uint_t      done;
74
75     u_char           *header_name_start;
76     u_char           *header_name_end;
77     u_char           *header_start;
78     u_char           *header_end;
79
80     ngx_pool_t      *pool;
81     ngx_log_t       *log;
82 };
83
84
85 static ngx_int_t ngx_ssl_stapling_file(ngx_conf_t *cf, ngx_ssl_t *ssl,
86     ngx_str_t *file);
87 static ngx_int_t ngx_ssl_stapling_issuer(ngx_conf_t *cf, ngx_ssl_t *ssl);
88 static ngx_int_t ngx_ssl_stapling_responder(ngx_conf_t *cf, ngx_ssl_t *ssl,
89     ngx_str_t *responder);
90
91 static int ngx_ssl_certificate_status_callback(ngx_ssl_conn_t *ssl_conn,
92     void *data);
93 static void ngx_ssl_stapling_update(ngx_ssl_stapling_t *staple);
94 static void ngx_ssl_stapling_ocsp_handler(ngx_ssl_ocsp_ctx_t *ctx);
95
96 static void ngx_ssl_stapling_cleanup(void *data);
97
98 static ngx_ssl_ocsp_ctx_t *ngx_ssl_ocsp_start(void);
99 static void ngx_ssl_ocsp_done(ngx_ssl_ocsp_ctx_t *ctx);
100 static void ngx_ssl_ocsp_request(ngx_ssl_ocsp_ctx_t *ctx);
101 static void ngx_ssl_ocsp_resolve_handler(ngx_resolver_ctx_t *resolve);
102 static void ngx_ssl_ocsp_connect(ngx_ssl_ocsp_ctx_t *ctx);
103 static void ngx_ssl_ocsp_write_handler(ngx_event_t *wev);
104 static void ngx_ssl_ocsp_read_handler(ngx_event_t *rev);
105 static void ngx_ssl_ocsp_dummy_handler(ngx_event_t *ev);
106
107 static ngx_int_t ngx_ssl_ocsp_create_request(ngx_ssl_ocsp_ctx_t *ctx);
108 static ngx_int_t ngx_ssl_ocsp_process_status_line(ngx_ssl_ocsp_ctx_t *ctx);
109 static ngx_int_t ngx_ssl_ocsp_parse_status_line(ngx_ssl_ocsp_ctx_t *ctx);
110 static ngx_int_t ngx_ssl_ocsp_process_headers(ngx_ssl_ocsp_ctx_t *ctx);
111 static ngx_int_t ngx_ssl_ocsp_parse_header_line(ngx_ssl_ocsp_ctx_t *ctx);
112 static ngx_int_t ngx_ssl_ocsp_process_body(ngx_ssl_ocsp_ctx_t *ctx);
113
114 static u_char *ngx_ssl_ocsp_log_error(ngx_log_t *log, u_char *buf, size_t len);
115
116
117 ngx_int_t
118 ngx_ssl_stapling(ngx_conf_t *cf, ngx_ssl_t *ssl, ngx_str_t *file,
119     ngx_str_t *responder, ngx_uint_t verify)
120 {
121     ngx_int_t      rc;
122     ngx_pool_cleanup_t *cfn;
123     ngx_ssl_stapling_t *staple;
124
125     staple = ngx_palloc(cf->pool, sizeof(ngx_ssl_stapling_t));
126     if (staple == NULL) {
127         return NGX_ERROR;
128     }
129
130     cfn = ngx_pool_cleanup_add(cf->pool, 0);
131     if (cfn == NULL) {
132         return NGX_ERROR;
133     }
134
135     cfn->handler = ngx_ssl_stapling_cleanup;
136     cfn->data = staple;
137
138     if (SSL_CTX_set_ex_data(ssl->ctx, ngx_ssl_stapling_index, staple)
139         == 0)
140     {
141         ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
142             "SSL_CTX_set_ex_data() failed");
143         return NGX_ERROR;
144     }

```

```

145 staple->ssl_ctx = ssl->ctx;
146 staple->timeout = 60000;
147 staple->verify = verify;
148
149 if (file->len) {
150     /* use OCSP response from the file */
151
152     if (ngx\_ssl\_stapling\_file(cf, ssl, file) != NGX\_OK) {
153         return NGX\_ERROR;
154     }
155
156     goto done;
157 }
158
159 rc = ngx\_ssl\_stapling\_issuer(cf, ssl);
160
161 if (rc == NGX\_DECLINED) {
162     return NGX\_OK;
163 }
164
165 if (rc != NGX\_OK) {
166     return NGX\_ERROR;
167 }
168
169 rc = ngx\_ssl\_stapling\_responder(cf, ssl, responder);
170
171 if (rc == NGX\_DECLINED) {
172     return NGX\_OK;
173 }
174
175 if (rc != NGX\_OK) {
176     return NGX\_ERROR;
177 }
178
179 done:
180
181 SSL_CTX_set_tlsext_status_cb(ssl->ctx, ngx\_ssl\_certificate\_status\_callback);
182 SSL_CTX_set_tlsext_status_arg(ssl->ctx, staple);
183
184 return NGX\_OK;
185 }
186
187
188
189 static ngx\_int\_t
190 ngx\_ssl\_stapling\_file(ngx\_conf\_t *cf, ngx\_ssl\_t *ssl, ngx\_str\_t *file)
191 {
192     BIO *bio;
193     int len;
194     u_char *p, *buf;
195     OCSP_RESPONSE *response;
196     ngx\_ssl\_stapling\_t *staple;
197
198     staple = SSL_CTX_get_ex_data(ssl->ctx, ngx\_ssl\_stapling\_index);
199
200     if (ngx\_conf\_full\_name(cf->cycle, file, 1) != NGX\_OK) {
201         return NGX\_ERROR;
202     }
203
204     bio = BIO_new_file((char *) file->data, "r");
205     if (bio == NULL) {
206         ngx\_ssl\_error(NGX\_LOG\_EMERG, ssl->log, 0,
207             "BIO_new_file(\"%s\") failed", file->data);
208         return NGX\_ERROR;
209     }
210
211     response = d2i_OCSP_RESPONSE_bio(bio, NULL);
212     if (response == NULL) {
213         ngx\_ssl\_error(NGX\_LOG\_EMERG, ssl->log, 0,
214             "d2i_OCSP_RESPONSE_bio(\"%s\") failed", file->data);
215         BIO_free(bio);
216         return NGX\_ERROR;
217     }
218
219     len = i2d_OCSP_RESPONSE(response, NULL);
220     if (len <= 0) {

```

```

221     ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
222                 "i2d_OCSP_RESPONSE(\"%s\") failed", file->data);
223     goto failed;
224 }
225
226 buf = ngx_alloc(len, ssl->log);
227 if (buf == NULL) {
228     goto failed;
229 }
230
231 p = buf;
232 len = i2d_OCSP_RESPONSE(response, &p);
233 if (len <= 0) {
234     ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
235                 "i2d_OCSP_RESPONSE(\"%s\") failed", file->data);
236     ngx_free(buf);
237     goto failed;
238 }
239
240 OCSP_RESPONSE_free(response);
241 BIO_free(bio);
242
243 staple->staple.data = buf;
244 staple->staple.len = len;
245
246 return NGX_OK;
247
248 failed:
249
250 OCSP_RESPONSE_free(response);
251 BIO_free(bio);
252
253 return NGX_ERROR;
254 }
255
256
257 static ngx_int_t
258 ngx_ssl_stapling_issuer(ngx_conf_t *cf, ngx_ssl_t *ssl)
259 {
260     int            i, n, rc;
261     X509           *cert, *issuer;
262     X509_STORE     *store;
263     X509_STORE_CTX *store_ctx;
264     STACK_OF(X509) *chain;
265     ngx_ssl_stapling_t *staple;
266
267     staple = SSL_CTX_get_ex_data(ssl->ctx, ngx_ssl_stapling_index);
268     cert = SSL_CTX_get_ex_data(ssl->ctx, ngx_ssl_certificate_index);
269
270     #if OPENSSL_VERSION_NUMBER >= 0x10001000L
271     SSL_CTX_get_extra_chain_certs(ssl->ctx, &chain);
272     #else
273     chain = ssl->ctx->extra_certs;
274     #endif
275
276     n = sk_X509_num(chain);
277
278     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, ssl->log, 0,
279                 "SSL get issuer: %d extra certs", n);
280
281     for (i = 0; i < n; i++) {
282         issuer = sk_X509_value(chain, i);
283         if (X509_check_issued(issuer, cert) == X509_V_OK) {
284             CRYPTO_add(&issuer->references, 1, CRYPTO_LOCK_X509);
285
286             ngx_log_debug1(NGX_LOG_DEBUG_EVENT, ssl->log, 0,
287                 "SSL get issuer: found %p in extra certs", issuer);
288
289             staple->cert = cert;
290             staple->issuer = issuer;
291
292             return NGX_OK;
293         }
294     }
295
296     store = SSL_CTX_get_cert_store(ssl->ctx);

```

```

297     if (store == NULL) {
298         ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
299             "SSL_CTX_get_cert_store() failed");
300         return NGX_ERROR;
301     }
302
303     store_ctx = X509_STORE_CTX_new();
304     if (store_ctx == NULL) {
305         ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
306             "X509_STORE_CTX_new() failed");
307         return NGX_ERROR;
308     }
309
310     if (X509_STORE_CTX_init(store_ctx, store, NULL, NULL) == 0) {
311         ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
312             "X509_STORE_CTX_init() failed");
313         return NGX_ERROR;
314     }
315
316     rc = X509_STORE_CTX_get1_issuer(&issuer, store_ctx, cert);
317
318     if (rc == -1) {
319         ngx_ssl_error(NGX_LOG_EMERG, ssl->log, 0,
320             "X509_STORE_CTX_get1_issuer() failed");
321         X509_STORE_CTX_free(store_ctx);
322         return NGX_ERROR;
323     }
324
325     if (rc == 0) {
326         ngx_log_error(NGX_LOG_WARN, ssl->log, 0,
327             "\"ssl_stapling\" ignored, issuer certificate not found");
328         X509_STORE_CTX_free(store_ctx);
329         return NGX_DECLINED;
330     }
331
332     X509_STORE_CTX_free(store_ctx);
333
334     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, ssl->log, 0,
335         "SSL get issuer: found %p in cert store", issuer);
336
337     staple->cert = cert;
338     staple->issuer = issuer;
339
340     return NGX_OK;
341 }
342
343
344 static ngx_int_t
345 ngx_ssl_stapling_responder(ngx_conf_t *cf, ngx_ssl_t *ssl, ngx_str_t *responder)
346 {
347     ngx_url_t          u;
348     char               *s;
349     ngx_ssl_stapling_t *staple;
350     STACK_OF(OPENSSL_STRING) *aia;
351
352     staple = SSL_CTX_get_ex_data(ssl->ctx, ngx_ssl_stapling_index);
353
354     if (responder->len == 0) {
355
356         /* extract OCSP responder URL from certificate */
357
358         aia = X509_get1_ocsp(staple->cert);
359         if (aia == NULL) {
360             ngx_log_error(NGX_LOG_WARN, ssl->log, 0,
361                 "\"ssl_stapling\" ignored, "
362                 "no OCSP responder URL in the certificate");
363             return NGX_DECLINED;
364         }
365
366         #if OPENSSL_VERSION_NUMBER >= 0x10000000L
367         s = sk_OPENSSL_STRING_value(aia, 0);
368         #else
369         s = sk_value(aia, 0);
370         #endif
371         if (s == NULL) {
372             ngx_log_error(NGX_LOG_WARN, ssl->log, 0,

```

```

373         "\ssl_stapling\" ignored, "
374         "no OCSP responder URL in the certificate");
375     X509_email_free(aia);
376     return NGX_DECLINED;
377 }
378
379     responder->len = ngx_strlen(s);
380     responder->data = ngx_palloc(cf->pool, responder->len);
381     if (responder->data == NULL) {
382         X509_email_free(aia);
383         return NGX_ERROR;
384     }
385
386     ngx_memcpy(responder->data, s, responder->len);
387     X509_email_free(aia);
388 }
389
390 ngx_memzero(&u, sizeof(ngx_url_t));
391
392     u.url = *responder;
393     u.default_port = 80;
394     u.uri_part = 1;
395
396     if (u.url.len > 7
397         && ngx_strncasecmp(u.url.data, (u_char *) "http://", 7) == 0)
398     {
399         u.url.len -= 7;
400         u.url.data += 7;
401
402     } else {
403         ngx_log_error(NGX_LOG_WARN, ssl->log, 0,
404             "\ssl_stapling\" ignored, "
405             "invalid URL prefix in OCSP responder \"%V\"", &u.url);
406         return NGX_DECLINED;
407     }
408
409     if (ngx_parse_url(cf->pool, &u) != NGX_OK) {
410         if (u.err) {
411             ngx_log_error(NGX_LOG_WARN, ssl->log, 0,
412                 "\ssl_stapling\" ignored, "
413                 "%s in OCSP responder \"%V\"", u.err, &u.url);
414             return NGX_DECLINED;
415         }
416
417         return NGX_ERROR;
418     }
419
420     staple->addrs = u.addrs;
421     staple->host = u.host;
422     staple->uri = u.uri;
423     staple->port = u.port;
424
425     if (staple->uri.len == 0) {
426         ngx_str_set(&staple->uri, "/");
427     }
428
429     return NGX_OK;
430 }
431
432
433 ngx_int_t
434 ngx_ssl_stapling_resolver(ngx_conf_t *cf, ngx_ssl_t *ssl,
435     ngx_resolver_t *resolver, ngx_msec_t resolver_timeout)
436 {
437     ngx_ssl_stapling_t *staple;
438
439     staple = SSL_CTX_get_ex_data(ssl->ctx, ngx_ssl_stapling_index);
440
441     staple->resolver = resolver;
442     staple->resolver_timeout = resolver_timeout;
443
444     return NGX_OK;
445 }
446
447
448 static int

```

```

449 ngx_ssl_certificate_status_callback(ngx_ssl_conn_t *ssl_conn, void *data)
450 {
451     int rc;
452     u_char *p;
453     ngx_connection_t *c;
454     ngx_ssl_stapling_t *staple;
455
456     c = ngx_ssl_get_connection(ssl_conn);
457
458     ngx_log_debug0(NGX_LOG_DEBUG_EVENT, c->log, 0,
459                 "SSL certificate status callback");
460
461     staple = data;
462     rc = SSL_TLSEXT_ERR_NOACK;
463
464     if (staple->staple.len) {
465         /* we have to copy ocsf response as OpenSSL will free it by itself */
466
467         p = OPENSSL_malloc(staple->staple.len);
468         if (p == NULL) {
469             ngx_ssl_error(NGX_LOG_ALERT, c->log, 0, "OPENSSL_malloc() failed");
470             return SSL_TLSEXT_ERR_NOACK;
471         }
472
473         ngx_memcpy(p, staple->staple.data, staple->staple.len);
474
475         SSL_set_tlsext_status_ocsp_resp(ssl_conn, p, staple->staple.len);
476
477         rc = SSL_TLSEXT_ERR_OK;
478     }
479
480     ngx_ssl_stapling_update(staple);
481
482     return rc;
483 }
484
485
486 static void
487 ngx_ssl_stapling_update(ngx_ssl_stapling_t *staple)
488 {
489     ngx_ssl_ocsp_ctx_t *ctx;
490
491     if (staple->host.len == 0
492         || staple->loading || staple->valid >= ngx_time())
493     {
494         return;
495     }
496
497     staple->loading = 1;
498
499     ctx = ngx_ssl_ocsp_start();
500     if (ctx == NULL) {
501         return;
502     }
503
504     ctx->cert = staple->cert;
505     ctx->issuer = staple->issuer;
506
507     ctx->addrs = staple->addrs;
508     ctx->host = staple->host;
509     ctx->uri = staple->uri;
510     ctx->port = staple->port;
511     ctx->timeout = staple->timeout;
512
513     ctx->resolver = staple->resolver;
514     ctx->resolver_timeout = staple->resolver_timeout;
515
516     ctx->handler = ngx_ssl_stapling_ocsp_handler;
517     ctx->data = staple;
518
519     ngx_ssl_ocsp_request(ctx);
520
521     return;
522 }
523
524

```

```

525 static void
526 ngx_ssl_stapling_ocsp_handler(ngx_ssl_ocsp_ctx_t *ctx)
527 {
528     #if OPENSSSL_VERSION_NUMBER >= 0x0090707fL
529         const
530     #endif
531         u_char          *p;
532         int             n;
533         size_t          len;
534         ngx_str_t       response;
535         X509_STORE       *store;
536         STACK_OF(X509)  *chain;
537         OCSP_CERTID     *id;
538         OCSP_RESPONSE   *ocsp;
539         OCSP_BASICRESP  *basic;
540         ngx_ssl_stapling_t *staple;
541         ASN1_GENERALIZEDTIME *thisupdate, *nextupdate;
542
543     staple = ctx->data;
544     ocsp = NULL;
545     basic = NULL;
546     id = NULL;
547
548     if (ctx->code != 200) {
549         goto error;
550     }
551
552     /* check the response */
553
554     len = ctx->response->last - ctx->response->pos;
555     p = ctx->response->pos;
556
557     ocsp = d2i_OCSP_RESPONSE(NULL, &p, len);
558     if (ocsp == NULL) {
559         ngx_ssl_error(NGX_LOG_ERR, ctx->log, 0,
560             "d2i_OCSP_RESPONSE() failed");
561         goto error;
562     }
563
564     n = OCSP_response_status(ocsp);
565
566     if (n != OCSP_RESPONSE_STATUS_SUCCESSFUL) {
567         ngx_log_error(NGX_LOG_ERR, ctx->log, 0,
568             "OCSP response not successful (%d: %s)",
569             n, OCSP_response_status_str(n));
570         goto error;
571     }
572
573     basic = OCSP_response_get1_basic(ocsp);
574     if (basic == NULL) {
575         ngx_ssl_error(NGX_LOG_ERR, ctx->log, 0,
576             "OCSP_response_get1_basic() failed");
577         goto error;
578     }
579
580     store = SSL_CTX_get_cert_store(staple->ssl_ctx);
581     if (store == NULL) {
582         ngx_ssl_error(NGX_LOG_CRIT, ctx->log, 0,
583             "SSL_CTX_get_cert_store() failed");
584         goto error;
585     }
586
587     #if OPENSSSL_VERSION_NUMBER >= 0x10001000L
588     SSL_CTX_get_extra_chain_certs(staple->ssl_ctx, &chain);
589 #else
590     chain = staple->ssl_ctx->extra_certs;
591 #endif
592
593     if (OCSP_basic_verify(basic, chain, store,
594         staple->verify ? OCSP_TRUSTOTHER : OCSP_NOVERIFY)
595         != 1)
596     {
597         ngx_ssl_error(NGX_LOG_ERR, ctx->log, 0,
598             "OCSP_basic_verify() failed");
599         goto error;
600     }

```

```

601 id = OCSP_cert_to_id(NULL, ctx->cert, ctx->issuer);
602 if (id == NULL) {
603     ngx_ssl_error(NGX_LOG_CRIT, ctx->log, 0,
604                 "OCSP_cert_to_id() failed");
605     goto error;
606 }
607
608 if (OCSP_resp_find_status(basic, id, &n, NULL, NULL,
609                          &thisupdate, &nextupdate)
610     != 1)
611 {
612     ngx_log_error(NGX_LOG_ERR, ctx->log, 0,
613                 "certificate status not found in the OCSP response");
614     goto error;
615 }
616
617 if (n != V_OCSP_CERTSTATUS_GOOD) {
618     ngx_log_error(NGX_LOG_ERR, ctx->log, 0,
619                 "certificate status \"%s\" in the OCSP response",
620                 OCSP_cert_status_str(n));
621     goto error;
622 }
623
624 if (OCSP_check_validity(thisupdate, nextupdate, 300, -1) != 1) {
625     ngx_ssl_error(NGX_LOG_ERR, ctx->log, 0,
626                 "OCSP_check_validity() failed");
627     goto error;
628 }
629
630 OCSP_CERTID_free(id);
631 OCSP_BASICRESP_free(basic);
632 OCSP_RESPONSE_free(ocsp);
633
634 /* copy the response to memory not in ctx->pool */
635
636 response.len = len;
637 response.data = ngx_alloc(response.len, ctx->log);
638
639 if (response.data == NULL) {
640     goto done;
641 }
642
643 ngx_memcpy(response.data, ctx->response->pos, response.len);
644
645 ngx_log_debug2(NGX_LOG_DEBUG_EVENT, ctx->log, 0,
646              "ssl ocp response, %s, %uz",
647              OCSP_cert_status_str(n), response.len);
648
649 if (staple->staple.data) {
650     ngx_free(staple->staple.data);
651 }
652
653 staple->staple = response;
654
655 done:
656
657 staple->loading = 0;
658 staple->valid = ngx_time() + 3600; /* ssl_stapling_valid */
659
660 ngx_ssl_ocsp_done(ctx);
661 return;
662
663 error:
664
665 staple->loading = 0;
666 staple->valid = ngx_time() + 300; /* ssl_stapling_err_valid */
667
668 if (id) {
669     OCSP_CERTID_free(id);
670 }
671
672 if (basic) {
673     OCSP_BASICRESP_free(basic);
674 }
675
676

```



```

677     if (ocsp) {
678         OCSP_RESPONSE_free(ocsp);
679     }
680
681     ngx\_ssl\_ocsp\_done(ctx);
682 }
683
684
685 static void
686 ngx\_ssl\_stapling\_cleanup(void *data)
687 {
688     ngx\_ssl\_stapling\_t *staple = data;
689
690     if (staple->issuer) {
691         X509_free(staple->issuer);
692     }
693
694     if (staple->staple.data) {
695         ngx\_free(staple->staple.data);
696     }
697 }
698
699
700 static ngx\_ssl\_ocsp\_ctx\_t *
701 ngx\_ssl\_ocsp\_start(void)
702 {
703     ngx\_log\_t *log;
704     ngx\_pool\_t *pool;
705     ngx\_ssl\_ocsp\_ctx\_t *ctx;
706
707     pool = ngx\_create\_pool(2048, ngx\_cycle->log);
708     if (pool == NULL) {
709         return NULL;
710     }
711
712     ctx = ngx\_pcalloc(pool, sizeof(ngx\_ssl\_ocsp\_ctx\_t));
713     if (ctx == NULL) {
714         ngx\_destroy\_pool(pool);
715         return NULL;
716     }
717
718     log = ngx\_palloc(pool, sizeof(ngx\_log\_t));
719     if (log == NULL) {
720         ngx\_destroy\_pool(pool);
721         return NULL;
722     }
723
724     ctx->pool = pool;
725
726     *log = *ctx->pool->log;
727
728     ctx->pool->log = log;
729     ctx->log = log;
730
731     log->handler = ngx\_ssl\_ocsp\_log\_error;
732     log->data = ctx;
733     log->action = "requesting certificate status";
734
735     return ctx;
736 }
737
738
739 static void
740 ngx\_ssl\_ocsp\_done(ngx\_ssl\_ocsp\_ctx\_t *ctx)
741 {
742     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_EVENT, ctx->log, 0,
743         "ssl ocsp done");
744
745     if (ctx->peer.connection) {
746         ngx\_close\_connection(ctx->peer.connection);
747     }
748
749     ngx\_destroy\_pool(ctx->pool);
750 }
751
752

```

```

753 static void
754 ngx_ssl_ocsp_error(ngx_ssl_ocsp_ctx_t *ctx)
755 {
756     ngx_log_debug0(NGX_LOG_DEBUG_EVENT, ctx->log, 0,
757                 "ssl ocsp error");
758
759     ctx->code = 0;
760     ctx->handler(ctx);
761 }
762
763
764 static void
765 ngx_ssl_ocsp_request(ngx_ssl_ocsp_ctx_t *ctx)
766 {
767     ngx_resolver_ctx_t *resolve, temp;
768
769     ngx_log_debug0(NGX_LOG_DEBUG_EVENT, ctx->log, 0,
770                 "ssl ocsp request");
771
772     if (ngx_ssl_ocsp_create_request(ctx) != NGX_OK) {
773         ngx_ssl_ocsp_error(ctx);
774         return;
775     }
776
777     if (ctx->resolver) {
778         /* resolve OCSP responder hostname */
779
780         temp.name = ctx->host;
781
782         resolve = ngx_resolve_start(ctx->resolver, &temp);
783         if (resolve == NULL) {
784             ngx_ssl_ocsp_error(ctx);
785             return;
786         }
787
788         if (resolve == NGX_NO_RESOLVER) {
789             ngx_log_error(NGX_LOG_WARN, ctx->log, 0,
790                 "no resolver defined to resolve %V", &ctx->host);
791             goto connect;
792         }
793
794         resolve->name = ctx->host;
795         resolve->handler = ngx_ssl_ocsp_resolve_handler;
796         resolve->data = ctx;
797         resolve->timeout = ctx->resolver_timeout;
798
799         if (ngx_resolve_name(resolve) != NGX_OK) {
800             ngx_ssl_ocsp_error(ctx);
801             return;
802         }
803
804         return;
805     }
806
807 connect:
808
809     ngx_ssl_ocsp_connect(ctx);
810 }
811
812
813 static void
814 ngx_ssl_ocsp_resolve_handler(ngx_resolver_ctx_t *resolve)
815 {
816     ngx_ssl_ocsp_ctx_t *ctx = resolve->data;
817
818     u_char *p;
819     size_t len;
820     in_port_t port;
821     socklen_t socklen;
822     ngx_uint_t i;
823     struct sockaddr *sockaddr;
824
825     ngx_log_debug0(NGX_LOG_DEBUG_EVENT, ctx->log, 0,
826                 "ssl ocsp resolve handler");
827
828     if (resolve->state) {

```

```

829     ngx_log_error(NGX_LOG_ERR, ctx->log, 0,
830                 "%V could not be resolved (%i: %s)",
831                 &resolve->name, resolve->state,
832                 ngx_resolver_strerror(resolve->state));
833     goto failed;
834 }
835
836 #if (NGX_DEBUG)
837 {
838     u_char    text[NGX_SOCKADDR_STRLEN];
839     ngx_str_t  addr;
840
841     addr.data = text;
842
843     for (i = 0; i < resolve->naddrs; i++) {
844         addr.len = ngx_sock_ntop(resolve->addrs[i].sockaddr,
845                                 resolve->addrs[i].socklen,
846                                 text, NGX_SOCKADDR_STRLEN, 0);
847
848         ngx_log_debug1(NGX_LOG_DEBUG_EVENT, ctx->log, 0,
849                       "name was resolved to %V", &addr);
850     }
851 }
852 #endif
853
854     ctx->naddrs = resolve->naddrs;
855     ctx->addrs = ngx_palloc(ctx->pool, ctx->naddrs * sizeof(ngx_addr_t));
856
857     if (ctx->addrs == NULL) {
858         goto failed;
859     }
860
861     port = htons(ctx->port);
862
863     for (i = 0; i < resolve->naddrs; i++) {
864         socklen = resolve->addrs[i].socklen;
865
866         sockaddr = ngx_palloc(ctx->pool, socklen);
867         if (sockaddr == NULL) {
868             goto failed;
869         }
870
871         ngx_memcpy(sockaddr, resolve->addrs[i].sockaddr, socklen);
872
873         switch (sockaddr->sa_family) {
874 #if (NGX_HAVE_INET6)
875             case AF_INET6:
876                 ((struct sockaddr_in6 *) sockaddr)->sin6_port = port;
877                 break;
878 #endif
879             default: /* AF_INET */
880                 ((struct sockaddr_in *) sockaddr)->sin_port = port;
881         }
882
883         ctx->addrs[i].sockaddr = sockaddr;
884         ctx->addrs[i].socklen = socklen;
885
886         p = ngx_palloc(ctx->pool, NGX_SOCKADDR_STRLEN);
887         if (p == NULL) {
888             goto failed;
889         }
890
891         len = ngx_sock_ntop(sockaddr, socklen, p, NGX_SOCKADDR_STRLEN, 1);
892
893         ctx->addrs[i].name.len = len;
894         ctx->addrs[i].name.data = p;
895     }
896
897     ngx_resolve_name_done(resolve);
898
899     ngx_ssl_ocsp_connect(ctx);
900     return;
901
902 failed:

```

```

905     ngx\_resolve\_name\_done(resolve);
906     ngx\_ssl\_ocsp\_error(ctx);
907 }
908
909
910
911 static void
912 ngx\_ssl\_ocsp\_connect(ngx\_ssl\_ocsp\_ctx\_t *ctx)
913 {
914     ngx\_int\_t    rc;
915
916     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_EVENT, ctx->log, 0,
917                 "ssl ocsp connect");
918
919     /* TODO: use all ip addresses */
920
921     ctx->peer.sockaddr = ctx->addrs[0].sockaddr;
922     ctx->peer.socklen = ctx->addrs[0].socklen;
923     ctx->peer.name = &ctx->addrs[0].name;
924     ctx->peer.get = ngx\_event\_get\_peer;
925     ctx->peer.log = ctx->log;
926     ctx->peer.log_error = NGX\_ERROR\_ERR;
927
928     rc = ngx\_event\_connect\_peer(&ctx->peer);
929
930     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_EVENT, ctx->log, 0,
931                 "ssl ocsp connect peer done");
932
933     if (rc == NGX\_ERROR || rc == NGX\_BUSY || rc == NGX\_DECLINED) {
934         ngx\_ssl\_ocsp\_error(ctx);
935         return;
936     }
937
938     ctx->peer.connection->data = ctx;
939     ctx->peer.connection->pool = ctx->pool;
940
941     ctx->peer.connection->read->handler = ngx\_ssl\_ocsp\_read\_handler;
942     ctx->peer.connection->write->handler = ngx\_ssl\_ocsp\_write\_handler;
943
944     ctx->process = ngx\_ssl\_ocsp\_process\_status\_line;
945
946     ngx\_add\_timer(ctx->peer.connection->read, ctx->timeout);
947     ngx\_add\_timer(ctx->peer.connection->write, ctx->timeout);
948
949     if (rc == NGX\_OK) {
950         ngx\_ssl\_ocsp\_write\_handler(ctx->peer.connection->write);
951         return;
952     }
953 }
954
955
956 static void
957 ngx\_ssl\_ocsp\_write\_handler(ngx\_event\_t *wev)
958 {
959     ssize\_t      n, size;
960     ngx\_connection\_t *c;
961     ngx\_ssl\_ocsp\_ctx\_t *ctx;
962
963     c = wev->data;
964     ctx = c->data;
965
966     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_EVENT, wev->log, 0,
967                 "ssl ocsp write handler");
968
969     if (wev->timedout) {
970         ngx\_log\_error(NGX\_LOG\_ERR, wev->log, NGX\_ETIMEDOUT,
971                 "OCSP responder timed out");
972         ngx\_ssl\_ocsp\_error(ctx);
973         return;
974     }
975
976     size = ctx->request->last - ctx->request->pos;
977
978     n = ngx\_send(c, ctx->request->pos, size);
979
980     if (n == NGX\_ERROR) {

```

```

981     ngx_ssl_ocsp_error(ctx);
982     return;
983 }
984
985 if (n > 0) {
986     ctx->request->pos += n;
987
988     if (n == size) {
989         wev->handler = ngx_ssl_ocsp_dummy_handler;
990
991         if (wev->timer_set) {
992             ngx_del_timer(wev);
993         }
994
995         if (ngx_handle_write_event(wev, 0) != NGX_OK) {
996             ngx_ssl_ocsp_error(ctx);
997         }
998
999         return;
1000     }
1001 }
1002
1003 if (!wev->timer_set) {
1004     ngx_add_timer(wev, ctx->timeout);
1005 }
1006 }
1007
1008
1009 static void
1010 ngx_ssl_ocsp_read_handler(ngx_event_t *rev)
1011 {
1012     ssize_t          n, size;
1013     ngx_int_t        rc;
1014     ngx_ssl_ocsp_ctx_t *ctx;
1015     ngx_connection_t *c;
1016
1017     c = rev->data;
1018     ctx = c->data;
1019
1020     ngx_log_debug0(NGX_LOG_DEBUG_EVENT, rev->log, 0,
1021                  "ssl ocsp read handler");
1022
1023     if (rev->timedout) {
1024         ngx_log_error(NGX_LOG_ERR, rev->log, NGX_ETIMEDOUT,
1025                     "OCSP responder timed out");
1026         ngx_ssl_ocsp_error(ctx);
1027         return;
1028     }
1029
1030     if (ctx->response == NULL) {
1031         ctx->response = ngx_create_temp_buf(ctx->pool, 16384);
1032         if (ctx->response == NULL) {
1033             ngx_ssl_ocsp_error(ctx);
1034             return;
1035         }
1036     }
1037
1038     for ( ;; ) {
1039
1040         size = ctx->response->end - ctx->response->last;
1041
1042         n = ngx_recv(c, ctx->response->last, size);
1043
1044         if (n > 0) {
1045             ctx->response->last += n;
1046
1047             rc = ctx->process(ctx);
1048
1049             if (rc == NGX_ERROR) {
1050                 ngx_ssl_ocsp_error(ctx);
1051                 return;
1052             }
1053
1054             continue;
1055         }
1056

```

```

1057     if (n == NGX\_AGAIN) {
1058
1059         if (ngx\_handle\_read\_event(rev, 0) != NGX\_OK) {
1060             ngx\_ssl\_ocsp\_error(ctx);
1061         }
1062
1063         return;
1064     }
1065
1066     break;
1067 }
1068
1069 ctx->done = 1;
1070
1071 rc = ctx->process(ctx);
1072
1073 if (rc == NGX\_DONE) {
1074     /* ctx->handler() was called */
1075     return;
1076 }
1077
1078 ngx\_log\_error(NGX\_LOG\_ERR, ctx->log, 0,
1079     "OCSP responder prematurely closed connection");
1080
1081 ngx\_ssl\_ocsp\_error(ctx);
1082 }
1083
1084
1085 static void
1086 ngx\_ssl\_ocsp\_dummy\_handler(ngx\_event\_t *ev)
1087 {
1088     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_EVENT, ev->log, 0,
1089         "ssl ocsp dummy handler");
1090 }
1091
1092
1093 static ngx\_int\_t
1094 ngx\_ssl\_ocsp\_create\_request(ngx\_ssl\_ocsp\_ctx\_t *ctx)
1095 {
1096     int len;
1097     u_char *p;
1098     uintptr_t escape;
1099     ngx\_str\_t binary, base64;
1100     ngx\_buf\_t *b;
1101     OCSP_CERTID *id;
1102     OCSP_REQUEST *ocsp;
1103
1104     ocsp = OCSP_REQUEST_new();
1105     if (ocsp == NULL) {
1106         ngx\_ssl\_error(NGX\_LOG\_CRIT, ctx->log, 0,
1107             "OCSP_REQUEST_new() failed");
1108         return NGX\_ERROR;
1109     }
1110
1111     id = OCSP_cert_to_id(NULL, ctx->cert, ctx->issuer);
1112     if (id == NULL) {
1113         ngx\_ssl\_error(NGX\_LOG\_CRIT, ctx->log, 0,
1114             "OCSP_cert_to_id() failed");
1115         goto failed;
1116     }
1117
1118     if (OCSP_request_add0_id(ocsp, id) == NULL) {
1119         ngx\_ssl\_error(NGX\_LOG\_CRIT, ctx->log, 0,
1120             "OCSP_request_add0_id() failed");
1121         goto failed;
1122     }
1123
1124     len = i2d_OCSP_REQUEST(ocsp, NULL);
1125     if (len <= 0) {
1126         ngx\_ssl\_error(NGX\_LOG\_CRIT, ctx->log, 0,
1127             "i2d_OCSP_REQUEST() failed");
1128         goto failed;
1129     }
1130
1131     binary.len = len;
1132     binary.data = ngx\_palloc(ctx->pool, len);

```

```

1133     if (binary.data == NULL) {
1134         goto failed;
1135     }
1136
1137     p = binary.data;
1138     len = i2d_OCSP_REQUEST(ocsp, &p);
1139     if (len <= 0) {
1140         ngx_ssl_error(NGX_LOG_EMERG, ctx->log, 0,
1141             "i2d_OCSP_REQUEST() failed");
1142         goto failed;
1143     }
1144
1145     base64.len = ngx_base64_encoded_length(binary.len);
1146     base64.data = ngx_palloc(ctx->pool, base64.len);
1147     if (base64.data == NULL) {
1148         goto failed;
1149     }
1150
1151     ngx_encode_base64(&base64, &binary);
1152
1153     escape = ngx_escape_uri(NULL, base64.data, base64.len,
1154         NGX_ESCAPE_URI_COMPONENT);
1155
1156     ngx_log_debug2(NGX_LOG_DEBUG_EVENT, ctx->log, 0,
1157         "ssl ocsf request length %z, escape %d",
1158         base64.len, escape);
1159
1160     len = sizeof("GET ") - 1 + ctx->uri.len + sizeof("/") - 1
1161         + base64.len + 2 * escape + sizeof(" HTTP/1.0" CRLF) - 1
1162         + sizeof("Host: ") - 1 + ctx->host.len + sizeof(CRLF) - 1
1163         + sizeof(CRLF) - 1;
1164
1165     b = ngx_create_temp_buf(ctx->pool, len);
1166     if (b == NULL) {
1167         goto failed;
1168     }
1169
1170     p = b->last;
1171
1172     p = ngx_cpymem(p, "GET ", sizeof("GET ") - 1);
1173     p = ngx_cpymem(p, ctx->uri.data, ctx->uri.len);
1174
1175     if (ctx->uri.data[ctx->uri.len - 1] != '/') {
1176         *p++ = '/';
1177     }
1178
1179     if (escape == 0) {
1180         p = ngx_cpymem(p, base64.data, base64.len);
1181     }
1182     else {
1183         p = (u_char *) ngx_escape_uri(p, base64.data, base64.len,
1184             NGX_ESCAPE_URI_COMPONENT);
1185     }
1186
1187     p = ngx_cpymem(p, " HTTP/1.0" CRLF, sizeof(" HTTP/1.0" CRLF) - 1);
1188     p = ngx_cpymem(p, "Host: ", sizeof("Host: ") - 1);
1189     p = ngx_cpymem(p, ctx->host.data, ctx->host.len);
1190     *p++ = CR; *p++ = LF;
1191
1192     /* add "\r\n" at the header end */
1193     *p++ = CR; *p++ = LF;
1194
1195     b->last = p;
1196     ctx->request = b;
1197
1198     OCSP_REQUEST_free(ocsp);
1199
1200     return NGX_OK;
1201
1202 failed:
1203
1204     OCSP_REQUEST_free(ocsp);
1205
1206     return NGX_ERROR;
1207 }
1208

```

```

1209 static ngx_int_t
1210 ngx_ssl_ocsp_process_status_line(ngx_ssl_ocsp_ctx_t *ctx)
1211 {
1212     ngx_int_t rc;
1213
1214     rc = ngx_ssl_ocsp_parse_status_line(ctx);
1215
1216     if (rc == NGX_OK) {
1217 #if 0
1218         ngx_log_debug2(NGX_LOG_DEBUG_EVENT, ctx->log, 0,
1219             "ssl ocsp status line \"%s\"",
1220             ctx->response->pos - ctx->response->start,
1221             ctx->response->start);
1222 #endif
1223
1224         ctx->process = ngx_ssl_ocsp_process_headers;
1225         return ctx->process(ctx);
1226     }
1227
1228     if (rc == NGX_AGAIN) {
1229         return NGX_AGAIN;
1230     }
1231
1232     /* rc == NGX_ERROR */
1233
1234     ngx_log_error(NGX_LOG_ERR, ctx->log, 0,
1235         "OCSP responder sent invalid response");
1236
1237     return NGX_ERROR;
1238 }
1239
1240
1241
1242 static ngx_int_t
1243 ngx_ssl_ocsp_parse_status_line(ngx_ssl_ocsp_ctx_t *ctx)
1244 {
1245     u_char ch;
1246     u_char *p;
1247     ngx_buf_t *b;
1248     enum {
1249         sw_start = 0,
1250         sw_H,
1251         sw_HT,
1252         sw_HTTP,
1253         sw_HTTP,
1254         sw_first_major_digit,
1255         sw_major_digit,
1256         sw_first_minor_digit,
1257         sw_minor_digit,
1258         sw_status,
1259         sw_space_after_status,
1260         sw_status_text,
1261         sw_almost_done
1262     } state;
1263
1264     ngx_log_debug0(NGX_LOG_DEBUG_EVENT, ctx->log, 0,
1265         "ssl ocsp process status line");
1266
1267     state = ctx->state;
1268     b = ctx->response;
1269
1270     for (p = b->pos; p < b->last; p++) {
1271         ch = *p;
1272
1273         switch (state) {
1274
1275             /* "HTTP/" */
1276             case sw_start:
1277                 switch (ch) {
1278                     case 'H':
1279                         state = sw_H;
1280                         break;
1281                     default:
1282                         return NGX_ERROR;
1283                 }
1284                 break;

```



```

1285
1286 case sw_H:
1287     switch (ch) {
1288         case 'T':
1289             state = sw_HT;
1290             break;
1291         default:
1292             return NGX\_ERROR;
1293     }
1294     break;
1295
1296 case sw_HT:
1297     switch (ch) {
1298         case 'T':
1299             state = sw_HTTP;
1300             break;
1301         default:
1302             return NGX\_ERROR;
1303     }
1304     break;
1305
1306 case sw_HTTP:
1307     switch (ch) {
1308         case 'P':
1309             state = sw_HTTPS;
1310             break;
1311         default:
1312             return NGX\_ERROR;
1313     }
1314     break;
1315
1316 case sw_HTTPS:
1317     switch (ch) {
1318         case '/':
1319             state = sw_first_major_digit;
1320             break;
1321         default:
1322             return NGX\_ERROR;
1323     }
1324     break;
1325
1326 /* the first digit of major HTTP version */
1327 case sw_first_major_digit:
1328     if (ch < '1' || ch > '9') {
1329         return NGX\_ERROR;
1330     }
1331
1332     state = sw_major_digit;
1333     break;
1334
1335 /* the major HTTP version or dot */
1336 case sw_major_digit:
1337     if (ch == '.') {
1338         state = sw_first_minor_digit;
1339         break;
1340     }
1341
1342     if (ch < '0' || ch > '9') {
1343         return NGX\_ERROR;
1344     }
1345
1346     break;
1347
1348 /* the first digit of minor HTTP version */
1349 case sw_first_minor_digit:
1350     if (ch < '0' || ch > '9') {
1351         return NGX\_ERROR;
1352     }
1353
1354     state = sw_minor_digit;
1355     break;
1356
1357 /* the minor HTTP version or the end of the request line */
1358 case sw_minor_digit:
1359     if (ch == ' ') {
1360         state = sw_status;

```

```

1361         break;
1362     }
1363
1364     if (ch < '0' || ch > '9') {
1365         return NGX\_ERROR;
1366     }
1367
1368     break;
1369
1370     /* HTTP status code */
1371     case sw_status:
1372         if (ch == ' ') {
1373             break;
1374         }
1375
1376         if (ch < '0' || ch > '9') {
1377             return NGX\_ERROR;
1378         }
1379
1380         ctx->code = ctx->code * 10 + ch - '0';
1381
1382         if (++ctx->count == 3) {
1383             state = sw_space_after_status;
1384         }
1385
1386         break;
1387
1388     /* space or end of line */
1389     case sw_space_after_status:
1390         switch (ch) {
1391             case ' ':
1392                 state = sw_status_text;
1393                 break;
1394             case '.': /* IIS may send 403.1, 403.2, etc */
1395                 state = sw_status_text;
1396                 break;
1397             case CR:
1398                 state = sw_almost_done;
1399                 break;
1400             case LF:
1401                 goto done;
1402             default:
1403                 return NGX\_ERROR;
1404         }
1405         break;
1406
1407     /* any text until end of line */
1408     case sw_status_text:
1409         switch (ch) {
1410             case CR:
1411                 state = sw_almost_done;
1412                 break;
1413             case LF:
1414                 goto done;
1415         }
1416         break;
1417
1418     /* end of status line */
1419     case sw_almost_done:
1420         switch (ch) {
1421             case LF:
1422                 goto done;
1423             default:
1424                 return NGX\_ERROR;
1425         }
1426     }
1427 }
1428
1429 b->pos = p;
1430 ctx->state = state;
1431
1432 return NGX\_AGAIN;
1433
1434 done:
1435
1436 b->pos = p + 1;

```

```

1437     ctx->state = sw_start;
1438
1439     return NGX_OK;
1440 }
1441
1442
1443 static ngx_int_t
1444 ngx_ssl_ocsp_process_headers(ngx_ssl_ocsp_ctx_t *ctx)
1445 {
1446     size_t    len;
1447     ngx_int_t rc;
1448
1449     ngx_log_debug0(NGX_LOG_DEBUG_EVENT, ctx->log, 0,
1450                  "ssl ocsp process headers");
1451
1452     for ( ;; ) {
1453         rc = ngx_ssl_ocsp_parse_header_line(ctx);
1454
1455         if (rc == NGX_OK) {
1456
1457             ngx_log_debug4(NGX_LOG_DEBUG_EVENT, ctx->log, 0,
1458                           "ssl ocsp header \"%s: %s\"",
1459                           ctx->header_name_end - ctx->header_name_start,
1460                           ctx->header_name_start,
1461                           ctx->header_end - ctx->header_start,
1462                           ctx->header_start);
1463
1464             len = ctx->header_name_end - ctx->header_name_start;
1465
1466             if (len == sizeof("Content-Type") - 1
1467                 && ngx_strncasecmp(ctx->header_name_start,
1468                                     (u_char *) "Content-Type",
1469                                     sizeof("Content-Type") - 1)
1470                 == 0)
1471             {
1472                 len = ctx->header_end - ctx->header_start;
1473
1474                 if (len != sizeof("application/ocsp-response") - 1
1475                     || ngx_strncasecmp(ctx->header_start,
1476                                         (u_char *) "application/ocsp-response",
1477                                         sizeof("application/ocsp-response") - 1)
1478                     != 0)
1479                 {
1480                     ngx_log_error(NGX_LOG_ERR, ctx->log, 0,
1481                                   "OCSP responder sent invalid "
1482                                   "\"Content-Type\" header: \"%s\"",
1483                                   ctx->header_end - ctx->header_start,
1484                                   ctx->header_start);
1485                     return NGX_ERROR;
1486                 }
1487
1488                 continue;
1489             }
1490
1491             /* TODO: honor Content-Length */
1492
1493             continue;
1494         }
1495
1496         if (rc == NGX_DONE) {
1497             break;
1498         }
1499
1500         if (rc == NGX_AGAIN) {
1501             return NGX_AGAIN;
1502         }
1503
1504         /* rc == NGX_ERROR */
1505
1506         ngx_log_error(NGX_LOG_ERR, ctx->log, 0,
1507                       "OCSP responder sent invalid response");
1508
1509         return NGX_ERROR;
1510     }
1511
1512     ctx->process = ngx_ssl_ocsp_process_body;

```

```

1513     return ctx->process(ctx);
1514 }
1515
1516 static ngx_int_t
1517 ngx_ssl_ocsp_parse_header_line(ngx_ssl_ocsp_ctx_t *ctx)
1518 {
1519     u_char      c, ch, *p;
1520     enum {
1521         sw_start = 0,
1522         sw_name,
1523         sw_space_before_value,
1524         sw_value,
1525         sw_space_after_value,
1526         sw_almost_done,
1527         sw_header_almost_done
1528     } state;
1529
1530     state = ctx->state;
1531
1532     for (p = ctx->response->pos; p < ctx->response->last; p++) {
1533         ch = *p;
1534
1535         #if 0
1536         ngx_log_debug3(NGX_LOG_DEBUG_EVENT, ctx->log, 0,
1537             "s:%d in: '%02Xd:%c'", state, ch, ch);
1538         #endif
1539
1540         switch (state) {
1541
1542             /* first char */
1543             case sw_start:
1544
1545                 switch (ch) {
1546                     case CR:
1547                         ctx->header_end = p;
1548                         state = sw_header_almost_done;
1549                         break;
1550                     case LF:
1551                         ctx->header_end = p;
1552                         goto header_done;
1553                     default:
1554                         state = sw_name;
1555                         ctx->header_name_start = p;
1556
1557                         c = (u_char) (ch | 0x20);
1558                         if (c >= 'a' && c <= 'z') {
1559                             break;
1560                         }
1561
1562                         if (ch >= '0' && ch <= '9') {
1563                             break;
1564                         }
1565
1566                         return NGX_ERROR;
1567                 }
1568                 break;
1569
1570             /* header name */
1571             case sw_name:
1572                 c = (u_char) (ch | 0x20);
1573                 if (c >= 'a' && c <= 'z') {
1574                     break;
1575                 }
1576
1577                 if (ch == ':') {
1578                     ctx->header_name_end = p;
1579                     state = sw_space_before_value;
1580                     break;
1581                 }
1582
1583                 if (ch == '-') {
1584                     break;
1585                 }
1586
1587                 if (ch >= '0' && ch <= '9') {
1588                     break;

```

```

1589     }
1590
1591     if (ch == CR) {
1592         ctx->header_name_end = p;
1593         ctx->header_start = p;
1594         ctx->header_end = p;
1595         state = sw_almost_done;
1596         break;
1597     }
1598
1599     if (ch == LF) {
1600         ctx->header_name_end = p;
1601         ctx->header_start = p;
1602         ctx->header_end = p;
1603         goto done;
1604     }
1605
1606     return NGX_ERROR;
1607
1608     /* space* before header value */
1609     case sw_space_before_value:
1610         switch (ch) {
1611             case ' ':
1612                 break;
1613             case CR:
1614                 ctx->header_start = p;
1615                 ctx->header_end = p;
1616                 state = sw_almost_done;
1617                 break;
1618             case LF:
1619                 ctx->header_start = p;
1620                 ctx->header_end = p;
1621                 goto done;
1622             default:
1623                 ctx->header_start = p;
1624                 state = sw_value;
1625                 break;
1626         }
1627         break;
1628
1629     /* header value */
1630     case sw_value:
1631         switch (ch) {
1632             case ' ':
1633                 ctx->header_end = p;
1634                 state = sw_space_after_value;
1635                 break;
1636             case CR:
1637                 ctx->header_end = p;
1638                 state = sw_almost_done;
1639                 break;
1640             case LF:
1641                 ctx->header_end = p;
1642                 goto done;
1643         }
1644         break;
1645
1646     /* space* before end of header line */
1647     case sw_space_after_value:
1648         switch (ch) {
1649             case ' ':
1650                 break;
1651             case CR:
1652                 state = sw_almost_done;
1653                 break;
1654             case LF:
1655                 goto done;
1656             default:
1657                 state = sw_value;
1658                 break;
1659         }
1660         break;
1661
1662     /* end of header line */
1663     case sw_almost_done:
1664         switch (ch) {

```

```

1665         case LF:
1666             goto done;
1667         default:
1668             return NGX_ERROR;
1669     }
1670
1671     /* end of header */
1672     case sw_header_almost_done:
1673         switch (ch) {
1674             case LF:
1675                 goto header_done;
1676             default:
1677                 return NGX_ERROR;
1678         }
1679     }
1680 }
1681
1682 ctx->response->pos = p;
1683 ctx->state = state;
1684
1685 return NGX_AGAIN;
1686
1687 done:
1688
1689 ctx->response->pos = p + 1;
1690 ctx->state = sw_start;
1691
1692 return NGX_OK;
1693
1694 header_done:
1695
1696 ctx->response->pos = p + 1;
1697 ctx->state = sw_start;
1698
1699 return NGX_DONE;
1700 }
1701
1702
1703 static ngx_int_t
1704 ngx_ssl_ocsp_process_body(ngx_ssl_ocsp_ctx_t *ctx)
1705 {
1706     ngx_log_debug0(NGX_LOG_DEBUG_EVENT, ctx->log, 0,
1707                  "ssl ocsp process body");
1708
1709     if (ctx->done) {
1710         ctx->handler(ctx);
1711         return NGX_DONE;
1712     }
1713
1714     return NGX_AGAIN;
1715 }
1716
1717
1718 static u_char *
1719 ngx_ssl_ocsp_log_error(ngx_log_t *log, u_char *buf, size_t len)
1720 {
1721     u_char          *p;
1722     ngx_ssl_ocsp_ctx_t *ctx;
1723
1724     p = buf;
1725
1726     if (log->action) {
1727         p = ngx_snprintf(buf, len, " while %s", log->action);
1728         len -= p - buf;
1729     }
1730
1731     ctx = log->data;
1732
1733     if (ctx) {
1734         p = ngx_snprintf(p, len, ", responder: %V", &ctx->host);
1735     }
1736
1737     return p;
1738 }
1739
1740

```

```
1741 #else
1742
1743
1744 ngx_int_t
1745 ngx_ssl_stapling(ngx_conf_t *cf, ngx_ssl_t *ssl, ngx_str_t *file,
1746 ngx_str_t *responder, ngx_uint_t verify)
1747 {
1748     ngx_log_error(NGX_LOG_WARN, ssl->log, 0,
1749         "\"ssl_stapling\" ignored, not supported");
1750
1751     return NGX_OK;
1752 }
1753
1754 ngx_int_t
1755 ngx_ssl_stapling_resolver(ngx_conf_t *cf, ngx_ssl_t *ssl,
1756 ngx_resolver_t *resolver, ngx_msec_t resolver_timeout)
1757 {
1758     return NGX_OK;
1759 }
1760
1761
1762 #endif
```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_resolver.h - nginx-1.7.10

### Data types defined

- [ngx\\_resolver\\_ctx\\_s](#)
- [ngx\\_resolver\\_ctx\\_t](#)
- [ngx\\_resolver\\_handler\\_pt](#)
- [ngx\\_resolver\\_node\\_t](#)
- [ngx\\_resolver\\_t](#)
- [ngx\\_udp\\_connection\\_t](#)

### Macros defined

- [NGX\\_NO\\_RESOLVER](#)
- [NGX\\_RESOLVER\\_MAX\\_RECURSION](#)
- [NGX\\_RESOLVE\\_A](#)
- [NGX\\_RESOLVE\\_AAAA](#)
- [NGX\\_RESOLVE\\_CNAME](#)
- [NGX\\_RESOLVE\\_DNAME](#)
- [NGX\\_RESOLVE\\_FORMERR](#)
- [NGX\\_RESOLVE\\_MX](#)
- [NGX\\_RESOLVE\\_NOTIMP](#)
- [NGX\\_RESOLVE\\_NXDOMAIN](#)
- [NGX\\_RESOLVE\\_PTR](#)
- [NGX\\_RESOLVE\\_REFUSED](#)
- [NGX\\_RESOLVE\\_SERVFAIL](#)
- [NGX\\_RESOLVE\\_TIMEDOUT](#)
- [NGX\\_RESOLVE\\_TXT](#)
- [\\_NGX\\_RESOLVER\\_H\\_INCLUDED](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10
11
```



```

12 #ifndef NGX_RESOLVER_H_INCLUDED
13 #define _NGX_RESOLVER_H_INCLUDED_
14
15
16 #define NGX_RESOLVE_A 1
17 #define NGX_RESOLVE_CNAME 5
18 #define NGX_RESOLVE_PTR 12
19 #define NGX_RESOLVE_MX 15
20 #define NGX_RESOLVE_TXT 16
21 #if (NGX_HAVE_INET6)
22 #define NGX_RESOLVE_AAAA 28
23 #endif
24 #define NGX_RESOLVE_DNAME 39
25
26 #define NGX_RESOLVE_FORMERR 1
27 #define NGX_RESOLVE_SERVFAIL 2
28 #define NGX_RESOLVE_NXDOMAIN 3
29 #define NGX_RESOLVE_NOTIMP 4
30 #define NGX_RESOLVE_REFUSED 5
31 #define NGX_RESOLVE_TIMEDOUT NGX_ETIMEDOUT
32
33
34 #define NGX_NO_RESOLVER (void *) -1
35
36 #define NGX_RESOLVER_MAX_RECURSION 50
37
38
39 typedef struct {
40     ngx_connection_t *connection;
41     struct sockaddr *sockaddr;
42     socklen_t socklen;
43     ngx_str_t server;
44     ngx_log_t log;
45 } ngx_udp_connection_t;
46
47
48 typedef struct ngx_resolver_ctx_s ngx_resolver_ctx_t;
49
50 typedef void (*ngx_resolver_handler_pt)(ngx_resolver_ctx_t *ctx);
51
52
53 typedef struct {
54     /* PTR: resolved name, A: name to resolve */
55     u_char *name;
56
57     ngx_queue_t queue;
58
59     /* event ident must be after 3 pointers as in ngx_connection_t */
60     ngx_int_t ident;
61
62     ngx_rbtree_node_t node;
63
64 #if (NGX_HAVE_INET6)
65     /* PTR: IPv6 address to resolve (IPv4 address is in rbtree node key) */
66     struct in6_addr addr6;
67 #endif
68
69     u_short nlen;
70     u_short qlen;
71
72     u_char *query;
73 #if (NGX_HAVE_INET6)
74     u_char *query6;
75 #endif
76
77     union {
78         in_addr_t addr;
79         in_addr_t *addrs;
80         u_char *cname;
81     } u;
82
83     u_char code;
84     u_short naddrs;
85     u_short cnlen;
86
87 #if (NGX_HAVE_INET6)

```

```

88     union {
89         struct in6_addr    addr6;
90         struct in6_addr    *addrs6;
91     } u6;
92
93     u_short                naddrs6;
94 #endif
95
96     time_t                 expire;
97     time_t                 valid;
98     uint32_t               ttl;
99
100     ngx_resolver_ctx_t    *waiting;
101 } ngx_resolver_node_t;
102
103
104 typedef struct {
105     /* has to be pointer because of "incomplete type" */
106     ngx_event_t          *event;
107     void                  *dummy;
108     ngx_log_t           *log;
109
110     /* event ident must be after 3 pointers as in ngx_connection_t */
111     ngx_int_t           ident;
112
113     /* simple round robin DNS peers balancer */
114     ngx_array_t         udp_connections;
115     ngx_uint_t         last_connection;
116
117     ngx_rbtree_t        name_rbtree;
118     ngx_rbtree_node_t  name_sentinel;
119
120     ngx_rbtree_t        addr_rbtree;
121     ngx_rbtree_node_t  addr_sentinel;
122
123     ngx_queue_t         name_resend_queue;
124     ngx_queue_t         addr_resend_queue;
125
126     ngx_queue_t         name_expire_queue;
127     ngx_queue_t         addr_expire_queue;
128
129 #if (NGX_HAVE_INET6)
130     ngx_uint_t          ipv6; /* unsigned ipv6:1; */
131     ngx_rbtree_t        addr6_rbtree;
132     ngx_rbtree_node_t  addr6_sentinel;
133     ngx_queue_t         addr6_resend_queue;
134     ngx_queue_t         addr6_expire_queue;
135 #endif
136
137     time_t                 resend_timeout;
138     time_t                 expire;
139     time_t                 valid;
140
141     ngx_uint_t          log_level;
142 } ngx_resolver_t;
143
144
145 struct ngx_resolver_ctx_s {
146     ngx_resolver_ctx_t    *next;
147     ngx_resolver_t        *resolver;
148     ngx_udp_connection_t *udp_connection;
149
150     ngx_int_t           state;
151     ngx_str_t          name;
152
153     ngx_uint_t         naddrs;
154     ngx_addr_t         *addrs;
155     ngx_addr_t         addr;
156     struct sockaddr_in     sin;
157
158     ngx_resolver_handler_pt handler;
159     void                  *data;
160     ngx_msec_t         timeout;
161
162     ngx_uint_t         quick; /* unsigned quick:1; */
163     ngx_uint_t         recursion;

```

```
164 ngx\_event\_t *event;
165 };
166
167
168 ngx\_resolver\_t *ngx\_resolver\_create(ngx\_conf\_t *cf, ngx\_str\_t *names,
169 ngx\_uint\_t n);
170 ngx\_resolver\_ctx\_t *ngx\_resolve\_start(ngx\_resolver\_t *r,
171 ngx\_resolver\_ctx\_t *temp);
172 ngx\_int\_t ngx\_resolve\_name(ngx\_resolver\_ctx\_t *ctx);
173 void ngx\_resolve\_name\_done(ngx\_resolver\_ctx\_t *ctx);
174 ngx\_int\_t ngx\_resolve\_addr(ngx\_resolver\_ctx\_t *ctx);
175 void ngx\_resolve\_addr\_done(ngx\_resolver\_ctx\_t *ctx);
176 char *ngx\_resolver\_strerror(ngx\_int\_t err);
177
178
179 #endif /* \_NGX\_RESOLVER\_H\_INCLUDED */
```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_resolver.c - nginx-1.7.10

### Data types defined

- [ngx\\_resolver\\_an\\_t](#)
- [ngx\\_resolver\\_hdr\\_t](#)
- [ngx\\_resolver\\_qs\\_t](#)

### Functions defined

- [ngx\\_resolve\\_addr](#)
- [ngx\\_resolve\\_addr\\_done](#)
- [ngx\\_resolve\\_name](#)
- [ngx\\_resolve\\_name\\_done](#)
- [ngx\\_resolve\\_name\\_locked](#)
- [ngx\\_resolve\\_start](#)
- [ngx\\_resolver\\_alloc](#)
- [ngx\\_resolver\\_calloc](#)
- [ngx\\_resolver\\_cleanup](#)
- [ngx\\_resolver\\_cleanup\\_tree](#)
- [ngx\\_resolver\\_copy](#)
- [ngx\\_resolver\\_create](#)
- [ngx\\_resolver\\_create\\_addr\\_query](#)
- [ngx\\_resolver\\_create\\_name\\_query](#)
- [ngx\\_resolver\\_dup](#)
- [ngx\\_resolver\\_expire](#)
- [ngx\\_resolver\\_export](#)
- [ngx\\_resolver\\_free](#)
- [ngx\\_resolver\\_free\\_locked](#)
- [ngx\\_resolver\\_free\\_node](#)
- [ngx\\_resolver\\_log\\_error](#)
- [ngx\\_resolver\\_lookup\\_addr](#)
- [ngx\\_resolver\\_lookup\\_addr6](#)
- [ngx\\_resolver\\_lookup\\_name](#)
- [ngx\\_resolver\\_process\\_a](#)

- [ngx\\_resolver\\_process\\_ptr](#)
- [ngx\\_resolver\\_process\\_response](#)
- [ngx\\_resolver\\_rbtrees\\_insert\\_addr6\\_value](#)
- [ngx\\_resolver\\_rbtrees\\_insert\\_value](#)
- [ngx\\_resolver\\_read\\_response](#)
- [ngx\\_resolver\\_resend](#)
- [ngx\\_resolver\\_resend\\_handler](#)
- [ngx\\_resolver\\_send\\_query](#)
- [ngx\\_resolver\\_strerror](#)
- [ngx\\_resolver\\_timeout\\_handler](#)
- [ngx\\_udp\\_connect](#)

## Macros defined

- [NGX\\_RESOLVER\\_UDP\\_SIZE](#)
- [ngx\\_resolver\\_node](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 #define NGX_RESOLVER_UDP_SIZE    4096
14
15
16 typedef struct {
17     u_char  ident_hi;
18     u_char  ident_lo;
19     u_char  flags_hi;
20     u_char  flags_lo;
21     u_char  nqs_hi;
22     u_char  nqs_lo;
23     u_char  nan_hi;
24     u_char  nan_lo;
25     u_char  nns_hi;
26     u_char  nns_lo;
27     u_char  nar_hi;
28     u_char  nar_lo;
29 } ngx_resolver_hdr_t;
30
31
32 typedef struct {
33     u_char  type_hi;
34     u_char  type_lo;
35     u_char  class_hi;
36     u_char  class_lo;
37 } ngx_resolver_qs_t;
38
39

```

```

40 typedef struct {
41     u_char type_hi;
42     u_char type_lo;
43     u_char class_hi;
44     u_char class_lo;
45     u_char ttl[4];
46     u_char len_hi;
47     u_char len_lo;
48 } ngx_resolver_an_t;
49
50
51 #define ngx_resolver_node(n)           \
52     (ngx_resolver_node_t *)         \
53     ((u_char *) (n) - offsetof(ngx_resolver_node_t, node))
54
55
56 ngx_int_t ngx_udp_connect(ngx_udp_connection_t *uc);
57
58
59 static void ngx_resolver_cleanup(void *data);
60 static void ngx_resolver_cleanup_tree(ngx_resolver_t *r, ngx_rbtree_t *tree);
61 static ngx_int_t ngx_resolve_name_locked(ngx_resolver_t *r,
62     ngx_resolver_ctx_t *ctx);
63 static void ngx_resolver_expire(ngx_resolver_t *r, ngx_rbtree_t *tree,
64     ngx_queue_t *queue);
65 static ngx_int_t ngx_resolver_send_query(ngx_resolver_t *r,
66     ngx_resolver_node_t *rn);
67 static ngx_int_t ngx_resolver_create_name_query(ngx_resolver_node_t *rn,
68     ngx_resolver_ctx_t *ctx);
69 static ngx_int_t ngx_resolver_create_addr_query(ngx_resolver_node_t *rn,
70     ngx_resolver_ctx_t *ctx);
71 static void ngx_resolver_resend_handler(ngx_event_t *ev);
72 static time_t ngx_resolver_resend(ngx_resolver_t *r, ngx_rbtree_t *tree,
73     ngx_queue_t *queue);
74 static void ngx_resolver_read_response(ngx_event_t *rev);
75 static void ngx_resolver_process_response(ngx_resolver_t *r, u_char *buf,
76     size_t n);
77 static void ngx_resolver_process_a(ngx_resolver_t *r, u_char *buf, size_t n,
78     ngx_uint_t ident, ngx_uint_t code, ngx_uint_t qtype,
79     ngx_uint_t nan, ngx_uint_t ans);
80 static void ngx_resolver_process_ptr(ngx_resolver_t *r, u_char *buf, size_t n,
81     ngx_uint_t ident, ngx_uint_t code, ngx_uint_t nan);
82 static ngx_resolver_node_t *ngx_resolver_lookup_name(ngx_resolver_t *r,
83     ngx_str_t *name, uint32_t hash);
84 static ngx_resolver_node_t *ngx_resolver_lookup_addr(ngx_resolver_t *r,
85     in_addr_t addr);
86 static void ngx_resolver_rbtree_insert_value(ngx_rbtree_node_t *temp,
87     ngx_rbtree_node_t *node, ngx_rbtree_node_t *sentinel);
88 static ngx_int_t ngx_resolver_copy(ngx_resolver_t *r, ngx_str_t *name,
89     u_char *buf, u_char *src, u_char *last);
90 static void ngx_resolver_timeout_handler(ngx_event_t *ev);
91 static void ngx_resolver_free_node(ngx_resolver_t *r, ngx_resolver_node_t *rn);
92 static void *ngx_resolver_alloc(ngx_resolver_t *r, size_t size);
93 static void *ngx_resolver_calloc(ngx_resolver_t *r, size_t size);
94 static void ngx_resolver_free(ngx_resolver_t *r, void *p);
95 static void ngx_resolver_free_locked(ngx_resolver_t *r, void *p);
96 static void *ngx_resolver_dup(ngx_resolver_t *r, void *src, size_t size);
97 static ngx_addr_t *ngx_resolver_export(ngx_resolver_t *r,
98     ngx_resolver_node_t *rn, ngx_uint_t rotate);
99 static u_char *ngx_resolver_log_error(ngx_log_t *log, u_char *buf, size_t len);
100
101 #if (NGX_HAVE_INET6)
102 static void ngx_resolver_rbtree_insert_addr6_value(ngx_rbtree_node_t *temp,
103     ngx_rbtree_node_t *node, ngx_rbtree_node_t *sentinel);
104 static ngx_resolver_node_t *ngx_resolver_lookup_addr6(ngx_resolver_t *r,
105     struct in6_addr *addr, uint32_t hash);
106 #endif
107
108
109 ngx_resolver_t *
110 ngx_resolver_create(ngx_conf_t *cf, ngx_str_t *names, ngx_uint_t n)
111 {
112     ngx_str_t          s;
113     ngx_url_t          u;
114     ngx_uint_t         i, j;
115     ngx_resolver_t     *r;

```

```

116     ngx_pool_cleanup_t *cIn;
117     ngx_udp_connection_t *uc;
118
119     cIn = ngx_pool_cleanup_add(cf->pool, 0);
120     if (cIn == NULL) {
121         return NULL;
122     }
123
124     cIn->handler = ngx_resolver_cleanup;
125
126     r = ngx_calloc(sizeof(ngx_resolver_t), cf->log);
127     if (r == NULL) {
128         return NULL;
129     }
130
131     cIn->data = r;
132
133     r->event = ngx_calloc(sizeof(ngx_event_t), cf->log);
134     if (r->event == NULL) {
135         return NULL;
136     }
137
138     ngx_rbtree_init(&r->name_rbtree, &r->name_sentinel,
139                   ngx_resolver_rbtree_insert_value);
140
141     ngx_rbtree_init(&r->addr_rbtree, &r->addr_sentinel,
142                   ngx_rbtree_insert_value);
143
144     ngx_queue_init(&r->name_resend_queue);
145     ngx_queue_init(&r->addr_resend_queue);
146
147     ngx_queue_init(&r->name_expire_queue);
148     ngx_queue_init(&r->addr_expire_queue);
149
150     #if (NGX_HAVE_INET6)
151     r->ipv6 = 1;
152
153     ngx_rbtree_init(&r->addr6_rbtree, &r->addr6_sentinel,
154                   ngx_resolver_rbtree_insert_addr6_value);
155
156     ngx_queue_init(&r->addr6_resend_queue);
157
158     ngx_queue_init(&r->addr6_expire_queue);
159     #endif
160
161     r->event->handler = ngx_resolver_resend_handler;
162     r->event->data = r;
163     r->event->log = &cf->cycle->new_log;
164     r->ident = -1;
165
166     r->resend_timeout = 5;
167     r->expire = 30;
168     r->valid = 0;
169
170     r->log = &cf->cycle->new_log;
171     r->log_level = NGX_LOG_ERR;
172
173     if (n) {
174         if (ngx_array_init(&r->udp_connections, cf->pool, n,
175                          sizeof(ngx_udp_connection_t))
176             != NGX_OK)
177         {
178             return NULL;
179         }
180     }
181
182     for (i = 0; i < n; i++) {
183         if (ngx_strncmp(names[i].data, "valid=", 6) == 0) {
184             s.len = names[i].len - 6;
185             s.data = names[i].data + 6;
186
187             r->valid = ngx_parse_time(&s, 1);
188
189             if (r->valid == (time_t) NGX_ERROR) {
190                 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
191                                   "invalid parameter: %V", &names[i]);

```

```

192         return NULL;
193     }
194
195     continue;
196 }
197
198 #if (NGX_HAVE_INET6)
199     if (ngx_strncmp(names[i].data, "ipv6=", 5) == 0) {
200
201         if (ngx_strcmp(&names[i].data[5], "on") == 0) {
202             r->ipv6 = 1;
203
204         } else if (ngx_strcmp(&names[i].data[5], "off") == 0) {
205             r->ipv6 = 0;
206
207         } else {
208             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
209                 "invalid parameter: %V", &names[i]);
210             return NULL;
211         }
212
213         continue;
214     }
215 #endif
216
217     ngx_memzero(&u, sizeof(ngx_url_t));
218
219     u.url = names[i];
220     u.default_port = 53;
221
222     if (ngx_parse_url(cf->pool, &u) != NGX_OK) {
223         if (u.err) {
224             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
225                 "%s in resolver \"%V\"",
226                 u.err, &u.url);
227         }
228
229         return NULL;
230     }
231
232     uc = ngx_array_push_n(&r->udp_connections, u.naddrs);
233     if (uc == NULL) {
234         return NULL;
235     }
236
237     ngx_memzero(uc, u.naddrs * sizeof(ngx_udp_connection_t));
238
239     for (j = 0; j < u.naddrs; j++) {
240         uc[j].sockaddr = u.addrs[j].sockaddr;
241         uc[j].socklen = u.addrs[j].socklen;
242         uc[j].server = u.addrs[j].name;
243     }
244 }
245
246 return r;
247 }
248
249
250 static void
251 ngx_resolver_cleanup(void *data)
252 {
253     ngx_resolver_t *r = data;
254
255     ngx_uint_t i;
256     ngx_udp_connection_t *uc;
257
258     if (r) {
259         ngx_log_debug0(NGX_LOG_DEBUG_CORE, ngx_cycle->log, 0,
260             "cleanup resolver");
261
262         ngx_resolver_cleanup_tree(r, &r->name_rbtrees);
263
264         ngx_resolver_cleanup_tree(r, &r->addr_rbtrees);
265
266         #if (NGX_HAVE_INET6)
267         ngx_resolver_cleanup_tree(r, &r->addr6_rbtrees);

```



```

268 #endif
269
270     if (r->event) {
271         ngx_free(r->event);
272     }
273
274
275     uc = r->udp_connections.elts;
276
277     for (i = 0; i < r->udp_connections.nelts; i++) {
278         if (uc[i].connection) {
279             ngx_close_connection(uc[i].connection);
280         }
281     }
282
283     ngx_free(r);
284 }
285 }
286
287
288 static void
289 ngx_resolver_cleanup_tree(ngx_resolver_t *r, ngx_rbtree_t *tree)
290 {
291     ngx_resolver_ctx_t *ctx, *next;
292     ngx_resolver_node_t *rn;
293
294     while (tree->root != tree->sentinel) {
295
296         rn = ngx_resolver_node(ngx_rbtree_min(tree->root, tree->sentinel));
297
298         ngx_queue_remove(&rn->queue);
299
300         for (ctx = rn->waiting; ctx; ctx = next) {
301             next = ctx->next;
302
303             if (ctx->event) {
304                 ngx_resolver_free(r, ctx->event);
305             }
306
307             ngx_resolver_free(r, ctx);
308         }
309
310         ngx_rbtree_delete(tree, &rn->node);
311
312         ngx_resolver_free_node(r, rn);
313     }
314 }
315
316
317 ngx_resolver_ctx_t *
318 ngx_resolve_start(ngx_resolver_t *r, ngx_resolver_ctx_t *temp)
319 {
320     in_addr_t      addr;
321     ngx_resolver_ctx_t *ctx;
322
323     if (temp) {
324         addr = ngx_inet_addr(temp->name.data, temp->name.len);
325
326         if (addr != INADDR_NONE) {
327             temp->resolver = r;
328             temp->state = NGX_OK;
329             temp->naddrs = 1;
330             temp->addrs = &temp->addr;
331             temp->addr.sockaddr = (struct sockaddr *) &temp->sin;
332             temp->addr.socklen = sizeof(struct sockaddr_in);
333             ngx_memzero(&temp->sin, sizeof(struct sockaddr_in));
334             temp->sin.sin_family = AF_INET;
335             temp->sin.sin_addr.s_addr = addr;
336             temp->quick = 1;
337
338             return temp;
339         }
340     }
341
342     if (r->udp_connections.nelts == 0) {
343         return NGX_NO_RESOLVER;

```

```

344     }
345
346     ctx = ngx_resolver_calloc(r, sizeof(ngx_resolver_ctx_t));
347
348     if (ctx) {
349         ctx->resolver = r;
350     }
351
352     return ctx;
353 }
354
355
356 ngx_int_t
357 ngx_resolve_name(ngx_resolver_ctx_t *ctx)
358 {
359     ngx_int_t      rc;
360     ngx_resolver_t *r;
361
362     r = ctx->resolver;
363
364     if (ctx->name.len > 0 && ctx->name.data[ctx->name.len - 1] == '.') {
365         ctx->name.len--;
366     }
367
368     ngx_log_debug1(NGX_LOG_DEBUG_CORE, r->log, 0,
369                  "resolve: \"%V\"", &ctx->name);
370
371     if (ctx->quick) {
372         ctx->handler(ctx);
373         return NGX_OK;
374     }
375
376     /* lock name mutex */
377
378     rc = ngx_resolve_name_locked(r, ctx);
379
380     if (rc == NGX_OK) {
381         return NGX_OK;
382     }
383
384     /* unlock name mutex */
385
386     if (rc == NGX_AGAIN) {
387         return NGX_OK;
388     }
389
390     /* NGX_ERROR */
391
392     if (ctx->event) {
393         ngx_resolver_free(r, ctx->event);
394     }
395
396     ngx_resolver_free(r, ctx);
397
398     return NGX_ERROR;
399 }
400
401
402 void
403 ngx_resolve_name_done(ngx_resolver_ctx_t *ctx)
404 {
405     uint32_t      hash;
406     ngx_resolver_t *r;
407     ngx_resolver_ctx_t *w, **p;
408     ngx_resolver_node_t *rn;
409
410     r = ctx->resolver;
411
412     ngx_log_debug1(NGX_LOG_DEBUG_CORE, r->log, 0,
413                  "resolve name done: %i", ctx->state);
414
415     if (ctx->quick) {
416         return;
417     }
418
419     if (ctx->event && ctx->event->timer_set) {

```

```

420     ngx_del_timer(ctx->event);
421 }
422
423 /* lock name mutex */
424
425 if (ctx->state == NGX_AGAIN) {
426
427     hash = ngx_crc32_short(ctx->name.data, ctx->name.len);
428
429     rn = ngx_resolver_lookup_name(r, &ctx->name, hash);
430
431     if (rn) {
432         p = &rn->waiting;
433         w = rn->waiting;
434
435         while (w) {
436             if (w == ctx) {
437                 *p = w->next;
438
439                 goto done;
440             }
441
442             p = &w->next;
443             w = w->next;
444         }
445     }
446
447     ngx_log_error(NGX_LOG_ALERT, r->log, 0,
448                 "could not cancel %V resolving", &ctx->name);
449 }
450
451 done:
452
453     ngx_resolver_expire(r, &r->name_rbtrees, &r->name_expire_queue);
454
455     /* unlock name mutex */
456
457     /* lock alloc mutex */
458
459     if (ctx->event) {
460         ngx_resolver_free_locked(r, ctx->event);
461     }
462
463     ngx_resolver_free_locked(r, ctx);
464
465     /* unlock alloc mutex */
466 }
467
468
469 static ngx_int_t
470 ngx_resolve_name_locked(ngx_resolver_t *r, ngx_resolver_ctx_t *ctx)
471 {
472     uint32_t      hash;
473     ngx_int_t     rc;
474     ngx_uint_t    naddrs;
475     ngx_addr_t    *addrs;
476     ngx_resolver_ctx_t *next;
477     ngx_resolver_node_t *rn;
478
479     ngx_strlow(ctx->name.data, ctx->name.data, ctx->name.len);
480
481     hash = ngx_crc32_short(ctx->name.data, ctx->name.len);
482
483     rn = ngx_resolver_lookup_name(r, &ctx->name, hash);
484
485     if (rn) {
486
487         if (rn->valid >= ngx_time()) {
488
489             ngx_log_debug0(NGX_LOG_DEBUG_CORE, r->log, 0, "resolve cached");
490
491             ngx_queue_remove(&rn->queue);
492
493             rn->expire = ngx_time() + r->expire;
494
495             ngx_queue_insert_head(&r->name_expire_queue, &rn->queue);

```

```

496         naddrs = (rn->naddrs == (u_short) -1) ? 0 : rn->naddrs;
497
498     #if (NGX_HAVE_INET6)
499         naddrs += (rn->naddrs6 == (u_short) -1) ? 0 : rn->naddrs6;
500     #endif
501
502     if (naddrs) {
503
504         if (naddrs == 1 && rn->naddrs == 1) {
505             addrs = NULL;
506
507         } else {
508             addrs = ngx_resolver_export(r, rn, 1);
509             if (addrs == NULL) {
510                 return NGX_ERROR;
511             }
512         }
513
514         ctx->next = rn->waiting;
515         rn->waiting = NULL;
516
517         /* unlock name mutex */
518
519         do {
520             ctx->state = NGX_OK;
521             ctx->naddrs = naddrs;
522
523             if (addrs == NULL) {
524                 ctx->addrs = &ctx->addr;
525                 ctx->addr.sockaddr = (struct sockaddr *) &ctx->sin;
526                 ctx->addr.socklen = sizeof(struct sockaddr_in);
527                 ngx_memzero(&ctx->sin, sizeof(struct sockaddr_in));
528                 ctx->sin.sin_family = AF_INET;
529                 ctx->sin.sin_addr.s_addr = rn->u.addr;
530
531             } else {
532                 ctx->addrs = addrs;
533             }
534
535             next = ctx->next;
536
537             ctx->handler(ctx);
538
539             ctx = next;
540         } while (ctx);
541
542         if (addrs != NULL) {
543             ngx_resolver_free(r, addrs->sockaddr);
544             ngx_resolver_free(r, addrs);
545         }
546
547         return NGX_OK;
548     }
549
550     /* NGX_RESOLVE_CNAME */
551
552     if (ctx->recursion++ < NGX_RESOLVER_MAX_RECURSION) {
553
554         ctx->name.len = rn->cnlen;
555         ctx->name.data = rn->u.cname;
556
557         return ngx_resolve_name_locked(r, ctx);
558     }
559
560     ctx->next = rn->waiting;
561     rn->waiting = NULL;
562
563     /* unlock name mutex */
564
565     do {
566         ctx->state = NGX_RESOLVE_NXDOMAIN;
567         next = ctx->next;
568
569         ctx->handler(ctx);
570
571         ctx = next;

```

```

572         } while (ctx);
573
574         return NGX\_OK;
575     }
576
577     if (rn->waiting) {
578
579         ctx->next = rn->waiting;
580         rn->waiting = ctx;
581         ctx->state = NGX\_AGAIN;
582
583         return NGX\_AGAIN;
584     }
585
586     ngx\_queue\_remove(&rn->queue);
587
588     /* lock alloc mutex */
589
590     if (rn->query) {
591         ngx\_resolver\_free\_locked(r, rn->query);
592         rn->query = NULL;
593     #if (NGX_HAVE_INET6)
594         rn->query6 = NULL;
595     #endif
596     }
597
598     if (rn->cnlen) {
599         ngx\_resolver\_free\_locked(r, rn->u.cname);
600     }
601
602     if (rn->naddrs > 1 && rn->naddrs != (u_short) -1) {
603         ngx\_resolver\_free\_locked(r, rn->u.addrs);
604     }
605
606     #if (NGX_HAVE_INET6)
607     if (rn->naddrs6 > 1 && rn->naddrs6 != (u_short) -1) {
608         ngx\_resolver\_free\_locked(r, rn->u6.addrs6);
609     }
610     #endif
611
612     /* unlock alloc mutex */
613
614     } else {
615
616         rn = ngx\_resolver\_alloc(r, sizeof(ngx\_resolver\_node\_t));
617         if (rn == NULL) {
618             return NGX\_ERROR;
619         }
620
621         rn->name = ngx\_resolver\_dup(r, ctx->name.data, ctx->name.len);
622         if (rn->name == NULL) {
623             ngx\_resolver\_free(r, rn);
624             return NGX\_ERROR;
625         }
626
627         rn->node.key = hash;
628         rn->nlen = (u_short) ctx->name.len;
629         rn->query = NULL;
630     #if (NGX_HAVE_INET6)
631         rn->query6 = NULL;
632     #endif
633
634         ngx\_rbtree\_insert(&r->name_rbtree, &rn->node);
635     }
636
637     rc = ngx\_resolver\_create\_name\_query(rn, ctx);
638
639     if (rc == NGX\_ERROR) {
640         goto failed;
641     }
642
643     if (rc == NGX\_DECLINED) {
644         ngx\_rbtree\_delete(&r->name_rbtree, &rn->node);
645
646         ngx\_resolver\_free(r, rn->query);
647         ngx\_resolver\_free(r, rn->name);

```

```

648     ngx_resolver_free(r, rn);
649
650     ctx->state = NGX_RESOLVE_NXDOMAIN;
651     ctx->handler(ctx);
652
653     return NGX_OK;
654 }
655
656 rn->naddrs = (u_short) -1;
657 #if (NGX_HAVE_INET6)
658 rn->naddrs6 = r->ipv6 ? (u_short) -1 : 0;
659 #endif
660
661 if (ngx_resolver_send_query(r, rn) != NGX_OK) {
662     goto failed;
663 }
664
665 if (ctx->event == NULL) {
666     ctx->event = ngx_resolver_calloc(r, sizeof(ngx_event_t));
667     if (ctx->event == NULL) {
668         goto failed;
669     }
670
671     ctx->event->handler = ngx_resolver_timeout_handler;
672     ctx->event->data = rn;
673     ctx->event->log = r->log;
674     rn->ident = -1;
675
676     ngx_add_timer(ctx->event, ctx->timeout);
677 }
678
679 if (ngx_queue_empty(&r->name_resend_queue)) {
680     ngx_add_timer(r->event, (ngx_msec_t) (r->resend_timeout * 1000));
681 }
682
683 rn->expire = ngx_time() + r->resend_timeout;
684
685 ngx_queue_insert_head(&r->name_resend_queue, &rn->queue);
686
687 rn->code = 0;
688 rn->cnlen = 0;
689 rn->valid = 0;
690 rn->tttl = NGX_MAX_UINT32_VALUE;
691 rn->waiting = ctx;
692
693 ctx->state = NGX_AGAIN;
694
695 return NGX_AGAIN;
696
697 failed:
698
699 ngx_rbtrees_delete(&r->name_rbtrees, &rn->node);
700
701 if (rn->query) {
702     ngx_resolver_free(r, rn->query);
703 }
704
705 ngx_resolver_free(r, rn->name);
706
707 ngx_resolver_free(r, rn);
708
709 return NGX_ERROR;
710 }
711
712
713 ngx_int_t
714 ngx_resolve_addr(ngx_resolver_ctx_t *ctx)
715 {
716     u_char          *name;
717     in_addr_t       addr;
718     ngx_queue_t     *resend_queue, *expire_queue;
719     ngx_rbtrees_t   *tree;
720     ngx_resolver_t  *r;
721     struct sockaddr_in *sin;
722     ngx_resolver_node_t *rn;
723 #if (NGX_HAVE_INET6)

```

```

724     uint32_t      hash;
725     struct sockaddr_in6 *sin6;
726 #endif
727
728 #if (NGX_SUPPRESS_WARN)
729     addr = 0;
730 #if (NGX_HAVE_INET6)
731     hash = 0;
732     sin6 = NULL;
733 #endif
734 #endif
735
736     r = ctx->resolver;
737
738     switch (ctx->addr.sockaddr->sa_family) {
739
740 #if (NGX_HAVE_INET6)
741     case AF_INET6:
742         sin6 = (struct sockaddr_in6 *) ctx->addr.sockaddr;
743         hash = ngx_crc32_short(sin6->sin6_addr.s6_addr, 16);
744
745         /* lock addr mutex */
746
747         rn = ngx_resolver_lookup_addr6(r, &sin6->sin6_addr, hash);
748
749         tree = &r->addr6_rbtrees;
750         resend_queue = &r->addr6_resend_queue;
751         expire_queue = &r->addr6_expire_queue;
752
753         break;
754 #endif
755
756     default: /* AF_INET */
757         sin = (struct sockaddr_in *) ctx->addr.sockaddr;
758         addr = ntohl(sin->sin_addr.s_addr);
759
760         /* lock addr mutex */
761
762         rn = ngx_resolver_lookup_addr(r, addr);
763
764         tree = &r->addr_rbtrees;
765         resend_queue = &r->addr_resend_queue;
766         expire_queue = &r->addr_expire_queue;
767     }
768
769     if (rn) {
770
771         if (rn->valid >= ngx_time()) {
772
773             ngx_log_debug0(NGX_LOG_DEBUG_CORE, r->log, 0, "resolve cached");
774
775             ngx_queue_remove(&rn->queue);
776
777             rn->expire = ngx_time() + r->expire;
778
779             ngx_queue_insert_head(expire_queue, &rn->queue);
780
781             name = ngx_resolver_dup(r, rn->name, rn->nlen);
782             if (name == NULL) {
783                 goto failed;
784             }
785
786             ctx->name.len = rn->nlen;
787             ctx->name.data = name;
788
789             /* unlock addr mutex */
790
791             ctx->state = NGX_OK;
792
793             ctx->handler(ctx);
794
795             ngx_resolver_free(r, name);
796
797             return NGX_OK;
798         }
799

```

```

800     if (rn->waiting) {
801
802         ctx->next = rn->waiting;
803         rn->waiting = ctx;
804         ctx->state = NGX_AGAIN;
805
806         /* unlock addr mutex */
807
808         return NGX_OK;
809     }
810
811     ngx_queue_remove(&rn->queue);
812
813     ngx_resolver_free(r, rn->query);
814     rn->query = NULL;
815 #if (NGX_HAVE_INET6)
816     rn->query6 = NULL;
817 #endif
818
819 } else {
820     rn = ngx_resolver_alloc(r, sizeof(ngx_resolver_node_t));
821     if (rn == NULL) {
822         goto failed;
823     }
824
825     switch (ctx->addr.sockaddr->sa_family) {
826
827 #if (NGX_HAVE_INET6)
828         case AF_INET6:
829             rn->addr6 = sin6->sin6_addr;
830             rn->node.key = hash;
831             break;
832 #endif
833
834         default: /* AF_INET */
835             rn->node.key = addr;
836     }
837
838     rn->query = NULL;
839 #if (NGX_HAVE_INET6)
840     rn->query6 = NULL;
841 #endif
842
843     ngx_rbtree_insert(tree, &rn->node);
844 }
845
846 if (ngx_resolver_create_addr_query(rn, ctx) != NGX_OK) {
847     goto failed;
848 }
849
850 rn->naddrs = (u_short) -1;
851 #if (NGX_HAVE_INET6)
852 rn->naddrs6 = (u_short) -1;
853 #endif
854
855 if (ngx_resolver_send_query(r, rn) != NGX_OK) {
856     goto failed;
857 }
858
859 ctx->event = ngx_resolver_calloc(r, sizeof(ngx_event_t));
860 if (ctx->event == NULL) {
861     goto failed;
862 }
863
864 ctx->event->handler = ngx_resolver_timeout_handler;
865 ctx->event->data = rn;
866 ctx->event->log = r->log;
867 rn->ident = -1;
868
869 ngx_add_timer(ctx->event, ctx->timeout);
870
871 if (ngx_queue_empty(resend_queue)) {
872     ngx_add_timer(r->event, (ngx_msec_t) (r->resend_timeout * 1000));
873 }
874
875 rn->expire = ngx_time() + r->resend_timeout;

```



```

876     ngx_queue_insert_head(resend_queue, &rn->queue);
877
878     rn->code = 0;
879     rn->cnlen = 0;
880     rn->name = NULL;
881     rn->nlen = 0;
882     rn->valid = 0;
883     rn->ttd = NGX_MAX_UINT32_VALUE;
884     rn->waiting = ctx;
885
886     /* unlock addr mutex */
887
888     ctx->state = NGX_AGAIN;
889
890     return NGX_OK;
891
892 failed:
893
894     if (rn) {
895         ngx_rbtree_delete(tree, &rn->node);
896
897         if (rn->query) {
898             ngx_resolver_free(r, rn->query);
899         }
900
901         ngx_resolver_free(r, rn);
902     }
903
904     /* unlock addr mutex */
905
906     if (ctx->event) {
907         ngx_resolver_free(r, ctx->event);
908     }
909
910     ngx_resolver_free(r, ctx);
911
912     return NGX_ERROR;
913 }
914
915 void
916 ngx_resolve_addr_done(ngx_resolver_ctx_t *ctx)
917 {
918     in_addr_t      addr;
919     ngx_queue_t    *expire_queue;
920     ngx_rbtree_t    *tree;
921     ngx_resolver_t *r;
922     ngx_resolver_ctx_t *w, **p;
923     struct sockaddr_in *sin;
924     ngx_resolver_node_t *rn;
925 #if (NGX_HAVE_INET6)
926     uint32_t      hash;
927     struct sockaddr_in6 *sin6;
928 #endif
929
930     r = ctx->resolver;
931
932     switch (ctx->addr.sockaddr->sa_family) {
933 #if (NGX_HAVE_INET6)
934     case AF_INET6:
935         tree = &r->addr6_rbtree;
936         expire_queue = &r->addr6_expire_queue;
937         break;
938 #endif
939     default: /* AF_INET */
940         tree = &r->addr_rbtree;
941         expire_queue = &r->addr_expire_queue;
942     }
943
944     ngx_log_debug1(NGX_LOG_DEBUG_CORE, r->log, 0,
945                  "resolve addr done: %i", ctx->state);
946
947     if (ctx->event && ctx->event->timer_set) {

```

```

952     ngx_del_timer(ctx->event);
953 }
954
955 /* lock addr mutex */
956
957 if (ctx->state == NGX_AGAIN) {
958
959     switch (ctx->addr.sockaddr->sa_family) {
960
961 #if (NGX_HAVE_INET6)
962     case AF_INET6:
963         sin6 = (struct sockaddr_in6 *) ctx->addr.sockaddr;
964         hash = ngx_crc32_short(sin6->sin6_addr.s6_addr, 16);
965         rn = ngx_resolver_lookup_addr6(r, &sin6->sin6_addr, hash);
966         break;
967 #endif
968
969     default: /* AF_INET */
970         sin = (struct sockaddr_in *) ctx->addr.sockaddr;
971         addr = ntohl(sin->sin_addr.s_addr);
972         rn = ngx_resolver_lookup_addr(r, addr);
973     }
974
975     if (rn) {
976         p = &rn->waiting;
977         w = rn->waiting;
978
979         while (w) {
980             if (w == ctx) {
981                 *p = w->next;
982
983                 goto done;
984             }
985
986             p = &w->next;
987             w = w->next;
988         }
989     }
990
991     {
992         u_char    text[NGX_SOCKADDR_STRLEN];
993         ngx_str_t  addrtext;
994
995         addrtext.data = text;
996         addrtext.len = ngx_sock_ntop(ctx->addr.sockaddr, ctx->addr.socklen,
997                                     text, NGX_SOCKADDR_STRLEN, 0);
998
999         ngx_log_error(NGX_LOG_ALERT, r->log, 0,
1000                    "could not cancel %V resolving", &addrtext);
1001     }
1002 }
1003
1004 done:
1005
1006     ngx_resolver_expire(r, tree, expire_queue);
1007
1008     /* unlock addr mutex */
1009
1010     /* lock alloc mutex */
1011
1012     if (ctx->event) {
1013         ngx_resolver_free_locked(r, ctx->event);
1014     }
1015
1016     ngx_resolver_free_locked(r, ctx);
1017
1018     /* unlock alloc mutex */
1019 }
1020
1021
1022 static void
1023 ngx_resolver_expire(ngx_resolver_t *r, ngx_rbtree_t *tree, ngx_queue_t *queue)
1024 {
1025     time_t    now;
1026     ngx_uint_t i;
1027     ngx_queue_t *q;

```

```

1028     ngx_resolver_node_t *rn;
1029
1030     ngx_log_debug0(NGX_LOG_DEBUG_CORE, r->log, 0, "resolver expire");
1031
1032     now = ngx_time();
1033
1034     for (i = 0; i < 2; i++) {
1035         if (ngx_queue_empty(queue)) {
1036             return;
1037         }
1038
1039         q = ngx_queue_last(queue);
1040
1041         rn = ngx_queue_data(q, ngx_resolver_node_t, queue);
1042
1043         if (now <= rn->expire) {
1044             return;
1045         }
1046
1047         ngx_log_debug2(NGX_LOG_DEBUG_CORE, r->log, 0,
1048             "resolver expire \"%s\"", (size_t) rn->nlen, rn->name);
1049
1050         ngx_queue_remove(q);
1051
1052         ngx_rbtree_delete(tree, &rn->node);
1053
1054         ngx_resolver_free_node(r, rn);
1055     }
1056 }
1057
1058
1059 static ngx_int_t
1060 ngx_resolver_send_query(ngx_resolver_t *r, ngx_resolver_node_t *rn)
1061 {
1062     ssize_t          n;
1063     ngx_udp_connection_t *uc;
1064
1065     uc = r->udp_connections.elts;
1066
1067     uc = &uc[r->last_connection++];
1068     if (r->last_connection == r->udp_connections.nelts) {
1069         r->last_connection = 0;
1070     }
1071
1072     if (uc->connection == NULL) {
1073
1074         uc->log = *r->log;
1075         uc->log.handler = ngx_resolver_log_error;
1076         uc->log.data = uc;
1077         uc->log.action = "resolving";
1078
1079         if (ngx_udp_connect(uc) != NGX_OK) {
1080             return NGX_ERROR;
1081         }
1082
1083         uc->connection->data = r;
1084         uc->connection->read->handler = ngx_resolver_read_response;
1085         uc->connection->read->resolver = 1;
1086     }
1087
1088     if (rn->naddrs == (u_short) -1) {
1089         n = ngx_send(uc->connection, rn->query, rn->qlen);
1090
1091         if (n == -1) {
1092             return NGX_ERROR;
1093         }
1094
1095         if ((size_t) n != (size_t) rn->qlen) {
1096             ngx_log_error(NGX_LOG_CRIT, &uc->log, 0, "send() incomplete");
1097             return NGX_ERROR;
1098         }
1099     }
1100
1101 #if (NGX_HAVE_INET6)
1102     if (rn->query6 && rn->naddrs6 == (u_short) -1) {
1103         n = ngx_send(uc->connection, rn->query6, rn->qlen);

```

```

1104     if (n == -1) {
1105         return NGX\_ERROR;
1106     }
1107
1108     if ((size_t) n != (size_t) rn->qlen) {
1109         ngx\_log\_error(NGX\_LOG\_CRIT, &uc->log, 0, "send() incomplete");
1110         return NGX\_ERROR;
1111     }
1112 }
1113 }
1114 #endif
1115
1116     return NGX\_OK;
1117 }
1118
1119
1120 static void
1121 ngx\_resolver\_resend\_handler(ngx\_event\_t *ev)
1122 {
1123     time_t         timer, atimer, ntimer;
1124 #if (NGX\_HAVE\_INET6)
1125     time_t         a6timer;
1126 #endif
1127     ngx\_resolver\_t *r;
1128
1129     r = ev->data;
1130
1131     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_CORE, r->log, 0,
1132                 "resolver resend handler");
1133
1134     /* lock name mutex */
1135
1136     ntimer = ngx\_resolver\_resend(r, &r->name_rbtrees, &r->name_resend_queue);
1137
1138     /* unlock name mutex */
1139
1140     /* lock addr mutex */
1141
1142     atimer = ngx\_resolver\_resend(r, &r->addr_rbtrees, &r->addr_resend_queue);
1143
1144     /* unlock addr mutex */
1145
1146 #if (NGX\_HAVE\_INET6)
1147
1148     /* lock addr6 mutex */
1149
1150     a6timer = ngx\_resolver\_resend(r, &r->addr6_rbtrees, &r->addr6_resend_queue);
1151
1152     /* unlock addr6 mutex */
1153
1154 #endif
1155
1156     timer = ntimer;
1157
1158     if (timer == 0) {
1159         timer = atimer;
1160     } else if (atimer) {
1161         timer = ngx\_min(timer, atimer);
1162     }
1163 }
1164
1165 #if (NGX\_HAVE\_INET6)
1166
1167     if (timer == 0) {
1168         timer = a6timer;
1169     } else if (a6timer) {
1170         timer = ngx\_min(timer, a6timer);
1171     }
1172 }
1173 #endif
1174
1175 #endif
1176
1177     if (timer) {
1178         ngx\_add\_timer(r->event, (ngx\_msec\_t) (timer * 1000));
1179     }

```

```

1180
1181
1182 static time_t
1183 ngx_resolver_resend(ngx_resolver_t *r, ngx_rbtree_t *tree, ngx_queue_t *queue)
1184 {
1185     time_t          now;
1186     ngx_queue_t     *q;
1187     ngx_resolver_node_t *rn;
1188
1189     now = ngx_time();
1190
1191     for ( ;; ) {
1192         if (ngx_queue_empty(queue)) {
1193             return 0;
1194         }
1195
1196         q = ngx_queue_last(queue);
1197
1198         rn = ngx_queue_data(q, ngx_resolver_node_t, queue);
1199
1200         if (now < rn->expire) {
1201             return rn->expire - now;
1202         }
1203
1204         ngx_log_debug3(NGX_LOG_DEBUG_CORE, r->log, 0,
1205             "resolver resend \"%s\" %p",
1206             (size_t) rn->nlen, rn->name, rn->waiting);
1207
1208         ngx_queue_remove(q);
1209
1210         if (rn->waiting) {
1211             (void) ngx_resolver_send_query(r, rn);
1212
1213             rn->expire = now + r->resend_timeout;
1214
1215             ngx_queue_insert_head(queue, q);
1216
1217             continue;
1218         }
1219     }
1220
1221     ngx_rbtree_delete(tree, &rn->node);
1222
1223     ngx_resolver_free_node(r, rn);
1224 }
1225 }
1226
1227
1228 static void
1229 ngx_resolver_read_response(ngx_event_t *rev)
1230 {
1231     ssize_t          n;
1232     ngx_connection_t *c;
1233     u_char           buf[NGX_RESOLVER_UDP_SIZE];
1234
1235     c = rev->data;
1236
1237     do {
1238         n = ngx_udp_recv(c, buf, NGX_RESOLVER_UDP_SIZE);
1239
1240         if (n < 0) {
1241             return;
1242         }
1243
1244         ngx_resolver_process_response(c->data, buf, n);
1245     } while (rev->ready);
1246 }
1247
1248
1249
1250 static void
1251 ngx_resolver_process_response(ngx_resolver_t *r, u_char *buf, size_t n)
1252 {
1253     char             *err;
1254     ngx_uint_t       i, times, ident, qident, flags, code, nqs, nan,
1255     qtype, qclass;

```

```

1256 #if (NGX_HAVE_INET6)
1257     ngx_uint_t         qident6;
1258 #endif
1259     ngx_queue_t        *q;
1260     ngx_resolver_qs_t  *qs;
1261     ngx_resolver_hdr_t *response;
1262     ngx_resolver_node_t *rn;
1263
1264     if (n < sizeof(ngx_resolver_hdr_t)) {
1265         goto short_response;
1266     }
1267
1268     response = (ngx_resolver_hdr_t *) buf;
1269
1270     ident = (response->ident_hi << 8) + response->ident_lo;
1271     flags = (response->flags_hi << 8) + response->flags_lo;
1272     nqs = (response->nqs_hi << 8) + response->nqs_lo;
1273     nan = (response->nan_hi << 8) + response->nan_lo;
1274
1275     ngx_log_debug6(NGX_LOG_DEBUG_CORE, r->log, 0,
1276         "resolver DNS response %ui fl:%04Xui %ui/%ui/%ud/%ud",
1277         ident, flags, nqs, nan,
1278         (response->nns_hi << 8) + response->nns_lo,
1279         (response->nar_hi << 8) + response->nar_lo);
1280
1281     /* response to a standard query */
1282     if ((flags & 0xf870) != 0x8000) {
1283         ngx_log_error(r->log_level, r->log, 0,
1284             "invalid DNS response %ui fl:%04Xui", ident, flags);
1285         return;
1286     }
1287
1288     code = flags & 0xf;
1289
1290     if (code == NGX_RESOLVE_FORMERR) {
1291
1292         times = 0;
1293
1294         for (q = ngx_queue_head(&r->name_resend_queue);
1295             q != ngx_queue_sentinel(&r->name_resend_queue) || times++ < 100;
1296             q = ngx_queue_next(q))
1297         {
1298             rn = ngx_queue_data(q, ngx_resolver_node_t, queue);
1299             qident = (rn->query[0] << 8) + rn->query[1];
1300
1301             if (qident == ident) {
1302                 goto dns_error_name;
1303             }
1304         }
1305     #if (NGX_HAVE_INET6)
1306         if (rn->query6) {
1307             qident6 = (rn->query6[0] << 8) + rn->query6[1];
1308
1309             if (qident6 == ident) {
1310                 goto dns_error_name;
1311             }
1312         }
1313     #endif
1314     }
1315
1316     goto dns_error;
1317 }
1318
1319 if (code > NGX_RESOLVE_REFUSED) {
1320     goto dns_error;
1321 }
1322
1323 if (nqs != 1) {
1324     err = "invalid number of questions in DNS response";
1325     goto done;
1326 }
1327
1328 i = sizeof(ngx_resolver_hdr_t);
1329
1330 while (i < (ngx_uint_t) n) {
1331     if (buf[i] == '\0') {

```

```

1332         goto found;
1333     }
1334
1335     i += 1 + buf[i];
1336 }
1337
1338 goto short_response;
1339
1340 found:
1341
1342 if (i++ == sizeof(ngx_resolver_hdr_t)) {
1343     err = "zero-length domain name in DNS response";
1344     goto done;
1345 }
1346
1347 if (i + sizeof(ngx_resolver_qs_t) + nan * (2 + sizeof(ngx_resolver_an_t))
1348     > (ngx_uint_t) n)
1349 {
1350     goto short_response;
1351 }
1352
1353 qs = (ngx_resolver_qs_t *) &buf[i];
1354
1355 qtype = (qs->type_hi << 8) + qs->type_lo;
1356 qclass = (qs->class_hi << 8) + qs->class_lo;
1357
1358 ngx_log_debug2(NGX_LOG_DEBUG_CORE, r->log, 0,
1359     "resolver DNS response qt:%ui cl:%ui", qtype, qclass);
1360
1361 if (qclass != 1) {
1362     ngx_log_error(r->log_level, r->log, 0,
1363         "unknown query class %ui in DNS response", qclass);
1364     return;
1365 }
1366
1367 switch (qtype) {
1368
1369     case NGX_RESOLVE_A:
1370 #if (NGX_HAVE_INET6)
1371     case NGX_RESOLVE_AAAA:
1372 #endif
1373
1374         ngx_resolver_process_a(r, buf, n, ident, code, qtype, nan,
1375             i + sizeof(ngx_resolver_qs_t));
1376
1377         break;
1378
1379     case NGX_RESOLVE_PTR:
1380
1381         ngx_resolver_process_ptr(r, buf, n, ident, code, nan);
1382
1383         break;
1384
1385     default:
1386         ngx_log_error(r->log_level, r->log, 0,
1387             "unknown query type %ui in DNS response", qtype);
1388         return;
1389 }
1390
1391 return;
1392
1393 short_response:
1394
1395     err = "short DNS response";
1396
1397 done:
1398
1399     ngx_log_error(r->log_level, r->log, 0, err);
1400
1401     return;
1402
1403 dns_error_name:
1404
1405     ngx_log_error(r->log_level, r->log, 0,
1406         "DNS error (%ui: %s), query id:%ui, name:\"%*s\"",
1407         code, ngx_resolver_strerror(code), ident,

```

```

1408         rn->nlen, rn->name);
1409     return;
1410
1411 dns_error:
1412
1413     ngx_log_error(r->log_level, r->log, 0,
1414                 "DNS error (%ui: %s), query id:%ui",
1415                 code, ngx_resolver_strerror(code), ident);
1416     return;
1417 }
1418
1419
1420 static void
1421 ngx_resolver_process_a(ngx_resolver_t *r, u_char *buf, size_t last,
1422                       ngx_uint_t ident, ngx_uint_t code, ngx_uint_t qtype,
1423                       ngx_uint_t nan, ngx_uint_t ans)
1424 {
1425     char *err;
1426     u_char *cname;
1427     size_t len;
1428     int32_t ttl;
1429     uint32_t hash;
1430     in_addr_t *addr;
1431     ngx_str_t name;
1432     ngx_addr_t *addrs;
1433     ngx_uint_t type, class, qident, naddrs, a, i, n, start;
1434     #if (NGX_HAVE_INET6)
1435     struct in6_addr *addr6;
1436     #endif
1437     ngx_resolver_an_t *an;
1438     ngx_resolver_ctx_t *ctx, *next;
1439     ngx_resolver_node_t *rn;
1440
1441     if (ngx_resolver_copy(r, &name, buf,
1442                          buf + sizeof(ngx_resolver_hdr_t), buf + last)
1443         != NGX_OK)
1444     {
1445         return;
1446     }
1447
1448     ngx_log_debug1(NGX_LOG_DEBUG_CORE, r->log, 0, "resolver qs:%V", &name);
1449
1450     hash = ngx_crc32_short(name.data, name.len);
1451
1452     /* lock name mutex */
1453
1454     rn = ngx_resolver_lookup_name(r, &name, hash);
1455
1456     if (rn == NULL) {
1457         ngx_log_error(r->log_level, r->log, 0,
1458                     "unexpected response for %V", &name);
1459         ngx_resolver_free(r, name.data);
1460         goto failed;
1461     }
1462
1463     switch (qtype) {
1464
1465     #if (NGX_HAVE_INET6)
1466     case NGX_RESOLVE_AAAA:
1467
1468         if (rn->query6 == NULL || rn->naddrs6 != (u_short) -1) {
1469             ngx_log_error(r->log_level, r->log, 0,
1470                         "unexpected response for %V", &name);
1471             ngx_resolver_free(r, name.data);
1472             goto failed;
1473         }
1474
1475         qident = (rn->query6[0] << 8) + rn->query6[1];
1476
1477         break;
1478     #endif
1479
1480     default: /* NGX_RESOLVE_A */
1481
1482         if (rn->query == NULL || rn->naddrs != (u_short) -1) {
1483             ngx_log_error(r->log_level, r->log, 0,

```



```

1484         "unexpected response for %V", &name);
1485         ngx_resolver_free(r, name.data);
1486         goto failed;
1487     }
1488
1489     qident = (rn->query[0] << 8) + rn->query[1];
1490 }
1491
1492 if (ident != qident) {
1493     ngx_log_error(r->log_level, r->log, 0,
1494         "wrong ident %ui response for %V, expect %ui",
1495         ident, &name, qident);
1496     ngx_resolver_free(r, name.data);
1497     goto failed;
1498 }
1499
1500 ngx_resolver_free(r, name.data);
1501
1502 if (code == 0 && rn->code) {
1503     code = rn->code;
1504 }
1505
1506 if (code == 0 && nan == 0) {
1507
1508 #if (NGX_HAVE_INET6)
1509     switch (qtype) {
1510
1511     case NGX_RESOLVE_AAAA:
1512
1513         rn->naddrs6 = 0;
1514
1515         if (rn->naddrs == (u_short) -1) {
1516             goto next;
1517         }
1518
1519         if (rn->naddrs) {
1520             goto export;
1521         }
1522
1523         break;
1524
1525     default: /* NGX_RESOLVE_A */
1526
1527         rn->naddrs = 0;
1528
1529         if (rn->naddrs6 == (u_short) -1) {
1530             goto next;
1531         }
1532
1533         if (rn->naddrs6) {
1534             goto export;
1535         }
1536     }
1537 #endif
1538
1539     code = NGX_RESOLVE_NXDOMAIN;
1540 }
1541
1542 if (code) {
1543
1544 #if (NGX_HAVE_INET6)
1545     switch (qtype) {
1546
1547     case NGX_RESOLVE_AAAA:
1548
1549         rn->naddrs6 = 0;
1550
1551         if (rn->naddrs == (u_short) -1) {
1552             rn->code = (u_char) code;
1553             goto next;
1554         }
1555
1556         break;
1557
1558     default: /* NGX_RESOLVE_A */

```

```

1560         rn->naddrs = 0;
1561
1562         if (rn->naddrs6 == (u_short) -1) {
1563             rn->code = (u_char) code;
1564             goto next;
1565         }
1566     }
1567 #endif
1568
1569     next = rn->waiting;
1570     rn->waiting = NULL;
1571
1572     ngx_queue_remove(&rn->queue);
1573
1574     ngx_rbtree_delete(&r->name_rbtree, &rn->node);
1575
1576     /* unlock name mutex */
1577
1578     while (next) {
1579         ctx = next;
1580         ctx->state = code;
1581         next = ctx->next;
1582
1583         ctx->handler(ctx);
1584     }
1585
1586     ngx_resolver_free_node(r, rn);
1587
1588     return;
1589 }
1590
1591 i = ans;
1592 naddrs = 0;
1593 cname = NULL;
1594
1595 for (a = 0; a < nan; a++) {
1596
1597     start = i;
1598
1599     while (i < last) {
1600
1601         if (buf[i] & 0xc0) {
1602             i += 2;
1603             goto found;
1604         }
1605
1606         if (buf[i] == 0) {
1607             i++;
1608             goto test_length;
1609         }
1610
1611         i += 1 + buf[i];
1612     }
1613
1614     goto short_response;
1615
1616 test_length:
1617
1618     if (i - start < 2) {
1619         err = "invalid name in DNS response";
1620         goto invalid;
1621     }
1622
1623 found:
1624
1625     if (i + sizeof(ngx_resolver_an_t) >= last) {
1626         goto short_response;
1627     }
1628
1629     an = (ngx_resolver_an_t *) &buf[i];
1630
1631     type = (an->type_hi << 8) + an->type_lo;
1632     class = (an->class_hi << 8) + an->class_lo;
1633     len = (an->len_hi << 8) + an->len_lo;
1634     ttl = (an->ttl[0] << 24) + (an->ttl[1] << 16)
1635           + (an->ttl[2] << 8) + (an->ttl[3]);

```

```

1636
1637     if (class != 1) {
1638         ngx_log_error(r->log_level, r->log, 0,
1639             "unexpected RR class %ui", class);
1640         goto failed;
1641     }
1642
1643     if (ttl < 0) {
1644         ttl = 0;
1645     }
1646
1647     rn->ttl = ngx_min(rn->ttl, (uint32_t) ttl);
1648
1649     i += sizeof(ngx_resolver_an_t);
1650
1651     switch (type) {
1652
1653     case NGX_RESOLVE_A:
1654
1655         if (qtype != NGX_RESOLVE_A) {
1656             err = "unexpected A record in DNS response";
1657             goto invalid;
1658         }
1659
1660         if (len != 4) {
1661             err = "invalid A record in DNS response";
1662             goto invalid;
1663         }
1664
1665         if (i + 4 > last) {
1666             goto short_response;
1667         }
1668
1669         naddrs++;
1670
1671         break;
1672
1673     #if (NGX_HAVE_INET6)
1674     case NGX_RESOLVE_AAAA:
1675
1676         if (qtype != NGX_RESOLVE_AAAA) {
1677             err = "unexpected AAAA record in DNS response";
1678             goto invalid;
1679         }
1680
1681         if (len != 16) {
1682             err = "invalid AAAA record in DNS response";
1683             goto invalid;
1684         }
1685
1686         if (i + 16 > last) {
1687             goto short_response;
1688         }
1689
1690         naddrs++;
1691
1692         break;
1693     #endif
1694
1695     case NGX_RESOLVE_CNAME:
1696
1697         cname = &buf[i];
1698
1699         break;
1700
1701     case NGX_RESOLVE_DNAME:
1702
1703         break;
1704
1705     default:
1706
1707         ngx_log_error(r->log_level, r->log, 0,
1708             "unexpected RR type %ui", type);
1709     }
1710
1711     i += len;

```

```

1712     }
1713
1714     ngx_log_debug3(NGX_LOG_DEBUG_CORE, r->log, 0,
1715         "resolver naddrs:%ui cname:%p ttl:%uD",
1716         naddrs, cname, rn->ttl);
1717
1718     if (naddrs) {
1719
1720         switch (qtype) {
1721
1722             #if (NGX_HAVE_INET6)
1723             case NGX_RESOLVE_AAAA:
1724
1725                 if (naddrs == 1) {
1726                     addr6 = &rn->u6.addr6;
1727                     rn->naddrs6 = 1;
1728
1729                 } else {
1730                     addr6 = ngx_resolver_alloc(r, naddrs * sizeof(struct in6_addr));
1731                     if (addr6 == NULL) {
1732                         goto failed;
1733                     }
1734
1735                     rn->u6.addr6 = addr6;
1736                     rn->naddrs6 = (u_short) naddrs;
1737                 }
1738
1739             #if (NGX_SUPPRESS_WARN)
1740                 addr = NULL;
1741             #endif
1742
1743                 break;
1744             #endif
1745
1746             default: /* NGX_RESOLVE_A */
1747
1748                 if (naddrs == 1) {
1749                     addr = &rn->u.addr;
1750                     rn->naddrs = 1;
1751
1752                 } else {
1753                     addr = ngx_resolver_alloc(r, naddrs * sizeof(in_addr_t));
1754                     if (addr == NULL) {
1755                         goto failed;
1756                     }
1757
1758                     rn->u.addr = addr;
1759                     rn->naddrs = (u_short) naddrs;
1760                 }
1761
1762             #if (NGX_HAVE_INET6 && NGX_SUPPRESS_WARN)
1763                 addr6 = NULL;
1764             #endif
1765         }
1766
1767         n = 0;
1768         i = ans;
1769
1770         for (a = 0; a < nan; a++) {
1771
1772             for ( ;; ) {
1773
1774                 if (buf[i] & 0xc0) {
1775                     i += 2;
1776                     break;
1777                 }
1778
1779                 if (buf[i] == 0) {
1780                     i++;
1781                     break;
1782                 }
1783
1784                 i += 1 + buf[i];
1785             }
1786
1787             an = (ngx_resolver_an_t *) &buf[i];

```

```

1788         type = (an->type_hi << 8) + an->type_lo;
1789         len = (an->len_hi << 8) + an->len_lo;
1790
1791         i += sizeof(ngx\_resolver\_an\_t);
1792
1793         if (type == NGX\_RESOLVE\_A) {
1794
1795             addr[n] = htonl((buf[i] << 24) + (buf[i + 1] << 16)
1796                 + (buf[i + 2] << 8) + (buf[i + 3]));
1797
1798             if (++n == naddrs) {
1799
1800                 #if \(NGX\_HAVE\_INET6\)
1801                     if (rn->naddrs6 == (u_short) -1) {
1802                         goto next;
1803                     }
1804                 #endif
1805
1806                 break;
1807             }
1808         }
1809     }
1810
1811     #if \(NGX\_HAVE\_INET6\)
1812     else if (type == NGX\_RESOLVE\_AAAA) {
1813
1814         ngx\_memcpy(addr6[n].s6_addr, &buf[i], 16);
1815
1816         if (++n == naddrs) {
1817
1818             if (rn->naddrs == (u_short) -1) {
1819                 goto next;
1820             }
1821
1822             break;
1823         }
1824     }
1825 #endif
1826
1827     i += len;
1828 }
1829 }
1830
1831 switch (qtype) {
1832
1833 #if \(NGX\_HAVE\_INET6\)
1834     case NGX\_RESOLVE\_AAAA:
1835
1836         if (rn->naddrs6 == (u_short) -1) {
1837             rn->naddrs6 = 0;
1838         }
1839
1840         break;
1841 #endif
1842
1843     default: /* NGX\_RESOLVE\_A */
1844
1845         if (rn->naddrs == (u_short) -1) {
1846             rn->naddrs = 0;
1847         }
1848     }
1849
1850     if (rn->naddrs != (u_short) -1
1851 #if \(NGX\_HAVE\_INET6\)
1852         && rn->naddrs6 != (u_short) -1
1853 #endif
1854         && rn->naddrs
1855 #if \(NGX\_HAVE\_INET6\)
1856         + rn->naddrs6
1857 #endif
1858         > 0)
1859     {
1860
1861 #if \(NGX\_HAVE\_INET6\)
1862     export:
1863 #endif

```

```

1864     naddrs = rn->naddrs;
1865
1866     #if (NGX_HAVE_INET6)
1867     naddrs += rn->naddrs6;
1868     #endif
1869
1870     if (naddrs == 1 && rn->naddrs == 1) {
1871         addrs = NULL;
1872
1873     } else {
1874         addrs = ngx_resolver_export(r, rn, 0);
1875         if (addrs == NULL) {
1876             goto failed;
1877         }
1878     }
1879
1880     ngx_queue_remove(&rn->queue);
1881
1882     rn->valid = ngx_time() + (r->valid ? r->valid : (time_t) rn->ttd);
1883     rn->expire = ngx_time() + r->expire;
1884
1885     ngx_queue_insert_head(&r->name_expire_queue, &rn->queue);
1886
1887     next = rn->waiting;
1888     rn->waiting = NULL;
1889
1890     /* unlock name mutex */
1891
1892     while (next) {
1893         ctx = next;
1894         ctx->state = NGX_OK;
1895         ctx->naddrs = naddrs;
1896
1897         if (addrs == NULL) {
1898             ctx->addrs = &ctx->addr;
1899             ctx->addr.sockaddr = (struct sockaddr *) &ctx->sin;
1900             ctx->addr.socklen = sizeof(struct sockaddr_in);
1901             ngx_memzero(&ctx->sin, sizeof(struct sockaddr_in));
1902             ctx->sin.sin_family = AF_INET;
1903             ctx->sin.sin_addr.s_addr = rn->u.addr;
1904
1905         } else {
1906             ctx->addrs = addrs;
1907         }
1908
1909         next = ctx->next;
1910
1911         ctx->handler(ctx);
1912     }
1913
1914     if (addrs != NULL) {
1915         ngx_resolver_free(r, addrs->sockaddr);
1916         ngx_resolver_free(r, addrs);
1917     }
1918
1919     ngx_resolver_free(r, rn->query);
1920     rn->query = NULL;
1921     #if (NGX_HAVE_INET6)
1922     rn->query6 = NULL;
1923     #endif
1924
1925     return;
1926 }
1927
1928 if (cname) {
1929     /* CNAME only */
1930
1931     if (rn->naddrs == (u_short) -1
1932     #if (NGX_HAVE_INET6)
1933         || rn->naddrs6 == (u_short) -1
1934     #endif
1935     )
1936     {
1937         goto next;
1938     }
1939 }

```

```

1940
1941     if (ngx_resolver_copy(r, &name, buf, cname, buf + last) != NGX_OK) {
1942         goto failed;
1943     }
1944
1945     ngx_log_debug1(NGX_LOG_DEBUG_CORE, r->log, 0,
1946         "resolver cname:\"%V\"", &name);
1947
1948     ngx_queue_remove(&rn->queue);
1949
1950     rn->cnlen = (u_short) name.len;
1951     rn->u.cname = name.data;
1952
1953     rn->valid = ngx_time() + (r->valid ? r->valid : (time_t) rn->ttd);
1954     rn->expire = ngx_time() + r->expire;
1955
1956     ngx_queue_insert_head(&r->name_expire_queue, &rn->queue);
1957
1958     ctx = rn->waiting;
1959     rn->waiting = NULL;
1960
1961     if (ctx) {
1962         ctx->name = name;
1963
1964         (void) ngx_resolve_name_locked(r, ctx);
1965     }
1966
1967     ngx_resolver_free(r, rn->query);
1968     rn->query = NULL;
1969 #if (NGX_HAVE_INET6)
1970     rn->query6 = NULL;
1971 #endif
1972
1973     /* unlock name mutex */
1974
1975     return;
1976 }
1977
1978 ngx_log_error(r->log_level, r->log, 0,
1979     "no A or CNAME types in DNS response");
1980 return;
1981
1982 short_response:
1983
1984     err = "short DNS response";
1985
1986 invalid:
1987
1988     /* unlock name mutex */
1989
1990     ngx_log_error(r->log_level, r->log, 0, err);
1991
1992     return;
1993
1994 failed:
1995
1996 next:
1997
1998     /* unlock name mutex */
1999
2000     return;
2001 }
2002
2003
2004 static void
2005 ngx_resolver_process_ptr(ngx_resolver_t *r, u_char *buf, size_t n,
2006     ngx_uint_t ident, ngx_uint_t code, ngx_uint_t nan)
2007 {
2008     char                *err;
2009     size_t              len;
2010     u_char              text[NGX_SOCKADDR_STRLEN];
2011     in_addr_t          addr;
2012     int32_t             ttl;
2013     ngx_int_t         octet;
2014     ngx_str_t         name;
2015     ngx_uint_t        i, mask, qident, class;

```

```

2016     ngx_queue_t      *expire_queue;
2017     ngx_rbtree_t     *tree;
2018     ngx_resolver_an_t *an;
2019     ngx_resolver_ctx_t *ctx, *next;
2020     ngx_resolver_node_t *rn;
2021 #if (NGX_HAVE_INET6)
2022     uint32_t          hash;
2023     ngx_int_t         digit;
2024     struct in6_addr   addr6;
2025 #endif
2026
2027     if (ngx_resolver_copy(r, NULL, buf,
2028                          buf + sizeof(ngx_resolver_hdr_t), buf + n)
2029         != NGX_OK)
2030     {
2031         return;
2032     }
2033
2034     /* AF_INET */
2035
2036     addr = 0;
2037     i = sizeof(ngx_resolver_hdr_t);
2038
2039     for (mask = 0; mask < 32; mask += 8) {
2040         len = buf[i++];
2041
2042         octet = ngx_atoi(&buf[i], len);
2043         if (octet == NGX_ERROR || octet > 255) {
2044             goto invalid_in_addr_arpa;
2045         }
2046
2047         addr += octet << mask;
2048         i += len;
2049     }
2050
2051     if (ngx_strcasecmp(&buf[i], (u_char *) "\7in-addr\4arpa") == 0) {
2052         i += sizeof("\7in-addr\4arpa");
2053
2054         /* lock addr mutex */
2055
2056         rn = ngx_resolver_lookup_addr(r, addr);
2057
2058         tree = &r->addr_rbtree;
2059         expire_queue = &r->addr_expire_queue;
2060
2061         addr = htonl(addr);
2062         name.len = ngx_inet_ntop(AF_INET, &addr, text, NGX_SOCKADDR_STRLEN);
2063         name.data = text;
2064
2065         goto valid;
2066     }
2067
2068 invalid_in_addr_arpa:
2069
2070 #if (NGX_HAVE_INET6)
2071
2072     i = sizeof(ngx_resolver_hdr_t);
2073
2074     for (octet = 15; octet >= 0; octet--) {
2075         if (buf[i++] != '\1') {
2076             goto invalid_ip6_arpa;
2077         }
2078
2079         digit = ngx_hextoi(&buf[i++], 1);
2080         if (digit == NGX_ERROR) {
2081             goto invalid_ip6_arpa;
2082         }
2083
2084         addr6.s6_addr[octet] = (u_char) digit;
2085
2086         if (buf[i++] != '\1') {
2087             goto invalid_ip6_arpa;
2088         }
2089
2090         digit = ngx_hextoi(&buf[i++], 1);
2091         if (digit == NGX_ERROR) {

```



```

2092     goto invalid_ip6_arpa;
2093 }
2094
2095     addr6.s6_addr[octet] += (u_char) (digit * 16);
2096 }
2097
2098     if (ngx_strcasecmp(&buf[i], (u_char *) "\3ip6\4arpa") == 0) {
2099         i += sizeof("\3ip6\4arpa");
2100
2101         /* lock addr mutex */
2102
2103         hash = ngx_crc32_short(addr6.s6_addr, 16);
2104         rn = ngx_resolver_lookup_addr6(r, &addr6, hash);
2105
2106         tree = &r->addr6_rbtrees;
2107         expire_queue = &r->addr6_expire_queue;
2108
2109         name.len = ngx_inet6_ntop(addr6.s6_addr, text, NGX_SOCKADDR_STRLEN);
2110         name.data = text;
2111
2112         goto valid;
2113     }
2114
2115 invalid_ip6_arpa:
2116 #endif
2117
2118     ngx_log_error(r->log_level, r->log, 0,
2119                 "invalid in-addr.arpa or ip6.arpa name in DNS response");
2120     return;
2121
2122 valid:
2123
2124     if (rn == NULL || rn->query == NULL) {
2125         ngx_log_error(r->log_level, r->log, 0,
2126                     "unexpected response for %V", &name);
2127         goto failed;
2128     }
2129
2130     qident = (rn->query[0] << 8) + rn->query[1];
2131
2132     if (ident != qident) {
2133         ngx_log_error(r->log_level, r->log, 0,
2134                     "wrong ident %ui response for %V, expect %ui",
2135                     ident, &name, qident);
2136         goto failed;
2137     }
2138
2139     if (code == 0 && nan == 0) {
2140         code = NGX_RESOLVE_NXDOMAIN;
2141     }
2142
2143     if (code) {
2144         next = rn->waiting;
2145         rn->waiting = NULL;
2146
2147         ngx_queue_remove(&rn->queue);
2148
2149         ngx_rbtrees_delete(tree, &rn->node);
2150
2151         /* unlock addr mutex */
2152
2153         while (next) {
2154             ctx = next;
2155             ctx->state = code;
2156             next = ctx->next;
2157
2158             ctx->handler(ctx);
2159         }
2160
2161         ngx_resolver_free_node(r, rn);
2162
2163         return;
2164     }
2165
2166     i += sizeof(ngx_resolver_qs_t);
2167

```

```

2168     if (i + 2 + sizeof(ngx\_resolver\_an\_t) >= n) {
2169         goto short_response;
2170     }
2171
2172     /* compression pointer to *.arpa */
2173
2174     if (buf[i] != 0xc0 || buf[i + 1] != sizeof(ngx\_resolver\_hdr\_t)) {
2175         err = "invalid in-addr.arpa or ip6.arpa name in DNS response";
2176         goto invalid;
2177     }
2178
2179     an = (ngx\_resolver\_an\_t *) &buf[i + 2];
2180
2181     class = (an->class_hi << 8) + an->class_lo;
2182     len = (an->len_hi << 8) + an->len_lo;
2183     ttl = (an->ttl[0] << 24) + (an->ttl[1] << 16)
2184         + (an->ttl[2] << 8) + (an->ttl[3]);
2185
2186     if (class != 1) {
2187         ngx\_log\_error(r->log_level, r->log, 0,
2188             "unexpected RR class %ui", class);
2189         goto failed;
2190     }
2191
2192     if (ttl < 0) {
2193         ttl = 0;
2194     }
2195
2196     ngx\_log\_debug3(NGX\_LOG\_DEBUG\_CORE, r->log, 0,
2197         "resolver qt:%ui cl:%ui len:%uz",
2198         (an->type_hi << 8) + an->type_lo,
2199         class, len);
2200
2201     i += 2 + sizeof(ngx\_resolver\_an\_t);
2202
2203     if (i + len > n) {
2204         goto short_response;
2205     }
2206
2207     if (ngx\_resolver\_copy(r, &name, buf, buf + i, buf + n) != NGX\_OK) {
2208         goto failed;
2209     }
2210
2211     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_CORE, r->log, 0, "resolver an:%v", &name);
2212
2213     if (name.len != (size_t) rn->nlen
2214         || ngx\_strncmp(name.data, rn->name, name.len) != 0)
2215     {
2216         if (rn->nlen) {
2217             ngx\_resolver\_free(r, rn->name);
2218         }
2219
2220         rn->nlen = (u_short) name.len;
2221         rn->name = name.data;
2222
2223         name.data = ngx\_resolver\_dup(r, rn->name, name.len);
2224         if (name.data == NULL) {
2225             goto failed;
2226         }
2227     }
2228
2229     ngx\_queue\_remove(&rn->queue);
2230
2231     rn->valid = ngx\_time() + (r->valid ? r->valid : ttl);
2232     rn->expire = ngx\_time() + r->expire;
2233
2234     ngx\_queue\_insert\_head(expire_queue, &rn->queue);
2235
2236     next = rn->waiting;
2237     rn->waiting = NULL;
2238
2239     /* unlock addr mutex */
2240
2241     while (next) {
2242         ctx = next;
2243         ctx->state = NGX\_OK;

```

```

2244     ctx->name = name;
2245     next = ctx->next;
2246
2247     ctx->handler(ctx);
2248 }
2249
2250     ngx_resolver_free(r, name.data);
2251
2252     return;
2253
2254 short_response:
2255
2256     err = "short DNS response";
2257
2258 invalid:
2259
2260     /* unlock addr mutex */
2261
2262     ngx_log_error(r->log_level, r->log, 0, err);
2263
2264     return;
2265
2266 failed:
2267
2268     /* unlock addr mutex */
2269
2270     return;
2271 }
2272
2273
2274 static ngx_resolver_node_t *
2275 ngx_resolver_lookup_name(ngx_resolver_t *r, ngx_str_t *name, uint32_t hash)
2276 {
2277     ngx_int_t rc;
2278     ngx_rbtree_node_t *node, *sentinel;
2279     ngx_resolver_node_t *rn;
2280
2281     node = r->name_rbtree.root;
2282     sentinel = r->name_rbtree.sentinel;
2283
2284     while (node != sentinel) {
2285
2286         if (hash < node->key) {
2287             node = node->left;
2288             continue;
2289         }
2290
2291         if (hash > node->key) {
2292             node = node->right;
2293             continue;
2294         }
2295
2296         /* hash == node->key */
2297
2298         rn = ngx_resolver_node(node);
2299
2300         rc = ngx_memn2cmp(name->data, rn->name, name->len, rn->nlen);
2301
2302         if (rc == 0) {
2303             return rn;
2304         }
2305
2306         node = (rc < 0) ? node->left : node->right;
2307     }
2308
2309     /* not found */
2310
2311     return NULL;
2312 }
2313
2314
2315 static ngx_resolver_node_t *
2316 ngx_resolver_lookup_addr(ngx_resolver_t *r, in_addr_t addr)
2317 {
2318     ngx_rbtree_node_t *node, *sentinel;
2319

```

```

2320     node = r->addr_rbtree.root;
2321     sentinel = r->addr_rbtree.sentinel;
2322
2323     while (node != sentinel) {
2324
2325         if (addr < node->key) {
2326             node = node->left;
2327             continue;
2328         }
2329
2330         if (addr > node->key) {
2331             node = node->right;
2332             continue;
2333         }
2334
2335         /* addr == node->key */
2336
2337         return ngx_resolver_node(node);
2338     }
2339
2340     /* not found */
2341
2342     return NULL;
2343 }
2344
2345
2346 #if (NGX_HAVE_INET6)
2347
2348 static ngx_resolver_node_t *
2349 ngx_resolver_lookup_addr6(ngx_resolver_t *r, struct in6_addr *addr,
2350                          uint32_t hash)
2351 {
2352     ngx_int_t      rc;
2353     ngx_rbtree_node_t *node, *sentinel;
2354     ngx_resolver_node_t *rn;
2355
2356     node = r->addr6_rbtree.root;
2357     sentinel = r->addr6_rbtree.sentinel;
2358
2359     while (node != sentinel) {
2360
2361         if (hash < node->key) {
2362             node = node->left;
2363             continue;
2364         }
2365
2366         if (hash > node->key) {
2367             node = node->right;
2368             continue;
2369         }
2370
2371         /* hash == node->key */
2372
2373         rn = ngx_resolver_node(node);
2374
2375         rc = ngx_memcmp(addr, &rn->addr6, 16);
2376
2377         if (rc == 0) {
2378             return rn;
2379         }
2380
2381         node = (rc < 0) ? node->left : node->right;
2382     }
2383
2384     /* not found */
2385
2386     return NULL;
2387 }
2388
2389 #endif
2390
2391
2392 static void
2393 ngx_resolver_rbtree_insert_value(ngx_rbtree_node_t *temp,
2394                                 ngx_rbtree_node_t *node, ngx_rbtree_node_t *sentinel)
2395 {

```

```

2396 ngx\_rbtree\_node\_t **p;
2397 ngx\_resolver\_node\_t *rn, *rn_temp;
2398
2399 for ( ;; ) {
2400
2401     if (node->key < temp->key) {
2402         p = &temp->left;
2403     } else if (node->key > temp->key) {
2404         p = &temp->right;
2405     } else { /* node->key == temp->key */
2406
2407         rn = ngx\_resolver\_node(node);
2408         rn_temp = ngx\_resolver\_node(temp);
2409
2410         p = (ngx\_memn2cmp(rn->name, rn_temp->name, rn->nlen, rn_temp->nlen)
2411             < 0) ? &temp->left : &temp->right;
2412     }
2413
2414     if (*p == sentinel) {
2415         break;
2416     }
2417
2418     temp = *p;
2419 }
2420
2421 *p = node;
2422 node->parent = temp;
2423 node->left = sentinel;
2424 node->right = sentinel;
2425 ngx\_rbt\_red(node);
2426 }
2427
2428 #if (NGX_HAVE_INET6)
2429
2430 static void
2431 ngx\_resolver\_rbtree\_insert\_addr6\_value(ngx\_rbtree\_node\_t *temp,
2432 ngx\_rbtree\_node\_t *node, ngx\_rbtree\_node\_t *sentinel)
2433 {
2434     ngx\_rbtree\_node\_t **p;
2435     ngx\_resolver\_node\_t *rn, *rn_temp;
2436
2437     for ( ;; ) {
2438
2439         if (node->key < temp->key) {
2440             p = &temp->left;
2441         } else if (node->key > temp->key) {
2442             p = &temp->right;
2443         } else { /* node->key == temp->key */
2444
2445             rn = ngx\_resolver\_node(node);
2446             rn_temp = ngx\_resolver\_node(temp);
2447
2448             p = (ngx\_memcmp(&rn->addr6, &rn_temp->addr6, 16)
2449                 < 0) ? &temp->left : &temp->right;
2450         }
2451
2452         if (*p == sentinel) {
2453             break;
2454         }
2455
2456         temp = *p;
2457     }
2458
2459     *p = node;
2460     node->parent = temp;
2461     node->left = sentinel;
2462     node->right = sentinel;

```

```

2472     ngx_rbt_red(node);
2473 }
2474
2475 #endif
2476
2477
2478 static ngx_int_t
2479 ngx_resolver_create_name_query(ngx_resolver_node_t *rn, ngx_resolver_ctx_t *ctx)
2480 {
2481     u_char          *p, *s;
2482     size_t          len, nlen;
2483     ngx_uint_t      ident;
2484     #if (NGX_HAVE_INET6)
2485     ngx_resolver_t  *r;
2486     #endif
2487     ngx_resolver_qs_t *qs;
2488     ngx_resolver_hdr_t *query;
2489
2490     nlen = ctx->name.len ? (1 + ctx->name.len + 1) : 1;
2491
2492     len = sizeof(ngx_resolver_hdr_t) + nlen + sizeof(ngx_resolver_qs_t);
2493
2494     #if (NGX_HAVE_INET6)
2495     r = ctx->resolver;
2496
2497     p = ngx_resolver_alloc(ctx->resolver, r->ipv6 ? len * 2 : len);
2498     #else
2499     p = ngx_resolver_alloc(ctx->resolver, len);
2500     #endif
2501     if (p == NULL) {
2502         return NGX_ERROR;
2503     }
2504
2505     rn->qlen = (u_short) len;
2506     rn->query = p;
2507
2508     #if (NGX_HAVE_INET6)
2509     if (r->ipv6) {
2510         rn->query6 = p + len;
2511     }
2512     #endif
2513
2514     query = (ngx_resolver_hdr_t *) p;
2515
2516     ident = ngx_random();
2517
2518     ngx_log_debug2(NGX_LOG_DEBUG_CORE, ctx->resolver->log, 0,
2519         "resolve: \"%V\" A %i", &ctx->name, ident & 0xffff);
2520
2521     query->ident_hi = (u_char) ((ident >> 8) & 0xff);
2522     query->ident_lo = (u_char) (ident & 0xff);
2523
2524     /* recursion query */
2525     query->flags_hi = 1; query->flags_lo = 0;
2526
2527     /* one question */
2528     query->nqs_hi = 0; query->nqs_lo = 1;
2529     query->nan_hi = 0; query->nan_lo = 0;
2530     query->nns_hi = 0; query->nns_lo = 0;
2531     query->nar_hi = 0; query->nar_lo = 0;
2532
2533     p += sizeof(ngx_resolver_hdr_t) + nlen;
2534
2535     qs = (ngx_resolver_qs_t *) p;
2536
2537     /* query type */
2538     qs->type_hi = 0; qs->type_lo = NGX_RESOLVE_A;
2539
2540     /* IN query class */
2541     qs->class_hi = 0; qs->class_lo = 1;
2542
2543     /* convert "www.example.com" to "\3www\7example\3com\0" */
2544
2545     len = 0;
2546     p--;
2547     *p-- = '\0';

```

```

2548
2549     if (ctx->name.len == 0) {
2550         return NGX\_DECLINED;
2551     }
2552
2553     for (s = ctx->name.data + ctx->name.len - 1; s >= ctx->name.data; s--) {
2554         if (*s != '.') {
2555             *p = *s;
2556             len++;
2557
2558         } else {
2559             if (len == 0 || len > 255) {
2560                 return NGX\_DECLINED;
2561             }
2562
2563             *p = (u_char) len;
2564             len = 0;
2565         }
2566
2567         p--;
2568     }
2569
2570     if (len == 0 || len > 255) {
2571         return NGX\_DECLINED;
2572     }
2573
2574     *p = (u_char) len;
2575
2576     #if (NGX_HAVE_INET6)
2577     if (!r->ipv6) {
2578         return NGX\_OK;
2579     }
2580
2581     p = rn->query6;
2582
2583     ngx\_memcpy(p, rn->query, rn->qlen);
2584
2585     query = (ngx\_resolver\_hdr\_t *) p;
2586
2587     ident = ngx\_random();
2588
2589     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_CORE, ctx->resolver->log, 0,
2590         "resolve: \"%V\" AAAA %i", &ctx->name, ident & 0xffff);
2591
2592     query->ident_hi = (u_char) ((ident >> 8) & 0xff);
2593     query->ident_lo = (u_char) (ident & 0xff);
2594
2595     p += sizeof(ngx\_resolver\_hdr\_t) + nlen;
2596
2597     qs = (ngx\_resolver\_qs\_t *) p;
2598
2599     qs->type_lo = NGX\_RESOLVE\_AAAA;
2600 #endif
2601
2602     return NGX\_OK;
2603 }
2604
2605
2606 static ngx\_int\_t
2607 ngx\_resolver\_create\_addr\_query(ngx\_resolver\_node\_t *rn, ngx\_resolver\_ctx\_t *ctx)
2608 {
2609     u_char                *p, *d;
2610     size_t                len;
2611     in_addr_t             addr;
2612     ngx\_int\_t              n;
2613     ngx\_uint\_t             ident;
2614     ngx\_resolver\_hdr\_t    *query;
2615     struct sockaddr_in    *sin;
2616     #if (NGX_HAVE_INET6)
2617     struct sockaddr_in6   *sin6;
2618 #endif
2619
2620     switch (ctx->addr.sockaddr->sa_family) {
2621
2622     #if (NGX_HAVE_INET6)
2623     case AF_INET6:

```

```

2624     len = sizeof(ngx_resolver_hdr_t)
2625         + 64 + sizeof(".ip6.arpa.") - 1
2626         + sizeof(ngx_resolver_qs_t);
2627
2628     break;
2629 #endif
2630
2631     default: /* AF_INET */
2632         len = sizeof(ngx_resolver_hdr_t)
2633             + sizeof(".255.255.255.255.in-addr.arpa.") - 1
2634             + sizeof(ngx_resolver_qs_t);
2635     }
2636
2637     p = ngx_resolver_alloc(ctx->resolver, len);
2638     if (p == NULL) {
2639         return NGX_ERROR;
2640     }
2641
2642     rn->query = p;
2643     query = (ngx_resolver_hdr_t *) p;
2644
2645     ident = ngx_random();
2646
2647     query->ident_hi = (u_char) ((ident >> 8) & 0xff);
2648     query->ident_lo = (u_char) (ident & 0xff);
2649
2650     /* recursion query */
2651     query->flags_hi = 1; query->flags_lo = 0;
2652
2653     /* one question */
2654     query->nqs_hi = 0; query->nqs_lo = 1;
2655     query->nan_hi = 0; query->nan_lo = 0;
2656     query->nns_hi = 0; query->nns_lo = 0;
2657     query->nar_hi = 0; query->nar_lo = 0;
2658
2659     p += sizeof(ngx_resolver_hdr_t);
2660
2661     switch (ctx->addr.sockaddr->sa_family) {
2662
2663 #if (NGX_HAVE_INET6)
2664     case AF_INET6:
2665         sin6 = (struct sockaddr_in6 *) ctx->addr.sockaddr;
2666
2667         for (n = 15; n >= 0; n--) {
2668             p = ngx_sprintf(p, "\\1%xd\\1%xd",
2669                             sin6->sin6_addr.s6_addr[n] & 0xf,
2670                             (sin6->sin6_addr.s6_addr[n] >> 4) & 0xf);
2671         }
2672
2673         p = ngx_cpymem(p, "\\3ip6\\4arpa\\0", 10);
2674
2675         break;
2676 #endif
2677
2678     default: /* AF_INET */
2679
2680         sin = (struct sockaddr_in *) ctx->addr.sockaddr;
2681         addr = ntohl(sin->sin_addr.s_addr);
2682
2683         for (n = 0; n < 32; n += 8) {
2684             d = ngx_sprintf(&p[1], "%ud", (addr >> n) & 0xff);
2685             *p = (u_char) (d - &p[1]);
2686             p = d;
2687         }
2688
2689         p = ngx_cpymem(p, "\\7in-addr\\4arpa\\0", 14);
2690     }
2691
2692     /* query type "PTR", IN query class */
2693     p = ngx_cpymem(p, "\\0\\14\\0\\1", 4);
2694
2695     rn->qlen = (u_short) (p - rn->query);
2696
2697     return NGX_OK;
2698 }
2699

```



```

2700 static ngx_int_t
2701 ngx_resolver_copy(ngx_resolver_t *r, ngx_str_t *name, u_char *buf, u_char *src,
2702 u_char *last)
2703 {
2704     char          *err;
2705     u_char        *p, *dst;
2706     ssize_t       len;
2707     ngx_uint_t    i, n;
2708
2709     p = src;
2710     len = -1;
2711
2712     /*
2713      * compression pointers allow to create endless loop, so we set limit;
2714      * 128 pointers should be enough to store 255-byte name
2715      */
2716
2717     for (i = 0; i < 128; i++) {
2718         n = *p++;
2719
2720         if (n == 0) {
2721             goto done;
2722         }
2723
2724         if (n & 0xc0) {
2725             n = ((n & 0x3f) << 8) + *p;
2726             p = &buf[n];
2727         } else {
2728             len += 1 + n;
2729             p = &p[n];
2730         }
2731
2732         if (p >= last) {
2733             err = "name is out of response";
2734             goto invalid;
2735         }
2736     }
2737
2738     err = "compression pointers loop";
2739
2740 invalid:
2741     ngx_log_error(r->log_level, r->log, 0, err);
2742     return NGX_ERROR;
2743
2744 done:
2745     if (name == NULL) {
2746         return NGX_OK;
2747     }
2748
2749     if (len == -1) {
2750         ngx_str_null(name);
2751         return NGX_OK;
2752     }
2753
2754     dst = ngx_resolver_alloc(r, len);
2755     if (dst == NULL) {
2756         return NGX_ERROR;
2757     }
2758
2759     name->data = dst;
2760
2761     n = *src++;
2762
2763     for ( ;; ) {
2764         if (n & 0xc0) {
2765             n = ((n & 0x3f) << 8) + *src;
2766             src = &buf[n];
2767
2768             n = *src++;
2769         } else {

```

```

2776     ngx_strlow(dst, src, n);
2777     dst += n;
2778     src += n;
2779
2780     n = *src++;
2781
2782     if (n != 0) {
2783         *dst++ = '.';
2784     }
2785 }
2786
2787 if (n == 0) {
2788     name->len = dst - name->data;
2789     return NGX_OK;
2790 }
2791 }
2792 }
2793
2794
2795 static void
2796 ngx_resolver_timeout_handler(ngx_event_t *ev)
2797 {
2798     ngx_resolver_ctx_t *ctx, *next;
2799     ngx_resolver_node_t *rn;
2800
2801     rn = ev->data;
2802     ctx = rn->waiting;
2803     rn->waiting = NULL;
2804
2805     do {
2806         ctx->state = NGX_RESOLVE_TIMEDOUT;
2807         next = ctx->next;
2808
2809         ctx->handler(ctx);
2810
2811         ctx = next;
2812     } while (ctx);
2813 }
2814
2815
2816 static void
2817 ngx_resolver_free_node(ngx_resolver_t *r, ngx_resolver_node_t *rn)
2818 {
2819     /* lock alloc mutex */
2820
2821     if (rn->query) {
2822         ngx_resolver_free_locked(r, rn->query);
2823     }
2824
2825     if (rn->name) {
2826         ngx_resolver_free_locked(r, rn->name);
2827     }
2828
2829     if (rn->cnlen) {
2830         ngx_resolver_free_locked(r, rn->u.cname);
2831     }
2832
2833     if (rn->naddrs > 1 && rn->naddrs != (u_short) -1) {
2834         ngx_resolver_free_locked(r, rn->u.addrs);
2835     }
2836
2837 #if (NGX_HAVE_INET6)
2838     if (rn->naddrs6 > 1 && rn->naddrs6 != (u_short) -1) {
2839         ngx_resolver_free_locked(r, rn->u6.addrs6);
2840     }
2841 #endif
2842
2843     ngx_resolver_free_locked(r, rn);
2844
2845     /* unlock alloc mutex */
2846 }
2847
2848
2849 static void *
2850 ngx_resolver_alloc(ngx_resolver_t *r, size_t size)
2851 {

```

```

2852     u_char  *p;
2853
2854     /* lock alloc mutex */
2855
2856     p = ngx_alloc(size, r->log);
2857
2858     /* unlock alloc mutex */
2859
2860     return p;
2861 }
2862
2863
2864 static void *
2865 ngx_resolver_calloc(ngx_resolver_t *r, size_t size)
2866 {
2867     u_char  *p;
2868
2869     p = ngx_resolver_alloc(r, size);
2870
2871     if (p) {
2872         ngx_memzero(p, size);
2873     }
2874
2875     return p;
2876 }
2877
2878
2879 static void
2880 ngx_resolver_free(ngx_resolver_t *r, void *p)
2881 {
2882     /* lock alloc mutex */
2883
2884     ngx_free(p);
2885
2886     /* unlock alloc mutex */
2887 }
2888
2889
2890 static void
2891 ngx_resolver_free_locked(ngx_resolver_t *r, void *p)
2892 {
2893     ngx_free(p);
2894 }
2895
2896
2897 static void *
2898 ngx_resolver_dup(ngx_resolver_t *r, void *src, size_t size)
2899 {
2900     void  *dst;
2901
2902     dst = ngx_resolver_alloc(r, size);
2903
2904     if (dst == NULL) {
2905         return dst;
2906     }
2907
2908     ngx_memcpy(dst, src, size);
2909
2910     return dst;
2911 }
2912
2913
2914 static ngx_addr_t *
2915 ngx_resolver_export(ngx_resolver_t *r, ngx_resolver_node_t *rn,
2916                    ngx_uint_t rotate)
2917 {
2918     ngx_addr_t      *dst;
2919     ngx_uint_t      d, i, j, n;
2920     u_char          (*sockaddr)[NGX_SOCKADDRLEN];
2921     in_addr_t       *addr;
2922     struct sockaddr_in *sin;
2923     #if (NGX_HAVE_INET6)
2924     struct in6_addr  *addr6;
2925     struct sockaddr_in6 *sin6;
2926     #endif
2927

```

```

2928     n = rn->naddrs;
2929     #if (NGX_HAVE_INET6)
2930     n += rn->naddrs6;
2931     #endif
2932
2933     dst = ngx_resolver_calloc(r, n * sizeof(ngx_addr_t));
2934     if (dst == NULL) {
2935         return NULL;
2936     }
2937
2938     sockaddr = ngx_resolver_calloc(r, n * NGX_SOCKADDRLEN);
2939     if (sockaddr == NULL) {
2940         ngx_resolver_free(r, dst);
2941         return NULL;
2942     }
2943
2944     i = 0;
2945     d = rotate ? ngx_random() % n : 0;
2946
2947     if (rn->naddrs) {
2948         j = rotate ? ngx_random() % rn->naddrs : 0;
2949
2950         addr = (rn->naddrs == 1) ? &rn->u.addr : rn->u.addrs;
2951
2952         do {
2953             sin = (struct sockaddr_in *) sockaddr[d];
2954             sin->sin_family = AF_INET;
2955             sin->sin_addr.s_addr = addr[j++];
2956             dst[d].sockaddr = (struct sockaddr *) sin;
2957             dst[d++].socklen = sizeof(struct sockaddr_in);
2958
2959             if (d == n) {
2960                 d = 0;
2961             }
2962
2963             if (j == rn->naddrs) {
2964                 j = 0;
2965             }
2966         } while (++i < rn->naddrs);
2967     }
2968
2969     #if (NGX_HAVE_INET6)
2970     if (rn->naddrs6) {
2971         j = rotate ? ngx_random() % rn->naddrs6 : 0;
2972
2973         addr6 = (rn->naddrs6 == 1) ? &rn->u6.addr6 : rn->u6.addrs6;
2974
2975         do {
2976             sin6 = (struct sockaddr_in6 *) sockaddr[d];
2977             sin6->sin6_family = AF_INET6;
2978             ngx_memcpy(sin6->sin6_addr.s6_addr, addr6[j++].s6_addr, 16);
2979             dst[d].sockaddr = (struct sockaddr *) sin6;
2980             dst[d++].socklen = sizeof(struct sockaddr_in6);
2981
2982             if (d == n) {
2983                 d = 0;
2984             }
2985
2986             if (j == rn->naddrs6) {
2987                 j = 0;
2988             }
2989         } while (++i < n);
2990     }
2991     #endif
2992
2993     return dst;
2994 }
2995
2996 char *
2997 ngx_resolver_strerror(ngx_int_t err)
2998 {
2999     static char *errors[] = {
3000         "Format error",      /* FORMERR */
3001         "Server failure",   /* SERVFAIL */
3002         "Host not found",   /* NXDOMAIN */

```

```

3004     "Unimplemented", /* NOTIMP */
3005     "Operation refused" /* REFUSED */
3006 };
3007
3008 if (err > 0 && err < 6) {
3009     return errors[err - 1];
3010 }
3011
3012 if (err == NGX_RESOLVE_TIMEDOUT) {
3013     return "Operation timed out";
3014 }
3015
3016 return "Unknown error";
3017 }
3018
3019
3020 static u_char *
3021 ngx_resolver_log_error(ngx_log_t *log, u_char *buf, size_t len)
3022 {
3023     u_char          *p;
3024     ngx_udp_connection_t *uc;
3025
3026     p = buf;
3027
3028     if (log->action) {
3029         p = ngx_snprintf(buf, len, " while %s", log->action);
3030         len -= p - buf;
3031     }
3032
3033     uc = log->data;
3034
3035     if (uc) {
3036         p = ngx_snprintf(p, len, ", resolver: %V", &uc->server);
3037     }
3038
3039     return p;
3040 }
3041
3042
3043 ngx_int_t
3044 ngx_udp_connect(ngx_udp_connection_t *uc)
3045 {
3046     int          rc;
3047     ngx_int_t    event;
3048     ngx_event_t *rev, *wev;
3049     ngx_socket_t s;
3050     ngx_connection_t *c;
3051
3052     s = ngx_socket(uc->sockaddr->sa_family, SOCK_DGRAM, 0);
3053
3054     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, &uc->log, 0, "UDP socket %d", s);
3055
3056     if (s == (ngx_socket_t) -1) {
3057         ngx_log_error(NGX_LOG_ALERT, &uc->log, ngx_socket_errno,
3058                     ngx_socket_n " failed");
3059         return NGX_ERROR;
3060     }
3061
3062     c = ngx_get_connection(s, &uc->log);
3063
3064     if (c == NULL) {
3065         if (ngx_close_socket(s) == -1) {
3066             ngx_log_error(NGX_LOG_ALERT, &uc->log, ngx_socket_errno,
3067                         ngx_close_socket_n " failed");
3068         }
3069
3070         return NGX_ERROR;
3071     }
3072
3073     if (ngx_nonblocking(s) == -1) {
3074         ngx_log_error(NGX_LOG_ALERT, &uc->log, ngx_socket_errno,
3075                     ngx_nonblocking_n " failed");
3076
3077         goto failed;
3078     }
3079

```

```

3080     rev = c->read;
3081     wev = c->write;
3082
3083     rev->log = &uc->log;
3084     wev->log = &uc->log;
3085
3086     uc->connection = c;
3087
3088     c->number = ngx\_atomic\_fetch\_add(ngx\_connection\_counter, 1);
3089
3090     ngx\_log\_debug3(NGX\_LOG\_DEBUG\_EVENT, &uc->log, 0,
3091         "connect to %V, fd:%d #%uA", &uc->server, s, c->number);
3092
3093     rc = connect(s, uc->sockaddr, uc->socklen);
3094
3095     /* TODO: aio, iocp */
3096
3097     if (rc == -1) {
3098         ngx\_log\_error(NGX\_LOG\_CRIT, &uc->log, ngx\_socket\_errno,
3099             "connect() failed");
3100
3101         goto failed;
3102     }
3103
3104     /* UDP sockets are always ready to write */
3105     wev->ready = 1;
3106
3107     if (ngx\_add\_event) {
3108
3109         event = (ngx\_event\_flags & NGX\_USE\_CLEAR\_EVENT) ?
3110             /* kqueue, epoll */ NGX\_CLEAR\_EVENT:
3111             /* select, poll, /dev/poll */ NGX\_LEVEL\_EVENT;
3112         /* eventport event type has no meaning: oneshot only */
3113
3114         if (ngx\_add\_event(rev, NGX\_READ\_EVENT, event) != NGX\_OK) {
3115             goto failed;
3116         }
3117
3118     } else {
3119         /* rtsig */
3120
3121         if (ngx\_add\_conn(c) == NGX\_ERROR) {
3122             goto failed;
3123         }
3124     }
3125
3126     return NGX\_OK;
3127
3128 failed:
3129
3130     ngx\_close\_connection(c);
3131     uc->connection = NULL;
3132
3133     return NGX\_ERROR;
3134 }

```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_parse.c - nginx-1.7.10

### Functions defined

- [ngx\\_parse\\_offset](#)
- [ngx\\_parse\\_size](#)
- [ngx\\_parse\\_time](#)

### Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10
11
12  ssize_t
13  ngx_parse_size(ngx_str_t *line)
14  {
15      u_char    unit;
16      size_t    len;
17      ssize_t   size;
18      ngx_int_t scale;
19
20      len = line->len;
21      unit = line->data[len - 1];
22
23      switch (unit) {
24      case 'K':
25      case 'k':
26          len--;
27          scale = 1024;
28          break;
29
30      case 'M':
31      case 'm':
32          len--;
33          scale = 1024 * 1024;
34          break;
35
36      default:
37          scale = 1;
38      }
39
40      size = ngx_atosz(line->data, len);
41      if (size == NGX_ERROR) {
42          return NGX_ERROR;
43      }
44
45      size *= scale;
46
47      return size;
48  }
49
50
51  off_t
52  ngx_parse_offset(ngx_str_t *line)
53  {
54      u_char    unit;
55      off_t     offset;
56      size_t    len;
57      ngx_int_t scale;
58
```

```

59     len = line->len;
60     unit = line->data[len - 1];
61
62     switch (unit) {
63     case 'K':
64     case 'k':
65         len--;
66         scale = 1024;
67         break;
68
69     case 'M':
70     case 'm':
71         len--;
72         scale = 1024 * 1024;
73         break;
74
75     case 'G':
76     case 'g':
77         len--;
78         scale = 1024 * 1024 * 1024;
79         break;
80
81     default:
82         scale = 1;
83     }
84
85     offset = ngx_atoof(line->data, len);
86     if (offset == NGX_ERROR) {
87         return NGX_ERROR;
88     }
89
90     offset *= scale;
91
92     return offset;
93 }
94
95
96 ngx_int_t
97 ngx_parse_time(ngx_str_t *line, ngx_uint_t is_sec)
98 {
99     u_char      *p, *last;
100     ngx_int_t   value, total, scale;
101     ngx_uint_t  max, valid;
102     enum {
103         st_start = 0,
104         st_year,
105         st_month,
106         st_week,
107         st_day,
108         st_hour,
109         st_min,
110         st_sec,
111         st_msec,
112         st_last
113     } step;
114
115     valid = 0;
116     value = 0;
117     total = 0;
118     step = is_sec ? st_start : st_month;
119     scale = is_sec ? 1 : 1000;
120
121     p = line->data;
122     last = p + line->len;
123
124     while (p < last) {
125
126         if (*p >= '0' && *p <= '9') {
127             value = value * 10 + (*p++ - '0');
128             valid = 1;
129             continue;
130         }
131
132         switch (*p++) {
133
134         case 'y':

```



```

135     if (step > st_start) {
136         return NGX\_ERROR;
137     }
138     step = st_year;
139     max = NGX\_MAX\_INT32\_VALUE / (60 * 60 * 24 * 365);
140     scale = 60 * 60 * 24 * 365;
141     break;
142
143 case 'M':
144     if (step >= st_month) {
145         return NGX\_ERROR;
146     }
147     step = st_month;
148     max = NGX\_MAX\_INT32\_VALUE / (60 * 60 * 24 * 30);
149     scale = 60 * 60 * 24 * 30;
150     break;
151
152 case 'w':
153     if (step >= st_week) {
154         return NGX\_ERROR;
155     }
156     step = st_week;
157     max = NGX\_MAX\_INT32\_VALUE / (60 * 60 * 24 * 7);
158     scale = 60 * 60 * 24 * 7;
159     break;
160
161 case 'd':
162     if (step >= st_day) {
163         return NGX\_ERROR;
164     }
165     step = st_day;
166     max = NGX\_MAX\_INT32\_VALUE / (60 * 60 * 24);
167     scale = 60 * 60 * 24;
168     break;
169
170 case 'h':
171     if (step >= st_hour) {
172         return NGX\_ERROR;
173     }
174     step = st_hour;
175     max = NGX\_MAX\_INT32\_VALUE / (60 * 60);
176     scale = 60 * 60;
177     break;
178
179 case 'm':
180     if (*p == 's') {
181         if (is_sec || step >= st_msec) {
182             return NGX\_ERROR;
183         }
184         p++;
185         step = st_msec;
186         max = NGX\_MAX\_INT32\_VALUE;
187         scale = 1;
188         break;
189     }
190
191     if (step >= st_min) {
192         return NGX\_ERROR;
193     }
194     step = st_min;
195     max = NGX\_MAX\_INT32\_VALUE / 60;
196     scale = 60;
197     break;
198
199 case 's':
200     if (step >= st_sec) {
201         return NGX\_ERROR;
202     }
203     step = st_sec;
204     max = NGX\_MAX\_INT32\_VALUE;
205     scale = 1;
206     break;
207
208 case ' ':
209     if (step >= st_sec) {
210         return NGX\_ERROR;

```

```
211     }
212     step = st_last;
213     max = NGX\_MAX\_INT32\_VALUE;
214     scale = 1;
215     break;
216
217     default:
218         return NGX\_ERROR;
219     }
220
221     if (step != st_msec && !is_sec) {
222         scale *= 1000;
223         max /= 1000;
224     }
225
226     if ((ngx\_uint\_t) value > max) {
227         return NGX\_ERROR;
228     }
229
230     total += value * scale;
231
232     if ((ngx\_uint\_t) total > NGX\_MAX\_INT32\_VALUE) {
233         return NGX\_ERROR;
234     }
235
236     value = 0;
237     scale = is_sec ? 1 : 1000;
238
239     while (p < last && *p == ' ') {
240         p++;
241     }
242 }
243
244 if (valid) {
245     return total + value * scale;
246 }
247
248 return NGX\_ERROR;
249 }
```

[One Level Up](#)

[Top Level](#)

# src/event/nginx\_event\_connect.h - nginx-1.7.10

## Data types defined

- [ngx\\_event\\_free\\_peer\\_pt](#)
- [ngx\\_event\\_get\\_peer\\_pt](#)
- [ngx\\_event\\_save\\_peer\\_session\\_pt](#)
- [ngx\\_event\\_set\\_peer\\_session\\_pt](#)
- [ngx\\_peer\\_connection\\_s](#)
- [ngx\\_peer\\_connection\\_t](#)

## Macros defined

- [NGX\\_PEER\\_FAILED](#)
- [NGX\\_PEER\\_KEEPALIVE](#)
- [NGX\\_PEER\\_NEXT](#)
- [\\_NGX\\_EVENT\\_CONNECT\\_H\\_INCLUDED\\_](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_EVENT_CONNECT_H_INCLUDED_
9 #define _NGX_EVENT_CONNECT_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14 #include <ngx_event.h>
15
16
17 #define NGX_PEER_KEEPALIVE          1
18 #define NGX_PEER_NEXT               2
19 #define NGX_PEER_FAILED             4
20
21
22 typedef struct ngx_peer_connection_s ngx_peer_connection_t;
23
24 typedef ngx_int_t (*ngx_event_get_peer_pt)(ngx_peer_connection_t *pc,
25     void *data);
26 typedef void (*ngx_event_free_peer_pt)(ngx_peer_connection_t *pc, void *data,
27     ngx_uint_t state);
28 #if (NGX_SSL)
29
30 typedef ngx_int_t (*ngx_event_set_peer_session_pt)(ngx_peer_connection_t *pc,
31     void *data);
32 typedef void (*ngx_event_save_peer_session_pt)(ngx_peer_connection_t *pc,
33     void *data);
34 #endif
35
36
37 struct ngx_peer_connection_s {
38     ngx_connection_t *connection;
```

```

39     struct sockaddr                *sockaddr;
40     socklen_t                     socklen;
41     ngx_str_t                     *name;
42
43
44     ngx_uint_t                    tries;
45     ngx_msec_t                    start_time;
46
47     ngx_event_get_peer_pt         get;
48     ngx_event_free_peer_pt        free;
49     void                          *data;
50
51 #if (NGX_SSL)
52     ngx_event_set_peer_session_pt  set_session;
53     ngx_event_save_peer_session_pt save_session;
54 #endif
55
56 #if (NGX_THREADS)
57     ngx_atomic_t                  *lock;
58 #endif
59
60     ngx_addr_t                   *local;
61
62     int                           rcvbuf;
63
64     ngx_log_t                     *log;
65
66     unsigned                       cached:1;
67
68     unsigned                       /* ngx_connection_log_error_e */
69     log_error:2;
70 };
71
72
73 ngx_int_t ngx_event_connect_peer(ngx_peer_connection_t *pc);
74 ngx_int_t ngx_event_get_peer(ngx_peer_connection_t *pc, void *data);
75
76
77 #endif /* NGX_EVENT_CONNECT_H_INCLUDED */

```

[One Level Up](#)

[Top Level](#)

## src/event/nginx\_event\_connect.c - nginx-1.7.10

### Functions defined

- [ngx\\_event\\_connect\\_peer](#)
- [ngx\\_event\\_get\\_peer](#)

### Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_event.h>
11 #include <ngx_event_connect.h>
12
13
14 ngx_int_t
15 ngx_event_connect_peer(ngx_peer_connection_t *pc)
16 {
17     int rc;
18     ngx_int_t event;
19     ngx_err_t err;
20     ngx_uint_t level;
21     ngx_socket_t s;
22     ngx_event_t *rev, *wev;
23     ngx_connection_t *c;
24
25     rc = pc->get(pc, pc->data);
26     if (rc != NGX_OK) {
27         return rc;
28     }
29
30     s = ngx_socket(pc->sockaddr->sa_family, SOCK_STREAM, 0);
31
32     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, pc->log, 0, "socket %d", s);
33
34     if (s == (ngx_socket_t) -1) {
35         ngx_log_error(NGX_LOG_ALERT, pc->log, ngx_socket_errno,
36                     ngx_socket_n " failed");
37         return NGX_ERROR;
38     }
39
40
41     c = ngx_get_connection(s, pc->log);
42
43     if (c == NULL) {
44         if (ngx_close_socket(s) == -1) {
45             ngx_log_error(NGX_LOG_ALERT, pc->log, ngx_socket_errno,
46                         ngx_close_socket_n "failed");
47         }
48
49         return NGX_ERROR;
50     }
51
52     if (pc->rcvbuf) {
53         if (setsockopt(s, SOL_SOCKET, SO_RCVBUF,
54                     (const void *) &pc->rcvbuf, sizeof(int)) == -1)
55         {
56             ngx_log_error(NGX_LOG_ALERT, pc->log, ngx_socket_errno,
57                         "setsockopt(SO_RCVBUF) failed");
58             goto failed;
59         }
60     }
```

```

61
62 if (ngx\_nonblocking(s) == -1) {
63     ngx\_log\_error(NGX_LOG_ALERT, pc->log, ngx\_socket\_errno,
64         ngx\_nonblocking\_n " failed");
65
66     goto failed;
67 }
68
69 if (pc->local) {
70     if (bind(s, pc->local->sockaddr, pc->local->socklen) == -1) {
71         ngx\_log\_error(NGX_LOG_CRIT, pc->log, ngx\_socket\_errno,
72             "bind(%V) failed", &pc->local->name);
73
74         goto failed;
75     }
76 }
77
78 c->recv = ngx\_recv;
79 c->send = ngx\_send;
80 c->recv_chain = ngx\_recv\_chain;
81 c->send_chain = ngx\_send\_chain;
82
83 c->sendfile = 1;
84
85 c->log_error = pc->log_error;
86
87 if (pc->sockaddr->sa_family == AF_UNIX) {
88     c->tcp_nopush = NGX_TCP_NOPUSH_DISABLED;
89     c->tcp_nodelay = NGX_TCP_NODELAY_DISABLED;
90
91 #if (NGX_SOLARIS)
92     /* Solaris's sendfilev\(\) supports AF_NCA, AF_INET, and AF_INET6 */
93     c->sendfile = 0;
94 #endif
95 }
96
97 rev = c->read;
98 wev = c->write;
99
100 rev->log = pc->log;
101 wev->log = pc->log;
102
103 pc->connection = c;
104
105 c->number = ngx\_atomic\_fetch\_add(ngx\_connection\_counter, 1);
106
107 if (ngx\_add\_conn) {
108     if (ngx\_add\_conn(c) == NGX_ERROR) {
109         goto failed;
110     }
111 }
112
113 ngx\_log\_debug3(NGX_LOG_DEBUG_EVENT, pc->log, 0,
114     "connect to %V, fd:%d #%uA", pc->name, s, c->number);
115
116 rc = connect(s, pc->sockaddr, pc->socklen);
117
118 if (rc == -1) {
119     err = ngx\_socket\_errno;
120
121     if (err != NGX_EINPROGRESS
122 #if (NGX_WIN32)
123         /* Winsock returns WSAEWOULDBLOCK (NGX\_EAGAIN) */
124         && err != NGX\_EAGAIN
125 #endif
126     )
127     {
128         if (err == NGX\_ECONNREFUSED
129 #if (NGX_LINUX)
130         /*
131          * Linux returns EAGAIN instead of ECONNREFUSED
132          * for unix sockets if listen queue is full
133          */
134         || err == NGX\_EAGAIN
135 #endif
136     )

```

```

137     || err == NGX\_ECONNRESET
138     || err == NGX\_ENETDOWN
139     || err == NGX\_ENETUNREACH
140     || err == NGX\_EHOSTDOWN
141     || err == NGX\_EHOSTUNREACH)
142     {
143         level = NGX\_LOG\_ERR;
144
145     } else {
146         level = NGX\_LOG\_CRIT;
147     }
148
149     ngx\_log\_error(level, c->log, err, "connect() to %V failed",
150                 pc->name);
151
152     ngx\_close\_connection(c);
153     pc->connection = NULL;
154
155     return NGX\_DECLINED;
156 }
157 }
158
159 if (ngx\_add\_conn) {
160     if (rc == -1) {
161
162         /* NGX\_EINPROGRESS */
163
164         return NGX\_AGAIN;
165     }
166
167     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_EVENT, pc->log, 0, "connected");
168
169     wev->ready = 1;
170
171     return NGX\_OK;
172 }
173
174 if (ngx\_event\_flags & NGX\_USE\_AIO\_EVENT) {
175
176     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_EVENT, pc->log, ngx\_socket\_errno,
177                 "connect(): %d", rc);
178
179     /* aio, iocp */
180
181     if (ngx\_blocking(s) == -1) {
182         ngx\_log\_error(NGX\_LOG\_ALERT, pc->log, ngx\_socket\_errno,
183                 ngx\_blocking\_n " failed");
184         goto failed;
185     }
186
187     /*
188      * FreeBSD's aio allows to post an operation on non-connected socket.
189      * NT does not support it.
190      *
191      * TODO: check in Win32, etc. As workaround we can use NGX\_ONESHOT\_EVENT
192      */
193
194     rev->ready = 1;
195     wev->ready = 1;
196
197     return NGX\_OK;
198 }
199
200 if (ngx\_event\_flags & NGX\_USE\_CLEAR\_EVENT) {
201
202     /* kqueue */
203
204     event = NGX\_CLEAR\_EVENT;
205
206 } else {
207
208     /* select, poll, /dev/poll */
209
210     event = NGX\_LEVEL\_EVENT;
211 }
212

```

```
213     if (ngx_add_event(rev, NGX\_READ\_EVENT, event) != NGX\_OK) {
214         goto failed;
215     }
216
217     if (rc == -1) {
218         /* NGX\_EINPROGRESS */
219
220         if (ngx_add_event(wev, NGX\_WRITE\_EVENT, event) != NGX\_OK) {
221             goto failed;
222         }
223
224         return NGX\_AGAIN;
225     }
226
227     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_EVENT, pc->log, 0, "connected");
228
229     wev->ready = 1;
230
231     return NGX\_OK;
232
233 failed:
234
235     ngx\_close\_connection(c);
236     pc->connection = NULL;
237
238     return NGX\_ERROR;
239 }
240
241
242
243 ngx\_int\_t
244 ngx\_event\_get\_peer(ngx\_peer\_connection\_t *pc, void *data)
245 {
246     return NGX\_OK;
247 }
```

[One Level Up](#)

[Top Level](#)



## src/os/unix/nginx\_solaris\_sendfilev\_chain.c - nginx-1.7.10

### Data types defined

- [sendfilevec](#)
- [sendfilevec\\_t](#)

### Functions defined

- [ngx\\_solaris\\_sendfilev\\_chain](#)
- [sendfilev](#)

### Macros defined

- [NGX\\_SENDFILEVECS](#)
- [SFV\\_FD\\_SELF](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 #if (NGX_TEST_BUILD_SOLARIS_SENDFILEV)
14
15 /* Solaris declarations */
16
17 typedef struct sendfilevec {
18     int     sfv_fd;
19     u_int   sfv_flag;
20     off_t   sfv_off;
21     size_t  sfv_len;
22 } sendfilevec_t;
23
24 #define SFV_FD_SELF  -2
25
26 static ssize_t sendfilev(int fd, const struct sendfilevec *vec,
27     int sfvcnt, size_t *xferred)
28 {
29     return -1;
30 }
31
32 ngx_chain_t *ngx_solaris_sendfilev_chain(ngx_connection_t *c, ngx_chain_t *in,
33     off_t limit);
34
35 #endif
36
37
38 #define NGX_SENDFILEVECS  NGX_IOVS_PREALLOCATE
39
40
41 ngx_chain_t *
42 ngx_solaris_sendfilev_chain(ngx_connection_t *c, ngx_chain_t *in, off_t limit)
43 {
44     int     fd;
```

```

45     u_char          *prev;
46     off_t           size, send, prev_send, aligned, fprev;
47     size_t          sent;
48     ssize_t         n;
49     ngx_int_t       eintr;
50     ngx_err_t       err;
51     ngx_uint_t      nsfv;
52     sendfilevec_t   *sfv, sfvs[NGX_SENDFILEVECS];
53     ngx_event_t     *wev;
54     ngx_chain_t     *cl;
55
56     wev = c->write;
57
58     if (!wev->ready) {
59         return in;
60     }
61
62     if (!c->sendfile) {
63         return ngx_writev_chain(c, in, limit);
64     }
65
66
67     /* the maximum limit size is the maximum size_t value - the page size */
68
69     if (limit == 0 || limit > (off_t) (NGX_MAX_SIZE_T_VALUE - ngx_pagesize)) {
70         limit = NGX_MAX_SIZE_T_VALUE - ngx_pagesize;
71     }
72
73
74     send = 0;
75
76     for ( ;; ) {
77         fd = SFV_FD_SELF;
78         prev = NULL;
79         fprev = 0;
80         sfv = NULL;
81         eintr = 0;
82         sent = 0;
83         prev_send = send;
84
85         nsfv = 0;
86
87         /* create the sendfilevec and coalesce the neighbouring bufs */
88
89         for (cl = in; cl && send < limit; cl = cl->next) {
90
91             if (ngx_buf_special(cl->buf)) {
92                 continue;
93             }
94
95             if (ngx_buf_in_memory_only(cl->buf)) {
96                 fd = SFV_FD_SELF;
97
98                 size = cl->buf->last - cl->buf->pos;
99
100                if (send + size > limit) {
101                    size = limit - send;
102                }
103
104                if (prev == cl->buf->pos) {
105                    sfv->sfv_len += (size_t) size;
106                } else {
107                    if (nsfv == NGX_SENDFILEVECS) {
108                        break;
109                    }
110                }
111
112                sfv = &sfvs[nsfv++];
113
114                sfv->sfv_fd = SFV_FD_SELF;
115                sfv->sfv_flag = 0;
116                sfv->sfv_off = (off_t) (uintptr_t) cl->buf->pos;
117                sfv->sfv_len = (size_t) size;
118            }
119
120            prev = cl->buf->pos + (size_t) size;

```

```

121         send += size;
122
123     } else {
124         prev = NULL;
125
126         size = cl->buf->file_last - cl->buf->file_pos;
127
128         if (send + size > limit) {
129             size = limit - send;
130
131             aligned = (cl->buf->file_pos + size + ngx\_pagesize - 1)
132                 & ~((off\_t) ngx\_pagesize - 1);
133
134             if (aligned <= cl->buf->file_last) {
135                 size = aligned - cl->buf->file_pos;
136             }
137         }
138
139         if (fd == cl->buf->file->fd && fprev == cl->buf->file_pos) {
140             sfv->sfv_len += (size\_t) size;
141
142         } else {
143             if (nsfv == NGX\_SENDFILEVECS) {
144                 break;
145             }
146
147             sfv = &sfvs[nsfv++];
148
149             fd = cl->buf->file->fd;
150             sfv->sfv_fd = fd;
151             sfv->sfv_flag = 0;
152             sfv->sfv_off = cl->buf->file_pos;
153             sfv->sfv_len = (size\_t) size;
154         }
155
156         fprev = cl->buf->file_pos + size;
157         send += size;
158     }
159 }
160
161 n = sendfilev(c->fd, sfvs, nsfv, &sent);
162
163 if (n == -1) {
164     err = ngx\_errno;
165
166     switch (err) {
167     case NGX\_EAGAIN:
168         break;
169
170     case NGX\_EINTR:
171         eintr = 1;
172         break;
173
174     default:
175         wev->error = 1;
176         ngx\_connection\_error(c, err, "sendfilev() failed");
177         return NGX\_CHAIN\_ERROR;
178     }
179
180     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_EVENT, c->log, err,
181         "sendfilev() sent only %uz bytes", sent);
182 }
183
184 ngx\_log\_debug2(NGX\_LOG\_DEBUG\_EVENT, c->log, 0,
185     "sendfilev: %z %z", n, sent);
186
187 c->sent += sent;
188
189 in = ngx\_chain\_update\_sent(in, sent);
190
191 if (eintr) {
192     send = prev_send + sent;
193     continue;
194 }
195
196 if (send - prev_send != (off\_t) sent) {

```

```
197         wev->ready = 0;
198         return in;
199     }
200
201     if (send >= limit || in == NULL) {
202         return in;
203     }
204 }
205 }
```

[One Level Up](#)

[Top Level](#)

# src/os/unix/nginx\_writev\_chain.c - nginx-1.7.10

## Functions defined

- [ngx\\_output\\_chain\\_to\\_iovec](#)
- [ngx\\_writev](#)
- [ngx\\_writev\\_chain](#)

## Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 ngx_chain_t *
14 ngx_writev_chain(ngx_connection_t *c, ngx_chain_t *in, off_t limit)
15 {
16     ssize_t      n, sent;
17     off_t        send, prev_send;
18     ngx_chain_t  *cl;
19     ngx_event_t  *wev;
20     ngx_iovec_t  vec;
21     struct iovec  iovs[NGX_IOVS_PREALLOCATE];
22
23     wev = c->write;
24
25     if (!wev->ready) {
26         return in;
27     }
28
29     #if (NGX_HAVE_KQUEUE)
30
31     if ((ngx_event_flags & NGX_USE_KQUEUE_EVENT) && wev->pending_eof) {
32         (void) ngx_connection_error(c, wev->kq_errno,
33             "kevent() reported about an closed connection");
34         wev->error = 1;
35         return NGX_CHAIN_ERROR;
36     }
37
38     #endif
39
40     /* the maximum limit size is the maximum size_t value - the page size */
41
42     if (limit == 0 || limit > (off_t) (NGX_MAX_SIZE_T_VALUE - ngx_pagesize)) {
43         limit = NGX_MAX_SIZE_T_VALUE - ngx_pagesize;
44     }
45
46     send = 0;
47
48     vec.iovs = iovs;
49     vec.nalloc = NGX_IOVS_PREALLOCATE;
50
51     for ( ;; ) {
52         prev_send = send;
53
54         /* create the iovec and coalesce the neighbouring bufs */
55
56         cl = ngx_output_chain_to_iovec(&vec, in, limit - send, c->log);
57
58         if (cl == NGX_CHAIN_ERROR) {
```



```

135         in->buf->temporary,
136         in->buf->recycled,
137         in->buf->in_file,
138         in->buf->start,
139         in->buf->pos,
140         in->buf->last,
141         in->buf->file,
142         in->buf->file_pos,
143         in->buf->file_last);
144
145         ngx\_debug\_point();
146
147         return NGX\_CHAIN\_ERROR;
148     }
149
150     size = in->buf->last - in->buf->pos;
151
152     if (size > limit - total) {
153         size = limit - total;
154     }
155
156     if (prev == in->buf->pos) {
157         iov->iov_len += size;
158
159     } else {
160         if (n == vec->nalloc) {
161             break;
162         }
163
164         iov = &vec->iovs[n++];
165
166         iov->iov_base = (void *) in->buf->pos;
167         iov->iov_len = size;
168     }
169
170     prev = in->buf->pos + size;
171     total += size;
172 }
173
174 vec->count = n;
175 vec->size = total;
176
177 return in;
178 }
179
180
181 ssize_t
182 ngx\_writew(ngx\_connection\_t *c, ngx\_iovec\_t *vec)
183 {
184     ssize_t    n;
185     ngx\_err\_t  err;
186
187     eintr:
188
189     n = writew(c->fd, vec->iovs, vec->count);
190
191     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_EVENT, c->log, 0,
192                  "writew: %z of %uz", n, vec->size);
193
194     if (n == -1) {
195         err = ngx\_errno;
196
197         switch (err) {
198             case NGX\_EAGAIN:
199                 ngx\_log\_debug0(NGX\_LOG\_DEBUG\_EVENT, c->log, err,
200                               "writew() not ready");
201                 return NGX\_AGAIN;
202
203             case NGX\_EINTR:
204                 ngx\_log\_debug0(NGX\_LOG\_DEBUG\_EVENT, c->log, err,
205                               "writew() was interrupted");
206                 goto eintr;
207
208             default:
209                 c->write->error = 1;
210                 ngx\_connection\_error(c, err, "writew() failed");

```

```
211         return NGX\_ERROR;  
212     }  
213 }  
214  
215 return n;  
216 }
```

[One Level Up](#)

[Top Level](#)



## src/http/modules/nginx\_http\_ssl\_module.c - nginx-1.7.10

### Global variables defined

- [ngx\\_http\\_ssl\\_commands](#)
- [ngx\\_http\\_ssl\\_module](#)
- [ngx\\_http\\_ssl\\_module\\_ctx](#)
- [ngx\\_http\\_ssl\\_protocols](#)
- [ngx\\_http\\_ssl\\_sess\\_id\\_ctx](#)
- [ngx\\_http\\_ssl\\_vars](#)
- [ngx\\_http\\_ssl\\_verify](#)

### Data types defined

- [ngx\\_ssl\\_variable\\_handler\\_pt](#)

### Functions defined

- [ngx\\_http\\_ssl\\_add\\_variables](#)
- [ngx\\_http\\_ssl\\_alpn\\_select](#)
- [ngx\\_http\\_ssl\\_create\\_srv\\_conf](#)
- [ngx\\_http\\_ssl\\_enable](#)
- [ngx\\_http\\_ssl\\_init](#)
- [ngx\\_http\\_ssl\\_merge\\_srv\\_conf](#)
- [ngx\\_http\\_ssl\\_npn\\_advertised](#)
- [ngx\\_http\\_ssl\\_password\\_file](#)
- [ngx\\_http\\_ssl\\_session\\_cache](#)
- [ngx\\_http\\_ssl\\_static\\_variable](#)
- [ngx\\_http\\_ssl\\_variable](#)

### Macros defined

- [NGX\\_DEFAULT\\_CIPHERS](#)
- [NGX\\_DEFAULT\\_ECDH\\_CURVE](#)
- [NGX\\_HTTP\\_NPN\\_ADVERTISE](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
```

```

5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 typedef ngx_int_t (*ngx_ssl_variable_handler_pt)(ngx_connection_t *c,
14     ngx_pool_t *pool, ngx_str_t *s);
15
16
17 #define NGX_DEFAULT_CIPHERS    "HIGH:!aNULL:!MD5"
18 #define NGX_DEFAULT_ECDH_CURVE "prime256v1"
19
20 #define NGX_HTTP_NPN_ADVERTISE "\x08http/1.1"
21
22
23 #ifdef TLSEXT_TYPE_application_layer_protocol_negotiation
24 static int ngx_http_ssl_alpn_select(ngx_ssl_conn_t *ssl_conn,
25     const unsigned char **out, unsigned char *outlen,
26     const unsigned char *in, unsigned int inlen, void *arg);
27 #endif
28
29 #ifdef TLSEXT_TYPE_next_proto_neg
30 static int ngx_http_ssl_npn_advertised(ngx_ssl_conn_t *ssl_conn,
31     const unsigned char **out, unsigned int *outlen, void *arg);
32 #endif
33
34 static ngx_int_t ngx_http_ssl_static_variable(ngx_http_request_t *r,
35     ngx_http_variable_value_t *v, uintptr_t data);
36 static ngx_int_t ngx_http_ssl_variable(ngx_http_request_t *r,
37     ngx_http_variable_value_t *v, uintptr_t data);
38
39 static ngx_int_t ngx_http_ssl_add_variables(ngx_conf_t *cf);
40 static void *ngx_http_ssl_create_srv_conf(ngx_conf_t *cf);
41 static char *ngx_http_ssl_merge_srv_conf(ngx_conf_t *cf,
42     void *parent, void *child);
43
44 static char *ngx_http_ssl_enable(ngx_conf_t *cf, ngx_command_t *cmd,
45     void *conf);
46 static char *ngx_http_ssl_password_file(ngx_conf_t *cf, ngx_command_t *cmd,
47     void *conf);
48 static char *ngx_http_ssl_session_cache(ngx_conf_t *cf, ngx_command_t *cmd,
49     void *conf);
50
51 static ngx_int_t ngx_http_ssl_init(ngx_conf_t *cf);
52
53
54 static ngx_conf_bitmask_t  ngx_http_ssl_protocols[] = {
55     { ngx_string("SSLv2"),  NGX_SSL_SSLv2 },
56     { ngx_string("SSLv3"),  NGX_SSL_SSLv3 },
57     { ngx_string("TLSv1"),  NGX_SSL_TLSv1 },
58     { ngx_string("TLSv1.1"), NGX_SSL_TLSv1_1 },
59     { ngx_string("TLSv1.2"), NGX_SSL_TLSv1_2 },
60     { ngx_null_string, 0 }
61 };
62
63
64 static ngx_conf_enum_t  ngx_http_ssl_verify[] = {
65     { ngx_string("off"), 0 },
66     { ngx_string("on"), 1 },
67     { ngx_string("optional"), 2 },
68     { ngx_string("optional_no_ca"), 3 },
69     { ngx_null_string, 0 }
70 };
71
72
73 static ngx_command_t  ngx_http_ssl_commands[] = {
74
75     { ngx_string("ssl"),
76     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_CONF_FLAG,
77     ngx_http_ssl_enable,
78     NGX_HTTP_SRV_CONF_OFFSET,
79     offsetof(ngx_http_ssl_srv_conf_t, enable),
80     NULL },

```

```

81 { ngx_string("ssl_certificate"),
82     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_CONF\_TAKE1,
83     ngx_conf_set_str_slot,
84     NGX\_HTTP\_SRV\_CONF\_OFFSET,
85     offsetof(ngx\_http\_ssl\_srv\_conf\_t, certificate),
86     NULL },
87
88
89 { ngx_string("ssl_certificate_key"),
90     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_CONF\_TAKE1,
91     ngx_conf_set_str_slot,
92     NGX\_HTTP\_SRV\_CONF\_OFFSET,
93     offsetof(ngx\_http\_ssl\_srv\_conf\_t, certificate_key),
94     NULL },
95
96 { ngx_string("ssl_password_file"),
97     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_CONF\_TAKE1,
98     ngx_http_ssl_password_file,
99     NGX\_HTTP\_SRV\_CONF\_OFFSET,
100     0,
101     NULL },
102
103 { ngx_string("ssl_dhparam"),
104     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_CONF\_TAKE1,
105     ngx_conf_set_str_slot,
106     NGX\_HTTP\_SRV\_CONF\_OFFSET,
107     offsetof(ngx\_http\_ssl\_srv\_conf\_t, dhparam),
108     NULL },
109
110 { ngx_string("ssl_ecdh_curve"),
111     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_CONF\_TAKE1,
112     ngx_conf_set_str_slot,
113     NGX\_HTTP\_SRV\_CONF\_OFFSET,
114     offsetof(ngx\_http\_ssl\_srv\_conf\_t, ecdh_curve),
115     NULL },
116
117 { ngx_string("ssl_protocols"),
118     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_CONF\_1MORE,
119     ngx_conf_set_bitmask_slot,
120     NGX\_HTTP\_SRV\_CONF\_OFFSET,
121     offsetof(ngx\_http\_ssl\_srv\_conf\_t, protocols),
122     &ngx\_http\_ssl\_protocols },
123
124 { ngx_string("ssl_ciphers"),
125     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_CONF\_TAKE1,
126     ngx_conf_set_str_slot,
127     NGX\_HTTP\_SRV\_CONF\_OFFSET,
128     offsetof(ngx\_http\_ssl\_srv\_conf\_t, ciphers),
129     NULL },
130
131 { ngx_string("ssl_buffer_size"),
132     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_CONF\_TAKE1,
133     ngx_conf_set_size_slot,
134     NGX\_HTTP\_SRV\_CONF\_OFFSET,
135     offsetof(ngx\_http\_ssl\_srv\_conf\_t, buffer_size),
136     NULL },
137
138 { ngx_string("ssl_verify_client"),
139     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_CONF\_TAKE1,
140     ngx_conf_set_enum_slot,
141     NGX\_HTTP\_SRV\_CONF\_OFFSET,
142     offsetof(ngx\_http\_ssl\_srv\_conf\_t, verify),
143     &ngx\_http\_ssl\_verify },
144
145 { ngx_string("ssl_verify_depth"),
146     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_CONF\_TAKE1,
147     ngx_conf_set_num_slot,
148     NGX\_HTTP\_SRV\_CONF\_OFFSET,
149     offsetof(ngx\_http\_ssl\_srv\_conf\_t, verify_depth),
150     NULL },
151
152 { ngx_string("ssl_client_certificate"),
153     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_CONF\_TAKE1,
154     ngx_conf_set_str_slot,
155     NGX\_HTTP\_SRV\_CONF\_OFFSET,
156     offsetof(ngx\_http\_ssl\_srv\_conf\_t, client_certificate),

```

```

157     NULL },
158
159 { ngx_string("ssl_trusted_certificate"),
160     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_CONF TAKE1,
161     ngx\_conf\_set\_str\_slot,
162     NGX\_HTTP\_SRV\_CONF\_OFFSET,
163     offsetof(ngx\_http\_ssl\_srv\_conf\_t, trusted_certificate),
164     NULL },
165
166 { ngx_string("ssl_prefer_server_ciphers"),
167     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_CONF FLAG,
168     ngx\_conf\_set\_flag\_slot,
169     NGX\_HTTP\_SRV\_CONF\_OFFSET,
170     offsetof(ngx\_http\_ssl\_srv\_conf\_t, prefer_server_ciphers),
171     NULL },
172
173 { ngx_string("ssl_session_cache"),
174     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_CONF TAKE12,
175     ngx\_http\_ssl\_session\_cache,
176     NGX\_HTTP\_SRV\_CONF\_OFFSET,
177     0,
178     NULL },
179
180 { ngx_string("ssl_session_tickets"),
181     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_CONF FLAG,
182     ngx\_conf\_set\_flag\_slot,
183     NGX\_HTTP\_SRV\_CONF\_OFFSET,
184     offsetof(ngx\_http\_ssl\_srv\_conf\_t, session_tickets),
185     NULL },
186
187 { ngx_string("ssl_session_ticket_key"),
188     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_CONF TAKE1,
189     ngx\_conf\_set\_str\_array\_slot,
190     NGX\_HTTP\_SRV\_CONF\_OFFSET,
191     offsetof(ngx\_http\_ssl\_srv\_conf\_t, session_ticket_keys),
192     NULL },
193
194 { ngx_string("ssl_session_timeout"),
195     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_CONF TAKE1,
196     ngx\_conf\_set\_sec\_slot,
197     NGX\_HTTP\_SRV\_CONF\_OFFSET,
198     offsetof(ngx\_http\_ssl\_srv\_conf\_t, session_timeout),
199     NULL },
200
201 { ngx_string("ssl_cr1"),
202     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_CONF TAKE1,
203     ngx\_conf\_set\_str\_slot,
204     NGX\_HTTP\_SRV\_CONF\_OFFSET,
205     offsetof(ngx\_http\_ssl\_srv\_conf\_t, cr1),
206     NULL },
207
208 { ngx_string("ssl_stapling"),
209     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_CONF FLAG,
210     ngx\_conf\_set\_flag\_slot,
211     NGX\_HTTP\_SRV\_CONF\_OFFSET,
212     offsetof(ngx\_http\_ssl\_srv\_conf\_t, stapling),
213     NULL },
214
215 { ngx_string("ssl_stapling_file"),
216     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_CONF TAKE1,
217     ngx\_conf\_set\_str\_slot,
218     NGX\_HTTP\_SRV\_CONF\_OFFSET,
219     offsetof(ngx\_http\_ssl\_srv\_conf\_t, stapling_file),
220     NULL },
221
222 { ngx_string("ssl_stapling_responder"),
223     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_CONF TAKE1,
224     ngx\_conf\_set\_str\_slot,
225     NGX\_HTTP\_SRV\_CONF\_OFFSET,
226     offsetof(ngx\_http\_ssl\_srv\_conf\_t, stapling_responder),
227     NULL },
228
229 { ngx_string("ssl_stapling_verify"),
230     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_CONF FLAG,
231     ngx\_conf\_set\_flag\_slot,
232     NGX\_HTTP\_SRV\_CONF\_OFFSET,

```

```

233     offsetof(ngx_http_ssl_srv_conf_t, stapling_verify),
234     NULL },
235
236     ngx_null_command
237 };
238
239
240 static ngx_http_module_t ngx_http_ssl_module_ctx = {
241     ngx_http_ssl_add_variables,          /* preconfiguration */
242     ngx_http_ssl_init,                  /* postconfiguration */
243
244     NULL,                                  /* create main configuration */
245     NULL,                                  /* init main configuration */
246
247     ngx_http_ssl_create_srv_conf,       /* create server configuration */
248     ngx_http_ssl_merge_srv_conf,       /* merge server configuration */
249
250     NULL,                                  /* create location configuration */
251     NULL                                   /* merge location configuration */
252 };
253
254
255 ngx_module_t ngx_http_ssl_module = {
256     NGX_MODULE_V1,
257     &ngx_http_ssl_module_ctx,           /* module context */
258     ngx_http_ssl_commands,            /* module directives */
259     NGX_HTTP_MODULE,                  /* module type */
260     NULL,                                /* init master */
261     NULL,                                /* init module */
262     NULL,                                /* init process */
263     NULL,                                /* init thread */
264     NULL,                                /* exit thread */
265     NULL,                                /* exit process */
266     NULL,                                /* exit master */
267     NGX_MODULE_V1_PADDING
268 };
269
270
271 static ngx_http_variable_t ngx_http_ssl_vars[] = {
272
273     { ngx_string("ssl_protocol"), NULL, ngx_http_ssl_static_variable,
274       (uintptr_t) ngx_ssl_get_protocol, NGX_HTTP_VAR_CHANGEABLE, 0 },
275
276     { ngx_string("ssl_cipher"), NULL, ngx_http_ssl_static_variable,
277       (uintptr_t) ngx_ssl_get_cipher_name, NGX_HTTP_VAR_CHANGEABLE, 0 },
278
279     { ngx_string("ssl_session_id"), NULL, ngx_http_ssl_variable,
280       (uintptr_t) ngx_ssl_get_session_id, NGX_HTTP_VAR_CHANGEABLE, 0 },
281
282     { ngx_string("ssl_session_reused"), NULL, ngx_http_ssl_variable,
283       (uintptr_t) ngx_ssl_get_session_reused, NGX_HTTP_VAR_CHANGEABLE, 0 },
284
285     { ngx_string("ssl_server_name"), NULL, ngx_http_ssl_variable,
286       (uintptr_t) ngx_ssl_get_server_name, NGX_HTTP_VAR_CHANGEABLE, 0 },
287
288     { ngx_string("ssl_client_cert"), NULL, ngx_http_ssl_variable,
289       (uintptr_t) ngx_ssl_get_certificate, NGX_HTTP_VAR_CHANGEABLE, 0 },
290
291     { ngx_string("ssl_client_raw_cert"), NULL, ngx_http_ssl_variable,
292       (uintptr_t) ngx_ssl_get_raw_certificate,
293       NGX_HTTP_VAR_CHANGEABLE, 0 },
294
295     { ngx_string("ssl_client_s_dn"), NULL, ngx_http_ssl_variable,
296       (uintptr_t) ngx_ssl_get_subject_dn, NGX_HTTP_VAR_CHANGEABLE, 0 },
297
298     { ngx_string("ssl_client_i_dn"), NULL, ngx_http_ssl_variable,
299       (uintptr_t) ngx_ssl_get_issuer_dn, NGX_HTTP_VAR_CHANGEABLE, 0 },
300
301     { ngx_string("ssl_client_serial"), NULL, ngx_http_ssl_variable,
302       (uintptr_t) ngx_ssl_get_serial_number, NGX_HTTP_VAR_CHANGEABLE, 0 },
303
304     { ngx_string("ssl_client_fingerprint"), NULL, ngx_http_ssl_variable,
305       (uintptr_t) ngx_ssl_get_fingerprint, NGX_HTTP_VAR_CHANGEABLE, 0 },
306
307     { ngx_string("ssl_client_verify"), NULL, ngx_http_ssl_variable,
308       (uintptr_t) ngx_ssl_get_client_verify, NGX_HTTP_VAR_CHANGEABLE, 0 },

```

```

309     { ngx_null_string, NULL, NULL, 0, 0, 0 }
310 };
311
312
313
314 static ngx_str_t ngx_http_ssl_sess_id_ctx = ngx_string("HTTP");
315
316
317 #ifdef TLSEXT_TYPE_application_layer_protocol_negotiation
318
319 static int
320 ngx_http_ssl_alpn_select(ngx_ssl_conn_t *ssl_conn, const unsigned char **out,
321     unsigned char *outlen, const unsigned char *in, unsigned int inlen,
322     void *arg)
323 {
324     unsigned int     srvlen;
325     unsigned char    *srv;
326     #if (NGX_DEBUG)
327     unsigned int     i;
328     #endif
329     #if (NGX_HTTP_SPDY)
330     ngx_http_connection_t *hc;
331     #endif
332     #if (NGX_HTTP_SPDY || NGX_DEBUG)
333     ngx_connection_t *c;
334
335     c = ngx_ssl_get_connection(ssl_conn);
336     #endif
337
338     #if (NGX_DEBUG)
339     for (i = 0; i < inlen; i += in[i] + 1) {
340         ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
341             "SSL ALPN supported by client: %*s", in[i], &in[i + 1]);
342     }
343     #endif
344
345     #if (NGX_HTTP_SPDY)
346     hc = c->data;
347
348     if (hc->addr_conf->spdy) {
349         srv = (unsigned char *) NGX_SPDY_NPN_ADVERTISE NGX_HTTP_NPN_ADVERTISE;
350         srvlen = sizeof(NGX_SPDY_NPN_ADVERTISE NGX_HTTP_NPN_ADVERTISE) - 1;
351     } else
352     #endif
353     {
354         srv = (unsigned char *) NGX_HTTP_NPN_ADVERTISE;
355         srvlen = sizeof(NGX_HTTP_NPN_ADVERTISE) - 1;
356     }
357
358     if (SSL_select_next_proto((unsigned char **) out, outlen, srv, srvlen,
359         in, inlen)
360         != OPENSSSL_NPN_NEGOTIATED)
361     {
362         return SSL_TLSEXT_ERR_NOACK;
363     }
364
365     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
366         "SSL ALPN selected: %*s", *outlen, *out);
367
368     return SSL_TLSEXT_ERR_OK;
369 }
370
371 #endif
372
373
374
375 #ifdef TLSEXT_TYPE_next_proto_neg
376
377 static int
378 ngx_http_ssl_npn_advertised(ngx_ssl_conn_t *ssl_conn,
379     const unsigned char **out, unsigned int *outlen, void *arg)
380 {
381     #if (NGX_HTTP_SPDY || NGX_DEBUG)
382     ngx_connection_t *c;
383
384     c = ngx_ssl_get_connection(ssl_conn);

```

```

385     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, c->log, 0, "SSL NPN advertised");
386 #endif
387
388 #if (NGX_HTTP_SPDY)
389 {
390     ngx_http_connection_t *hc;
391
392     hc = c->data;
393
394     if (hc->addr_conf->spdy) {
395         *out = (unsigned char *) NGX_SPDY_NPN_ADVERTISE NGX_HTTP_NPN_ADVERTISE;
396         *outlen = sizeof(NGX_SPDY_NPN_ADVERTISE NGX_HTTP_NPN_ADVERTISE) - 1;
397
398         return SSL_TLSEXT_ERR_OK;
399     }
400 }
401 #endif
402
403 *out = (unsigned char *) NGX_HTTP_NPN_ADVERTISE;
404 *outlen = sizeof(NGX_HTTP_NPN_ADVERTISE) - 1;
405
406 return SSL_TLSEXT_ERR_OK;
407 }
408
409 #endif
410
411
412 static ngx_int_t
413 ngx_http_ssl_static_variable(ngx_http_request_t *r,
414     ngx_http_variable_value_t *v, uintptr_t data)
415 {
416     ngx_ssl_variable_handler_pt handler = (ngx_ssl_variable_handler_pt) data;
417
418     size_t len;
419     ngx_str_t s;
420
421     if (r->connection->ssl) {
422
423         (void) handler(r->connection, NULL, &s);
424
425         v->data = s.data;
426
427         for (len = 0; v->data[len]; len++) { /* void */ }
428
429         v->len = len;
430         v->valid = 1;
431         v->no_cacheable = 0;
432         v->not_found = 0;
433
434         return NGX_OK;
435     }
436
437     v->not_found = 1;
438
439     return NGX_OK;
440 }
441
442
443 static ngx_int_t
444 ngx_http_ssl_variable(ngx_http_request_t *r, ngx_http_variable_value_t *v,
445     uintptr_t data)
446 {
447     ngx_ssl_variable_handler_pt handler = (ngx_ssl_variable_handler_pt) data;
448
449     ngx_str_t s;
450
451     if (r->connection->ssl) {
452
453         if (handler(r->connection, r->pool, &s) != NGX_OK) {
454             return NGX_ERROR;
455         }
456
457         v->len = s.len;
458         v->data = s.data;
459
460         if (v->len) {

```

```

461         v->valid = 1;
462         v->no_cacheable = 0;
463         v->not_found = 0;
464
465         return NGX\_OK;
466     }
467 }
468
469 v->not_found = 1;
470
471 return NGX\_OK;
472 }
473
474
475 static ngx\_int\_t
476 ngx\_http\_ssl\_add\_variables(ngx\_conf\_t *cf)
477 {
478     ngx\_http\_variable\_t *var, *v;
479
480     for (v = ngx\_http\_ssl\_vars; v->name.len; v++) {
481         var = ngx\_http\_add\_variable(cf, &v->name, v->flags);
482         if (var == NULL) {
483             return NGX\_ERROR;
484         }
485
486         var->get_handler = v->get_handler;
487         var->data = v->data;
488     }
489
490     return NGX\_OK;
491 }
492
493
494 static void *
495 ngx\_http\_ssl\_create\_srv\_conf(ngx\_conf\_t *cf)
496 {
497     ngx\_http\_ssl\_srv\_conf\_t *sscf;
498
499     sscf = ngx\_palloc(cf->pool, sizeof(ngx\_http\_ssl\_srv\_conf\_t));
500     if (sscf == NULL) {
501         return NULL;
502     }
503
504     /*
505     * set by ngx\_palloc\(\):
506     *
507     *     sscf->protocols = 0;
508     *     sscf->certificate = { 0, NULL };
509     *     sscf->certificate_key = { 0, NULL };
510     *     sscf->dhparam = { 0, NULL };
511     *     sscf->ecdh_curve = { 0, NULL };
512     *     sscf->client_certificate = { 0, NULL };
513     *     sscf->trusted_certificate = { 0, NULL };
514     *     sscf->crl = { 0, NULL };
515     *     sscf->ciphers = { 0, NULL };
516     *     sscf->shm_zone = NULL;
517     *     sscf->stapling_file = { 0, NULL };
518     *     sscf->stapling_responder = { 0, NULL };
519     */
520
521     sscf->enable = NGX\_CONF\_UNSET;
522     sscf->prefer_server_ciphers = NGX\_CONF\_UNSET;
523     sscf->buffer_size = NGX\_CONF\_UNSET\_SIZE;
524     sscf->verify = NGX\_CONF\_UNSET\_UINT;
525     sscf->verify_depth = NGX\_CONF\_UNSET\_UINT;
526     sscf->passwords = NGX\_CONF\_UNSET\_PTR;
527     sscf->builtin_session_cache = NGX\_CONF\_UNSET;
528     sscf->session_timeout = NGX\_CONF\_UNSET;
529     sscf->session_tickets = NGX\_CONF\_UNSET;
530     sscf->session_ticket_keys = NGX\_CONF\_UNSET\_PTR;
531     sscf->stapling = NGX\_CONF\_UNSET;
532     sscf->stapling_verify = NGX\_CONF\_UNSET;
533
534     return sscf;
535 }
536

```



```

537 static char *
538 ngx_http_ssl_merge_srv_conf(ngx_conf_t *cf, void *parent, void *child)
539 {
540     ngx_http_ssl_srv_conf_t *prev = parent;
541     ngx_http_ssl_srv_conf_t *conf = child;
542
543     ngx_pool_cleanup_t *cIn;
544
545     if (conf->enable == NGX_CONF_UNSET) {
546         if (prev->enable == NGX_CONF_UNSET) {
547             conf->enable = 0;
548
549         } else {
550             conf->enable = prev->enable;
551             conf->file = prev->file;
552             conf->line = prev->line;
553         }
554     }
555 }
556
557 ngx_conf_merge_value(conf->session_timeout,
558     prev->session_timeout, 300);
559
560 ngx_conf_merge_value(conf->prefer_server_ciphers,
561     prev->prefer_server_ciphers, 0);
562
563 ngx_conf_merge_bitmask_value(conf->protocols, prev->protocols,
564     (NGX_CONF_BITMASK_SET|NGX_SSL_SSLv3|NGX_SSL_TLSv1
565     |NGX_SSL_TLSv1_1|NGX_SSL_TLSv1_2));
566
567 ngx_conf_merge_size_value(conf->buffer_size, prev->buffer_size,
568     NGX_SSL_BUFSIZE);
569
570 ngx_conf_merge_uint_value(conf->verify, prev->verify, 0);
571 ngx_conf_merge_uint_value(conf->verify_depth, prev->verify_depth, 1);
572
573 ngx_conf_merge_str_value(conf->certificate, prev->certificate, "");
574 ngx_conf_merge_str_value(conf->certificate_key, prev->certificate_key, "");
575
576 ngx_conf_merge_ptr_value(conf->passwords, prev->passwords, NULL);
577
578 ngx_conf_merge_str_value(conf->dhparam, prev->dhparam, "");
579
580 ngx_conf_merge_str_value(conf->client_certificate, prev->client_certificate,
581     "");
582 ngx_conf_merge_str_value(conf->trusted_certificate,
583     prev->trusted_certificate, "");
584 ngx_conf_merge_str_value(conf->crl, prev->crl, "");
585
586 ngx_conf_merge_str_value(conf->ecdh_curve, prev->ecdh_curve,
587     NGX_DEFAULT_ECDH_CURVE);
588
589 ngx_conf_merge_str_value(conf->ciphers, prev->ciphers, NGX_DEFAULT_CIPHERS);
590
591 ngx_conf_merge_value(conf->stapling, prev->stapling, 0);
592 ngx_conf_merge_value(conf->stapling_verify, prev->stapling_verify, 0);
593 ngx_conf_merge_str_value(conf->stapling_file, prev->stapling_file, "");
594 ngx_conf_merge_str_value(conf->stapling_responder,
595     prev->stapling_responder, "");
596
597 conf->ssl.log = cf->log;
598
599 if (conf->enable) {
600
601     if (conf->certificate.len == 0) {
602         ngx_log_error(NGX_LOG_EMERG, cf->log, 0,
603             "no \"ssl_certificate\" is defined for "
604             "the \"ssl\" directive in %s:%ui",
605             conf->file, conf->line);
606         return NGX_CONF_ERROR;
607     }
608
609     if (conf->certificate_key.len == 0) {
610         ngx_log_error(NGX_LOG_EMERG, cf->log, 0,
611             "no \"ssl_certificate_key\" is defined for "
612             "the \"ssl\" directive in %s:%ui",

```

```

613         conf->file, conf->line);
614     return NGX\_CONF\_ERROR;
615 }
616
617 } else {
618
619     if (conf->certificate.len == 0) {
620         return NGX\_CONF\_OK;
621     }
622
623     if (conf->certificate_key.len == 0) {
624         ngx\_log\_error(NGX\_LOG\_EMERG, cf->log, 0,
625             "no \"ssl_certificate_key\" is defined "
626             "for certificate \"%V\"", &conf->certificate);
627         return NGX\_CONF\_ERROR;
628     }
629 }
630
631 if (ngx\_ssl\_create(&conf->ssl, conf->protocols, conf) != NGX\_OK) {
632     return NGX\_CONF\_ERROR;
633 }
634
635 #ifdef SSL\_CTRL\_SET\_TLSEXT\_HOSTNAME
636
637     if (SSL\_CTX\_set\_tlsext\_servername\_callback(conf->ssl.ctx,
638         ngx\_http\_ssl\_servername)
639         == 0)
640     {
641         ngx\_log\_error(NGX\_LOG\_WARN, cf->log, 0,
642             "nginx was built with SNI support, however, now it is linked "
643             "dynamically to an OpenSSL library which has no tlsext support, "
644             "therefore SNI is not available");
645     }
646
647 #endif
648
649 #ifdef TLSEXT\_TYPE\_application\_layer\_protocol\_negotiation
650     SSL\_CTX\_set\_alpn\_select\_cb(conf->ssl.ctx, ngx\_http\_ssl\_alpn\_select, NULL);
651 #endif
652
653 #ifdef TLSEXT\_TYPE\_next\_proto\_neg
654     SSL\_CTX\_set\_next\_protos\_advertised\_cb(conf->ssl.ctx,
655         ngx\_http\_ssl\_npn\_advertised, NULL);
656 #endif
657
658     cln = ngx\_pool\_cleanup\_add(cf->pool, 0);
659     if (cln == NULL) {
660         return NGX\_CONF\_ERROR;
661     }
662
663     cln->handler = ngx\_ssl\_cleanup\_ctx;
664     cln->data = &conf->ssl;
665
666     if (ngx\_ssl\_certificate(cf, &conf->ssl, &conf->certificate,
667         &conf->certificate_key, conf->passwords)
668         != NGX\_OK)
669     {
670         return NGX\_CONF\_ERROR;
671     }
672
673     if (SSL\_CTX\_set\_cipher\_list(conf->ssl.ctx,
674         (const char *) conf->ciphers.data)
675         == 0)
676     {
677         ngx\_ssl\_error(NGX\_LOG\_EMERG, cf->log, 0,
678             "SSL_CTX_set_cipher_list(\"%V\") failed",
679             &conf->ciphers);
680         return NGX\_CONF\_ERROR;
681     }
682
683     conf->ssl.buffer_size = conf->buffer_size;
684
685     if (conf->verify) {
686
687         if (conf->client_certificate.len == 0 && conf->verify != 3) {
688             ngx\_log\_error(NGX\_LOG\_EMERG, cf->log, 0,

```

```

689         "no ssl_client_certificate for ssl_client_verify");
690     return NGX_CONF_ERROR;
691 }
692
693 if (ngx_ssl_client_certificate(cf, &conf->ssl,
694                             &conf->client_certificate,
695                             conf->verify_depth)
696     != NGX_OK)
697 {
698     return NGX_CONF_ERROR;
699 }
700 }
701
702 if (ngx_ssl_trusted_certificate(cf, &conf->ssl,
703                               &conf->trusted_certificate,
704                               conf->verify_depth)
705     != NGX_OK)
706 {
707     return NGX_CONF_ERROR;
708 }
709
710 if (ngx_ssl_crl(cf, &conf->ssl, &conf->crl) != NGX_OK) {
711     return NGX_CONF_ERROR;
712 }
713
714 if (conf->prefer_server_ciphers) {
715     SSL_CTX_set_options(conf->ssl.ctx, SSL_OP_CIPHER_SERVER_PREFERENCE);
716 }
717
718 /* a temporary 512-bit RSA key is required for export versions of MSIE */
719 SSL_CTX_set_tmp_rsa_callback(conf->ssl.ctx, ngx_ssl_rsa512_key_callback);
720
721 if (ngx_ssl_dhparam(cf, &conf->ssl, &conf->dhparam) != NGX_OK) {
722     return NGX_CONF_ERROR;
723 }
724
725 if (ngx_ssl_ecdh_curve(cf, &conf->ssl, &conf->ecdh_curve) != NGX_OK) {
726     return NGX_CONF_ERROR;
727 }
728
729 ngx_conf_merge_value(conf->builtin_session_cache,
730                     prev->builtin_session_cache, NGX_SSL_NONE_SCACHE);
731
732 if (conf->shm_zone == NULL) {
733     conf->shm_zone = prev->shm_zone;
734 }
735
736 if (ngx_ssl_session_cache(&conf->ssl, &ngx_http_ssl_sess_id_ctx,
737                          conf->builtin_session_cache,
738                          conf->shm_zone, conf->session_timeout)
739     != NGX_OK)
740 {
741     return NGX_CONF_ERROR;
742 }
743
744 ngx_conf_merge_value(conf->session_tickets, prev->session_tickets, 1);
745
746 #ifndef SSL_OP_NO_TICKET
747     if (!conf->session_tickets) {
748         SSL_CTX_set_options(conf->ssl.ctx, SSL_OP_NO_TICKET);
749     }
750 #endif
751
752 ngx_conf_merge_ptr_value(conf->session_ticket_keys,
753                         prev->session_ticket_keys, NULL);
754
755 if (ngx_ssl_session_ticket_keys(cf, &conf->ssl, conf->session_ticket_keys)
756     != NGX_OK)
757 {
758     return NGX_CONF_ERROR;
759 }
760
761 if (conf->stapling) {
762     if (ngx_ssl_stapling(cf, &conf->ssl, &conf->stapling_file,
763                        &conf->stapling_responder, conf->stapling_verify)

```

```

765         != NGX_OK)
766     {
767         return NGX_CONF_ERROR;
768     }
769 }
770 }
771 }
772     return NGX_CONF_OK;
773 }
774
775
776 static char *
777 ngx_http_ssl_enable(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
778 {
779     ngx_http_ssl_srv_conf_t *sscf = conf;
780
781     char *rv;
782
783     rv = ngx_conf_set_flag_slot(cf, cmd, conf);
784
785     if (rv != NGX_CONF_OK) {
786         return rv;
787     }
788
789     sscf->file = cf->conf_file->file.name.data;
790     sscf->line = cf->conf_file->line;
791
792     return NGX_CONF_OK;
793 }
794
795
796 static char *
797 ngx_http_ssl_password_file(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
798 {
799     ngx_http_ssl_srv_conf_t *sscf = conf;
800
801     ngx_str_t *value;
802
803     if (sscf->passwords != NGX_CONF_UNSET_PTR) {
804         return "is duplicate";
805     }
806
807     value = cf->args->elts;
808
809     sscf->passwords = ngx_ssl_read_password_file(cf, &value[1]);
810
811     if (sscf->passwords == NULL) {
812         return NGX_CONF_ERROR;
813     }
814
815     return NGX_CONF_OK;
816 }
817
818
819 static char *
820 ngx_http_ssl_session_cache(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
821 {
822     ngx_http_ssl_srv_conf_t *sscf = conf;
823
824     size_t len;
825     ngx_str_t *value, name, size;
826     ngx_int_t n;
827     ngx_uint_t i, j;
828
829     value = cf->args->elts;
830
831     for (i = 1; i < cf->args->nelts; i++) {
832
833         if (ngx_strcmp(value[i].data, "off") == 0) {
834             sscf->builtin_session_cache = NGX_SSL_NO_SCACHE;
835             continue;
836         }
837
838         if (ngx_strcmp(value[i].data, "none") == 0) {
839             sscf->builtin_session_cache = NGX_SSL_NONE_SCACHE;
840             continue;

```

```

841     }
842
843     if (ngx_strcmp(value[i].data, "builtin") == 0) {
844         sscf->builtin_session_cache = NGX\_SSL\_DFLT\_BUILTIN\_SCACHE;
845         continue;
846     }
847
848     if (value[i].len > sizeof("builtin:") - 1
849         && ngx_strncmp(value[i].data, "builtin:", sizeof("builtin:") - 1)
850             == 0)
851     {
852         n = ngx_atoi(value[i].data + sizeof("builtin:") - 1,
853                     value[i].len - (sizeof("builtin:") - 1));
854
855         if (n == NGX\_ERROR) {
856             goto invalid;
857         }
858
859         sscf->builtin_session_cache = n;
860
861         continue;
862     }
863
864     if (value[i].len > sizeof("shared:") - 1
865         && ngx_strncmp(value[i].data, "shared:", sizeof("shared:") - 1)
866             == 0)
867     {
868         len = 0;
869
870         for (j = sizeof("shared:") - 1; j < value[i].len; j++) {
871             if (value[i].data[j] == ':') {
872                 break;
873             }
874
875             len++;
876         }
877
878         if (len == 0) {
879             goto invalid;
880         }
881
882         name.len = len;
883         name.data = value[i].data + sizeof("shared:") - 1;
884
885         size.len = value[i].len - j - 1;
886         size.data = name.data + len + 1;
887
888         n = ngx_parse_size(&size);
889
890         if (n == NGX\_ERROR) {
891             goto invalid;
892         }
893
894         if (n < (ngx\_int\_t) (8 * ngx\_pagesize)) {
895             ngx_conf_log_error(NGX\_LOG\_EMERG, cf, 0,
896                               "session cache \"%V\" is too small",
897                               &value[i]);
898
899             return NGX\_CONF\_ERROR;
900         }
901
902         sscf->shm_zone = ngx_shared_memory_add(cf, &name, n,
903                                               &ngx_http_ssl_module);
904         if (sscf->shm_zone == NULL) {
905             return NGX\_CONF\_ERROR;
906         }
907
908         sscf->shm_zone->init = ngx_ssl_session_cache_init;
909
910         continue;
911     }
912
913     goto invalid;
914 }
915
916 if (sscf->shm_zone && sscf->builtin_session_cache == NGX\_CONF\_UNSET) {

```

```

917     sscf->builtin_session_cache = NGX\_SSL\_NO\_BUILTIN\_SCACHE;
918 }
919
920     return NGX\_CONF\_OK;
921
922 invalid:
923
924     ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
925         "invalid session cache \"%V\"", &value[i]);
926
927     return NGX\_CONF\_ERROR;
928 }
929
930
931 static ngx\_int\_t
932 ngx\_http\_ssl\_init(ngx\_conf\_t *cf)
933 {
934     ngx\_uint\_t          s;
935     ngx\_http\_ssl\_srv\_conf\_t *sscf;
936     ngx\_http\_core\_loc\_conf\_t *clcf;
937     ngx\_http\_core\_srv\_conf\_t **cscfp;
938     ngx\_http\_core\_main\_conf\_t *cmcf;
939
940     cmcf = ngx\_http\_conf\_get\_module\_main\_conf(cf, ngx\_http\_core\_module);
941     cscfp = cmcf->servers.elts;
942
943     for (s = 0; s < cmcf->servers.nelts; s++) {
944
945         sscf = cscfp[s]->ctx->srv_conf[ngx\_http\_ssl\_module.ctx_index];
946
947         if (sscf->ssl.ctx == NULL || !sscf->stapling) {
948             continue;
949         }
950
951         clcf = cscfp[s]->ctx->loc_conf[ngx\_http\_core\_module.ctx_index];
952
953         if (ngx\_ssl\_stapling\_resolver(cf, &sscf->ssl, clcf->resolver,
954             clcf->resolver_timeout)
955             != NGX\_OK)
956         {
957             return NGX\_ERROR;
958         }
959     }
960
961     return NGX\_OK;
962 }

```

[One Level Up](#)

[Top Level](#)

## src/mail/nginx\_mail\_ssl\_module.c - nginx-1.7.10

### Global variables defined

- [ngx\\_mail\\_ssl\\_commands](#)
- [ngx\\_mail\\_ssl\\_module](#)
- [ngx\\_mail\\_ssl\\_module\\_ctx](#)
- [ngx\\_mail\\_ssl\\_protocols](#)
- [ngx\\_mail\\_ssl\\_sess\\_id\\_ctx](#)
- [ngx\\_mail\\_starttls\\_state](#)

### Functions defined

- [ngx\\_mail\\_ssl\\_create\\_conf](#)
- [ngx\\_mail\\_ssl\\_enable](#)
- [ngx\\_mail\\_ssl\\_merge\\_conf](#)
- [ngx\\_mail\\_ssl\\_password\\_file](#)
- [ngx\\_mail\\_ssl\\_session\\_cache](#)
- [ngx\\_mail\\_ssl\\_starttls](#)

### Macros defined

- [NGX\\_DEFAULT\\_CIPHERS](#)
- [NGX\\_DEFAULT\\_ECDH\\_CURVE](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_mail.h>
11
12
13 #define NGX_DEFAULT_CIPHERS    "HIGH:!aNULL:!MD5"
14 #define NGX_DEFAULT_ECDH_CURVE "prime256v1"
15
16
17 static void *ngx_mail_ssl_create_conf(ngx_conf_t *cf);
18 static char *ngx_mail_ssl_merge_conf(ngx_conf_t *cf, void *parent, void *child);
19
20 static char *ngx_mail_ssl_enable(ngx_conf_t *cf, ngx_command_t *cmd,
21     void *conf);
22 static char *ngx_mail_ssl_starttls(ngx_conf_t *cf, ngx_command_t *cmd,
23     void *conf);
24 static char *ngx_mail_ssl_password_file(ngx_conf_t *cf, ngx_command_t *cmd,
25     void *conf);
26 static char *ngx_mail_ssl_session_cache(ngx_conf_t *cf, ngx_command_t *cmd,
```

```

27     void *conf);
28
29
30 static ngx_conf_enum_t  ngx_mail_starttls_state[] = {
31     { ngx_string("off"),  NGX_MAIL_STARTTLS_OFF },
32     { ngx_string("on"),   NGX_MAIL_STARTTLS_ON  },
33     { ngx_string("only"), NGX_MAIL_STARTTLS_ONLY },
34     { ngx_null_string, 0 }
35 };
36
37
38
39 static ngx_conf_bitmask_t  ngx_mail_ssl_protocols[] = {
40     { ngx_string("SSLv2"),  NGX_SSL_SSLv2 },
41     { ngx_string("SSLv3"),  NGX_SSL_SSLv3 },
42     { ngx_string("TLSv1"),  NGX_SSL_TLSv1 },
43     { ngx_string("TLSv1.1"), NGX_SSL_TLSv1_1 },
44     { ngx_string("TLSv1.2"), NGX_SSL_TLSv1_2 },
45     { ngx_null_string, 0 }
46 };
47
48
49 static ngx_command_t  ngx_mail_ssl_commands[] = {
50
51     { ngx_string("ssl"),
52       NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_FLAG,
53       ngx_mail_ssl_enable,
54       NGX_MAIL_SRV_CONF_OFFSET,
55       offsetof(ngx_mail_ssl_conf_t, enable),
56       NULL },
57
58     { ngx_string("starttls"),
59       NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_TAKE1,
60       ngx_mail_ssl_starttls,
61       NGX_MAIL_SRV_CONF_OFFSET,
62       offsetof(ngx_mail_ssl_conf_t, starttls),
63       ngx_mail_starttls_state },
64
65     { ngx_string("ssl_certificate"),
66       NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_TAKE1,
67       ngx_conf_set_str_slot,
68       NGX_MAIL_SRV_CONF_OFFSET,
69       offsetof(ngx_mail_ssl_conf_t, certificate),
70       NULL },
71
72     { ngx_string("ssl_certificate_key"),
73       NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_TAKE1,
74       ngx_conf_set_str_slot,
75       NGX_MAIL_SRV_CONF_OFFSET,
76       offsetof(ngx_mail_ssl_conf_t, certificate_key),
77       NULL },
78
79     { ngx_string("ssl_password_file"),
80       NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_TAKE1,
81       ngx_mail_ssl_password_file,
82       NGX_MAIL_SRV_CONF_OFFSET,
83       0,
84       NULL },
85
86     { ngx_string("ssl_dhparam"),
87       NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_TAKE1,
88       ngx_conf_set_str_slot,
89       NGX_MAIL_SRV_CONF_OFFSET,
90       offsetof(ngx_mail_ssl_conf_t, dhparam),
91       NULL },
92
93     { ngx_string("ssl_ecdh_curve"),
94       NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_TAKE1,
95       ngx_conf_set_str_slot,
96       NGX_MAIL_SRV_CONF_OFFSET,
97       offsetof(ngx_mail_ssl_conf_t, ecdh_curve),
98       NULL },
99
100    { ngx_string("ssl_protocols"),
101      NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_1MORE,
102      ngx_conf_set_bitmask_slot,

```



```

103     NGX\_MAIL\_SRV\_CONF\_OFFSET,
104     offsetof\(ngx\\_mail\\_ssl\\_conf\\_t, protocols\),
105     &ngx\\_mail\\_ssl\\_protocols },
106
107     { ngx\_string\("ssl\_ciphers"\),
108     NGX\_MAIL\_MAIN\_CONF|NGX\_MAIL\_SRV\_CONF|NGX\_CONF\_TAKE1,
109     ngx\_conf\_set\_str\_slot,
110     NGX\_MAIL\_SRV\_CONF\_OFFSET,
111     offsetof\(ngx\\_mail\\_ssl\\_conf\\_t, ciphers\),
112     NULL },
113
114     { ngx\_string\("ssl\_prefer\_server\_ciphers"\),
115     NGX\_MAIL\_MAIN\_CONF|NGX\_MAIL\_SRV\_CONF|NGX\_CONF\_FLAG,
116     ngx\_conf\_set\_flag\_slot,
117     NGX\_MAIL\_SRV\_CONF\_OFFSET,
118     offsetof\(ngx\\_mail\\_ssl\\_conf\\_t, prefer\_server\_ciphers\),
119     NULL },
120
121     { ngx\_string\("ssl\_session\_cache"\),
122     NGX\_MAIL\_MAIN\_CONF|NGX\_MAIL\_SRV\_CONF|NGX\_CONF\_TAKE12,
123     ngx\_mail\_ssl\_session\_cache,
124     NGX\_MAIL\_SRV\_CONF\_OFFSET,
125     0,
126     NULL },
127
128     { ngx\_string\("ssl\_session\_tickets"\),
129     NGX\_MAIL\_MAIN\_CONF|NGX\_MAIL\_SRV\_CONF|NGX\_CONF\_FLAG,
130     ngx\_conf\_set\_flag\_slot,
131     NGX\_MAIL\_SRV\_CONF\_OFFSET,
132     offsetof\(ngx\\_mail\\_ssl\\_conf\\_t, session\_tickets\),
133     NULL },
134
135     { ngx\_string\("ssl\_session\_ticket\_key"\),
136     NGX\_MAIL\_MAIN\_CONF|NGX\_MAIL\_SRV\_CONF|NGX\_CONF\_TAKE1,
137     ngx\_conf\_set\_str\_array\_slot,
138     NGX\_MAIL\_SRV\_CONF\_OFFSET,
139     offsetof\(ngx\\_mail\\_ssl\\_conf\\_t, session\_ticket\_keys\),
140     NULL },
141
142     { ngx\_string\("ssl\_session\_timeout"\),
143     NGX\_MAIL\_MAIN\_CONF|NGX\_MAIL\_SRV\_CONF|NGX\_CONF\_TAKE1,
144     ngx\_conf\_set\_sec\_slot,
145     NGX\_MAIL\_SRV\_CONF\_OFFSET,
146     offsetof\(ngx\\_mail\\_ssl\\_conf\\_t, session\_timeout\),
147     NULL },
148
149     ngx\_null\_command
150 };
151
152
153 static ngx\_mail\_module\_t ngx\_mail\_ssl\_module\_ctx = {
154     NULL, /* protocol */
155
156     NULL, /* create main configuration */
157     NULL, /* init main configuration */
158
159     ngx\_mail\_ssl\_create\_conf, /* create server configuration */
160     ngx\_mail\_ssl\_merge\_conf /* merge server configuration */
161 };
162
163
164 ngx\_module\_t ngx\_mail\_ssl\_module = {
165     NGX\_MODULE\_V1, /* module context */
166     &ngx\\_mail\\_ssl\\_module\\_ctx, /* module directives */
167     ngx\_mail\_ssl\_commands, /* module type */
168     NGX\_MAIL\_MODULE, /* init master */
169     NULL, /* init module */
170     NULL, /* init process */
171     NULL, /* init thread */
172     NULL, /* exit thread */
173     NULL, /* exit process */
174     NULL, /* exit master */
175     NGX\_MODULE\_V1\_PADDING
176 };
177
178

```

```

179 static ngx_str_t ngx_mail_ssl_sess_id_ctx = ngx_string("MAIL");
180
181
182
183 static void *
184 ngx_mail_ssl_create_conf(ngx_conf_t *cf)
185 {
186     ngx_mail_ssl_conf_t *scf;
187
188     scf = ngx_palloc(cf->pool, sizeof(ngx_mail_ssl_conf_t));
189     if (scf == NULL) {
190         return NULL;
191     }
192
193     /*
194     * set by ngx_palloc():
195     *
196     *     scf->protocols = 0;
197     *     scf->certificate = { 0, NULL };
198     *     scf->certificate_key = { 0, NULL };
199     *     scf->dhparam = { 0, NULL };
200     *     scf->ecdh_curve = { 0, NULL };
201     *     scf->ciphers = { 0, NULL };
202     *     scf->shm_zone = NULL;
203     */
204
205     scf->enable = NGX_CONF_UNSET;
206     scf->starttls = NGX_CONF_UNSET_UINT;
207     scf->passwords = NGX_CONF_UNSET_PTR;
208     scf->prefer_server_ciphers = NGX_CONF_UNSET;
209     scf->builtin_session_cache = NGX_CONF_UNSET;
210     scf->session_timeout = NGX_CONF_UNSET;
211     scf->session_tickets = NGX_CONF_UNSET;
212     scf->session_ticket_keys = NGX_CONF_UNSET_PTR;
213
214     return scf;
215 }
216
217
218 static char *
219 ngx_mail_ssl_merge_conf(ngx_conf_t *cf, void *parent, void *child)
220 {
221     ngx_mail_ssl_conf_t *prev = parent;
222     ngx_mail_ssl_conf_t *conf = child;
223
224     char *mode;
225     ngx_pool_cleanup_t *cfn;
226
227     ngx_conf_merge_value(conf->enable, prev->enable, 0);
228     ngx_conf_merge_uint_value(conf->starttls, prev->starttls,
229                               NGX_MAIL_STARTTLS_OFF);
230
231     ngx_conf_merge_value(conf->session_timeout,
232                           prev->session_timeout, 300);
233
234     ngx_conf_merge_value(conf->prefer_server_ciphers,
235                           prev->prefer_server_ciphers, 0);
236
237     ngx_conf_merge_bitmask_value(conf->protocols, prev->protocols,
238                                  (NGX_CONF_BITMASK_SET|NGX_SSL_SSLv3|NGX_SSL_TLSv1
239                                   |NGX_SSL_TLSv1_1|NGX_SSL_TLSv1_2));
240
241     ngx_conf_merge_str_value(conf->certificate, prev->certificate, "");
242     ngx_conf_merge_str_value(conf->certificate_key, prev->certificate_key, "");
243
244     ngx_conf_merge_ptr_value(conf->passwords, prev->passwords, NULL);
245
246     ngx_conf_merge_str_value(conf->dhparam, prev->dhparam, "");
247
248     ngx_conf_merge_str_value(conf->ecdh_curve, prev->ecdh_curve,
249                               NGX_DEFAULT_ECDH_CURVE);
250
251     ngx_conf_merge_str_value(conf->ciphers, prev->ciphers, NGX_DEFAULT_CIPHERS);
252
253
254     conf->ssl.log = cf->log;

```

```

255     if (conf->enable) {
256         mode = "ssl";
257     }
258
259     } else if (conf->starttls != NGX\_MAIL\_STARTTLS\_OFF) {
260         mode = "starttls";
261     }
262     } else {
263         mode = "";
264     }
265
266     if (conf->file == NULL) {
267         conf->file = prev->file;
268         conf->line = prev->line;
269     }
270
271     if (*mode) {
272
273         if (conf->certificate.len == 0) {
274             ngx\_log\_error(NGX\_LOG\_EMERG, cf->log, 0,
275                 "no \"ssl_certificate\" is defined for "
276                 "the \"%s\" directive in %s:%ui",
277                 mode, conf->file, conf->line);
278             return NGX\_CONF\_ERROR;
279         }
280
281         if (conf->certificate_key.len == 0) {
282             ngx\_log\_error(NGX\_LOG\_EMERG, cf->log, 0,
283                 "no \"ssl_certificate_key\" is defined for "
284                 "the \"%s\" directive in %s:%ui",
285                 mode, conf->file, conf->line);
286             return NGX\_CONF\_ERROR;
287         }
288
289     } else {
290
291         if (conf->certificate.len == 0) {
292             return NGX\_CONF\_OK;
293         }
294
295         if (conf->certificate_key.len == 0) {
296             ngx\_log\_error(NGX\_LOG\_EMERG, cf->log, 0,
297                 "no \"ssl_certificate_key\" is defined "
298                 "for certificate \"%V\"",
299                 &conf->certificate);
300             return NGX\_CONF\_ERROR;
301         }
302     }
303
304     if (ngx\_ssl\_create(&conf->ssl, conf->protocols, NULL) != NGX\_OK) {
305         return NGX\_CONF\_ERROR;
306     }
307
308     cln = ngx\_pool\_cleanup\_add(cf->pool, 0);
309     if (cln == NULL) {
310         return NGX\_CONF\_ERROR;
311     }
312
313     cln->handler = ngx\_ssl\_cleanup\_ctx;
314     cln->data = &conf->ssl;
315
316     if (ngx\_ssl\_certificate(cf, &conf->ssl, &conf->certificate,
317         &conf->certificate_key, conf->passwords)
318         != NGX\_OK)
319     {
320         return NGX\_CONF\_ERROR;
321     }
322
323     if (SSL\_CTX\_set\_cipher\_list(conf->ssl.ctx,
324         (const char *) conf->ciphers.data)
325         == 0)
326     {
327         ngx\_ssl\_error(NGX\_LOG\_EMERG, cf->log, 0,
328             "SSL_CTX_set_cipher_list(\"%V\") failed",
329             &conf->ciphers);
330         return NGX\_CONF\_ERROR;

```

```

331 }
332
333 if (conf->prefer_server_ciphers) {
334     SSL_CTX_set_options(conf->ssl.ctx, SSL_OP_CIPHER_SERVER_PREFERENCE);
335 }
336
337 SSL_CTX_set_tmp_rsa_callback(conf->ssl.ctx, ngx\_ssl\_rsa512\_key\_callback);
338
339 if (ngx\_ssl\_dhparam(cf, &conf->ssl, &conf->dhparam) != NGX\_OK) {
340     return NGX\_CONF\_ERROR;
341 }
342
343 if (ngx\_ssl\_ecdh\_curve(cf, &conf->ssl, &conf->ecdh_curve) != NGX\_OK) {
344     return NGX\_CONF\_ERROR;
345 }
346
347 ngx\_conf\_merge\_value(conf->builtin_session_cache,
348     prev->builtin_session_cache, NGX\_SSL\_NONE\_SCACHE);
349
350 if (conf->shm_zone == NULL) {
351     conf->shm_zone = prev->shm_zone;
352 }
353
354 if (ngx\_ssl\_session\_cache(&conf->ssl, &ngx\_mail\_ssl\_sess\_id\_ctx,
355     conf->builtin_session_cache,
356     conf->shm_zone, conf->session_timeout)
357     != NGX\_OK)
358 {
359     return NGX\_CONF\_ERROR;
360 }
361
362 ngx\_conf\_merge\_value(conf->session_tickets,
363     prev->session_tickets, 1);
364
365 #ifndef SSL\_OP\_NO\_TICKET
366     if (!conf->session_tickets) {
367         SSL_CTX_set_options(conf->ssl.ctx, SSL\_OP\_NO\_TICKET);
368     }
369 #endif
370
371 ngx\_conf\_merge\_ptr\_value(conf->session_ticket_keys,
372     prev->session_ticket_keys, NULL);
373
374 if (ngx\_ssl\_session\_ticket\_keys(cf, &conf->ssl, conf->session_ticket_keys)
375     != NGX\_OK)
376 {
377     return NGX\_CONF\_ERROR;
378 }
379
380 return NGX\_CONF\_OK;
381 }
382
383
384 static char *
385 ngx\_mail\_ssl\_enable(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
386 {
387     ngx\_mail\_ssl\_conf\_t *scf = conf;
388
389     char *rv;
390
391     rv = ngx\_conf\_set\_flag\_slot(cf, cmd, conf);
392
393     if (rv != NGX\_CONF\_OK) {
394         return rv;
395     }
396
397     if (scf->enable && (ngx\_int\_t) scf->starttls > NGX\_MAIL\_STARTTLS\_OFF) {
398         ngx\_conf\_log\_error(NGX\_LOG\_WARN, cf, 0,
399             "\"starttls\" directive conflicts with \"ssl on\"");
400         return NGX\_CONF\_ERROR;
401     }
402
403     scf->file = cf->conf_file->file.name.data;
404     scf->line = cf->conf_file->line;
405
406     return NGX\_CONF\_OK;

```

```

407 }
408
409
410 static char *
411 ngx_mail_ssl_starttls(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
412 {
413     ngx_mail_ssl_conf_t *scf = conf;
414
415     char *rv;
416
417     rv = ngx_conf_set_enum_slot(cf, cmd, conf);
418
419     if (rv != NGX_CONF_OK) {
420         return rv;
421     }
422
423     if (scf->enable == 1 && (ngx_int_t) scf->starttls > NGX_MAIL_STARTTLS_OFF) {
424         ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
425             "\"ssl\" directive conflicts with \"starttls\"");
426         return NGX_CONF_ERROR;
427     }
428
429     scf->file = cf->conf_file->file.name.data;
430     scf->line = cf->conf_file->line;
431
432     return NGX_CONF_OK;
433 }
434
435
436 static char *
437 ngx_mail_ssl_password_file(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
438 {
439     ngx_mail_ssl_conf_t *scf = conf;
440
441     ngx_str_t *value;
442
443     if (scf->passwords != NGX_CONF_UNSET_PTR) {
444         return "is duplicate";
445     }
446
447     value = cf->args->elts;
448
449     scf->passwords = ngx_ssl_read_password_file(cf, &value[1]);
450
451     if (scf->passwords == NULL) {
452         return NGX_CONF_ERROR;
453     }
454
455     return NGX_CONF_OK;
456 }
457
458
459 static char *
460 ngx_mail_ssl_session_cache(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
461 {
462     ngx_mail_ssl_conf_t *scf = conf;
463
464     size_t len;
465     ngx_str_t *value, name, size;
466     ngx_int_t n;
467     ngx_uint_t i, j;
468
469     value = cf->args->elts;
470
471     for (i = 1; i < cf->args->nelts; i++) {
472
473         if (ngx_strcmp(value[i].data, "off") == 0) {
474             scf->builtin_session_cache = NGX_SSL_NO_SCACHE;
475             continue;
476         }
477
478         if (ngx_strcmp(value[i].data, "none") == 0) {
479             scf->builtin_session_cache = NGX_SSL_NONE_SCACHE;
480             continue;
481         }
482

```

```

483     if (ngx_strcmp(value[i].data, "builtin") == 0) {
484         scf->builtin_session_cache = NGX_SSL_DFLT_BUILTIN_SCACHE;
485         continue;
486     }
487
488     if (value[i].len > sizeof("builtin:") - 1
489         && ngx_strncmp(value[i].data, "builtin:", sizeof("builtin:") - 1)
490             == 0)
491     {
492         n = ngx_atoi(value[i].data + sizeof("builtin:") - 1,
493                     value[i].len - (sizeof("builtin:") - 1));
494
495         if (n == NGX_ERROR) {
496             goto invalid;
497         }
498
499         scf->builtin_session_cache = n;
500
501         continue;
502     }
503
504     if (value[i].len > sizeof("shared:") - 1
505         && ngx_strncmp(value[i].data, "shared:", sizeof("shared:") - 1)
506             == 0)
507     {
508         len = 0;
509
510         for (j = sizeof("shared:") - 1; j < value[i].len; j++) {
511             if (value[i].data[j] == ':') {
512                 break;
513             }
514
515             len++;
516         }
517
518         if (len == 0) {
519             goto invalid;
520         }
521
522         name.len = len;
523         name.data = value[i].data + sizeof("shared:") - 1;
524
525         size.len = value[i].len - j - 1;
526         size.data = name.data + len + 1;
527
528         n = ngx_parse_size(&size);
529
530         if (n == NGX_ERROR) {
531             goto invalid;
532         }
533
534         if (n < (ngx_int_t) (8 * ngx_pagesize)) {
535             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
536                             "session cache \"%V\" is too small",
537                             &value[i]);
538
539             return NGX_CONF_ERROR;
540         }
541
542         scf->shm_zone = ngx_shared_memory_add(cf, &name, n,
543                                             &ngx_mail_ssl_module);
544         if (scf->shm_zone == NULL) {
545             return NGX_CONF_ERROR;
546         }
547
548         scf->shm_zone->init = ngx_ssl_session_cache_init;
549
550         continue;
551     }
552
553     goto invalid;
554 }
555
556 if (scf->shm_zone && scf->builtin_session_cache == NGX_CONF_UNSET) {
557     scf->builtin_session_cache = NGX_SSL_NO_BUILTIN_SCACHE;
558 }

```

```
559     return NGX\_CONF\_OK;  
560  
561  
562 invalid:  
563  
564     ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,  
565         "invalid session cache \"%V\"", &value[i]);  
566  
567     return NGX\_CONF\_ERROR;  
568 }
```

[One Level Up](#)

[Top Level](#)

## src/mail/ - nginx-1.7.10

- [ngx\\_mail.c](#)
- [ngx\\_mail.h](#)
- [ngx\\_mail\\_auth\\_http\\_module.c](#)
- [ngx\\_mail\\_core\\_module.c](#)
- [ngx\\_mail\\_handler.c](#)
- [ngx\\_mail\\_imap\\_handler.c](#)
- [ngx\\_mail\\_imap\\_module.c](#)
- [ngx\\_mail\\_imap\\_module.h](#)
- [ngx\\_mail\\_parse.c](#)
- [ngx\\_mail\\_pop3\\_handler.c](#)
- [ngx\\_mail\\_pop3\\_module.c](#)
- [ngx\\_mail\\_pop3\\_module.h](#)
- [ngx\\_mail\\_proxy\\_module.c](#)
- [ngx\\_mail\\_smtp\\_handler.c](#)
- [ngx\\_mail\\_smtp\\_module.c](#)
- [ngx\\_mail\\_smtp\\_module.h](#)
- [ngx\\_mail\\_ssl\\_module.c](#)
- [ngx\\_mail\\_ssl\\_module.h](#)



## src/mail/nginx\_mail.c - nginx-1.7.10

### Global variables defined

- [ngx\\_mail\\_commands](#)
- [ngx\\_mail\\_max\\_module](#)
- [ngx\\_mail\\_module](#)
- [ngx\\_mail\\_module\\_ctx](#)

### Functions defined

- [ngx\\_mail\\_add\\_addr](#)
- [ngx\\_mail\\_add\\_addr6](#)
- [ngx\\_mail\\_add\\_ports](#)
- [ngx\\_mail\\_block](#)
- [ngx\\_mail\\_cmp\\_conf\\_addr](#)
- [ngx\\_mail\\_optimize\\_servers](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11 #include <ngx_mail.h>
12
13
14 static char *ngx_mail_block(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
15 static ngx_int_t ngx_mail_add_ports(ngx_conf_t *cf, ngx_array_t *ports,
16     ngx_mail_listen_t *listen);
17 static char *ngx_mail_optimize_servers(ngx_conf_t *cf, ngx_array_t *ports);
18 static ngx_int_t ngx_mail_add_addr(ngx_conf_t *cf, ngx_mail_port_t *mport,
19     ngx_mail_conf_addr_t *addr);
20 #if (NGX_HAVE_INET6)
21 static ngx_int_t ngx_mail_add_addr6(ngx_conf_t *cf, ngx_mail_port_t *mport,
22     ngx_mail_conf_addr_t *addr);
23 #endif
24 static ngx_int_t ngx_mail_cmp_conf_addr(const void *one, const void *two);
25
26
27 ngx_uint_t ngx_mail_max_module;
28
29
30 static ngx_command_t ngx_mail_commands[] = {
31
32     { ngx_string("mail"),
33       NGX_MAIN_CONF|NGX_CONF_BLOCK|NGX_CONF_NOARGS,
34       ngx_mail_block,
35       0,
36       0,
37       NULL },
38
```

```

39     { ngx_string("imap"),
40       NGX_MAIN_CONF|NGX_CONF_BLOCK|NGX_CONF_NOARGS,
41       ngx_mail_block,
42       0,
43       0,
44       NULL },
45
46     ngx_null_command
47 };
48
49
50 static ngx_core_module_t  ngx_mail_module_ctx = {
51     ngx_string("mail"),
52     NULL,
53     NULL
54 };
55
56
57 ngx_module_t  ngx_mail_module = {
58     NGX_MODULE_V1,
59     &ngx_mail_module_ctx,          /* module context */
60     ngx_mail_commands,           /* module directives */
61     NGX_CORE_MODULE,             /* module type */
62     NULL,                         /* init master */
63     NULL,                         /* init module */
64     NULL,                         /* init process */
65     NULL,                         /* init thread */
66     NULL,                         /* exit thread */
67     NULL,                         /* exit process */
68     NULL,                         /* exit master */
69     NGX_MODULE_V1_PADDING
70 };
71
72
73 static char *
74 ngx_mail_block(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
75 {
76     char          *rv;
77     ngx_uint_t    i, m, mi, s;
78     ngx_conf_t    pcf;
79     ngx_array_t   ports;
80     ngx_mail_listen_t  *listen;
81     ngx_mail_module_t  *module;
82     ngx_mail_conf_ctx_t  *ctx;
83     ngx_mail_core_srv_conf_t  **cscfp;
84     ngx_mail_core_main_conf_t  *cmcf;
85
86     if (cmd->name.data[0] == 'i') {
87         ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
88             "the \"imap\" directive is deprecated, "
89             "use the \"mail\" directive instead");
90     }
91
92     /* the main mail context */
93
94     ctx = ngx_palloc(cf->pool, sizeof(ngx_mail_conf_ctx_t));
95     if (ctx == NULL) {
96         return NGX_CONF_ERROR;
97     }
98
99     *(ngx_mail_conf_ctx_t **) conf = ctx;
100
101     /* count the number of the http modules and set up their indices */
102
103     ngx_mail_max_module = 0;
104     for (m = 0; ngx_modules[m]; m++) {
105         if (ngx_modules[m]->type != NGX_MAIL_MODULE) {
106             continue;
107         }
108
109         ngx_modules[m]->ctx_index = ngx_mail_max_module++;
110     }
111
112
113     /* the mail main_conf context, it is the same in the all mail contexts */
114

```

```

115 ctx->main_conf = ngx_palloc(cf->pool,
116                          sizeof(void *) * ngx_mail_max_module);
117 if (ctx->main_conf == NULL) {
118     return NGX_CONF_ERROR;
119 }
120
121
122 /*
123  * the mail null srv_conf context, it is used to merge
124  * the server{}s' srv_conf's
125  */
126
127 ctx->srv_conf = ngx_palloc(cf->pool, sizeof(void *) * ngx_mail_max_module);
128 if (ctx->srv_conf == NULL) {
129     return NGX_CONF_ERROR;
130 }
131
132
133 /*
134  * create the main_conf's and the null srv_conf's of the all mail modules
135  */
136
137 for (m = 0; ngx_modules[m]; m++) {
138     if (ngx_modules[m]->type != NGX_MAIL_MODULE) {
139         continue;
140     }
141
142     module = ngx_modules[m]->ctx;
143     mi = ngx_modules[m]->ctx_index;
144
145     if (module->create_main_conf) {
146         ctx->main_conf[mi] = module->create_main_conf(cf);
147         if (ctx->main_conf[mi] == NULL) {
148             return NGX_CONF_ERROR;
149         }
150     }
151
152     if (module->create_srv_conf) {
153         ctx->srv_conf[mi] = module->create_srv_conf(cf);
154         if (ctx->srv_conf[mi] == NULL) {
155             return NGX_CONF_ERROR;
156         }
157     }
158 }
159
160
161 /* parse inside the mail{} block */
162
163 pcf = *cf;
164 cf->ctx = ctx;
165
166 cf->module_type = NGX_MAIL_MODULE;
167 cf->cmd_type = NGX_MAIL_MAIN_CONF;
168 rv = ngx_conf_parse(cf, NULL);
169
170 if (rv != NGX_CONF_OK) {
171     *cf = pcf;
172     return rv;
173 }
174
175
176 /* init mail{} main_conf's, merge the server{}s' srv_conf's */
177
178 cmcf = ctx->main_conf[ngx_mail_core_module.ctx_index];
179 cscfp = cmcf->servers.elts;
180
181 for (m = 0; ngx_modules[m]; m++) {
182     if (ngx_modules[m]->type != NGX_MAIL_MODULE) {
183         continue;
184     }
185
186     module = ngx_modules[m]->ctx;
187     mi = ngx_modules[m]->ctx_index;
188
189     /* init mail{} main_conf's */
190

```

```

191     cf->ctx = ctx;
192
193     if (module->init_main_conf) {
194         rv = module->init_main_conf(cf, ctx->main_conf[mi]);
195         if (rv != NGX\_CONF\_OK) {
196             *cf = pcf;
197             return rv;
198         }
199     }
200
201     for (s = 0; s < cmcf->servers.nelts; s++) {
202
203         /* merge the server{s}' srv_conf's */
204
205         cf->ctx = cscfp[s]->ctx;
206
207         if (module->merge_srv_conf) {
208             rv = module->merge_srv_conf(cf,
209                                     ctx->srv_conf[mi],
210                                     cscfp[s]->ctx->srv_conf[mi]);
211             if (rv != NGX\_CONF\_OK) {
212                 *cf = pcf;
213                 return rv;
214             }
215         }
216     }
217 }
218
219 *cf = pcf;
220
221
222 if (ngx\_array\_init(&ports, cf->temp_pool, 4, sizeof\(ngx\_mail\_conf\_port\_t\))
223     != NGX\_OK)
224 {
225     return NGX\_CONF\_ERROR;
226 }
227
228 listen = cmcf->listen.elts;
229
230 for (i = 0; i < cmcf->listen.nelts; i++) {
231     if (ngx\_mail\_add\_ports(cf, &ports, &listen[i]) != NGX\_OK) {
232         return NGX\_CONF\_ERROR;
233     }
234 }
235
236 return ngx\_mail\_optimize\_servers(cf, &ports);
237 }
238
239
240 static ngx\_int\_t
241 ngx\_mail\_add\_ports(ngx\_conf\_t *cf, ngx\_array\_t *ports,
242                   ngx\_mail\_listen\_t *listen)
243 {
244     in\_port\_t          p;
245     ngx\_uint\_t         i;
246     struct sockaddr    *sa;
247     struct sockaddr_in *sin;
248     ngx\_mail\_conf\_port\_t *port;
249     ngx\_mail\_conf\_addr\_t *addr;
250     #if (NGX\_HAVE\_INET6)
251     struct sockaddr_in6 *sin6;
252 #endif
253
254     sa = (struct sockaddr *) &listen->sockaddr;
255
256     switch (sa->sa_family) {
257
258     #if (NGX\_HAVE\_INET6)
259     case AF_INET6:
260         sin6 = (struct sockaddr_in6 *) sa;
261         p = sin6->sin6_port;
262         break;
263     #endif
264
265     #if (NGX\_HAVE\_UNIX\_DOMAIN)
266     case AF_UNIX:

```

```

267     p = 0;
268     break;
269 #endif
270
271     default: /* AF_INET */
272         sin = (struct sockaddr_in *) sa;
273         p = sin->sin_port;
274         break;
275     }
276
277     port = ports->elts;
278     for (i = 0; i < ports->nelts; i++) {
279         if (p == port[i].port && sa->sa_family == port[i].family) {
280
281             /* a port is already in the port list */
282
283             port = &port[i];
284             goto found;
285         }
286     }
287
288     /* add a port to the port list */
289
290     port = ngx_array_push(ports);
291     if (port == NULL) {
292         return NGX_ERROR;
293     }
294
295     port->family = sa->sa_family;
296     port->port = p;
297
298     if (ngx_array_init(&port->addrs, cf->temp_pool, 2,
299                     sizeof(ngx_mail_conf_addr_t))
300         != NGX_OK)
301     {
302         return NGX_ERROR;
303     }
304
305 found:
306
307     addr = ngx_array_push(&port->addrs);
308     if (addr == NULL) {
309         return NGX_ERROR;
310     }
311
312     addr->sockaddr = (struct sockaddr *) &listen->sockaddr;
313     addr->socklen = listen->socklen;
314     addr->ctx = listen->ctx;
315     addr->bind = listen->bind;
316     addr->wildcard = listen->wildcard;
317     addr->so_keepalive = listen->so_keepalive;
318 #if (NGX_HAVE_KEEPALIVE_TUNABLE)
319     addr->tcp_keepidle = listen->tcp_keepidle;
320     addr->tcp_keepintvl = listen->tcp_keepintvl;
321     addr->tcp_keepcnt = listen->tcp_keepcnt;
322 #endif
323 #if (NGX_MAIL_SSL)
324     addr->ssl = listen->ssl;
325 #endif
326 #if (NGX_HAVE_INET6 && defined IPV6_V6ONLY)
327     addr->ipv6only = listen->ipv6only;
328 #endif
329
330     return NGX_OK;
331 }
332
333
334 static char *
335 ngx_mail_optimize_servers(ngx_conf_t *cf, ngx_array_t *ports)
336 {
337     ngx_uint_t          i, p, last, bind_wildcard;
338     ngx_listening_t    *ls;
339     ngx_mail_port_t    *mport;
340     ngx_mail_conf_port_t *port;
341     ngx_mail_conf_addr_t *addr;
342

```

```

343 port = ports->elts;
344 for (p = 0; p < ports->nelts; p++) {
345
346     ngx_sort(port[p].addrs.elts, (size_t) port[p].addrs.nelts,
347         sizeof(ngx_mail_conf_addr_t), ngx_mail_cmp_conf_addrs);
348
349     addr = port[p].addrs.elts;
350     last = port[p].addrs.nelts;
351
352     /*
353      * if there is the binding to the "*:port" then we need to bind()
354      * to the "*:port" only and ignore the other bindings
355      */
356
357     if (addr[last - 1].wildcard) {
358         addr[last - 1].bind = 1;
359         bind_wildcard = 1;
360
361     } else {
362         bind_wildcard = 0;
363     }
364
365     i = 0;
366
367     while (i < last) {
368
369         if (bind_wildcard && !addr[i].bind) {
370             i++;
371             continue;
372         }
373
374         ls = ngx_create_listening(cf, addr[i].sockaddr, addr[i].socklen);
375         if (ls == NULL) {
376             return NGX_CONF_ERROR;
377         }
378
379         ls->addr_ntop = 1;
380         ls->handler = ngx_mail_init_connection;
381         ls->pool_size = 256;
382
383         /* TODO: error_log directive */
384         ls->logp = &cf->cycle->new_log;
385         ls->log.data = &ls->addr_text;
386         ls->log.handler = ngx_accept_log_error;
387
388         ls->keepalive = addr[i].so_keepalive;
389 #if (NGX_HAVE_KEEPALIVE_TUNABLE)
390         ls->keepidle = addr[i].tcp_keepidle;
391         ls->keepintvl = addr[i].tcp_keepintvl;
392         ls->keepcnt = addr[i].tcp_keepcnt;
393 #endif
394
395 #if (NGX_HAVE_INET6 && defined IPV6_V6ONLY)
396         ls->ipv6only = addr[i].ipv6only;
397 #endif
398
399         mport = ngx_palloc(cf->pool, sizeof(ngx_mail_port_t));
400         if (mport == NULL) {
401             return NGX_CONF_ERROR;
402         }
403
404         ls->servers = mport;
405
406         if (i == last - 1) {
407             mport->naddrs = last;
408
409         } else {
410             mport->naddrs = 1;
411             i = 0;
412         }
413
414         switch (ls->sockaddr->sa_family) {
415 #if (NGX_HAVE_INET6)
416             case AF_INET6:
417                 if (ngx_mail_add_addrs6(cf, mport, addr) != NGX_OK) {
418                     return NGX_CONF_ERROR;

```

```

419     }
420     break;
421 #endif
422     default: /* AF_INET */
423     if (ngx_mail_add_addrs(cf, mport, addr) != NGX_OK) {
424         return NGX_CONF_ERROR;
425     }
426     break;
427 }
428
429     addr++;
430     last--;
431 }
432 }
433
434     return NGX_CONF_OK;
435 }
436
437
438 static ngx_int_t
439 ngx_mail_add_addrs(ngx_conf_t *cf, ngx_mail_port_t *mport,
440 ngx_mail_conf_addr_t *addr)
441 {
442     u_char          *p;
443     size_t          len;
444     ngx_uint_t      i;
445     ngx_mail_in_addr_t *addrs;
446     struct sockaddr_in *sin;
447     u_char          buf[NGX_SOCKADDR_STRLEN];
448
449     mport->addrs = ngx_palloc(cf->pool,
450                             mport->naddrs * sizeof(ngx_mail_in_addr_t));
451     if (mport->addrs == NULL) {
452         return NGX_ERROR;
453     }
454
455     addrs = mport->addrs;
456
457     for (i = 0; i < mport->naddrs; i++) {
458
459         sin = (struct sockaddr_in *) addr[i].sockaddr;
460         addrs[i].addr = sin->sin_addr.s_addr;
461
462         addrs[i].conf.ctx = addr[i].ctx;
463 #if (NGX_MAIL_SSL)
464         addrs[i].conf.ssl = addr[i].ssl;
465 #endif
466
467         len = ngx_sock_ntop(addr[i].sockaddr, addr[i].socklen, buf,
468                             NGX_SOCKADDR_STRLEN, 1);
469
470         p = ngx_pnalloc(cf->pool, len);
471         if (p == NULL) {
472             return NGX_ERROR;
473         }
474
475         ngx_memcpy(p, buf, len);
476
477         addrs[i].conf.addr_text.len = len;
478         addrs[i].conf.addr_text.data = p;
479     }
480
481     return NGX_OK;
482 }
483
484
485 #if (NGX_HAVE_INET6)
486
487 static ngx_int_t
488 ngx_mail_add_addrs6(ngx_conf_t *cf, ngx_mail_port_t *mport,
489 ngx_mail_conf_addr_t *addr)
490 {
491     u_char          *p;
492     size_t          len;
493     ngx_uint_t      i;
494     ngx_mail_in6_addr_t *addrs6;

```

```

495     struct sockaddr_in6 *sin6;
496     u_char                buf[NGX_SOCKADDR_STRLEN];
497
498     mport->addrs = ngx_palloc(cf->pool,
499                             mport->naddrs * sizeof(ngx_mail_in6_addr_t));
500     if (mport->addrs == NULL) {
501         return NGX_ERROR;
502     }
503
504     addrs6 = mport->addrs;
505
506     for (i = 0; i < mport->naddrs; i++) {
507
508         sin6 = (struct sockaddr_in6 *) addr[i].sockaddr;
509         addrs6[i].addr6 = sin6->sin6_addr;
510
511         addrs6[i].conf.ctx = addr[i].ctx;
512         #if (NGX_MAIL_SSL)
513         addrs6[i].conf.ssl = addr[i].ssl;
514         #endif
515
516         len = ngx_sock_ntop(addr[i].sockaddr, addr[i].socklen, buf,
517                             NGX_SOCKADDR_STRLEN, 1);
518
519         p = ngx_pnalloc(cf->pool, len);
520         if (p == NULL) {
521             return NGX_ERROR;
522         }
523
524         ngx_memcpy(p, buf, len);
525
526         addrs6[i].conf.addr_text.len = len;
527         addrs6[i].conf.addr_text.data = p;
528     }
529
530     return NGX_OK;
531 }
532
533 #endif
534
535
536 static ngx_int_t
537 ngx_mail_cmp_conf_addrs(const void *one, const void *two)
538 {
539     ngx_mail_conf_addr_t *first, *second;
540
541     first = (ngx_mail_conf_addr_t *) one;
542     second = (ngx_mail_conf_addr_t *) two;
543
544     if (first->wildcard) {
545         /* a wildcard must be the last resort, shift it to the end */
546         return 1;
547     }
548
549     if (second->wildcard) {
550         /* a wildcard must be the last resort, shift it to the end */
551         return -1;
552     }
553
554     if (first->bind && !second->bind) {
555         /* shift explicit bind()ed addresses to the start */
556         return -1;
557     }
558
559     if (!first->bind && second->bind) {
560         /* shift explicit bind()ed addresses to the start */
561         return 1;
562     }
563
564     /* do not sort by default */
565
566     return 0;
567 }

```



## src/mail/nginx\_mail.h - nginx-1.7.10

### Data types defined

- [ngx\\_imap\\_state\\_e](#)
- [ngx\\_mail\\_addr\\_conf\\_t](#)
- [ngx\\_mail\\_auth\\_state\\_pt](#)
- [ngx\\_mail\\_conf\\_addr\\_t](#)
- [ngx\\_mail\\_conf\\_ctx\\_t](#)
- [ngx\\_mail\\_conf\\_port\\_t](#)
- [ngx\\_mail\\_core\\_main\\_conf\\_t](#)
- [ngx\\_mail\\_core\\_srv\\_conf\\_t](#)
- [ngx\\_mail\\_in6\\_addr\\_t](#)
- [ngx\\_mail\\_in\\_addr\\_t](#)
- [ngx\\_mail\\_init\\_protocol\\_pt](#)
- [ngx\\_mail\\_init\\_session\\_pt](#)
- [ngx\\_mail\\_listen\\_t](#)
- [ngx\\_mail\\_log\\_ctx\\_t](#)
- [ngx\\_mail\\_module\\_t](#)
- [ngx\\_mail\\_parse\\_command\\_pt](#)
- [ngx\\_mail\\_port\\_t](#)
- [ngx\\_mail\\_protocol\\_s](#)
- [ngx\\_mail\\_protocol\\_t](#)
- [ngx\\_mail\\_proxy\\_ctx\\_t](#)
- [ngx\\_mail\\_session\\_t](#)
- [ngx\\_pop3\\_state\\_e](#)
- [ngx\\_smtp\\_state\\_e](#)

### Macros defined

- [NGX\\_IMAP\\_AUTHENTICATE](#)
- [NGX\\_IMAP\\_CAPABILITY](#)
- [NGX\\_IMAP\\_LOGIN](#)
- [NGX\\_IMAP\\_LOGOUT](#)
- [NGX\\_IMAP\\_NEXT](#)

- [NGX\\_IMAP\\_NOOP](#)
- [NGX\\_IMAP\\_STARTTLS](#)
- [NGX\\_MAIL\\_AUTH\\_APOP](#)
- [NGX\\_MAIL\\_AUTH\\_APOP\\_ENABLED](#)
- [NGX\\_MAIL\\_AUTH\\_CRAM\\_MD5](#)
- [NGX\\_MAIL\\_AUTH\\_CRAM\\_MD5\\_ENABLED](#)
- [NGX\\_MAIL\\_AUTH\\_LOGIN](#)
- [NGX\\_MAIL\\_AUTH\\_LOGIN\\_ENABLED](#)
- [NGX\\_MAIL\\_AUTH\\_LOGIN\\_USERNAME](#)
- [NGX\\_MAIL\\_AUTH\\_NONE](#)
- [NGX\\_MAIL\\_AUTH\\_NONE\\_ENABLED](#)
- [NGX\\_MAIL\\_AUTH\\_PLAIN](#)
- [NGX\\_MAIL\\_AUTH\\_PLAIN\\_ENABLED](#)
- [NGX\\_MAIL\\_IMAP\\_PROTOCOL](#)
- [NGX\\_MAIL\\_MAIN\\_CONF](#)
- [NGX\\_MAIL\\_MAIN\\_CONF\\_OFFSET](#)
- [NGX\\_MAIL\\_MODULE](#)
- [NGX\\_MAIL\\_PARSE\\_INVALID\\_COMMAND](#)
- [NGX\\_MAIL\\_POP3\\_PROTOCOL](#)
- [NGX\\_MAIL\\_SMTP\\_PROTOCOL](#)
- [NGX\\_MAIL\\_SRV\\_CONF](#)
- [NGX\\_MAIL\\_SRV\\_CONF\\_OFFSET](#)
- [NGX\\_POP3\\_APOP](#)
- [NGX\\_POP3\\_AUTH](#)
- [NGX\\_POP3\\_CAPA](#)
- [NGX\\_POP3\\_DELE](#)
- [NGX\\_POP3\\_LIST](#)
- [NGX\\_POP3\\_NOOP](#)
- [NGX\\_POP3\\_PASS](#)
- [NGX\\_POP3\\_QUIT](#)
- [NGX\\_POP3\\_RETR](#)
- [NGX\\_POP3\\_RSET](#)
- [NGX\\_POP3\\_STAT](#)

- [NGX\\_POP3\\_STLS](#)
- [NGX\\_POP3\\_TOP](#)
- [NGX\\_POP3\\_UIDL](#)
- [NGX\\_POP3\\_USER](#)
- [NGX\\_SMTP\\_AUTH](#)
- [NGX\\_SMTP\\_DATA](#)
- [NGX\\_SMTP\\_EHLO](#)
- [NGX\\_SMTP\\_EXPN](#)
- [NGX\\_SMTP\\_HELO](#)
- [NGX\\_SMTP\\_HELP](#)
- [NGX\\_SMTP\\_MAIL](#)
- [NGX\\_SMTP\\_NOOP](#)
- [NGX\\_SMTP\\_QUIT](#)
- [NGX\\_SMTP\\_RCPT](#)
- [NGX\\_SMTP\\_RSET](#)
- [NGX\\_SMTP\\_STARTTLS](#)
- [NGX\\_SMTP\\_VRFY](#)
- [\\_NGX\\_MAIL\\_H\\_INCLUDED\\_](#)
- [ngx\\_mail\\_conf\\_get\\_module\\_main\\_conf](#)
- [ngx\\_mail\\_conf\\_get\\_module\\_srv\\_conf](#)
- [ngx\\_mail\\_delete\\_ctx](#)
- [ngx\\_mail\\_get\\_module\\_ctx](#)
- [ngx\\_mail\\_get\\_module\\_main\\_conf](#)
- [ngx\\_mail\\_get\\_module\\_srv\\_conf](#)
- [ngx\\_mail\\_set\\_ctx](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_MAIL_H_INCLUDED_
9 #define _NGX_MAIL_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14 #include <ngx_event.h>
15 #include <ngx_event_connect.h>
16
```

```

17 #if (NGX_MAIL_SSL)
18 #include <ngx_mail_ssl_module.h>
19 #endif
20
21
22
23 typedef struct {
24     void                **main_conf;
25     void                **srv_conf;
26 } ngx_mail_conf_ctx_t;
27
28
29 typedef struct {
30     u_char              sockaddr[NGX_SOCKADDRLEN];
31     socklen_t          socklen;
32
33     /* server ctx */
34     ngx_mail_conf_ctx_t *ctx;
35
36     unsigned            bind:1;
37     unsigned            wildcard:1;
38 #if (NGX_MAIL_SSL)
39     unsigned            ssl:1;
40 #endif
41 #if (NGX_HAVE_INET6 && defined IPV6_V6ONLY)
42     unsigned            ipv6only:1;
43 #endif
44     unsigned            so_keepalive:2;
45 #if (NGX_HAVE_KEEPALIVE_TUNABLE)
46     int                 tcp_keepidle;
47     int                 tcp_keepintvl;
48     int                 tcp_keepcnt;
49 #endif
50 } ngx_mail_listen_t;
51
52
53 typedef struct {
54     ngx_mail_conf_ctx_t *ctx;
55     ngx_str_t           addr_text;
56 #if (NGX_MAIL_SSL)
57     ngx_uint_t          ssl;    /* unsigned ssl:1; */
58 #endif
59 } ngx_mail_addr_conf_t;
60
61 typedef struct {
62     in_addr_t           addr;
63     ngx_mail_addr_conf_t conf;
64 } ngx_mail_in_addr_t;
65
66
67 #if (NGX_HAVE_INET6)
68
69 typedef struct {
70     struct in6_addr     addr6;
71     ngx_mail_addr_conf_t conf;
72 } ngx_mail_in6_addr_t;
73
74 #endif
75
76
77 typedef struct {
78     /* ngx_mail_in_addr_t or ngx_mail_in6_addr_t */
79     void                *addrs;
80     ngx_uint_t          naddrs;
81 } ngx_mail_port_t;
82
83
84 typedef struct {
85     int                 family;
86     in_port_t           port;
87     ngx_array_t         addrs;    /* array of ngx_mail_conf_addr_t */
88 } ngx_mail_conf_port_t;
89
90
91 typedef struct {
92     struct sockaddr     *sockaddr;

```

```

93     socklen_t                socklen;
94
95     ngx\_mail\_conf\_ctx\_t        *ctx;
96
97     unsigned                 bind:1;
98     unsigned                 wildcard:1;
99 #if (NGX_MAIL_SSL)
100     unsigned                 ssl:1;
101 #endif
102 #if (NGX_HAVE_INET6 && defined IPV6_V6ONLY)
103     unsigned                 ipv6only:1;
104 #endif
105     unsigned                 so_keepalive:2;
106 #if (NGX_HAVE_KEEPALIVE_TUNABLE)
107     int                      tcp_keepidle;
108     int                      tcp_keepintvl;
109     int                      tcp_keepcnt;
110 #endif
111 } ngx_mail_conf_addr_t;
112
113
114 typedef struct {
115     ngx\_array\_t                servers;        /* ngx\_mail\_core\_srv\_conf\_t */
116     ngx\_array\_t                listen;       /* ngx\_mail\_listen\_t */
117 } ngx_mail_core_main_conf_t;
118
119
120 #define NGX_MAIL_POP3_PROTOCOL  0
121 #define NGX_MAIL_IMAP_PROTOCOL  1
122 #define NGX_MAIL_SMTP_PROTOCOL  2
123
124
125 typedef struct ngx\_mail\_protocol\_s ngx_mail_protocol_t;
126
127
128 typedef struct {
129     ngx\_mail\_protocol\_t        *protocol;
130
131     ngx\_msec\_t                 timeout;
132     ngx\_msec\_t                 resolver_timeout;
133
134     ngx\_flag\_t                 so_keepalive;
135
136     ngx\_str\_t                  server_name;
137
138     u_char                    *file_name;
139     ngx\_int\_t                   line;
140
141     ngx\_resolver\_t             *resolver;
142
143     /* server ctx */
144     ngx\_mail\_conf\_ctx\_t        *ctx;
145 } ngx_mail_core_srv_conf_t;
146
147
148 typedef enum {
149     ngx_pop3_start = 0,
150     ngx_pop3_user,
151     ngx_pop3_passwd,
152     ngx_pop3_auth_login_username,
153     ngx_pop3_auth_login_password,
154     ngx_pop3_auth_plain,
155     ngx_pop3_auth_cram_md5
156 } ngx_pop3_state_e;
157
158
159 typedef enum {
160     ngx_imap_start = 0,
161     ngx_imap_auth_login_username,
162     ngx_imap_auth_login_password,
163     ngx_imap_auth_plain,
164     ngx_imap_auth_cram_md5,
165     ngx_imap_login,
166     ngx_imap_user,
167     ngx_imap_passwd
168 } ngx_imap_state_e;

```

```

169
170
171 typedef enum {
172     ngx_smtp_start = 0,
173     ngx_smtp_auth_login_username,
174     ngx_smtp_auth_login_password,
175     ngx_smtp_auth_plain,
176     ngx_smtp_auth_cram_md5,
177     ngx_smtp_helo,
178     ngx_smtp_helo_xclient,
179     ngx_smtp_helo_from,
180     ngx_smtp_xclient,
181     ngx_smtp_xclient_from,
182     ngx_smtp_xclient_helo,
183     ngx_smtp_from,
184     ngx_smtp_to
185 } ngx_smtp_state_e;
186
187
188 typedef struct {
189     ngx\_peer\_connection\_t    upstream;
190     ngx\_buf\_t                *buffer;
191 } ngx_mail_proxy_ctx_t;
192
193
194 typedef struct {
195     uint32_t                signature;           /* "MAIL" */
196
197     ngx\_connection\_t        *connection;
198
199     ngx\_str\_t                out;
200     ngx\_buf\_t                *buffer;
201
202     void                    **ctx;
203     void                    **main_conf;
204     void                    **srv_conf;
205
206     ngx\_resolver\_ctx\_t      *resolver_ctx;
207
208     ngx\_mail\_proxy\_ctx\_t    *proxy;
209
210     ngx\_uint\_t              mail_state;
211
212     unsigned                protocol:3;
213     unsigned                blocked:1;
214     unsigned                quit:1;
215     unsigned                quoted:1;
216     unsigned                backslash:1;
217     unsigned                no_sync_literal:1;
218     unsigned                starttls:1;
219     unsigned                esmtp:1;
220     unsigned                auth_method:3;
221     unsigned                auth_wait:1;
222
223     ngx\_str\_t                login;
224     ngx\_str\_t                passwd;
225
226     ngx\_str\_t                salt;
227     ngx\_str\_t                tag;
228     ngx\_str\_t                tagged_line;
229     ngx\_str\_t                text;
230
231     ngx\_str\_t                *addr_text;
232     ngx\_str\_t                host;
233     ngx\_str\_t                smtp_helo;
234     ngx\_str\_t                smtp_from;
235     ngx\_str\_t                smtp_to;
236
237     ngx\_str\_t                cmd;
238
239     ngx\_uint\_t              command;
240     ngx\_array\_t            args;
241
242     ngx\_uint\_t              login_attempt;
243
244     /* used to parse POP3/IMAP/SMTP command */

```

```

245     ngx_uint_t         state;
246     u_char             *cmd_start;
247     u_char             *arg_start;
248     u_char             *arg_end;
249     ngx_uint_t         literal_len;
250 } ngx_mail_session_t;
251
252
253
254 typedef struct {
255     ngx_str_t           *client;
256     ngx_mail_session_t *session;
257 } ngx_mail_log_ctx_t;
258
259
260 #define NGX_POP3_USER           1
261 #define NGX_POP3_PASS           2
262 #define NGX_POP3_CAPA           3
263 #define NGX_POP3_QUIT           4
264 #define NGX_POP3_NOOP           5
265 #define NGX_POP3_STLS           6
266 #define NGX_POP3_APOP           7
267 #define NGX_POP3_AUTH           8
268 #define NGX_POP3_STAT           9
269 #define NGX_POP3_LIST           10
270 #define NGX_POP3_RETR           11
271 #define NGX_POP3_DELE           12
272 #define NGX_POP3_RSET           13
273 #define NGX_POP3_TOP            14
274 #define NGX_POP3_UIDL           15
275
276
277 #define NGX_IMAP_LOGIN           1
278 #define NGX_IMAP_LOGOUT          2
279 #define NGX_IMAP_CAPABILITY      3
280 #define NGX_IMAP_NOOP            4
281 #define NGX_IMAP_STARTTLS        5
282
283 #define NGX_IMAP_NEXT            6
284
285 #define NGX_IMAP_AUTHENTICATE    7
286
287
288 #define NGX_SMTP_HELO            1
289 #define NGX_SMTP_EHLO            2
290 #define NGX_SMTP_AUTH            3
291 #define NGX_SMTP_QUIT            4
292 #define NGX_SMTP_NOOP            5
293 #define NGX_SMTP_MAIL            6
294 #define NGX_SMTP_RSET            7
295 #define NGX_SMTP_RCPT            8
296 #define NGX_SMTP_DATA            9
297 #define NGX_SMTP_VRFY            10
298 #define NGX_SMTP_EXPN            11
299 #define NGX_SMTP_HELP            12
300 #define NGX_SMTP_STARTTLS        13
301
302
303 #define NGX_MAIL_AUTH_PLAIN       0
304 #define NGX_MAIL_AUTH_LOGIN       1
305 #define NGX_MAIL_AUTH_LOGIN_USERNAME 2
306 #define NGX_MAIL_AUTH_APOP        3
307 #define NGX_MAIL_AUTH_CRAM_MD5    4
308 #define NGX_MAIL_AUTH_NONE        5
309
310
311 #define NGX_MAIL_AUTH_PLAIN_ENABLED 0x0002
312 #define NGX_MAIL_AUTH_LOGIN_ENABLED 0x0004
313 #define NGX_MAIL_AUTH_APOP_ENABLED 0x0008
314 #define NGX_MAIL_AUTH_CRAM_MD5_ENABLED 0x0010
315 #define NGX_MAIL_AUTH_NONE_ENABLED 0x0020
316
317
318 #define NGX_MAIL_PARSE_INVALID_COMMAND 20
319
320

```

```

321 typedef void (*ngx_mail_init_session_pt)(ngx_mail_session_t *s,
322     ngx_connection_t *c);
323 typedef void (*ngx_mail_init_protocol_pt)(ngx_event_t *rev);
324 typedef void (*ngx_mail_auth_state_pt)(ngx_event_t *rev);
325 typedef ngx_int_t (*ngx_mail_parse_command_pt)(ngx_mail_session_t *s);
326
327
328 struct ngx_mail_protocol_s {
329     ngx_str_t      name;
330     in_port_t      port[4];
331     ngx_uint_t     type;
332
333     ngx_mail_init_session_pt  init_session;
334     ngx_mail_init_protocol_pt init_protocol;
335     ngx_mail_parse_command_pt parse_command;
336     ngx_mail_auth_state_pt   auth_state;
337
338     ngx_str_t      internal_server_error;
339 };
340
341
342 typedef struct {
343     ngx_mail_protocol_t *protocol;
344
345     void (*create_main_conf)(ngx_conf_t *cf);
346     char (*init_main_conf)(ngx_conf_t *cf, void *conf);
347
348     void (*create_srv_conf)(ngx_conf_t *cf);
349     char (*merge_srv_conf)(ngx_conf_t *cf, void *prev,
350         void *conf);
351 } ngx_mail_module_t;
352
353
354 #define NGX_MAIL_MODULE      0x4C49414D    /* "MAIL" */
355
356 #define NGX_MAIL_MAIN_CONF   0x02000000
357 #define NGX_MAIL_SRV_CONF   0x04000000
358
359
360 #define NGX_MAIL_MAIN_CONF_OFFSET  offsetof(ngx_mail_conf_ctx_t, main_conf)
361 #define NGX_MAIL_SRV_CONF_OFFSET  offsetof(ngx_mail_conf_ctx_t, srv_conf)
362
363
364 #define ngx_mail_get_module_ctx(s, module)    (s)->ctx[module.ctx_index]
365 #define ngx_mail_set_ctx(s, c, module)       s->ctx[module.ctx_index] = c;
366 #define ngx_mail_delete_ctx(s, module)       s->ctx[module.ctx_index] = NULL;
367
368
369 #define ngx_mail_get_module_main_conf(s, module) \
370     (s)->main_conf[module.ctx_index]
371 #define ngx_mail_get_module_srv_conf(s, module) (s)->srv_conf[module.ctx_index]
372
373 #define ngx_mail_conf_get_module_main_conf(cf, module) \
374     ((ngx_mail_conf_ctx_t *) cf->ctx)->main_conf[module.ctx_index]
375 #define ngx_mail_conf_get_module_srv_conf(cf, module) \
376     ((ngx_mail_conf_ctx_t *) cf->ctx)->srv_conf[module.ctx_index]
377
378
379 #if (NGX_MAIL_SSL)
380 void ngx_mail_starttls_handler(ngx_event_t *rev);
381 ngx_int_t ngx_mail_starttls_only(ngx_mail_session_t *s, ngx_connection_t *c);
382 #endif
383
384
385 void ngx_mail_init_connection(ngx_connection_t *c);
386
387 ngx_int_t ngx_mail_salt(ngx_mail_session_t *s, ngx_connection_t *c,
388     ngx_mail_core_srv_conf_t *cscf);
389 ngx_int_t ngx_mail_auth_plain(ngx_mail_session_t *s, ngx_connection_t *c,
390     ngx_uint_t n);
391 ngx_int_t ngx_mail_auth_login_username(ngx_mail_session_t *s,
392     ngx_connection_t *c, ngx_uint_t n);
393 ngx_int_t ngx_mail_auth_login_password(ngx_mail_session_t *s,
394     ngx_connection_t *c);
395 ngx_int_t ngx_mail_auth_cram_md5_salt(ngx_mail_session_t *s,
396     ngx_connection_t *c, char *prefix, size_t len);

```



```
397 ngx\_int\_t ngx\_mail\_auth\_cram\_md5(ngx\_mail\_session\_t *s, ngx\_connection\_t *c);
398 ngx\_int\_t ngx\_mail\_auth\_parse(ngx\_mail\_session\_t *s, ngx\_connection\_t *c);
399
400 void ngx\_mail\_send(ngx\_event\_t *wev);
401 ngx\_int\_t ngx\_mail\_read\_command(ngx\_mail\_session\_t *s, ngx\_connection\_t *c);
402 void ngx\_mail\_auth(ngx\_mail\_session\_t *s, ngx\_connection\_t *c);
403 void ngx\_mail\_close\_connection(ngx\_connection\_t *c);
404 void ngx\_mail\_session\_internal\_server\_error(ngx\_mail\_session\_t *s);
405 u_char *ngx\_mail\_log\_error(ngx\_log\_t *log, u_char *buf, size_t len);
406
407
408 char *ngx\_mail\_capabilities(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf);
409
410
411 /* STUB */
412 void ngx\_mail\_proxy\_init(ngx\_mail\_session\_t *s, ngx\_addr\_t *peer);
413 void ngx\_mail\_auth\_http\_init(ngx\_mail\_session\_t *s);
414 /**/
415
416
417 extern ngx\_uint\_t ngx\_mail\_max\_module;
418 extern ngx\_module\_t ngx\_mail\_core\_module;
419
420
421 #endif /* NGX_MAIL_H_INCLUDED */
```

[One Level Up](#)

[Top Level](#)

## src/mail/nginx\_mail\_handler.c - nginx-1.7.10

### Functions defined

- [ngx\\_mail\\_auth](#)
- [ngx\\_mail\\_auth\\_cram\\_md5](#)
- [ngx\\_mail\\_auth\\_cram\\_md5\\_salt](#)
- [ngx\\_mail\\_auth\\_login\\_password](#)
- [ngx\\_mail\\_auth\\_login\\_username](#)
- [ngx\\_mail\\_auth\\_plain](#)
- [ngx\\_mail\\_close\\_connection](#)
- [ngx\\_mail\\_init\\_connection](#)
- [ngx\\_mail\\_init\\_session](#)
- [ngx\\_mail\\_log\\_error](#)
- [ngx\\_mail\\_read\\_command](#)
- [ngx\\_mail\\_salt](#)
- [ngx\\_mail\\_send](#)
- [ngx\\_mail\\_session\\_internal\\_server\\_error](#)
- [ngx\\_mail\\_ssl\\_handshake\\_handler](#)
- [ngx\\_mail\\_ssl\\_init\\_connection](#)
- [ngx\\_mail\\_starttls\\_handler](#)
- [ngx\\_mail\\_starttls\\_only](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11 #include <ngx_mail.h>
12
13
14 static void ngx_mail_init_session(ngx_connection_t *c);
15
16 #if (NGX_MAIL_SSL)
17 static void ngx_mail_ssl_init_connection(ngx_ssl_t *ssl, ngx_connection_t *c);
18 static void ngx_mail_ssl_handshake_handler(ngx_connection_t *c);
19 #endif
20
21
22 void
23 ngx_mail_init_connection(ngx_connection_t *c)
24 {
```

```

25     size_t          len;
26     ngx_uint_t     i;
27     ngx_mail_port_t *port;
28     struct sockaddr *sa;
29     struct sockaddr_in *sin;
30     ngx_mail_log_ctx_t *ctx;
31     ngx_mail_in_addr_t *addr;
32     ngx_mail_session_t *s;
33     ngx_mail_addr_conf_t *addr_conf;
34     u_char          text[NGX_SOCKADDR_STRLEN];
35 #if (NGX_HAVE_INET6)
36     struct sockaddr_in6 *sin6;
37     ngx_mail_in6_addr_t *addr6;
38 #endif
39
40
41     /* find the server configuration for the address:port */
42
43     port = c->listening->servers;
44
45     if (port->naddrs > 1) {
46
47         /*
48          * There are several addresses on this port and one of them
49          * is the "*:port" wildcard so getsockname() is needed to determine
50          * the server address.
51          *
52          * AcceptEx() already gave this address.
53          */
54
55         if (ngx_connection_local_sockaddr(c, NULL, 0) != NGX_OK) {
56             ngx_mail_close_connection(c);
57             return;
58         }
59
60         sa = c->local_sockaddr;
61
62         switch (sa->sa_family) {
63
64 #if (NGX_HAVE_INET6)
65             case AF_INET6:
66                 sin6 = (struct sockaddr_in6 *) sa;
67
68                 addr6 = port->addrs;
69
70                 /* the last address is "*" */
71
72                 for (i = 0; i < port->naddrs - 1; i++) {
73                     if (ngx_memcmp(&addr6[i].addr6, &sin6->sin6_addr, 16) == 0) {
74                         break;
75                     }
76                 }
77
78                 addr_conf = &addr6[i].conf;
79
80                 break;
81 #endif
82
83             default: /* AF_INET */
84                 sin = (struct sockaddr_in *) sa;
85
86                 addr = port->addrs;
87
88                 /* the last address is "*" */
89
90                 for (i = 0; i < port->naddrs - 1; i++) {
91                     if (addr[i].addr == sin->sin_addr.s_addr) {
92                         break;
93                     }
94                 }
95
96                 addr_conf = &addr[i].conf;
97
98                 break;
99         }
100

```

```

101     } else {
102         switch (c->local_sockaddr->sa_family) {
103
104     #if (NGX_HAVE_INET6)
105         case AF_INET6:
106             addr6 = port->addrs;
107             addr_conf = &addr6[0].conf;
108             break;
109     #endif
110
111         default: /* AF_INET */
112             addr = port->addrs;
113             addr_conf = &addr[0].conf;
114             break;
115     }
116 }
117
118 s = ngx_palloc(c->pool, sizeof(ngx_mail_session_t));
119 if (s == NULL) {
120     ngx_mail_close_connection(c);
121     return;
122 }
123
124 s->signature = NGX_MAIL_MODULE;
125
126 s->main_conf = addr_conf->ctx->main_conf;
127 s->srv_conf = addr_conf->ctx->srv_conf;
128
129 s->addr_text = &addr_conf->addr_text;
130
131 c->data = s;
132 s->connection = c;
133
134 len = ngx_sock_ntop(c->sockaddr, c->socklen, text, NGX_SOCKADDR_STRLEN, 1);
135
136 ngx_log_error(NGX_LOG_INFO, c->log, 0, "%uA client %*s connected to %V",
137             c->number, len, text, s->addr_text);
138
139 ctx = ngx_palloc(c->pool, sizeof(ngx_mail_log_ctx_t));
140 if (ctx == NULL) {
141     ngx_mail_close_connection(c);
142     return;
143 }
144
145 ctx->client = &c->addr_text;
146 ctx->session = s;
147
148 c->log->connection = c->number;
149 c->log->handler = ngx_mail_log_error;
150 c->log->data = ctx;
151 c->log->action = "sending client greeting line";
152
153 c->log_error = NGX_ERROR_INFO;
154
155 #if (NGX_MAIL_SSL)
156 {
157     ngx_mail_ssl_conf_t *sslcf;
158
159     sslcf = ngx_mail_get_module_srv_conf(s, ngx_mail_ssl_module);
160
161     if (sslcf->enable) {
162         c->log->action = "SSL handshaking";
163
164         ngx_mail_ssl_init_connection(&sslcf->ssl, c);
165         return;
166     }
167
168     if (addr_conf->ssl) {
169         c->log->action = "SSL handshaking";
170
171         if (sslcf->ssl.ctx == NULL) {
172             ngx_log_error(NGX_LOG_ERR, c->log, 0,
173                 "no \"ssl_certificate\" is defined "
174                 "in server listening on SSL port");
175             ngx_mail_close_connection(c);
176

```

```

177         return;
178     }
179
180     ngx\_mail\_ssl\_init\_connection(&sslcf->ssl, c);
181     return;
182 }
183
184 }
185 #endif
186
187     ngx\_mail\_init\_session(c);
188 }
189
190
191 #if (NGX_MAIL_SSL)
192
193 void
194 ngx\_mail\_starttls\_handler(ngx\_event\_t *rev)
195 {
196     ngx\_connection\_t *c;
197     ngx\_mail\_session\_t *s;
198     ngx\_mail\_ssl\_conf\_t *sslcf;
199
200     c = rev->data;
201     s = c->data;
202     s->starttls = 1;
203
204     c->log->action = "in starttls state";
205
206     sslcf = ngx\_mail\_get\_module\_srv\_conf(s, ngx\_mail\_ssl\_module);
207
208     ngx\_mail\_ssl\_init\_connection(&sslcf->ssl, c);
209 }
210
211
212 static void
213 ngx\_mail\_ssl\_init\_connection(ngx\_ssl\_t *ssl, ngx\_connection\_t *c)
214 {
215     ngx\_mail\_session\_t *s;
216     ngx\_mail\_core\_srv\_conf\_t *cscf;
217
218     if (ngx\_ssl\_create\_connection(ssl, c, 0) == NGX\_ERROR) {
219         ngx\_mail\_close\_connection(c);
220         return;
221     }
222
223     if (ngx\_ssl\_handshake(c) == NGX\_AGAIN) {
224
225         s = c->data;
226
227         cscf = ngx\_mail\_get\_module\_srv\_conf(s, ngx\_mail\_core\_module);
228
229         ngx\_add\_timer(c->read, cscf->timeout);
230
231         c->ssl->handler = ngx\_mail\_ssl\_handshake\_handler;
232
233         return;
234     }
235
236     ngx\_mail\_ssl\_handshake\_handler(c);
237 }
238
239
240 static void
241 ngx\_mail\_ssl\_handshake\_handler(ngx\_connection\_t *c)
242 {
243     ngx\_mail\_session\_t *s;
244     ngx\_mail\_core\_srv\_conf\_t *cscf;
245
246     if (c->ssl->handshaked) {
247
248         s = c->data;
249
250         if (s->starttls) {
251             cscf = ngx\_mail\_get\_module\_srv\_conf(s, ngx\_mail\_core\_module);
252

```

```

253         c->read->handler = cscf->protocol->init_protocol;
254         c->write->handler = ngx_mail_send;
255
256         cscf->protocol->init_protocol(c->read);
257
258         return;
259     }
260
261     c->read->ready = 0;
262
263     ngx_mail_init_session(c);
264     return;
265 }
266
267 ngx_mail_close_connection(c);
268 }
269
270 #endif
271
272
273 static void
274 ngx_mail_init_session(ngx_connection_t *c)
275 {
276     ngx_mail_session_t *s;
277     ngx_mail_core_srv_conf_t *cscf;
278
279     s = c->data;
280
281     cscf = ngx_mail_get_module_srv_conf(s, ngx_mail_core_module);
282
283     s->protocol = cscf->protocol->type;
284
285     s->ctx = ngx_palloc(c->pool, sizeof(void *) * ngx_mail_max_module);
286     if (s->ctx == NULL) {
287         ngx_mail_session_internal_server_error(s);
288         return;
289     }
290
291     c->write->handler = ngx_mail_send;
292
293     cscf->protocol->init_session(s, c);
294 }
295
296
297 ngx_int_t
298 ngx_mail_salt(ngx_mail_session_t *s, ngx_connection_t *c,
299 ngx_mail_core_srv_conf_t *cscf)
300 {
301     s->salt.data = ngx_pnalloc(c->pool,
302                               sizeof("<18446744073709551616.@>" CRLF) - 1
303                               + NGX_TIME_T_LEN
304                               + cscf->server_name.len);
305     if (s->salt.data == NULL) {
306         return NGX_ERROR;
307     }
308
309     s->salt.len = ngx_sprintf(s->salt.data, "<%u1.%T@%V>" CRLF,
310                             ngx_random(), ngx_time(), &cscf->server_name)
311                 - s->salt.data;
312
313     return NGX_OK;
314 }
315
316
317 #if (NGX_MAIL_SSL)
318
319 ngx_int_t
320 ngx_mail_starttls_only(ngx_mail_session_t *s, ngx_connection_t *c)
321 {
322     ngx_mail_ssl_conf_t *sslcf;
323
324     if (c->ssl) {
325         return 0;
326     }
327
328     sslcf = ngx_mail_get_module_srv_conf(s, ngx_mail_ssl_module);

```

```

329     if (sslcf->starttls == NGX_MAIL_STARTTLS_ONLY) {
330         return 1;
331     }
332 }
333
334     return 0;
335 }
336
337 #endif
338
339
340 ngx_int_t
341 ngx_mail_auth_plain(ngx_mail_session_t *s, ngx_connection_t *c, ngx_uint_t n)
342 {
343     u_char    *p, *last;
344     ngx_str_t *arg, plain;
345
346     arg = s->args.elts;
347
348     #if (NGX_DEBUG_MAIL_PASSWD)
349     ngx_log_debug1(NGX_LOG_DEBUG_MAIL, c->log, 0,
350                 "mail auth plain: \"%V\"", &arg[n]);
351     #endif
352
353     plain.data = ngx_pnalloc(c->pool, ngx_base64_decoded_length(arg[n].len));
354     if (plain.data == NULL) {
355         return NGX_ERROR;
356     }
357
358     if (ngx_decode_base64(&plain, &arg[n]) != NGX_OK) {
359         ngx_log_error(NGX_LOG_INFO, c->log, 0,
360                 "client sent invalid base64 encoding in AUTH PLAIN command");
361         return NGX_MAIL_PARSE_INVALID_COMMAND;
362     }
363
364     p = plain.data;
365     last = p + plain.len;
366
367     while (p < last && *p++) { /* void */ }
368
369     if (p == last) {
370         ngx_log_error(NGX_LOG_INFO, c->log, 0,
371                 "client sent invalid login in AUTH PLAIN command");
372         return NGX_MAIL_PARSE_INVALID_COMMAND;
373     }
374
375     s->login.data = p;
376
377     while (p < last && *p) { p++; }
378
379     if (p == last) {
380         ngx_log_error(NGX_LOG_INFO, c->log, 0,
381                 "client sent invalid password in AUTH PLAIN command");
382         return NGX_MAIL_PARSE_INVALID_COMMAND;
383     }
384
385     s->login.len = p++ - s->login.data;
386
387     s->passwd.len = last - p;
388     s->passwd.data = p;
389
390     #if (NGX_DEBUG_MAIL_PASSWD)
391     ngx_log_debug2(NGX_LOG_DEBUG_MAIL, c->log, 0,
392                 "mail auth plain: \"%V\" \"%V\"", &s->login, &s->passwd);
393     #endif
394
395     return NGX_DONE;
396 }
397
398
399 ngx_int_t
400 ngx_mail_auth_login_username(ngx_mail_session_t *s, ngx_connection_t *c,
401 ngx_uint_t n)
402 {
403     ngx_str_t *arg;
404

```

```

405     arg = s->args.elts;
406
407     ngx_log_debug1(NGX_LOG_DEBUG_MAIL, c->log, 0,
408                 "mail auth login username: \"%V\"", &arg[n]);
409
410     s->login.data = ngx_pnalloc(c->pool, ngx_base64_decoded_length(arg[n].len));
411     if (s->login.data == NULL) {
412         return NGX_ERROR;
413     }
414
415     if (ngx_decode_base64(&s->login, &arg[n]) != NGX_OK) {
416         ngx_log_error(NGX_LOG_INFO, c->log, 0,
417                     "client sent invalid base64 encoding in AUTH LOGIN command");
418         return NGX_MAIL_PARSE_INVALID_COMMAND;
419     }
420
421     ngx_log_debug1(NGX_LOG_DEBUG_MAIL, c->log, 0,
422                 "mail auth login username: \"%V\"", &s->login);
423
424     return NGX_OK;
425 }
426
427
428 ngx_int_t
429 ngx_mail_auth_login_password(ngx_mail_session_t *s, ngx_connection_t *c)
430 {
431     ngx_str_t *arg;
432
433     arg = s->args.elts;
434
435     #if (NGX_DEBUG_MAIL_PASSWD)
436     ngx_log_debug1(NGX_LOG_DEBUG_MAIL, c->log, 0,
437                 "mail auth login password: \"%V\"", &arg[0]);
438     #endif
439
440     s->passwd.data = ngx_pnalloc(c->pool,
441                                ngx_base64_decoded_length(arg[0].len));
442     if (s->passwd.data == NULL) {
443         return NGX_ERROR;
444     }
445
446     if (ngx_decode_base64(&s->passwd, &arg[0]) != NGX_OK) {
447         ngx_log_error(NGX_LOG_INFO, c->log, 0,
448                     "client sent invalid base64 encoding in AUTH LOGIN command");
449         return NGX_MAIL_PARSE_INVALID_COMMAND;
450     }
451
452     #if (NGX_DEBUG_MAIL_PASSWD)
453     ngx_log_debug1(NGX_LOG_DEBUG_MAIL, c->log, 0,
454                 "mail auth login password: \"%V\"", &s->passwd);
455     #endif
456
457     return NGX_DONE;
458 }
459
460
461 ngx_int_t
462 ngx_mail_auth_cram_md5_salt(ngx_mail_session_t *s, ngx_connection_t *c,
463                             char *prefix, size_t len)
464 {
465     u_char *p;
466     ngx_str_t salt;
467     ngx_uint_t n;
468
469     p = ngx_pnalloc(c->pool, len + ngx_base64_encoded_length(s->salt.len) + 2);
470     if (p == NULL) {
471         return NGX_ERROR;
472     }
473
474     salt.data = ngx_cpymem(p, prefix, len);
475     s->salt.len -= 2;
476
477     ngx_encode_base64(&salt, &s->salt);
478
479     s->salt.len += 2;
480     n = len + salt.len;

```



```

481     p[n++] = CR; p[n++] = LF;
482
483     s->out.len = n;
484     s->out.data = p;
485
486     return NGX_OK;
487 }
488
489
490 ngx_int_t
491 ngx_mail_auth_cram_md5(ngx_mail_session_t *s, ngx_connection_t *c)
492 {
493     u_char      *p, *last;
494     ngx_str_t   *arg;
495
496     arg = s->args.elts;
497
498     ngx_log_debug1(NGX_LOG_DEBUG_MAIL, c->log, 0,
499                  "mail auth cram-md5: \"%V\"", &arg[0]);
500
501     s->login.data = ngx_pnalloc(c->pool, ngx_base64_decoded_length(arg[0].len));
502     if (s->login.data == NULL) {
503         return NGX_ERROR;
504     }
505
506     if (ngx_decode_base64(&s->login, &arg[0]) != NGX_OK) {
507         ngx_log_error(NGX_LOG_INFO, c->log, 0,
508                      "client sent invalid base64 encoding in AUTH CRAM-MD5 command");
509         return NGX_MAIL_PARSE_INVALID_COMMAND;
510     }
511
512     p = s->login.data;
513     last = p + s->login.len;
514
515     while (p < last) {
516         if (*p++ == ' ') {
517             s->login.len = p - s->login.data - 1;
518             s->passwd.len = last - p;
519             s->passwd.data = p;
520             break;
521         }
522     }
523
524     if (s->passwd.len != 32) {
525         ngx_log_error(NGX_LOG_INFO, c->log, 0,
526                      "client sent invalid CRAM-MD5 hash in AUTH CRAM-MD5 command");
527         return NGX_MAIL_PARSE_INVALID_COMMAND;
528     }
529
530     ngx_log_debug2(NGX_LOG_DEBUG_MAIL, c->log, 0,
531                  "mail auth cram-md5: \"%V\" \"%V\"", &s->login, &s->passwd);
532
533     s->auth_method = NGX_MAIL_AUTH_CRAM_MD5;
534
535     return NGX_DONE;
536 }
537
538
539 void
540 ngx_mail_send(ngx_event_t *wev)
541 {
542     ngx_int_t      n;
543     ngx_connection_t *c;
544     ngx_mail_session_t *s;
545     ngx_mail_core_srv_conf_t *cscf;
546
547     c = wev->data;
548     s = c->data;
549
550     if (wev->timedout) {
551         ngx_log_error(NGX_LOG_INFO, c->log, NGX_ETIMEDOUT, "client timed out");
552         c->timedout = 1;
553         ngx_mail_close_connection(c);
554         return;
555     }
556

```

```

557     if (s->out.len == 0) {
558         if (ngx\_handle\_write\_event(c->write, 0) != NGX\_OK) {
559             ngx\_mail\_close\_connection(c);
560         }
561     }
562     return;
563 }
564
565 n = c->send(c, s->out.data, s->out.len);
566
567 if (n > 0) {
568     s->out.data += n;
569     s->out.len -= n;
570
571     if (s->out.len != 0) {
572         goto again;
573     }
574
575     if (wev->timer_set) {
576         ngx\_del\_timer(wev);
577     }
578
579     if (s->quit) {
580         ngx\_mail\_close\_connection(c);
581         return;
582     }
583
584     if (s->blocked) {
585         c->read->handler(c->read);
586     }
587
588     return;
589 }
590
591 if (n == NGX\_ERROR) {
592     ngx\_mail\_close\_connection(c);
593     return;
594 }
595
596 /* n == NGX\_AGAIN */
597
598 again:
599
600 cscf = ngx\_mail\_get\_module\_srv\_conf(s, ngx\_mail\_core\_module);
601
602 ngx\_add\_timer(c->write, cscf->timeout);
603
604 if (ngx\_handle\_write\_event(c->write, 0) != NGX\_OK) {
605     ngx\_mail\_close\_connection(c);
606     return;
607 }
608 }
609
610
611 ngx\_int\_t
612 ngx\_mail\_read\_command(ngx\_mail\_session\_t *s, ngx\_connection\_t *c)
613 {
614     ssize\_t                n;
615     ngx\_int\_t             rc;
616     ngx\_str\_t             l;
617     ngx\_mail\_core\_srv\_conf\_t *cscf;
618
619     n = c->recv(c, s->buffer->last, s->buffer->end - s->buffer->last);
620
621     if (n == NGX\_ERROR || n == 0) {
622         ngx\_mail\_close\_connection(c);
623         return NGX\_ERROR;
624     }
625
626     if (n > 0) {
627         s->buffer->last += n;
628     }
629
630     if (n == NGX\_AGAIN) {
631         if (ngx\_handle\_read\_event(c->read, 0) != NGX\_OK) {
632             ngx\_mail\_session\_internal\_server\_error(s);

```

```

633         return NGX\_ERROR;
634     }
635
636     if (s->buffer->pos == s->buffer->last) {
637         return NGX\_AGAIN;
638     }
639 }
640
641 cscf = ngx\_mail\_get\_module\_srv\_conf(s, ngx\_mail\_core\_module);
642
643 rc = cscf->protocol->parse_command(s);
644
645 if (rc == NGX\_AGAIN) {
646
647     if (s->buffer->last < s->buffer->end) {
648         return rc;
649     }
650
651     l.len = s->buffer->last - s->buffer->start;
652     l.data = s->buffer->start;
653
654     ngx\_log\_error(NGX\_LOG\_INFO, c->log, 0,
655                 "client sent too long command \"%V\"", &l);
656
657     s->quit = 1;
658
659     return NGX\_MAIL\_PARSE\_INVALID\_COMMAND;
660 }
661
662 if (rc == NGX\_IMAP\_NEXT || rc == NGX\_MAIL\_PARSE\_INVALID\_COMMAND) {
663     return rc;
664 }
665
666 if (rc == NGX\_ERROR) {
667     ngx\_mail\_close\_connection(c);
668     return NGX\_ERROR;
669 }
670
671 return NGX\_OK;
672 }
673
674
675 void
676 ngx\_mail\_auth(ngx\_mail\_session\_t *s, ngx\_connection\_t *c)
677 {
678     s->args.nelts = 0;
679
680     if (s->buffer->pos == s->buffer->last) {
681         s->buffer->pos = s->buffer->start;
682         s->buffer->last = s->buffer->start;
683     }
684
685     s->state = 0;
686
687     if (c->read->timer_set) {
688         ngx\_del\_timer(c->read);
689     }
690
691     s->login_attempt++;
692
693     ngx\_mail\_auth\_http\_init(s);
694 }
695
696
697 void
698 ngx\_mail\_session\_internal\_server\_error(ngx\_mail\_session\_t *s)
699 {
700     ngx\_mail\_core\_srv\_conf\_t *cscf;
701
702     cscf = ngx\_mail\_get\_module\_srv\_conf(s, ngx\_mail\_core\_module);
703
704     s->out = cscf->protocol->internal_server_error;
705     s->quit = 1;
706
707     ngx\_mail\_send(s->connection->write);
708 }

```

```

709
710
711 void
712 ngx_mail_close_connection(ngx_connection_t *c)
713 {
714     ngx_pool_t *pool;
715
716     ngx_log_debug1(NGX_LOG_DEBUG_MAIL, c->log, 0,
717                 "close mail connection: %d", c->fd);
718
719     #if (NGX_MAIL_SSL)
720
721     if (c->ssl) {
722         if (ngx_ssl_shutdown(c) == NGX_AGAIN) {
723             c->ssl->handler = ngx_mail_close_connection;
724             return;
725         }
726     }
727
728     #endif
729
730     #if (NGX_STAT_STUB)
731     (void) ngx_atomic_fetch_add(ngx_stat_active, -1);
732     #endif
733
734     c->destroyed = 1;
735
736     pool = c->pool;
737
738     ngx_close_connection(c);
739
740     ngx_destroy_pool(pool);
741 }
742
743
744 u_char *
745 ngx_mail_log_error(ngx_log_t *log, u_char *buf, size_t len)
746 {
747     u_char          *p;
748     ngx_mail_session_t *s;
749     ngx_mail_log_ctx_t *ctx;
750
751     if (log->action) {
752         p = ngx_snprintf(buf, len, " while %s", log->action);
753         len -= p - buf;
754         buf = p;
755     }
756
757     ctx = log->data;
758
759     p = ngx_snprintf(buf, len, ", client: %V", ctx->client);
760     len -= p - buf;
761     buf = p;
762
763     s = ctx->session;
764
765     if (s == NULL) {
766         return p;
767     }
768
769     p = ngx_snprintf(buf, len, "%s, server: %V",
770                     s->starttls ? " using starttls" : "",
771                     s->addr_text);
772     len -= p - buf;
773     buf = p;
774
775     if (s->login.len == 0) {
776         return p;
777     }
778
779     p = ngx_snprintf(buf, len, ", login: \"%V\"", &s->login);
780     len -= p - buf;
781     buf = p;
782
783     if (s->proxy == NULL) {
784         return p;

```

```
785     }  
786  
787     p = ngx_snprintf(buf, len, ", upstream: %V", s->proxy->upstream.name);  
788  
789     return p;  
790 }
```

[One Level Up](#)

[Top Level](#)

# src/mail/nginx\_mail\_ssl\_module.h - nginx-1.7.10

## Data types defined

- [ngx\\_mail\\_ssl\\_conf\\_t](#)

## Macros defined

- [NGX\\_MAIL\\_STARTTLS\\_OFF](#)
- [NGX\\_MAIL\\_STARTTLS\\_ON](#)
- [NGX\\_MAIL\\_STARTTLS\\_ONLY](#)
- [\\_NGX\\_MAIL\\_SSL\\_H\\_INCLUDED](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_MAIL\_SSL\_H\_INCLUDED
9 #define \_NGX\_MAIL\_SSL\_H\_INCLUDED
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14 #include <ngx_mail.h>
15
16
17 #define NGX\_MAIL\_STARTTLS\_OFF    0
18 #define NGX\_MAIL\_STARTTLS\_ON     1
19 #define NGX\_MAIL\_STARTTLS\_ONLY    2
20
21
22 typedef struct {
23     ngx\_flag\_t        enable;
24     ngx\_flag\_t        prefer_server_ciphers;
25
26     ngx\_ssl\_t         ssl;
27
28     ngx\_uint\_t        starttls;
29     ngx\_uint\_t        protocols;
30
31     ssize_t          builtin_session_cache;
32
33     time_t           session_timeout;
34
35     ngx\_str\_t         certificate;
36     ngx\_str\_t         certificate_key;
37     ngx\_str\_t         dhparam;
38     ngx\_str\_t         ecdh_curve;
39
40     ngx\_str\_t         ciphers;
41
42     ngx\_array\_t      *passwords;
43
44     ngx\_shm\_zone\_t  *shm_zone;
45
46     ngx\_flag\_t        session_tickets;
47     ngx\_array\_t      *session_ticket_keys;
48
49     u_char           *file;
50     ngx\_uint\_t        line;
```

```
51 } ngx_mail_ssl_conf_t;  
52  
53  
54 extern ngx_module_t ngx_mail_ssl_module;  
55  
56  
57 #endif /* NGX_MAIL_SSL_H_INCLUDED */
```

[One Level Up](#)

[Top Level](#)

## src/mail/nginx\_mail\_core\_module.c - nginx-1.7.10

### Global variables defined

- [ngx\\_conf\\_deprecated\\_so\\_keepalive](#)
- [ngx\\_mail\\_core\\_commands](#)
- [ngx\\_mail\\_core\\_module](#)
- [ngx\\_mail\\_core\\_module\\_ctx](#)

### Functions defined

- [ngx\\_mail\\_capabilities](#)
- [ngx\\_mail\\_core\\_create\\_main\\_conf](#)
- [ngx\\_mail\\_core\\_create\\_srv\\_conf](#)
- [ngx\\_mail\\_core\\_listen](#)
- [ngx\\_mail\\_core\\_merge\\_srv\\_conf](#)
- [ngx\\_mail\\_core\\_protocol](#)
- [ngx\\_mail\\_core\\_resolver](#)
- [ngx\\_mail\\_core\\_server](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11 #include <ngx_mail.h>
12
13
14 static void *ngx_mail_core_create_main_conf(ngx_conf_t *cf);
15 static void *ngx_mail_core_create_srv_conf(ngx_conf_t *cf);
16 static char *ngx_mail_core_merge_srv_conf(ngx_conf_t *cf, void *parent,
17     void *child);
18 static char *ngx_mail_core_server(ngx_conf_t *cf, ngx_command_t *cmd,
19     void *conf);
20 static char *ngx_mail_core_listen(ngx_conf_t *cf, ngx_command_t *cmd,
21     void *conf);
22 static char *ngx_mail_core_protocol(ngx_conf_t *cf, ngx_command_t *cmd,
23     void *conf);
24 static char *ngx_mail_core_resolver(ngx_conf_t *cf, ngx_command_t *cmd,
25     void *conf);
26
27
28 static ngx_conf_deprecated_t ngx_conf_deprecated_so_keepalive = {
29     ngx_conf_deprecated, "so_keepalive",
30     "so_keepalive\" parameter of the \"listen"
31 };
32
33
34 static ngx_command_t ngx_mail_core_commands[] = {
```



```

35
36 { ngx_string("server"),
37     NGX_MAIL_MAIN_CONF|NGX_CONF_BLOCK|NGX_CONF_NOARGS,
38     ngx_mail_core_server,
39     0,
40     0,
41     NULL },
42
43 { ngx_string("listen"),
44     NGX_MAIL_SRV_CONF|NGX_CONF_1MORE,
45     ngx_mail_core_listen,
46     NGX_MAIL_SRV_CONF_OFFSET,
47     0,
48     NULL },
49
50 { ngx_string("protocol"),
51     NGX_MAIL_SRV_CONF|NGX_CONF_TAKE1,
52     ngx_mail_core_protocol,
53     NGX_MAIL_SRV_CONF_OFFSET,
54     0,
55     NULL },
56
57 { ngx_string("so_keepalive"),
58     NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_FLAG,
59     ngx_conf_set_flag_slot,
60     NGX_MAIL_SRV_CONF_OFFSET,
61     offsetof(ngx_mail_core_srv_conf_t, so_keepalive),
62     &ngx_conf_deprecated_so_keepalive },
63
64 { ngx_string("timeout"),
65     NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_TAKE1,
66     ngx_conf_set_msec_slot,
67     NGX_MAIL_SRV_CONF_OFFSET,
68     offsetof(ngx_mail_core_srv_conf_t, timeout),
69     NULL },
70
71 { ngx_string("server_name"),
72     NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_TAKE1,
73     ngx_conf_set_str_slot,
74     NGX_MAIL_SRV_CONF_OFFSET,
75     offsetof(ngx_mail_core_srv_conf_t, server_name),
76     NULL },
77
78 { ngx_string("resolver"),
79     NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_1MORE,
80     ngx_mail_core_resolver,
81     NGX_MAIL_SRV_CONF_OFFSET,
82     0,
83     NULL },
84
85 { ngx_string("resolver_timeout"),
86     NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_TAKE1,
87     ngx_conf_set_msec_slot,
88     NGX_MAIL_SRV_CONF_OFFSET,
89     offsetof(ngx_mail_core_srv_conf_t, resolver_timeout),
90     NULL },
91
92     ngx_null_command
93 };
94
95
96 static ngx_mail_module_t  ngx_mail_core_module_ctx = {
97     NULL,                                /* protocol */
98
99     ngx_mail_core_create_main_conf,      /* create main configuration */
100    NULL,                                  /* init main configuration */
101
102     ngx_mail_core_create_srv_conf,       /* create server configuration */
103     ngx_mail_core_merge_srv_conf        /* merge server configuration */
104 };
105
106
107 ngx_module_t  ngx_mail_core_module = {
108     NGX_MODULE_V1,
109     &ngx_mail_core_module_ctx,          /* module context */
110     ngx_mail_core_commands,            /* module directives */

```

```

111     NGX_MAIL_MODULE,                               /* module type */
112     NULL,                                           /* init master */
113     NULL,                                           /* init module */
114     NULL,                                           /* init process */
115     NULL,                                           /* init thread */
116     NULL,                                           /* exit thread */
117     NULL,                                           /* exit process */
118     NULL,                                           /* exit master */
119     NGX_MODULE_V1_PADDING
120 };
121
122
123 static void *
124 ngx_mail_core_create_main_conf(ngx_conf_t *cf)
125 {
126     ngx_mail_core_main_conf_t *cmcf;
127
128     cmcf = ngx_palloc(cf->pool, sizeof(ngx_mail_core_main_conf_t));
129     if (cmcf == NULL) {
130         return NULL;
131     }
132
133     if (ngx_array_init(&cmcf->servers, cf->pool, 4,
134         sizeof(ngx_mail_core_srv_conf_t *))
135         != NGX_OK)
136     {
137         return NULL;
138     }
139
140     if (ngx_array_init(&cmcf->listen, cf->pool, 4, sizeof(ngx_mail_listen_t))
141         != NGX_OK)
142     {
143         return NULL;
144     }
145
146     return cmcf;
147 }
148
149
150 static void *
151 ngx_mail_core_create_srv_conf(ngx_conf_t *cf)
152 {
153     ngx_mail_core_srv_conf_t *cscf;
154
155     cscf = ngx_palloc(cf->pool, sizeof(ngx_mail_core_srv_conf_t));
156     if (cscf == NULL) {
157         return NULL;
158     }
159
160     /*
161      * set by ngx_palloc():
162      *
163      *     cscf->protocol = NULL;
164      */
165
166     cscf->timeout = NGX_CONF_UNSET_MSEC;
167     cscf->resolver_timeout = NGX_CONF_UNSET_MSEC;
168     cscf->so_keepalive = NGX_CONF_UNSET;
169
170     cscf->resolver = NGX_CONF_UNSET_PTR;
171
172     cscf->file_name = cf->conf_file->file.name.data;
173     cscf->line = cf->conf_file->line;
174
175     return cscf;
176 }
177
178
179 static char *
180 ngx_mail_core_merge_srv_conf(ngx_conf_t *cf, void *parent, void *child)
181 {
182     ngx_mail_core_srv_conf_t *prev = parent;
183     ngx_mail_core_srv_conf_t *conf = child;
184
185     ngx_conf_merge_msec_value(conf->timeout, prev->timeout, 60000);
186     ngx_conf_merge_msec_value(conf->resolver_timeout, prev->resolver_timeout,

```

```

187         30000);
188
189     ngx_conf_merge_value(conf->so_keepalive, prev->so_keepalive, 0);
190
191
192     ngx_conf_merge_str_value(conf->server_name, prev->server_name, "");
193
194     if (conf->server_name.len == 0) {
195         conf->server_name = cf->cycle->hostname;
196     }
197
198     if (conf->protocol == NULL) {
199         ngx_log_error(NGX_LOG_EMERG, cf->log, 0,
200             "unknown mail protocol for server in %s:%ui",
201             conf->file_name, conf->line);
202         return NGX_CONF_ERROR;
203     }
204
205     ngx_conf_merge_ptr_value(conf->resolver, prev->resolver, NULL);
206
207     return NGX_CONF_OK;
208 }
209
210
211 static char *
212 ngx_mail_core_server(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
213 {
214     char *rv;
215     void *mconf;
216     ngx_uint_t m;
217     ngx_conf_t pcf;
218     ngx_mail_module_t *module;
219     ngx_mail_conf_ctx_t *ctx, *mail_ctx;
220     ngx_mail_core_srv_conf_t *cscf, **cscfp;
221     ngx_mail_core_main_conf_t *cmcf;
222
223     ctx = ngx_palloc(cf->pool, sizeof(ngx_mail_conf_ctx_t));
224     if (ctx == NULL) {
225         return NGX_CONF_ERROR;
226     }
227
228     mail_ctx = cf->ctx;
229     ctx->main_conf = mail_ctx->main_conf;
230
231     /* the server{}'s srv_conf */
232
233     ctx->srv_conf = ngx_palloc(cf->pool, sizeof(void *) * ngx_mail_max_module);
234     if (ctx->srv_conf == NULL) {
235         return NGX_CONF_ERROR;
236     }
237
238     for (m = 0; ngx_modules[m]; m++) {
239         if (ngx_modules[m]->type != NGX_MAIL_MODULE) {
240             continue;
241         }
242
243         module = ngx_modules[m]->ctx;
244
245         if (module->create_srv_conf) {
246             mconf = module->create_srv_conf(cf);
247             if (mconf == NULL) {
248                 return NGX_CONF_ERROR;
249             }
250
251             ctx->srv_conf[ngx_modules[m]->ctx_index] = mconf;
252         }
253     }
254
255     /* the server configuration context */
256
257     cscf = ctx->srv_conf[ngx_mail_core_module.ctx_index];
258     cscf->ctx = ctx;
259
260     cmcf = ctx->main_conf[ngx_mail_core_module.ctx_index];
261
262     cscfp = ngx_array_push(&cmcf->servers);

```

```

263     if (cscfp == NULL) {
264         return NGX\_CONF\_ERROR;
265     }
266
267     *cscfp = cscf;
268
269
270     /* parse inside server{} */
271
272     pcf = *cf;
273     cf->ctx = ctx;
274     cf->cmd_type = NGX\_MAIL\_SRV\_CONF;
275
276     rv = ngx\_conf\_parse(cf, NULL);
277
278     *cf = pcf;
279
280     return rv;
281 }
282
283
284 static char *
285 ngx\_mail\_core\_listen(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
286 {
287     ngx\_mail\_core\_srv\_conf\_t *cscf = conf;
288
289     size_t          len, off;
290     in_port_t       port;
291     ngx\_str\_t       *value;
292     ngx\_url\_t       u;
293     ngx\_uint\_t      i, m;
294     struct sockaddr *sa;
295     ngx\_mail\_listen\_t *ls;
296     ngx\_mail\_module\_t *module;
297     struct sockaddr_in *sin;
298     ngx\_mail\_core\_main\_conf\_t *cmcf;
299     #if (NGX_HAVE_INET6)
300     struct sockaddr_in6 *sin6;
301     #endif
302
303     value = cf->args->elts;
304
305     ngx\_memzero(&u, sizeof(ngx\_url\_t));
306
307     u.url = value[1];
308     u.listen = 1;
309
310     if (ngx\_parse\_url(cf->pool, &u) != NGX\_OK) {
311         if (u.err) {
312             ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
313                 "%s in \"%V\" of the \"listen\" directive",
314                 u.err, &u.url);
315         }
316
317         return NGX\_CONF\_ERROR;
318     }
319
320     cmcf = ngx\_mail\_conf\_get\_module\_main\_conf(cf, ngx\_mail\_core\_module);
321
322     ls = cmcf->listen.elts;
323
324     for (i = 0; i < cmcf->listen.nelts; i++) {
325
326         sa = (struct sockaddr *) ls[i].sockaddr;
327
328         if (sa->sa_family != u.family) {
329             continue;
330         }
331
332         switch (sa->sa_family) {
333
334         #if (NGX_HAVE_INET6)
335             case AF_INET6:
336                 off = offsetof(struct sockaddr_in6, sin6_addr);
337                 len = 16;
338                 sin6 = (struct sockaddr_in6 *) sa;

```

```

339         port = sin6->sin6_port;
340         break;
341     #endif
342
343     #if (NGX_HAVE_UNIX_DOMAIN)
344         case AF_UNIX:
345             off = offsetof(struct sockaddr_un, sun_path);
346             len = sizeof(((struct sockaddr_un *) sa)->sun_path);
347             port = 0;
348             break;
349     #endif
350
351     default: /* AF_INET */
352         off = offsetof(struct sockaddr_in, sin_addr);
353         len = 4;
354         sin = (struct sockaddr_in *) sa;
355         port = sin->sin_port;
356         break;
357     }
358
359     if (ngx_memcmp(ls[i].sockaddr + off, u.sockaddr + off, len) != 0) {
360         continue;
361     }
362
363     if (port != u.port) {
364         continue;
365     }
366
367     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
368         "duplicate \"%V\" address and port pair", &u.url);
369     return NGX_CONF_ERROR;
370 }
371
372 ls = ngx_array_push(&cmcf->listen);
373 if (ls == NULL) {
374     return NGX_CONF_ERROR;
375 }
376
377 ngx_memzero(ls, sizeof(ngx_mail_listen_t));
378
379 ngx_memcpy(ls->sockaddr, u.sockaddr, u.socklen);
380
381 ls->socklen = u.socklen;
382 ls->wildcard = u.wildcard;
383 ls->ctx = cf->ctx;
384
385 #if (NGX_HAVE_INET6 && defined IPV6_V6ONLY)
386     ls->ipv6only = 1;
387 #endif
388
389 if (cscf->protocol == NULL) {
390     for (m = 0; ngx_modules[m]; m++) {
391         if (ngx_modules[m]->type != NGX_MAIL_MODULE) {
392             continue;
393         }
394
395         module = ngx_modules[m]->ctx;
396
397         if (module->protocol == NULL) {
398             continue;
399         }
400
401         for (i = 0; module->protocol->port[i]; i++) {
402             if (module->protocol->port[i] == u.port) {
403                 cscf->protocol = module->protocol;
404                 break;
405             }
406         }
407     }
408 }
409
410 for (i = 2; i < cf->args->nelts; i++) {
411
412     if (ngx_strcmp(value[i].data, "bind") == 0) {
413         ls->bind = 1;
414         continue;

```

```

415     }
416
417     if (ngx_strncmp(value[i].data, "ipv6only=0", 10) == 0) {
418 #if (NGX_HAVE_INET6 && defined IPV6_V6ONLY)
419         struct sockaddr *sa;
420         u_char          buf[NGX_SOCKADDR_STRLEN];
421
422         sa = (struct sockaddr *) ls->sockaddr;
423
424         if (sa->sa_family == AF_INET6) {
425
426             if (ngx_strcmp(&value[i].data[10], "n") == 0) {
427                 ls->ipv6only = 1;
428
429             } else if (ngx_strcmp(&value[i].data[10], "ff") == 0) {
430                 ls->ipv6only = 0;
431
432             } else {
433                 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
434                     "invalid ipv6only flags \"%s\"",
435                     &value[i].data[9]);
436                 return NGX_CONF_ERROR;
437             }
438
439             ls->bind = 1;
440
441         } else {
442             len = ngx_sock_ntop(sa, ls->socklen, buf,
443                 NGX_SOCKADDR_STRLEN, 1);
444
445             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
446                 "ipv6only is not supported "
447                 "on addr \"%s\"", ignored, len, buf);
448         }
449
450         continue;
451 #else
452         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
453             "bind ipv6only is not supported "
454             "on this platform");
455         return NGX_CONF_ERROR;
456 #endif
457     }
458
459     if (ngx_strcmp(value[i].data, "ssl") == 0) {
460 #if (NGX_MAIL_SSL)
461         ls->ssl = 1;
462         continue;
463 #else
464         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
465             "the \"ssl\" parameter requires "
466             "\"ngx_mail_ssl_module\"");
467         return NGX_CONF_ERROR;
468 #endif
469     }
470
471     if (ngx_strncmp(value[i].data, "so_keepalive=", 13) == 0) {
472
473         if (ngx_strcmp(&value[i].data[13], "on") == 0) {
474             ls->so_keepalive = 1;
475
476         } else if (ngx_strcmp(&value[i].data[13], "off") == 0) {
477             ls->so_keepalive = 2;
478
479         } else {
480
481 #if (NGX_HAVE_KEEPALIVE_TUNABLE)
482             u_char *p, *end;
483             ngx_str_t s;
484
485             end = value[i].data + value[i].len;
486             s.data = value[i].data + 13;
487
488             p = ngx_strlchr(s.data, end, ':');
489             if (p == NULL) {
490                 p = end;

```

```

491     }
492
493     if (p > s.data) {
494         s.len = p - s.data;
495
496         ls->tcp_keepidle = ngx_parse_time(&s, 1);
497         if (ls->tcp_keepidle == (time_t) NGX_ERROR) {
498             goto invalid_so_keepalive;
499         }
500     }
501
502     s.data = (p < end) ? (p + 1) : end;
503
504     p = ngx_strlchr(s.data, end, ':');
505     if (p == NULL) {
506         p = end;
507     }
508
509     if (p > s.data) {
510         s.len = p - s.data;
511
512         ls->tcp_keepintvl = ngx_parse_time(&s, 1);
513         if (ls->tcp_keepintvl == (time_t) NGX_ERROR) {
514             goto invalid_so_keepalive;
515         }
516     }
517
518     s.data = (p < end) ? (p + 1) : end;
519
520     if (s.data < end) {
521         s.len = end - s.data;
522
523         ls->tcp_keepcnt = ngx_atoi(s.data, s.len);
524         if (ls->tcp_keepcnt == NGX_ERROR) {
525             goto invalid_so_keepalive;
526         }
527     }
528
529     if (ls->tcp_keepidle == 0 && ls->tcp_keepintvl == 0
530         && ls->tcp_keepcnt == 0)
531     {
532         goto invalid_so_keepalive;
533     }
534
535     ls->so_keepalive = 1;
536
537 #else
538
539     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
540         "the \"so_keepalive\" parameter accepts \"
541         \"only \"on\" or \"off\" on this platform");
542     return NGX_CONF_ERROR;
543
544 #endif
545 }
546
547 ls->bind = 1;
548
549 continue;
550
551 #if (NGX_HAVE_KEEPALIVE_TUNABLE)
552     invalid_so_keepalive:
553
554     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
555         "invalid so_keepalive value: \"%s\"",
556         &value[i].data[13]);
557     return NGX_CONF_ERROR;
558 #endif
559 }
560
561 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
562     "the invalid \"%V\" parameter", &value[i]);
563 return NGX_CONF_ERROR;
564 }
565
566 return NGX_CONF_OK;

```

```

567 }
568
569
570 static char *
571 ngx_mail_core_protocol(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
572 {
573     ngx_mail_core_srv_conf_t *cscf = conf;
574
575     ngx_str_t      *value;
576     ngx_uint_t      m;
577     ngx_mail_module_t *module;
578
579     value = cf->args->elts;
580
581     for (m = 0; ngx_modules[m]; m++) {
582         if (ngx_modules[m]->type != NGX_MAIL_MODULE) {
583             continue;
584         }
585
586         module = ngx_modules[m]->ctx;
587
588         if (module->protocol
589             && ngx_strcmp(module->protocol->name.data, value[1].data) == 0)
590         {
591             cscf->protocol = module->protocol;
592
593             return NGX_CONF_OK;
594         }
595     }
596
597     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
598         "unknown protocol \"%V\"", &value[1]);
599     return NGX_CONF_ERROR;
600 }
601
602
603 static char *
604 ngx_mail_core_resolver(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
605 {
606     ngx_mail_core_srv_conf_t *cscf = conf;
607
608     ngx_str_t *value;
609
610     value = cf->args->elts;
611
612     if (cscf->resolver != NGX_CONF_UNSET_PTR) {
613         return "is duplicate";
614     }
615
616     if (ngx_strcmp(value[1].data, "off") == 0) {
617         cscf->resolver = NULL;
618         return NGX_CONF_OK;
619     }
620
621     cscf->resolver = ngx_resolver_create(cf, &value[1], cf->args->nelts - 1);
622     if (cscf->resolver == NULL) {
623         return NGX_CONF_ERROR;
624     }
625
626     return NGX_CONF_OK;
627 }
628
629
630 char *
631 ngx_mail_capabilities(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
632 {
633     char *p = conf;
634
635     ngx_str_t *c, *value;
636     ngx_uint_t i;
637     ngx_array_t *a;
638
639     a = (ngx_array_t *) (p + cmd->offset);
640
641     value = cf->args->elts;
642

```



```
643     for (i = 1; i < cf->args->nelts; i++) {
644         c = ngx_array_push(a);
645         if (c == NULL) {
646             return NGX_CONF_ERROR;
647         }
648
649         *c = value[i];
650     }
651
652     return NGX_CONF_OK;
653 }
```

[One Level Up](#)

[Top Level](#)

## src/mail/nginx\_mail\_auth\_http\_module.c - nginx-1.7.10

### Global variables defined

- [ngx\\_mail\\_auth\\_http\\_commands](#)
- [ngx\\_mail\\_auth\\_http\\_method](#)
- [ngx\\_mail\\_auth\\_http\\_module](#)
- [ngx\\_mail\\_auth\\_http\\_module\\_ctx](#)
- [ngx\\_mail\\_smtp\\_errcode](#)

### Data types defined

- [ngx\\_mail\\_auth\\_http\\_conf\\_t](#)
- [ngx\\_mail\\_auth\\_http\\_ctx\\_s](#)
- [ngx\\_mail\\_auth\\_http\\_ctx\\_t](#)
- [ngx\\_mail\\_auth\\_http\\_handler\\_pt](#)

### Functions defined

- [ngx\\_mail\\_auth\\_http](#)
- [ngx\\_mail\\_auth\\_http\\_block\\_read](#)
- [ngx\\_mail\\_auth\\_http\\_create\\_conf](#)
- [ngx\\_mail\\_auth\\_http\\_create\\_request](#)
- [ngx\\_mail\\_auth\\_http\\_dummy\\_handler](#)
- [ngx\\_mail\\_auth\\_http\\_escape](#)
- [ngx\\_mail\\_auth\\_http\\_header](#)
- [ngx\\_mail\\_auth\\_http\\_ignore\\_status\\_line](#)
- [ngx\\_mail\\_auth\\_http\\_init](#)
- [ngx\\_mail\\_auth\\_http\\_merge\\_conf](#)
- [ngx\\_mail\\_auth\\_http\\_parse\\_header\\_line](#)
- [ngx\\_mail\\_auth\\_http\\_process\\_headers](#)
- [ngx\\_mail\\_auth\\_http\\_read\\_handler](#)
- [ngx\\_mail\\_auth\\_http\\_write\\_handler](#)
- [ngx\\_mail\\_auth\\_sleep\\_handler](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
```

```

4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_event.h>
11 #include <ngx_event_connect.h>
12 #include <ngx_mail.h>
13
14
15 typedef struct {
16     ngx_addr_t             *peer;
17
18     ngx_msec_t             timeout;
19
20     ngx_str_t              host_header;
21     ngx_str_t              uri;
22     ngx_str_t              header;
23
24     ngx_array_t            *headers;
25
26     u_char                 *file;
27     ngx_uint_t             line;
28 } ngx_mail_auth_http_conf_t;
29
30
31 typedef struct ngx_mail_auth_http_ctx_s ngx_mail_auth_http_ctx_t;
32
33 typedef void (*ngx_mail_auth_http_handler_pt)(ngx_mail_session_t *s,
34     ngx_mail_auth_http_ctx_t *ctx);
35
36 struct ngx_mail_auth_http_ctx_s {
37     ngx_buf_t              *request;
38     ngx_buf_t              *response;
39     ngx_peer_connection_t  peer;
40
41     ngx_mail_auth_http_handler_pt handler;
42
43     ngx_uint_t             state;
44
45     u_char                 *header_name_start;
46     u_char                 *header_name_end;
47     u_char                 *header_start;
48     u_char                 *header_end;
49
50     ngx_str_t              addr;
51     ngx_str_t              port;
52     ngx_str_t              err;
53     ngx_str_t              errmsg;
54     ngx_str_t              errcode;
55
56     time_t                 sleep;
57
58     ngx_pool_t             *pool;
59 };
60
61
62 static void ngx_mail_auth_http_write_handler(ngx_event_t *wev);
63 static void ngx_mail_auth_http_read_handler(ngx_event_t *rev);
64 static void ngx_mail_auth_http_ignore_status_line(ngx_mail_session_t *s,
65     ngx_mail_auth_http_ctx_t *ctx);
66 static void ngx_mail_auth_http_process_headers(ngx_mail_session_t *s,
67     ngx_mail_auth_http_ctx_t *ctx);
68 static void ngx_mail_auth_sleep_handler(ngx_event_t *rev);
69 static ngx_int_t ngx_mail_auth_http_parse_header_line(ngx_mail_session_t *s,
70     ngx_mail_auth_http_ctx_t *ctx);
71 static void ngx_mail_auth_http_block_read(ngx_event_t *rev);
72 static void ngx_mail_auth_http_dummy_handler(ngx_event_t *ev);
73 static ngx_buf_t *ngx_mail_auth_http_create_request(ngx_mail_session_t *s,
74     ngx_pool_t *pool, ngx_mail_auth_http_conf_t *ahcf);
75 static ngx_int_t ngx_mail_auth_http_escape(ngx_pool_t *pool, ngx_str_t *text,
76     ngx_str_t *escaped);
77
78 static void *ngx_mail_auth_http_create_conf(ngx_conf_t *cf);
79 static char *ngx_mail_auth_http_merge_conf(ngx_conf_t *cf, void *parent,

```

```

80     void *child);
81 static char *ngx_mail_auth_http(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
82 static char *ngx_mail_auth_http_header(ngx_conf_t *cf, ngx_command_t *cmd,
83     void *conf);
84
85
86 static ngx_command_t  ngx_mail_auth_http_commands[] = {
87
88     { ngx_string("auth_http"),
89       NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_TAKE1,
90       ngx_mail_auth_http,
91       NGX_MAIL_SRV_CONF_OFFSET,
92       0,
93       NULL },
94
95     { ngx_string("auth_http_timeout"),
96       NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_TAKE1,
97       ngx_conf_set_msec_slot,
98       NGX_MAIL_SRV_CONF_OFFSET,
99       offsetof(ngx_mail_auth_http_conf_t, timeout),
100      NULL },
101
102     { ngx_string("auth_http_header"),
103       NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_TAKE2,
104       ngx_mail_auth_http_header,
105       NGX_MAIL_SRV_CONF_OFFSET,
106       0,
107       NULL },
108
109     ngx_null_command
110 };
111
112
113 static ngx_mail_module_t  ngx_mail_auth_http_module_ctx = {
114     NULL,                                     /* protocol */
115
116     NULL,                                     /* create main configuration */
117     NULL,                                     /* init main configuration */
118
119     ngx_mail_auth_http_create_conf,          /* create server configuration */
120     ngx_mail_auth_http_merge_conf           /* merge server configuration */
121 };
122
123
124 ngx_module_t  ngx_mail_auth_http_module = {
125     NGX_MODULE_V1,
126     &ngx_mail_auth_http_module_ctx,         /* module context */
127     ngx_mail_auth_http_commands,           /* module directives */
128     NGX_MAIL_MODULE,                       /* module type */
129     NULL,                                   /* init master */
130     NULL,                                   /* init module */
131     NULL,                                   /* init process */
132     NULL,                                   /* init thread */
133     NULL,                                   /* exit thread */
134     NULL,                                   /* exit process */
135     NULL,                                   /* exit master */
136     NGX_MODULE_V1_PADDING
137 };
138
139
140 static ngx_str_t  ngx_mail_auth_http_method[] = {
141     ngx_string("plain"),
142     ngx_string("plain"),
143     ngx_string("plain"),
144     ngx_string("apop"),
145     ngx_string("cram-md5"),
146     ngx_string("none")
147 };
148
149 static ngx_str_t  ngx_mail_smtp_errcode = ngx_string("535 5.7.0");
150
151
152 void
153 ngx_mail_auth_http_init(ngx_mail_session_t *s)
154 {
155     ngx_int_t      rc;

```

```

156     ngx\_pool\_t *pool;
157     ngx\_mail\_auth\_http\_ctx\_t *ctx;
158     ngx\_mail\_auth\_http\_conf\_t *ahcf;
159
160     s->connection->log->action = "in http auth state";
161
162     pool = ngx\_create\_pool(2048, s->connection->log);
163     if (pool == NULL) {
164         ngx\_mail\_session\_internal\_server\_error(s);
165         return;
166     }
167
168     ctx = ngx\_palloc(pool, sizeof(ngx\_mail\_auth\_http\_ctx\_t));
169     if (ctx == NULL) {
170         ngx\_destroy\_pool(pool);
171         ngx\_mail\_session\_internal\_server\_error(s);
172         return;
173     }
174
175     ctx->pool = pool;
176
177     ahcf = ngx\_mail\_get\_module\_srv\_conf(s, ngx\_mail\_auth\_http\_module);
178
179     ctx->request = ngx\_mail\_auth\_http\_create\_request(s, pool, ahcf);
180     if (ctx->request == NULL) {
181         ngx\_destroy\_pool(ctx->pool);
182         ngx\_mail\_session\_internal\_server\_error(s);
183         return;
184     }
185
186     ngx\_mail\_set\_ctx(s, ctx, ngx\_mail\_auth\_http\_module);
187
188     ctx->peer.sockaddr = ahcf->peer->sockaddr;
189     ctx->peer.socklen = ahcf->peer->socklen;
190     ctx->peer.name = &ahcf->peer->name;
191     ctx->peer.get = ngx\_event\_get\_peer;
192     ctx->peer.log = s->connection->log;
193     ctx->peer.log_error = NGX_ERROR_ERR;
194
195     rc = ngx\_event\_connect\_peer(&ctx->peer);
196
197     if (rc == NGX\_ERROR || rc == NGX\_BUSY || rc == NGX\_DECLINED) {
198         if (ctx->peer.connection) {
199             ngx\_close\_connection(ctx->peer.connection);
200         }
201
202         ngx\_destroy\_pool(ctx->pool);
203         ngx\_mail\_session\_internal\_server\_error(s);
204         return;
205     }
206
207     ctx->peer.connection->data = s;
208     ctx->peer.connection->pool = s->connection->pool;
209
210     s->connection->read->handler = ngx\_mail\_auth\_http\_block\_read;
211     ctx->peer.connection->read->handler = ngx\_mail\_auth\_http\_read\_handler;
212     ctx->peer.connection->write->handler = ngx\_mail\_auth\_http\_write\_handler;
213
214     ctx->handler = ngx\_mail\_auth\_http\_ignore\_status\_line;
215
216     ngx\_add\_timer(ctx->peer.connection->read, ahcf->timeout);
217     ngx\_add\_timer(ctx->peer.connection->write, ahcf->timeout);
218
219     if (rc == NGX\_OK) {
220         ngx\_mail\_auth\_http\_write\_handler(ctx->peer.connection->write);
221         return;
222     }
223 }
224
225
226 static void
227 ngx\_mail\_auth\_http\_write\_handler(ngx\_event\_t *wev)
228 {
229     ssize_t n, size;
230     ngx\_connection\_t *c;
231     ngx\_mail\_session\_t *s;

```

```

232 ngx\_mail\_auth\_http\_ctx\_t *ctx;
233 ngx\_mail\_auth\_http\_conf\_t *ahcf;
234
235 c = wev->data;
236 s = c->data;
237
238 ctx = ngx\_mail\_get\_module\_ctx(s, ngx\_mail\_auth\_http\_module);
239
240 ngx\_log\_debug0(NGX\_LOG\_DEBUG\_MAIL, wev->log, 0,
241 "mail auth http write handler");
242
243 if (wev->timedout) {
244 ngx\_log\_error(NGX\_LOG\_ERR, wev->log, NGX\_ETIMEDOUT,
245 "auth http server %V timed out", ctx->peer.name);
246 ngx\_close\_connection(c);
247 ngx\_destroy\_pool(ctx->pool);
248 ngx\_mail\_session\_internal\_server\_error(s);
249 return;
250 }
251
252 size = ctx->request->last - ctx->request->pos;
253
254 n = ngx\_send(c, ctx->request->pos, size);
255
256 if (n == NGX\_ERROR) {
257 ngx\_close\_connection(c);
258 ngx\_destroy\_pool(ctx->pool);
259 ngx\_mail\_session\_internal\_server\_error(s);
260 return;
261 }
262
263 if (n > 0) {
264 ctx->request->pos += n;
265
266 if (n == size) {
267 wev->handler = ngx\_mail\_auth\_http\_dummy\_handler;
268
269 if (wev->timer_set) {
270 ngx\_del\_timer(wev);
271 }
272
273 if (ngx\_handle\_write\_event(wev, 0) != NGX\_OK) {
274 ngx\_close\_connection(c);
275 ngx\_destroy\_pool(ctx->pool);
276 ngx\_mail\_session\_internal\_server\_error(s);
277 }
278
279 return;
280 }
281 }
282
283 if (!wev->timer_set) {
284 ahcf = ngx\_mail\_get\_module\_srv\_conf(s, ngx\_mail\_auth\_http\_module);
285 ngx\_add\_timer(wev, ahcf->timeout);
286 }
287 }
288
289
290 static void
291 ngx\_mail\_auth\_http\_read\_handler(ngx\_event\_t *rev)
292 {
293 ssize\_t n, size;
294 ngx\_connection\_t *c;
295 ngx\_mail\_session\_t *s;
296 ngx\_mail\_auth\_http\_ctx\_t *ctx;
297
298 c = rev->data;
299 s = c->data;
300
301 ngx\_log\_debug0(NGX\_LOG\_DEBUG\_MAIL, rev->log, 0,
302 "mail auth http read handler");
303
304 ctx = ngx\_mail\_get\_module\_ctx(s, ngx\_mail\_auth\_http\_module);
305
306 if (rev->timedout) {
307 ngx\_log\_error(NGX\_LOG\_ERR, rev->log, NGX\_ETIMEDOUT,

```

```

308         "auth http server %V timed out", ctx->peer.name);
309     ngx_close_connection(c);
310     ngx_destroy_pool(ctx->pool);
311     ngx_mail_session_internal_server_error(s);
312     return;
313 }
314
315 if (ctx->response == NULL) {
316     ctx->response = ngx_create_temp_buf(ctx->pool, 1024);
317     if (ctx->response == NULL) {
318         ngx_close_connection(c);
319         ngx_destroy_pool(ctx->pool);
320         ngx_mail_session_internal_server_error(s);
321         return;
322     }
323 }
324
325 size = ctx->response->end - ctx->response->last;
326
327 n = ngx_recv(c, ctx->response->pos, size);
328
329 if (n > 0) {
330     ctx->response->last += n;
331
332     ctx->handler(s, ctx);
333     return;
334 }
335
336 if (n == NGX_AGAIN) {
337     return;
338 }
339
340 ngx_close_connection(c);
341 ngx_destroy_pool(ctx->pool);
342 ngx_mail_session_internal_server_error(s);
343 }
344
345
346 static void
347 ngx_mail_auth_http_ignore_status_line(ngx_mail_session_t *s,
348 ngx_mail_auth_http_ctx_t *ctx)
349 {
350     u_char *p, ch;
351     enum {
352         sw_start = 0,
353         sw_H,
354         sw_HT,
355         sw_HTTP,
356         sw_HTTP,
357         sw_skip,
358         sw_almost_done
359     } state;
360
361     ngx_log_debug0(NGX_LOG_DEBUG_MAIL, s->connection->log, 0,
362         "mail auth http process status line");
363
364     state = ctx->state;
365
366     for (p = ctx->response->pos; p < ctx->response->last; p++) {
367         ch = *p;
368
369         switch (state) {
370
371             /* "HTTP/" */
372             case sw_start:
373                 if (ch == 'H') {
374                     state = sw_H;
375                     break;
376                 }
377                 goto next;
378
379             case sw_H:
380                 if (ch == 'T') {
381                     state = sw_HT;
382                     break;
383                 }

```

```

384         goto next;
385
386     case sw_HT:
387         if (ch == 'T') {
388             state = sw_HTTP;
389             break;
390         }
391         goto next;
392
393     case sw_HTTP:
394         if (ch == 'P') {
395             state = sw_HTTP;
396             break;
397         }
398         goto next;
399
400     case sw_HTTP:
401         if (ch == '/') {
402             state = sw_skip;
403             break;
404         }
405         goto next;
406
407     /* any text until end of line */
408     case sw_skip:
409         switch (ch) {
410             case CR:
411                 state = sw_almost_done;
412
413                 break;
414             case LF:
415                 goto done;
416         }
417         break;
418
419     /* end of status line */
420     case sw_almost_done:
421         if (ch == LF) {
422             goto done;
423         }
424
425         ngx_log_error(NGX_LOG_ERR, s->connection->log, 0,
426                     "auth http server &V sent invalid response",
427                     ctx->peer.name);
428         ngx_close_connection(ctx->peer.connection);
429         ngx_destroy_pool(ctx->pool);
430         ngx_mail_session_internal_server_error(s);
431         return;
432     }
433 }
434
435 ctx->response->pos = p;
436 ctx->state = state;
437
438 return;
439
440 next:
441
442     p = ctx->response->start - 1;
443
444 done:
445
446     ctx->response->pos = p + 1;
447     ctx->state = 0;
448     ctx->handler = ngx_mail_auth_http_process_headers;
449     ctx->handler(s, ctx);
450 }
451
452
453 static void
454 ngx_mail_auth_http_process_headers(ngx_mail_session_t *s,
455 ngx_mail_auth_http_ctx_t *ctx)
456 {
457     u_char          *p;
458     time_t          timer;
459     size_t          len, size;

```





```

536     p = ngx_pnalloc(s->connection->pool, size);
537     if (p == NULL) {
538         ngx_close_connection(ctx->peer.connection);
539         ngx_destroy_pool(ctx->pool);
540         ngx_mail_session_internal_server_error(s);
541         return;
542     }
543
544     ctx->err.data = p;
545
546     switch (s->protocol) {
547
548     case NGX_MAIL_POP3_PROTOCOL:
549         *p++ = '-'; *p++ = 'E'; *p++ = 'R'; *p++ = 'R'; *p++ = ' ';
550         break;
551
552     case NGX_MAIL_IMAP_PROTOCOL:
553         p = ngx_cpymem(p, s->tag.data, s->tag.len);
554         *p++ = 'N'; *p++ = 'O'; *p++ = ' ';
555         break;
556
557     default: /* NGX_MAIL_SMTP_PROTOCOL */
558         break;
559     }
560
561     p = ngx_cpymem(p, ctx->header_start, len);
562     *p++ = CR; *p++ = LF;
563
564     ctx->err.len = p - ctx->err.data;
565
566     continue;
567 }
568
569 if (len == sizeof("Auth-Server") - 1
570     && ngx_strncasecmp(ctx->header_name_start,
571                       (u_char *) "Auth-Server",
572                       sizeof("Auth-Server") - 1)
573     == 0)
574 {
575     ctx->addr.len = ctx->header_end - ctx->header_start;
576     ctx->addr.data = ctx->header_start;
577
578     continue;
579 }
580
581 if (len == sizeof("Auth-Port") - 1
582     && ngx_strncasecmp(ctx->header_name_start,
583                       (u_char *) "Auth-Port",
584                       sizeof("Auth-Port") - 1)
585     == 0)
586 {
587     ctx->port.len = ctx->header_end - ctx->header_start;
588     ctx->port.data = ctx->header_start;
589
590     continue;
591 }
592
593 if (len == sizeof("Auth-User") - 1
594     && ngx_strncasecmp(ctx->header_name_start,
595                       (u_char *) "Auth-User",
596                       sizeof("Auth-User") - 1)
597     == 0)
598 {
599     s->login.len = ctx->header_end - ctx->header_start;
600
601     s->login.data = ngx_pnalloc(s->connection->pool, s->login.len);
602     if (s->login.data == NULL) {
603         ngx_close_connection(ctx->peer.connection);
604         ngx_destroy_pool(ctx->pool);
605         ngx_mail_session_internal_server_error(s);
606         return;
607     }
608
609     ngx_memcpy(s->login.data, ctx->header_start, s->login.len);
610
611     continue;

```

```

612     }
613
614     if (len == sizeof("Auth-Pass") - 1
615         && ngx_strncasecmp(ctx->header_name_start,
616             (u_char *) "Auth-Pass",
617             sizeof("Auth-Pass") - 1)
618         == 0)
619     {
620         s->passwd.len = ctx->header_end - ctx->header_start;
621
622         s->passwd.data = ngx_pnalloc(s->connection->pool,
623             s->passwd.len);
624         if (s->passwd.data == NULL) {
625             ngx_close_connection(ctx->peer.connection);
626             ngx_destroy_pool(ctx->pool);
627             ngx_mail_session_internal_server_error(s);
628             return;
629         }
630
631         ngx_memcpy(s->passwd.data, ctx->header_start, s->passwd.len);
632
633         continue;
634     }
635
636     if (len == sizeof("Auth-Wait") - 1
637         && ngx_strncasecmp(ctx->header_name_start,
638             (u_char *) "Auth-Wait",
639             sizeof("Auth-Wait") - 1)
640         == 0)
641     {
642         n = ngx_atoi(ctx->header_start,
643             ctx->header_end - ctx->header_start);
644
645         if (n != NGX_ERROR) {
646             ctx->sleep = n;
647         }
648
649         continue;
650     }
651
652     if (len == sizeof("Auth-Error-Code") - 1
653         && ngx_strncasecmp(ctx->header_name_start,
654             (u_char *) "Auth-Error-Code",
655             sizeof("Auth-Error-Code") - 1)
656         == 0)
657     {
658         ctx->errcode.len = ctx->header_end - ctx->header_start;
659
660         ctx->errcode.data = ngx_pnalloc(s->connection->pool,
661             ctx->errcode.len);
662         if (ctx->errcode.data == NULL) {
663             ngx_close_connection(ctx->peer.connection);
664             ngx_destroy_pool(ctx->pool);
665             ngx_mail_session_internal_server_error(s);
666             return;
667         }
668
669         ngx_memcpy(ctx->errcode.data, ctx->header_start,
670             ctx->errcode.len);
671
672         continue;
673     }
674
675     /* ignore other headers */
676
677     continue;
678 }
679
680 if (rc == NGX_DONE) {
681     ngx_log_debug0(NGX_LOG_DEBUG_MAIL, s->connection->log, 0,
682         "mail auth http header done");
683
684     ngx_close_connection(ctx->peer.connection);
685
686     if (ctx->err.len) {
687

```

```

688     ngx_log_error(NGX_LOG_INFO, s->connection->log, 0,
689                 "client login failed: \"%V\"", &ctx->errmsg);
690
691     if (s->protocol == NGX_MAIL_SMTP_PROTOCOL) {
692
693         if (ctx->errcode.len == 0) {
694             ctx->errcode = ngx_mail_smtp_errcode;
695         }
696
697         ctx->err.len = ctx->errcode.len + ctx->errmsg.len
698                     + sizeof(" " CRLF) - 1;
699
700         p = ngx_pnalloc(s->connection->pool, ctx->err.len);
701         if (p == NULL) {
702             ngx_destroy_pool(ctx->pool);
703             ngx_mail_session_internal_server_error(s);
704             return;
705         }
706
707         ctx->err.data = p;
708
709         p = ngx_cpymem(p, ctx->errcode.data, ctx->errcode.len);
710         *p++ = ' ';
711         p = ngx_cpymem(p, ctx->errmsg.data, ctx->errmsg.len);
712         *p++ = CR; *p = LF;
713     }
714
715     s->out = ctx->err;
716     timer = ctx->sleep;
717
718     ngx_destroy_pool(ctx->pool);
719
720     if (timer == 0) {
721         s->quit = 1;
722         ngx_mail_send(s->connection->write);
723         return;
724     }
725
726     ngx_add_timer(s->connection->read, (ngx_msec_t) (timer * 1000));
727
728     s->connection->read->handler = ngx_mail_auth_sleep_handler;
729
730     return;
731 }
732
733 if (s->auth_wait) {
734     timer = ctx->sleep;
735
736     ngx_destroy_pool(ctx->pool);
737
738     if (timer == 0) {
739         ngx_mail_auth_http_init(s);
740         return;
741     }
742
743     ngx_add_timer(s->connection->read, (ngx_msec_t) (timer * 1000));
744
745     s->connection->read->handler = ngx_mail_auth_sleep_handler;
746
747     return;
748 }
749
750 if (ctx->addr.len == 0 || ctx->port.len == 0) {
751     ngx_log_error(NGX_LOG_ERR, s->connection->log, 0,
752                 "auth http server %V did not send server or port",
753                 ctx->peer.name);
754     ngx_destroy_pool(ctx->pool);
755     ngx_mail_session_internal_server_error(s);
756     return;
757 }
758
759 if (s->passwd.data == NULL
760     && s->protocol != NGX_MAIL_SMTP_PROTOCOL)
761 {
762     ngx_log_error(NGX_LOG_ERR, s->connection->log, 0,
763                 "auth http server %V did not send password",

```

```

764         ctx->peer.name);
765     ngx_destroy_pool(ctx->pool);
766     ngx_mail_session_internal_server_error(s);
767     return;
768 }
769
770 peer = ngx_palloc(s->connection->pool, sizeof(ngx_addr_t));
771 if (peer == NULL) {
772     ngx_destroy_pool(ctx->pool);
773     ngx_mail_session_internal_server_error(s);
774     return;
775 }
776
777 rc = ngx_parse_addr(s->connection->pool, peer,
778                  ctx->addr.data, ctx->addr.len);
779
780 switch (rc) {
781 case NGX_OK:
782     break;
783
784 case NGX_DECLINED:
785     ngx_log_error(NGX_LOG_ERR, s->connection->log, 0,
786                 "auth http server %V sent invalid server "
787                 "address:\'%V\'",
788                 ctx->peer.name, &ctx->addr);
789     /* fall through */
790
791 default:
792     ngx_destroy_pool(ctx->pool);
793     ngx_mail_session_internal_server_error(s);
794     return;
795 }
796
797 port = ngx_atoi(ctx->port.data, ctx->port.len);
798 if (port == NGX_ERROR || port < 1 || port > 65535) {
799     ngx_log_error(NGX_LOG_ERR, s->connection->log, 0,
800                 "auth http server %V sent invalid server "
801                 "port:\'%V\'",
802                 ctx->peer.name, &ctx->port);
803     ngx_destroy_pool(ctx->pool);
804     ngx_mail_session_internal_server_error(s);
805     return;
806 }
807
808 switch (peer->sockaddr->sa_family) {
809
810 #if (NGX_HAVE_INET6)
811 case AF_INET6:
812     sin6 = (struct sockaddr_in6 *) peer->sockaddr;
813     sin6->sin6_port = htons((in_port_t) port);
814     break;
815 #endif
816
817 default: /* AF_INET */
818     sin = (struct sockaddr_in *) peer->sockaddr;
819     sin->sin_port = htons((in_port_t) port);
820     break;
821 }
822
823 len = ctx->addr.len + 1 + ctx->port.len;
824
825 peer->name.len = len;
826
827 peer->name.data = ngx_pnalloc(s->connection->pool, len);
828 if (peer->name.data == NULL) {
829     ngx_destroy_pool(ctx->pool);
830     ngx_mail_session_internal_server_error(s);
831     return;
832 }
833
834 len = ctx->addr.len;
835
836 ngx_memcpy(peer->name.data, ctx->addr.data, len);
837
838 peer->name.data[len++] = ':';
839

```

```

840         ngx_memcpy(peer->name.data + len, ctx->port.data, ctx->port.len);
841
842         ngx_destroy_pool(ctx->pool);
843         ngx_mail_proxy_init(s, peer);
844
845         return;
846     }
847
848     if (rc == NGX_AGAIN ) {
849         return;
850     }
851
852     /* rc == NGX_ERROR */
853
854     ngx_log_error(NGX_LOG_ERR, s->connection->log, 0,
855                 "auth http server %V sent invalid header in response",
856                 ctx->peer.name);
857     ngx_close_connection(ctx->peer.connection);
858     ngx_destroy_pool(ctx->pool);
859     ngx_mail_session_internal_server_error(s);
860
861     return;
862 }
863 }
864
865
866 static void
867 ngx_mail_auth_sleep_handler(ngx_event_t *rev)
868 {
869     ngx_connection_t      *c;
870     ngx_mail_session_t    *s;
871     ngx_mail_core_srv_conf_t  *cscf;
872
873     ngx_log_debug0(NGX_LOG_DEBUG_MAIL, rev->log, 0, "mail auth sleep handler");
874
875     c = rev->data;
876     s = c->data;
877
878     if (rev->timedout) {
879
880         rev->timedout = 0;
881
882         if (s->auth_wait) {
883             s->auth_wait = 0;
884             ngx_mail_auth_http_init(s);
885             return;
886         }
887
888         cscf = ngx_mail_get_module_srv_conf(s, ngx_mail_core_module);
889
890         rev->handler = cscf->protocol->auth_state;
891
892         s->mail_state = 0;
893         s->auth_method = NGX_MAIL_AUTH_PLAIN;
894
895         c->log->action = "in auth state";
896
897         ngx_mail_send(c->write);
898
899         if (c->destroyed) {
900             return;
901         }
902
903         ngx_add_timer(rev, cscf->timeout);
904
905         if (rev->ready) {
906             rev->handler(rev);
907             return;
908         }
909
910         if (ngx_handle_read_event(rev, 0) != NGX_OK) {
911             ngx_mail_close_connection(c);
912         }
913
914         return;
915     }

```



```

992     if (ch >= '0' && ch <= '9') {
993         break;
994     }
995
996     if (ch == CR) {
997         ctx->header_name_end = p;
998         ctx->header_start = p;
999         ctx->header_end = p;
1000         state = sw_almost_done;
1001         break;
1002     }
1003
1004     if (ch == LF) {
1005         ctx->header_name_end = p;
1006         ctx->header_start = p;
1007         ctx->header_end = p;
1008         goto done;
1009     }
1010
1011     return NGX_ERROR;
1012
1013     /* space* before header value */
1014     case sw_space_before_value:
1015         switch (ch) {
1016             case ' ':
1017                 break;
1018             case CR:
1019                 ctx->header_start = p;
1020                 ctx->header_end = p;
1021                 state = sw_almost_done;
1022                 break;
1023             case LF:
1024                 ctx->header_start = p;
1025                 ctx->header_end = p;
1026                 goto done;
1027             default:
1028                 ctx->header_start = p;
1029                 state = sw_value;
1030                 break;
1031         }
1032         break;
1033
1034     /* header value */
1035     case sw_value:
1036         switch (ch) {
1037             case ' ':
1038                 ctx->header_end = p;
1039                 state = sw_space_after_value;
1040                 break;
1041             case CR:
1042                 ctx->header_end = p;
1043                 state = sw_almost_done;
1044                 break;
1045             case LF:
1046                 ctx->header_end = p;
1047                 goto done;
1048         }
1049         break;
1050
1051     /* space* before end of header line */
1052     case sw_space_after_value:
1053         switch (ch) {
1054             case ' ':
1055                 break;
1056             case CR:
1057                 state = sw_almost_done;
1058                 break;
1059             case LF:
1060                 goto done;
1061             default:
1062                 state = sw_value;
1063                 break;
1064         }
1065         break;
1066
1067     /* end of header line */

```



```

1068     case sw_almost_done:
1069         switch (ch) {
1070             case LF:
1071                 goto done;
1072             default:
1073                 return NGX\_ERROR;
1074         }
1075
1076         /* end of header */
1077     case sw_header_almost_done:
1078         switch (ch) {
1079             case LF:
1080                 goto header_done;
1081             default:
1082                 return NGX\_ERROR;
1083         }
1084     }
1085 }
1086
1087 ctx->response->pos = p;
1088 ctx->state = state;
1089
1090 return NGX\_AGAIN;
1091
1092 done:
1093
1094 ctx->response->pos = p + 1;
1095 ctx->state = sw_start;
1096
1097 return NGX\_OK;
1098
1099 header_done:
1100
1101 ctx->response->pos = p + 1;
1102 ctx->state = sw_start;
1103
1104 return NGX\_DONE;
1105 }
1106
1107
1108 static void
1109 ngx_mail_auth_http_block_read(ngx\_event\_t *rev)
1110 {
1111     ngx\_connection\_t *c;
1112     ngx\_mail\_session\_t *s;
1113     ngx\_mail\_auth\_http\_ctx\_t *ctx;
1114
1115     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_MAIL, rev->log, 0,
1116                 "mail auth http block read");
1117
1118     if (ngx\_handle\_read\_event(rev, 0) != NGX\_OK) {
1119         c = rev->data;
1120         s = c->data;
1121
1122         ctx = ngx\_mail\_get\_module\_ctx(s, ngx\_mail\_auth\_http\_module);
1123
1124         ngx\_close\_connection(ctx->peer.connection);
1125         ngx\_destroy\_pool(ctx->pool);
1126         ngx\_mail\_session\_internal\_server\_error(s);
1127     }
1128 }
1129
1130
1131 static void
1132 ngx_mail_auth_http_dummy_handler(ngx\_event\_t *ev)
1133 {
1134     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_MAIL, ev->log, 0,
1135                 "mail auth http dummy handler");
1136 }
1137
1138
1139 static ngx\_buf\_t *
1140 ngx_mail_auth_http_create_request(ngx\_mail\_session\_t *s, ngx\_pool\_t *pool,
1141     ngx\_mail\_auth\_http\_conf\_t *ahcf)
1142 {
1143     size_t len;

```

```

1144 ngx_buf_t *b;
1145 ngx_str_t login, passwd;
1146 ngx_mail_core_srv_conf_t *cscf;
1147
1148 if (ngx_mail_auth_http_escape(pool, &s->login, &login) != NGX_OK) {
1149     return NULL;
1150 }
1151
1152 if (ngx_mail_auth_http_escape(pool, &s->passwd, &passwd) != NGX_OK) {
1153     return NULL;
1154 }
1155
1156 cscf = ngx_mail_get_module_srv_conf(s, ngx_mail_core_module);
1157
1158 len = sizeof("GET ") - 1 + ahcf->uri.len + sizeof(" HTTP/1.0" CRLF) - 1
1159     + sizeof("Host: ") - 1 + ahcf->host_header.len + sizeof(CRLF) - 1
1160     + sizeof("Auth-Method: ") - 1
1161     + ngx_mail_auth_http_method[s->auth_method].len
1162     + sizeof(CRLF) - 1
1163     + sizeof("Auth-User: ") - 1 + login.len + sizeof(CRLF) - 1
1164     + sizeof("Auth-Pass: ") - 1 + passwd.len + sizeof(CRLF) - 1
1165     + sizeof("Auth-Salt: ") - 1 + s->salt.len
1166     + sizeof("Auth-Protocol: ") - 1 + cscf->protocol->name.len
1167     + sizeof(CRLF) - 1
1168     + sizeof("Auth-Login-Attempt: ") - 1 + NGX_INT_T_LEN
1169     + sizeof(CRLF) - 1
1170     + sizeof("Client-IP: ") - 1 + s->connection->addr_text.len
1171     + sizeof(CRLF) - 1
1172     + sizeof("Client-Host: ") - 1 + s->host.len + sizeof(CRLF) - 1
1173     + sizeof("Auth-SMTP-Helo: ") - 1 + s->smtp_helo.len
1174     + sizeof("Auth-SMTP-From: ") - 1 + s->smtp_from.len
1175     + sizeof("Auth-SMTP-To: ") - 1 + s->smtp_to.len
1176     + ahcf->header.len
1177     + sizeof(CRLF) - 1;
1178
1179 b = ngx_create_temp_buf(pool, len);
1180 if (b == NULL) {
1181     return NULL;
1182 }
1183
1184 b->last = ngx_cpymem(b->last, "GET ", sizeof("GET ") - 1);
1185 b->last = ngx_copy(b->last, ahcf->uri.data, ahcf->uri.len);
1186 b->last = ngx_cpymem(b->last, " HTTP/1.0" CRLF,
1187     sizeof(" HTTP/1.0" CRLF) - 1);
1188
1189 b->last = ngx_cpymem(b->last, "Host: ", sizeof("Host: ") - 1);
1190 b->last = ngx_copy(b->last, ahcf->host_header.data,
1191     ahcf->host_header.len);
1192 *b->last++ = CR; *b->last++ = LF;
1193
1194 b->last = ngx_cpymem(b->last, "Auth-Method: ",
1195     sizeof("Auth-Method: ") - 1);
1196 b->last = ngx_cpymem(b->last,
1197     ngx_mail_auth_http_method[s->auth_method].data,
1198     ngx_mail_auth_http_method[s->auth_method].len);
1199 *b->last++ = CR; *b->last++ = LF;
1200
1201 b->last = ngx_cpymem(b->last, "Auth-User: ", sizeof("Auth-User: ") - 1);
1202 b->last = ngx_copy(b->last, login.data, login.len);
1203 *b->last++ = CR; *b->last++ = LF;
1204
1205 b->last = ngx_cpymem(b->last, "Auth-Pass: ", sizeof("Auth-Pass: ") - 1);
1206 b->last = ngx_copy(b->last, passwd.data, passwd.len);
1207 *b->last++ = CR; *b->last++ = LF;
1208
1209 if (s->auth_method != NGX_MAIL_AUTH_PLAIN && s->salt.len) {
1210     b->last = ngx_cpymem(b->last, "Auth-Salt: ", sizeof("Auth-Salt: ") - 1);
1211     b->last = ngx_copy(b->last, s->salt.data, s->salt.len);
1212
1213     s->passwd.data = NULL;
1214 }
1215
1216 b->last = ngx_cpymem(b->last, "Auth-Protocol: ",
1217     sizeof("Auth-Protocol: ") - 1);
1218 b->last = ngx_cpymem(b->last, cscf->protocol->name.data,
1219     cscf->protocol->name.len);

```

```

1220 *b->last++ = CR; *b->last++ = LF;
1221
1222 b->last = ngx_sprintf(b->last, "Auth-Login-Attempt: %ui" CRLF,
1223 s->login_attempt);
1224
1225 b->last = ngx_cpymem(b->last, "Client-IP: ", sizeof("Client-IP: ") - 1);
1226 b->last = ngx_copy(b->last, s->connection->addr_text.data,
1227 s->connection->addr_text.len);
1228 *b->last++ = CR; *b->last++ = LF;
1229
1230 if (s->host.len) {
1231     b->last = ngx_cpymem(b->last, "Client-Host: ",
1232 sizeof("Client-Host: ") - 1);
1233     b->last = ngx_copy(b->last, s->host.data, s->host.len);
1234     *b->last++ = CR; *b->last++ = LF;
1235 }
1236
1237 if (s->auth_method == NGX_MAIL_AUTH_NONE) {
1238
1239     /* HELO, MAIL FROM, and RCPT TO can't contain CRLF, no need to escape */
1240
1241     b->last = ngx_cpymem(b->last, "Auth-SMTP-Helo: ",
1242 sizeof("Auth-SMTP-Helo: ") - 1);
1243     b->last = ngx_copy(b->last, s->smtp_helo.data, s->smtp_helo.len);
1244     *b->last++ = CR; *b->last++ = LF;
1245
1246     b->last = ngx_cpymem(b->last, "Auth-SMTP-From: ",
1247 sizeof("Auth-SMTP-From: ") - 1);
1248     b->last = ngx_copy(b->last, s->smtp_from.data, s->smtp_from.len);
1249     *b->last++ = CR; *b->last++ = LF;
1250
1251     b->last = ngx_cpymem(b->last, "Auth-SMTP-To: ",
1252 sizeof("Auth-SMTP-To: ") - 1);
1253     b->last = ngx_copy(b->last, s->smtp_to.data, s->smtp_to.len);
1254     *b->last++ = CR; *b->last++ = LF;
1255 }
1256
1257
1258 if (ahcf->header.len) {
1259     b->last = ngx_copy(b->last, ahcf->header.data, ahcf->header.len);
1260 }
1261
1262 /* add "\r\n" at the header end */
1263 *b->last++ = CR; *b->last++ = LF;
1264
1265 #if (NGX_DEBUG_MAIL_PASSWD)
1266 {
1267     ngx_str_t l;
1268
1269     l.len = b->last - b->pos;
1270     l.data = b->pos;
1271     ngx_log_debug1(NGX_LOG_DEBUG_MAIL, s->connection->log, 0,
1272 "mail auth http header:%N\"%V\"", &l);
1273 }
1274 #endif
1275
1276     return b;
1277 }
1278
1279
1280 static ngx_int_t
1281 ngx_mail_auth_http_escape(ngx_pool_t *pool, ngx_str_t *text, ngx_str_t *escaped)
1282 {
1283     u_char *p;
1284     uintptr_t n;
1285
1286     n = ngx_escape_uri(NULL, text->data, text->len, NGX_ESCAPE_MAIL_AUTH);
1287
1288     if (n == 0) {
1289         *escaped = *text;
1290         return NGX_OK;
1291     }
1292
1293     escaped->len = text->len + n * 2;
1294
1295     p = ngx_pnalloc(pool, escaped->len);

```

```

1296     if (p == NULL) {
1297         return NGX\_ERROR;
1298     }
1299
1300     (void) ngx\_escape\_uri(p, text->data, text->len, NGX\_ESCAPE\_MAIL\_AUTH);
1301
1302     escaped->data = p;
1303
1304     return NGX\_OK;
1305 }
1306
1307
1308 static void *
1309 ngx\_mail\_auth\_http\_create\_conf(ngx\_conf\_t *cf)
1310 {
1311     ngx\_mail\_auth\_http\_conf\_t *ahcf;
1312
1313     ahcf = ngx\_palloc(cf->pool, sizeof(ngx\_mail\_auth\_http\_conf\_t));
1314     if (ahcf == NULL) {
1315         return NULL;
1316     }
1317
1318     ahcf->timeout = NGX\_CONF\_UNSET\_MSEC;
1319
1320     ahcf->file = cf->conf_file->file.name.data;
1321     ahcf->line = cf->conf_file->line;
1322
1323     return ahcf;
1324 }
1325
1326
1327 static char *
1328 ngx\_mail\_auth\_http\_merge\_conf(ngx\_conf\_t *cf, void *parent, void *child)
1329 {
1330     ngx\_mail\_auth\_http\_conf\_t *prev = parent;
1331     ngx\_mail\_auth\_http\_conf\_t *conf = child;
1332
1333     u_char *p;
1334     size_t len;
1335     ngx\_uint\_t i;
1336     ngx\_table\_elt\_t *header;
1337
1338     if (conf->peer == NULL) {
1339         conf->peer = prev->peer;
1340         conf->host_header = prev->host_header;
1341         conf->uri = prev->uri;
1342
1343         if (conf->peer == NULL) {
1344             ngx\_log\_error(NGX\_LOG\_EMERG, cf->log, 0,
1345                 "no \"auth_http\" is defined for server in %s:%ui",
1346                 conf->file, conf->line);
1347
1348             return NGX\_CONF\_ERROR;
1349         }
1350     }
1351
1352     ngx\_conf\_merge\_msec\_value(conf->timeout, prev->timeout, 60000);
1353
1354     if (conf->headers == NULL) {
1355         conf->headers = prev->headers;
1356         conf->header = prev->header;
1357     }
1358
1359     if (conf->headers && conf->header.len == 0) {
1360         len = 0;
1361         header = conf->headers->elts;
1362         for (i = 0; i < conf->headers->nelts; i++) {
1363             len += header[i].key.len + 2 + header[i].value.len + 2;
1364         }
1365
1366         p = ngx\_pnalloc(cf->pool, len);
1367         if (p == NULL) {
1368             return NGX\_CONF\_ERROR;
1369         }
1370
1371         conf->header.len = len;

```

```

1372     conf->header.data = p;
1373
1374     for (i = 0; i < conf->headers->nelts; i++) {
1375         p = ngx_cpymem(p, header[i].key.data, header[i].key.len);
1376         *p++ = ':'; *p++ = ' ';
1377         p = ngx_cpymem(p, header[i].value.data, header[i].value.len);
1378         *p++ = CR; *p++ = LF;
1379     }
1380 }
1381
1382 return NGX_CONF_OK;
1383 }
1384
1385
1386 static char *
1387 ngx_mail_auth_http(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
1388 {
1389     ngx_mail_auth_http_conf_t *ahcf = conf;
1390
1391     ngx_str_t *value;
1392     ngx_url_t u;
1393
1394     value = cf->args->elts;
1395
1396     ngx_memzero(&u, sizeof(ngx_url_t));
1397
1398     u.url = value[1];
1399     u.default_port = 80;
1400     u.uri_part = 1;
1401
1402     if (ngx_strncmp(u.url.data, "http://", 7) == 0) {
1403         u.url.len -= 7;
1404         u.url.data += 7;
1405     }
1406
1407     if (ngx_parse_url(cf->pool, &u) != NGX_OK) {
1408         if (u.err) {
1409             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1410                 "%s in auth_http \"%V\"", u.err, &u.url);
1411         }
1412
1413         return NGX_CONF_ERROR;
1414     }
1415
1416     ahcf->peer = u.addrs;
1417
1418     if (u.family != AF_UNIX) {
1419         ahcf->host_header = u.host;
1420
1421     } else {
1422         ngx_str_set(&ahcf->host_header, "localhost");
1423     }
1424
1425     ahcf->uri = u.uri;
1426
1427     if (ahcf->uri.len == 0) {
1428         ngx_str_set(&ahcf->uri, "/");
1429     }
1430
1431     return NGX_CONF_OK;
1432 }
1433
1434
1435 static char *
1436 ngx_mail_auth_http_header(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
1437 {
1438     ngx_mail_auth_http_conf_t *ahcf = conf;
1439
1440     ngx_str_t *value;
1441     ngx_table_elt_t *header;
1442
1443     if (ahcf->headers == NULL) {
1444         ahcf->headers = ngx_array_create(cf->pool, 1, sizeof(ngx_table_elt_t));
1445         if (ahcf->headers == NULL) {
1446             return NGX_CONF_ERROR;
1447         }

```

```
1448     }
1449
1450     header = ngx_array_push(ahcf->headers);
1451     if (header == NULL) {
1452         return NGX_CONF_ERROR;
1453     }
1454
1455     value = cf->args->elts;
1456
1457     header->key = value[1];
1458     header->value = value[2];
1459
1460     return NGX_CONF_OK;
1461 }
```

[One Level Up](#)

[Top Level](#)

# src/mail/nginx\_mail\_proxy\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_mail\\_proxy\\_commands](#)
- [ngx\\_mail\\_proxy\\_module](#)
- [ngx\\_mail\\_proxy\\_module\\_ctx](#)
- [smtp\\_auth\\_ok](#)

## Data types defined

- [ngx\\_mail\\_proxy\\_conf\\_t](#)

## Functions defined

- [ngx\\_mail\\_proxy\\_block\\_read](#)
- [ngx\\_mail\\_proxy\\_close\\_session](#)
- [ngx\\_mail\\_proxy\\_create\\_conf](#)
- [ngx\\_mail\\_proxy\\_dummy\\_handler](#)
- [ngx\\_mail\\_proxy\\_handler](#)
- [ngx\\_mail\\_proxy\\_imap\\_handler](#)
- [ngx\\_mail\\_proxy\\_init](#)
- [ngx\\_mail\\_proxy\\_internal\\_server\\_error](#)
- [ngx\\_mail\\_proxy\\_merge\\_conf](#)
- [ngx\\_mail\\_proxy\\_pop3\\_handler](#)
- [ngx\\_mail\\_proxy\\_read\\_response](#)
- [ngx\\_mail\\_proxy\\_smtp\\_handler](#)
- [ngx\\_mail\\_proxy\\_upstream\\_error](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11 #include <ngx_event_connect.h>
12 #include <ngx_mail.h>
13
14
15 typedef struct {
16     ngx_flag_t  enable;
17     ngx_flag_t  pass_error_message;
```

```

18     ngx_flag_t  xclient;
19     size_t      buffer_size;
20     ngx_msec_t  timeout;
21 } ngx_mail_proxy_conf_t;
22
23
24 static void ngx_mail_proxy_block_read(ngx_event_t *rev);
25 static void ngx_mail_proxy_pop3_handler(ngx_event_t *rev);
26 static void ngx_mail_proxy_imap_handler(ngx_event_t *rev);
27 static void ngx_mail_proxy_smtp_handler(ngx_event_t *rev);
28 static void ngx_mail_proxy_dummy_handler(ngx_event_t *ev);
29 static ngx_int_t ngx_mail_proxy_read_response(ngx_mail_session_t *s,
30     ngx_uint_t state);
31 static void ngx_mail_proxy_handler(ngx_event_t *ev);
32 static void ngx_mail_proxy_upstream_error(ngx_mail_session_t *s);
33 static void ngx_mail_proxy_internal_server_error(ngx_mail_session_t *s);
34 static void ngx_mail_proxy_close_session(ngx_mail_session_t *s);
35 static void *ngx_mail_proxy_create_conf(ngx_conf_t *cf);
36 static char *ngx_mail_proxy_merge_conf(ngx_conf_t *cf, void *parent,
37     void *child);
38
39
40 static ngx_command_t  ngx_mail_proxy_commands[] = {
41
42     { ngx_string("proxy"),
43       NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_FLAG,
44       ngx_conf_set_flag_slot,
45       NGX_MAIL_SRV_CONF_OFFSET,
46       offsetof(ngx_mail_proxy_conf_t, enable),
47       NULL },
48
49     { ngx_string("proxy_buffer"),
50       NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_TAKE1,
51       ngx_conf_set_size_slot,
52       NGX_MAIL_SRV_CONF_OFFSET,
53       offsetof(ngx_mail_proxy_conf_t, buffer_size),
54       NULL },
55
56     { ngx_string("proxy_timeout"),
57       NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_TAKE1,
58       ngx_conf_set_msec_slot,
59       NGX_MAIL_SRV_CONF_OFFSET,
60       offsetof(ngx_mail_proxy_conf_t, timeout),
61       NULL },
62
63     { ngx_string("proxy_pass_error_message"),
64       NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_FLAG,
65       ngx_conf_set_flag_slot,
66       NGX_MAIL_SRV_CONF_OFFSET,
67       offsetof(ngx_mail_proxy_conf_t, pass_error_message),
68       NULL },
69
70     { ngx_string("xclient"),
71       NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_FLAG,
72       ngx_conf_set_flag_slot,
73       NGX_MAIL_SRV_CONF_OFFSET,
74       offsetof(ngx_mail_proxy_conf_t, xclient),
75       NULL },
76
77     ngx_null_command
78 };
79
80
81 static ngx_mail_module_t  ngx_mail_proxy_module_ctx = {
82     NULL,                                     /* protocol */
83
84     NULL,                                     /* create main configuration */
85     NULL,                                     /* init main configuration */
86
87     ngx_mail_proxy_create_conf,              /* create server configuration */
88     ngx_mail_proxy_merge_conf                /* merge server configuration */
89 };
90
91
92 ngx_module_t  ngx_mail_proxy_module = {
93     NGX_MODULE_V1,

```





```

170     if (s->proxy->buffer == NULL) {
171         ngx_mail_proxy_internal_server_error(s);
172         return;
173     }
174
175     s->out.len = 0;
176
177     switch (s->protocol) {
178
179     case NGX_MAIL_POP3_PROTOCOL:
180         p->upstream.connection->read->handler = ngx_mail_proxy_pop3_handler;
181         s->mail_state = ngx_pop3_start;
182         break;
183
184     case NGX_MAIL_IMAP_PROTOCOL:
185         p->upstream.connection->read->handler = ngx_mail_proxy_imap_handler;
186         s->mail_state = ngx_imap_start;
187         break;
188
189     default: /* NGX_MAIL_SMTP_PROTOCOL */
190         p->upstream.connection->read->handler = ngx_mail_proxy_smtp_handler;
191         s->mail_state = ngx_smtp_start;
192         break;
193     }
194 }
195
196
197 static void
198 ngx_mail_proxy_block_read(ngx_event_t *rev)
199 {
200     ngx_connection_t    *c;
201     ngx_mail_session_t  *s;
202
203     ngx_log_debug0(NGX_LOG_DEBUG_MAIL, rev->log, 0, "mail proxy block read");
204
205     if (ngx_handle_read_event(rev, 0) != NGX_OK) {
206         c = rev->data;
207         s = c->data;
208
209         ngx_mail_proxy_close_session(s);
210     }
211 }
212
213
214 static void
215 ngx_mail_proxy_pop3_handler(ngx_event_t *rev)
216 {
217     u_char                *p;
218     ngx_int_t             rc;
219     ngx_str_t             line;
220     ngx_connection_t      *c;
221     ngx_mail_session_t    *s;
222     ngx_mail_proxy_conf_t *pcf;
223
224     ngx_log_debug0(NGX_LOG_DEBUG_MAIL, rev->log, 0,
225                  "mail proxy pop3 auth handler");
226
227     c = rev->data;
228     s = c->data;
229
230     if (rev->timedout) {
231         ngx_log_error(NGX_LOG_INFO, c->log, NGX_ETIMEDOUT,
232                     "upstream timed out");
233         c->timedout = 1;
234         ngx_mail_proxy_internal_server_error(s);
235         return;
236     }
237
238     rc = ngx_mail_proxy_read_response(s, 0);
239
240     if (rc == NGX_AGAIN) {
241         return;
242     }
243
244     if (rc == NGX_ERROR) {
245         ngx_mail_proxy_upstream_error(s);

```

```

246     return;
247 }
248
249 switch (s->mail_state) {
250
251 case ngx_pop3_start:
252     ngx_log_debug0(NGX_LOG_DEBUG_MAIL, rev->log, 0, "mail proxy send user");
253
254     s->connection->log->action = "sending user name to upstream";
255
256     line.len = sizeof("USER ") - 1 + s->login.len + 2;
257     line.data = ngx_pnalloc(c->pool, line.len);
258     if (line.data == NULL) {
259         ngx_mail_proxy_internal_server_error(s);
260         return;
261     }
262
263     p = ngx_cpymem(line.data, "USER ", sizeof("USER ") - 1);
264     p = ngx_cpymem(p, s->login.data, s->login.len);
265     *p++ = CR; *p = LF;
266
267     s->mail_state = ngx_pop3_user;
268     break;
269
270 case ngx_pop3_user:
271     ngx_log_debug0(NGX_LOG_DEBUG_MAIL, rev->log, 0, "mail proxy send pass");
272
273     s->connection->log->action = "sending password to upstream";
274
275     line.len = sizeof("PASS ") - 1 + s->passwd.len + 2;
276     line.data = ngx_pnalloc(c->pool, line.len);
277     if (line.data == NULL) {
278         ngx_mail_proxy_internal_server_error(s);
279         return;
280     }
281
282     p = ngx_cpymem(line.data, "PASS ", sizeof("PASS ") - 1);
283     p = ngx_cpymem(p, s->passwd.data, s->passwd.len);
284     *p++ = CR; *p = LF;
285
286     s->mail_state = ngx_pop3_passwd;
287     break;
288
289 case ngx_pop3_passwd:
290     s->connection->read->handler = ngx_mail_proxy_handler;
291     s->connection->write->handler = ngx_mail_proxy_handler;
292     rev->handler = ngx_mail_proxy_handler;
293     c->write->handler = ngx_mail_proxy_handler;
294
295     pcf = ngx_mail_get_module_srv_conf(s, ngx_mail_proxy_module);
296     ngx_add_timer(s->connection->read, pcf->timeout);
297     ngx_del_timer(c->read);
298
299     c->log->action = NULL;
300     ngx_log_error(NGX_LOG_INFO, c->log, 0, "client logged in");
301
302     ngx_mail_proxy_handler(s->connection->write);
303
304     return;
305
306 default:
307     #if (NGX_SUPPRESS_WARN)
308         ngx_str_null(&line);
309     #endif
310     break;
311 }
312
313 if (c->send(c, line.data, line.len) < (ssize_t) line.len) {
314     /*
315      * we treat the incomplete sending as NGX_ERROR
316      * because it is very strange here
317      */
318     ngx_mail_proxy_internal_server_error(s);
319     return;
320 }
321

```

```

322     s->proxy->buffer->pos = s->proxy->buffer->start;
323     s->proxy->buffer->last = s->proxy->buffer->start;
324 }
325
326
327 static void
328 ngx_mail_proxy_imap_handler(ngx_event_t *rev)
329 {
330     u_char          *p;
331     ngx_int_t       rc;
332     ngx_str_t       line;
333     ngx_connection_t *c;
334     ngx_mail_session_t *s;
335     ngx_mail_proxy_conf_t *pcf;
336
337     ngx_log_debug0(NGX_LOG_DEBUG_MAIL, rev->log, 0,
338                  "mail proxy imap auth handler");
339
340     c = rev->data;
341     s = c->data;
342
343     if (rev->timedout) {
344         ngx_log_error(NGX_LOG_INFO, c->log, NGX_ETIMEDOUT,
345                      "upstream timed out");
346         c->timedout = 1;
347         ngx_mail_proxy_internal_server_error(s);
348         return;
349     }
350
351     rc = ngx_mail_proxy_read_response(s, s->mail_state);
352
353     if (rc == NGX_AGAIN) {
354         return;
355     }
356
357     if (rc == NGX_ERROR) {
358         ngx_mail_proxy_upstream_error(s);
359         return;
360     }
361
362     switch (s->mail_state) {
363
364     case ngx_imap_start:
365         ngx_log_debug0(NGX_LOG_DEBUG_MAIL, rev->log, 0,
366                      "mail proxy send login");
367
368         s->connection->log->action = "sending LOGIN command to upstream";
369
370         line.len = s->tag.len + sizeof("LOGIN ") - 1
371                  + 1 + NGX_SIZE_T_LEN + 1 + 2;
372         line.data = ngx_pnalloc(c->pool, line.len);
373         if (line.data == NULL) {
374             ngx_mail_proxy_internal_server_error(s);
375             return;
376         }
377
378         line.len = ngx_sprintf(line.data, "%VLOGIN {%uz}" CRLF,
379                               &s->tag, s->login.len)
380                  - line.data;
381
382         s->mail_state = ngx_imap_login;
383         break;
384
385     case ngx_imap_login:
386         ngx_log_debug0(NGX_LOG_DEBUG_MAIL, rev->log, 0, "mail proxy send user");
387
388         s->connection->log->action = "sending user name to upstream";
389
390         line.len = s->login.len + 1 + 1 + NGX_SIZE_T_LEN + 1 + 2;
391         line.data = ngx_pnalloc(c->pool, line.len);
392         if (line.data == NULL) {
393             ngx_mail_proxy_internal_server_error(s);
394             return;
395         }
396
397         line.len = ngx_sprintf(line.data, "%V {%uz}" CRLF,

```

```

398         &s->login, s->passwd.len)
399     - line.data;
400
401     s->mail_state = ngx_imap_user;
402     break;
403
404     case ngx_imap_user:
405         ngx_log_debug0(NGX_LOG_DEBUG_MAIL, rev->log, 0,
406             "mail proxy send passwd");
407
408         s->connection->log->action = "sending password to upstream";
409
410         line.len = s->passwd.len + 2;
411         line.data = ngx_pnalloc(c->pool, line.len);
412         if (line.data == NULL) {
413             ngx_mail_proxy_internal_server_error(s);
414             return;
415         }
416
417         p = ngx_cpymem(line.data, s->passwd.data, s->passwd.len);
418         *p++ = CR; *p = LF;
419
420         s->mail_state = ngx_imap_passwd;
421         break;
422
423     case ngx_imap_passwd:
424         s->connection->read->handler = ngx_mail_proxy_handler;
425         s->connection->write->handler = ngx_mail_proxy_handler;
426         rev->handler = ngx_mail_proxy_handler;
427         c->write->handler = ngx_mail_proxy_handler;
428
429         pcf = ngx_mail_get_module_srv_conf(s, ngx_mail_proxy_module);
430         ngx_add_timer(s->connection->read, pcf->timeout);
431         ngx_del_timer(c->read);
432
433         c->log->action = NULL;
434         ngx_log_error(NGX_LOG_INFO, c->log, 0, "client logged in");
435
436         ngx_mail_proxy_handler(s->connection->write);
437
438         return;
439
440     default:
441     #if (NGX_SUPPRESS_WARN)
442         ngx_str_null(&line);
443     #endif
444     break;
445 }
446
447 if (c->send(c, line.data, line.len) < (ssize_t) line.len) {
448     /*
449     * we treat the incomplete sending as NGX_ERROR
450     * because it is very strange here
451     */
452     ngx_mail_proxy_internal_server_error(s);
453     return;
454 }
455
456 s->proxy->buffer->pos = s->proxy->buffer->start;
457 s->proxy->buffer->last = s->proxy->buffer->start;
458 }
459
460
461 static void
462 ngx_mail_proxy_smtp_handler(ngx_event_t *rev)
463 {
464     u_char          *p;
465     ngx_int_t       rc;
466     ngx_str_t       line;
467     ngx_buf_t       *b;
468     ngx_connection_t *c;
469     ngx_mail_session_t *s;
470     ngx_mail_proxy_conf_t *pcf;
471     ngx_mail_core_srv_conf_t *cscf;
472
473     ngx_log_debug0(NGX_LOG_DEBUG_MAIL, rev->log, 0,

```

```

474         "mail proxy smtp auth handler");
475
476     c = rev->data;
477     s = c->data;
478
479     if (rev->timedout) {
480         ngx_log_error(NGX_LOG_INFO, c->log, NGX_ETIMEDOUT,
481             "upstream timed out");
482         c->timedout = 1;
483         ngx_mail_proxy_internal_server_error(s);
484         return;
485     }
486
487     rc = ngx_mail_proxy_read_response(s, s->mail_state);
488
489     if (rc == NGX_AGAIN) {
490         return;
491     }
492
493     if (rc == NGX_ERROR) {
494         ngx_mail_proxy_upstream_error(s);
495         return;
496     }
497
498     switch (s->mail_state) {
499
500     case ngx_smtp_start:
501         ngx_log_debug0(NGX_LOG_DEBUG_MAIL, rev->log, 0, "mail proxy send ehlo");
502
503         s->connection->log->action = "sending HELO/EHLO to upstream";
504
505         cscf = ngx_mail_get_module_srv_conf(s, ngx_mail_core_module);
506
507         line.len = sizeof("HELO ") - 1 + cscf->server_name.len + 2;
508         line.data = ngx_pnalloc(c->pool, line.len);
509         if (line.data == NULL) {
510             ngx_mail_proxy_internal_server_error(s);
511             return;
512         }
513
514         pcf = ngx_mail_get_module_srv_conf(s, ngx_mail_proxy_module);
515
516         p = ngx_cpymem(line.data,
517             ((s->esmtplib || pcf->xclient) ? "EHLO " : "HELO "),
518             sizeof("HELO ") - 1);
519
520         p = ngx_cpymem(p, cscf->server_name.data, cscf->server_name.len);
521         *p++ = CR; *p = LF;
522
523         if (pcf->xclient) {
524             s->mail_state = ngx_smtp_helo_xclient;
525
526         } else if (s->auth_method == NGX_MAIL_AUTH_NONE) {
527             s->mail_state = ngx_smtp_helo_from;
528
529         } else {
530             s->mail_state = ngx_smtp_helo;
531         }
532
533         break;
534
535     case ngx_smtp_helo_xclient:
536         ngx_log_debug0(NGX_LOG_DEBUG_MAIL, rev->log, 0,
537             "mail proxy send xclient");
538
539         s->connection->log->action = "sending XCLIENT to upstream";
540
541         line.len = sizeof("XCLIENT ADDR= LOGIN= NAME="
542             CRLF) - 1
543             + s->connection->addr_text.len + s->login.len + s->host.len;
544
545     #if (NGX_HAVE_INET6)
546         if (s->connection->sockaddr->sa_family == AF_INET6) {
547             line.len += sizeof("IPV6:") - 1;
548         }
549     #endif

```

```

550     line.data = ngx_pnalloc(c->pool, line.len);
551     if (line.data == NULL) {
552         ngx_mail_proxy_internal_server_error(s);
553         return;
554     }
555 }
556
557 p = ngx_cpymem(line.data, "XCLIENT ADDR=", sizeof("XCLIENT ADDR=") - 1);
558
559 #if (NGX_HAVE_INET6)
560     if (s->connection->sockaddr->sa_family == AF_INET6) {
561         p = ngx_cpymem(p, "IPV6:", sizeof("IPV6:") - 1);
562     }
563 #endif
564
565 p = ngx_copy(p, s->connection->addr_text.data,
566             s->connection->addr_text.len);
567
568 if (s->login.len) {
569     p = ngx_cpymem(p, " LOGIN=", sizeof(" LOGIN=") - 1);
570     p = ngx_copy(p, s->login.data, s->login.len);
571 }
572
573 p = ngx_cpymem(p, " NAME=", sizeof(" NAME=") - 1);
574 p = ngx_copy(p, s->host.data, s->host.len);
575
576 *p++ = CR; *p++ = LF;
577
578 line.len = p - line.data;
579
580 if (s->smtp_helo.len) {
581     s->mail_state = ngx_smtp_xclient_helo;
582 }
583 else if (s->auth_method == NGX_MAIL_AUTH_NONE) {
584     s->mail_state = ngx_smtp_xclient_from;
585 }
586 else {
587     s->mail_state = ngx_smtp_xclient;
588 }
589
590 break;
591
592 case ngx_smtp_xclient_helo:
593     ngx_log_debug0(NGX_LOG_DEBUG_MAIL, rev->log, 0,
594                 "mail proxy send client ehlo");
595
596     s->connection->log->action = "sending client HELO/EHLO to upstream";
597
598     line.len = sizeof("HELO " CRLF) - 1 + s->smtp_helo.len;
599
600     line.data = ngx_pnalloc(c->pool, line.len);
601     if (line.data == NULL) {
602         ngx_mail_proxy_internal_server_error(s);
603         return;
604     }
605
606     line.len = ngx_sprintf(line.data,
607                          ((s->esmtplib) ? "EHLO %V" CRLF : "HELO %V" CRLF),
608                          &s->smtp_helo)
609               - line.data;
610
611     s->mail_state = (s->auth_method == NGX_MAIL_AUTH_NONE) ?
612                   ngx_smtp_helo_from : ngx_smtp_helo;
613
614     break;
615
616 case ngx_smtp_helo_from:
617 case ngx_smtp_xclient_from:
618     ngx_log_debug0(NGX_LOG_DEBUG_MAIL, rev->log, 0,
619                 "mail proxy send mail from");
620
621     s->connection->log->action = "sending MAIL FROM to upstream";
622
623     line.len = s->smtp_from.len + sizeof(CRLF) - 1;
624     line.data = ngx_pnalloc(c->pool, line.len);
625     if (line.data == NULL) {

```

```

626     ngx_mail_proxy_internal_server_error(s);
627     return;
628 }
629
630 p = ngx_cpymem(line.data, s->smtp_from.data, s->smtp_from.len);
631 *p++ = CR; *p = LF;
632
633 s->mail_state = ngx_smtp_from;
634
635 break;
636
637 case ngx_smtp_from:
638     ngx_log_debug0(NGX_LOG_DEBUG_MAIL, rev->log, 0,
639                 "mail proxy send rcpt to");
640
641     s->connection->log->action = "sending RCPT TO to upstream";
642
643     line.len = s->smtp_to.len + sizeof(CRLF) - 1;
644     line.data = ngx_pnalloc(c->pool, line.len);
645     if (line.data == NULL) {
646         ngx_mail_proxy_internal_server_error(s);
647         return;
648     }
649
650     p = ngx_cpymem(line.data, s->smtp_to.data, s->smtp_to.len);
651     *p++ = CR; *p = LF;
652
653     s->mail_state = ngx_smtp_to;
654
655     break;
656
657 case ngx_smtp_helo:
658 case ngx_smtp_xclient:
659 case ngx_smtp_to:
660
661     b = s->proxy->buffer;
662
663     if (s->auth_method == NGX_MAIL_AUTH_NONE) {
664         b->pos = b->start;
665
666     } else {
667         ngx_memcpy(b->start, smtp_auth_ok, sizeof(smtp_auth_ok) - 1);
668         b->last = b->start + sizeof(smtp_auth_ok) - 1;
669     }
670
671     s->connection->read->handler = ngx_mail_proxy_handler;
672     s->connection->write->handler = ngx_mail_proxy_handler;
673     rev->handler = ngx_mail_proxy_handler;
674     c->write->handler = ngx_mail_proxy_handler;
675
676     pcf = ngx_mail_get_module_srv_conf(s, ngx_mail_proxy_module);
677     ngx_add_timer(s->connection->read, pcf->timeout);
678     ngx_del_timer(c->read);
679
680     c->log->action = NULL;
681     ngx_log_error(NGX_LOG_INFO, c->log, 0, "client logged in");
682
683     if (s->buffer->pos == s->buffer->last) {
684         ngx_mail_proxy_handler(s->connection->write);
685
686     } else {
687         ngx_mail_proxy_handler(c->write);
688     }
689
690     return;
691
692 default:
693 #if (NGX_SUPPRESS_WARN)
694     ngx_str_null(&line);
695 #endif
696     break;
697 }
698
699 if (c->send(c, line.data, line.len) < (ssize_t) line.len) {
700     /*
701     * we treat the incomplete sending as NGX_ERROR

```



```

702     * because it is very strange here
703     */
704     ngx_mail_proxy_internal_server_error(s);
705     return;
706 }
707
708 s->proxy->buffer->pos = s->proxy->buffer->start;
709 s->proxy->buffer->last = s->proxy->buffer->start;
710 }
711
712 static void
713 ngx_mail_proxy_dummy_handler(ngx_event_t *wev)
714 {
715     ngx_connection_t    *c;
716     ngx_mail_session_t  *s;
717
718     ngx_log_debug0(NGX_LOG_DEBUG_MAIL, wev->log, 0, "mail proxy dummy handler");
719
720     if (ngx_handle_write_event(wev, 0) != NGX_OK) {
721         c = wev->data;
722         s = c->data;
723
724         ngx_mail_proxy_close_session(s);
725     }
726 }
727
728
729 static ngx_int_t
730 ngx_mail_proxy_read_response(ngx_mail_session_t *s, ngx_uint_t state)
731 {
732     u_char                *p, *m;
733     ssize_t               n;
734     ngx_buf_t             *b;
735     ngx_mail_proxy_conf_t *pcf;
736
737     s->connection->log->action = "reading response from upstream";
738
739     b = s->proxy->buffer;
740
741     n = s->proxy->upstream.connection->recv(s->proxy->upstream.connection,
742     b->last, b->end - b->last);
743
744     if (n == NGX_ERROR || n == 0) {
745         return NGX_ERROR;
746     }
747
748     if (n == NGX_AGAIN) {
749         return NGX_AGAIN;
750     }
751
752     b->last += n;
753
754     if (b->last - b->pos < 4) {
755         return NGX_AGAIN;
756     }
757
758     if (*(b->last - 2) != CR || *(b->last - 1) != LF) {
759         if (b->last == b->end) {
760             *(b->last - 1) = '\0';
761             ngx_log_error(NGX_LOG_ERR, s->connection->log, 0,
762             "upstream sent too long response line: \"%s\"",
763             b->pos);
764             return NGX_ERROR;
765         }
766     }
767
768     return NGX_AGAIN;
769 }
770
771 p = b->pos;
772
773 switch (s->protocol) {
774
775 case NGX_MAIL_POP3_PROTOCOL:
776     if (p[0] == '+' && p[1] == '0' && p[2] == 'K') {
777         return NGX_OK;

```

```

778     }
779     break;
780
781 case NGX_MAIL_IMAP_PROTOCOL:
782     switch (state) {
783
784     case ngx_imap_start:
785         if (p[0] == '*' && p[1] == ' ' && p[2] == '0' && p[3] == 'K') {
786             return NGX_OK;
787         }
788         break;
789
790     case ngx_imap_login:
791     case ngx_imap_user:
792         if (p[0] == '+') {
793             return NGX_OK;
794         }
795         break;
796
797     case ngx_imap_passwd:
798         if (ngx_strncmp(p, s->tag.data, s->tag.len) == 0) {
799             p += s->tag.len;
800             if (p[0] == '0' && p[1] == 'K') {
801                 return NGX_OK;
802             }
803         }
804         break;
805     }
806
807     break;
808
809 default: /* NGX_MAIL_SMTP_PROTOCOL */
810
811     if (p[3] == '-') {
812         /* multiline reply, check if we got last line */
813
814         m = b->last - (sizeof(CRLF "200" CRLF) - 1);
815
816         while (m > p) {
817             if (m[0] == CR && m[1] == LF) {
818                 break;
819             }
820
821             m--;
822         }
823
824         if (m <= p || m[5] == '-') {
825             return NGX_AGAIN;
826         }
827     }
828
829     switch (state) {
830
831     case ngx_smtp_start:
832         if (p[0] == '2' && p[1] == '2' && p[2] == '0') {
833             return NGX_OK;
834         }
835         break;
836
837     case ngx_smtp_helo:
838     case ngx_smtp_helo_xclient:
839     case ngx_smtp_helo_from:
840     case ngx_smtp_from:
841         if (p[0] == '2' && p[1] == '5' && p[2] == '0') {
842             return NGX_OK;
843         }
844         break;
845
846     case ngx_smtp_xclient:
847     case ngx_smtp_xclient_from:
848     case ngx_smtp_xclient_helo:
849         if (p[0] == '2' && (p[1] == '2' || p[1] == '5') && p[2] == '0') {
850             return NGX_OK;
851         }
852         break;
853

```

```

854     case ngx_smtp_to:
855         return NGX_OK;
856     }
857
858     break;
859 }
860
861 pcf = ngx_mail_get_module_srv_conf(s, ngx_mail_proxy_module);
862
863 if (pcf->pass_error_message == 0) {
864     *(b->last - 2) = '\0';
865     ngx_log_error(NGX_LOG_ERR, s->connection->log, 0,
866         "upstream sent invalid response: \"%s\"", p);
867     return NGX_ERROR;
868 }
869
870 s->out.len = b->last - p - 2;
871 s->out.data = p;
872
873 ngx_log_error(NGX_LOG_INFO, s->connection->log, 0,
874     "upstream sent invalid response: \"%V\"", &s->out);
875
876 s->out.len = b->last - b->pos;
877 s->out.data = b->pos;
878
879 return NGX_ERROR;
880 }
881
882
883 static void
884 ngx_mail_proxy_handler(ngx_event_t *ev)
885 {
886     char                *action, *recv_action, *send_action;
887     size_t              size;
888     ssize_t             n;
889     ngx_buf_t          *b;
890     ngx_uint_t          do_write;
891     ngx_connection_t   *c, *src, *dst;
892     ngx_mail_session_t *s;
893     ngx_mail_proxy_conf_t *pcf;
894
895     c = ev->data;
896     s = c->data;
897
898     if (ev->timedout) {
899         c->log->action = "proxying";
900
901         if (c == s->connection) {
902             ngx_log_error(NGX_LOG_INFO, c->log, NGX_ETIMEDOUT,
903                 "client timed out");
904             c->timedout = 1;
905
906         } else {
907             ngx_log_error(NGX_LOG_INFO, c->log, NGX_ETIMEDOUT,
908                 "upstream timed out");
909         }
910
911         ngx_mail_proxy_close_session(s);
912         return;
913     }
914
915     if (c == s->connection) {
916         if (ev->write) {
917             recv_action = "proxying and reading from upstream";
918             send_action = "proxying and sending to client";
919             src = s->proxy->upstream.connection;
920             dst = c;
921             b = s->proxy->buffer;
922
923         } else {
924             recv_action = "proxying and reading from client";
925             send_action = "proxying and sending to upstream";
926             src = c;
927             dst = s->proxy->upstream.connection;
928             b = s->buffer;
929         }

```

```

930 } else {
931     if (ev->write) {
932         rcv_action = "proxying and reading from client";
933         send_action = "proxying and sending to upstream";
934         src = s->connection;
935         dst = c;
936         b = s->buffer;
937
938     } else {
939         rcv_action = "proxying and reading from upstream";
940         send_action = "proxying and sending to client";
941         src = c;
942         dst = s->connection;
943         b = s->proxy->buffer;
944     }
945 }
946 }
947
948 do_write = ev->write ? 1 : 0;
949
950 ngx_log_debug3(NGX_LOG_DEBUG_MAIL, ev->log, 0,
951     "mail proxy handler: %d, #%d > #%d",
952     do_write, src->fd, dst->fd);
953
954 for ( ;; ) {
955     if (do_write) {
956
957         size = b->last - b->pos;
958
959         if (size && dst->write->ready) {
960             c->log->action = send_action;
961
962             n = dst->send(dst, b->pos, size);
963
964             if (n == NGX_ERROR) {
965                 ngx_mail_proxy_close_session(s);
966                 return;
967             }
968
969             if (n > 0) {
970                 b->pos += n;
971
972                 if (b->pos == b->last) {
973                     b->pos = b->start;
974                     b->last = b->start;
975                 }
976             }
977         }
978     }
979
980     size = b->end - b->last;
981
982     if (size && src->read->ready) {
983         c->log->action = rcv_action;
984
985         n = src->recv(src, b->last, size);
986
987         if (n == NGX_AGAIN || n == 0) {
988             break;
989         }
990
991         if (n > 0) {
992             do_write = 1;
993             b->last += n;
994
995             continue;
996         }
997
998         if (n == NGX_ERROR) {
999             src->read->eof = 1;
1000         }
1001     }
1002 }
1003
1004 break;
1005 }

```

```

1006
1007     c->log->action = "proxying";
1008
1009     if ((s->connection->read->eof && s->buffer->pos == s->buffer->last)
1010         || (s->proxy->upstream.connection->read->eof
1011             && s->proxy->buffer->pos == s->proxy->buffer->last)
1012         || (s->connection->read->eof
1013             && s->proxy->upstream.connection->read->eof))
1014     {
1015         action = c->log->action;
1016         c->log->action = NULL;
1017         ngx_log_error(NGX_LOG_INFO, c->log, 0, "proxied session done");
1018         c->log->action = action;
1019
1020         ngx_mail_proxy_close_session(s);
1021         return;
1022     }
1023
1024     if (ngx_handle_write_event(dst->write, 0) != NGX_OK) {
1025         ngx_mail_proxy_close_session(s);
1026         return;
1027     }
1028
1029     if (ngx_handle_read_event(dst->read, 0) != NGX_OK) {
1030         ngx_mail_proxy_close_session(s);
1031         return;
1032     }
1033
1034     if (ngx_handle_write_event(src->write, 0) != NGX_OK) {
1035         ngx_mail_proxy_close_session(s);
1036         return;
1037     }
1038
1039     if (ngx_handle_read_event(src->read, 0) != NGX_OK) {
1040         ngx_mail_proxy_close_session(s);
1041         return;
1042     }
1043
1044     if (c == s->connection) {
1045         pcf = ngx_mail_get_module_srv_conf(s, ngx_mail_proxy_module);
1046         ngx_add_timer(c->read, pcf->timeout);
1047     }
1048 }
1049
1050
1051 static void
1052 ngx_mail_proxy_upstream_error(ngx_mail_session_t *s)
1053 {
1054     if (s->proxy->upstream.connection) {
1055         ngx_log_debug1(NGX_LOG_DEBUG_MAIL, s->connection->log, 0,
1056             "close mail proxy connection: %d",
1057             s->proxy->upstream.connection->fd);
1058
1059         ngx_close_connection(s->proxy->upstream.connection);
1060     }
1061
1062     if (s->out.len == 0) {
1063         ngx_mail_session_internal_server_error(s);
1064         return;
1065     }
1066
1067     s->quit = 1;
1068     ngx_mail_send(s->connection->write);
1069 }
1070
1071
1072 static void
1073 ngx_mail_proxy_internal_server_error(ngx_mail_session_t *s)
1074 {
1075     if (s->proxy->upstream.connection) {
1076         ngx_log_debug1(NGX_LOG_DEBUG_MAIL, s->connection->log, 0,
1077             "close mail proxy connection: %d",
1078             s->proxy->upstream.connection->fd);
1079
1080         ngx_close_connection(s->proxy->upstream.connection);
1081     }

```

```

1082     ngx\_mail\_session\_internal\_server\_error(s);
1083 }
1084 }
1085
1086
1087 static void
1088 ngx\_mail\_proxy\_close\_session(ngx\_mail\_session\_t *s)
1089 {
1090     if (s->proxy->upstream.connection) {
1091         ngx\_log\_debug1(NGX\_LOG\_DEBUG\_MAIL, s->connection->log, 0,
1092             "close mail proxy connection: %d",
1093             s->proxy->upstream.connection->fd);
1094
1095         ngx\_close\_connection(s->proxy->upstream.connection);
1096     }
1097
1098     ngx\_mail\_close\_connection(s->connection);
1099 }
1100
1101
1102 static void *
1103 ngx\_mail\_proxy\_create\_conf(ngx\_conf\_t *cf)
1104 {
1105     ngx\_mail\_proxy\_conf\_t *pcf;
1106
1107     pcf = ngx\_palloc(cf->pool, sizeof(ngx\_mail\_proxy\_conf\_t));
1108     if (pcf == NULL) {
1109         return NULL;
1110     }
1111
1112     pcf->enable = NGX\_CONF\_UNSET;
1113     pcf->pass_error_message = NGX\_CONF\_UNSET;
1114     pcf->xclient = NGX\_CONF\_UNSET;
1115     pcf->buffer_size = NGX\_CONF\_UNSET\_SIZE;
1116     pcf->timeout = NGX\_CONF\_UNSET\_MSEC;
1117
1118     return pcf;
1119 }
1120
1121
1122 static char *
1123 ngx\_mail\_proxy\_merge\_conf(ngx\_conf\_t *cf, void *parent, void *child)
1124 {
1125     ngx\_mail\_proxy\_conf\_t *prev = parent;
1126     ngx\_mail\_proxy\_conf\_t *conf = child;
1127
1128     ngx\_conf\_merge\_value(conf->enable, prev->enable, 0);
1129     ngx\_conf\_merge\_value(conf->pass_error_message, prev->pass_error_message, 0);
1130     ngx\_conf\_merge\_value(conf->xclient, prev->xclient, 1);
1131     ngx\_conf\_merge\_size\_value(conf->buffer_size, prev->buffer_size,
1132         (size_t) ngx\_pagesize);
1133     ngx\_conf\_merge\_msec\_value(conf->timeout, prev->timeout, 24 * 60 * 60000);
1134
1135     return NGX\_CONF\_OK;
1136 }

```

[One Level Up](#)

[Top Level](#)

# src/mail/nginx\_mail\_parse.c - nginx-1.7.10

## Functions defined

- [ngx\\_mail\\_auth\\_parse](#)
- [ngx\\_mail\\_imap\\_parse\\_command](#)
- [ngx\\_mail\\_pop3\\_parse\\_command](#)
- [ngx\\_mail\\_smtp\\_parse\\_command](#)

## Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_event.h>
11 #include <ngx_mail.h>
12 #include <ngx_mail_pop3_module.h>
13 #include <ngx_mail_imap_module.h>
14 #include <ngx_mail_smtp_module.h>
15
16
17 ngx_int_t
18 ngx_mail_pop3_parse_command(ngx_mail_session_t *s)
19 {
20     u_char      ch, *p, *c, c0, c1, c2, c3;
21     ngx_str_t   *arg;
22     enum {
23         sw_start = 0,
24         sw_spaces_before_argument,
25         sw_argument,
26         sw_almost_done
27     } state;
28
29     state = s->state;
30
31     for (p = s->buffer->pos; p < s->buffer->last; p++) {
32         ch = *p;
33
34         switch (state) {
35
36             /* POP3 command */
37             case sw_start:
38                 if (ch == ' ' || ch == CR || ch == LF) {
39                     c = s->buffer->start;
40
41                     if (p - c == 4) {
42
43                         c0 = ngx_toupper(c[0]);
44                         c1 = ngx_toupper(c[1]);
45                         c2 = ngx_toupper(c[2]);
46                         c3 = ngx_toupper(c[3]);
47
48                         if (c0 == 'U' && c1 == 'S' && c2 == 'E' && c3 == 'R')
49                             {
50                                 s->command = NGX_POP3_USER;
51
52                             } else if (c0 == 'P' && c1 == 'A' && c2 == 'S' && c3 == 'S')
53                             {
54                                 s->command = NGX_POP3_PASS;
55
56                             } else if (c0 == 'A' && c1 == 'P' && c2 == 'O' && c3 == 'P')
```

```

57     {
58         s->command = NGX\_POP3\_APOP;
59
60     } else if (c0 == 'Q' && c1 == 'U' && c2 == 'I' && c3 == 'T')
61     {
62         s->command = NGX\_POP3\_QUIT;
63
64     } else if (c0 == 'C' && c1 == 'A' && c2 == 'P' && c3 == 'A')
65     {
66         s->command = NGX\_POP3\_CAPA;
67
68     } else if (c0 == 'A' && c1 == 'U' && c2 == 'T' && c3 == 'H')
69     {
70         s->command = NGX\_POP3\_AUTH;
71
72     } else if (c0 == 'N' && c1 == 'O' && c2 == 'O' && c3 == 'P')
73     {
74         s->command = NGX\_POP3\_NOOP;
75 #if (NGX_MAIL_SSL)
76     } else if (c0 == 'S' && c1 == 'T' && c2 == 'L' && c3 == 'S')
77     {
78         s->command = NGX\_POP3\_STLS;
79 #endif
80     } else {
81         goto invalid;
82     }
83
84     } else {
85         goto invalid;
86     }
87
88     switch (ch) {
89     case ' ':
90         state = sw_spaces_before_argument;
91         break;
92     case CR:
93         state = sw_almost_done;
94         break;
95     case LF:
96         goto done;
97     }
98     break;
99 }
100
101 if ((ch < 'A' || ch > 'Z') && (ch < 'a' || ch > 'z')) {
102     goto invalid;
103 }
104
105 break;
106
107 case sw_spaces_before_argument:
108     switch (ch) {
109     case ' ':
110         break;
111     case CR:
112         state = sw_almost_done;
113         s->arg_end = p;
114         break;
115     case LF:
116         s->arg_end = p;
117         goto done;
118     default:
119         if (s->args.nelts <= 2) {
120             state = sw_argument;
121             s->arg_start = p;
122             break;
123         }
124         goto invalid;
125     }
126     break;
127
128 case sw_argument:
129     switch (ch) {
130
131     case ' ':
132

```



```

133      /*
134      * the space should be considered as part of the at username
135      * or password, but not of argument in other commands
136      */
137
138      if (s->command == NGX\_POP3\_USER
139          || s->command == NGX\_POP3\_PASS)
140      {
141          break;
142      }
143
144      /* fall through */
145
146      case CR:
147      case LF:
148          arg = ngx\_array\_push(&s->args);
149          if (arg == NULL) {
150              return NGX\_ERROR;
151          }
152          arg->len = p - s->arg_start;
153          arg->data = s->arg_start;
154          s->arg_start = NULL;
155
156          switch (ch) {
157              case ' ':
158                  state = sw_spaces_before_argument;
159                  break;
160              case CR:
161                  state = sw_almost_done;
162                  break;
163              case LF:
164                  goto done;
165          }
166          break;
167
168          default:
169              break;
170      }
171      break;
172
173      case sw_almost_done:
174          switch (ch) {
175              case LF:
176                  goto done;
177              default:
178                  goto invalid;
179          }
180      }
181  }
182
183  s->buffer->pos = p;
184  s->state = state;
185
186  return NGX\_AGAIN;
187
188  done:
189
190  s->buffer->pos = p + 1;
191
192  if (s->arg_start) {
193      arg = ngx\_array\_push(&s->args);
194      if (arg == NULL) {
195          return NGX\_ERROR;
196      }
197      arg->len = s->arg_end - s->arg_start;
198      arg->data = s->arg_start;
199      s->arg_start = NULL;
200  }
201
202  s->state = (s->command != NGX\_POP3\_AUTH) ? sw_start : sw_argument;
203
204  return NGX\_OK;
205
206  invalid:
207
208  s->state = sw_start;

```



```

285         && (c[1] == 'O' || c[1] == 'o')
286         && (c[2] == 'O' || c[2] == 'o')
287         && (c[3] == 'P' || c[3] == 'p'))
288     {
289         s->command = NGX_IMAP_NOOP;
290
291     } else {
292         goto invalid;
293     }
294     break;
295
296 case 5:
297     if ((c[0] == 'L' || c[0] == 'l')
298         && (c[1] == 'O' || c[1] == 'o')
299         && (c[2] == 'G' || c[2] == 'g')
300         && (c[3] == 'I' || c[3] == 'i')
301         && (c[4] == 'N' || c[4] == 'n'))
302     {
303         s->command = NGX_IMAP_LOGIN;
304
305     } else {
306         goto invalid;
307     }
308     break;
309
310 case 6:
311     if ((c[0] == 'L' || c[0] == 'l')
312         && (c[1] == 'O' || c[1] == 'o')
313         && (c[2] == 'G' || c[2] == 'g')
314         && (c[3] == 'O' || c[3] == 'o')
315         && (c[4] == 'U' || c[4] == 'u')
316         && (c[5] == 'T' || c[5] == 't'))
317     {
318         s->command = NGX_IMAP_LOGOUT;
319
320     } else {
321         goto invalid;
322     }
323     break;
324
325 #if (NGX_MAIL_SSL)
326 case 8:
327     if ((c[0] == 'S' || c[0] == 's')
328         && (c[1] == 'T' || c[1] == 't')
329         && (c[2] == 'A' || c[2] == 'a')
330         && (c[3] == 'R' || c[3] == 'r')
331         && (c[4] == 'T' || c[4] == 't')
332         && (c[5] == 'T' || c[5] == 't')
333         && (c[6] == 'L' || c[6] == 'l')
334         && (c[7] == 'S' || c[7] == 's'))
335     {
336         s->command = NGX_IMAP_STARTTLS;
337
338     } else {
339         goto invalid;
340     }
341     break;
342 #endif
343
344 case 10:
345     if ((c[0] == 'C' || c[0] == 'c')
346         && (c[1] == 'A' || c[1] == 'a')
347         && (c[2] == 'P' || c[2] == 'p')
348         && (c[3] == 'A' || c[3] == 'a')
349         && (c[4] == 'B' || c[4] == 'b')
350         && (c[5] == 'I' || c[5] == 'i')
351         && (c[6] == 'L' || c[6] == 'l')
352         && (c[7] == 'I' || c[7] == 'i')
353         && (c[8] == 'T' || c[8] == 't')
354         && (c[9] == 'Y' || c[9] == 'y'))
355     {
356         s->command = NGX_IMAP_CAPABILITY;
357
358     } else {
359         goto invalid;
360     }

```

```

361         break;
362
363     case 12:
364         if ((c[0] == 'A' || c[0] == 'a')
365             && (c[1] == 'U' || c[1] == 'u')
366             && (c[2] == 'T' || c[2] == 't')
367             && (c[3] == 'H' || c[3] == 'h')
368             && (c[4] == 'E' || c[4] == 'e')
369             && (c[5] == 'N' || c[5] == 'n')
370             && (c[6] == 'T' || c[6] == 't')
371             && (c[7] == 'I' || c[7] == 'i')
372             && (c[8] == 'C' || c[8] == 'c')
373             && (c[9] == 'A' || c[9] == 'a')
374             && (c[10] == 'T' || c[10] == 't')
375             && (c[11] == 'E' || c[11] == 'e'))
376         {
377             s->command = NGX\_IMAP\_AUTHENTICATE;
378
379         } else {
380             goto invalid;
381         }
382         break;
383
384     default:
385         goto invalid;
386     }
387
388     switch (ch) {
389     case ' ':
390         state = sw_spaces_before_argument;
391         break;
392     case CR:
393         state = sw_almost_done;
394         break;
395     case LF:
396         goto done;
397     }
398     break;
399 }
400
401 if ((ch < 'A' || ch > 'Z') && (ch < 'a' || ch > 'z')) {
402     goto invalid;
403 }
404
405 break;
406
407 case sw_spaces_before_argument:
408     switch (ch) {
409     case ' ':
410         break;
411     case CR:
412         state = sw_almost_done;
413         s->arg_end = p;
414         break;
415     case LF:
416         s->arg_end = p;
417         goto done;
418     case '":
419         if (s->args.nelts <= 2) {
420             s->quoted = 1;
421             s->arg_start = p + 1;
422             state = sw_argument;
423             break;
424         }
425         goto invalid;
426     case '{':
427         if (s->args.nelts <= 2) {
428             state = sw_literal;
429             break;
430         }
431         goto invalid;
432     default:
433         if (s->args.nelts <= 2) {
434             s->arg_start = p;
435             state = sw_argument;
436             break;

```

```

437     }
438     goto invalid;
439 }
440 break;
441
442 case sw_argument:
443     if (ch == ' ' && s->quoted) {
444         break;
445     }
446
447     switch (ch) {
448     case '":
449         if (!s->quoted) {
450             break;
451         }
452         s->quoted = 0;
453         /* fall through */
454     case ' ':
455     case CR:
456     case LF:
457         arg = ngx_array_push(&s->args);
458         if (arg == NULL) {
459             return NGX_ERROR;
460         }
461         arg->len = p - s->arg_start;
462         arg->data = s->arg_start;
463         s->arg_start = NULL;
464
465         switch (ch) {
466         case '"':
467         case ' ':
468             state = sw_spaces_before_argument;
469             break;
470         case CR:
471             state = sw_almost_done;
472             break;
473         case LF:
474             goto done;
475         }
476         break;
477     case '\\':
478         if (s->quoted) {
479             s->backslash = 1;
480             state = sw_backslash;
481         }
482         break;
483     }
484     break;
485
486 case sw_backslash:
487     switch (ch) {
488     case CR:
489     case LF:
490         goto invalid;
491     default:
492         state = sw_argument;
493     }
494     break;
495
496 case sw_literal:
497     if (ch >= '0' && ch <= '9') {
498         s->literal_len = s->literal_len * 10 + (ch - '0');
499         break;
500     }
501     if (ch == '}') {
502         state = sw_start_literal_argument;
503         break;
504     }
505     if (ch == '+') {
506         state = sw_no_sync_literal_argument;
507         break;
508     }
509     goto invalid;
510
511 case sw_no_sync_literal_argument:
512     if (ch == '}') {

```

```

513         s->no_sync_literal = 1;
514         state = sw_start_literal_argument;
515         break;
516     }
517     goto invalid;
518
519     case sw_start_literal_argument:
520         switch (ch) {
521             case CR:
522                 break;
523             case LF:
524                 s->buffer->pos = p + 1;
525                 s->arg_start = p + 1;
526                 if (s->no_sync_literal == 0) {
527                     s->state = sw_literal_argument;
528                     return NGX_IMAP_NEXT;
529                 }
530                 state = sw_literal_argument;
531                 s->no_sync_literal = 0;
532                 break;
533             default:
534                 goto invalid;
535         }
536         break;
537
538     case sw_literal_argument:
539         if (s->literal_len && --s->literal_len) {
540             break;
541         }
542
543         arg = ngx_array_push(&s->args);
544         if (arg == NULL) {
545             return NGX_ERROR;
546         }
547         arg->len = p + 1 - s->arg_start;
548         arg->data = s->arg_start;
549         s->arg_start = NULL;
550         state = sw_end_literal_argument;
551
552         break;
553
554     case sw_end_literal_argument:
555         switch (ch) {
556             case '{':
557                 if (s->args.nelts <= 2) {
558                     state = sw_literal;
559                     break;
560                 }
561                 goto invalid;
562             case CR:
563                 state = sw_almost_done;
564                 break;
565             case LF:
566                 goto done;
567             default:
568                 state = sw_spaces_before_argument;
569                 break;
570         }
571         break;
572
573     case sw_almost_done:
574         switch (ch) {
575             case LF:
576                 goto done;
577             default:
578                 goto invalid;
579         }
580     }
581 }
582
583 s->buffer->pos = p;
584 s->state = state;
585
586 return NGX_AGAIN;
587
588 done:

```

```

589     s->buffer->pos = p + 1;
590
591     if (s->arg_start) {
592         arg = ngx_array_push(&s->args);
593         if (arg == NULL) {
594             return NGX_ERROR;
595         }
596         arg->len = s->arg_end - s->arg_start;
597         arg->data = s->arg_start;
598
599         s->arg_start = NULL;
600         s->cmd_start = NULL;
601         s->quoted = 0;
602         s->no_sync_literal = 0;
603         s->literal_len = 0;
604     }
605
606     s->state = (s->command != NGX_IMAP_AUTHENTICATE) ? sw_start : sw_argument;
607
608     return NGX_OK;
609
610 invalid:
611
612     s->state = sw_start;
613     s->quoted = 0;
614     s->no_sync_literal = 0;
615     s->literal_len = 0;
616
617     return NGX_MAIL_PARSE_INVALID_COMMAND;
618 }
619
620
621 ngx_int_t
622 ngx_mail_smtp_parse_command(ngx_mail_session_t *s)
623 {
624     u_char      ch, *p, *c, c0, c1, c2, c3;
625     ngx_str_t   *arg;
626     enum {
627         sw_start = 0,
628         sw_command,
629         sw_invalid,
630         sw_spaces_before_argument,
631         sw_argument,
632         sw_almost_done
633     } state;
634
635     state = s->state;
636
637     for (p = s->buffer->pos; p < s->buffer->last; p++) {
638         ch = *p;
639
640         switch (state) {
641
642             /* SMTP command */
643             case sw_start:
644                 s->cmd_start = p;
645                 state = sw_command;
646
647                 /* fall through */
648
649             case sw_command:
650                 if (ch == ' ' || ch == CR || ch == LF) {
651                     c = s->cmd_start;
652
653                     if (p - c == 4) {
654
655                         c0 = ngx_toupper(c[0]);
656                         c1 = ngx_toupper(c[1]);
657                         c2 = ngx_toupper(c[2]);
658                         c3 = ngx_toupper(c[3]);
659
660                         if (c0 == 'H' && c1 == 'E' && c2 == 'L' && c3 == 'O')
661                             {
662                                 s->command = NGX_SMTP_HELO;
663
664

```

```

665     } else if (c0 == 'E' && c1 == 'H' && c2 == 'L' && c3 == 'O')
666     {
667         s->command = NGX SMTP EHLO;
668
669     } else if (c0 == 'Q' && c1 == 'U' && c2 == 'I' && c3 == 'T')
670     {
671         s->command = NGX SMTP QUIT;
672
673     } else if (c0 == 'A' && c1 == 'U' && c2 == 'T' && c3 == 'H')
674     {
675         s->command = NGX SMTP AUTH;
676
677     } else if (c0 == 'N' && c1 == 'O' && c2 == 'O' && c3 == 'P')
678     {
679         s->command = NGX SMTP NOOP;
680
681     } else if (c0 == 'M' && c1 == 'A' && c2 == 'I' && c3 == 'L')
682     {
683         s->command = NGX SMTP MAIL;
684
685     } else if (c0 == 'R' && c1 == 'S' && c2 == 'E' && c3 == 'T')
686     {
687         s->command = NGX SMTP RSET;
688
689     } else if (c0 == 'R' && c1 == 'C' && c2 == 'P' && c3 == 'T')
690     {
691         s->command = NGX SMTP RCPT;
692
693     } else if (c0 == 'V' && c1 == 'R' && c2 == 'E' && c3 == 'Y')
694     {
695         s->command = NGX SMTP VRFY;
696
697     } else if (c0 == 'E' && c1 == 'X' && c2 == 'P' && c3 == 'N')
698     {
699         s->command = NGX SMTP EXPN;
700
701     } else if (c0 == 'H' && c1 == 'E' && c2 == 'L' && c3 == 'P')
702     {
703         s->command = NGX SMTP HELP;
704
705     } else {
706         goto invalid;
707     }
708 #if (NGX_MAIL_SSL)
709     } else if (p - c == 8) {
710
711         if ((c[0] == 'S' || c[0] == 's')
712             && (c[1] == 'T' || c[1] == 't')
713             && (c[2] == 'A' || c[2] == 'a')
714             && (c[3] == 'R' || c[3] == 'r')
715             && (c[4] == 'T' || c[4] == 't')
716             && (c[5] == 'T' || c[5] == 't')
717             && (c[6] == 'L' || c[6] == 'l')
718             && (c[7] == 'S' || c[7] == 's'))
719         {
720             s->command = NGX SMTP STARTTLS;
721
722         } else {
723             goto invalid;
724         }
725 #endif
726     } else {
727         goto invalid;
728     }
729
730     s->cmd.data = s->cmd_start;
731     s->cmd.len = p - s->cmd_start;
732
733     switch (ch) {
734     case ' ':
735         state = sw_spaces_before_argument;
736         break;
737     case CR:
738         state = sw_almost_done;
739         break;
740     case LF:

```



```

741         goto done;
742     }
743     break;
744 }
745
746 if ((ch < 'A' || ch > 'Z') && (ch < 'a' || ch > 'z')) {
747     goto invalid;
748 }
749
750 break;
751
752 case sw_invalid:
753     goto invalid;
754
755 case sw_spaces_before_argument:
756     switch (ch) {
757     case ' ':
758         break;
759     case CR:
760         state = sw_almost_done;
761         s->arg_end = p;
762         break;
763     case LF:
764         s->arg_end = p;
765         goto done;
766     default:
767         if (s->args.nelts <= 10) {
768             state = sw_argument;
769             s->arg_start = p;
770             break;
771         }
772         goto invalid;
773     }
774     break;
775
776 case sw_argument:
777     switch (ch) {
778     case ' ':
779     case CR:
780     case LF:
781         arg = ngx_array_push(&s->args);
782         if (arg == NULL) {
783             return NGX_ERROR;
784         }
785         arg->len = p - s->arg_start;
786         arg->data = s->arg_start;
787         s->arg_start = NULL;
788
789         switch (ch) {
790         case ' ':
791             state = sw_spaces_before_argument;
792             break;
793         case CR:
794             state = sw_almost_done;
795             break;
796         case LF:
797             goto done;
798         }
799         break;
800
801     default:
802         break;
803     }
804     break;
805
806 case sw_almost_done:
807     switch (ch) {
808     case LF:
809         goto done;
810     default:
811         goto invalid;
812     }
813 }
814 }
815
816 s->buffer->pos = p;

```

```

817     s->state = state;
818
819     return NGX\_AGAIN;
820
821 done:
822
823     s->buffer->pos = p + 1;
824
825     if (s->arg_start) {
826         arg = ngx\_array\_push(&s->args);
827         if (arg == NULL) {
828             return NGX\_ERROR;
829         }
830         arg->len = s->arg_end - s->arg_start;
831         arg->data = s->arg_start;
832         s->arg_start = NULL;
833     }
834
835     s->state = (s->command != NGX\_SMTP\_AUTH) ? sw_start : sw_argument;
836
837     return NGX\_OK;
838
839 invalid:
840
841     s->state = sw_invalid;
842     s->arg_start = NULL;
843
844     /* skip invalid command till LF */
845
846     for (p = s->buffer->pos; p < s->buffer->last; p++) {
847         if (*p == LF) {
848             s->state = sw_start;
849             p++;
850             break;
851         }
852     }
853
854     s->buffer->pos = p;
855
856     return NGX\_MAIL\_PARSE\_INVALID\_COMMAND;
857 }
858
859
860 ngx\_int\_t
861 ngx\_mail\_auth\_parse(ngx\_mail\_session\_t *s, ngx\_connection\_t *c)
862 {
863     ngx\_str\_t          *arg;
864
865     #if (NGX\_MAIL\_SSL)
866     if (ngx\_mail\_starttls\_only(s, c)) {
867         return NGX\_MAIL\_PARSE\_INVALID\_COMMAND;
868     }
869     #endif
870
871     if (s->args.nelts == 0) {
872         return NGX\_MAIL\_PARSE\_INVALID\_COMMAND;
873     }
874
875     arg = s->args.elts;
876
877     if (arg[0].len == 5) {
878
879         if (ngx\_strncasecmp(arg[0].data, (u_char *) "LOGIN", 5) == 0) {
880
881             if (s->args.nelts == 1) {
882                 return NGX\_MAIL\_AUTH\_LOGIN;
883             }
884
885             if (s->args.nelts == 2) {
886                 return NGX\_MAIL\_AUTH\_LOGIN\_USERNAME;
887             }
888
889             return NGX\_MAIL\_PARSE\_INVALID\_COMMAND;
890         }
891
892         if (ngx\_strncasecmp(arg[0].data, (u_char *) "PLAIN", 5) == 0) {

```

```
893         if (s->args.nelts == 1) {
894             return NGX\_MAIL\_AUTH\_PLAIN;
895         }
896
897         if (s->args.nelts == 2) {
898             return ngx\_mail\_auth\_plain(s, c, 1);
899         }
900     }
901
902     return NGX\_MAIL\_PARSE\_INVALID\_COMMAND;
903 }
904
905 if (arg[0].len == 8) {
906     if (s->args.nelts != 1) {
907         return NGX\_MAIL\_PARSE\_INVALID\_COMMAND;
908     }
909
910     if (ngx\_strncasecmp(arg[0].data, (u_char *) "CRAM-MD5", 8) == 0) {
911         return NGX\_MAIL\_AUTH\_CRAM\_MD5;
912     }
913 }
914
915 return NGX\_MAIL\_PARSE\_INVALID\_COMMAND;
916 }
917 }
918 }
```

[One Level Up](#)

[Top Level](#)

# src/mail/nginx\_mail\_pop3\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_mail\\_pop3\\_auth\\_cram\\_md5\\_capability](#)
- [ngx\\_mail\\_pop3\\_auth\\_methods](#)
- [ngx\\_mail\\_pop3\\_auth\\_plain\\_capability](#)
- [ngx\\_mail\\_pop3\\_commands](#)
- [ngx\\_mail\\_pop3\\_default\\_capabilities](#)
- [ngx\\_mail\\_pop3\\_module](#)
- [ngx\\_mail\\_pop3\\_module\\_ctx](#)
- [ngx\\_mail\\_pop3\\_protocol](#)

## Functions defined

- [ngx\\_mail\\_pop3\\_create\\_srv\\_conf](#)
- [ngx\\_mail\\_pop3\\_merge\\_srv\\_conf](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11 #include <ngx_mail.h>
12 #include <ngx_mail_pop3_module.h>
13
14
15 static void *ngx_mail_pop3_create_srv_conf(ngx_conf_t *cf);
16 static char *ngx_mail_pop3_merge_srv_conf(ngx_conf_t *cf, void *parent,
17     void *child);
18
19
20 static ngx_str_t ngx_mail_pop3_default_capabilities[] = {
21     ngx_string("TOP"),
22     ngx_string("USER"),
23     ngx_string("UIDL"),
24     ngx_null_string
25 };
26
27
28 static ngx_conf_bitmask_t ngx_mail_pop3_auth_methods[] = {
29     { ngx_string("plain"), NGX_MAIL_AUTH_PLAIN_ENABLED },
30     { ngx_string("apop"), NGX_MAIL_AUTH_APOP_ENABLED },
31     { ngx_string("cram-md5"), NGX_MAIL_AUTH_CRAM_MD5_ENABLED },
32     { ngx_null_string, 0 }
33 };
34
35
36 static ngx_str_t ngx_mail_pop3_auth_plain_capability =
37     ngx_string("+OK methods supported:" CRLF
38     "LOGIN" CRLF
```

```

39         "PLAIN" CRLF
40         "." CRLF);
41
42
43 static ngx_str_t  ngx_mail_pop3_auth_cram_md5_capability =
44     ngx_string("+OK methods supported:" CRLF
45         "LOGIN" CRLF
46         "PLAIN" CRLF
47         "CRAM-MD5" CRLF
48         "." CRLF);
49
50
51 static ngx_mail_protocol_t  ngx_mail_pop3_protocol = {
52     ngx_string("pop3"),
53     { 110, 995, 0, 0 },
54     NGX_MAIL_POP3_PROTOCOL,
55
56     ngx_mail_pop3_init_session,
57     ngx_mail_pop3_init_protocol,
58     ngx_mail_pop3_parse_command,
59     ngx_mail_pop3_auth_state,
60
61     ngx_string("-ERR internal server error" CRLF)
62 };
63
64
65 static ngx_command_t  ngx_mail_pop3_commands[] = {
66
67     { ngx_string("pop3_capabilities"),
68       NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_1MORE,
69       ngx_mail_capabilities,
70       NGX_MAIL_SRV_CONF_OFFSET,
71       offsetof(ngx_mail_pop3_srv_conf_t, capabilities),
72       NULL },
73
74     { ngx_string("pop3_auth"),
75       NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_1MORE,
76       ngx_conf_set_bitmask_slot,
77       NGX_MAIL_SRV_CONF_OFFSET,
78       offsetof(ngx_mail_pop3_srv_conf_t, auth_methods),
79       &ngx_mail_pop3_auth_methods },
80
81     ngx_null_command
82 };
83
84
85 static ngx_mail_module_t  ngx_mail_pop3_module_ctx = {
86     &ngx_mail_pop3_protocol,          /* protocol */
87
88     NULL,                             /* create main configuration */
89     NULL,                             /* init main configuration */
90
91     ngx_mail_pop3_create_srv_conf,     /* create server configuration */
92     ngx_mail_pop3_merge_srv_conf      /* merge server configuration */
93 };
94
95
96 ngx_module_t  ngx_mail_pop3_module = {
97     NGX_MODULE_V1,
98     &ngx_mail_pop3_module_ctx,        /* module context */
99     ngx_mail_pop3_commands,           /* module directives */
100    NGX_MAIL_MODULE,                   /* module type */
101    NULL,                              /* init master */
102    NULL,                              /* init module */
103    NULL,                              /* init process */
104    NULL,                              /* init thread */
105    NULL,                              /* exit thread */
106    NULL,                              /* exit process */
107    NULL,                              /* exit master */
108    NGX_MODULE_V1_PADDING
109 };
110
111
112 static void *
113 ngx_mail_pop3_create_srv_conf(ngx_conf_t *cf)
114 {

```

```

115     ngx_mail_pop3_srv_conf_t *pscf;
116
117     pscf = ngx_pcalloc(cf->pool, sizeof(ngx_mail_pop3_srv_conf_t));
118     if (pscf == NULL) {
119         return NULL;
120     }
121
122     if (ngx_array_init(&pscf->capabilities, cf->pool, 4, sizeof(ngx_str_t))
123         != NGX_OK)
124     {
125         return NULL;
126     }
127
128     return pscf;
129 }
130
131
132 static char *
133 ngx_mail_pop3_merge_srv_conf(ngx_conf_t *cf, void *parent, void *child)
134 {
135     ngx_mail_pop3_srv_conf_t *prev = parent;
136     ngx_mail_pop3_srv_conf_t *conf = child;
137
138     u_char      *p;
139     size_t      size, stls_only_size;
140     ngx_str_t   *c, *d;
141     ngx_uint_t  i;
142
143     ngx_conf_merge_bitmask_value(conf->auth_methods,
144                                 prev->auth_methods,
145                                 (NGX_CONF_BITMASK_SET
146                                  |NGX_MAIL_AUTH_PLAIN_ENABLED));
147
148     if (conf->capabilities.nelts == 0) {
149         conf->capabilities = prev->capabilities;
150     }
151
152     if (conf->capabilities.nelts == 0) {
153
154         for (d = ngx_mail_pop3_default_capabilities; d->len; d++) {
155             c = ngx_array_push(&conf->capabilities);
156             if (c == NULL) {
157                 return NGX_CONF_ERROR;
158             }
159
160             *c = *d;
161         }
162     }
163
164     size = sizeof("+OK Capability list follows" CRLF) - 1
165           + sizeof("." CRLF) - 1;
166
167     stls_only_size = size + sizeof("STLS" CRLF) - 1;
168
169     c = conf->capabilities.elts;
170     for (i = 0; i < conf->capabilities.nelts; i++) {
171         size += c[i].len + sizeof(CRLF) - 1;
172
173         if (ngx_strcasecmp(c[i].data, (u_char *) "USER") == 0) {
174             continue;
175         }
176
177         stls_only_size += c[i].len + sizeof(CRLF) - 1;
178     }
179
180     if (conf->auth_methods & NGX_MAIL_AUTH_CRAM_MD5_ENABLED) {
181         size += sizeof("SASL LOGIN PLAIN CRAM-MD5" CRLF) - 1;
182
183     } else {
184         size += sizeof("SASL LOGIN PLAIN" CRLF) - 1;
185     }
186
187     p = ngx_pnalloc(cf->pool, size);
188     if (p == NULL) {
189         return NGX_CONF_ERROR;
190     }

```

```

191 conf->capability.len = size;
192 conf->capability.data = p;
193
194
195 p = ngx_cpymem(p, "+OK Capability list follows" CRLF,
196             sizeof("+OK Capability list follows" CRLF) - 1);
197
198 for (i = 0; i < conf->capabilities.nelts; i++) {
199     p = ngx_cpymem(p, c[i].data, c[i].len);
200     *p++ = CR; *p++ = LF;
201 }
202
203 if (conf->auth_methods & NGX_MAIL_AUTH_CRAM_MD5_ENABLED) {
204     p = ngx_cpymem(p, "SASL LOGIN PLAIN CRAM-MD5" CRLF,
205                 sizeof("SASL LOGIN PLAIN CRAM-MD5" CRLF) - 1);
206
207 } else {
208     p = ngx_cpymem(p, "SASL LOGIN PLAIN" CRLF,
209                 sizeof("SASL LOGIN PLAIN" CRLF) - 1);
210 }
211
212 *p++ = '.'; *p++ = CR; *p = LF;
213
214
215 size += sizeof("STLS" CRLF) - 1;
216
217 p = ngx_pnalloc(cf->pool, size);
218 if (p == NULL) {
219     return NGX_CONF_ERROR;
220 }
221
222 conf->starttls_capability.len = size;
223 conf->starttls_capability.data = p;
224
225 p = ngx_cpymem(p, conf->capability.data,
226             conf->capability.len - (sizeof("." CRLF) - 1));
227
228 p = ngx_cpymem(p, "STLS" CRLF, sizeof("STLS" CRLF) - 1);
229 *p++ = '.'; *p++ = CR; *p = LF;
230
231
232 if (conf->auth_methods & NGX_MAIL_AUTH_CRAM_MD5_ENABLED) {
233     conf->auth_capability = ngx_mail_pop3_auth_cram_md5_capability;
234
235 } else {
236     conf->auth_capability = ngx_mail_pop3_auth_plain_capability;
237 }
238
239
240 p = ngx_pnalloc(cf->pool, stls_only_size);
241 if (p == NULL) {
242     return NGX_CONF_ERROR;
243 }
244
245 conf->starttls_only_capability.len = stls_only_size;
246 conf->starttls_only_capability.data = p;
247
248 p = ngx_cpymem(p, "+OK Capability list follows" CRLF,
249             sizeof("+OK Capability list follows" CRLF) - 1);
250
251 for (i = 0; i < conf->capabilities.nelts; i++) {
252     if (ngx_strcasecmp(c[i].data, (u_char *) "USER") == 0) {
253         continue;
254     }
255
256     p = ngx_cpymem(p, c[i].data, c[i].len);
257     *p++ = CR; *p++ = LF;
258 }
259
260 p = ngx_cpymem(p, "STLS" CRLF, sizeof("STLS" CRLF) - 1);
261 *p++ = '.'; *p++ = CR; *p = LF;
262
263 return NGX_CONF_OK;
264 }

```

# src/mail/nginx\_mail\_pop3\_handler.c - nginx-1.7.10

## Global variables defined

- [pop3\\_greeting](#)
- [pop3\\_invalid\\_command](#)
- [pop3\\_next](#)
- [pop3\\_ok](#)
- [pop3\\_password](#)
- [pop3\\_username](#)

## Functions defined

- [ngx\\_mail\\_pop3\\_apop](#)
- [ngx\\_mail\\_pop3\\_auth](#)
- [ngx\\_mail\\_pop3\\_auth\\_state](#)
- [ngx\\_mail\\_pop3\\_capa](#)
- [ngx\\_mail\\_pop3\\_init\\_protocol](#)
- [ngx\\_mail\\_pop3\\_init\\_session](#)
- [ngx\\_mail\\_pop3\\_pass](#)
- [ngx\\_mail\\_pop3\\_stls](#)
- [ngx\\_mail\\_pop3\\_user](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11 #include <ngx_mail.h>
12 #include <ngx_mail_pop3_module.h>
13
14
15 static ngx_int_t ngx_mail_pop3_user(ngx_mail_session_t *s, ngx_connection_t *c);
16 static ngx_int_t ngx_mail_pop3_pass(ngx_mail_session_t *s, ngx_connection_t *c);
17 static ngx_int_t ngx_mail_pop3_capa(ngx_mail_session_t *s, ngx_connection_t *c,
18     ngx_int_t stls);
19 static ngx_int_t ngx_mail_pop3_stls(ngx_mail_session_t *s, ngx_connection_t *c);
20 static ngx_int_t ngx_mail_pop3_apop(ngx_mail_session_t *s, ngx_connection_t *c);
21 static ngx_int_t ngx_mail_pop3_auth(ngx_mail_session_t *s, ngx_connection_t *c);
22
23
24 static u_char pop3_greeting[] = "+OK POP3 ready" CRLF;
25 static u_char pop3_ok[] = "+OK" CRLF;
26 static u_char pop3_next[] = "+ " CRLF;
27 static u_char pop3_username[] = "+ VXN1cm5hbWU6" CRLF;
```



```

28 static u_char pop3_password[] = "+ UGFzc3dvcmQ6" CRLF;
29 static u_char pop3_invalid_command[] = "-ERR invalid command" CRLF;
30
31
32 void
33 ngx_mail_pop3_init_session(ngx_mail_session_t *s, ngx_connection_t *c)
34 {
35     u_char *p;
36     ngx_mail_core_srv_conf_t *cscf;
37     ngx_mail_pop3_srv_conf_t *pscfc;
38
39     pscfc = ngx_mail_get_module_srv_conf(s, ngx_mail_pop3_module);
40     cscf = ngx_mail_get_module_srv_conf(s, ngx_mail_core_module);
41
42     if (pscfc->auth_methods
43         & (NGX_MAIL_AUTH_APOP_ENABLED|NGX_MAIL_AUTH_CRAM_MD5_ENABLED))
44     {
45         if (ngx_mail_salt(s, c, cscf) != NGX_OK) {
46             ngx_mail_session_internal_server_error(s);
47             return;
48         }
49
50         s->out.data = ngx_pnalloc(c->pool, sizeof(pop3_greeting) + s->salt.len);
51         if (s->out.data == NULL) {
52             ngx_mail_session_internal_server_error(s);
53             return;
54         }
55
56         p = ngx_cpymem(s->out.data, pop3_greeting, sizeof(pop3_greeting) - 3);
57         *p++ = ' ';
58         p = ngx_cpymem(p, s->salt.data, s->salt.len);
59
60         s->out.len = p - s->out.data;
61
62     } else {
63         ngx_str_set(&s->out, pop3_greeting);
64     }
65
66     c->read->handler = ngx_mail_pop3_init_protocol;
67
68     ngx_add_timer(c->read, cscf->timeout);
69
70     if (ngx_handle_read_event(c->read, 0) != NGX_OK) {
71         ngx_mail_close_connection(c);
72     }
73
74     ngx_mail_send(c->write);
75 }
76
77
78 void
79 ngx_mail_pop3_init_protocol(ngx_event_t *rev)
80 {
81     ngx_connection_t *c;
82     ngx_mail_session_t *s;
83
84     c = rev->data;
85
86     c->log->action = "in auth state";
87
88     if (rev->timedout) {
89         ngx_log_error(NGX_LOG_INFO, c->log, NGX_ETIMEDOUT, "client timed out");
90         c->timedout = 1;
91         ngx_mail_close_connection(c);
92         return;
93     }
94
95     s = c->data;
96
97     if (s->buffer == NULL) {
98         if (ngx_array_init(&s->args, c->pool, 2, sizeof(ngx_str_t))
99             == NGX_ERROR)
100         {
101             ngx_mail_session_internal_server_error(s);
102             return;
103         }

```

```

104     s->buffer = ngx_create_temp_buf(c->pool, 128);
105     if (s->buffer == NULL) {
106         ngx_mail_session_internal_server_error(s);
107         return;
108     }
109 }
110
111
112 s->mail_state = ngx_pop3_start;
113 c->read->handler = ngx_mail_pop3_auth_state;
114
115 ngx_mail_pop3_auth_state(rev);
116 }
117
118
119 void
120 ngx_mail_pop3_auth_state(ngx_event_t *rev)
121 {
122     ngx_int_t      rc;
123     ngx_connection_t *c;
124     ngx_mail_session_t *s;
125
126     c = rev->data;
127     s = c->data;
128
129     ngx_log_debug0(NGX_LOG_DEBUG_MAIL, c->log, 0, "pop3 auth state");
130
131     if (rev->timedout) {
132         ngx_log_error(NGX_LOG_INFO, c->log, NGX_ETIMEDOUT, "client timed out");
133         c->timedout = 1;
134         ngx_mail_close_connection(c);
135         return;
136     }
137
138     if (s->out.len) {
139         ngx_log_debug0(NGX_LOG_DEBUG_MAIL, c->log, 0, "pop3 send handler busy");
140         s->blocked = 1;
141         return;
142     }
143
144     s->blocked = 0;
145
146     rc = ngx_mail_read_command(s, c);
147
148     if (rc == NGX_AGAIN || rc == NGX_ERROR) {
149         return;
150     }
151
152     ngx_str_set(&s->out, pop3_ok);
153
154     if (rc == NGX_OK) {
155         switch (s->mail_state) {
156
157             case ngx_pop3_start:
158
159                 switch (s->command) {
160
161                     case NGX_POP3_USER:
162                         rc = ngx_mail_pop3_user(s, c);
163                         break;
164
165                     case NGX_POP3_CAPA:
166                         rc = ngx_mail_pop3_capa(s, c, 1);
167                         break;
168
169                     case NGX_POP3_APOP:
170                         rc = ngx_mail_pop3_apop(s, c);
171                         break;
172
173                     case NGX_POP3_AUTH:
174                         rc = ngx_mail_pop3_auth(s, c);
175                         break;
176
177                     case NGX_POP3_QUIT:
178                         s->quit = 1;
179                         break;

```

```

180
181     case NGX\_POP3\_NOOP:
182         break;
183
184     case NGX\_POP3\_STLS:
185         rc = ngx\_mail\_pop3\_stls(s, c);
186         break;
187
188     default:
189         rc = NGX\_MAIL\_PARSE\_INVALID\_COMMAND;
190         break;
191 }
192
193     break;
194
195 case ngx_pop3_user:
196
197     switch (s->command) {
198
199     case NGX\_POP3\_PASS:
200         rc = ngx\_mail\_pop3\_pass(s, c);
201         break;
202
203     case NGX\_POP3\_CAPA:
204         rc = ngx\_mail\_pop3\_capa(s, c, 0);
205         break;
206
207     case NGX\_POP3\_QUIT:
208         s->quit = 1;
209         break;
210
211     case NGX\_POP3\_NOOP:
212         break;
213
214     default:
215         rc = NGX\_MAIL\_PARSE\_INVALID\_COMMAND;
216         break;
217 }
218
219     break;
220
221 /* suppress warnings */
222 case ngx_pop3_passwd:
223     break;
224
225 case ngx_pop3_auth_login_username:
226     rc = ngx\_mail\_auth\_login\_username(s, c, 0);
227
228     ngx\_str\_set(&s->out, pop3\_password);
229     s->mail_state = ngx_pop3_auth_login_password;
230     break;
231
232 case ngx_pop3_auth_login_password:
233     rc = ngx\_mail\_auth\_login\_password(s, c);
234     break;
235
236 case ngx_pop3_auth_plain:
237     rc = ngx\_mail\_auth\_plain(s, c, 0);
238     break;
239
240 case ngx_pop3_auth_cram_md5:
241     rc = ngx\_mail\_auth\_cram\_md5(s, c);
242     break;
243 }
244 }
245
246 switch (rc) {
247
248 case NGX\_DONE:
249     ngx\_mail\_auth(s, c);
250     return;
251
252 case NGX\_ERROR:
253     ngx\_mail\_session\_internal\_server\_error(s);
254     return;
255

```

```

256 case NGX_MAIL_PARSE_INVALID_COMMAND:
257     s->mail_state = ngx_pop3_start;
258     s->state = 0;
259
260     ngx_str_set(&s->out, pop3_invalid_command);
261
262     /* fall through */
263
264 case NGX_OK:
265
266     s->args.nelts = 0;
267     s->buffer->pos = s->buffer->start;
268     s->buffer->last = s->buffer->start;
269
270     if (s->state) {
271         s->arg_start = s->buffer->start;
272     }
273
274     ngx_mail_send(c->write);
275 }
276 }
277
278 static ngx_int_t
279 ngx_mail_pop3_user(ngx_mail_session_t *s, ngx_connection_t *c)
280 {
281     ngx_str_t *arg;
282
283     #if (NGX_MAIL_SSL)
284     if (ngx_mail_starttls_only(s, c)) {
285         return NGX_MAIL_PARSE_INVALID_COMMAND;
286     }
287     #endif
288
289     if (s->args.nelts != 1) {
290         return NGX_MAIL_PARSE_INVALID_COMMAND;
291     }
292
293     arg = s->args.elts;
294     s->login.len = arg[0].len;
295     s->login.data = ngx_pnalloc(c->pool, s->login.len);
296     if (s->login.data == NULL) {
297         return NGX_ERROR;
298     }
299
300     ngx_memcpy(s->login.data, arg[0].data, s->login.len);
301
302     ngx_log_debug1(NGX_LOG_DEBUG_MAIL, c->log, 0,
303         "pop3 login: \"%V\"", &s->login);
304
305     s->mail_state = ngx_pop3_user;
306
307     return NGX_OK;
308 }
309
310
311 static ngx_int_t
312 ngx_mail_pop3_pass(ngx_mail_session_t *s, ngx_connection_t *c)
313 {
314     ngx_str_t *arg;
315
316     if (s->args.nelts != 1) {
317         return NGX_MAIL_PARSE_INVALID_COMMAND;
318     }
319
320     arg = s->args.elts;
321     s->passwd.len = arg[0].len;
322     s->passwd.data = ngx_pnalloc(c->pool, s->passwd.len);
323     if (s->passwd.data == NULL) {
324         return NGX_ERROR;
325     }
326
327     ngx_memcpy(s->passwd.data, arg[0].data, s->passwd.len);
328
329     #if (NGX_DEBUG_MAIL_PASSWD)
330     ngx_log_debug1(NGX_LOG_DEBUG_MAIL, c->log, 0,
331         "pop3 passwd: \"%V\"", &s->passwd);

```

```

332 #endif
333
334     return NGX_DONE;
335 }
336
337
338 static ngx_int_t
339 ngx_mail_pop3_capa(ngx_mail_session_t *s, ngx_connection_t *c, ngx_int_t stls)
340 {
341     ngx_mail_pop3_srv_conf_t *pscf;
342
343     pscf = ngx_mail_get_module_srv_conf(s, ngx_mail_pop3_module);
344
345     #if (NGX_MAIL_SSL)
346
347     if (stls && c->ssl == NULL) {
348         ngx_mail_ssl_conf_t *sslcf;
349
350         sslcf = ngx_mail_get_module_srv_conf(s, ngx_mail_ssl_module);
351
352         if (sslcf->starttls == NGX_MAIL_STARTTLS_ON) {
353             s->out = pscf->starttls_capability;
354             return NGX_OK;
355         }
356
357         if (sslcf->starttls == NGX_MAIL_STARTTLS_ONLY) {
358             s->out = pscf->starttls_only_capability;
359             return NGX_OK;
360         }
361     }
362
363     #endif
364
365     s->out = pscf->capability;
366     return NGX_OK;
367 }
368
369
370 static ngx_int_t
371 ngx_mail_pop3_stls(ngx_mail_session_t *s, ngx_connection_t *c)
372 {
373     #if (NGX_MAIL_SSL)
374         ngx_mail_ssl_conf_t *sslcf;
375
376         if (c->ssl == NULL) {
377             sslcf = ngx_mail_get_module_srv_conf(s, ngx_mail_ssl_module);
378             if (sslcf->starttls) {
379                 c->read->handler = ngx_mail_starttls_handler;
380                 return NGX_OK;
381             }
382         }
383
384     #endif
385
386     return NGX_MAIL_PARSE_INVALID_COMMAND;
387 }
388
389
390 static ngx_int_t
391 ngx_mail_pop3_apop(ngx_mail_session_t *s, ngx_connection_t *c)
392 {
393     ngx_str_t *arg;
394     ngx_mail_pop3_srv_conf_t *pscf;
395
396     #if (NGX_MAIL_SSL)
397     if (ngx_mail_starttls_only(s, c)) {
398         return NGX_MAIL_PARSE_INVALID_COMMAND;
399     }
400     #endif
401
402     if (s->args.nelts != 2) {
403         return NGX_MAIL_PARSE_INVALID_COMMAND;
404     }
405
406     pscf = ngx_mail_get_module_srv_conf(s, ngx_mail_pop3_module);
407

```

```

408     if (!(pscf->auth_methods & NGX_MAIL_AUTH_APOP_ENABLED)) {
409         return NGX_MAIL_PARSE_INVALID_COMMAND;
410     }
411
412     arg = s->args.elts;
413
414     s->login.len = arg[0].len;
415     s->login.data = ngx_pnalloc(c->pool, s->login.len);
416     if (s->login.data == NULL) {
417         return NGX_ERROR;
418     }
419
420     ngx_memcpy(s->login.data, arg[0].data, s->login.len);
421
422     s->passwd.len = arg[1].len;
423     s->passwd.data = ngx_pnalloc(c->pool, s->passwd.len);
424     if (s->passwd.data == NULL) {
425         return NGX_ERROR;
426     }
427
428     ngx_memcpy(s->passwd.data, arg[1].data, s->passwd.len);
429
430     ngx_log_debug2(NGX_LOG_DEBUG_MAIL, c->log, 0,
431                  "pop3 apop: \"%V\" \"%V\"", &s->login, &s->passwd);
432
433     s->auth_method = NGX_MAIL_AUTH_APOP;
434
435     return NGX_DONE;
436 }
437
438
439 static ngx_int_t
440 ngx_mail_pop3_auth(ngx_mail_session_t *s, ngx_connection_t *c)
441 {
442     ngx_int_t          rc;
443     ngx_mail_pop3_srv_conf_t *pscf;
444
445     #if (NGX_MAIL_SSL)
446     if (ngx_mail_starttls_only(s, c)) {
447         return NGX_MAIL_PARSE_INVALID_COMMAND;
448     }
449     #endif
450
451     pscf = ngx_mail_get_module_srv_conf(s, ngx_mail_pop3_module);
452
453     if (s->args.nelts == 0) {
454         s->out = pscf->auth_capability;
455         s->state = 0;
456
457         return NGX_OK;
458     }
459
460     rc = ngx_mail_auth_parse(s, c);
461
462     switch (rc) {
463
464     case NGX_MAIL_AUTH_LOGIN:
465
466         ngx_str_set(&s->out, pop3_username);
467         s->mail_state = ngx_pop3_auth_login_username;
468
469         return NGX_OK;
470
471     case NGX_MAIL_AUTH_LOGIN_USERNAME:
472
473         ngx_str_set(&s->out, pop3_password);
474         s->mail_state = ngx_pop3_auth_login_password;
475
476         return ngx_mail_auth_login_username(s, c, 1);
477
478     case NGX_MAIL_AUTH_PLAIN:
479
480         ngx_str_set(&s->out, pop3_next);
481         s->mail_state = ngx_pop3_auth_plain;
482
483         return NGX_OK;

```

```
484
485 case NGX\_MAIL\_AUTH\_CRAM\_MD5:
486
487     if (!(pscf->auth_methods & NGX\_MAIL\_AUTH\_CRAM\_MD5\_ENABLED)) {
488         return NGX\_MAIL\_PARSE\_INVALID\_COMMAND;
489     }
490
491     if (ngx\_mail\_auth\_cram\_md5\_salt(s, c, "+ ", 2) == NGX\_OK) {
492         s->mail_state = ngx_pop3_auth_cram_md5;
493         return NGX\_OK;
494     }
495
496     return NGX\_ERROR;
497 }
498
499 return rc;
500 }
```

[One Level Up](#)

[Top Level](#)

# src/mail/nginx\_mail\_pop3\_module.h - nginx-1.7.10

## Data types defined

- [ngx\\_mail\\_pop3\\_srv\\_conf\\_t](#)

## Macros defined

- [\\_NGX\\_MAIL\\_POP3\\_MODULE\\_H\\_INCLUDED](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_MAIL\_POP3\_MODULE\_H\_INCLUDED
9 #define \_NGX\_MAIL\_POP3\_MODULE\_H\_INCLUDED
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14 #include <ngx_mail.h>
15
16
17 typedef struct {
18     ngx\_str\_t    capability;
19     ngx\_str\_t    starttls_capability;
20     ngx\_str\_t    starttls_only_capability;
21     ngx\_str\_t    auth_capability;
22
23     ngx\_uint\_t   auth_methods;
24
25     ngx\_array\_t capabilities;
26 } ngx\_mail\_pop3\_srv\_conf\_t;
27
28
29 void ngx\_mail\_pop3\_init\_session(ngx\_mail\_session\_t *s, ngx\_connection\_t *c);
30 void ngx\_mail\_pop3\_init\_protocol(ngx\_event\_t *rev);
31 void ngx\_mail\_pop3\_auth\_state(ngx\_event\_t *rev);
32 ngx\_int\_t ngx\_mail\_pop3\_parse\_command(ngx\_mail\_session\_t *s);
33
34
35 extern ngx\_module\_t ngx\_mail\_pop3\_module;
36
37
38 #endif /* \_NGX\_MAIL\_POP3\_MODULE\_H\_INCLUDED */
```



# src/mail/nginx\_mail\_imap\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_mail\\_imap\\_auth\\_methods](#)
- [ngx\\_mail\\_imap\\_auth\\_methods\\_names](#)
- [ngx\\_mail\\_imap\\_commands](#)
- [ngx\\_mail\\_imap\\_default\\_capabilities](#)
- [ngx\\_mail\\_imap\\_module](#)
- [ngx\\_mail\\_imap\\_module\\_ctx](#)
- [ngx\\_mail\\_imap\\_protocol](#)

## Functions defined

- [ngx\\_mail\\_imap\\_create\\_srv\\_conf](#)
- [ngx\\_mail\\_imap\\_merge\\_srv\\_conf](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11 #include <ngx_mail.h>
12 #include <ngx_mail_imap_module.h>
13
14
15 static void *ngx_mail_imap_create_srv_conf(ngx_conf_t *cf);
16 static char *ngx_mail_imap_merge_srv_conf(ngx_conf_t *cf, void *parent,
17     void *child);
18
19
20 static ngx_str_t ngx_mail_imap_default_capabilities[] = {
21     ngx_string("IMAP4"),
22     ngx_string("IMAP4rev1"),
23     ngx_string("UIDPLUS"),
24     ngx_null_string
25 };
26
27
28 static ngx_conf_bitmask_t ngx_mail_imap_auth_methods[] = {
29     { ngx_string("plain"), NGX_MAIL_AUTH_PLAIN_ENABLED },
30     { ngx_string("login"), NGX_MAIL_AUTH_LOGIN_ENABLED },
31     { ngx_string("cram-md5"), NGX_MAIL_AUTH_CRAM_MD5_ENABLED },
32     { ngx_null_string, 0 }
33 };
34
35
36 static ngx_str_t ngx_mail_imap_auth_methods_names[] = {
37     ngx_string("AUTH=PLAIN"),
38     ngx_string("AUTH=LOGIN"),
39     ngx_null_string, /* APOP */
40     ngx_string("AUTH=CRAM-MD5"),
41     ngx_null_string /* NONE */

```

```

42 };
43
44
45 static ngx_mail_protocol_t ngx_mail_imap_protocol = {
46     ngx_string("imap"),
47     { 143, 993, 0, 0 },
48     NGX_MAIL_IMAP_PROTOCOL,
49
50     ngx_mail_imap_init_session,
51     ngx_mail_imap_init_protocol,
52     ngx_mail_imap_parse_command,
53     ngx_mail_imap_auth_state,
54
55     ngx_string("* BAD internal server error" CRLF)
56 };
57
58
59 static ngx_command_t ngx_mail_imap_commands[] = {
60
61     { ngx_string("imap_client_buffer"),
62       NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_TAKE1,
63       ngx_conf_set_size_slot,
64       NGX_MAIL_SRV_CONF_OFFSET,
65       offsetof(ngx_mail_imap_srv_conf_t, client_buffer_size),
66       NULL },
67
68     { ngx_string("imap_capabilities"),
69       NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_1MORE,
70       ngx_mail_capabilities,
71       NGX_MAIL_SRV_CONF_OFFSET,
72       offsetof(ngx_mail_imap_srv_conf_t, capabilities),
73       NULL },
74
75     { ngx_string("imap_auth"),
76       NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_1MORE,
77       ngx_conf_set_bitmask_slot,
78       NGX_MAIL_SRV_CONF_OFFSET,
79       offsetof(ngx_mail_imap_srv_conf_t, auth_methods),
80       &ngx_mail_imap_auth_methods },
81
82     ngx_null_command
83 };
84
85
86 static ngx_mail_module_t ngx_mail_imap_module_ctx = {
87     &ngx_mail_imap_protocol,          /* protocol */
88
89     NULL,                             /* create main configuration */
90     NULL,                             /* init main configuration */
91
92     ngx_mail_imap_create_srv_conf,    /* create server configuration */
93     ngx_mail_imap_merge_srv_conf     /* merge server configuration */
94 };
95
96
97 ngx_module_t ngx_mail_imap_module = {
98     NGX_MODULE_V1,
99     &ngx_mail_imap_module_ctx,        /* module context */
100    ngx_mail_imap_commands,           /* module directives */
101    NGX_MAIL_MODULE,                  /* module type */
102    NULL,                             /* init master */
103    NULL,                             /* init module */
104    NULL,                             /* init process */
105    NULL,                             /* init thread */
106    NULL,                             /* exit thread */
107    NULL,                             /* exit process */
108    NULL,                             /* exit master */
109    NGX_MODULE_V1_PADDING
110 };
111
112
113 static void *
114 ngx_mail_imap_create_srv_conf(ngx_conf_t *cf)
115 {
116     ngx_mail_imap_srv_conf_t *iscf;
117

```

```

118     iscf = ngx_palloc(cf->pool, sizeof(ngx_mail_imap_srv_conf_t));
119     if (iscf == NULL) {
120         return NULL;
121     }
122
123     iscf->client_buffer_size = NGX_CONF_UNSET_SIZE;
124
125     if (ngx_array_init(&iscf->capabilities, cf->pool, 4, sizeof(ngx_str_t))
126         != NGX_OK)
127     {
128         return NULL;
129     }
130
131     return iscf;
132 }
133
134
135 static char *
136 ngx_mail_imap_merge_srv_conf(ngx_conf_t *cf, void *parent, void *child)
137 {
138     ngx_mail_imap_srv_conf_t *prev = parent;
139     ngx_mail_imap_srv_conf_t *conf = child;
140
141     u_char      *p, *auth;
142     size_t      size;
143     ngx_str_t   *c, *d;
144     ngx_uint_t  i, m;
145
146     ngx_conf_merge_size_value(conf->client_buffer_size,
147                               prev->client_buffer_size,
148                               (size_t) ngx_pagesize);
149
150     ngx_conf_merge_bitmask_value(conf->auth_methods,
151                                  prev->auth_methods,
152                                  (NGX_CONF_BITMASK_SET
153                                   |NGX_MAIL_AUTH_PLAIN_ENABLED));
154
155
156     if (conf->capabilities.nelts == 0) {
157         conf->capabilities = prev->capabilities;
158     }
159
160     if (conf->capabilities.nelts == 0) {
161
162         for (d = ngx_mail_imap_default_capabilities; d->len; d++) {
163             c = ngx_array_push(&conf->capabilities);
164             if (c == NULL) {
165                 return NGX_CONF_ERROR;
166             }
167
168             *c = *d;
169         }
170     }
171
172     size = sizeof("** CAPABILITY" CRLF) - 1;
173
174     c = conf->capabilities.elts;
175     for (i = 0; i < conf->capabilities.nelts; i++) {
176         size += 1 + c[i].len;
177     }
178
179     for (m = NGX_MAIL_AUTH_PLAIN_ENABLED, i = 0;
180          m <= NGX_MAIL_AUTH_CRAM_MD5_ENABLED;
181          m <<= 1, i++)
182     {
183         if (m & conf->auth_methods) {
184             size += 1 + ngx_mail_imap_auth_methods_names[i].len;
185         }
186     }
187
188     p = ngx_pnalloc(cf->pool, size);
189     if (p == NULL) {
190         return NGX_CONF_ERROR;
191     }
192
193     conf->capability.len = size;

```

```

194 conf->capability.data = p;
195
196 p = ngx_cpymem(p, "* CAPABILITY", sizeof("* CAPABILITY") - 1);
197
198 for (i = 0; i < conf->capabilities.nelts; i++) {
199     *p++ = ' ';
200     p = ngx_cpymem(p, c[i].data, c[i].len);
201 }
202
203 auth = p;
204
205 for (m = NGX_MAIL_AUTH_PLAIN_ENABLED, i = 0;
206      m <= NGX_MAIL_AUTH_CRAM_MD5_ENABLED;
207      m <<= 1, i++)
208 {
209     if (m & conf->auth_methods) {
210         *p++ = ' ';
211         p = ngx_cpymem(p, ngx_mail_imap_auth_methods_names[i].data,
212                       ngx_mail_imap_auth_methods_names[i].len);
213     }
214 }
215
216 *p++ = CR; *p = LF;
217
218 size += sizeof(" STARTTLS") - 1;
219
220 p = ngx_pnalloc(cf->pool, size);
221 if (p == NULL) {
222     return NGX_CONF_ERROR;
223 }
224
225 conf->starttls_capability.len = size;
226 conf->starttls_capability.data = p;
227
228 p = ngx_cpymem(p, conf->capability.data,
229               conf->capability.len - (sizeof(CRLF) - 1));
230 p = ngx_cpymem(p, " STARTTLS", sizeof(" STARTTLS") - 1);
231 *p++ = CR; *p = LF;
232
233 size = (auth - conf->capability.data) + sizeof(CRLF) - 1
234       + sizeof(" STARTTLS LOGINDISABLED") - 1;
235
236 p = ngx_pnalloc(cf->pool, size);
237 if (p == NULL) {
238     return NGX_CONF_ERROR;
239 }
240
241 conf->starttls_only_capability.len = size;
242 conf->starttls_only_capability.data = p;
243
244 p = ngx_cpymem(p, conf->capability.data,
245               auth - conf->capability.data);
246 p = ngx_cpymem(p, " STARTTLS LOGINDISABLED",
247               sizeof(" STARTTLS LOGINDISABLED") - 1);
248 *p++ = CR; *p = LF;
249
250 return NGX_CONF_OK;
251 }

```

[One Level Up](#)

[Top Level](#)

# src/mail/nginx\_mail\_imap\_handler.c - nginx-1.7.10

## Global variables defined

- [imap\\_bye](#)
- [imap\\_greeting](#)
- [imap\\_invalid\\_command](#)
- [imap\\_next](#)
- [imap\\_ok](#)
- [imap\\_password](#)
- [imap\\_plain\\_next](#)
- [imap\\_star](#)
- [imap\\_username](#)

## Functions defined

- [ngx\\_mail\\_imap\\_auth\\_state](#)
- [ngx\\_mail\\_imap\\_authenticate](#)
- [ngx\\_mail\\_imap\\_capability](#)
- [ngx\\_mail\\_imap\\_init\\_protocol](#)
- [ngx\\_mail\\_imap\\_init\\_session](#)
- [ngx\\_mail\\_imap\\_login](#)
- [ngx\\_mail\\_imap\\_starttls](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11 #include <ngx_mail.h>
12 #include <ngx_mail_imap_module.h>
13
14
15 static ngx_int_t ngx_mail_imap_login(ngx_mail_session_t *s,
16     ngx_connection_t *c);
17 static ngx_int_t ngx_mail_imap_authenticate(ngx_mail_session_t *s,
18     ngx_connection_t *c);
19 static ngx_int_t ngx_mail_imap_capability(ngx_mail_session_t *s,
20     ngx_connection_t *c);
21 static ngx_int_t ngx_mail_imap_starttls(ngx_mail_session_t *s,
22     ngx_connection_t *c);
23
24
25 static u_char  imap_greeting[] = "* OK IMAP4 ready" CRLF;
```

```

26 static u_char  imap_star[] = "* ";
27 static u_char  imap_ok[] = "OK completed" CRLF;
28 static u_char  imap_next[] = "+ OK" CRLF;
29 static u_char  imap_plain_next[] = "+ " CRLF;
30 static u_char  imap_username[] = "+ VXNlcm5hbWU6" CRLF;
31 static u_char  imap_password[] = "+ UGFzc3dvcmQ6" CRLF;
32 static u_char  imap_bye[] = "* BYE" CRLF;
33 static u_char  imap_invalid_command[] = "BAD invalid command" CRLF;
34
35
36 void
37 ngx_mail_imap_init_session(ngx_mail_session_t *s, ngx_connection_t *c)
38 {
39     ngx_mail_core_srv_conf_t *cscf;
40
41     cscf = ngx_mail_get_module_srv_conf(s, ngx_mail_core_module);
42
43     ngx_str_set(&s->out, imap_greeting);
44
45     c->read->handler = ngx_mail_imap_init_protocol;
46
47     ngx_add_timer(c->read, cscf->timeout);
48
49     if (ngx_handle_read_event(c->read, 0) != NGX_OK) {
50         ngx_mail_close_connection(c);
51     }
52
53     ngx_mail_send(c->write);
54 }
55
56
57 void
58 ngx_mail_imap_init_protocol(ngx_event_t *rev)
59 {
60     ngx_connection_t *c;
61     ngx_mail_session_t *s;
62     ngx_mail_imap_srv_conf_t *iscf;
63
64     c = rev->data;
65
66     c->log->action = "in auth state";
67
68     if (rev->timedout) {
69         ngx_log_error(NGX_LOG_INFO, c->log, NGX_ETIMEDOUT, "client timed out");
70         c->timedout = 1;
71         ngx_mail_close_connection(c);
72         return;
73     }
74
75     s = c->data;
76
77     if (s->buffer == NULL) {
78         if (ngx_array_init(&s->args, c->pool, 2, sizeof(ngx_str_t))
79             == NGX_ERROR)
80         {
81             ngx_mail_session_internal_server_error(s);
82             return;
83         }
84
85         iscf = ngx_mail_get_module_srv_conf(s, ngx_mail_imap_module);
86
87         s->buffer = ngx_create_temp_buf(c->pool, iscf->client_buffer_size);
88         if (s->buffer == NULL) {
89             ngx_mail_session_internal_server_error(s);
90             return;
91         }
92     }
93
94     s->mail_state = ngx_imap_start;
95     c->read->handler = ngx_mail_imap_auth_state;
96
97     ngx_mail_imap_auth_state(rev);
98 }
99
100
101 void

```

```

102 ngx_mail_imap_auth_state(ngx_event_t *rev)
103 {
104     u_char          *p, *dst, *src, *end;
105     ngx_str_t       *arg;
106     ngx_int_t       rc;
107     ngx_uint_t      tag, i;
108     ngx_connection_t *c;
109     ngx_mail_session_t *s;
110
111     c = rev->data;
112     s = c->data;
113
114     ngx_log_debug0(NGX_LOG_DEBUG_MAIL, c->log, 0, "imap auth state");
115
116     if (rev->timedout) {
117         ngx_log_error(NGX_LOG_INFO, c->log, NGX_ETIMEDOUT, "client timed out");
118         c->timedout = 1;
119         ngx_mail_close_connection(c);
120         return;
121     }
122
123     if (s->out.len) {
124         ngx_log_debug0(NGX_LOG_DEBUG_MAIL, c->log, 0, "imap send handler busy");
125         s->blocked = 1;
126         return;
127     }
128
129     s->blocked = 0;
130
131     rc = ngx_mail_read_command(s, c);
132
133     if (rc == NGX_AGAIN || rc == NGX_ERROR) {
134         return;
135     }
136
137     tag = 1;
138     s->text.len = 0;
139     ngx_str_set(&s->out, imap_ok);
140
141     if (rc == NGX_OK) {
142
143         ngx_log_debug1(NGX_LOG_DEBUG_MAIL, c->log, 0, "imap auth command: %i",
144             s->command);
145
146         if (s->backslash) {
147
148             arg = s->args.elts;
149
150             for (i = 0; i < s->args.nelts; i++) {
151                 dst = arg[i].data;
152                 end = dst + arg[i].len;
153
154                 for (src = dst; src < end; dst++) {
155                     *dst = *src;
156                     if (*src++ == '\\') {
157                         *dst = *src++;
158                     }
159                 }
160
161                 arg[i].len = dst - arg[i].data;
162             }
163
164             s->backslash = 0;
165         }
166
167         switch (s->mail_state) {
168
169         case ngx_imap_start:
170
171             switch (s->command) {
172
173             case NGX_IMAP_LOGIN:
174                 rc = ngx_mail_imap_login(s, c);
175                 break;
176
177             case NGX_IMAP_AUTHENTICATE:

```

```

178         rc = ngx_mail_imap_authenticate(s, c);
179         tag = (rc != NGX_OK);
180         break;
181
182     case NGX_IMAP_CAPABILITY:
183         rc = ngx_mail_imap_capability(s, c);
184         break;
185
186     case NGX_IMAP_LOGOUT:
187         s->quit = 1;
188         ngx_str_set(&s->text, imap_bye);
189         break;
190
191     case NGX_IMAP_NOOP:
192         break;
193
194     case NGX_IMAP_STARTTLS:
195         rc = ngx_mail_imap_starttls(s, c);
196         break;
197
198     default:
199         rc = NGX_MAIL_PARSE_INVALID_COMMAND;
200         break;
201     }
202
203     break;
204
205     case ngx_imap_auth_login_username:
206         rc = ngx_mail_auth_login_username(s, c, 0);
207
208         tag = 0;
209         ngx_str_set(&s->out, imap_password);
210         s->mail_state = ngx_imap_auth_login_password;
211
212         break;
213
214     case ngx_imap_auth_login_password:
215         rc = ngx_mail_auth_login_password(s, c);
216         break;
217
218     case ngx_imap_auth_plain:
219         rc = ngx_mail_auth_plain(s, c, 0);
220         break;
221
222     case ngx_imap_auth_cram_md5:
223         rc = ngx_mail_auth_cram_md5(s, c);
224         break;
225     }
226
227 } else if (rc == NGX_IMAP_NEXT) {
228     tag = 0;
229     ngx_str_set(&s->out, imap_next);
230 }
231
232 switch (rc) {
233
234     case NGX_DONE:
235         ngx_mail_auth(s, c);
236         return;
237
238     case NGX_ERROR:
239         ngx_mail_session_internal_server_error(s);
240         return;
241
242     case NGX_MAIL_PARSE_INVALID_COMMAND:
243         s->state = 0;
244         ngx_str_set(&s->out, imap_invalid_command);
245         s->mail_state = ngx_imap_start;
246         break;
247     }
248
249 if (tag) {
250     if (s->tag.len == 0) {
251         ngx_str_set(&s->tag, imap_star);
252     }
253

```



```

254     if (s->tagged_line.len < s->tag.len + s->text.len + s->out.len) {
255         s->tagged_line.len = s->tag.len + s->text.len + s->out.len;
256         s->tagged_line.data = ngx_pnalloc(c->pool, s->tagged_line.len);
257         if (s->tagged_line.data == NULL) {
258             ngx_mail_close_connection(c);
259             return;
260         }
261     }
262
263     p = s->tagged_line.data;
264
265     if (s->text.len) {
266         p = ngx_cpymem(p, s->text.data, s->text.len);
267     }
268
269     p = ngx_cpymem(p, s->tag.data, s->tag.len);
270     ngx_memcpy(p, s->out.data, s->out.len);
271
272     s->out.len = s->text.len + s->tag.len + s->out.len;
273     s->out.data = s->tagged_line.data;
274 }
275
276 if (rc != NGX_IMAP_NEXT) {
277     s->args.nelts = 0;
278
279     if (s->state) {
280         /* preserve tag */
281         s->arg_start = s->buffer->start + s->tag.len;
282         s->buffer->pos = s->arg_start;
283         s->buffer->last = s->arg_start;
284     } else {
285         s->buffer->pos = s->buffer->start;
286         s->buffer->last = s->buffer->start;
287         s->tag.len = 0;
288     }
289 }
290 }
291
292 ngx_mail_send(c->write);
293 }
294
295
296 static ngx_int_t
297 ngx_mail_imap_login(ngx_mail_session_t *s, ngx_connection_t *c)
298 {
299     ngx_str_t *arg;
300
301     #if (NGX_MAIL_SSL)
302     if (ngx_mail_starttls_only(s, c)) {
303         return NGX_MAIL_PARSE_INVALID_COMMAND;
304     }
305     #endif
306
307     arg = s->args.elts;
308
309     if (s->args.nelts != 2 || arg[0].len == 0) {
310         return NGX_MAIL_PARSE_INVALID_COMMAND;
311     }
312
313     s->login.len = arg[0].len;
314     s->login.data = ngx_pnalloc(c->pool, s->login.len);
315     if (s->login.data == NULL) {
316         return NGX_ERROR;
317     }
318
319     ngx_memcpy(s->login.data, arg[0].data, s->login.len);
320
321     s->passwd.len = arg[1].len;
322     s->passwd.data = ngx_pnalloc(c->pool, s->passwd.len);
323     if (s->passwd.data == NULL) {
324         return NGX_ERROR;
325     }
326
327     ngx_memcpy(s->passwd.data, arg[1].data, s->passwd.len);
328
329     #if (NGX_DEBUG_MAIL_PASSWD)

```

```

330     ngx_log_debug2(NGX_LOG_DEBUG_MAIL, c->log, 0,
331                 "imap login:\"%V\" passwd:\"%V\"",
332                 &s->login, &s->passwd);
333 #else
334     ngx_log_debug1(NGX_LOG_DEBUG_MAIL, c->log, 0,
335                 "imap login:\"%V\"", &s->login);
336 #endif
337
338     return NGX_DONE;
339 }
340
341
342 static ngx_int_t
343 ngx_mail_imap_authenticate(ngx_mail_session_t *s, ngx_connection_t *c)
344 {
345     ngx_int_t          rc;
346     ngx_mail_core_srv_conf_t *cscf;
347     ngx_mail_imap_srv_conf_t *iscf;
348
349     #if (NGX_MAIL_SSL)
350     if (ngx_mail_starttls_only(s, c)) {
351         return NGX_MAIL_PARSE_INVALID_COMMAND;
352     }
353     #endif
354
355     rc = ngx_mail_auth_parse(s, c);
356
357     switch (rc) {
358
359     case NGX_MAIL_AUTH_LOGIN:
360
361         ngx_str_set(&s->out, imap_username);
362         s->mail_state = ngx_imap_auth_login_username;
363
364         return NGX_OK;
365
366     case NGX_MAIL_AUTH_LOGIN_USERNAME:
367
368         ngx_str_set(&s->out, imap_password);
369         s->mail_state = ngx_imap_auth_login_password;
370
371         return ngx_mail_auth_login_username(s, c, 1);
372
373     case NGX_MAIL_AUTH_PLAIN:
374
375         ngx_str_set(&s->out, imap_plain_next);
376         s->mail_state = ngx_imap_auth_plain;
377
378         return NGX_OK;
379
380     case NGX_MAIL_AUTH_CRAM_MD5:
381
382         iscf = ngx_mail_get_module_srv_conf(s, ngx_mail_imap_module);
383
384         if (!(iscf->auth_methods & NGX_MAIL_AUTH_CRAM_MD5_ENABLED)) {
385             return NGX_MAIL_PARSE_INVALID_COMMAND;
386         }
387
388         if (s->salt.data == NULL) {
389             cscf = ngx_mail_get_module_srv_conf(s, ngx_mail_core_module);
390
391             if (ngx_mail_salt(s, c, cscf) != NGX_OK) {
392                 return NGX_ERROR;
393             }
394         }
395
396         if (ngx_mail_auth_cram_md5_salt(s, c, "+ ", 2) == NGX_OK) {
397             s->mail_state = ngx_imap_auth_cram_md5;
398             return NGX_OK;
399         }
400
401         return NGX_ERROR;
402     }
403
404     return rc;
405 }

```

```

406
407
408 static ngx_int_t
409 ngx_mail_imap_capability(ngx_mail_session_t *s, ngx_connection_t *c)
410 {
411     ngx_mail_imap_srv_conf_t *iscf;
412
413     iscf = ngx_mail_get_module_srv_conf(s, ngx_mail_imap_module);
414
415     #if (NGX_MAIL_SSL)
416
417     if (c->ssl == NULL) {
418         ngx_mail_ssl_conf_t *sslcf;
419
420         sslcf = ngx_mail_get_module_srv_conf(s, ngx_mail_ssl_module);
421
422         if (sslcf->starttls == NGX_MAIL_STARTTLS_ON) {
423             s->text = iscf->starttls_capability;
424             return NGX_OK;
425         }
426
427         if (sslcf->starttls == NGX_MAIL_STARTTLS_ONLY) {
428             s->text = iscf->starttls_only_capability;
429             return NGX_OK;
430         }
431     }
432     #endif
433
434     s->text = iscf->capability;
435
436     return NGX_OK;
437 }
438
439
440 static ngx_int_t
441 ngx_mail_imap_starttls(ngx_mail_session_t *s, ngx_connection_t *c)
442 {
443     #if (NGX_MAIL_SSL)
444         ngx_mail_ssl_conf_t *sslcf;
445
446         if (c->ssl == NULL) {
447             sslcf = ngx_mail_get_module_srv_conf(s, ngx_mail_ssl_module);
448             if (sslcf->starttls) {
449                 c->read->handler = ngx_mail_starttls_handler;
450                 return NGX_OK;
451             }
452         }
453     #endif
454
455     return NGX_MAIL_PARSE_INVALID_COMMAND;
456 }

```

[One Level Up](#)

[Top Level](#)

# src/mail/nginx\_mail\_imap\_module.h - nginx-1.7.10

## Data types defined

- [ngx\\_mail\\_imap\\_srv\\_conf\\_t](#)

## Macros defined

- [\\_NGX\\_MAIL\\_IMAP\\_MODULE\\_H\\_INCLUDED](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_MAIL\_IMAP\_MODULE\_H\_INCLUDED
9 #define \_NGX\_MAIL\_IMAP\_MODULE\_H\_INCLUDED
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14 #include <ngx_mail.h>
15
16
17 typedef struct {
18     size_t      client_buffer_size;
19
20     ngx\_str\_t    capability;
21     ngx\_str\_t    starttls_capability;
22     ngx\_str\_t    starttls_only_capability;
23
24     ngx\_uint\_t  auth_methods;
25
26     ngx\_array\_t capabilities;
27 } ngx\_mail\_imap\_srv\_conf\_t;
28
29
30 void ngx\_mail\_imap\_init\_session(ngx\_mail\_session\_t *s, ngx\_connection\_t *c);
31 void ngx\_mail\_imap\_init\_protocol(ngx\_event\_t *rev);
32 void ngx\_mail\_imap\_auth\_state(ngx\_event\_t *rev);
33 ngx\_int\_t ngx\_mail\_imap\_parse\_command(ngx\_mail\_session\_t *s);
34
35
36 extern ngx\_module\_t ngx\_mail\_imap\_module;
37
38
39 #endif /* \_NGX\_MAIL\_IMAP\_MODULE\_H\_INCLUDED */
```

# src/mail/nginx\_mail\_smtp\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_mail\\_smtp\\_auth\\_methods](#)
- [ngx\\_mail\\_smtp\\_auth\\_methods\\_names](#)
- [ngx\\_mail\\_smtp\\_commands](#)
- [ngx\\_mail\\_smtp\\_module](#)
- [ngx\\_mail\\_smtp\\_module\\_ctx](#)
- [ngx\\_mail\\_smtp\\_protocol](#)

## Functions defined

- [ngx\\_mail\\_smtp\\_create\\_srv\\_conf](#)
- [ngx\\_mail\\_smtp\\_merge\\_srv\\_conf](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11 #include <ngx_mail.h>
12 #include <ngx_mail_smtp_module.h>
13
14
15 static void *ngx_mail_smtp_create_srv_conf(ngx_conf_t *cf);
16 static char *ngx_mail_smtp_merge_srv_conf(ngx_conf_t *cf, void *parent,
17     void *child);
18
19
20 static ngx_conf_bitmask_t ngx_mail_smtp_auth_methods[] = {
21     { ngx_string("plain"), NGX_MAIL_AUTH_PLAIN_ENABLED },
22     { ngx_string("login"), NGX_MAIL_AUTH_LOGIN_ENABLED },
23     { ngx_string("cram-md5"), NGX_MAIL_AUTH_CRAM_MD5_ENABLED },
24     { ngx_string("none"), NGX_MAIL_AUTH_NONE_ENABLED },
25     { ngx_null_string, 0 }
26 };
27
28
29 static ngx_str_t ngx_mail_smtp_auth_methods_names[] = {
30     ngx_string("PLAIN"),
31     ngx_string("LOGIN"),
32     ngx_null_string, /* APOP */
33     ngx_string("CRAM-MD5"),
34     ngx_null_string /* NONE */
35 };
36
37
38 static ngx_mail_protocol_t ngx_mail_smtp_protocol = {
39     ngx_string("smtp"),
40     { 25, 465, 587, 0 },
41     NGX_MAIL_SMTP_PROTOCOL,
42
43     ngx_mail_smtp_init_session,
```

```

44     ngx_mail_smtp_init_protocol,
45     ngx_mail_smtp_parse_command,
46     ngx_mail_smtp_auth_state,
47
48     ngx_string("451 4.3.2 Internal server error" CRLF)
49 };
50
51
52 static ngx_command_t  ngx_mail_smtp_commands[] = {
53
54     { ngx_string("smtp_client_buffer"),
55       NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_TAKE1,
56       ngx_conf_set_size_slot,
57       NGX_MAIL_SRV_CONF_OFFSET,
58       offsetof(ngx_mail_smtp_srv_conf_t, client_buffer_size),
59       NULL },
60
61     { ngx_string("smtp_greeting_delay"),
62       NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_TAKE1,
63       ngx_conf_set_msec_slot,
64       NGX_MAIL_SRV_CONF_OFFSET,
65       offsetof(ngx_mail_smtp_srv_conf_t, greeting_delay),
66       NULL },
67
68     { ngx_string("smtp_capabilities"),
69       NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_1MORE,
70       ngx_mail_capabilities,
71       NGX_MAIL_SRV_CONF_OFFSET,
72       offsetof(ngx_mail_smtp_srv_conf_t, capabilities),
73       NULL },
74
75     { ngx_string("smtp_auth"),
76       NGX_MAIL_MAIN_CONF|NGX_MAIL_SRV_CONF|NGX_CONF_1MORE,
77       ngx_conf_set_bitmask_slot,
78       NGX_MAIL_SRV_CONF_OFFSET,
79       offsetof(ngx_mail_smtp_srv_conf_t, auth_methods),
80       &ngx_mail_smtp_auth_methods },
81
82     ngx_null_command
83 };
84
85
86 static ngx_mail_module_t  ngx_mail_smtp_module_ctx = {
87     &ngx_mail_smtp_protocol,          /* protocol */
88
89     NULL,                             /* create main configuration */
90     NULL,                             /* init main configuration */
91
92     ngx_mail_smtp_create_srv_conf,     /* create server configuration */
93     ngx_mail_smtp_merge_srv_conf      /* merge server configuration */
94 };
95
96
97 ngx_module_t  ngx_mail_smtp_module = {
98     NGX_MODULE_V1,
99     &ngx_mail_smtp_module_ctx,        /* module context */
100    ngx_mail_smtp_commands,           /* module directives */
101    NGX_MAIL_MODULE,                  /* module type */
102    NULL,                             /* init master */
103    NULL,                             /* init module */
104    NULL,                             /* init process */
105    NULL,                             /* init thread */
106    NULL,                             /* exit thread */
107    NULL,                             /* exit process */
108    NULL,                             /* exit master */
109    NGX_MODULE_V1_PADDING
110 };
111
112
113 static void *
114 ngx_mail_smtp_create_srv_conf(ngx_conf_t *cf)
115 {
116     ngx_mail_smtp_srv_conf_t  *sscf;
117
118     sscf = ngx_palloc(cf->pool, sizeof(ngx_mail_smtp_srv_conf_t));
119     if (sscf == NULL) {

```

```

120     return NULL;
121 }
122
123 sscf->client_buffer_size = NGX\_CONF\_UNSET\_SIZE;
124 sscf->greeting_delay = NGX\_CONF\_UNSET\_MSEC;
125
126 if (ngx\_array\_init(&sscf->capabilities, cf->pool, 4, sizeof\(ngx\_str\_t\))
127     != NGX\_OK)
128 {
129     return NULL;
130 }
131
132 return sscf;
133 }
134
135
136 static char *
137 ngx_mail_smtp_merge_srv_conf(ngx\_conf\_t *cf, void *parent, void *child)
138 {
139     ngx\_mail\_smtp\_srv\_conf\_t *prev = parent;
140     ngx\_mail\_smtp\_srv\_conf\_t *conf = child;
141
142     u_char                *p, *auth, *last;
143     size_t                size;
144     ngx\_str\_t             *c;
145     ngx\_uint\_t           i, m, auth_enabled;
146     ngx\_mail\_core\_srv\_conf\_t *cscf;
147
148     ngx\_conf\_merge\_size\_value(conf->client_buffer_size,
149                             prev->client_buffer_size,
150                             (size_t) ngx\_pagesize);
151
152     ngx\_conf\_merge\_msec\_value(conf->greeting_delay,
153                             prev->greeting_delay, 0);
154
155     ngx\_conf\_merge\_bitmask\_value(conf->auth_methods,
156                                 prev->auth_methods,
157                                 (NGX\_CONF\_BITMASK\_SET
158                                  |NGX\_MAIL\_AUTH\_PLAIN\_ENABLED
159                                  |NGX\_MAIL\_AUTH\_LOGIN\_ENABLED));
160
161
162     cscf = ngx\_mail\_conf\_get\_module\_srv\_conf(cf, ngx\_mail\_core\_module);
163
164     size = sizeof("220  ESMTP ready" CRLF) - 1 + cscf->server_name.len;
165
166     p = ngx\_pnalloc(cf->pool, size);
167     if (p == NULL) {
168         return NGX\_CONF\_ERROR;
169     }
170
171     conf->greeting.len = size;
172     conf->greeting.data = p;
173
174     *p++ = '2'; *p++ = '2'; *p++ = '0'; *p++ = ' ';
175     p = ngx\_cpymem(p, cscf->server_name.data, cscf->server_name.len);
176     ngx\_memcpy(p, " ESMTP ready" CRLF, sizeof(" ESMTP ready" CRLF) - 1);
177
178
179     size = sizeof("250 " CRLF) - 1 + cscf->server_name.len;
180
181     p = ngx\_pnalloc(cf->pool, size);
182     if (p == NULL) {
183         return NGX\_CONF\_ERROR;
184     }
185
186     conf->server_name.len = size;
187     conf->server_name.data = p;
188
189     *p++ = '2'; *p++ = '5'; *p++ = '0'; *p++ = ' ';
190     p = ngx\_cpymem(p, cscf->server_name.data, cscf->server_name.len);
191     *p++ = CR; *p = LF;
192
193
194     if (conf->capabilities.nelts == 0) {
195         conf->capabilities = prev->capabilities;

```

```

196 }
197
198 size = sizeof("250-") - 1 + cscf->server_name.len + sizeof(CRLF) - 1;
199
200 c = conf->capabilities.elts;
201 for (i = 0; i < conf->capabilities.nelts; i++) {
202     size += sizeof("250 ") - 1 + c[i].len + sizeof(CRLF) - 1;
203 }
204
205 auth_enabled = 0;
206
207 for (m = NGX_MAIL_AUTH_PLAIN_ENABLED, i = 0;
208     m <= NGX_MAIL_AUTH_CRAM_MD5_ENABLED;
209     m <<= 1, i++)
210 {
211     if (m & conf->auth_methods) {
212         size += 1 + ngx_mail_smtp_auth_methods_names[i].len;
213         auth_enabled = 1;
214     }
215 }
216
217 if (auth_enabled) {
218     size += sizeof("250 AUTH") - 1 + sizeof(CRLF) - 1;
219 }
220
221 p = ngx_pnalloc(cf->pool, size);
222 if (p == NULL) {
223     return NGX_CONF_ERROR;
224 }
225
226 conf->capability.len = size;
227 conf->capability.data = p;
228
229 last = p;
230
231 *p++ = '2'; *p++ = '5'; *p++ = '0'; *p++ = '-';
232 p = ngx_cpymem(p, cscf->server_name.data, cscf->server_name.len);
233 *p++ = CR; *p++ = LF;
234
235 for (i = 0; i < conf->capabilities.nelts; i++) {
236     last = p;
237     *p++ = '2'; *p++ = '5'; *p++ = '0'; *p++ = '-';
238     p = ngx_cpymem(p, c[i].data, c[i].len);
239     *p++ = CR; *p++ = LF;
240 }
241
242 auth = p;
243
244 if (auth_enabled) {
245     last = p;
246
247     *p++ = '2'; *p++ = '5'; *p++ = '0'; *p++ = ' ';
248     *p++ = 'A'; *p++ = 'U'; *p++ = 'T'; *p++ = 'H';
249
250     for (m = NGX_MAIL_AUTH_PLAIN_ENABLED, i = 0;
251         m <= NGX_MAIL_AUTH_CRAM_MD5_ENABLED;
252         m <<= 1, i++)
253     {
254         if (m & conf->auth_methods) {
255             *p++ = ' ';
256             p = ngx_cpymem(p, ngx_mail_smtp_auth_methods_names[i].data,
257                           ngx_mail_smtp_auth_methods_names[i].len);
258         }
259     }
260
261     *p++ = CR; *p = LF;
262
263 } else {
264     last[3] = ' ';
265 }
266
267 size += sizeof("250 STARTTLS" CRLF) - 1;
268
269 p = ngx_pnalloc(cf->pool, size);
270 if (p == NULL) {
271     return NGX_CONF_ERROR;

```



```
272 }
273
274 conf->starttls_capability.len = size;
275 conf->starttls_capability.data = p;
276
277 p = ngx_cpymem(p, conf->capability.data, conf->capability.len);
278
279 p = ngx_cpymem(p, "250 STARTTLS" CRLF, sizeof("250 STARTTLS" CRLF) - 1);
280
281 p = conf->starttls_capability.data
282     + (last - conf->capability.data) + 3;
283 *p = '-';
284
285 size = (auth - conf->capability.data)
286     + sizeof("250 STARTTLS" CRLF) - 1;
287
288 p = ngx_pnalloc(cf->pool, size);
289 if (p == NULL) {
290     return NGX_CONF_ERROR;
291 }
292
293 conf->starttls_only_capability.len = size;
294 conf->starttls_only_capability.data = p;
295
296 p = ngx_cpymem(p, conf->capability.data, auth - conf->capability.data);
297
298 ngx_memcpy(p, "250 STARTTLS" CRLF, sizeof("250 STARTTLS" CRLF) - 1);
299
300 if (last < auth) {
301     p = conf->starttls_only_capability.data
302         + (last - conf->capability.data) + 3;
303     *p = '-';
304 }
305
306 return NGX_CONF_OK;
307 }
```

[One Level Up](#)

[Top Level](#)

## src/mail/nginx\_mail\_smtp\_handler.c - nginx-1.7.10

### Global variables defined

- [smtp\\_auth\\_required](#)
- [smtp\\_bad\\_sequence](#)
- [smtp\\_bye](#)
- [smtp\\_invalid\\_argument](#)
- [smtp\\_invalid\\_command](#)
- [smtp\\_invalid\\_pipelining](#)
- [smtp\\_next](#)
- [smtp\\_ok](#)
- [smtp\\_password](#)
- [smtp\\_starttls](#)
- [smtp\\_tempunavail](#)
- [smtp\\_unavailable](#)
- [smtp\\_username](#)

### Functions defined

- [ngx\\_mail\\_smtp\\_auth](#)
- [ngx\\_mail\\_smtp\\_auth\\_state](#)
- [ngx\\_mail\\_smtp\\_create\\_buffer](#)
- [ngx\\_mail\\_smtp\\_discard\\_command](#)
- [ngx\\_mail\\_smtp\\_greeting](#)
- [ngx\\_mail\\_smtp\\_helo](#)
- [ngx\\_mail\\_smtp\\_init\\_protocol](#)
- [ngx\\_mail\\_smtp\\_init\\_session](#)
- [ngx\\_mail\\_smtp\\_invalid\\_pipelining](#)
- [ngx\\_mail\\_smtp\\_log\\_rejected\\_command](#)
- [ngx\\_mail\\_smtp\\_mail](#)
- [ngx\\_mail\\_smtp\\_rcpt](#)
- [ngx\\_mail\\_smtp\\_resolve\\_addr\\_handler](#)
- [ngx\\_mail\\_smtp\\_resolve\\_name](#)
- [ngx\\_mail\\_smtp\\_resolve\\_name\\_handler](#)

- [ngx\\_mail\\_smtp\\_rset](#)
- [ngx\\_mail\\_smtp\\_starttls](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11 #include <ngx_mail.h>
12 #include <ngx_mail_smtp_module.h>
13
14
15 static void ngx_mail_smtp_resolve_addr_handler(ngx_resolver_ctx_t *ctx);
16 static void ngx_mail_smtp_resolve_name(ngx_event_t *rev);
17 static void ngx_mail_smtp_resolve_name_handler(ngx_resolver_ctx_t *ctx);
18 static void ngx_mail_smtp_greeting(ngx_mail_session_t *s, ngx_connection_t *c);
19 static void ngx_mail_smtp_invalid_pipelining(ngx_event_t *rev);
20 static ngx_int_t ngx_mail_smtp_create_buffer(ngx_mail_session_t *s,
21     ngx_connection_t *c);
22
23 static ngx_int_t ngx_mail_smtp_helo(ngx_mail_session_t *s, ngx_connection_t *c);
24 static ngx_int_t ngx_mail_smtp_auth(ngx_mail_session_t *s, ngx_connection_t *c);
25 static ngx_int_t ngx_mail_smtp_mail(ngx_mail_session_t *s, ngx_connection_t *c);
26 static ngx_int_t ngx_mail_smtp_starttls(ngx_mail_session_t *s,
27     ngx_connection_t *c);
28 static ngx_int_t ngx_mail_smtp_rset(ngx_mail_session_t *s, ngx_connection_t *c);
29 static ngx_int_t ngx_mail_smtp_rcpt(ngx_mail_session_t *s, ngx_connection_t *c);
30
31 static ngx_int_t ngx_mail_smtp_discard_command(ngx_mail_session_t *s,
32     ngx_connection_t *c, char *err);
33 static void ngx_mail_smtp_log_rejected_command(ngx_mail_session_t *s,
34     ngx_connection_t *c, char *err);
35
36
37 static u_char smtp_ok[] = "250 2.0.0 OK" CRLF;
38 static u_char smtp_bye[] = "221 2.0.0 Bye" CRLF;
39 static u_char smtp_starttls[] = "220 2.0.0 Start TLS" CRLF;
40 static u_char smtp_next[] = "334 " CRLF;
41 static u_char smtp_username[] = "334 VXNlcm5hbWU6" CRLF;
42 static u_char smtp_password[] = "334 UGFzc3dvcmQ6" CRLF;
43 static u_char smtp_invalid_command[] = "500 5.5.1 Invalid command" CRLF;
44 static u_char smtp_invalid_pipelining[] =
45     "503 5.5.0 Improper use of SMTP command pipelining" CRLF;
46 static u_char smtp_invalid_argument[] = "501 5.5.4 Invalid argument" CRLF;
47 static u_char smtp_auth_required[] = "530 5.7.1 Authentication required" CRLF;
48 static u_char smtp_bad_sequence[] = "503 5.5.1 Bad sequence of commands" CRLF;
49
50
51 static ngx_str_t smtp_unavailable = ngx_string("[UNAVAILABLE]");
52 static ngx_str_t smtp_tempunavail = ngx_string("[TEMPUNAVAIL]");
53
54
55 void
56 ngx_mail_smtp_init_session(ngx_mail_session_t *s, ngx_connection_t *c)
57 {
58     ngx_resolver_ctx_t *ctx;
59     ngx_mail_core_srv_conf_t *cscf;
60
61     cscf = ngx_mail_get_module_srv_conf(s, ngx_mail_core_module);
62
63     if (cscf->resolver == NULL) {
64         s->host = smtp_unavailable;
65         ngx_mail_smtp_greeting(s, c);
66         return;
67     }
68

```

```

69 #if (NGX_HAVE_UNIX_DOMAIN)
70     if (c->sockaddr->sa_family == AF_UNIX) {
71         s->host = smtp\_tempunavail;
72         ngx\_mail\_smtp\_greeting(s, c);
73         return;
74     }
75 #endif
76
77     c->log->action = "in resolving client address";
78
79     ctx = ngx\_resolve\_start(cscf->resolver, NULL);
80     if (ctx == NULL) {
81         ngx\_mail\_close\_connection(c);
82         return;
83     }
84
85     ctx->addr.sockaddr = c->sockaddr;
86     ctx->addr.socklen = c->socklen;
87     ctx->handler = ngx\_mail\_smtp\_resolve\_addr\_handler;
88     ctx->data = s;
89     ctx->timeout = cscf->resolver_timeout;
90
91     if (ngx\_resolve\_addr(ctx) != NGX_OK) {
92         ngx\_mail\_close\_connection(c);
93     }
94 }
95
96
97 static void
98 ngx\_mail\_smtp\_resolve\_addr\_handler(ngx\_resolver\_ctx\_t *ctx)
99 {
100     ngx\_connection\_t *c;
101     ngx\_mail\_session\_t *s;
102
103     s = ctx->data;
104     c = s->connection;
105
106     if (ctx->state) {
107         ngx\_log\_error(NGX_LOG_ERR, c->log, 0,
108             "%V could not be resolved (%i: %s)",
109             &c->addr_text, ctx->state,
110             ngx\_resolver\_strerror(ctx->state));
111
112         if (ctx->state == NGX_RESOLVE_NXDOMAIN) {
113             s->host = smtp\_unavailable;
114
115         } else {
116             s->host = smtp\_tempunavail;
117         }
118
119         ngx\_resolve\_addr\_done(ctx);
120
121         ngx\_mail\_smtp\_greeting(s, s->connection);
122
123         return;
124     }
125
126     c->log->action = "in resolving client hostname";
127
128     s->host.data = ngx\_pstrdup(c->pool, &ctx->name);
129     if (s->host.data == NULL) {
130         ngx\_resolve\_addr\_done(ctx);
131         ngx\_mail\_close\_connection(c);
132         return;
133     }
134
135     s->host.len = ctx->name.len;
136
137     ngx\_resolve\_addr\_done(ctx);
138
139     ngx\_log\_debug1(NGX_LOG_DEBUG_MAIL, c->log, 0,
140         "address resolved: %V", &s->host);
141
142     c->read->handler = ngx\_mail\_smtp\_resolve\_name;
143
144     ngx\_post\_event(c->read, &ngx\_posted\_events);

```

```

145 }
146
147
148 static void
149 ngx_mail_smtp_resolve_name(ngx_event_t *rev)
150 {
151     ngx_connection_t      *c;
152     ngx_mail_session_t    *s;
153     ngx_resolver_ctx_t    *ctx;
154     ngx_mail_core_srv_conf_t *cscf;
155
156     c = rev->data;
157     s = c->data;
158
159     cscf = ngx_mail_get_module_srv_conf(s, ngx_mail_core_module);
160
161     ctx = ngx_resolve_start(cscf->resolver, NULL);
162     if (ctx == NULL) {
163         ngx_mail_close_connection(c);
164         return;
165     }
166
167     ctx->name = s->host;
168     ctx->handler = ngx_mail_smtp_resolve_name_handler;
169     ctx->data = s;
170     ctx->timeout = cscf->resolver_timeout;
171
172     if (ngx_resolve_name(ctx) != NGX_OK) {
173         ngx_mail_close_connection(c);
174     }
175 }
176
177
178 static void
179 ngx_mail_smtp_resolve_name_handler(ngx_resolver_ctx_t *ctx)
180 {
181     ngx_uint_t            i;
182     ngx_connection_t      *c;
183     ngx_mail_session_t    *s;
184
185     s = ctx->data;
186     c = s->connection;
187
188     if (ctx->state) {
189         ngx_log_error(NGX_LOG_ERR, c->log, 0,
190                     "\"%V\" could not be resolved (%i: %s)",
191                     &ctx->name, ctx->state,
192                     ngx_resolver_strerror(ctx->state));
193
194         if (ctx->state == NGX_RESOLVE_NXDOMAIN) {
195             s->host = smtp_unavailable;
196
197         } else {
198             s->host = smtp_tempunavail;
199         }
200     } else {
201
202         #if (NGX_DEBUG)
203         {
204             u_char    text[NGX_SOCKADDR_STRLEN];
205             ngx_str_t  addr;
206
207             addr.data = text;
208
209             for (i = 0; i < ctx->naddrs; i++) {
210                 addr.len = ngx_sock_ntop(ctx->addrs[i].sockaddr,
211                                         ctx->addrs[i].socklen,
212                                         text, NGX_SOCKADDR_STRLEN, 0);
213
214                 ngx_log_debug1(NGX_LOG_DEBUG_MAIL, c->log, 0,
215                               "name was resolved to %V", &addr);
216             }
217         }
218     }
219 #endif
220

```

```

221     for (i = 0; i < ctx->naddrs; i++) {
222         if (ngx\_cmp\_sockaddr(ctx->addrs[i].sockaddr, ctx->addrs[i].socklen,
223             c->sockaddr, c->socklen, 0)
224             == NGX\_OK)
225             {
226                 goto found;
227             }
228     }
229
230     s->host = smtp\_unavailable;
231 }
232
233 found:
234
235     ngx\_resolve\_name\_done(ctx);
236
237     ngx\_mail\_smtp\_greeting(s, c);
238 }
239
240
241 static void
242 ngx\_mail\_smtp\_greeting(ngx\_mail\_session\_t *s, ngx\_connection\_t *c)
243 {
244     ngx\_msec\_t          timeout;
245     ngx\_mail\_core\_srv\_conf\_t *cscf;
246     ngx\_mail\_smtp\_srv\_conf\_t *sscf;
247
248     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_MAIL, c->log, 0,
249         "smtp greeting for \"%V\"", &s->host);
250
251     cscf = ngx\_mail\_get\_module\_srv\_conf(s, ngx\_mail\_core\_module);
252     sscf = ngx\_mail\_get\_module\_srv\_conf(s, ngx\_mail\_smtp\_module);
253
254     timeout = sscf->greeting_delay ? sscf->greeting_delay : cscf->timeout;
255     ngx\_add\_timer(c->read, timeout);
256
257     if (ngx\_handle\_read\_event(c->read, 0) != NGX\_OK) {
258         ngx\_mail\_close\_connection(c);
259     }
260
261     if (sscf->greeting_delay) {
262         c->read->handler = ngx\_mail\_smtp\_invalid\_pipelining;
263         return;
264     }
265
266     c->read->handler = ngx\_mail\_smtp\_init\_protocol;
267
268     s->out = sscf->greeting;
269
270     ngx\_mail\_send(c->write);
271 }
272
273
274 static void
275 ngx\_mail\_smtp\_invalid\_pipelining(ngx\_event\_t *rev)
276 {
277     ngx\_connection\_t      *c;
278     ngx\_mail\_session\_t    *s;
279     ngx\_mail\_core\_srv\_conf\_t *cscf;
280     ngx\_mail\_smtp\_srv\_conf\_t *sscf;
281
282     c = rev->data;
283     s = c->data;
284
285     c->log->action = "in delay pipelining state";
286
287     if (rev->timedout) {
288
289         ngx\_log\_debug0(NGX\_LOG\_DEBUG\_MAIL, c->log, 0, "delay greeting");
290
291         rev->timedout = 0;
292
293         cscf = ngx\_mail\_get\_module\_srv\_conf(s, ngx\_mail\_core\_module);
294
295         c->read->handler = ngx\_mail\_smtp\_init\_protocol;
296

```

```

297     ngx_add_timer(c->read, cscf->timeout);
298
299     if (ngx_handle_read_event(c->read, 0) != NGX_OK) {
300         ngx_mail_close_connection(c);
301         return;
302     }
303
304     sscf = ngx_mail_get_module_srv_conf(s, ngx_mail_smtp_module);
305
306     s->out = sscf->greeting;
307
308 } else {
309
310     ngx_log_debug0(NGX_LOG_DEBUG_MAIL, c->log, 0, "invalid pipelining");
311
312     if (s->buffer == NULL) {
313         if (ngx_mail_smtp_create_buffer(s, c) != NGX_OK) {
314             return;
315         }
316     }
317
318     if (ngx_mail_smtp_discard_command(s, c,
319                                     "client was rejected before greeting: \"%V\\\""
320                                     != NGX_OK)
321     {
322         return;
323     }
324
325     ngx_str_set(&s->out, smtp_invalid_pipelining);
326     s->quit = 1;
327 }
328
329 ngx_mail_send(c->write);
330 }
331
332
333 void
334 ngx_mail_smtp_init_protocol(ngx_event_t *rev)
335 {
336     ngx_connection_t *c;
337     ngx_mail_session_t *s;
338
339     c = rev->data;
340
341     c->log->action = "in auth state";
342
343     if (rev->timedout) {
344         ngx_log_error(NGX_LOG_INFO, c->log, NGX_ETIMEDOUT, "client timed out");
345         c->timedout = 1;
346         ngx_mail_close_connection(c);
347         return;
348     }
349
350     s = c->data;
351
352     if (s->buffer == NULL) {
353         if (ngx_mail_smtp_create_buffer(s, c) != NGX_OK) {
354             return;
355         }
356     }
357
358     s->mail_state = ngx_smtp_start;
359     c->read->handler = ngx_mail_smtp_auth_state;
360
361     ngx_mail_smtp_auth_state(rev);
362 }
363
364
365 static ngx_int_t
366 ngx_mail_smtp_create_buffer(ngx_mail_session_t *s, ngx_connection_t *c)
367 {
368     ngx_mail_smtp_srv_conf_t *sscf;
369
370     if (ngx_array_init(&s->args, c->pool, 2, sizeof(ngx_str_t)) == NGX_ERROR) {
371         ngx_mail_session_internal_server_error(s);
372         return NGX_ERROR;

```





```

449         break;
450
451     case NGX\_SMTP\_RSET:
452         rc = ngx\_mail\_smtp\_rset(s, c);
453         break;
454
455     case NGX\_SMTP\_NOOP:
456         break;
457
458     case NGX\_SMTP\_STARTTLS:
459         rc = ngx\_mail\_smtp\_starttls(s, c);
460         ngx\_str\_set(&s->out, smtp\_starttls);
461         break;
462
463     default:
464         rc = NGX\_MAIL\_PARSE\_INVALID\_COMMAND;
465         break;
466     }
467
468     break;
469
470 case ngx_smtp_auth_login_username:
471     rc = ngx\_mail\_auth\_login\_username(s, c, 0);
472
473     ngx\_str\_set(&s->out, smtp\_password);
474     s->mail_state = ngx_smtp_auth_login_password;
475     break;
476
477 case ngx_smtp_auth_login_password:
478     rc = ngx\_mail\_auth\_login\_password(s, c);
479     break;
480
481 case ngx_smtp_auth_plain:
482     rc = ngx\_mail\_auth\_plain(s, c, 0);
483     break;
484
485 case ngx_smtp_auth_cram_md5:
486     rc = ngx\_mail\_auth\_cram\_md5(s, c);
487     break;
488     }
489 }
490
491 if (s->buffer->pos < s->buffer->last) {
492     s->blocked = 1;
493 }
494
495 switch (rc) {
496
497 case NGX\_DONE:
498     ngx\_mail\_auth(s, c);
499     return;
500
501 case NGX\_ERROR:
502     ngx\_mail\_session\_internal\_server\_error(s);
503     return;
504
505 case NGX\_MAIL\_PARSE\_INVALID\_COMMAND:
506     s->mail_state = ngx_smtp_start;
507     s->state = 0;
508     ngx\_str\_set(&s->out, smtp\_invalid\_command);
509
510     /* fall through */
511
512 case NGX\_OK:
513     s->args.nelts = 0;
514
515     if (s->buffer->pos == s->buffer->last) {
516         s->buffer->pos = s->buffer->start;
517         s->buffer->last = s->buffer->start;
518     }
519
520     if (s->state) {
521         s->arg_start = s->buffer->pos;
522     }
523
524     ngx\_mail\_send(c->write);

```

```

525     }
526 }
527
528
529 static ngx_int_t
530 ngx_mail_smtp_helo(ngx_mail_session_t *s, ngx_connection_t *c)
531 {
532     ngx_str_t          *arg;
533     ngx_mail_smtp_srv_conf_t *sscf;
534
535     if (s->args.nelts != 1) {
536         ngx_str_set(&s->out, smtp_invalid_argument);
537         s->state = 0;
538         return NGX_OK;
539     }
540
541     arg = s->args.elts;
542
543     s->smtp_helo.len = arg[0].len;
544
545     s->smtp_helo.data = ngx_pnalloc(c->pool, arg[0].len);
546     if (s->smtp_helo.data == NULL) {
547         return NGX_ERROR;
548     }
549
550     ngx_memcpy(s->smtp_helo.data, arg[0].data, arg[0].len);
551
552     ngx_str_null(&s->smtp_from);
553     ngx_str_null(&s->smtp_to);
554
555     sscf = ngx_mail_get_module_srv_conf(s, ngx_mail_smtp_module);
556
557     if (s->command == NGX_SMTP_HELO) {
558         s->out = sscf->server_name;
559
560     } else {
561         s->esmtpl = 1;
562
563         #if (NGX_MAIL_SSL)
564
565             if (c->ssl == NULL) {
566                 ngx_mail_ssl_conf_t *sslcf;
567
568                 sslcf = ngx_mail_get_module_srv_conf(s, ngx_mail_ssl_module);
569
570                 if (sslcf->starttls == NGX_MAIL_STARTTLS_ON) {
571                     s->out = sscf->starttls_capability;
572                     return NGX_OK;
573                 }
574
575                 if (sslcf->starttls == NGX_MAIL_STARTTLS_ONLY) {
576                     s->out = sscf->starttls_only_capability;
577                     return NGX_OK;
578                 }
579             }
580         #endif
581
582         s->out = sscf->capability;
583     }
584
585     return NGX_OK;
586 }
587
588
589 static ngx_int_t
590 ngx_mail_smtp_auth(ngx_mail_session_t *s, ngx_connection_t *c)
591 {
592     ngx_int_t          rc;
593     ngx_mail_core_srv_conf_t *cscf;
594     ngx_mail_smtp_srv_conf_t *sscf;
595
596     #if (NGX_MAIL_SSL)
597     if (ngx_mail_starttls_only(s, c)) {
598         return NGX_MAIL_PARSE_INVALID_COMMAND;
599     }
600 #endif

```

```

601 if (s->args.nelts == 0) {
602     ngx_str_set(&s->out, smtp_invalid_argument);
603     s->state = 0;
604     return NGX_OK;
605 }
606
607 rc = ngx_mail_auth_parse(s, c);
608
609 switch (rc) {
610
611     case NGX_MAIL_AUTH_LOGIN:
612         ngx_str_set(&s->out, smtp_username);
613         s->mail_state = ngx_smtp_auth_login_username;
614         return NGX_OK;
615
616     case NGX_MAIL_AUTH_LOGIN_USERNAME:
617         ngx_str_set(&s->out, smtp_password);
618         s->mail_state = ngx_smtp_auth_login_password;
619         return ngx_mail_auth_login_username(s, c, 1);
620
621     case NGX_MAIL_AUTH_PLAIN:
622         ngx_str_set(&s->out, smtp_next);
623         s->mail_state = ngx_smtp_auth_plain;
624         return NGX_OK;
625
626     case NGX_MAIL_AUTH_CRAM_MD5:
627         sscf = ngx_mail_get_module_srv_conf(s, ngx_mail_smtp_module);
628         if (!(sscf->auth_methods & NGX_MAIL_AUTH_CRAM_MD5_ENABLED)) {
629             return NGX_MAIL_PARSE_INVALID_COMMAND;
630         }
631         if (s->salt.data == NULL) {
632             cscf = ngx_mail_get_module_srv_conf(s, ngx_mail_core_module);
633             if (ngx_mail_salt(s, c, cscf) != NGX_OK) {
634                 return NGX_ERROR;
635             }
636         }
637         if (ngx_mail_auth_cram_md5_salt(s, c, "334 ", 4) == NGX_OK) {
638             s->mail_state = ngx_smtp_auth_cram_md5;
639             return NGX_OK;
640         }
641         return NGX_ERROR;
642     }
643
644     return rc;
645 }
646
647 static ngx_int_t
648 ngx_mail_smtp_mail(ngx_mail_session_t *s, ngx_connection_t *c)
649 {
650     ngx_str_t          *arg, cmd;
651     ngx_mail_smtp_srv_conf_t *sscf;
652
653     sscf = ngx_mail_get_module_srv_conf(s, ngx_mail_smtp_module);
654
655     if (!(sscf->auth_methods & NGX_MAIL_AUTH_NONE_ENABLED)) {
656         ngx_mail_smtp_log_rejected_command(s, c, "client was rejected: \"%V\"",
657         ngx_str_set(&s->out, smtp_auth_required);
658         return NGX_OK;
659     }
660
661     /* auth none */

```

```

677     if (s->smtp_from.len) {
678         ngx_str_set(&s->out, smtp_bad_sequence);
679         return NGX_OK;
680     }
681
682     if (s->args.nelts == 0) {
683         ngx_str_set(&s->out, smtp_invalid_argument);
684         return NGX_OK;
685     }
686
687     arg = s->args.elts;
688     arg += s->args.nelts - 1;
689
690     cmd.len = arg->data + arg->len - s->cmd.data;
691     cmd.data = s->cmd.data;
692
693     s->smtp_from.len = cmd.len;
694
695     s->smtp_from.data = ngx_pnalloc(c->pool, cmd.len);
696     if (s->smtp_from.data == NULL) {
697         return NGX_ERROR;
698     }
699
700     ngx_memcpy(s->smtp_from.data, cmd.data, cmd.len);
701
702     ngx_log_debug1(NGX_LOG_DEBUG_MAIL, c->log, 0,
703                  "smtp mail from: \"%V\"", &s->smtp_from);
704
705     ngx_str_set(&s->out, smtp_ok);
706
707     return NGX_OK;
708 }
709
710
711 static ngx_int_t
712 ngx_mail_smtp_rcpt(ngx_mail_session_t *s, ngx_connection_t *c)
713 {
714     ngx_str_t *arg, cmd;
715
716     if (s->smtp_from.len == 0) {
717         ngx_str_set(&s->out, smtp_bad_sequence);
718         return NGX_OK;
719     }
720
721     if (s->args.nelts == 0) {
722         ngx_str_set(&s->out, smtp_invalid_argument);
723         return NGX_OK;
724     }
725
726     arg = s->args.elts;
727     arg += s->args.nelts - 1;
728
729     cmd.len = arg->data + arg->len - s->cmd.data;
730     cmd.data = s->cmd.data;
731
732     s->smtp_to.len = cmd.len;
733
734     s->smtp_to.data = ngx_pnalloc(c->pool, cmd.len);
735     if (s->smtp_to.data == NULL) {
736         return NGX_ERROR;
737     }
738
739     ngx_memcpy(s->smtp_to.data, cmd.data, cmd.len);
740
741     ngx_log_debug1(NGX_LOG_DEBUG_MAIL, c->log, 0,
742                  "smtp rcpt to: \"%V\"", &s->smtp_to);
743
744     s->auth_method = NGX_MAIL_AUTH_NONE;
745
746     return NGX_DONE;
747 }
748
749
750 static ngx_int_t
751 ngx_mail_smtp_rset(ngx_mail_session_t *s, ngx_connection_t *c)
752 {

```

```

753     ngx_str_null(&s->smtp_from);
754     ngx_str_null(&s->smtp_to);
755     ngx_str_set(&s->out, smtp_ok);
756
757     return NGX_OK;
758 }
759
760
761 static ngx_int_t
762 ngx_mail_smtp_starttls(ngx_mail_session_t *s, ngx_connection_t *c)
763 {
764     #if (NGX_MAIL_SSL)
765         ngx_mail_ssl_conf_t *sslcf;
766
767         if (c->ssl == NULL) {
768             sslcf = ngx_mail_get_module_srv_conf(s, ngx_mail_ssl_module);
769             if (sslcf->starttls) {
770
771                 /*
772                  * RFC3207 requires us to discard any knowledge
773                  * obtained from client before STARTTLS.
774                  */
775
776                 ngx_str_null(&s->smtp_helo);
777                 ngx_str_null(&s->smtp_from);
778                 ngx_str_null(&s->smtp_to);
779
780                 s->buffer->pos = s->buffer->start;
781                 s->buffer->last = s->buffer->start;
782
783                 c->read->handler = ngx_mail_starttls_handler;
784                 return NGX_OK;
785             }
786         }
787     #endif
788
789     return NGX_MAIL_PARSE_INVALID_COMMAND;
790 }
791
792
793
794 static ngx_int_t
795 ngx_mail_smtp_discard_command(ngx_mail_session_t *s, ngx_connection_t *c,
796     char *err)
797 {
798     ssize_t    n;
799
800     n = c->recv(c, s->buffer->last, s->buffer->end - s->buffer->last);
801
802     if (n == NGX_ERROR || n == 0) {
803         ngx_mail_close_connection(c);
804         return NGX_ERROR;
805     }
806
807     if (n > 0) {
808         s->buffer->last += n;
809     }
810
811     if (n == NGX_AGAIN) {
812         if (ngx_handle_read_event(c->read, 0) != NGX_OK) {
813             ngx_mail_session_internal_server_error(s);
814             return NGX_ERROR;
815         }
816
817         return NGX_AGAIN;
818     }
819
820     ngx_mail_smtp_log_rejected_command(s, c, err);
821
822     s->buffer->pos = s->buffer->start;
823     s->buffer->last = s->buffer->start;
824
825     return NGX_OK;
826 }
827
828

```

```
829 static void
830 ngx_mail_smtp_log_rejected_command(ngx_mail_session_t *s, ngx_connection_t *c,
831 char *err)
832 {
833     u_char    ch;
834     ngx_str_t cmd;
835     ngx_uint_t i;
836
837     if (c->log->log_level < NGX_LOG_INFO) {
838         return;
839     }
840
841     cmd.len = s->buffer->last - s->buffer->start;
842     cmd.data = s->buffer->start;
843
844     for (i = 0; i < cmd.len; i++) {
845         ch = cmd.data[i];
846
847         if (ch != CR && ch != LF) {
848             continue;
849         }
850
851         cmd.data[i] = '_';
852     }
853
854     cmd.len = i;
855
856     ngx_log_error(NGX_LOG_INFO, c->log, 0, err, &cmd);
857 }
```

[One Level Up](#)

[Top Level](#)

# src/mail/nginx\_mail\_smtp\_module.h - nginx-1.7.10

## Data types defined

- [ngx\\_mail\\_smtp\\_srv\\_conf\\_t](#)

## Macros defined

- [\\_NGX\\_MAIL\\_SMTP\\_MODULE\\_H\\_INCLUDED](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_MAIL_SMTP_MODULE_H_INCLUDED
9 #define _NGX_MAIL_SMTP_MODULE_H_INCLUDED
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14 #include <ngx_mail.h>
15 #include <ngx_mail_smtp_module.h>
16
17
18 typedef struct {
19     ngx_msec_t    greeting_delay;
20
21     size_t        client_buffer_size;
22
23     ngx_str_t     capability;
24     ngx_str_t     starttls_capability;
25     ngx_str_t     starttls_only_capability;
26
27     ngx_str_t     server_name;
28     ngx_str_t     greeting;
29
30     ngx_uint_t    auth_methods;
31
32     ngx_array_t   capabilities;
33 } ngx_mail_smtp_srv_conf_t;
34
35
36 void ngx_mail_smtp_init_session(ngx_mail_session_t *s, ngx_connection_t *c);
37 void ngx_mail_smtp_init_protocol(ngx_event_t *rev);
38 void ngx_mail_smtp_auth_state(ngx_event_t *rev);
39 ngx_int_t ngx_mail_smtp_parse_command(ngx_mail_session_t *s);
40
41
42 extern ngx_module_t ngx_mail_smtp_module;
43
44
45 #endif /* _NGX_MAIL_SMTP_MODULE_H_INCLUDED */
```

## src/event/nginx\_event\_accept.c - nginx-1.7.10

### Functions defined

- [ngx\\_accept\\_log\\_error](#)
- [ngx\\_close\\_accepted\\_connection](#)
- [ngx\\_disable\\_accept\\_events](#)
- [ngx\\_enable\\_accept\\_events](#)
- [ngx\\_event\\_accept](#)
- [ngx\\_trylock\\_accept\\_mutex](#)

### Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 static ngx_int_t ngx_enable_accept_events(ngx_cycle_t *cycle);
14 static ngx_int_t ngx_disable_accept_events(ngx_cycle_t *cycle);
15 static void ngx_close_accepted_connection(ngx_connection_t *c);
16
17
18 void
19 ngx_event_accept(ngx_event_t *ev)
20 {
21     socklen_t      socklen;
22     ngx_err_t      err;
23     ngx_log_t      *log;
24     ngx_uint_t     level;
25     ngx_socket_t   s;
26     ngx_event_t    *rev, *wev;
27     ngx_listening_t *ls;
28     ngx_connection_t *c, *lc;
29     ngx_event_conf_t *ecf;
30     u_char         sa[NGX_SOCKADDRLEN];
31     #if (NGX_HAVE_ACCEPT4)
32     static ngx_uint_t use_accept4 = 1;
33     #endif
34
35     if (ev->timedout) {
36         if (ngx_enable_accept_events((ngx_cycle_t *) ngx_cycle) != NGX_OK) {
37             return;
38         }
39
40         ev->timedout = 0;
41     }
42
43     ecf = ngx_event_get_conf(ngx_cycle->conf_ctx, ngx_event_core_module);
44
45     if (ngx_event_flags & NGX_USE_RTSIG_EVENT) {
46         ev->available = 1;
47
48     } else if (!(ngx_event_flags & NGX_USE_KQUEUE_EVENT)) {
49         ev->available = ecf->multi_accept;
50     }
51 }
```



```

52     lc = ev->data;
53     ls = lc->listening;
54     ev->ready = 0;
55
56     ngx_log_debug2(NGX_LOG_DEBUG_EVENT, ev->log, 0,
57                  "accept on %V, ready: %d", &ls->addr_text, ev->available);
58
59     do {
60         socklen = NGX_SOCKADDRLEN;
61
62         #if (NGX_HAVE_ACCEPT4)
63             if (use_accept4) {
64                 s = accept4(lc->fd, (struct sockaddr *) sa, &socklen,
65                             SOCK_NONBLOCK);
66             } else {
67                 s = accept(lc->fd, (struct sockaddr *) sa, &socklen);
68             }
69         #else
70             s = accept(lc->fd, (struct sockaddr *) sa, &socklen);
71         #endif
72
73         if (s == (ngx_socket_t) -1) {
74             err = ngx_socket_errno;
75
76             if (err == NGX_EAGAIN) {
77                 ngx_log_debug0(NGX_LOG_DEBUG_EVENT, ev->log, err,
78                               "accept() not ready");
79                 return;
80             }
81
82             level = NGX_LOG_ALERT;
83
84             if (err == NGX_ECONNABORTED) {
85                 level = NGX_LOG_ERR;
86
87             } else if (err == NGX_EMFILE || err == NGX_ENFILE) {
88                 level = NGX_LOG_CRIT;
89             }
90
91             #if (NGX_HAVE_ACCEPT4)
92                 ngx_log_error(level, ev->log, err,
93                               use_accept4 ? "accept4() failed" : "accept() failed");
94
95             if (use_accept4 && err == NGX_ENOSYS) {
96                 use_accept4 = 0;
97                 ngx_inherited_nonblocking = 0;
98                 continue;
99             }
100         #else
101             ngx_log_error(level, ev->log, err, "accept() failed");
102         #endif
103
104         if (err == NGX_ECONNABORTED) {
105             if (ngx_event_flags & NGX_USE_KQUEUE_EVENT) {
106                 ev->available--;
107             }
108
109             if (ev->available) {
110                 continue;
111             }
112         }
113
114         if (err == NGX_EMFILE || err == NGX_ENFILE) {
115             if (ngx_disable_accept_events((ngx_cycle_t *) ngx_cycle)
116                 != NGX_OK)
117             {
118                 return;
119             }
120
121             if (ngx_use_accept_mutex) {
122                 if (ngx_accept_mutex_held) {
123                     ngx_shmtx_unlock(&ngx_accept_mutex);
124                     ngx_accept_mutex_held = 0;
125                 }
126
127                 ngx_accept_disabled = 1;

```

```

128         } else {
129             ngx\_add\_timer(ev, ecf->accept_mutex_delay);
130         }
131     }
132 }
133
134     return;
135 }
136
137 #if (NGX_STAT_STUB)
138     (void) ngx\_atomic\_fetch\_add(ngx\_stat\_accepted, 1);
139 #endif
140
141     ngx\_accept\_disabled = ngx\_cycle->connection_n / 8
142         - ngx\_cycle->free_connection_n;
143
144     c = ngx\_get\_connection(s, ev->log);
145
146     if (c == NULL) {
147         if (ngx\_close\_socket(s) == -1) {
148             ngx\_log\_error(NGX_LOG_ALERT, ev->log, ngx\_socket\_errno,
149                 ngx\_close\_socket\_n " failed");
150         }
151     }
152     return;
153 }
154
155 #if (NGX_STAT_STUB)
156     (void) ngx\_atomic\_fetch\_add(ngx\_stat\_active, 1);
157 #endif
158
159     c->pool = ngx\_create\_pool(ls->pool_size, ev->log);
160     if (c->pool == NULL) {
161         ngx\_close\_accepted\_connection(c);
162         return;
163     }
164
165     c->sockaddr = ngx\_palloc(c->pool, socklen);
166     if (c->sockaddr == NULL) {
167         ngx\_close\_accepted\_connection(c);
168         return;
169     }
170
171     ngx\_memcpy(c->sockaddr, sa, socklen);
172
173     log = ngx\_palloc(c->pool, sizeof(ngx\_log\_t));
174     if (log == NULL) {
175         ngx\_close\_accepted\_connection(c);
176         return;
177     }
178
179     /* set a blocking mode for aio and non-blocking mode for others */
180
181     if (ngx\_inherited\_nonblocking) {
182         if (ngx\_event\_flags & NGX_USE_AIO_EVENT) {
183             if (ngx\_blocking(s) == -1) {
184                 ngx\_log\_error(NGX_LOG_ALERT, ev->log, ngx\_socket\_errno,
185                     ngx\_blocking\_n " failed");
186                 ngx\_close\_accepted\_connection(c);
187                 return;
188             }
189         }
190     }
191     } else {
192         if (!(ngx\_event\_flags & (NGX_USE_AIO_EVENT|NGX_USE_RTSIG_EVENT))) {
193             if (ngx\_nonblocking(s) == -1) {
194                 ngx\_log\_error(NGX_LOG_ALERT, ev->log, ngx\_socket\_errno,
195                     ngx\_nonblocking\_n " failed");
196                 ngx\_close\_accepted\_connection(c);
197                 return;
198             }
199         }
200     }
201
202     *log = ls->log;
203

```

```

204     c->recv = ngx_recv;
205     c->send = ngx_send;
206     c->recv_chain = ngx_recv_chain;
207     c->send_chain = ngx_send_chain;
208
209     c->log = log;
210     c->pool->log = log;
211
212     c->socklen = socklen;
213     c->listening = ls;
214     c->local_sockaddr = ls->sockaddr;
215     c->local_socklen = ls->socklen;
216
217     c->unexpected_eof = 1;
218
219     #if (NGX_HAVE_UNIX_DOMAIN)
220         if (c->sockaddr->sa_family == AF_UNIX) {
221             c->tcp_nopush = NGX_TCP_NOPUSH_DISABLED;
222             c->tcp_nodelay = NGX_TCP_NODELAY_DISABLED;
223         #if (NGX_SOLARIS)
224             /* Solaris's sendfilev() supports AF_NCA, AF_INET, and AF_INET6 */
225             c->sendfile = 0;
226         #endif
227     }
228 #endif
229
230     rev = c->read;
231     wev = c->write;
232
233     wev->ready = 1;
234
235     if (ngx_event_flags & (NGX_USE_AIO_EVENT|NGX_USE_RTSIG_EVENT)) {
236         /* rtsig, aio, iocp */
237         rev->ready = 1;
238     }
239
240     if (ev->deferred_accept) {
241         rev->ready = 1;
242     #if (NGX_HAVE_KQUEUE)
243         rev->available = 1;
244     #endif
245     }
246
247     rev->log = log;
248     wev->log = log;
249
250     /*
251     * TODO: MT: - ngx_atomic_fetch_add()
252     *             or protection by critical section or light mutex
253     *
254     * TODO: MP: - allocated in a shared memory
255     *             - ngx_atomic_fetch_add()
256     *             or protection by critical section or light mutex
257     */
258
259     c->number = ngx_atomic_fetch_add(ngx_connection_counter, 1);
260
261     #if (NGX_STAT_STUB)
262     (void) ngx_atomic_fetch_add(ngx_stat_handled, 1);
263     #endif
264
265     if (ls->addr_ntop) {
266         c->addr_text.data = ngx_pnalloc(c->pool, ls->addr_text_max_len);
267         if (c->addr_text.data == NULL) {
268             ngx_close_accepted_connection(c);
269             return;
270         }
271
272         c->addr_text.len = ngx_sock_ntop(c->sockaddr, c->socklen,
273             c->addr_text.data,
274             ls->addr_text_max_len, 0);
275
276         if (c->addr_text.len == 0) {
277             ngx_close_accepted_connection(c);
278             return;
279         }
280     }

```

```

280
281 #if (NGX_DEBUG)
282     {
283
284         ngx_str_t          addr;
285         struct sockaddr_in *sin;
286         ngx_cidr_t         *cidr;
287         ngx_uint_t         i;
288         u_char             text[NGX_SOCKADDR_STRLEN];
289 #if (NGX_HAVE_INET6)
290         struct sockaddr_in6 *sin6;
291         ngx_uint_t         n;
292 #endif
293
294         cidr = ecf->debug_connection.elts;
295         for (i = 0; i < ecf->debug_connection.nelts; i++) {
296             if (cidr[i].family != (ngx_uint_t) c->sockaddr->sa_family) {
297                 goto next;
298             }
299
300             switch (cidr[i].family) {
301
302 #if (NGX_HAVE_INET6)
303                 case AF_INET6:
304                     sin6 = (struct sockaddr_in6 *) c->sockaddr;
305                     for (n = 0; n < 16; n++) {
306                         if ((sin6->sin6_addr.s6_addr[n]
307                             & cidr[i].u.in6.mask.s6_addr[n])
308                             != cidr[i].u.in6.addr.s6_addr[n])
309                             {
310                                 goto next;
311                             }
312                         }
313                     break;
314 #endif
315
316 #if (NGX_HAVE_UNIX_DOMAIN)
317                 case AF_UNIX:
318                     break;
319 #endif
320
321                 default: /* AF_INET */
322                     sin = (struct sockaddr_in *) c->sockaddr;
323                     if ((sin->sin_addr.s_addr & cidr[i].u.in.mask)
324                         != cidr[i].u.in.addr)
325                         {
326                             goto next;
327                         }
328                     break;
329             }
330
331             log->log_level = NGX_LOG_DEBUG_CONNECTION|NGX_LOG_DEBUG_ALL;
332             break;
333
334         next:
335             continue;
336     }
337
338     if (log->log_level & NGX_LOG_DEBUG_EVENT) {
339         addr.data = text;
340         addr.len = ngx_sock_ntop(c->sockaddr, c->socklen, text,
341                                 NGX_SOCKADDR_STRLEN, 1);
342
343         ngx_log_debug3(NGX_LOG_DEBUG_EVENT, log, 0,
344                      "%uA accept: %V fd:%d", c->number, &addr, s);
345     }
346 }
347 #endif
348
349
350 if (ngx_add_conn && (ngx_event_flags & NGX_USE_EPOLL_EVENT) == 0) {
351     if (ngx_add_conn(c) == NGX_ERROR) {
352         ngx_close_accepted_connection(c);
353         return;
354     }
355 }

```

```

356     log->data = NULL;
357     log->handler = NULL;
358
359     ls->handler(c);
360
361     if (ngx_event_flags & NGX_USE_KQUEUE_EVENT) {
362         ev->available--;
363     }
364
365 } while (ev->available);
366 }
367
368
369
370 ngx_int_t
371 ngx_trylock_accept_mutex(ngx_cycle_t *cycle)
372 {
373     if (ngx_shmtx_trylock(&ngx_accept_mutex)) {
374
375         ngx_log_debug0(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
376             "accept mutex locked");
377
378         if (ngx_accept_mutex_held
379             && ngx_accept_events == 0
380             && !(ngx_event_flags & NGX_USE_RTSG_EVENT))
381         {
382             return NGX_OK;
383         }
384
385         if (ngx_enable_accept_events(cycle) == NGX_ERROR) {
386             ngx_shmtx_unlock(&ngx_accept_mutex);
387             return NGX_ERROR;
388         }
389
390         ngx_accept_events = 0;
391         ngx_accept_mutex_held = 1;
392
393         return NGX_OK;
394     }
395
396     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
397         "accept mutex lock failed: %ui", ngx_accept_mutex_held);
398
399     if (ngx_accept_mutex_held) {
400         if (ngx_disable_accept_events(cycle) == NGX_ERROR) {
401             return NGX_ERROR;
402         }
403
404         ngx_accept_mutex_held = 0;
405     }
406
407     return NGX_OK;
408 }
409
410
411 static ngx_int_t
412 ngx_enable_accept_events(ngx_cycle_t *cycle)
413 {
414     ngx_uint_t     i;
415     ngx_listening_t *ls;
416     ngx_connection_t *c;
417
418     ls = cycle->listening.elts;
419     for (i = 0; i < cycle->listening.nelts; i++) {
420
421         c = ls[i].connection;
422
423         if (c->read->active) {
424             continue;
425         }
426
427         if (ngx_event_flags & NGX_USE_RTSG_EVENT) {
428
429             if (ngx_add_conn(c) == NGX_ERROR) {
430                 return NGX_ERROR;
431             }

```

```

432     } else {
433         if (ngx_add_event(c->read, NGX_READ_EVENT, 0) == NGX_ERROR) {
434             return NGX_ERROR;
435         }
436     }
437 }
438 }
439
440 return NGX_OK;
441 }
442
443
444 static ngx_int_t
445 ngx_disable_accept_events(ngx_cycle_t *cycle)
446 {
447     ngx_uint_t i;
448     ngx_listening_t *ls;
449     ngx_connection_t *c;
450
451     ls = cycle->listening.elts;
452     for (i = 0; i < cycle->listening.nelts; i++) {
453
454         c = ls[i].connection;
455
456         if (!c->read->active) {
457             continue;
458         }
459
460         if (ngx_event_flags & NGX_USE_RTSIG_EVENT) {
461             if (ngx_del_conn(c, NGX_DISABLE_EVENT) == NGX_ERROR) {
462                 return NGX_ERROR;
463             }
464
465         } else {
466             if (ngx_del_event(c->read, NGX_READ_EVENT, NGX_DISABLE_EVENT)
467                 == NGX_ERROR)
468             {
469                 return NGX_ERROR;
470             }
471         }
472     }
473
474     return NGX_OK;
475 }
476
477
478 static void
479 ngx_close_accepted_connection(ngx_connection_t *c)
480 {
481     ngx_socket_t fd;
482
483     ngx_free_connection(c);
484
485     fd = c->fd;
486     c->fd = (ngx_socket_t) -1;
487
488     if (ngx_close_socket(fd) == -1) {
489         ngx_log_error(NGX_LOG_ALERT, c->log, ngx_socket_errno,
490             ngx_close_socket_n " failed");
491     }
492
493     if (c->pool) {
494         ngx_destroy_pool(c->pool);
495     }
496
497     #if (NGX_STAT_STUB)
498     (void) ngx_atomic_fetch_add(ngx_stat_active, -1);
499     #endif
500 }
501
502
503 u_char *
504 ngx_accept_log_error(ngx_log_t *log, u_char *buf, size_t len)
505 {
506     return ngx_snprintf(buf, len, " while accepting new connection on %V",
507         log->data);

```

[One Level Up](#)

[Top Level](#)

## src/http/nginx\_http\_core\_module.c - nginx-1.7.10

### Global variables defined

- [ngx\\_conf\\_deprecated\\_open\\_file\\_cache\\_retest](#)
- [ngx\\_conf\\_deprecated\\_optimize\\_server\\_names](#)
- [ngx\\_conf\\_deprecated\\_satisfy\\_any](#)
- [ngx\\_http\\_client\\_temp\\_path](#)
- [ngx\\_http\\_core\\_aio](#)
- [ngx\\_http\\_core\\_commands](#)
- [ngx\\_http\\_core\\_default\\_types](#)
- [ngx\\_http\\_core\\_get\\_method](#)
- [ngx\\_http\\_core\\_if\\_modified\\_since](#)
- [ngx\\_http\\_core\\_image\\_gif\\_type](#)
- [ngx\\_http\\_core\\_image\\_jpeg\\_type](#)
- [ngx\\_http\\_core\\_keepalive\\_disable](#)
- [ngx\\_http\\_core\\_lingering\\_close](#)
- [ngx\\_http\\_core\\_lowat\\_post](#)
- [ngx\\_http\\_core\\_module](#)
- [ngx\\_http\\_core\\_module\\_ctx](#)
- [ngx\\_http\\_core\\_pool\\_size\\_p](#)
- [ngx\\_http\\_core\\_request\\_body\\_in\\_file](#)
- [ngx\\_http\\_core\\_satisfy](#)
- [ngx\\_http\\_core\\_text\\_html\\_type](#)
- [ngx\\_http\\_gzip\\_http\\_version](#)
- [ngx\\_http\\_gzip\\_no\\_cache](#)
- [ngx\\_http\\_gzip\\_no\\_store](#)
- [ngx\\_http\\_gzip\\_private](#)
- [ngx\\_http\\_gzip\\_proxied\\_mask](#)
- [ngx\\_methods\\_names](#)

### Data types defined

- [ngx\\_http\\_method\\_name\\_t](#)

### Functions defined



- [ngx http auth basic user](#)
- [ngx http cleanup add](#)
- [ngx http core access phase](#)
- [ngx http core content phase](#)
- [ngx http core create loc conf](#)
- [ngx http core create main conf](#)
- [ngx http core create srv conf](#)
- [ngx http core directio](#)
- [ngx http core error log](#)
- [ngx http core error page](#)
- [ngx http core find config phase](#)
- [ngx http core find location](#)
- [ngx http core find static location](#)
- [ngx http core generic phase](#)
- [ngx http core init main conf](#)
- [ngx http core internal](#)
- [ngx http core keepalive](#)
- [ngx http core limit except](#)
- [ngx http core listen](#)
- [ngx http core location](#)
- [ngx http core lowat check](#)
- [ngx http core merge loc conf](#)
- [ngx http core merge srv conf](#)
- [ngx http core open file cache](#)
- [ngx http core pool size](#)
- [ngx http core post access phase](#)
- [ngx http core post rewrite phase](#)
- [ngx http core preconfiguration](#)
- [ngx http core regex location](#)
- [ngx http core resolver](#)
- [ngx http core rewrite phase](#)
- [ngx http core root](#)
- [ngx http core run phases](#)

- [ngx http core server](#)
- [ngx http core server name](#)
- [ngx http core try files](#)
- [ngx http core try files phase](#)
- [ngx http core type](#)
- [ngx http core types](#)
- [ngx http disable symlinks](#)
- [ngx http get forwarded addr](#)
- [ngx http get forwarded addr internal](#)
- [ngx http gzip accept encoding](#)
- [ngx http gzip disable](#)
- [ngx http gzip ok](#)
- [ngx http gzip quantity](#)
- [ngx http handler](#)
- [ngx http internal redirect](#)
- [ngx http map uri to path](#)
- [ngx http named location](#)
- [ngx http output filter](#)
- [ngx http send header](#)
- [ngx http send response](#)
- [ngx http set content type](#)
- [ngx http set disable symlinks](#)
- [ngx http set etag](#)
- [ngx http set exten](#)
- [ngx http subrequest](#)
- [ngx http test content type](#)
- [ngx http update location config](#)
- [ngx http weak etag](#)

## Macros defined

- [NGX\\_HTTP\\_REQUEST\\_BODY\\_FILE\\_CLEAN](#)
- [NGX\\_HTTP\\_REQUEST\\_BODY\\_FILE\\_OFF](#)
- [NGX\\_HTTP\\_REQUEST\\_BODY\\_FILE\\_ON](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 typedef struct {
14     u_char    *name;
15     uint32_t  method;
16 } ngx_http_method_name_t;
17
18
19 #define NGX_HTTP_REQUEST_BODY_FILE_OFF    0
20 #define NGX_HTTP_REQUEST_BODY_FILE_ON    1
21 #define NGX_HTTP_REQUEST_BODY_FILE_CLEAN  2
22
23
24 static ngx_int_t ngx_http_core_find_location(ngx_http_request_t *r);
25 static ngx_int_t ngx_http_core_find_static_location(ngx_http_request_t *r,
26     ngx_http_location_tree_node_t *node);
27
28 static ngx_int_t ngx_http_core_preconfiguration(ngx_conf_t *cf);
29 static void *ngx_http_core_create_main_conf(ngx_conf_t *cf);
30 static char *ngx_http_core_init_main_conf(ngx_conf_t *cf, void *conf);
31 static void *ngx_http_core_create_srv_conf(ngx_conf_t *cf);
32 static char *ngx_http_core_merge_srv_conf(ngx_conf_t *cf,
33     void *parent, void *child);
34 static void *ngx_http_core_create_loc_conf(ngx_conf_t *cf);
35 static char *ngx_http_core_merge_loc_conf(ngx_conf_t *cf,
36     void *parent, void *child);
37
38 static char *ngx_http_core_server(ngx_conf_t *cf, ngx_command_t *cmd,
39     void *dummy);
40 static char *ngx_http_core_location(ngx_conf_t *cf, ngx_command_t *cmd,
41     void *dummy);
42 static ngx_int_t ngx_http_core_regex_location(ngx_conf_t *cf,
43     ngx_http_core_loc_conf_t *clcf, ngx_str_t *regex, ngx_uint_t caseless);
44
45 static char *ngx_http_core_types(ngx_conf_t *cf, ngx_command_t *cmd,
46     void *conf);
47 static char *ngx_http_core_type(ngx_conf_t *cf, ngx_command_t *dummy,
48     void *conf);
49
50 static char *ngx_http_core_listen(ngx_conf_t *cf, ngx_command_t *cmd,
51     void *conf);
52 static char *ngx_http_core_server_name(ngx_conf_t *cf, ngx_command_t *cmd,
53     void *conf);
54 static char *ngx_http_core_root(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
55 static char *ngx_http_core_limit_except(ngx_conf_t *cf, ngx_command_t *cmd,
56     void *conf);
57 static char *ngx_http_core_directio(ngx_conf_t *cf, ngx_command_t *cmd,
58     void *conf);
59 static char *ngx_http_core_error_page(ngx_conf_t *cf, ngx_command_t *cmd,
60     void *conf);
61 static char *ngx_http_core_try_files(ngx_conf_t *cf, ngx_command_t *cmd,
62     void *conf);
63 static char *ngx_http_core_open_file_cache(ngx_conf_t *cf, ngx_command_t *cmd,
64     void *conf);
65 static char *ngx_http_core_error_log(ngx_conf_t *cf, ngx_command_t *cmd,
66     void *conf);
67 static char *ngx_http_core_keepalive(ngx_conf_t *cf, ngx_command_t *cmd,
68     void *conf);
69 static char *ngx_http_core_internal(ngx_conf_t *cf, ngx_command_t *cmd,
70     void *conf);
71 static char *ngx_http_core_resolver(ngx_conf_t *cf, ngx_command_t *cmd,
72     void *conf);
73 #if (NGX_HTTP_GZIP)
```

```

74 static ngx_int_t ngx_http_gzip_accept_encoding(ngx_str_t *ae);
75 static ngx_uint_t ngx_http_gzip_quantity(u_char *p, u_char *last);
76 static char *ngx_http_gzip_disable(ngx_conf_t *cf, ngx_command_t *cmd,
77     void *conf);
78 #endif
79 static ngx_int_t ngx_http_get_forwarded_addr_internal(ngx_http_request_t *r,
80     ngx_addr_t *addr, u_char *xff, size_t xfflen, ngx_array_t *proxies,
81     int recursive);
82 #if (NGX_HAVE_OPENAT)
83 static char *ngx_http_disable_symlinks(ngx_conf_t *cf, ngx_command_t *cmd,
84     void *conf);
85 #endif
86
87 static char *ngx_http_core_lowat_check(ngx_conf_t *cf, void *post, void *data);
88 static char *ngx_http_core_pool_size(ngx_conf_t *cf, void *post, void *data);
89
90 static ngx_conf_post_t  ngx_http_core_lowat_post =
91     { ngx_http_core_lowat_check };
92
93 static ngx_conf_post_handler_pt  ngx_http_core_pool_size_p =
94     ngx_http_core_pool_size;
95
96 static ngx_conf_deprecated_t  ngx_conf_deprecated_optimize_server_names = {
97     ngx_conf_deprecated, "optimize_server_names", "server_name_in_redirect"
98 };
99
100 static ngx_conf_deprecated_t  ngx_conf_deprecated_open_file_cache_retest = {
101     ngx_conf_deprecated, "open_file_cache_retest", "open_file_cache_valid"
102 };
103
104 static ngx_conf_deprecated_t  ngx_conf_deprecated_satisfy_any = {
105     ngx_conf_deprecated, "satisfy_any", "satisfy"
106 };
107
108
109 static ngx_conf_enum_t  ngx_http_core_request_body_in_file[] = {
110     { ngx_string("off"),  NGX_HTTP_REQUEST_BODY_FILE_OFF },
111     { ngx_string("on"),  NGX_HTTP_REQUEST_BODY_FILE_ON },
112     { ngx_string("clean"),  NGX_HTTP_REQUEST_BODY_FILE_CLEAN },
113     { ngx_null_string, 0 }
114 };
115
116
117 #if (NGX_HAVE_FILE_AIO)
118
119 static ngx_conf_enum_t  ngx_http_core_aio[] = {
120     { ngx_string("off"),  NGX_HTTP_AIO_OFF },
121     { ngx_string("on"),  NGX_HTTP_AIO_ON },
122     #if (NGX_HAVE_AIO_SENDFILE)
123     { ngx_string("sendfile"),  NGX_HTTP_AIO_SENDFILE },
124     #endif
125     { ngx_null_string, 0 }
126 };
127
128 #endif
129
130
131 static ngx_conf_enum_t  ngx_http_core_satisfy[] = {
132     { ngx_string("all"),  NGX_HTTP_SATISFY_ALL },
133     { ngx_string("any"),  NGX_HTTP_SATISFY_ANY },
134     { ngx_null_string, 0 }
135 };
136
137
138 static ngx_conf_enum_t  ngx_http_core_lingering_close[] = {
139     { ngx_string("off"),  NGX_HTTP_LINGERING_OFF },
140     { ngx_string("on"),  NGX_HTTP_LINGERING_ON },
141     { ngx_string("always"),  NGX_HTTP_LINGERING_ALWAYS },
142     { ngx_null_string, 0 }
143 };
144
145
146 static ngx_conf_enum_t  ngx_http_core_if_modified_since[] = {
147     { ngx_string("off"),  NGX_HTTP_IMS_OFF },
148     { ngx_string("exact"),  NGX_HTTP_IMS_EXACT },
149     { ngx_string("before"),  NGX_HTTP_IMS_BEFORE },

```

```

150     { ngx_null_string, 0 }
151 };
152
153
154 static ngx_conf_bitmask_t  ngx_http_core_keepalive_disable[] = {
155     { ngx_string("none"),  NGX_HTTP_KEEPALIVE_DISABLE_NONE },
156     { ngx_string("msie6"),  NGX_HTTP_KEEPALIVE_DISABLE_MSIE6 },
157     { ngx_string("safari"),  NGX_HTTP_KEEPALIVE_DISABLE_SAFARI },
158     { ngx_null_string, 0 }
159 };
160
161
162 static ngx_path_init_t  ngx_http_client_temp_path = {
163     ngx_string(NGX_HTTP_CLIENT_TEMP_PATH), { 0, 0, 0 }
164 };
165
166
167 #if (NGX_HTTP_GZIP)
168
169 static ngx_conf_enum_t  ngx_http_gzip_http_version[] = {
170     { ngx_string("1.0"),  NGX_HTTP_VERSION_10 },
171     { ngx_string("1.1"),  NGX_HTTP_VERSION_11 },
172     { ngx_null_string, 0 }
173 };
174
175
176 static ngx_conf_bitmask_t  ngx_http_gzip_proxied_mask[] = {
177     { ngx_string("off"),  NGX_HTTP_GZIP_PROXIED_OFF },
178     { ngx_string("expired"),  NGX_HTTP_GZIP_PROXIED_EXPIRED },
179     { ngx_string("no-cache"),  NGX_HTTP_GZIP_PROXIED_NO_CACHE },
180     { ngx_string("no-store"),  NGX_HTTP_GZIP_PROXIED_NO_STORE },
181     { ngx_string("private"),  NGX_HTTP_GZIP_PROXIED_PRIVATE },
182     { ngx_string("no_last_modified"),  NGX_HTTP_GZIP_PROXIED_NO_LM },
183     { ngx_string("no_etag"),  NGX_HTTP_GZIP_PROXIED_NO_ETAG },
184     { ngx_string("auth"),  NGX_HTTP_GZIP_PROXIED_AUTH },
185     { ngx_string("any"),  NGX_HTTP_GZIP_PROXIED_ANY },
186     { ngx_null_string, 0 }
187 };
188
189
190 static ngx_str_t  ngx_http_gzip_no_cache = ngx_string("no-cache");
191 static ngx_str_t  ngx_http_gzip_no_store = ngx_string("no-store");
192 static ngx_str_t  ngx_http_gzip_private = ngx_string("private");
193
194 #endif
195
196
197 static ngx_command_t  ngx_http_core_commands[] = {
198
199     { ngx_string("variables_hash_max_size"),
200       NGX_HTTP_MAIN_CONF|NGX_CONF_TAKE1,
201       ngx_conf_set_num_slot,
202       NGX_HTTP_MAIN_CONF_OFFSET,
203       offsetof(ngx_http_core_main_conf_t, variables_hash_max_size),
204       NULL },
205
206     { ngx_string("variables_hash_bucket_size"),
207       NGX_HTTP_MAIN_CONF|NGX_CONF_TAKE1,
208       ngx_conf_set_num_slot,
209       NGX_HTTP_MAIN_CONF_OFFSET,
210       offsetof(ngx_http_core_main_conf_t, variables_hash_bucket_size),
211       NULL },
212
213     { ngx_string("server_names_hash_max_size"),
214       NGX_HTTP_MAIN_CONF|NGX_CONF_TAKE1,
215       ngx_conf_set_num_slot,
216       NGX_HTTP_MAIN_CONF_OFFSET,
217       offsetof(ngx_http_core_main_conf_t, server_names_hash_max_size),
218       NULL },
219
220     { ngx_string("server_names_hash_bucket_size"),
221       NGX_HTTP_MAIN_CONF|NGX_CONF_TAKE1,
222       ngx_conf_set_num_slot,
223       NGX_HTTP_MAIN_CONF_OFFSET,
224       offsetof(ngx_http_core_main_conf_t, server_names_hash_bucket_size),
225       NULL },

```

```

226 { ngx_string("server"),
227     NGX\_HTTP\_MAIN\_CONF | NGX\_CONF\_BLOCK | NGX\_CONF\_NOARGS,
228     ngx_http_core_server,
229     0,
230     0,
231     NULL },
232
233
234 { ngx_string("connection_pool_size"),
235     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_CONF TAKE1,
236     ngx_conf_set_size_slot,
237     NGX\_HTTP\_SRV\_CONF\_OFFSET,
238     offsetof(ngx_http_core_srv_conf_t, connection_pool_size),
239     &ngx_http_core_pool_size_p },
240
241 { ngx_string("request_pool_size"),
242     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_CONF TAKE1,
243     ngx_conf_set_size_slot,
244     NGX\_HTTP\_SRV\_CONF\_OFFSET,
245     offsetof(ngx_http_core_srv_conf_t, request_pool_size),
246     &ngx_http_core_pool_size_p },
247
248 { ngx_string("client_header_timeout"),
249     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_CONF TAKE1,
250     ngx_conf_set_msec_slot,
251     NGX\_HTTP\_SRV\_CONF\_OFFSET,
252     offsetof(ngx_http_core_srv_conf_t, client_header_timeout),
253     NULL },
254
255 { ngx_string("client_header_buffer_size"),
256     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_CONF TAKE1,
257     ngx_conf_set_size_slot,
258     NGX\_HTTP\_SRV\_CONF\_OFFSET,
259     offsetof(ngx_http_core_srv_conf_t, client_header_buffer_size),
260     NULL },
261
262 { ngx_string("large_client_header_buffers"),
263     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_CONF TAKE2,
264     ngx_conf_set_bufs_slot,
265     NGX\_HTTP\_SRV\_CONF\_OFFSET,
266     offsetof(ngx_http_core_srv_conf_t, large_client_header_buffers),
267     NULL },
268
269 { ngx_string("optimize_server_names"),
270     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_CONF FLAG,
271     ngx_conf_set_flag_slot,
272     NGX\_HTTP\_LOC\_CONF\_OFFSET,
273     offsetof(ngx_http_core_loc_conf_t, server_name_in_redirect),
274     &ngx_conf_deprecated_optimize_server_names },
275
276 { ngx_string("ignore_invalid_headers"),
277     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_CONF FLAG,
278     ngx_conf_set_flag_slot,
279     NGX\_HTTP\_SRV\_CONF\_OFFSET,
280     offsetof(ngx_http_core_srv_conf_t, ignore_invalid_headers),
281     NULL },
282
283 { ngx_string("merge_slashes"),
284     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_CONF FLAG,
285     ngx_conf_set_flag_slot,
286     NGX\_HTTP\_SRV\_CONF\_OFFSET,
287     offsetof(ngx_http_core_srv_conf_t, merge_slashes),
288     NULL },
289
290 { ngx_string("underscores_in_headers"),
291     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_CONF FLAG,
292     ngx_conf_set_flag_slot,
293     NGX\_HTTP\_SRV\_CONF\_OFFSET,
294     offsetof(ngx_http_core_srv_conf_t, underscores_in_headers),
295     NULL },
296
297 { ngx_string("location"),
298     NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF BLOCK | NGX\_CONF TAKE12,
299     ngx_http_core_location,
300     NGX\_HTTP\_SRV\_CONF\_OFFSET,
301     0,

```

```

302     NULL },
303
304 { ngx_string("listen"),
305     NGX_HTTP_SRV_CONF|NGX_CONF_1MORE,
306     ngx_http_core_listen,
307     NGX_HTTP_SRV_CONF_OFFSET,
308     0,
309     NULL },
310
311 { ngx_string("server_name"),
312     NGX_HTTP_SRV_CONF|NGX_CONF_1MORE,
313     ngx_http_core_server_name,
314     NGX_HTTP_SRV_CONF_OFFSET,
315     0,
316     NULL },
317
318 { ngx_string("types_hash_max_size"),
319     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
320     ngx_conf_set_num_slot,
321     NGX_HTTP_LOC_CONF_OFFSET,
322     offsetof(ngx_http_core_loc_conf_t, types_hash_max_size),
323     NULL },
324
325 { ngx_string("types_hash_bucket_size"),
326     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
327     ngx_conf_set_num_slot,
328     NGX_HTTP_LOC_CONF_OFFSET,
329     offsetof(ngx_http_core_loc_conf_t, types_hash_bucket_size),
330     NULL },
331
332 { ngx_string("types"),
333     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF
334     |NGX_CONF_BLOCK|NGX_CONF_NOARGS,
335     ngx_http_core_types,
336     NGX_HTTP_LOC_CONF_OFFSET,
337     0,
338     NULL },
339
340 { ngx_string("default_type"),
341     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
342     ngx_conf_set_str_slot,
343     NGX_HTTP_LOC_CONF_OFFSET,
344     offsetof(ngx_http_core_loc_conf_t, default_type),
345     NULL },
346
347 { ngx_string("root"),
348     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_HTTP_LIF_CONF
349     |NGX_CONF_TAKE1,
350     ngx_http_core_root,
351     NGX_HTTP_LOC_CONF_OFFSET,
352     0,
353     NULL },
354
355 { ngx_string("alias"),
356     NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
357     ngx_http_core_root,
358     NGX_HTTP_LOC_CONF_OFFSET,
359     0,
360     NULL },
361
362 { ngx_string("limit_except"),
363     NGX_HTTP_LOC_CONF|NGX_CONF_BLOCK|NGX_CONF_1MORE,
364     ngx_http_core_limit_except,
365     NGX_HTTP_LOC_CONF_OFFSET,
366     0,
367     NULL },
368
369 { ngx_string("client_max_body_size"),
370     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
371     ngx_conf_set_off_slot,
372     NGX_HTTP_LOC_CONF_OFFSET,
373     offsetof(ngx_http_core_loc_conf_t, client_max_body_size),
374     NULL },
375
376 { ngx_string("client_body_buffer_size"),
377     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,

```

```

378     ngx_conf_set_size_slot,
379     NGX_HTTP_LOC_CONF_OFFSET,
380     offsetof(ngx_http_core_loc_conf_t, client_body_buffer_size),
381     NULL },
382
383 { ngx_string("client_body_timeout"),
384     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
385     ngx_conf_set_msec_slot,
386     NGX_HTTP_LOC_CONF_OFFSET,
387     offsetof(ngx_http_core_loc_conf_t, client_body_timeout),
388     NULL },
389
390 { ngx_string("client_body_temp_path"),
391     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1234,
392     ngx_conf_set_path_slot,
393     NGX_HTTP_LOC_CONF_OFFSET,
394     offsetof(ngx_http_core_loc_conf_t, client_body_temp_path),
395     NULL },
396
397 { ngx_string("client_body_in_file_only"),
398     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
399     ngx_conf_set_enum_slot,
400     NGX_HTTP_LOC_CONF_OFFSET,
401     offsetof(ngx_http_core_loc_conf_t, client_body_in_file_only),
402     &ngx_http_core_request_body_in_file },
403
404 { ngx_string("client_body_in_single_buffer"),
405     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
406     ngx_conf_set_flag_slot,
407     NGX_HTTP_LOC_CONF_OFFSET,
408     offsetof(ngx_http_core_loc_conf_t, client_body_in_single_buffer),
409     NULL },
410
411 { ngx_string("sendfile"),
412     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_HTTP_LIF_CONF
413         |NGX_CONF_FLAG,
414     ngx_conf_set_flag_slot,
415     NGX_HTTP_LOC_CONF_OFFSET,
416     offsetof(ngx_http_core_loc_conf_t, sendfile),
417     NULL },
418
419 { ngx_string("sendfile_max_chunk"),
420     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
421     ngx_conf_set_size_slot,
422     NGX_HTTP_LOC_CONF_OFFSET,
423     offsetof(ngx_http_core_loc_conf_t, sendfile_max_chunk),
424     NULL },
425
426 #if (NGX_HAVE_FILE_AIO)
427
428 { ngx_string("aio"),
429     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
430     ngx_conf_set_enum_slot,
431     NGX_HTTP_LOC_CONF_OFFSET,
432     offsetof(ngx_http_core_loc_conf_t, aio),
433     &ngx_http_core_aio },
434
435 #endif
436
437 { ngx_string("read_ahead"),
438     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
439     ngx_conf_set_size_slot,
440     NGX_HTTP_LOC_CONF_OFFSET,
441     offsetof(ngx_http_core_loc_conf_t, read_ahead),
442     NULL },
443
444 { ngx_string("directio"),
445     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
446     ngx_http_core_directio,
447     NGX_HTTP_LOC_CONF_OFFSET,
448     0,
449     NULL },
450
451 { ngx_string("directio_alignment"),
452     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
453     ngx_conf_set_off_slot,

```



```

454     NGX\_HTTP\_LOC\_CONF\_OFFSET,
455     offsetof\(ngx\\_http\\_core\\_loc\\_conf\\_t, directio\\_alignment\),
456     NULL },
457
458 { ngx\_string\("tcp\_nopush"\),
459   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_FLAG,
460   ngx\_conf\_set\_flag\_slot,
461   NGX\_HTTP\_LOC\_CONF\_OFFSET,
462   offsetof\(ngx\\_http\\_core\\_loc\\_conf\\_t, tcp\\_nopush\),
463   NULL },
464
465 { ngx\_string\("tcp\_nodelay"\),
466   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_FLAG,
467   ngx\_conf\_set\_flag\_slot,
468   NGX\_HTTP\_LOC\_CONF\_OFFSET,
469   offsetof\(ngx\\_http\\_core\\_loc\\_conf\\_t, tcp\\_nodelay\),
470   NULL },
471
472 { ngx\_string\("send\_timeout"\),
473   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
474   ngx\_conf\_set\_msec\_slot,
475   NGX\_HTTP\_LOC\_CONF\_OFFSET,
476   offsetof\(ngx\\_http\\_core\\_loc\\_conf\\_t, send\\_timeout\),
477   NULL },
478
479 { ngx\_string\("send\_lowat"\),
480   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
481   ngx\_conf\_set\_size\_slot,
482   NGX\_HTTP\_LOC\_CONF\_OFFSET,
483   offsetof\(ngx\\_http\\_core\\_loc\\_conf\\_t, send\\_lowat\),
484   &ngx\\_http\\_core\\_lowat\\_post },
485
486 { ngx\_string\("postpone\_output"\),
487   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
488   ngx\_conf\_set\_size\_slot,
489   NGX\_HTTP\_LOC\_CONF\_OFFSET,
490   offsetof\(ngx\\_http\\_core\\_loc\\_conf\\_t, postpone\\_output\),
491   NULL },
492
493 { ngx\_string\("limit\_rate"\),
494   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_HTTP\_LIF\_CONF
495     |NGX\_CONF\_TAKE1,
496   ngx\_conf\_set\_size\_slot,
497   NGX\_HTTP\_LOC\_CONF\_OFFSET,
498   offsetof\(ngx\\_http\\_core\\_loc\\_conf\\_t, limit\\_rate\),
499   NULL },
500
501 { ngx\_string\("limit\_rate\_after"\),
502   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_HTTP\_LIF\_CONF
503     |NGX\_CONF\_TAKE1,
504   ngx\_conf\_set\_size\_slot,
505   NGX\_HTTP\_LOC\_CONF\_OFFSET,
506   offsetof\(ngx\\_http\\_core\\_loc\\_conf\\_t, limit\\_rate\\_after\),
507   NULL },
508
509 { ngx\_string\("keepalive\_timeout"\),
510   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE12,
511   ngx\_http\_core\_keepalive,
512   NGX\_HTTP\_LOC\_CONF\_OFFSET,
513   0,
514   NULL },
515
516 { ngx\_string\("keepalive\_requests"\),
517   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
518   ngx\_conf\_set\_num\_slot,
519   NGX\_HTTP\_LOC\_CONF\_OFFSET,
520   offsetof\(ngx\\_http\\_core\\_loc\\_conf\\_t, keepalive\\_requests\),
521   NULL },
522
523 { ngx\_string\("keepalive\_disable"\),
524   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE12,
525   ngx\_conf\_set\_bitmask\_slot,
526   NGX\_HTTP\_LOC\_CONF\_OFFSET,
527   offsetof\(ngx\\_http\\_core\\_loc\\_conf\\_t, keepalive\\_disable\),
528   &ngx\\_http\\_core\\_keepalive\\_disable },
529

```

```

530 { ngx_string("satisfy"),
531     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
532     ngx_conf_set_enum_slot,
533     NGX\_HTTP\_LOC\_CONF\_OFFSET,
534     offsetof(ngx\_http\_core\_loc\_conf\_t, satisfy),
535     &ngx\_http\_core\_satisfy },
536
537 { ngx_string("satisfy_any"),
538     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_FLAG,
539     ngx_conf_set_flag_slot,
540     NGX\_HTTP\_LOC\_CONF\_OFFSET,
541     offsetof(ngx\_http\_core\_loc\_conf\_t, satisfy),
542     &ngx\_conf\_deprecated\_satisfy\_any },
543
544 { ngx_string("internal"),
545     NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_NOARGS,
546     ngx_http_core_internal,
547     NGX\_HTTP\_LOC\_CONF\_OFFSET,
548     0,
549     NULL },
550
551 { ngx_string("lingering_close"),
552     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
553     ngx_conf_set_enum_slot,
554     NGX\_HTTP\_LOC\_CONF\_OFFSET,
555     offsetof(ngx\_http\_core\_loc\_conf\_t, lingering_close),
556     &ngx\_http\_core\_lingering\_close },
557
558 { ngx_string("lingering_time"),
559     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
560     ngx_conf_set_msec_slot,
561     NGX\_HTTP\_LOC\_CONF\_OFFSET,
562     offsetof(ngx\_http\_core\_loc\_conf\_t, lingering_time),
563     NULL },
564
565 { ngx_string("lingering_timeout"),
566     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
567     ngx_conf_set_msec_slot,
568     NGX\_HTTP\_LOC\_CONF\_OFFSET,
569     offsetof(ngx\_http\_core\_loc\_conf\_t, lingering_timeout),
570     NULL },
571
572 { ngx_string("reset_timedout_connection"),
573     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_FLAG,
574     ngx_conf_set_flag_slot,
575     NGX\_HTTP\_LOC\_CONF\_OFFSET,
576     offsetof(ngx\_http\_core\_loc\_conf\_t, reset_timedout_connection),
577     NULL },
578
579 { ngx_string("server_name_in_redirect"),
580     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_FLAG,
581     ngx_conf_set_flag_slot,
582     NGX\_HTTP\_LOC\_CONF\_OFFSET,
583     offsetof(ngx\_http\_core\_loc\_conf\_t, server_name_in_redirect),
584     NULL },
585
586 { ngx_string("port_in_redirect"),
587     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_FLAG,
588     ngx_conf_set_flag_slot,
589     NGX\_HTTP\_LOC\_CONF\_OFFSET,
590     offsetof(ngx\_http\_core\_loc\_conf\_t, port_in_redirect),
591     NULL },
592
593 { ngx_string("msie_padding"),
594     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_FLAG,
595     ngx_conf_set_flag_slot,
596     NGX\_HTTP\_LOC\_CONF\_OFFSET,
597     offsetof(ngx\_http\_core\_loc\_conf\_t, msie_padding),
598     NULL },
599
600 { ngx_string("msie_refresh"),
601     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_FLAG,
602     ngx_conf_set_flag_slot,
603     NGX\_HTTP\_LOC\_CONF\_OFFSET,
604     offsetof(ngx\_http\_core\_loc\_conf\_t, msie_refresh),
605     NULL },

```

```

606 { ngx_string("log_not_found"),
607     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_FLAG,
608     ngx\_conf\_set\_flag\_slot,
609     NGX\_HTTP\_LOC\_CONF\_OFFSET,
610     offsetof\(ngx\_http\_core\_loc\_conf\_t, log\_not\_found\),
611     NULL },
612
613
614 { ngx_string("log_subrequest"),
615     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_FLAG,
616     ngx\_conf\_set\_flag\_slot,
617     NGX\_HTTP\_LOC\_CONF\_OFFSET,
618     offsetof\(ngx\_http\_core\_loc\_conf\_t, log\_subrequest\),
619     NULL },
620
621
622 { ngx_string("recursive_error_pages"),
623     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_FLAG,
624     ngx\_conf\_set\_flag\_slot,
625     NGX\_HTTP\_LOC\_CONF\_OFFSET,
626     offsetof\(ngx\_http\_core\_loc\_conf\_t, recursive\_error\_pages\),
627     NULL },
628
629
630 { ngx_string("server_tokens"),
631     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_FLAG,
632     ngx\_conf\_set\_flag\_slot,
633     NGX\_HTTP\_LOC\_CONF\_OFFSET,
634     offsetof\(ngx\_http\_core\_loc\_conf\_t, server\_tokens\),
635     NULL },
636
637
638 { ngx_string("if_modified_since"),
639     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
640     ngx\_conf\_set\_enum\_slot,
641     NGX\_HTTP\_LOC\_CONF\_OFFSET,
642     offsetof\(ngx\_http\_core\_loc\_conf\_t, if\_modified\_since\),
643     &ngx\_http\_core\_if\_modified\_since },
644
645
646 { ngx_string("max_ranges"),
647     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
648     ngx\_conf\_set\_num\_slot,
649     NGX\_HTTP\_LOC\_CONF\_OFFSET,
650     offsetof\(ngx\_http\_core\_loc\_conf\_t, max\_ranges\),
651     NULL },
652
653
654 { ngx_string("chunked_transfer_encoding"),
655     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_FLAG,
656     ngx\_conf\_set\_flag\_slot,
657     NGX\_HTTP\_LOC\_CONF\_OFFSET,
658     offsetof\(ngx\_http\_core\_loc\_conf\_t, chunked\_transfer\_encoding\),
659     NULL },
660
661
662 { ngx_string("etag"),
663     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_FLAG,
664     ngx\_conf\_set\_flag\_slot,
665     NGX\_HTTP\_LOC\_CONF\_OFFSET,
666     offsetof\(ngx\_http\_core\_loc\_conf\_t, etag\),
667     NULL },
668
669
670 { ngx_string("error_page"),
671     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_HTTP\_LIF\_CONF
672     | NGX\_CONF\_2MORE,
673     ngx\_http\_core\_error\_page,
674     NGX\_HTTP\_LOC\_CONF\_OFFSET,
675     0,
676     NULL },
677
678
679 { ngx_string("try_files"),
680     NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_2MORE,
681     ngx\_http\_core\_try\_files,
682     NGX\_HTTP\_LOC\_CONF\_OFFSET,
683     0,
684     NULL },
685
686
687 { ngx_string("post_action"),
688     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_HTTP\_LIF\_CONF
689     | NGX\_CONF\_TAKE1,
690     ngx\_conf\_set\_str\_slot,

```

```

682     NGX\_HTTP\_LOC\_CONF\_OFFSET,
683     offsetof(ngx\_http\_core\_loc\_conf\_t, post_action),
684     NULL },
685
686 { ngx\_string\("error\_log"\),
687   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_1MORE,
688   ngx\_http\_core\_error\_log,
689   NGX\_HTTP\_LOC\_CONF\_OFFSET,
690   0,
691   NULL },
692
693 { ngx\_string\("open\_file\_cache"\),
694   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE12,
695   ngx\_http\_core\_open\_file\_cache,
696   NGX\_HTTP\_LOC\_CONF\_OFFSET,
697   offsetof(ngx\_http\_core\_loc\_conf\_t, open_file_cache),
698   NULL },
699
700 { ngx\_string\("open\_file\_cache\_valid"\),
701   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
702   ngx\_conf\_set\_sec\_slot,
703   NGX\_HTTP\_LOC\_CONF\_OFFSET,
704   offsetof(ngx\_http\_core\_loc\_conf\_t, open_file_cache_valid),
705   NULL },
706
707 { ngx\_string\("open\_file\_cache\_retest"\),
708   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
709   ngx\_conf\_set\_sec\_slot,
710   NGX\_HTTP\_LOC\_CONF\_OFFSET,
711   offsetof(ngx\_http\_core\_loc\_conf\_t, open_file_cache_valid),
712   &ngx\_conf\_deprecated\_open\_file\_cache\_retest },
713
714 { ngx\_string\("open\_file\_cache\_min\_uses"\),
715   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
716   ngx\_conf\_set\_num\_slot,
717   NGX\_HTTP\_LOC\_CONF\_OFFSET,
718   offsetof(ngx\_http\_core\_loc\_conf\_t, open_file_cache_min_uses),
719   NULL },
720
721 { ngx\_string\("open\_file\_cache\_errors"\),
722   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_FLAG,
723   ngx\_conf\_set\_flag\_slot,
724   NGX\_HTTP\_LOC\_CONF\_OFFSET,
725   offsetof(ngx\_http\_core\_loc\_conf\_t, open_file_cache_errors),
726   NULL },
727
728 { ngx\_string\("open\_file\_cache\_events"\),
729   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_FLAG,
730   ngx\_conf\_set\_flag\_slot,
731   NGX\_HTTP\_LOC\_CONF\_OFFSET,
732   offsetof(ngx\_http\_core\_loc\_conf\_t, open_file_cache_events),
733   NULL },
734
735 { ngx\_string\("resolver"\),
736   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_1MORE,
737   ngx\_http\_core\_resolver,
738   NGX\_HTTP\_LOC\_CONF\_OFFSET,
739   0,
740   NULL },
741
742 { ngx\_string\("resolver\_timeout"\),
743   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
744   ngx\_conf\_set\_msec\_slot,
745   NGX\_HTTP\_LOC\_CONF\_OFFSET,
746   offsetof(ngx\_http\_core\_loc\_conf\_t, resolver_timeout),
747   NULL },
748
749 #if (NGX_HTTP_GZIP)
750
751 { ngx\_string\("gzip\_vary"\),
752   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_FLAG,
753   ngx\_conf\_set\_flag\_slot,
754   NGX\_HTTP\_LOC\_CONF\_OFFSET,
755   offsetof(ngx\_http\_core\_loc\_conf\_t, gzip_vary),
756   NULL },
757

```

```

758 { ngx_string("gzip_http_version"),
759     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
760     ngx_conf_set_enum_slot,
761     NGX_HTTP_LOC_CONF_OFFSET,
762     offsetof(ngx_http_core_loc_conf_t, gzip_http_version),
763     &ngx_http_gzip_http_version },
764
765 { ngx_string("gzip_proxied"),
766     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_1MORE,
767     ngx_conf_set_bitmask_slot,
768     NGX_HTTP_LOC_CONF_OFFSET,
769     offsetof(ngx_http_core_loc_conf_t, gzip_proxied),
770     &ngx_http_gzip_proxied_mask },
771
772 { ngx_string("gzip_disable"),
773     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_1MORE,
774     ngx_http_gzip_disable,
775     NGX_HTTP_LOC_CONF_OFFSET,
776     0,
777     NULL },
778
779 #endif
780
781 #if (NGX_HAVE_OPENAT)
782
783 { ngx_string("disable_symlinks"),
784     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE12,
785     ngx_http_disable_symlinks,
786     NGX_HTTP_LOC_CONF_OFFSET,
787     0,
788     NULL },
789
790 #endif
791
792     ngx_null_command
793 };
794
795
796 static ngx_http_module_t ngx_http_core_module_ctx = {
797     ngx_http_core_preconfiguration,    /* preconfiguration */
798     NULL,                               /* postconfiguration */
799
800     ngx_http_core_create_main_conf,    /* create main configuration */
801     ngx_http_core_init_main_conf,     /* init main configuration */
802
803     ngx_http_core_create_srv_conf,    /* create server configuration */
804     ngx_http_core_merge_srv_conf,     /* merge server configuration */
805
806     ngx_http_core_create_loc_conf,    /* create location configuration */
807     ngx_http_core_merge_loc_conf     /* merge location configuration */
808 };
809
810
811 ngx_module_t ngx_http_core_module = {
812     NGX_MODULE_V1,
813     &ngx_http_core_module_ctx,        /* module context */
814     ngx_http_core_commands,           /* module directives */
815     NGX_HTTP_MODULE,                  /* module type */
816     NULL,                              /* init master */
817     NULL,                              /* init module */
818     NULL,                              /* init process */
819     NULL,                              /* init thread */
820     NULL,                              /* exit thread */
821     NULL,                              /* exit process */
822     NULL,                              /* exit master */
823     NGX_MODULE_V1_PADDING
824 };
825
826
827 ngx_str_t ngx_http_core_get_method = { 3, (u_char *) "GET " };
828
829
830 void
831 ngx_http_handler(ngx_http_request_t *r)
832 {
833     ngx_http_core_main_conf_t *cmcf;

```

```

834     r->connection->log->action = NULL;
835
836
837     r->connection->unexpected_eof = 0;
838
839     if (!r->internal) {
840         switch (r->headers_in.connection_type) {
841             case 0:
842                 r->keepalive = (r->http_version > NGX\_HTTP\_VERSION\_10);
843                 break;
844
845             case NGX\_HTTP\_CONNECTION\_CLOSE:
846                 r->keepalive = 0;
847                 break;
848
849             case NGX\_HTTP\_CONNECTION\_KEEP\_ALIVE:
850                 r->keepalive = 1;
851                 break;
852         }
853
854         r->lingering_close = (r->headers_in.content_length_n > 0
855                             || r->headers_in.chunked);
856         r->phase_handler = 0;
857
858     } else {
859         cmcf = ngx\_http\_get\_module\_main\_conf(r, ngx\_http\_core\_module);
860         r->phase_handler = cmcf->phase_engine.server_rewrite_index;
861     }
862
863     r->valid_location = 1;
864     #if (NGX\_HTTP\_GZIP)
865     r->gzip_tested = 0;
866     r->gzip_ok = 0;
867     r->gzip_vary = 0;
868 #endif
869
870     r->write_event_handler = ngx\_http\_core\_run\_phases;
871     ngx\_http\_core\_run\_phases(r);
872 }
873
874
875 void
876 ngx\_http\_core\_run\_phases(ngx\_http\_request\_t *r)
877 {
878     ngx\_int\_t          rc;
879     ngx\_http\_phase\_handler\_t *ph;
880     ngx\_http\_core\_main\_conf\_t *cmcf;
881
882     cmcf = ngx\_http\_get\_module\_main\_conf(r, ngx\_http\_core\_module);
883
884     ph = cmcf->phase_engine.handlers;
885
886     while (ph[r->phase_handler].checker){
887
888         rc = ph[r->phase_handler].checker(r, &ph[r->phase_handler]);
889
890         if (rc == NGX\_OK) {
891             return;
892         }
893     }
894 }
895
896
897 ngx\_int\_t
898 ngx\_http\_core\_generic\_phase(ngx\_http\_request\_t *r, ngx\_http\_phase\_handler\_t *ph)
899 {
900     ngx\_int\_t rc;
901
902     /*
903      * generic phase checker,
904      * used by the post read and pre-access phases
905      */
906
907     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
908                  "generic phase: %ui", r->phase_handler);
909 }

```

```

910     rc = ph->handler(r);
911
912     if (rc == NGX\_OK) {
913         r->phase_handler = ph->next;
914         return NGX\_AGAIN;
915     }
916
917     if (rc == NGX\_DECLINED) {
918         r->phase_handler++;
919         return NGX\_AGAIN;
920     }
921
922     if (rc == NGX\_AGAIN || rc == NGX\_DONE) {
923         return NGX\_OK;
924     }
925
926     /* rc == NGX\_ERROR || rc == NGX\_HTTP\_... */
927
928     ngx\_http\_finalize\_request(r, rc);
929
930     return NGX\_OK;
931 }
932
933
934 ngx\_int\_t
935 ngx\_http\_core\_rewrite\_phase(ngx\_http\_request\_t *r, ngx\_http\_phase\_handler\_t *ph)
936 {
937     ngx\_int\_t rc;
938
939     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
940         "rewrite phase: %ui", r->phase_handler);
941
942     rc = ph->handler(r);
943
944     if (rc == NGX\_DECLINED) {
945         r->phase_handler++;
946         return NGX\_AGAIN;
947     }
948
949     if (rc == NGX\_DONE) {
950         return NGX\_OK;
951     }
952
953     /* NGX\_OK, NGX\_AGAIN, NGX\_ERROR, NGX\_HTTP\_... */
954
955     ngx\_http\_finalize\_request(r, rc);
956
957     return NGX\_OK;
958 }
959
960
961 ngx\_int\_t
962 ngx\_http\_core\_find\_config\_phase(ngx\_http\_request\_t *r,
963     ngx\_http\_phase\_handler\_t *ph)
964 {
965     u\_char *p;
966     size\_t len;
967     ngx\_int\_t rc;
968     ngx\_http\_core\_loc\_conf\_t *clcf;
969
970     r->content_handler = NULL;
971     r->uri_changed = 0;
972
973     rc = ngx\_http\_core\_find\_location(r);
974
975     if (rc == NGX\_ERROR) {
976         ngx\_http\_finalize\_request(r, NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);
977         return NGX\_OK;
978     }
979
980     clcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_core\_module);
981
982     if (!r->internal && clcf->internal) {
983         ngx\_http\_finalize\_request(r, NGX\_HTTP\_NOT\_FOUND);
984         return NGX\_OK;
985     }

```

```

986 ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
987     "using configuration \"%s%V\"",
988     (clcf->noname ? "*" : (clcf->exact_match ? "=" : "")),
989     &clcf->name);
991
992 ngx_http_update_location_config(r);
993
994 ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
995     "http cl:%0 max:%0",
996     r->headers_in.content_length_n, clcf->client_max_body_size);
997
998 if (r->headers_in.content_length_n != -1
999     && !r->discard_body
1000     && clcf->client_max_body_size
1001     && clcf->client_max_body_size < r->headers_in.content_length_n)
1002 {
1003     ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1004         "client intended to send too large body: %0 bytes",
1005         r->headers_in.content_length_n);
1006
1007     r->expect_tested = 1;
1008     (void) ngx_http_discard_request_body(r);
1009     ngx_http_finalize_request(r, NGX_HTTP_REQUEST_ENTITY_TOO_LARGE);
1010     return NGX_OK;
1011 }
1012
1013 if (rc == NGX_DONE) {
1014     ngx_http_clear_location(r);
1015
1016     r->headers_out.location = ngx_list_push(&r->headers_out.headers);
1017     if (r->headers_out.location == NULL) {
1018         ngx_http_finalize_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
1019         return NGX_OK;
1020     }
1021
1022     /*
1023      * we do not need to set the r->headers_out.location->hash and
1024      * r->headers_out.location->key fields
1025      */
1026
1027     if (r->args.len == 0) {
1028         r->headers_out.location->value = clcf->name;
1029
1030     } else {
1031         len = clcf->name.len + 1 + r->args.len;
1032         p = ngx_pnalloc(r->pool, len);
1033
1034         if (p == NULL) {
1035             ngx_http_finalize_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
1036             return NGX_OK;
1037         }
1038
1039         r->headers_out.location->value.len = len;
1040         r->headers_out.location->value.data = p;
1041
1042         p = ngx_cpymem(p, clcf->name.data, clcf->name.len);
1043         *p++ = '?';
1044         ngx_memcpy(p, r->args.data, r->args.len);
1045     }
1046
1047     ngx_http_finalize_request(r, NGX_HTTP_MOVED_PERMANENTLY);
1048     return NGX_OK;
1049 }
1050
1051 r->phase_handler++;
1052 return NGX_AGAIN;
1053 }
1054
1055
1056 ngx_int_t
1057 ngx_http_core_post_rewrite_phase(ngx_http_request_t *r,
1058     ngx_http_phase_handler_t *ph)
1059 {
1060     ngx_http_core_srv_conf_t *cscf;
1061

```



```

1062 ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
1063     "post rewrite phase: %ui", r->phase_handler);
1064
1065 if (!r->uri_changed) {
1066     r->phase_handler++;
1067     return NGX\_AGAIN;
1068 }
1069
1070 ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
1071     "uri changes: %d", r->uri_changes);
1072
1073 /*
1074  * gcc before 3.3 compiles the broken code for
1075  *     if (r->uri_changes-- == 0)
1076  * if the r->uri_changes is defined as
1077  *     unsigned uri_changes:4
1078  */
1079
1080 r->uri_changes--;
1081
1082 if (r->uri_changes == 0) {
1083     ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, 0,
1084         "rewrite or internal redirection cycle "
1085         "while processing \"%V\"", &r->uri);
1086
1087     ngx\_http\_finalize\_request(r, NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);
1088     return NGX\_OK;
1089 }
1090
1091 r->phase_handler = ph->next;
1092
1093 cscf = ngx\_http\_get\_module\_srv\_conf(r, ngx\_http\_core\_module);
1094 r->loc_conf = cscf->ctx->loc_conf;
1095
1096 return NGX\_AGAIN;
1097 }
1098
1099
1100 ngx\_int\_t
1101 ngx\_http\_core\_access\_phase(ngx\_http\_request\_t *r, ngx\_http\_phase\_handler\_t *ph)
1102 {
1103     ngx\_int\_t rc;
1104     ngx\_http\_core\_loc\_conf\_t *clcf;
1105
1106     if (r != r->main) {
1107         r->phase_handler = ph->next;
1108         return NGX\_AGAIN;
1109     }
1110
1111     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
1112         "access phase: %ui", r->phase_handler);
1113
1114     rc = ph->handler(r);
1115
1116     if (rc == NGX\_DECLINED) {
1117         r->phase_handler++;
1118         return NGX\_AGAIN;
1119     }
1120
1121     if (rc == NGX\_AGAIN || rc == NGX\_DONE) {
1122         return NGX\_OK;
1123     }
1124
1125     clcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_core\_module);
1126
1127     if (clcf->satisfy == NGX\_HTTP\_SATISFY\_ALL) {
1128
1129         if (rc == NGX\_OK) {
1130             r->phase_handler++;
1131             return NGX\_AGAIN;
1132         }
1133
1134     } else {
1135         if (rc == NGX\_OK) {
1136             r->access_code = 0;
1137

```

```

1138         if (r->headers_out.www_authenticate) {
1139             r->headers_out.www_authenticate->hash = 0;
1140         }
1141
1142         r->phase_handler = ph->next;
1143         return NGX\_AGAIN;
1144     }
1145
1146     if (rc == NGX\_HTTP\_FORBIDDEN || rc == NGX\_HTTP\_UNAUTHORIZED) {
1147         if (r->access_code != NGX\_HTTP\_UNAUTHORIZED) {
1148             r->access_code = rc;
1149         }
1150
1151         r->phase_handler++;
1152         return NGX\_AGAIN;
1153     }
1154 }
1155
1156 /* rc == NGX\_ERROR || rc == NGX\_HTTP\_... */
1157
1158 ngx\_http\_finalize\_request(r, rc);
1159 return NGX\_OK;
1160 }
1161
1162
1163 ngx\_int\_t
1164 ngx\_http\_core\_post\_access\_phase(ngx\_http\_request\_t *r,
1165     ngx\_http\_phase\_handler\_t *ph)
1166 {
1167     ngx\_int\_t access_code;
1168
1169     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
1170         "post access phase: %ui", r->phase_handler);
1171
1172     access_code = r->access_code;
1173
1174     if (access_code) {
1175         if (access_code == NGX\_HTTP\_FORBIDDEN) {
1176             ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, 0,
1177                 "access forbidden by rule");
1178         }
1179
1180         r->access_code = 0;
1181         ngx\_http\_finalize\_request(r, access_code);
1182         return NGX\_OK;
1183     }
1184
1185     r->phase_handler++;
1186     return NGX\_AGAIN;
1187 }
1188
1189
1190 ngx\_int\_t
1191 ngx\_http\_core\_try\_files\_phase(ngx\_http\_request\_t *r,
1192     ngx\_http\_phase\_handler\_t *ph)
1193 {
1194     size\_t len, root, alias, reserve, allocated;
1195     u\_char *p, *name;
1196     ngx\_str\_t path, args;
1197     ngx\_uint\_t test_dir;
1198     ngx\_http\_try\_file\_t *tf;
1199     ngx\_open\_file\_info\_t of;
1200     ngx\_http\_script\_code\_pt code;
1201     ngx\_http\_script\_engine\_t e;
1202     ngx\_http\_core\_loc\_conf\_t *clcf;
1203     ngx\_http\_script\_len\_code\_pt lcode;
1204
1205     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
1206         "try files phase: %ui", r->phase_handler);
1207
1208     clcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_core\_module);
1209
1210     if (clcf->try_files == NULL) {
1211         r->phase_handler++;
1212         return NGX\_AGAIN;
1213     }

```

```

1214 allocated = 0;
1215 root = 0;
1216 name = NULL;
1217 /* suppress MSVC warning */
1218 path.data = NULL;
1219
1220
1221 tf = clcf->try_files;
1222
1223 alias = clcf->alias;
1224
1225 for ( ;; ) {
1226
1227     if (tf->lengths) {
1228         ngx_memzero(&e, sizeof(ngx_http_script_engine_t));
1229
1230         e.ip = tf->lengths->elts;
1231         e.request = r;
1232
1233         /* 1 is for terminating '\0' as in static names */
1234         len = 1;
1235
1236         while (*(uintptr_t *) e.ip) {
1237             lcode = *(ngx_http_script_len_code_pt *) e.ip;
1238             len += lcode(&e);
1239         }
1240
1241     } else {
1242         len = tf->name.len;
1243     }
1244
1245     if (!alias) {
1246         reserve = len > r->uri.len ? len - r->uri.len : 0;
1247
1248     } else if (alias == NGX_MAX_SIZE_T_VALUE) {
1249         reserve = len;
1250
1251     } else {
1252         reserve = len > r->uri.len - alias ? len - (r->uri.len - alias) : 0;
1253     }
1254
1255     if (reserve > allocated || !allocated) {
1256
1257         /* 16 bytes are preallocation */
1258         allocated = reserve + 16;
1259
1260         if (ngx_http_map_uri_to_path(r, &path, &root, allocated) == NULL) {
1261             ngx_http_finalize_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
1262             return NGX_OK;
1263         }
1264
1265         name = path.data + root;
1266     }
1267
1268     if (tf->values == NULL) {
1269
1270         /* tf->name.len includes the terminating '\0' */
1271
1272         ngx_memcpy(name, tf->name.data, tf->name.len);
1273
1274         path.len = (name + tf->name.len - 1) - path.data;
1275
1276     } else {
1277         e.ip = tf->values->elts;
1278         e.pos = name;
1279         e.flushed = 1;
1280
1281         while (*(uintptr_t *) e.ip) {
1282             code = *(ngx_http_script_code_pt *) e.ip;
1283             code((ngx_http_script_engine_t *) &e);
1284         }
1285
1286         path.len = e.pos - path.data;
1287
1288         *e.pos = '\0';
1289

```

```

1290     if (alias && ngx_strncmp(name, clcf->name.data, alias) == 0) {
1291         ngx_memmove(name, name + alias, len - alias);
1292         path.len -= alias;
1293     }
1294 }
1295
1296 test_dir = tf->test_dir;
1297
1298 tf++;
1299
1300 ngx_log_debug3(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1301     "trying to use %s: \"%s\" \"%s\"",
1302     test_dir ? "dir" : "file", name, path.data);
1303
1304 if (tf->lengths == NULL && tf->name.len == 0) {
1305
1306     if (tf->code) {
1307         ngx_http_finalize_request(r, tf->code);
1308         return NGX_OK;
1309     }
1310
1311     path.len -= root;
1312     path.data += root;
1313
1314     if (path.data[0] == '@') {
1315         (void) ngx_http_named_location(r, &path);
1316     }
1317     else {
1318         ngx_http_split_args(r, &path, &args);
1319
1320         (void) ngx_http_internal_redirect(r, &path, &args);
1321     }
1322
1323     ngx_http_finalize_request(r, NGX_DONE);
1324     return NGX_OK;
1325 }
1326
1327 ngx_memzero(&of, sizeof(ngx_open_file_info_t));
1328
1329 of.read_ahead = clcf->read_ahead;
1330 of.directio = clcf->directio;
1331 of.valid = clcf->open_file_cache_valid;
1332 of.min_uses = clcf->open_file_cache_min_uses;
1333 of.test_only = 1;
1334 of.errors = clcf->open_file_cache_errors;
1335 of.events = clcf->open_file_cache_events;
1336
1337 if (ngx_http_set_disable_symlinks(r, clcf, &path, &of) != NGX_OK) {
1338     ngx_http_finalize_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
1339     return NGX_OK;
1340 }
1341
1342 if (ngx_open_cached_file(clcf->open_file_cache, &path, &of, r->pool)
1343     != NGX_OK)
1344 {
1345     if (of.err != NGX_ENOENT
1346         && of.err != NGX_ENOTDIR
1347         && of.err != NGX_ENAMETOOLONG)
1348     {
1349         ngx_log_error(NGX_LOG_CRIT, r->connection->log, of.err,
1350             "%s \"%s\" failed", of.failed, path.data);
1351     }
1352
1353     continue;
1354 }
1355
1356 if (of.is_dir != test_dir) {
1357     continue;
1358 }
1359
1360 path.len -= root;
1361 path.data += root;
1362
1363 if (!alias) {
1364     r->uri = path;
1365

```

```

1366     } else if (alias == NGX_MAX_SIZE_T_VALUE) {
1367         if (!test_dir) {
1368             r->uri = path;
1369             r->add_uri_to_alias = 1;
1370         }
1371
1372     } else {
1373         r->uri.len = alias + path.len;
1374         r->uri.data = ngx_pnalloc(r->pool, r->uri.len);
1375         if (r->uri.data == NULL) {
1376             ngx_http_finalize_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
1377             return NGX_OK;
1378         }
1379
1380         p = ngx_copy(r->uri.data, clcf->name.data, alias);
1381         ngx_memcpy(p, name, path.len);
1382     }
1383
1384     ngx_http_set_exten(r);
1385
1386     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1387                  "try file uri: \"%V\"", &r->uri);
1388
1389     r->phase_handler++;
1390     return NGX_AGAIN;
1391 }
1392
1393 /* not reached */
1394 }
1395
1396
1397 ngx_int_t
1398 ngx_http_core_content_phase(ngx_http_request_t *r,
1399                             ngx_http_phase_handler_t *ph)
1400 {
1401     size_t    root;
1402     ngx_int_t rc;
1403     ngx_str_t path;
1404
1405     if (r->content_handler) {
1406         r->write_event_handler = ngx_http_request_empty_handler;
1407         ngx_http_finalize_request(r, r->content_handler(r));
1408         return NGX_OK;
1409     }
1410
1411     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1412                  "content phase: %ui", r->phase_handler);
1413
1414     rc = ph->handler(r);
1415
1416     if (rc != NGX_DECLINED) {
1417         ngx_http_finalize_request(r, rc);
1418         return NGX_OK;
1419     }
1420
1421     /* rc == NGX_DECLINED */
1422
1423     ph++;
1424
1425     if (ph->checker) {
1426         r->phase_handler++;
1427         return NGX_AGAIN;
1428     }
1429
1430     /* no content handler was found */
1431
1432     if (r->uri.data[r->uri.len - 1] == '/') {
1433
1434         if (ngx_http_map_uri_to_path(r, &path, &root, 0) != NULL) {
1435             ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1436                          "directory index of \"%s\" is forbidden", path.data);
1437         }
1438
1439         ngx_http_finalize_request(r, NGX_HTTP_FORBIDDEN);
1440         return NGX_OK;
1441     }

```

```

1442     ngx_log_error(NGX_LOG_ERR, r->connection->log, 0, "no handler found");
1443
1444
1445     ngx_http_finalize_request(r, NGX_HTTP_NOT_FOUND);
1446     return NGX_OK;
1447 }
1448
1449
1450 void
1451 ngx_http_update_location_config(ngx_http_request_t *r)
1452 {
1453     ngx_http_core_loc_conf_t *clcf;
1454
1455     clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
1456
1457     if (r->method & clcf->limit_except) {
1458         r->loc_conf = clcf->limit_except_loc_conf;
1459         clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
1460     }
1461
1462     if (r == r->main) {
1463         ngx_http_set_connection_log(r->connection, clcf->error_log);
1464     }
1465
1466     if ((ngx_io.flags & NGX_IO_SENDFILE) && clcf->sendfile) {
1467         r->connection->sendfile = 1;
1468     }
1469     else {
1470         r->connection->sendfile = 0;
1471     }
1472
1473     if (clcf->client_body_in_file_only) {
1474         r->request_body_in_file_only = 1;
1475         r->request_body_in_persistent_file = 1;
1476         r->request_body_in_clean_file =
1477             clcf->client_body_in_file_only == NGX_HTTP_REQUEST_BODY_FILE_CLEAN;
1478         r->request_body_file_log_level = NGX_LOG_NOTICE;
1479     }
1480     else {
1481         r->request_body_file_log_level = NGX_LOG_WARN;
1482     }
1483
1484     r->request_body_in_single_buf = clcf->client_body_in_single_buffer;
1485
1486     if (r->keepalive) {
1487         if (clcf->keepalive_timeout == 0) {
1488             r->keepalive = 0;
1489         }
1490         else if (r->connection->requests >= clcf->keepalive_requests) {
1491             r->keepalive = 0;
1492         }
1493         else if (r->headers_in.msie6
1494                 && r->method == NGX_HTTP_POST
1495                 && (clcf->keepalive_disable
1496                    & NGX_HTTP_KEEPALIVE_DISABLE_MSIE6))
1497         {
1498             /*
1499              * MSIE may wait for some time if an response for
1500              * a POST request was sent over a keepalive connection
1501              */
1502             r->keepalive = 0;
1503         }
1504         else if (r->headers_in.safari
1505                 && (clcf->keepalive_disable
1506                    & NGX_HTTP_KEEPALIVE_DISABLE_SAFARI))
1507         {
1508             /*
1509              * Safari may send a POST request to a closed keepalive
1510              * connection and may stall for some time, see
1511              * https://bugs.webkit.org/show_bug.cgi?id=5760
1512              */
1513             r->keepalive = 0;
1514         }
1515     }
1516
1517     if (!clcf->tcp_nopush) {

```

```

1518     /* disable TCP_NOPUSH/TCP_CORK use */
1519     r->connection->tcp_nopush = NGX_TCP_NOPUSH_DISABLED;
1520 }
1521
1522 if (r->limit_rate == 0) {
1523     r->limit_rate = clcf->limit_rate;
1524 }
1525
1526 if (clcf->handler) {
1527     r->content_handler = clcf->handler;
1528 }
1529 }
1530
1531
1532 /*
1533 * NGX_OK         - exact or regex match
1534 * NGX_DONE      - auto redirect
1535 * NGX_AGAIN     - inclusive match
1536 * NGX_ERROR     - regex error
1537 * NGX_DECLINED - no match
1538 */
1539
1540 static ngx_int_t
1541 ngx_http_core_find_location(ngx_http_request_t *r)
1542 {
1543     ngx_int_t          rc;
1544     ngx_http_core_loc_conf_t *pclcf;
1545     #if (NGX_PCRE)
1546         ngx_int_t      n;
1547         ngx_uint_t     noregex;
1548         ngx_http_core_loc_conf_t *clcf, **clcfp;
1549
1550         noregex = 0;
1551     #endif
1552
1553     pclcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
1554
1555     rc = ngx_http_core_find_static_location(r, pclcf->static_locations);
1556
1557     if (rc == NGX_AGAIN) {
1558
1559         #if (NGX_PCRE)
1560             clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
1561
1562             noregex = clcf->noregex;
1563         #endif
1564
1565         /* look up nested locations */
1566
1567         rc = ngx_http_core_find_location(r);
1568     }
1569
1570     if (rc == NGX_OK || rc == NGX_DONE) {
1571         return rc;
1572     }
1573
1574     /* rc == NGX_DECLINED or rc == NGX_AGAIN in nested location */
1575
1576     #if (NGX_PCRE)
1577
1578     if (noregex == 0 && pclcf->regex_locations) {
1579
1580         for (clcfp = pclcf->regex_locations; *clcfp; clcfp++) {
1581
1582             ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1583                 "test location: ~ \"%V\"", &(*clcfp)->name);
1584
1585             n = ngx_http_regex_exec(r, (*clcfp)->regex, &r->uri);
1586
1587             if (n == NGX_OK) {
1588                 r->loc_conf = (*clcfp)->loc_conf;
1589
1590                 /* look up nested locations */
1591
1592                 rc = ngx_http_core_find_location(r);
1593

```

```

1594         return (rc == NGX_ERROR) ? rc : NGX_OK;
1595     }
1596
1597     if (n == NGX_DECLINED) {
1598         continue;
1599     }
1600
1601     return NGX_ERROR;
1602 }
1603 }
1604 #endif
1605
1606     return rc;
1607 }
1608
1609
1610 /*
1611 * NGX_OK         - exact match
1612 * NGX_DONE      - auto redirect
1613 * NGX_AGAIN     - inclusive match
1614 * NGX_DECLINED - no match
1615 */
1616
1617 static ngx_int_t
1618 ngx_http_core_find_static_location(ngx_http_request_t *r,
1619 ngx_http_location_tree_node_t *node)
1620 {
1621     u_char      *uri;
1622     size_t      len, n;
1623     ngx_int_t  rc, rv;
1624
1625     len = r->uri.len;
1626     uri = r->uri.data;
1627
1628     rv = NGX_DECLINED;
1629
1630     for ( ;; ) {
1631
1632         if (node == NULL) {
1633             return rv;
1634         }
1635
1636         ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1637             "test location: \"%*s\"", node->len, node->name);
1638
1639         n = (len <= (size_t) node->len) ? len : node->len;
1640
1641         rc = ngx_filename_cmp(uri, node->name, n);
1642
1643         if (rc != 0) {
1644             node = (rc < 0) ? node->left : node->right;
1645
1646             continue;
1647         }
1648
1649         if (len > (size_t) node->len) {
1650
1651             if (node->inclusive) {
1652
1653                 r->loc_conf = node->inclusive->loc_conf;
1654                 rv = NGX_AGAIN;
1655
1656                 node = node->tree;
1657                 uri += n;
1658                 len -= n;
1659
1660                 continue;
1661             }
1662
1663             /* exact only */
1664
1665             node = node->right;
1666
1667             continue;
1668         }
1669

```



```

1670     if (len == (size_t) node->len) {
1671         if (node->exact) {
1672             r->loc_conf = node->exact->loc_conf;
1673             return NGX\_OK;
1674         }
1675         } else {
1676             r->loc_conf = node->inclusive->loc_conf;
1677             return NGX\_AGAIN;
1678         }
1679     }
1680 }
1681
1682 /* len < node->len */
1683
1684 if (len + 1 == (size_t) node->len && node->auto_redirect) {
1685     r->loc_conf = (node->exact) ? node->exact->loc_conf:
1686                     node->inclusive->loc_conf;
1687     rv = NGX\_DONE;
1688 }
1689
1690 node = node->left;
1691 }
1692 }
1693 }
1694
1695
1696 void *
1697 ngx\_http\_test\_content\_type(ngx\_http\_request\_t *r, ngx\_hash\_t *types_hash)
1698 {
1699     u_char    c, *lowercase;
1700     size_t    len;
1701     ngx\_uint\_t i, hash;
1702
1703     if (types_hash->size == 0) {
1704         return (void *) 4;
1705     }
1706
1707     if (r->headers_out.content_type.len == 0) {
1708         return NULL;
1709     }
1710
1711     len = r->headers_out.content_type_len;
1712
1713     if (r->headers_out.content_type_lowercase == NULL) {
1714         lowercase = ngx\_pnalloc(r->pool, len);
1715         if (lowercase == NULL) {
1716             return NULL;
1717         }
1718     }
1719
1720     r->headers_out.content_type_lowercase = lowercase;
1721
1722     hash = 0;
1723
1724     for (i = 0; i < len; i++) {
1725         c = ngx\_tolower(r->headers_out.content_type.data[i]);
1726         hash = ngx\_hash(hash, c);
1727         lowercase[i] = c;
1728     }
1729
1730     r->headers_out.content_type_hash = hash;
1731 }
1732
1733 return ngx\_hash\_find(types_hash, r->headers_out.content_type_hash,
1734                     r->headers_out.content_type_lowercase, len);
1735 }
1736
1737
1738 ngx\_int\_t
1739 ngx\_http\_set\_content\_type(ngx\_http\_request\_t *r)
1740 {
1741     u_char    c, *exten;
1742     ngx\_str\_t *type;
1743     ngx\_uint\_t i, hash;
1744     ngx\_http\_core\_loc\_conf\_t *clcf;
1745

```

```

1746     if (r->headers_out.content_type.len) {
1747         return NGX\_OK;
1748     }
1749
1750     clcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_core\_module);
1751
1752     if (r->exten.len) {
1753
1754         hash = 0;
1755
1756         for (i = 0; i < r->exten.len; i++) {
1757             c = r->exten.data[i];
1758
1759             if (c >= 'A' && c <= 'Z') {
1760
1761                 exten = ngx\_pnalloc(r->pool, r->exten.len);
1762                 if (exten == NULL) {
1763                     return NGX\_ERROR;
1764                 }
1765
1766                 hash = ngx\_hash\_strlow(exten, r->exten.data, r->exten.len);
1767
1768                 r->exten.data = exten;
1769
1770                 break;
1771             }
1772
1773             hash = ngx\_hash(hash, c);
1774         }
1775
1776         type = ngx\_hash\_find(&clcf->types_hash, hash,
1777                             r->exten.data, r->exten.len);
1778
1779         if (type) {
1780             r->headers_out.content_type_len = type->len;
1781             r->headers_out.content_type = *type;
1782
1783             return NGX\_OK;
1784         }
1785     }
1786
1787     r->headers_out.content_type_len = clcf->default_type.len;
1788     r->headers_out.content_type = clcf->default_type;
1789
1790     return NGX\_OK;
1791 }
1792
1793
1794 void
1795 ngx\_http\_set\_exten(ngx\_http\_request\_t *r)
1796 {
1797     ngx\_int\_t i;
1798
1799     ngx\_str\_null(&r->exten);
1800
1801     for (i = r->uri.len - 1; i > 1; i--) {
1802         if (r->uri.data[i] == '.' && r->uri.data[i - 1] != '/') {
1803
1804             r->exten.len = r->uri.len - i - 1;
1805             r->exten.data = &r->uri.data[i + 1];
1806
1807             return;
1808
1809         } else if (r->uri.data[i] == '/') {
1810             return;
1811         }
1812     }
1813
1814     return;
1815 }
1816
1817
1818 ngx\_int\_t
1819 ngx\_http\_set\_etag(ngx\_http\_request\_t *r)
1820 {
1821     ngx\_table\_elt\_t *etag;

```

```

1822 ngx\_http\_core\_loc\_conf\_t *clcf;
1823
1824 clcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_core\_module);
1825
1826 if (!clcf->etag) {
1827     return NGX\_OK;
1828 }
1829
1830 etag = ngx\_list\_push(&r->headers_out.headers);
1831 if (etag == NULL) {
1832     return NGX\_ERROR;
1833 }
1834
1835 etag->hash = 1;
1836 ngx\_str\_set(&etag->key, "ETag");
1837
1838 etag->value.data = ngx\_pnalloc(r->pool, NGX\_OFF\_T\_LEN + NGX\_TIME\_T\_LEN + 3);
1839 if (etag->value.data == NULL) {
1840     etag->hash = 0;
1841     return NGX\_ERROR;
1842 }
1843
1844 etag->value.len = ngx\_sprintf(etag->value.data, "\\%xT-%x0\\",
1845                               r->headers_out.last_modified_time,
1846                               r->headers_out.content_length_n)
1847     - etag->value.data;
1848
1849 r->headers_out.etag = etag;
1850
1851 return NGX\_OK;
1852 }
1853
1854
1855 void
1856 ngx\_http\_weak\_etag(ngx\_http\_request\_t *r)
1857 {
1858     size\_t          len;
1859     u\_char         *p;
1860     ngx\_table\_elt\_t *etag;
1861
1862     etag = r->headers_out.etag;
1863
1864     if (etag == NULL) {
1865         return;
1866     }
1867
1868     if (etag->value.len > 2
1869         && etag->value.data[0] == 'W'
1870         && etag->value.data[1] == '/')
1871     {
1872         return;
1873     }
1874
1875     if (etag->value.len < 1 || etag->value.data[0] != '') {
1876         r->headers_out.etag->hash = 0;
1877         r->headers_out.etag = NULL;
1878         return;
1879     }
1880
1881     p = ngx\_pnalloc(r->pool, etag->value.len + 2);
1882     if (p == NULL) {
1883         r->headers_out.etag->hash = 0;
1884         r->headers_out.etag = NULL;
1885         return;
1886     }
1887
1888     len = ngx\_sprintf(p, "W/%V", &etag->value) - p;
1889
1890     etag->value.data = p;
1891     etag->value.len = len;
1892 }
1893
1894
1895 ngx\_int\_t
1896 ngx\_http\_send\_response(ngx\_http\_request\_t *r, ngx\_uint\_t status,
1897 ngx\_str\_t *ct, ngx\_http\_complex\_value\_t *cv)

```

```

1898 {
1899     ngx_int_t    rc;
1900     ngx_str_t    val;
1901     ngx_buf_t    *b;
1902     ngx_chain_t  out;
1903
1904     if (ngx_http_discard_request_body(r) != NGX_OK) {
1905         return NGX_HTTP_INTERNAL_SERVER_ERROR;
1906     }
1907
1908     r->headers_out.status = status;
1909
1910     if (ngx_http_complex_value(r, cv, &val) != NGX_OK) {
1911         return NGX_HTTP_INTERNAL_SERVER_ERROR;
1912     }
1913
1914     if (status == NGX_HTTP_MOVED_PERMANENTLY
1915         || status == NGX_HTTP_MOVED_TEMPORARILY
1916         || status == NGX_HTTP_SEE_OTHER
1917         || status == NGX_HTTP_TEMPORARY_REDIRECT)
1918     {
1919         ngx_http_clear_location(r);
1920
1921         r->headers_out.location = ngx_list_push(&r->headers_out.headers);
1922         if (r->headers_out.location == NULL) {
1923             return NGX_HTTP_INTERNAL_SERVER_ERROR;
1924         }
1925
1926         r->headers_out.location->hash = 1;
1927         ngx_str_set(&r->headers_out.location->key, "Location");
1928         r->headers_out.location->value = val;
1929
1930         return status;
1931     }
1932
1933     r->headers_out.content_length_n = val.len;
1934
1935     if (ct) {
1936         r->headers_out.content_type_len = ct->len;
1937         r->headers_out.content_type = *ct;
1938     }
1939     else {
1940         if (ngx_http_set_content_type(r) != NGX_OK) {
1941             return NGX_HTTP_INTERNAL_SERVER_ERROR;
1942         }
1943     }
1944
1945     if (r->method == NGX_HTTP_HEAD || (r != r->main && val.len == 0)) {
1946         return ngx_http_send_header(r);
1947     }
1948
1949     b = ngx_palloc(r->pool, sizeof(ngx_buf_t));
1950     if (b == NULL) {
1951         return NGX_HTTP_INTERNAL_SERVER_ERROR;
1952     }
1953
1954     b->pos = val.data;
1955     b->last = val.data + val.len;
1956     b->memory = val.len ? 1 : 0;
1957     b->last_buf = (r == r->main) ? 1 : 0;
1958     b->last_in_chain = 1;
1959
1960     out.buf = b;
1961     out.next = NULL;
1962
1963     rc = ngx_http_send_header(r);
1964
1965     if (rc == NGX_ERROR || rc > NGX_OK || r->header_only) {
1966         return rc;
1967     }
1968
1969     return ngx_http_output_filter(r, &out);
1970 }
1971
1972
1973 ngx_int_t

```

```

1974 ngx_http_send_header(ngx_http_request_t *r)
1975 {
1976     if (r->post_action) {
1977         return NGX_OK;
1978     }
1979
1980     if (r->header_sent) {
1981         ngx_log_error(NGX_LOG_ALERT, r->connection->log, 0,
1982             "header already sent");
1983         return NGX_ERROR;
1984     }
1985
1986     if (r->err_status) {
1987         r->headers_out.status = r->err_status;
1988         r->headers_out.status_line.len = 0;
1989     }
1990
1991     return ngx_http_top_header_filter(r);
1992 }
1993
1994
1995 ngx_int_t
1996 ngx_http_output_filter(ngx_http_request_t *r, ngx_chain_t *in)
1997 {
1998     ngx_int_t      rc;
1999     ngx_connection_t *c;
2000
2001     c = r->connection;
2002
2003     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
2004         "http output filter \"%V?%V\"", &r->uri, &r->args);
2005
2006     rc = ngx_http_top_body_filter(r, in);
2007
2008     if (rc == NGX_ERROR) {
2009         /* NGX_ERROR may be returned by any filter */
2010         c->error = 1;
2011     }
2012
2013     return rc;
2014 }
2015
2016
2017 u_char *
2018 ngx_http_map_uri_to_path(ngx_http_request_t *r, ngx_str_t *path,
2019     size_t *root_length, size_t reserved)
2020 {
2021     u_char          *last;
2022     size_t          alias;
2023     ngx_http_core_loc_conf_t *clcf;
2024
2025     clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
2026
2027     alias = clcf->alias;
2028
2029     if (alias && !r->valid_location) {
2030         ngx_log_error(NGX_LOG_ALERT, r->connection->log, 0,
2031             "\"alias\" cannot be used in location \"%V\" "
2032             "where URI was rewritten", &clcf->name);
2033         return NULL;
2034     }
2035
2036     if (clcf->root_lengths == NULL) {
2037
2038         *root_length = clcf->root.len;
2039
2040         path->len = clcf->root.len + reserved + r->uri.len - alias + 1;
2041
2042         path->data = ngx_pnalloc(r->pool, path->len);
2043         if (path->data == NULL) {
2044             return NULL;
2045         }
2046
2047         last = ngx_copy(path->data, clcf->root.data, clcf->root.len);
2048
2049     } else {

```

```

2050     if (alias == NGX_MAX_SIZE_T_VALUE) {
2051         reserved += r->add_uri_to_alias ? r->uri.len + 1 : 1;
2052     } else {
2053         reserved += r->uri.len - alias + 1;
2054     }
2055 }
2056
2057 if (ngx_http_script_run(r, path, clcf->root_lengths->elts, reserved,
2058     clcf->root_values->elts)
2059     == NULL)
2060 {
2061     return NULL;
2062 }
2063
2064 if (ngx_get_full_name(r->pool, (ngx_str_t *) &ngx_cycle->prefix, path)
2065     != NGX_OK)
2066 {
2067     return NULL;
2068 }
2069
2070 *root_length = path->len - reserved;
2071 last = path->data + *root_length;
2072
2073 if (alias == NGX_MAX_SIZE_T_VALUE) {
2074     if (!r->add_uri_to_alias) {
2075         *last = '\\0';
2076         return last;
2077     }
2078 }
2079
2080 alias = 0;
2081 }
2082 }
2083
2084 last = ngx_cpystrn(last, r->uri.data + alias, r->uri.len - alias + 1);
2085
2086 return last;
2087 }
2088
2089

```

#### ngx\_int\_t

```

2091 ngx_http_auth_basic_user(ngx_http_request_t *r)
2092 {
2093     ngx_str_t  auth, encoded;
2094     ngx_uint_t len;
2095
2096     if (r->headers_in.user.len == 0 && r->headers_in.user.data != NULL) {
2097         return NGX_DECLINED;
2098     }
2099
2100     if (r->headers_in.authorization == NULL) {
2101         r->headers_in.user.data = (u_char *) "";
2102         return NGX_DECLINED;
2103     }
2104
2105     encoded = r->headers_in.authorization->value;
2106
2107     if (encoded.len < sizeof("Basic ") - 1
2108         || ngx_strncascmp(encoded.data, (u_char *) "Basic ",
2109             sizeof("Basic ") - 1)
2110         != 0)
2111     {
2112         r->headers_in.user.data = (u_char *) "";
2113         return NGX_DECLINED;
2114     }
2115
2116     encoded.len -= sizeof("Basic ") - 1;
2117     encoded.data += sizeof("Basic ") - 1;
2118
2119     while (encoded.len && encoded.data[0] == ' ') {
2120         encoded.len--;
2121         encoded.data++;
2122     }
2123
2124     if (encoded.len == 0) {
2125         r->headers_in.user.data = (u_char *) "";

```

```

2126     return NGX\_DECLINED;
2127 }
2128
2129 auth.len = ngx\_base64\_decoded\_length(encoded.len);
2130 auth.data = ngx\_pnalloc(r->pool, auth.len + 1);
2131 if (auth.data == NULL) {
2132     return NGX\_ERROR;
2133 }
2134
2135 if (ngx\_decode\_base64(&auth, &encoded) != NGX\_OK) {
2136     r->headers_in.user.data = (u_char *) "";
2137     return NGX\_DECLINED;
2138 }
2139
2140 auth.data[auth.len] = '\0';
2141
2142 for (len = 0; len < auth.len; len++) {
2143     if (auth.data[len] == ':') {
2144         break;
2145     }
2146 }
2147
2148 if (len == 0 || len == auth.len) {
2149     r->headers_in.user.data = (u_char *) "";
2150     return NGX\_DECLINED;
2151 }
2152
2153 r->headers_in.user.len = len;
2154 r->headers_in.user.data = auth.data;
2155 r->headers_in.passwd.len = auth.len - len - 1;
2156 r->headers_in.passwd.data = &auth.data[len + 1];
2157
2158 return NGX\_OK;
2159 }
2160
2161
2162 #if (NGX_HTTP_GZIP)
2163
2164 ngx\_int\_t
2165 ngx\_http\_gzip\_ok(ngx\_http\_request\_t *r)
2166 {
2167     time\_t                date, expires;
2168     ngx\_uint\_t           p;
2169     ngx\_array\_t         *cc;
2170     ngx\_table\_elt\_t     *e, *d, *ae;
2171     ngx\_http\_core\_loc\_conf\_t *clcf;
2172
2173     r->gzip_tested = 1;
2174
2175     if (r != r->main) {
2176         return NGX\_DECLINED;
2177     }
2178
2179     #if (NGX_HTTP_SPDY)
2180     if (r->spdy_stream) {
2181         r->gzip_ok = 1;
2182         return NGX\_OK;
2183     }
2184     #endif
2185
2186     ae = r->headers_in.accept_encoding;
2187     if (ae == NULL) {
2188         return NGX\_DECLINED;
2189     }
2190
2191     if (ae->value.len < sizeof("gzip") - 1) {
2192         return NGX\_DECLINED;
2193     }
2194
2195     /*
2196     * test first for the most common case "gzip,...":
2197     * MSIE:    "gzip, deflate"
2198     * Firefox: "gzip, deflate"
2199     * Chrome:  "gzip, deflate, sdch"
2200     * Safari:  "gzip, deflate"
2201     * Opera:   "gzip, deflate"

```





```

2278     {
2279         goto ok;
2280     }
2281
2282     if ((p & NGX\_HTTP\_GZIP\_PROXIED\_NO\_STORE)
2283         && ngx\_http\_parse\_multi\_header\_lines(cc, &ngx\_http\_gzip\_no\_store,
2284                                             NULL)
2285         >= 0)
2286     {
2287         goto ok;
2288     }
2289
2290     if ((p & NGX\_HTTP\_GZIP\_PROXIED\_PRIVATE)
2291         && ngx\_http\_parse\_multi\_header\_lines(cc, &ngx\_http\_gzip\_private,
2292                                             NULL)
2293         >= 0)
2294     {
2295         goto ok;
2296     }
2297
2298     return NGX\_DECLINED;
2299 }
2300
2301 if ((p & NGX\_HTTP\_GZIP\_PROXIED\_NO\_LM) && r->headers_out.last_modified) {
2302     return NGX\_DECLINED;
2303 }
2304
2305 if ((p & NGX\_HTTP\_GZIP\_PROXIED\_NO\_ETAG) && r->headers_out.etag) {
2306     return NGX\_DECLINED;
2307 }
2308
2309 ok:
2310
2311 #if (NGX\_PCRE)
2312
2313     if (clcf->gzip_disable && r->headers_in.user_agent) {
2314
2315         if (ngx\_regex\_exec\_array(clcf->gzip_disable,
2316                                 &r->headers_in.user_agent->value,
2317                                 r->connection->log)
2318             != NGX\_DECLINED)
2319         {
2320             return NGX\_DECLINED;
2321         }
2322     }
2323
2324 #endif
2325
2326     r->gzip_ok = 1;
2327
2328     return NGX\_OK;
2329 }
2330
2331
2332 /*
2333  * gzip is enabled for the following quantities:
2334  *   "gzip; q=0.001" ... "gzip; q=1.000"
2335  * gzip is disabled for the following quantities:
2336  *   "gzip; q=0" ... "gzip; q=0.000", and for any invalid cases
2337  */
2338
2339 static ngx\_int\_t
2340 ngx\_http\_gzip\_accept\_encoding(ngx\_str\_t *ae)
2341 {
2342     u_char *p, *start, *last;
2343
2344     start = ae->data;
2345     last = start + ae->len;
2346
2347     for ( ;; ) {
2348         p = ngx\_strcasestrn(start, "gzip", 4 - 1);
2349         if (p == NULL) {
2350             return NGX\_DECLINED;
2351         }
2352
2353         if (p == start || (*(p - 1) == ',' || *(p - 1) == ' ')) {

```

```

2354         break;
2355     }
2356
2357     start = p + 4;
2358 }
2359
2360 p += 4;
2361
2362 while (p < last) {
2363     switch (*p++) {
2364         case ',':
2365             return NGX\_OK;
2366         case ';':
2367             goto quantity;
2368         case ' ':
2369             continue;
2370         default:
2371             return NGX\_DECLINED;
2372     }
2373 }
2374
2375 return NGX\_OK;
2376
2377 quantity:
2378
2379     while (p < last) {
2380         switch (*p++) {
2381             case 'q':
2382             case 'Q':
2383                 goto equal;
2384             case ' ':
2385                 continue;
2386             default:
2387                 return NGX\_DECLINED;
2388         }
2389     }
2390
2391     return NGX\_OK;
2392
2393 equal:
2394
2395     if (p + 2 > last || *p++ != '=') {
2396         return NGX\_DECLINED;
2397     }
2398
2399     if (ngx\_http\_gzip\_quantity(p, last) == 0) {
2400         return NGX\_DECLINED;
2401     }
2402
2403     return NGX\_OK;
2404 }
2405
2406
2407 static ngx\_uint\_t
2408 ngx\_http\_gzip\_quantity(u_char *p, u_char *last)
2409 {
2410     u_char    c;
2411     ngx\_uint\_t  n, q;
2412
2413     c = *p++;
2414
2415     if (c != '0' && c != '1') {
2416         return 0;
2417     }
2418
2419     q = (c - '0') * 100;
2420
2421     if (p == last) {
2422         return q;
2423     }
2424
2425     c = *p++;
2426
2427     if (c == ',' || c == ' ') {
2428         return q;
2429     }

```

```

2430
2431     if (c != '.') {
2432         return 0;
2433     }
2434
2435     n = 0;
2436
2437     while (p < last) {
2438         c = *p++;
2439
2440         if (c == ',' || c == ' ') {
2441             break;
2442         }
2443
2444         if (c >= '0' && c <= '9') {
2445             q += c - '0';
2446             n++;
2447             continue;
2448         }
2449
2450         return 0;
2451     }
2452
2453     if (q > 100 || n > 3) {
2454         return 0;
2455     }
2456
2457     return q;
2458 }
2459
2460 #endif
2461
2462
2463 ngx_int_t
2464 ngx_http_subrequest(ngx_http_request_t *r,
2465 ngx_str_t *uri, ngx_str_t *args, ngx_http_request_t **psr,
2466 ngx_http_post_subrequest_t *ps, ngx_uint_t flags)
2467 {
2468     ngx_time_t          *tp;
2469     ngx_connection_t   *c;
2470     ngx_http_request_t *sr;
2471     ngx_http_core_srv_conf_t *cscf;
2472     ngx_http_postponed_request_t *pr, *p;
2473
2474     r->main->subrequests--;
2475
2476     if (r->main->subrequests == 0) {
2477         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
2478             "subrequests cycle while processing \"%V\"", uri);
2479         r->main->subrequests = 1;
2480         return NGX_ERROR;
2481     }
2482
2483     sr = ngx_palloc(r->pool, sizeof(ngx_http_request_t));
2484     if (sr == NULL) {
2485         return NGX_ERROR;
2486     }
2487
2488     sr->signature = NGX_HTTP_MODULE;
2489
2490     c = r->connection;
2491     sr->connection = c;
2492
2493     sr->ctx = ngx_palloc(r->pool, sizeof(void *) * ngx_http_max_module);
2494     if (sr->ctx == NULL) {
2495         return NGX_ERROR;
2496     }
2497
2498     if (ngx_list_init(&sr->headers_out.headers, r->pool, 20,
2499         sizeof(ngx_table_elt_t))
2500         != NGX_OK)
2501     {
2502         return NGX_ERROR;
2503     }
2504
2505     cscf = ngx_http_get_module_srv_conf(r, ngx_http_core_module);

```

```

2506 sr->main_conf = cscf->ctx->main_conf;
2507 sr->srv_conf = cscf->ctx->srv_conf;
2508 sr->loc_conf = cscf->ctx->loc_conf;
2509
2510 sr->pool = r->pool;
2511
2512 sr->headers_in = r->headers_in;
2513
2514 ngx_http_clear_content_length(sr);
2515 ngx_http_clear_accept_ranges(sr);
2516 ngx_http_clear_last_modified(sr);
2517
2518 sr->request_body = r->request_body;
2519
2520 #if (NGX_HTTP_SPDY)
2521 sr->spdy_stream = r->spdy_stream;
2522 #endif
2523
2524 sr->method = NGX_HTTP_GET;
2525 sr->http_version = r->http_version;
2526
2527 sr->request_line = r->request_line;
2528 sr->uri = *uri;
2529
2530 if (args) {
2531     sr->args = *args;
2532 }
2533
2534 ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
2535               "http subrequest \"%V?%V\"", uri, &sr->args);
2536
2537 sr->subrequest_in_memory = (flags & NGX_HTTP_SUBREQUEST_IN_MEMORY) != 0;
2538 sr->waited = (flags & NGX_HTTP_SUBREQUEST_WAITED) != 0;
2539
2540 sr->unparsed_uri = r->unparsed_uri;
2541 sr->method_name = ngx_http_core_get_method;
2542 sr->http_protocol = r->http_protocol;
2543
2544 ngx_http_set_exten(sr);
2545
2546 sr->main = r->main;
2547 sr->parent = r;
2548 sr->post_subrequest = ps;
2549 sr->read_event_handler = ngx_http_request_empty_handler;
2550 sr->write_event_handler = ngx_http_handler;
2551
2552 if (c->data == r && r->postponed == NULL) {
2553     c->data = sr;
2554 }
2555
2556 sr->variables = r->variables;
2557
2558 sr->log_handler = r->log_handler;
2559
2560 pr = ngx_palloc(r->pool, sizeof(ngx_http_postponed_request_t));
2561 if (pr == NULL) {
2562     return NGX_ERROR;
2563 }
2564
2565 pr->request = sr;
2566 pr->out = NULL;
2567 pr->next = NULL;
2568
2569 if (r->postponed) {
2570     for (p = r->postponed; p->next; p = p->next) { /* void */ }
2571     p->next = pr;
2572 } else {
2573     r->postponed = pr;
2574 }
2575
2576 sr->internal = 1;
2577
2578 sr->discard_body = r->discard_body;
2579 sr->expect_tested = 1;
2580 sr->main_filter_need_in_memory = r->main_filter_need_in_memory;

```

```

2582     sr->uri_changes = NGX\_HTTP\_MAX\_URI\_CHANGES + 1;
2583
2584
2585     tp = ngx\_timeofday();
2586     sr->start_sec = tp->sec;
2587     sr->start_msec = tp->msec;
2588
2589     r->main->count++;
2590
2591     *psr = sr;
2592
2593     return ngx\_http\_post\_request(sr, NULL);
2594 }
2595
2596
2597 ngx\_int\_t
2598 ngx\_http\_internal\_redirect(ngx\_http\_request\_t *r,
2599     ngx\_str\_t *uri, ngx\_str\_t *args)
2600 {
2601     ngx\_http\_core\_srv\_conf\_t *cscf;
2602
2603     r->uri_changes--;
2604
2605     if (r->uri_changes == 0) {
2606         ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, 0,
2607             "rewrite or internal redirection cycle "
2608             "while internally redirecting to \"%V\"", uri);
2609
2610         r->main->count++;
2611         ngx\_http\_finalize\_request(r, NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);
2612         return NGX\_DONE;
2613     }
2614
2615     r->uri = *uri;
2616
2617     if (args) {
2618         r->args = *args;
2619     } else {
2620         ngx\_str\_null(&r->args);
2621     }
2622
2623
2624     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
2625         "internal redirect: \"%V?%V\"", uri, &r->args);
2626
2627     ngx\_http\_set\_exten(r);
2628
2629     /* clear the modules contexts */
2630     ngx\_memzero(r->ctx, sizeof(void *) * ngx\_http\_max\_module);
2631
2632     cscf = ngx\_http\_get\_module\_srv\_conf(r, ngx\_http\_core\_module);
2633     r->loc_conf = cscf->ctx->loc_conf;
2634
2635     ngx\_http\_update\_location\_config(r);
2636
2637     #if (NGX\_HTTP\_CACHE)
2638     r->cache = NULL;
2639 #endif
2640
2641     r->internal = 1;
2642     r->valid_unparsed_uri = 0;
2643     r->add_uri_to_alias = 0;
2644     r->main->count++;
2645
2646     ngx\_http\_handler(r);
2647
2648     return NGX\_DONE;
2649 }
2650
2651
2652 ngx\_int\_t
2653 ngx\_http\_named\_location(ngx\_http\_request\_t *r, ngx\_str\_t *name)
2654 {
2655     ngx\_http\_core\_srv\_conf\_t *cscf;
2656     ngx\_http\_core\_loc\_conf\_t **clcfp;
2657     ngx\_http\_core\_main\_conf\_t *cmcf;

```

```

2658     r->main->count++;
2659     r->uri_changes--;
2660
2661
2662     if (r->uri_changes == 0) {
2663         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
2664             "rewrite or internal redirection cycle "
2665             "while redirect to named location \"%V\"", name);
2666
2667         ngx_http_finalize_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
2668         return NGX_DONE;
2669     }
2670
2671     if (r->uri.len == 0) {
2672         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
2673             "empty URI in redirect to named location \"%V\"", name);
2674
2675         ngx_http_finalize_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
2676         return NGX_DONE;
2677     }
2678
2679     cscf = ngx_http_get_module_srv_conf(r, ngx_http_core_module);
2680
2681     if (cscf->named_locations) {
2682         for (clcfp = cscf->named_locations; *clcfp; clcfp++) {
2683             ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
2684                 "test location: \"%V\"", &(*clcfp)->name);
2685
2686             if (name->len != (*clcfp)->name.len
2687                 || ngx_strncmp(name->data, (*clcfp)->name.data, name->len) != 0)
2688             {
2689                 continue;
2690             }
2691
2692             ngx_log_debug3(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
2693                 "using location: %V \"%V?%V\"",
2694                 name, &r->uri, &r->args);
2695
2696             r->internal = 1;
2697             r->content_handler = NULL;
2698             r->uri_changed = 0;
2699             r->loc_conf = (*clcfp)->loc_conf;
2700
2701             /* clear the modules contexts */
2702             ngx_memzero(r->ctx, sizeof(void *) * ngx_http_max_module);
2703
2704             ngx_http_update_location_config(r);
2705
2706             cmcf = ngx_http_get_module_main_conf(r, ngx_http_core_module);
2707
2708             r->phase_handler = cmcf->phase_engine.location_rewrite_index;
2709
2710             r->write_event_handler = ngx_http_core_run_phases;
2711             ngx_http_core_run_phases(r);
2712
2713             return NGX_DONE;
2714         }
2715     }
2716
2717     ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
2718         "could not find named location \"%V\"", name);
2719
2720     ngx_http_finalize_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
2721
2722     return NGX_DONE;
2723 }
2724
2725
2726
2727
2728 ngx_http_cleanup_t *
2729 ngx_http_cleanup_add(ngx_http_request_t *r, size_t size)
2730 {
2731     ngx_http_cleanup_t *cln;
2732
2733     r = r->main;

```

```

2734     cln = ngx_palloc(r->pool, sizeof(ngx_http_cleanup_t));
2735     if (cln == NULL) {
2736         return NULL;
2737     }
2738
2739
2740     if (size) {
2741         cln->data = ngx_palloc(r->pool, size);
2742         if (cln->data == NULL) {
2743             return NULL;
2744         }
2745
2746     } else {
2747         cln->data = NULL;
2748     }
2749
2750     cln->handler = NULL;
2751     cln->next = r->cleanup;
2752
2753     r->cleanup = cln;
2754
2755     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
2756                  "http cleanup add: %p", cln);
2757
2758     return cln;
2759 }
2760
2761
2762 ngx_int_t
2763 ngx_http_set_disable_symlinks(ngx_http_request_t *r,
2764                               ngx_http_core_loc_conf_t *clcf, ngx_str_t *path, ngx_open_file_info_t *of)
2765 {
2766     #if (NGX_HAVE_OPENAT)
2767         u_char      *p;
2768         ngx_str_t    from;
2769
2770         of->disable_symlinks = clcf->disable_symlinks;
2771
2772         if (clcf->disable_symlinks_from == NULL) {
2773             return NGX_OK;
2774         }
2775
2776         if (ngx_http_complex_value(r, clcf->disable_symlinks_from, &from)
2777             != NGX_OK)
2778         {
2779             return NGX_ERROR;
2780         }
2781
2782         if (from.len == 0
2783             || from.len > path->len
2784             || ngx_memcmp(path->data, from.data, from.len) != 0)
2785         {
2786             return NGX_OK;
2787         }
2788
2789         if (from.len == path->len) {
2790             of->disable_symlinks = NGX_DISABLE_SYMLINKS_OFF;
2791             return NGX_OK;
2792         }
2793
2794         p = path->data + from.len;
2795
2796         if (*p == '/') {
2797             of->disable_symlinks_from = from.len;
2798             return NGX_OK;
2799         }
2800
2801         p--;
2802
2803         if (*p == '/') {
2804             of->disable_symlinks_from = from.len - 1;
2805         }
2806     #endif
2807
2808     return NGX_OK;
2809 }

```

```

2810
2811
2812 ngx_int_t
2813 ngx_http_get_forwarded_addr(ngx_http_request_t *r, ngx_addr_t *addr,
2814 ngx_array_t *headers, ngx_str_t *value, ngx_array_t *proxies,
2815 int recursive)
2816 {
2817     ngx_int_t          rc;
2818     ngx_uint_t        i, found;
2819     ngx_table_elt_t   **h;
2820
2821     if (headers == NULL) {
2822         return ngx_http_get_forwarded_addr_internal(r, addr, value->data,
2823                                                     value->len, proxies,
2824                                                     recursive);
2825     }
2826
2827     i = headers->nelts;
2828     h = headers->elts;
2829
2830     rc = NGX_DECLINED;
2831
2832     found = 0;
2833
2834     while (i-- > 0) {
2835         rc = ngx_http_get_forwarded_addr_internal(r, addr, h[i]->value.data,
2836                                                     h[i]->value.len, proxies,
2837                                                     recursive);
2838
2839         if (!recursive) {
2840             break;
2841         }
2842
2843         if (rc == NGX_DECLINED && found) {
2844             rc = NGX_DONE;
2845             break;
2846         }
2847
2848         if (rc != NGX_OK) {
2849             break;
2850         }
2851
2852         found = 1;
2853     }
2854
2855     return rc;
2856 }
2857
2858
2859 static ngx_int_t
2860 ngx_http_get_forwarded_addr_internal(ngx_http_request_t *r, ngx_addr_t *addr,
2861 u_char *xff, size_t xfflen, ngx_array_t *proxies, int recursive)
2862 {
2863     u_char          *p;
2864     in_addr_t      inaddr;
2865     ngx_int_t      rc;
2866     ngx_addr_t     paddr;
2867     ngx_cidr_t     *caddr;
2868     ngx_uint_t     family, i;
2869     #if (NGX_HAVE_INET6)
2870     ngx_uint_t     n;
2871     struct in6_addr *inaddr6;
2872     #endif
2873
2874     #if (NGX_SUPPRESS_WARN)
2875     inaddr = 0;
2876     #if (NGX_HAVE_INET6)
2877     inaddr6 = NULL;
2878     #endif
2879     #endif
2880
2881     family = addr->sockaddr->sa_family;
2882
2883     if (family == AF_INET) {
2884         inaddr = ((struct sockaddr_in *) addr->sockaddr)->sin_addr.s_addr;
2885     }

```



```

2886
2887 #if (NGX_HAVE_INET6)
2888     else if (family == AF_INET6) {
2889         inaddr6 = &((struct sockaddr_in6 *) addr->sockaddr)->sin6_addr;
2890
2891         if (IN6_IS_ADDR_V4MAPPED(inaddr6)) {
2892             family = AF_INET;
2893
2894             p = inaddr6->s6_addr;
2895
2896             inaddr = p[12] << 24;
2897             inaddr += p[13] << 16;
2898             inaddr += p[14] << 8;
2899             inaddr += p[15];
2900
2901             inaddr = htonl(inaddr);
2902         }
2903     }
2904 #endif
2905
2906     for (cldr = proxies->elts, i = 0; i < proxies->nelts; i++) {
2907         if (cldr[i].family != family) {
2908             goto next;
2909         }
2910
2911         switch (family) {
2912
2913         #if (NGX_HAVE_INET6)
2914             case AF_INET6:
2915                 for (n = 0; n < 16; n++) {
2916                     if ((inaddr6->s6_addr[n] & cldr[i].u.in6.mask.s6_addr[n])
2917                         != cldr[i].u.in6.addr.s6_addr[n])
2918                         {
2919                             goto next;
2920                         }
2921                     }
2922                 break;
2923         #endif
2924
2925         #if (NGX_HAVE_UNIX_DOMAIN)
2926             case AF_UNIX:
2927                 break;
2928         #endif
2929
2930         default: /* AF_INET */
2931             if ((inaddr & cldr[i].u.in.mask) != cldr[i].u.in.addr) {
2932                 goto next;
2933             }
2934             break;
2935         }
2936
2937         for (p = xff + xfflen - 1; p > xff; p--, xfflen--) {
2938             if (*p != ' ' && *p != ',') {
2939                 break;
2940             }
2941         }
2942
2943         for ( /* void */ ; p > xff; p--) {
2944             if (*p == ' ' || *p == ',') {
2945                 p++;
2946                 break;
2947             }
2948         }
2949
2950         if (ngx_parse_addr(r->pool, &paddr, p, xfflen - (p - xff)) != NGX_OK) {
2951             return NGX_DECLINED;
2952         }
2953
2954         *addr = paddr;
2955
2956         if (recursive && p > xff) {
2957             rc = ngx_http_get_forwarded_addr_internal(r, addr, xff, p - 1 - xff,
2958                 proxies, 1);
2959
2960             if (rc == NGX_DECLINED) {
2961                 return NGX_DONE;

```

```

2962     }
2963
2964     /* rc == NGX_OK || rc == NGX_DONE */
2965     return rc;
2966 }
2967
2968     return NGX_OK;
2969
2970 next:
2971     continue;
2972 }
2973
2974     return NGX_DECLINED;
2975 }
2976
2977
2978 static char *
2979 ngx_http_core_server(ngx_conf_t *cf, ngx_command_t *cmd, void *dummy)
2980 {
2981     char                *rv;
2982     void                *mconf;
2983     ngx_uint_t         i;
2984     ngx_conf_t       pcf;
2985     ngx_http_module_t *module;
2986     struct sockaddr_in  *sin;
2987     ngx_http_conf_ctx_t *ctx, *http_ctx;
2988     ngx_http_listen_opt_t lsopt;
2989     ngx_http_core_srv_conf_t *cscf, **cscfp;
2990     ngx_http_core_main_conf_t *cmcf;
2991
2992     ctx = ngx_palloc(cf->pool, sizeof(ngx_http_conf_ctx_t));
2993     if (ctx == NULL) {
2994         return NGX_CONF_ERROR;
2995     }
2996
2997     http_ctx = cf->ctx;
2998     ctx->main_conf = http_ctx->main_conf;
2999
3000     /* the server{}'s srv_conf */
3001
3002     ctx->srv_conf = ngx_palloc(cf->pool, sizeof(void *) * ngx_http_max_module);
3003     if (ctx->srv_conf == NULL) {
3004         return NGX_CONF_ERROR;
3005     }
3006
3007     /* the server{}'s loc_conf */
3008
3009     ctx->loc_conf = ngx_palloc(cf->pool, sizeof(void *) * ngx_http_max_module);
3010     if (ctx->loc_conf == NULL) {
3011         return NGX_CONF_ERROR;
3012     }
3013
3014     for (i = 0; ngx_modules[i]; i++) {
3015         if (ngx_modules[i]->type != NGX_HTTP_MODULE) {
3016             continue;
3017         }
3018
3019         module = ngx_modules[i]->ctx;
3020
3021         if (module->create_srv_conf) {
3022             mconf = module->create_srv_conf(cf);
3023             if (mconf == NULL) {
3024                 return NGX_CONF_ERROR;
3025             }
3026
3027             ctx->srv_conf[ngx_modules[i]->ctx_index] = mconf;
3028         }
3029
3030         if (module->create_loc_conf) {
3031             mconf = module->create_loc_conf(cf);
3032             if (mconf == NULL) {
3033                 return NGX_CONF_ERROR;
3034             }
3035
3036             ctx->loc_conf[ngx_modules[i]->ctx_index] = mconf;
3037         }

```

```

3038     }
3039
3040
3041     /* the server configuration context */
3042
3043     cscf = ctx->srv_conf[ngx_http_core_module.ctx_index];
3044     cscf->ctx = ctx;
3045
3046
3047     cmcf = ctx->main_conf[ngx_http_core_module.ctx_index];
3048
3049     cscfp = ngx_array_push(&cmcf->servers);
3050     if (cscfp == NULL) {
3051         return NGX_CONF_ERROR;
3052     }
3053
3054     *cscfp = cscf;
3055
3056
3057     /* parse inside server{} */
3058
3059     pcf = *cf;
3060     cf->ctx = ctx;
3061     cf->cmd_type = NGX_HTTP_SRV_CONF;
3062
3063     rv = ngx_conf_parse(cf, NULL);
3064
3065     *cf = pcf;
3066
3067     if (rv == NGX_CONF_OK && !cscf->listen) {
3068         ngx_memzero(&lsopt, sizeof(ngx_http_listen_opt_t));
3069
3070         sin = &lsopt.u.sockaddr_in;
3071
3072         sin->sin_family = AF_INET;
3073 #if (NGX_WIN32)
3074         sin->sin_port = htons(80);
3075 #else
3076         sin->sin_port = htons((getuid() == 0) ? 80 : 8000);
3077 #endif
3078         sin->sin_addr.s_addr = INADDR_ANY;
3079
3080         lsopt.socklen = sizeof(struct sockaddr_in);
3081
3082         lsopt.backlog = NGX_LISTEN_BACKLOG;
3083         lsopt.rcvbuf = -1;
3084         lsopt.sndbuf = -1;
3085 #if (NGX_HAVE_SETFIB)
3086         lsopt.setfib = -1;
3087 #endif
3088 #if (NGX_HAVE_TCP_FASTOPEN)
3089         lsopt.fastopen = -1;
3090 #endif
3091         lsopt.wildcard = 1;
3092
3093         (void) ngx_sock_ntop(&lsopt.u.sockaddr, lsopt.socklen, lsopt.addr,
3094                             NGX_SOCKADDR_STRLEN, 1);
3095
3096         if (ngx_http_add_listen(cf, cscf, &lsopt) != NGX_OK) {
3097             return NGX_CONF_ERROR;
3098         }
3099     }
3100
3101     return rv;
3102 }
3103
3104
3105 static char *
3106 ngx_http_core_location(ngx_conf_t *cf, ngx_command_t *cmd, void *dummy)
3107 {
3108     char                *rv;
3109     u_char              *mod;
3110     size_t              len;
3111     ngx_str_t          *value, *name;
3112     ngx_uint_t         i;
3113     ngx_conf_t         save;

```

```

3114     ngx\_http\_module\_t *module;
3115     ngx\_http\_conf\_ctx\_t *ctx, *pctx;
3116     ngx\_http\_core\_loc\_conf\_t *clcf, *pclcf;
3117
3118     ctx = ngx\_palloc(cf->pool, sizeof(ngx\_http\_conf\_ctx\_t));
3119     if (ctx == NULL) {
3120         return NGX\_CONF\_ERROR;
3121     }
3122
3123     pctx = cf->ctx;
3124     ctx->main_conf = pctx->main_conf;
3125     ctx->srv_conf = pctx->srv_conf;
3126
3127     ctx->loc_conf = ngx\_palloc(cf->pool, sizeof(void *) * ngx\_http\_max\_module);
3128     if (ctx->loc_conf == NULL) {
3129         return NGX\_CONF\_ERROR;
3130     }
3131
3132     for (i = 0; ngx_modules[i]; i++) {
3133         if (ngx_modules[i]->type != NGX\_HTTP\_MODULE) {
3134             continue;
3135         }
3136
3137         module = ngx_modules[i]->ctx;
3138
3139         if (module->create_loc_conf) {
3140             ctx->loc_conf[ngx_modules[i]->ctx_index] =
3141                 module->create_loc_conf(cf);
3142             if (ctx->loc_conf[ngx_modules[i]->ctx_index] == NULL) {
3143                 return NGX\_CONF\_ERROR;
3144             }
3145         }
3146     }
3147
3148     clcf = ctx->loc_conf[ngx\_http\_core\_module.ctx_index];
3149     clcf->loc_conf = ctx->loc_conf;
3150
3151     value = cf->args->elts;
3152
3153     if (cf->args->nelts == 3) {
3154
3155         len = value[1].len;
3156         mod = value[1].data;
3157         name = &value[2];
3158
3159         if (len == 1 && mod[0] == '=') {
3160
3161             clcf->name = *name;
3162             clcf->exact_match = 1;
3163
3164         } else if (len == 2 && mod[0] == '^' && mod[1] == '~') {
3165
3166             clcf->name = *name;
3167             clcf->noregex = 1;
3168
3169         } else if (len == 1 && mod[0] == '~') {
3170
3171             if (ngx\_http\_core\_regex\_location(cf, clcf, name, 0) != NGX\_OK) {
3172                 return NGX\_CONF\_ERROR;
3173             }
3174
3175         } else if (len == 2 && mod[0] == '~' && mod[1] == '*') {
3176
3177             if (ngx\_http\_core\_regex\_location(cf, clcf, name, 1) != NGX\_OK) {
3178                 return NGX\_CONF\_ERROR;
3179             }
3180
3181         } else {
3182             ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
3183                 "invalid location modifier \"%V\"", &value[1]);
3184             return NGX\_CONF\_ERROR;
3185         }
3186
3187     } else {
3188
3189         name = &value[1];

```

```

3190     if (name->data[0] == '=') {
3191
3192         clcf->name.len = name->len - 1;
3193         clcf->name.data = name->data + 1;
3194         clcf->exact_match = 1;
3195
3196     } else if (name->data[0] == '^' && name->data[1] == '~') {
3197
3198         clcf->name.len = name->len - 2;
3199         clcf->name.data = name->data + 2;
3200         clcf->noregex = 1;
3201
3202     } else if (name->data[0] == '~') {
3203
3204         name->len--;
3205         name->data++;
3206
3207         if (name->data[0] == '*') {
3208
3209             name->len--;
3210             name->data++;
3211
3212             if (ngx\_http\_core\_regex\_location(cf, clcf, name, 1) != NGX\_OK) {
3213                 return NGX\_CONF\_ERROR;
3214             }
3215
3216         } else {
3217             if (ngx\_http\_core\_regex\_location(cf, clcf, name, 0) != NGX\_OK) {
3218                 return NGX\_CONF\_ERROR;
3219             }
3220         }
3221     }
3222 } else {
3223
3224     clcf->name = *name;
3225
3226     if (name->data[0] == '@') {
3227         clcf->named = 1;
3228     }
3229 }
3230 }
3231 }
3232
3233 pclcf = pctx->loc_conf[ngx\_http\_core\_module.ctx_index];
3234
3235 if (pclcf->name.len) {
3236
3237     /* nested location */
3238
3239     #if 0
3240     clcf->prev_location = pclcf;
3241     #endif
3242
3243     if (pclcf->exact_match) {
3244         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
3245             "location \"%V\" cannot be inside "
3246             "the exact location \"%V\"",
3247             &clcf->name, &pclcf->name);
3248         return NGX\_CONF\_ERROR;
3249     }
3250
3251     if (pclcf->named) {
3252         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
3253             "location \"%V\" cannot be inside "
3254             "the named location \"%V\"",
3255             &clcf->name, &pclcf->name);
3256         return NGX\_CONF\_ERROR;
3257     }
3258
3259     if (clcf->named) {
3260         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
3261             "named location \"%V\" can be "
3262             "on the server level only",
3263             &clcf->name);
3264         return NGX\_CONF\_ERROR;
3265     }

```

```

3266         len = pclcf->name.len;
3267
3268
3269 #if (NGX_PCRE)
3270     if (clcf->regex == NULL
3271         && ngx_filename_cmp(clcf->name.data, pclcf->name.data, len) != 0)
3272 #else
3273     if (ngx_filename_cmp(clcf->name.data, pclcf->name.data, len) != 0)
3274 #endif
3275     {
3276         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
3277             "location \"%V\" is outside location \"%V\"",
3278             &clcf->name, &pclcf->name);
3279         return NGX_CONF_ERROR;
3280     }
3281 }
3282
3283 if (ngx_http_add_location(cf, &pclcf->locations, clcf) != NGX_OK) {
3284     return NGX_CONF_ERROR;
3285 }
3286
3287 save = *cf;
3288 cf->ctx = ctx;
3289 cf->cmd_type = NGX_HTTP_LOC_CONF;
3290
3291 rv = ngx_conf_parse(cf, NULL);
3292
3293 *cf = save;
3294
3295 return rv;
3296 }
3297
3298
3299 static ngx_int_t
3300 ngx_http_core_regex_location(ngx_conf_t *cf, ngx_http_core_loc_conf_t *clcf,
3301     ngx_str_t *regex, ngx_uint_t caseless)
3302 {
3303     #if (NGX_PCRE)
3304     ngx_regex_compile_t rc;
3305     u_char errstr[NGX_MAX_CONF_ERRSTR];
3306
3307     ngx_memzero(&rc, sizeof(ngx_regex_compile_t));
3308
3309     rc.pattern = *regex;
3310     rc.err.len = NGX_MAX_CONF_ERRSTR;
3311     rc.err.data = errstr;
3312
3313     #if (NGX_HAVE_CASELESS_FILESYSTEM)
3314     rc.options = NGX_REGEX_CASELESS;
3315     #else
3316     rc.options = caseless ? NGX_REGEX_CASELESS : 0;
3317     #endif
3318
3319     clcf->regex = ngx_http_regex_compile(cf, &rc);
3320     if (clcf->regex == NULL) {
3321         return NGX_ERROR;
3322     }
3323
3324     clcf->name = *regex;
3325
3326     return NGX_OK;
3327
3328 #else
3329     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
3330         "using regex \"%V\" requires PCRE library",
3331         regex);
3332     return NGX_ERROR;
3333 #endif
3334 }
3335 #endif
3336 }
3337
3338
3339 static char *
3340 ngx_http_core_types(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
3341 {

```

```

3342     ngx_http_core_loc_conf_t *clcf = conf;
3343
3344     char          *rv;
3345     ngx_conf_t   save;
3346
3347     if (clcf->types == NULL) {
3348         clcf->types = ngx_array_create(cf->pool, 64, sizeof(ngx_hash_key_t));
3349         if (clcf->types == NULL) {
3350             return NGX_CONF_ERROR;
3351         }
3352     }
3353
3354     save = *cf;
3355     cf->handler = ngx_http_core_type;
3356     cf->handler_conf = conf;
3357
3358     rv = ngx_conf_parse(cf, NULL);
3359
3360     *cf = save;
3361
3362     return rv;
3363 }
3364
3365
3366 static char *
3367 ngx_http_core_type(ngx_conf_t *cf, ngx_command_t *dummy, void *conf)
3368 {
3369     ngx_http_core_loc_conf_t *clcf = conf;
3370
3371     ngx_str_t      *value, *content_type, *old;
3372     ngx_uint_t     i, n, hash;
3373     ngx_hash_key_t *type;
3374
3375     value = cf->args->elts;
3376
3377     if (ngx_strcmp(value[0].data, "include") == 0) {
3378         if (cf->args->nelts != 2) {
3379             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
3380                 "invalid number of arguments"
3381                 " in \"include\" directive");
3382             return NGX_CONF_ERROR;
3383         }
3384
3385         return ngx_conf_include(cf, dummy, conf);
3386     }
3387
3388     content_type = ngx_palloc(cf->pool, sizeof(ngx_str_t));
3389     if (content_type == NULL) {
3390         return NGX_CONF_ERROR;
3391     }
3392
3393     *content_type = value[0];
3394
3395     for (i = 1; i < cf->args->nelts; i++) {
3396
3397         hash = ngx_hash_strlow(value[i].data, value[i].data, value[i].len);
3398
3399         type = clcf->types->elts;
3400         for (n = 0; n < clcf->types->nelts; n++) {
3401             if (ngx_strcmp(value[i].data, type[n].key.data) == 0) {
3402                 old = type[n].value;
3403                 type[n].value = content_type;
3404
3405                 ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
3406                     "duplicate extension \"%V\", "
3407                     "content type: \"%V\", "
3408                     "previous content type: \"%V\"",
3409                     &value[i], content_type, old);
3410                 goto next;
3411             }
3412         }
3413
3414         type = ngx_array_push(clcf->types);
3415         if (type == NULL) {
3416             return NGX_CONF_ERROR;
3417

```

```

3418     }
3419
3420     type->key = value[i];
3421     type->key_hash = hash;
3422     type->value = content_type;
3423
3424     next:
3425         continue;
3426     }
3427
3428     return NGX\_CONF\_OK;
3429 }
3430
3431
3432 static ngx\_int\_t
3433 ngx\_http\_core\_preconfiguration(ngx\_conf\_t *cf)
3434 {
3435     return ngx\_http\_variables\_add\_core\_vars(cf);
3436 }
3437
3438
3439 static void *
3440 ngx\_http\_core\_create\_main\_conf(ngx\_conf\_t *cf)
3441 {
3442     ngx\_http\_core\_main\_conf\_t *cmcf;
3443
3444     cmcf = ngx\_palloc(cf->pool, sizeof(ngx\_http\_core\_main\_conf\_t));
3445     if (cmcf == NULL) {
3446         return NULL;
3447     }
3448
3449     if (ngx\_array\_init(&cmcf->servers, cf->pool, 4,
3450         sizeof(ngx\_http\_core\_srv\_conf\_t *)))
3451         != NGX\_OK)
3452     {
3453         return NULL;
3454     }
3455
3456     cmcf->server_names_hash_max_size = NGX\_CONF\_UNSET\_UINT;
3457     cmcf->server_names_hash_bucket_size = NGX\_CONF\_UNSET\_UINT;
3458
3459     cmcf->variables_hash_max_size = NGX\_CONF\_UNSET\_UINT;
3460     cmcf->variables_hash_bucket_size = NGX\_CONF\_UNSET\_UINT;
3461
3462     return cmcf;
3463 }
3464
3465
3466 static char *
3467 ngx\_http\_core\_init\_main\_conf(ngx\_conf\_t *cf, void *conf)
3468 {
3469     ngx\_http\_core\_main\_conf\_t *cmcf = conf;
3470
3471     ngx\_conf\_init\_uint\_value(cmcf->server_names_hash_max_size, 512);
3472     ngx\_conf\_init\_uint\_value(cmcf->server_names_hash_bucket_size,
3473         ngx\_cacheline\_size);
3474
3475     cmcf->server_names_hash_bucket_size =
3476         ngx\_align(cmcf->server_names_hash_bucket_size, ngx\_cacheline\_size);
3477
3478
3479     ngx\_conf\_init\_uint\_value(cmcf->variables_hash_max_size, 1024);
3480     ngx\_conf\_init\_uint\_value(cmcf->variables_hash_bucket_size, 64);
3481
3482     cmcf->variables_hash_bucket_size =
3483         ngx\_align(cmcf->variables_hash_bucket_size, ngx\_cacheline\_size);
3484
3485     if (cmcf->ncaptures) {
3486         cmcf->ncaptures = (cmcf->ncaptures + 1) * 3;
3487     }
3488
3489     return NGX\_CONF\_OK;
3490 }
3491
3492
3493 static void *

```



```

3494 ngx_http_core_create_srv_conf(ngx_conf_t *cf)
3495 {
3496     ngx_http_core_srv_conf_t *cscf;
3497
3498     cscf = ngx_palloc(cf->pool, sizeof(ngx_http_core_srv_conf_t));
3499     if (cscf == NULL) {
3500         return NULL;
3501     }
3502
3503     /*
3504      * set by ngx_palloc():
3505      *
3506      *     conf->client_large_buffers.num = 0;
3507      */
3508
3509     if (ngx_array_init(&cscf->server_names, cf->temp_pool, 4,
3510         sizeof(ngx_http_server_name_t))
3511         != NGX_OK)
3512     {
3513         return NULL;
3514     }
3515
3516     cscf->connection_pool_size = NGX_CONF_UNSET_SIZE;
3517     cscf->request_pool_size = NGX_CONF_UNSET_SIZE;
3518     cscf->client_header_timeout = NGX_CONF_UNSET_MSEC;
3519     cscf->client_header_buffer_size = NGX_CONF_UNSET_SIZE;
3520     cscf->ignore_invalid_headers = NGX_CONF_UNSET;
3521     cscf->merge_slashes = NGX_CONF_UNSET;
3522     cscf->underscores_in_headers = NGX_CONF_UNSET;
3523
3524     return cscf;
3525 }
3526
3527
3528 static char *
3529 ngx_http_core_merge_srv_conf(ngx_conf_t *cf, void *parent, void *child)
3530 {
3531     ngx_http_core_srv_conf_t *prev = parent;
3532     ngx_http_core_srv_conf_t *conf = child;
3533
3534     ngx_str_t name;
3535     ngx_http_server_name_t *sn;
3536
3537     /* TODO: it does not merge, it inits only */
3538
3539     ngx_conf_merge_size_value(conf->connection_pool_size,
3540         prev->connection_pool_size, 256);
3541     ngx_conf_merge_size_value(conf->request_pool_size,
3542         prev->request_pool_size, 4096);
3543     ngx_conf_merge_msec_value(conf->client_header_timeout,
3544         prev->client_header_timeout, 60000);
3545     ngx_conf_merge_size_value(conf->client_header_buffer_size,
3546         prev->client_header_buffer_size, 1024);
3547     ngx_conf_merge_bufs_value(conf->large_client_header_buffers,
3548         prev->large_client_header_buffers,
3549         4, 8192);
3550
3551     if (conf->large_client_header_buffers.size < conf->connection_pool_size) {
3552         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
3553             "the \"large_client_header_buffers\" size must be "
3554             "equal to or greater than \"connection_pool_size\"");
3555         return NGX_CONF_ERROR;
3556     }
3557
3558     ngx_conf_merge_value(conf->ignore_invalid_headers,
3559         prev->ignore_invalid_headers, 1);
3560
3561     ngx_conf_merge_value(conf->merge_slashes, prev->merge_slashes, 1);
3562
3563     ngx_conf_merge_value(conf->underscores_in_headers,
3564         prev->underscores_in_headers, 0);
3565
3566     if (conf->server_names.nelts == 0) {
3567         /* the array has 4 empty preallocated elements, so push cannot fail */
3568         sn = ngx_array_push(&conf->server_names);
3569         #if (NGX_PCRE)

```

```

3570     sn->regex = NULL;
3571 #endif
3572     sn->server = conf;
3573     ngx_str_set(&sn->name, "");
3574 }
3575
3576     sn = conf->server_names.elts;
3577     name = sn[0].name;
3578
3579 #if (NGX_PCRE)
3580     if (sn->regex) {
3581         name.len++;
3582         name.data--;
3583     } else
3584 #endif
3585
3586     if (name.data[0] == '.') {
3587         name.len--;
3588         name.data++;
3589     }
3590
3591     conf->server_name.len = name.len;
3592     conf->server_name.data = ngx_pstrdup(cf->pool, &name);
3593     if (conf->server_name.data == NULL) {
3594         return NGX_CONF_ERROR;
3595     }
3596
3597     return NGX_CONF_OK;
3598 }
3599
3600
3601 static void *
3602 ngx_http_core_create_loc_conf(ngx_conf_t *cf)
3603 {
3604     ngx_http_core_loc_conf_t *clcf;
3605
3606     clcf = ngx_palloc(cf->pool, sizeof(ngx_http_core_loc_conf_t));
3607     if (clcf == NULL) {
3608         return NULL;
3609     }
3610
3611     /*
3612     * set by ngx_palloc():
3613     *
3614     *     clcf->root = { 0, NULL };
3615     *     clcf->limit_except = 0;
3616     *     clcf->post_action = { 0, NULL };
3617     *     clcf->types = NULL;
3618     *     clcf->default_type = { 0, NULL };
3619     *     clcf->error_log = NULL;
3620     *     clcf->error_pages = NULL;
3621     *     clcf->try_files = NULL;
3622     *     clcf->client_body_path = NULL;
3623     *     clcf->regex = NULL;
3624     *     clcf->exact_match = 0;
3625     *     clcf->auto_redirect = 0;
3626     *     clcf->alias = 0;
3627     *     clcf->gzip_proxied = 0;
3628     *     clcf->keepalive_disable = 0;
3629     */
3630
3631     clcf->client_max_body_size = NGX_CONF_UNSET;
3632     clcf->client_body_buffer_size = NGX_CONF_UNSET_SIZE;
3633     clcf->client_body_timeout = NGX_CONF_UNSET_MSEC;
3634     clcf->satisfy = NGX_CONF_UNSET_UINT;
3635     clcf->if_modified_since = NGX_CONF_UNSET_UINT;
3636     clcf->max_ranges = NGX_CONF_UNSET_UINT;
3637     clcf->client_body_in_file_only = NGX_CONF_UNSET_UINT;
3638     clcf->client_body_in_single_buffer = NGX_CONF_UNSET;
3639     clcf->internal = NGX_CONF_UNSET;
3640     clcf->sendfile = NGX_CONF_UNSET;
3641     clcf->sendfile_max_chunk = NGX_CONF_UNSET_SIZE;
3642 #if (NGX_HAVE_FILE_AIO)
3643     clcf->aio = NGX_CONF_UNSET;
3644 #endif
3645     clcf->read_ahead = NGX_CONF_UNSET_SIZE;

```

```

3646     clcf->directio = NGX\_CONF\_UNSET;
3647     clcf->directio_alignment = NGX\_CONF\_UNSET;
3648     clcf->tcp_nopush = NGX\_CONF\_UNSET;
3649     clcf->tcp_nodelay = NGX\_CONF\_UNSET;
3650     clcf->send_timeout = NGX\_CONF\_UNSET\_MSEC;
3651     clcf->send_lowat = NGX\_CONF\_UNSET\_SIZE;
3652     clcf->postpone_output = NGX\_CONF\_UNSET\_SIZE;
3653     clcf->limit_rate = NGX\_CONF\_UNSET\_SIZE;
3654     clcf->limit_rate_after = NGX\_CONF\_UNSET\_SIZE;
3655     clcf->keepalive_timeout = NGX\_CONF\_UNSET\_MSEC;
3656     clcf->keepalive_header = NGX\_CONF\_UNSET;
3657     clcf->keepalive_requests = NGX\_CONF\_UNSET\_UINT;
3658     clcf->lingering_close = NGX\_CONF\_UNSET\_UINT;
3659     clcf->lingering_time = NGX\_CONF\_UNSET\_MSEC;
3660     clcf->lingering_timeout = NGX\_CONF\_UNSET\_MSEC;
3661     clcf->resolver_timeout = NGX\_CONF\_UNSET\_MSEC;
3662     clcf->reset_timedout_connection = NGX\_CONF\_UNSET;
3663     clcf->server_name_in_redirect = NGX\_CONF\_UNSET;
3664     clcf->port_in_redirect = NGX\_CONF\_UNSET;
3665     clcf->msie_padding = NGX\_CONF\_UNSET;
3666     clcf->msie_refresh = NGX\_CONF\_UNSET;
3667     clcf->log_not_found = NGX\_CONF\_UNSET;
3668     clcf->log_subrequest = NGX\_CONF\_UNSET;
3669     clcf->recursive_error_pages = NGX\_CONF\_UNSET;
3670     clcf->server_tokens = NGX\_CONF\_UNSET;
3671     clcf->chunked_transfer_encoding = NGX\_CONF\_UNSET;
3672     clcf->etag = NGX\_CONF\_UNSET;
3673     clcf->types_hash_max_size = NGX\_CONF\_UNSET\_UINT;
3674     clcf->types_hash_bucket_size = NGX\_CONF\_UNSET\_UINT;
3675
3676     clcf->open_file_cache = NGX\_CONF\_UNSET\_PTR;
3677     clcf->open_file_cache_valid = NGX\_CONF\_UNSET;
3678     clcf->open_file_cache_min_uses = NGX\_CONF\_UNSET\_UINT;
3679     clcf->open_file_cache_errors = NGX\_CONF\_UNSET;
3680     clcf->open_file_cache_events = NGX\_CONF\_UNSET;
3681
3682     #if (NGX\_HTTP\_GZIP)
3683         clcf->gzip_vary = NGX\_CONF\_UNSET;
3684         clcf->gzip_http_version = NGX\_CONF\_UNSET\_UINT;
3685     #if (NGX\_PCRE)
3686         clcf->gzip_disable = NGX\_CONF\_UNSET\_PTR;
3687     #endif
3688         clcf->gzip_disable_msie6 = 3;
3689     #if (NGX\_HTTP\_DEGRADATION)
3690         clcf->gzip_disable_degradation = 3;
3691     #endif
3692 #endif
3693
3694     #if (NGX\_HAVE\_OPENAT)
3695         clcf->disable_symlinks = NGX\_CONF\_UNSET\_UINT;
3696         clcf->disable_symlinks_from = NGX\_CONF\_UNSET\_PTR;
3697     #endif
3698
3699     return clcf;
3700 }
3701
3702
3703 static ngx_str_t ngx_http_core_text_html_type = ngx\_string\("text/html"\);
3704 static ngx_str_t ngx_http_core_image_gif_type = ngx\_string\("image/gif"\);
3705 static ngx_str_t ngx_http_core_image_jpeg_type = ngx\_string\("image/jpeg"\);
3706
3707 static ngx_hash_key_t ngx_http_core_default_types[] = {
3708     { ngx\_string\("html"\), 0, &ngx_http_core_text_html_type },
3709     { ngx\_string\("gif"\), 0, &ngx_http_core_image_gif_type },
3710     { ngx\_string\("jpg"\), 0, &ngx_http_core_image_jpeg_type },
3711     { ngx\_null\_string, 0, NULL }
3712 };
3713
3714
3715 static char *
3716 ngx_http_core_merge_loc_conf(ngx\_conf\_t *cf, void *parent, void *child)
3717 {
3718     ngx\_http\_core\_loc\_conf\_t *prev = parent;
3719     ngx\_http\_core\_loc\_conf\_t *conf = child;
3720
3721     ngx\_uint\_t i;

```

```

3722 ngx\_hash\_key\_t *type;
3723 ngx\_hash\_init\_t types_hash;
3724
3725 if (conf->root.data == NULL) {
3726
3727     conf->alias = prev->alias;
3728     conf->root = prev->root;
3729     conf->root_lengths = prev->root_lengths;
3730     conf->root_values = prev->root_values;
3731
3732     if (prev->root.data == NULL) {
3733         ngx\_str\_set(&conf->root, "html");
3734
3735         if (ngx\_conf\_full\_name(cf->cycle, &conf->root, 0) != NGX\_OK) {
3736             return NGX\_CONF\_ERROR;
3737         }
3738     }
3739 }
3740
3741 if (conf->post_action.data == NULL) {
3742     conf->post_action = prev->post_action;
3743 }
3744
3745 ngx\_conf\_merge\_uint\_value(conf->types_hash_max_size,
3746                             prev->types_hash_max_size, 1024);
3747
3748 ngx\_conf\_merge\_uint\_value(conf->types_hash_bucket_size,
3749                             prev->types_hash_bucket_size, 64);
3750
3751 conf->types_hash_bucket_size = ngx\_align(conf->types_hash_bucket_size,
3752                                         ngx\_cacheline\_size);
3753
3754 /*
3755  * the special handling of the "types" directive in the "http" section
3756  * to inherit the http's conf->types_hash to all servers
3757 */
3758
3759 if (prev->types && prev->types_hash.buckets == NULL) {
3760
3761     types_hash.hash = &prev->types_hash;
3762     types_hash.key = ngx\_hash\_key\_lc;
3763     types_hash.max_size = conf->types_hash_max_size;
3764     types_hash.bucket_size = conf->types_hash_bucket_size;
3765     types_hash.name = "types_hash";
3766     types_hash.pool = cf->pool;
3767     types_hash.temp_pool = NULL;
3768
3769     if (ngx\_hash\_init(&types_hash, prev->types->elts, prev->types->nelts)
3770         != NGX\_OK)
3771     {
3772         return NGX\_CONF\_ERROR;
3773     }
3774 }
3775
3776 if (conf->types == NULL) {
3777     conf->types = prev->types;
3778     conf->types_hash = prev->types_hash;
3779 }
3780
3781 if (conf->types == NULL) {
3782     conf->types = ngx\_array\_create(cf->pool, 3, sizeof\(ngx\_hash\_key\_t\));
3783     if (conf->types == NULL) {
3784         return NGX\_CONF\_ERROR;
3785     }
3786
3787     for (i = 0; ngx\_http\_core\_default\_types[i].key.len; i++) {
3788         type = ngx\_array\_push(conf->types);
3789         if (type == NULL) {
3790             return NGX\_CONF\_ERROR;
3791         }
3792
3793         type->key = ngx\_http\_core\_default\_types[i].key;
3794         type->key_hash =
3795             ngx\_hash\_key\_lc(ngx\_http\_core\_default\_types[i].key.data,
3796                            ngx\_http\_core\_default\_types[i].key.len);
3797         type->value = ngx\_http\_core\_default\_types[i].value;

```

```

3798     }
3799 }
3800
3801 if (conf->types_hash.buckets == NULL) {
3802
3803     types_hash.hash = &conf->types_hash;
3804     types_hash.key = ngx_hash_key_lc;
3805     types_hash.max_size = conf->types_hash_max_size;
3806     types_hash.bucket_size = conf->types_hash_bucket_size;
3807     types_hash.name = "types_hash";
3808     types_hash.pool = cf->pool;
3809     types_hash.temp_pool = NULL;
3810
3811     if (ngx_hash_init(&types_hash, conf->types->elts, conf->types->nelts)
3812         != NGX_OK)
3813     {
3814         return NGX_CONF_ERROR;
3815     }
3816 }
3817
3818 if (conf->error_log == NULL) {
3819     if (prev->error_log) {
3820         conf->error_log = prev->error_log;
3821     } else {
3822         conf->error_log = &cf->cycle->new_log;
3823     }
3824 }
3825
3826 if (conf->error_pages == NULL && prev->error_pages) {
3827     conf->error_pages = prev->error_pages;
3828 }
3829
3830 ngx_conf_merge_str_value(conf->default_type,
3831     prev->default_type, "text/plain");
3832
3833 ngx_conf_merge_off_value(conf->client_max_body_size,
3834     prev->client_max_body_size, 1 * 1024 * 1024);
3835 ngx_conf_merge_size_value(conf->client_body_buffer_size,
3836     prev->client_body_buffer_size,
3837     (size_t) 2 * ngx_pagesize);
3838 ngx_conf_merge_msec_value(conf->client_body_timeout,
3839     prev->client_body_timeout, 60000);
3840
3841 ngx_conf_merge_bitmask_value(conf->keepalive_disable,
3842     prev->keepalive_disable,
3843     (NGX_CONF_BITMASK_SET
3844     |NGX_HTTP_KEEPALIVE_DISABLE_MSIE6));
3845 ngx_conf_merge_uint_value(conf->satisfy, prev->satisfy,
3846     NGX_HTTP_SATISFY_ALL);
3847 ngx_conf_merge_uint_value(conf->if_modified_since, prev->if_modified_since,
3848     NGX_HTTP_IMS_EXACT);
3849 ngx_conf_merge_uint_value(conf->max_ranges, prev->max_ranges,
3850     NGX_MAX_INT32_VALUE);
3851 ngx_conf_merge_uint_value(conf->client_body_in_file_only,
3852     prev->client_body_in_file_only,
3853     NGX_HTTP_REQUEST_BODY_FILE_OFF);
3854 ngx_conf_merge_value(conf->client_body_in_single_buffer,
3855     prev->client_body_in_single_buffer, 0);
3856 ngx_conf_merge_value(conf->internal, prev->internal, 0);
3857 ngx_conf_merge_value(conf->sendfile, prev->sendfile, 0);
3858 ngx_conf_merge_size_value(conf->sendfile_max_chunk,
3859     prev->sendfile_max_chunk, 0);
3860 #if (NGX_HAVE_FILE_AIO)
3861     ngx_conf_merge_value(conf->aio, prev->aio, NGX_HTTP_AIO_OFF);
3862 #endif
3863 ngx_conf_merge_size_value(conf->read_ahead, prev->read_ahead, 0);
3864 ngx_conf_merge_off_value(conf->directio, prev->directio,
3865     NGX_OPEN_FILE_DIRECTIO_OFF);
3866 ngx_conf_merge_off_value(conf->directio_alignment, prev->directio_alignment,
3867     512);
3868 ngx_conf_merge_value(conf->tcp_nopush, prev->tcp_nopush, 0);
3869 ngx_conf_merge_value(conf->tcp_nodelay, prev->tcp_nodelay, 1);
3870
3871 ngx_conf_merge_msec_value(conf->send_timeout, prev->send_timeout, 60000);
3872 ngx_conf_merge_size_value(conf->send_lowat, prev->send_lowat, 0);
3873 ngx_conf_merge_size_value(conf->postpone_output, prev->postpone_output,

```

```

3874         1460);
3875     ngx_conf_merge_size_value(conf->limit_rate, prev->limit_rate, 0);
3876     ngx_conf_merge_size_value(conf->limit_rate_after, prev->limit_rate_after,
3877     0);
3878     ngx_conf_merge_msec_value(conf->keepalive_timeout,
3879     prev->keepalive_timeout, 75000);
3880     ngx_conf_merge_sec_value(conf->keepalive_header,
3881     prev->keepalive_header, 0);
3882     ngx_conf_merge_uint_value(conf->keepalive_requests,
3883     prev->keepalive_requests, 100);
3884     ngx_conf_merge_uint_value(conf->lingering_close,
3885     prev->lingering_close, NGX_HTTP_LINGERING_ON);
3886     ngx_conf_merge_msec_value(conf->lingering_time,
3887     prev->lingering_time, 30000);
3888     ngx_conf_merge_msec_value(conf->lingering_timeout,
3889     prev->lingering_timeout, 5000);
3890     ngx_conf_merge_msec_value(conf->resolver_timeout,
3891     prev->resolver_timeout, 30000);
3892
3893     if (conf->resolver == NULL) {
3894
3895         if (prev->resolver == NULL) {
3896
3897             /*
3898              * create dummy resolver in http {} context
3899              * to inherit it in all servers
3900              */
3901
3902             prev->resolver = ngx_resolver_create(cf, NULL, 0);
3903             if (prev->resolver == NULL) {
3904                 return NGX_CONF_ERROR;
3905             }
3906         }
3907
3908         conf->resolver = prev->resolver;
3909     }
3910
3911     if (ngx_conf_merge_path_value(cf, &conf->client_body_temp_path,
3912     prev->client_body_temp_path,
3913     &ngx_http_client_temp_path)
3914     != NGX_OK)
3915     {
3916         return NGX_CONF_ERROR;
3917     }
3918
3919     ngx_conf_merge_value(conf->reset_timedout_connection,
3920     prev->reset_timedout_connection, 0);
3921     ngx_conf_merge_value(conf->server_name_in_redirect,
3922     prev->server_name_in_redirect, 0);
3923     ngx_conf_merge_value(conf->port_in_redirect, prev->port_in_redirect, 1);
3924     ngx_conf_merge_value(conf->msie_padding, prev->msie_padding, 1);
3925     ngx_conf_merge_value(conf->msie_refresh, prev->msie_refresh, 0);
3926     ngx_conf_merge_value(conf->log_not_found, prev->log_not_found, 1);
3927     ngx_conf_merge_value(conf->log_subrequest, prev->log_subrequest, 0);
3928     ngx_conf_merge_value(conf->recursive_error_pages,
3929     prev->recursive_error_pages, 0);
3930     ngx_conf_merge_value(conf->server_tokens, prev->server_tokens, 1);
3931     ngx_conf_merge_value(conf->chunked_transfer_encoding,
3932     prev->chunked_transfer_encoding, 1);
3933     ngx_conf_merge_value(conf->etag, prev->etag, 1);
3934
3935     ngx_conf_merge_ptr_value(conf->open_file_cache,
3936     prev->open_file_cache, NULL);
3937
3938     ngx_conf_merge_sec_value(conf->open_file_cache_valid,
3939     prev->open_file_cache_valid, 60);
3940
3941     ngx_conf_merge_uint_value(conf->open_file_cache_min_uses,
3942     prev->open_file_cache_min_uses, 1);
3943
3944     ngx_conf_merge_sec_value(conf->open_file_cache_errors,
3945     prev->open_file_cache_errors, 0);
3946
3947     ngx_conf_merge_sec_value(conf->open_file_cache_events,
3948     prev->open_file_cache_events, 0);
3949     #if (NGX_HTTP_GZIP)

```

```

3950     ngx_conf_merge_value(conf->gzip_vary, prev->gzip_vary, 0);
3951     ngx_conf_merge_uint_value(conf->gzip_http_version, prev->gzip_http_version,
3952                               NGX_HTTP_VERSION_11);
3953     ngx_conf_merge_bitmask_value(conf->gzip_proxied, prev->gzip_proxied,
3954                                  (NGX_CONF_BITMASK_SET|NGX_HTTP_GZIP_PROXIED_OFF));
3955
3956
3957     #if (NGX_PCRE)
3958         ngx_conf_merge_ptr_value(conf->gzip_disable, prev->gzip_disable, NULL);
3959     #endif
3960
3961     if (conf->gzip_disable_msie6 == 3) {
3962         conf->gzip_disable_msie6 =
3963             (prev->gzip_disable_msie6 == 3) ? 0 : prev->gzip_disable_msie6;
3964     }
3965
3966     #if (NGX_HTTP_DEGRADATION)
3967
3968     if (conf->gzip_disable_degradation == 3) {
3969         conf->gzip_disable_degradation =
3970             (prev->gzip_disable_degradation == 3) ?
3971             0 : prev->gzip_disable_degradation;
3972     }
3973
3974     #endif
3975 #endif
3976
3977     #if (NGX_HAVE_OPENAT)
3978         ngx_conf_merge_uint_value(conf->disable_symlinks, prev->disable_symlinks,
3979                                   NGX_DISABLE_SYMLINKS_OFF);
3980         ngx_conf_merge_ptr_value(conf->disable_symlinks_from,
3981                                   prev->disable_symlinks_from, NULL);
3982     #endif
3983
3984     return NGX_CONF_OK;
3985 }
3986
3987
3988 static char *
3989 ngx_http_core_listen(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
3990 {
3991     ngx_http_core_srv_conf_t *cscf = conf;
3992
3993     ngx_str_t      *value, size;
3994     ngx_url_t      u;
3995     ngx_uint_t      n;
3996     ngx_http_listen_opt_t  lsopt;
3997
3998     cscf->listen = 1;
3999
4000     value = cf->args->elts;
4001
4002     ngx_memzero(&u, sizeof(ngx_url_t));
4003
4004     u.url = value[1];
4005     u.listen = 1;
4006     u.default_port = 80;
4007
4008     if (ngx_parse_url(cf->pool, &u) != NGX_OK) {
4009         if (u.err) {
4010             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
4011                               "%s in \"%V\" of the \"listen\" directive",
4012                               u.err, &u.url);
4013         }
4014
4015         return NGX_CONF_ERROR;
4016     }
4017
4018     ngx_memzero(&lsopt, sizeof(ngx_http_listen_opt_t));
4019
4020     ngx_memcpy(&lsopt.u.sockaddr, u.sockaddr, u.socklen);
4021
4022     lsopt.socklen = u.socklen;
4023     lsopt.backlog = NGX_LISTEN_BACKLOG;
4024     lsopt.rcvbuf = -1;
4025     lsopt.sndbuf = -1;

```

```

4026 #if (NGX_HAVE_SETFIB)
4027     lsopt.setfib = -1;
4028 #endif
4029 #if (NGX_HAVE_TCP_FASTOPEN)
4030     lsopt.fastopen = -1;
4031 #endif
4032     lsopt.wildcard = u.wildcard;
4033 #if (NGX_HAVE_INET6 && defined IPV6_V6ONLY)
4034     lsopt.ipv6only = 1;
4035 #endif
4036
4037     (void) ngx_sock_ntop(&lsopt.u.sockaddr, lsopt.socklen, lsopt.addr,
4038                         NGX_SOCKADDR_STRLEN, 1);
4039
4040     for (n = 2; n < cf->args->nelts; n++) {
4041
4042         if (ngx_strcmp(value[n].data, "default_server") == 0
4043             || ngx_strcmp(value[n].data, "default") == 0)
4044         {
4045             lsopt.default_server = 1;
4046             continue;
4047         }
4048
4049         if (ngx_strcmp(value[n].data, "bind") == 0) {
4050             lsopt.set = 1;
4051             lsopt.bind = 1;
4052             continue;
4053         }
4054
4055         #if (NGX_HAVE_SETFIB)
4056         if (ngx_strncmp(value[n].data, "setfib=", 7) == 0) {
4057             lsopt.setfib = ngx_atoi(value[n].data + 7, value[n].len - 7);
4058             lsopt.set = 1;
4059             lsopt.bind = 1;
4060
4061             if (lsopt.setfib == NGX_ERROR) {
4062                 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
4063                                     "invalid setfib \"%V\"", &value[n]);
4064                 return NGX_CONF_ERROR;
4065             }
4066
4067             continue;
4068         }
4069         #endif
4070
4071         #if (NGX_HAVE_TCP_FASTOPEN)
4072         if (ngx_strncmp(value[n].data, "fastopen=", 9) == 0) {
4073             lsopt.fastopen = ngx_atoi(value[n].data + 9, value[n].len - 9);
4074             lsopt.set = 1;
4075             lsopt.bind = 1;
4076
4077             if (lsopt.fastopen == NGX_ERROR) {
4078                 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
4079                                     "invalid fastopen \"%V\"", &value[n]);
4080                 return NGX_CONF_ERROR;
4081             }
4082
4083             continue;
4084         }
4085         #endif
4086
4087         if (ngx_strncmp(value[n].data, "backlog=", 8) == 0) {
4088             lsopt.backlog = ngx_atoi(value[n].data + 8, value[n].len - 8);
4089             lsopt.set = 1;
4090             lsopt.bind = 1;
4091
4092             if (lsopt.backlog == NGX_ERROR || lsopt.backlog == 0) {
4093                 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
4094                                     "invalid backlog \"%V\"", &value[n]);
4095                 return NGX_CONF_ERROR;
4096             }
4097
4098             continue;
4099         }
4100
4101         if (ngx_strncmp(value[n].data, "rcvbuf=", 7) == 0) {

```



```

4102     size.len = value[n].len - 7;
4103     size.data = value[n].data + 7;
4104
4105     lsopt.rcvbuf = ngx_parse_size(&size);
4106     lsopt.set = 1;
4107     lsopt.bind = 1;
4108
4109     if (lsopt.rcvbuf == NGX_ERROR) {
4110         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
4111             "invalid rcvbuf \"%V\"", &value[n]);
4112         return NGX_CONF_ERROR;
4113     }
4114
4115     continue;
4116 }
4117
4118 if (ngx_strncmp(value[n].data, "sndbuf=", 7) == 0) {
4119     size.len = value[n].len - 7;
4120     size.data = value[n].data + 7;
4121
4122     lsopt.sndbuf = ngx_parse_size(&size);
4123     lsopt.set = 1;
4124     lsopt.bind = 1;
4125
4126     if (lsopt.sndbuf == NGX_ERROR) {
4127         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
4128             "invalid sndbuf \"%V\"", &value[n]);
4129         return NGX_CONF_ERROR;
4130     }
4131
4132     continue;
4133 }
4134
4135 if (ngx_strncmp(value[n].data, "accept_filter=", 14) == 0) {
4136 #if (NGX_HAVE_DEFERRED_ACCEPT && defined SO_ACCEPTFILTER)
4137     lsopt.accept_filter = (char *) &value[n].data[14];
4138     lsopt.set = 1;
4139     lsopt.bind = 1;
4140 #else
4141     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
4142         "accept filters \"%V\" are not supported "
4143         "on this platform, ignored",
4144         &value[n]);
4145 #endif
4146     continue;
4147 }
4148
4149 if (ngx_strcmp(value[n].data, "deferred") == 0) {
4150 #if (NGX_HAVE_DEFERRED_ACCEPT && defined TCP_DEFER_ACCEPT)
4151     lsopt.deferred_accept = 1;
4152     lsopt.set = 1;
4153     lsopt.bind = 1;
4154 #else
4155     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
4156         "the deferred accept is not supported "
4157         "on this platform, ignored");
4158 #endif
4159     continue;
4160 }
4161
4162 if (ngx_strncmp(value[n].data, "ipv6only=0", 10) == 0) {
4163 #if (NGX_HAVE_INET6 && defined IPV6_V6ONLY)
4164     struct sockaddr *sa;
4165
4166     sa = &lsopt.u.sockaddr;
4167
4168     if (sa->sa_family == AF_INET6) {
4169
4170         if (ngx_strcmp(&value[n].data[10], "n") == 0) {
4171             lsopt.ipv6only = 1;
4172
4173         } else if (ngx_strcmp(&value[n].data[10], "ff") == 0) {
4174             lsopt.ipv6only = 0;
4175
4176         } else {
4177             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,

```

```

4178             "invalid ipv6only flags \"%s\"",
4179             &value[n].data[9]);
4180         return NGX_CONF_ERROR;
4181     }
4182
4183     lsopt.set = 1;
4184     lsopt.bind = 1;
4185
4186     } else {
4187         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
4188             "ipv6only is not supported "
4189             "on addr \"%s\", ignored", lsopt.addr);
4190     }
4191
4192     continue;
4193 #else
4194     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
4195         "ipv6only is not supported "
4196         "on this platform");
4197     return NGX_CONF_ERROR;
4198 #endif
4199 }
4200
4201     if (ngx_strcmp(value[n].data, "ssl") == 0) {
4202 #if (NGX_HTTP_SSL)
4203         lsopt.ssl = 1;
4204         continue;
4205 #else
4206         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
4207             "the \"ssl\" parameter requires "
4208             "ngx_http_ssl_module");
4209         return NGX_CONF_ERROR;
4210 #endif
4211     }
4212
4213     if (ngx_strcmp(value[n].data, "spdy") == 0) {
4214 #if (NGX_HTTP_SPDY)
4215         lsopt.spdy = 1;
4216         continue;
4217 #else
4218         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
4219             "the \"spdy\" parameter requires "
4220             "ngx_http_spdy_module");
4221         return NGX_CONF_ERROR;
4222 #endif
4223     }
4224
4225     if (ngx_strncmp(value[n].data, "so_keepalive=", 13) == 0) {
4226
4227         if (ngx_strcmp(&value[n].data[13], "on") == 0) {
4228             lsopt.so_keepalive = 1;
4229
4230         } else if (ngx_strcmp(&value[n].data[13], "off") == 0) {
4231             lsopt.so_keepalive = 2;
4232
4233         } else {
4234
4235 #if (NGX_HAVE_KEEPALIVE_TUNABLE)
4236             u_char    *p, *end;
4237             ngx_str_t  s;
4238
4239             end = value[n].data + value[n].len;
4240             s.data = value[n].data + 13;
4241
4242             p = ngx_strlchr(s.data, end, ':');
4243             if (p == NULL) {
4244                 p = end;
4245             }
4246
4247             if (p > s.data) {
4248                 s.len = p - s.data;
4249
4250                 lsopt.tcp_keepidle = ngx_parse_time(&s, 1);
4251                 if (lsopt.tcp_keepidle == (time_t) NGX_ERROR) {
4252                     goto invalid_so_keepalive;
4253                 }

```

```

4254     }
4255
4256     s.data = (p < end) ? (p + 1) : end;
4257
4258     p = ngx_strlchr(s.data, end, ':');
4259     if (p == NULL) {
4260         p = end;
4261     }
4262
4263     if (p > s.data) {
4264         s.len = p - s.data;
4265
4266         lsopt.tcp_keepintvl = ngx_parse_time(&s, 1);
4267         if (lsopt.tcp_keepintvl == (time_t) NGX_ERROR) {
4268             goto invalid_so_keepalive;
4269         }
4270     }
4271
4272     s.data = (p < end) ? (p + 1) : end;
4273
4274     if (s.data < end) {
4275         s.len = end - s.data;
4276
4277         lsopt.tcp_keepcnt = ngx_atoi(s.data, s.len);
4278         if (lsopt.tcp_keepcnt == NGX_ERROR) {
4279             goto invalid_so_keepalive;
4280         }
4281     }
4282
4283     if (lsopt.tcp_keepidle == 0 && lsopt.tcp_keepintvl == 0
4284         && lsopt.tcp_keepcnt == 0)
4285     {
4286         goto invalid_so_keepalive;
4287     }
4288
4289     lsopt.so_keepalive = 1;
4290
4291 #else
4292
4293     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
4294         "the \"so_keepalive\" parameter accepts \"
4295         \"only \"on\" or \"off\" on this platform");
4296     return NGX_CONF_ERROR;
4297
4298 #endif
4299     }
4300
4301     lsopt.set = 1;
4302     lsopt.bind = 1;
4303
4304     continue;
4305
4306 #if (NGX_HAVE_KEEPALIVE_TUNABLE)
4307     invalid_so_keepalive:
4308
4309         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
4310             "invalid so_keepalive value: \"%s\"",
4311             &value[n].data[13]);
4312     return NGX_CONF_ERROR;
4313 #endif
4314     }
4315
4316     if (ngx_strcmp(value[n].data, "proxy_protocol") == 0) {
4317         lsopt.proxy_protocol = 1;
4318         continue;
4319     }
4320
4321     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
4322         "invalid parameter \"%V\"", &value[n]);
4323     return NGX_CONF_ERROR;
4324 }
4325
4326 if (ngx_http_add_listen(cf, cscf, &lsopt) == NGX_OK) {
4327     return NGX_CONF_OK;
4328 }
4329

```

```

4330     return NGX\_CONF\_ERROR;
4331 }
4332
4333
4334 static char *
4335 ngx\_http\_core\_server\_name(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
4336 {
4337     ngx\_http\_core\_srv\_conf\_t *cscf = conf;
4338
4339     u_char          ch;
4340     ngx\_str\_t       *value;
4341     ngx\_uint\_t      i;
4342     ngx\_http\_server\_name\_t *sn;
4343
4344     value = cf->args->elts;
4345
4346     for (i = 1; i < cf->args->nelts; i++) {
4347
4348         ch = value[i].data[0];
4349
4350         if ((ch == '*' && (value[i].len < 3 || value[i].data[1] != '.'))
4351             || (ch == '.' && value[i].len < 2))
4352         {
4353             ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
4354                 "server name \"%V\" is invalid", &value[i]);
4355             return NGX\_CONF\_ERROR;
4356         }
4357
4358         if (ngx\_strchr(value[i].data, '/')) {
4359             ngx\_conf\_log\_error(NGX\_LOG\_WARN, cf, 0,
4360                 "server name \"%V\" has suspicious symbols",
4361                 &value[i]);
4362         }
4363
4364         sn = ngx\_array\_push(&cscf->server_names);
4365         if (sn == NULL) {
4366             return NGX\_CONF\_ERROR;
4367         }
4368
4369         #if (NGX_PCRE)
4370         sn->regex = NULL;
4371         #endif
4372         sn->server = cscf;
4373
4374         if (ngx\_strcasecmp(value[i].data, (u_char *) "$hostname") == 0) {
4375             sn->name = cf->cycle->hostname;
4376         } else {
4377             sn->name = value[i];
4378         }
4379
4380         if (value[i].data[0] != '~') {
4381             ngx\_strlow(sn->name.data, sn->name.data, sn->name.len);
4382             continue;
4383         }
4384
4385         #if (NGX_PCRE)
4386         {
4387             u_char          *p;
4388             ngx\_regex\_compile\_t rc;
4389             u_char          errstr[NGX\_MAX\_CONF\_ERRSTR];
4390
4391             if (value[i].len == 1) {
4392                 ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
4393                     "empty regex in server name \"%V\"", &value[i]);
4394                 return NGX\_CONF\_ERROR;
4395             }
4396
4397             value[i].len--;
4398             value[i].data++;
4399
4400             ngx\_memzero(&rc, sizeof(ngx\_regex\_compile\_t));
4401
4402             rc.pattern = value[i];
4403             rc.err.len = NGX\_MAX\_CONF\_ERRSTR;
4404             rc.err.data = errstr;

```

```

4406     for (p = value[i].data; p < value[i].data + value[i].len; p++) {
4407         if (*p >= 'A' && *p <= 'Z') {
4408             rc.options = NGX\_REGEX\_CASELESS;
4409             break;
4410         }
4411     }
4412 }
4413
4414 sn->regex = ngx\_http\_regex\_compile(cf, &rc);
4415 if (sn->regex == NULL) {
4416     return NGX\_CONF\_ERROR;
4417 }
4418
4419 sn->name = value[i];
4420 cscf->captures = (rc.captures > 0);
4421 }
4422 #else
4423     ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
4424         "using regex \"%V\" "
4425         "requires PCRE library", &value[i]);
4426
4427     return NGX\_CONF\_ERROR;
4428 #endif
4429 }
4430
4431 return NGX\_CONF\_OK;
4432 }
4433
4434
4435 static char *
4436 ngx\_http\_core\_root(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
4437 {
4438     ngx\_http\_core\_loc\_conf\_t *clcf = conf;
4439
4440     ngx\_str\_t          *value;
4441     ngx\_int\_t          alias;
4442     ngx\_uint\_t         n;
4443     ngx\_http\_script\_compile\_t  sc;
4444
4445     alias = (cmd->name.len == sizeof("alias") - 1) ? 1 : 0;
4446
4447     if (clcf->root.data) {
4448
4449         if ((clcf->alias != 0) == alias) {
4450             ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
4451                 "\"%V\" directive is duplicate",
4452                 &cmd->name);
4453         } else {
4454             ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
4455                 "\"%V\" directive is duplicate, "
4456                 "\"%s\" directive was specified earlier",
4457                 &cmd->name, clcf->alias ? "alias" : "root");
4458         }
4459
4460         return NGX\_CONF\_ERROR;
4461     }
4462
4463     if (clcf->named && alias) {
4464         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
4465             "the \"%alias\" directive cannot be used "
4466             "inside the named location");
4467
4468         return NGX\_CONF\_ERROR;
4469     }
4470
4471     value = cf->args->elts;
4472
4473     if (ngx\_strstr(value[1].data, "$document_root")
4474         || ngx\_strstr(value[1].data, "${document_root}"))
4475     {
4476         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
4477             "the $document_root variable cannot be used "
4478             "in the \"%V\" directive",
4479             &cmd->name);
4480
4481         return NGX\_CONF\_ERROR;

```

```

4482     }
4483
4484     if (ngx_strstr(value[1].data, "$realpath_root")
4485         || ngx_strstr(value[1].data, "${realpath_root}"))
4486     {
4487         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
4488             "the $realpath_root variable cannot be used "
4489             "in the \"%V\" directive",
4490             &cmd->name);
4491
4492         return NGX_CONF_ERROR;
4493     }
4494
4495     clcf->alias = alias ? clcf->name.len : 0;
4496     clcf->root = value[1];
4497
4498     if (!alias && clcf->root.data[clcf->root.len - 1] == '/') {
4499         clcf->root.len--;
4500     }
4501
4502     if (clcf->root.data[0] != '$') {
4503         if (ngx_conf_full_name(cf->cycle, &clcf->root, 0) != NGX_OK) {
4504             return NGX_CONF_ERROR;
4505         }
4506     }
4507
4508     n = ngx_http_script_variables_count(&clcf->root);
4509
4510     ngx_memzero(&sc, sizeof(ngx_http_script_compile_t));
4511     sc.variables = n;
4512
4513     #if (NGX_PCRE)
4514     if (alias && clcf->regex) {
4515         clcf->alias = NGX_MAX_SIZE_T_VALUE;
4516         n = 1;
4517     }
4518     #endif
4519
4520     if (n) {
4521         sc.cf = cf;
4522         sc.source = &clcf->root;
4523         sc.lengths = &clcf->root_lengths;
4524         sc.values = &clcf->root_values;
4525         sc.complete_lengths = 1;
4526         sc.complete_values = 1;
4527
4528         if (ngx_http_script_compile(&sc) != NGX_OK) {
4529             return NGX_CONF_ERROR;
4530         }
4531     }
4532
4533     return NGX_CONF_OK;
4534 }
4535
4536
4537 static ngx_http_method_name_t  ngx_methods_names[] = {
4538     { (u_char *) "GET",          (uint32_t) ~NGX_HTTP_GET },
4539     { (u_char *) "HEAD",        (uint32_t) ~NGX_HTTP_HEAD },
4540     { (u_char *) "POST",        (uint32_t) ~NGX_HTTP_POST },
4541     { (u_char *) "PUT",          (uint32_t) ~NGX_HTTP_PUT },
4542     { (u_char *) "DELETE",      (uint32_t) ~NGX_HTTP_DELETE },
4543     { (u_char *) "MKCOL",        (uint32_t) ~NGX_HTTP_MKCOL },
4544     { (u_char *) "COPY",         (uint32_t) ~NGX_HTTP_COPY },
4545     { (u_char *) "MOVE",         (uint32_t) ~NGX_HTTP_MOVE },
4546     { (u_char *) "OPTIONS",      (uint32_t) ~NGX_HTTP_OPTIONS },
4547     { (u_char *) "PROPFIND",     (uint32_t) ~NGX_HTTP_PROPFIND },
4548     { (u_char *) "PROPPATCH",   (uint32_t) ~NGX_HTTP_PROPPATCH },
4549     { (u_char *) "LOCK",         (uint32_t) ~NGX_HTTP_LOCK },
4550     { (u_char *) "UNLOCK",       (uint32_t) ~NGX_HTTP_UNLOCK },
4551     { (u_char *) "PATCH",       (uint32_t) ~NGX_HTTP_PATCH },
4552     { NULL, 0 }
4553 };
4554
4555
4556 static char *
4557 ngx_http_core_limit_except(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)

```

```

4558 {
4559     ngx\_http\_core\_loc\_conf\_t *pclcf = conf;
4560
4561     char                *rv;
4562     void                *mconf;
4563     ngx\_str\_t           *value;
4564     ngx\_uint\_t          i;
4565     ngx\_conf\_t         save;
4566     ngx\_http\_module\_t  *module;
4567     ngx\_http\_conf\_ctx\_t *ctx, *pctx;
4568     ngx\_http\_method\_name\_t *name;
4569     ngx\_http\_core\_loc\_conf\_t *clcf;
4570
4571     if (pclcf->limit_except) {
4572         return "duplicate";
4573     }
4574
4575     pclcf->limit_except = 0xffffffff;
4576
4577     value = cf->args->elts;
4578
4579     for (i = 1; i < cf->args->nelts; i++) {
4580         for (name = ngx\_methods\_names; name->name; name++) {
4581
4582             if (ngx\_strcasecmp(value[i].data, name->name) == 0) {
4583                 pclcf->limit_except &= name->method;
4584                 goto next;
4585             }
4586         }
4587
4588         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
4589             "invalid method \"%V\"", &value[i]);
4590         return NGX\_CONF\_ERROR;
4591
4592     next:
4593         continue;
4594     }
4595
4596     if (!(pclcf->limit_except & NGX\_HTTP\_GET)) {
4597         pclcf->limit_except &= (uint32_t) ~NGX\_HTTP\_HEAD;
4598     }
4599
4600     ctx = ngx\_palloc(cf->pool, sizeof(ngx\_http\_conf\_ctx\_t));
4601     if (ctx == NULL) {
4602         return NGX\_CONF\_ERROR;
4603     }
4604
4605     pctx = cf->ctx;
4606     ctx->main_conf = pctx->main_conf;
4607     ctx->srv_conf = pctx->srv_conf;
4608
4609     ctx->loc_conf = ngx\_palloc(cf->pool, sizeof(void *) * ngx\_http\_max\_module);
4610     if (ctx->loc_conf == NULL) {
4611         return NGX\_CONF\_ERROR;
4612     }
4613
4614     for (i = 0; ngx_modules[i]; i++) {
4615         if (ngx_modules[i]->type != NGX\_HTTP\_MODULE) {
4616             continue;
4617         }
4618
4619         module = ngx_modules[i]->ctx;
4620
4621         if (module->create_loc_conf) {
4622             mconf = module->create_loc_conf(cf);
4623             if (mconf == NULL) {
4624                 return NGX\_CONF\_ERROR;
4625             }
4626
4627             ctx->loc_conf[ngx_modules[i]->ctx_index] = mconf;
4628         }
4629     }
4630 }
4631
4632
4633     clcf = ctx->loc_conf[ngx\_http\_core\_module.ctx_index];

```

```

4634     pclcf->limit_except_loc_conf = ctx->loc_conf;
4635     clcf->loc_conf = ctx->loc_conf;
4636     clcf->name = pclcf->name;
4637     clcf->noname = 1;
4638     clcf->lmt_excpt = 1;
4639
4640     if (ngx_http_add_location(cf, &pclcf->locations, clcf) != NGX_OK) {
4641         return NGX_CONF_ERROR;
4642     }
4643
4644     save = *cf;
4645     cf->ctx = ctx;
4646     cf->cmd_type = NGX_HTTP_LMT_CONF;
4647
4648     rv = ngx_conf_parse(cf, NULL);
4649
4650     *cf = save;
4651
4652     return rv;
4653 }
4654
4655
4656 static char *
4657 ngx_http_core_directio(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
4658 {
4659     ngx_http_core_loc_conf_t *clcf = conf;
4660
4661     ngx_str_t *value;
4662
4663     if (clcf->directio != NGX_CONF_UNSET) {
4664         return "is duplicate";
4665     }
4666
4667     value = cf->args->elts;
4668
4669     if (ngx_strcmp(value[1].data, "off") == 0) {
4670         clcf->directio = NGX_OPEN_FILE_DIRECTIO_OFF;
4671         return NGX_CONF_OK;
4672     }
4673
4674     clcf->directio = ngx_parse_offset(&value[1]);
4675     if (clcf->directio == (off_t) NGX_ERROR) {
4676         return "invalid value";
4677     }
4678
4679     return NGX_CONF_OK;
4680 }
4681
4682
4683 static char *
4684 ngx_http_core_error_page(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
4685 {
4686     ngx_http_core_loc_conf_t *clcf = conf;
4687
4688     u_char *p;
4689     ngx_int_t overwrite;
4690     ngx_str_t *value, uri, args;
4691     ngx_uint_t i, n;
4692     ngx_http_err_page_t *err;
4693     ngx_http_complex_value_t cv;
4694     ngx_http_compile_complex_value_t ccv;
4695
4696     if (clcf->error_pages == NULL) {
4697         clcf->error_pages = ngx_array_create(cf->pool, 4,
4698             sizeof(ngx_http_err_page_t));
4699         if (clcf->error_pages == NULL) {
4700             return NGX_CONF_ERROR;
4701         }
4702     }
4703
4704     value = cf->args->elts;
4705
4706     i = cf->args->nelts - 2;
4707
4708     if (value[i].data[0] == '=') {
4709         if (i == 1) {

```



```

4710     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
4711         "invalid value \"%V\"", &value[i]);
4712     return NGX_CONF_ERROR;
4713 }
4714
4715 if (value[i].len > 1) {
4716     overwrite = ngx_atoi(&value[i].data[1], value[i].len - 1);
4717
4718     if (overwrite == NGX_ERROR) {
4719         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
4720             "invalid value \"%V\"", &value[i]);
4721         return NGX_CONF_ERROR;
4722     }
4723
4724 } else {
4725     overwrite = 0;
4726 }
4727
4728 n = 2;
4729
4730 } else {
4731     overwrite = -1;
4732     n = 1;
4733 }
4734
4735 uri = value[cf->args->nelts - 1];
4736
4737 ngx_memzero(&ccv, sizeof(ngx_http_compile_complex_value_t));
4738
4739 ccv.cf = cf;
4740 ccv.value = &uri;
4741 ccv.complex_value = &cv;
4742
4743 if (ngx_http_compile_complex_value(&ccv) != NGX_OK) {
4744     return NGX_CONF_ERROR;
4745 }
4746
4747 ngx_str_null(&args);
4748
4749 if (cv.lengths == NULL && uri.len && uri.data[0] == '/') {
4750     p = (u_char *) ngx_strchr(uri.data, '?');
4751
4752     if (p) {
4753         cv.value.len = p - uri.data;
4754         cv.value.data = uri.data;
4755         p++;
4756         args.len = (uri.data + uri.len) - p;
4757         args.data = p;
4758     }
4759 }
4760
4761 for (i = 1; i < cf->args->nelts - n; i++) {
4762     err = ngx_array_push(&clcf->error_pages);
4763     if (err == NULL) {
4764         return NGX_CONF_ERROR;
4765     }
4766
4767     err->status = ngx_atoi(value[i].data, value[i].len);
4768
4769     if (err->status == NGX_ERROR || err->status == 499) {
4770         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
4771             "invalid value \"%V\"", &value[i]);
4772         return NGX_CONF_ERROR;
4773     }
4774
4775     if (err->status < 300 || err->status > 599) {
4776         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
4777             "value \"%V\" must be between 300 and 599",
4778             &value[i]);
4779         return NGX_CONF_ERROR;
4780     }
4781
4782     err->overwrite = overwrite;
4783
4784     if (overwrite == -1) {
4785         switch (err->status) {

```

```

4786     case NGX\_HTTP\_TO\_HTTPS:
4787     case NGX\_HTTPS\_CERT\_ERROR:
4788     case NGX\_HTTPS\_NO\_CERT:
4789         err->overwrite = NGX\_HTTP\_BAD\_REQUEST;
4790     default:
4791         break;
4792     }
4793 }
4794
4795     err->value = cv;
4796     err->args = args;
4797 }
4798
4799     return NGX\_CONF\_OK;
4800 }
4801
4802
4803 static char *
4804 ngx\_http\_core\_try\_files(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
4805 {
4806     ngx\_http\_core\_loc\_conf\_t *clcf = conf;
4807
4808     ngx\_str\_t          *value;
4809     ngx\_int\_t         code;
4810     ngx\_uint\_t        i, n;
4811     ngx\_http\_try\_file\_t *tf;
4812     ngx\_http\_script\_compile\_t sc;
4813     ngx\_http\_core\_main\_conf\_t *cmcf;
4814
4815     if (clcf->try_files) {
4816         return "is duplicate";
4817     }
4818
4819     cmcf = ngx\_http\_conf\_get\_module\_main\_conf(cf, ngx\_http\_core\_module);
4820
4821     cmcf->try_files = 1;
4822
4823     tf = ngx\_palloc(cf->pool, cf->args->nelts * sizeof(ngx\_http\_try\_file\_t));
4824     if (tf == NULL) {
4825         return NGX\_CONF\_ERROR;
4826     }
4827
4828     clcf->try_files = tf;
4829
4830     value = cf->args->elts;
4831
4832     for (i = 0; i < cf->args->nelts - 1; i++) {
4833
4834         tf[i].name = value[i + 1];
4835
4836         if (tf[i].name.len > 0
4837             && tf[i].name.data[tf[i].name.len - 1] == '/'
4838             && i + 2 < cf->args->nelts)
4839         {
4840             tf[i].test_dir = 1;
4841             tf[i].name.len--;
4842             tf[i].name.data[tf[i].name.len] = '\\0';
4843         }
4844
4845         n = ngx\_http\_script\_variables\_count(&tf[i].name);
4846
4847         if (n) {
4848             ngx\_memzero(&sc, sizeof(ngx\_http\_script\_compile\_t));
4849
4850             sc.cf = cf;
4851             sc.source = &tf[i].name;
4852             sc.lengths = &tf[i].lengths;
4853             sc.values = &tf[i].values;
4854             sc.variables = n;
4855             sc.complete_lengths = 1;
4856             sc.complete_values = 1;
4857
4858             if (ngx\_http\_script\_compile(&sc) != NGX\_OK) {
4859                 return NGX\_CONF\_ERROR;
4860             }
4861

```

```

4862     } else {
4863         /* add trailing '\0' to length */
4864         tf[i].name.len++;
4865     }
4866 }
4867
4868 if (tf[i - 1].name.data[0] == '=') {
4869
4870     code = ngx_atoi(tf[i - 1].name.data + 1, tf[i - 1].name.len - 2);
4871
4872     if (code == NGX_ERROR || code > 999) {
4873         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
4874             "invalid code \"%s\"",
4875             tf[i - 1].name.data + 1, tf[i - 1].name.len - 2);
4876         return NGX_CONF_ERROR;
4877     }
4878
4879     tf[i].code = code;
4880 }
4881
4882 return NGX_CONF_OK;
4883 }
4884
4885
4886 static char *
4887 ngx_http_core_open_file_cache(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
4888 {
4889     ngx_http_core_loc_conf_t *clcf = conf;
4890
4891     time_t    inactive;
4892     ngx_str_t *value, s;
4893     ngx_int_t max;
4894     ngx_uint_t i;
4895
4896     if (clcf->open_file_cache != NGX_CONF_UNSET_PTR) {
4897         return "is duplicate";
4898     }
4899
4900     value = cf->args->elts;
4901
4902     max = 0;
4903     inactive = 60;
4904
4905     for (i = 1; i < cf->args->nelts; i++) {
4906
4907         if (ngx_strncmp(value[i].data, "max=", 4) == 0) {
4908
4909             max = ngx_atoi(value[i].data + 4, value[i].len - 4);
4910             if (max <= 0) {
4911                 goto failed;
4912             }
4913
4914             continue;
4915         }
4916
4917         if (ngx_strncmp(value[i].data, "inactive=", 9) == 0) {
4918
4919             s.len = value[i].len - 9;
4920             s.data = value[i].data + 9;
4921
4922             inactive = ngx_parse_time(&s, 1);
4923             if (inactive == (time_t) NGX_ERROR) {
4924                 goto failed;
4925             }
4926
4927             continue;
4928         }
4929
4930         if (ngx_strcmp(value[i].data, "off") == 0) {
4931
4932             clcf->open_file_cache = NULL;
4933
4934             continue;
4935         }
4936
4937     failed:

```

```

4938     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
4939         "invalid \"open_file_cache\" parameter \"%V\"",
4940         &value[i]);
4941     return NGX_CONF_ERROR;
4942 }
4943
4944
4945 if (clcf->open_file_cache == NULL) {
4946     return NGX_CONF_OK;
4947 }
4948
4949 if (max == 0) {
4950     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
4951         "\"open_file_cache\" must have the \"max\" parameter");
4952     return NGX_CONF_ERROR;
4953 }
4954
4955 clcf->open_file_cache = ngx_open_file_cache_init(cf->pool, max, inactive);
4956 if (clcf->open_file_cache) {
4957     return NGX_CONF_OK;
4958 }
4959
4960 return NGX_CONF_ERROR;
4961 }
4962
4963
4964 static char *
4965 ngx_http_core_error_log(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
4966 {
4967     ngx_http_core_loc_conf_t *clcf = conf;
4968
4969     return ngx_log_set_log(cf, &clcf->error_log);
4970 }
4971
4972
4973 static char *
4974 ngx_http_core_keepalive(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
4975 {
4976     ngx_http_core_loc_conf_t *clcf = conf;
4977
4978     ngx_str_t *value;
4979
4980     if (clcf->keepalive_timeout != NGX_CONF_UNSET_MSEC) {
4981         return "is duplicate";
4982     }
4983
4984     value = cf->args->elts;
4985
4986     clcf->keepalive_timeout = ngx_parse_time(&value[1], 0);
4987
4988     if (clcf->keepalive_timeout == (ngx_msec_t) NGX_ERROR) {
4989         return "invalid value";
4990     }
4991
4992     if (cf->args->nelts == 2) {
4993         return NGX_CONF_OK;
4994     }
4995
4996     clcf->keepalive_header = ngx_parse_time(&value[2], 1);
4997
4998     if (clcf->keepalive_header == (time_t) NGX_ERROR) {
4999         return "invalid value";
5000     }
5001
5002     return NGX_CONF_OK;
5003 }
5004
5005
5006 static char *
5007 ngx_http_core_internal(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
5008 {
5009     ngx_http_core_loc_conf_t *clcf = conf;
5010
5011     if (clcf->internal != NGX_CONF_UNSET) {
5012         return "is duplicate";
5013     }

```

```

5014     clcf->internal = 1;
5015
5016     return NGX_CONF_OK;
5017 }
5018
5019
5020
5021 static char *
5022 ngx_http_core_resolver(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
5023 {
5024     ngx_http_core_loc_conf_t *clcf = conf;
5025
5026     ngx_str_t *value;
5027
5028     if (clcf->resolver) {
5029         return "is duplicate";
5030     }
5031
5032     value = cf->args->elts;
5033
5034     clcf->resolver = ngx_resolver_create(cf, &value[1], cf->args->nelts - 1);
5035     if (clcf->resolver == NULL) {
5036         return NGX_CONF_ERROR;
5037     }
5038
5039     return NGX_CONF_OK;
5040 }
5041
5042
5043 #if (NGX_HTTP_GZIP)
5044
5045 static char *
5046 ngx_http_gzip_disable(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
5047 {
5048     ngx_http_core_loc_conf_t *clcf = conf;
5049
5050     #if (NGX_PCRE)
5051
5052     ngx_str_t *value;
5053     ngx_uint_t i;
5054     ngx_regex_elt_t *re;
5055     ngx_regex_compile_t rc;
5056     u_char errstr[NGX_MAX_CONF_ERRSTR];
5057
5058     if (clcf->gzip_disable == NGX_CONF_UNSET_PTR) {
5059         clcf->gzip_disable = ngx_array_create(cf->pool, 2,
5060                                             sizeof(ngx_regex_elt_t));
5061         if (clcf->gzip_disable == NULL) {
5062             return NGX_CONF_ERROR;
5063         }
5064     }
5065
5066     value = cf->args->elts;
5067
5068     ngx_memzero(&rc, sizeof(ngx_regex_compile_t));
5069
5070     rc.pool = cf->pool;
5071     rc.err.len = NGX_MAX_CONF_ERRSTR;
5072     rc.err.data = errstr;
5073
5074     for (i = 1; i < cf->args->nelts; i++) {
5075
5076         if (ngx_strcmp(value[i].data, "msie6") == 0) {
5077             clcf->gzip_disable_msie6 = 1;
5078             continue;
5079         }
5080
5081     #if (NGX_HTTP_DEGRADATION)
5082
5083         if (ngx_strcmp(value[i].data, "degradation") == 0) {
5084             clcf->gzip_disable_degradation = 1;
5085             continue;
5086         }
5087
5088     #endif
5089

```

```

5090     re = ngx_array_push(&clcf->gzip_disable);
5091     if (re == NULL) {
5092         return NGX_CONF_ERROR;
5093     }
5094
5095     rc.pattern = value[i];
5096     rc.options = NGX_REGEX_CASELESS;
5097
5098     if (ngx_regex_compile(&rc) != NGX_OK) {
5099         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0, "%V", &rc.err);
5100         return NGX_CONF_ERROR;
5101     }
5102
5103     re->regex = rc.regex;
5104     re->name = value[i].data;
5105 }
5106
5107 return NGX_CONF_OK;
5108
5109 #else
5110     ngx_str_t    *value;
5111     ngx_uint_t    i;
5112
5113     value = cf->args->elts;
5114
5115     for (i = 1; i < cf->args->nelts; i++) {
5116         if (ngx_strcmp(value[i].data, "msie6") == 0) {
5117             clcf->gzip_disable_msie6 = 1;
5118             continue;
5119         }
5120
5121         #if (NGX_HTTP_DEGRADATION)
5122
5123             if (ngx_strcmp(value[i].data, "degradation") == 0) {
5124                 clcf->gzip_disable_degradation = 1;
5125                 continue;
5126             }
5127
5128         #endif
5129
5130         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
5131             "without PCRE library \"gzip_disable\" supports "
5132             "builtin \"msie6\" and \"degradation\" mask only");
5133
5134         return NGX_CONF_ERROR;
5135     }
5136
5137     return NGX_CONF_OK;
5138
5139 #endif
5140 }
5141
5142 #endif
5143
5144 #if (NGX_HAVE_OPENAT)
5145
5146 static char *
5147 ngx_http_disable_symlinks(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
5148 {
5149     ngx_http_core_loc_conf_t *clcf = conf;
5150
5151     ngx_str_t    *value;
5152     ngx_uint_t    i;
5153     ngx_http_compile_complex_value_t    ccv;
5154
5155     if (clcf->disable_symlinks != NGX_CONF_UNSET_UINT) {
5156         return "is duplicate";
5157     }
5158
5159     value = cf->args->elts;
5160
5161     for (i = 1; i < cf->args->nelts; i++) {
5162         if (ngx_strcmp(value[i].data, "off") == 0) {
5163             clcf->disable_symlinks = NGX_DISABLE_SYMLINKS_OFF;

```

```

5166     continue;
5167 }
5168
5169 if (ngx_strcmp(value[i].data, "if_not_owner") == 0) {
5170     clcf->disable_symlinks = NGX\_DISABLE\_SYMLINKS\_NOTOWNER;
5171     continue;
5172 }
5173
5174 if (ngx_strcmp(value[i].data, "on") == 0) {
5175     clcf->disable_symlinks = NGX\_DISABLE\_SYMLINKS\_ON;
5176     continue;
5177 }
5178
5179 if (ngx_strncmp(value[i].data, "from=", 5) == 0) {
5180     value[i].len -= 5;
5181     value[i].data += 5;
5182
5183     ngx\_memzero(&ccv, sizeof(ngx\_http\_compile\_complex\_value\_t));
5184
5185     ccv.cf = cf;
5186     ccv.value = &value[i];
5187     ccv.complex_value = ngx\_palloc(cf->pool,
5188                                     sizeof(ngx\_http\_compile\_complex\_value\_t));
5189     if (ccv.complex_value == NULL) {
5190         return NGX\_CONF\_ERROR;
5191     }
5192
5193     if (ngx\_http\_compile\_complex\_value(&ccv) != NGX\_OK) {
5194         return NGX\_CONF\_ERROR;
5195     }
5196
5197     clcf->disable_symlinks_from = ccv.complex_value;
5198
5199     continue;
5200 }
5201
5202 ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
5203                    "invalid parameter \"%V\"", &value[i]);
5204 return NGX\_CONF\_ERROR;
5205 }
5206
5207 if (clcf->disable_symlinks == NGX\_CONF\_UNSET\_UINT) {
5208     ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
5209                        "\"%V\" must have \"off\", \"on\" "
5210                        "or \"if_not_owner\" parameter",
5211                        &cmd->name);
5212     return NGX\_CONF\_ERROR;
5213 }
5214
5215 if (cf->args->nelts == 2) {
5216     clcf->disable_symlinks_from = NULL;
5217     return NGX\_CONF\_OK;
5218 }
5219
5220 if (clcf->disable_symlinks_from == NGX\_CONF\_UNSET\_PTR) {
5221     ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
5222                        "duplicate parameters \"%V %V\"",
5223                        &value[1], &value[2]);
5224     return NGX\_CONF\_ERROR;
5225 }
5226
5227 if (clcf->disable_symlinks == NGX\_DISABLE\_SYMLINKS\_OFF) {
5228     ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
5229                        "\"from=\" cannot be used with \"off\" parameter");
5230     return NGX\_CONF\_ERROR;
5231 }
5232
5233 return NGX\_CONF\_OK;
5234 }
5235
5236 #endif
5237
5238
5239 static char *
5240 ngx\_http\_core\_lowat\_check(ngx\_conf\_t *cf, void *post, void *data)
5241 {

```

```

5242 #if (NGX_FREEBSD)
5243     ssize_t *np = data;
5244
5245     if ((u_long) *np >= ngx_freebsd_net_inet_tcp_sendspace) {
5246         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
5247             "\"send_lowat\" must be less than %d ",
5248             "(sysctl net.inet.tcp.sendspace)",
5249             ngx_freebsd_net_inet_tcp_sendspace);
5250
5251         return NGX_CONF_ERROR;
5252     }
5253
5254 #elif !(NGX_HAVE_SO_SNDLOWAT)
5255     ssize_t *np = data;
5256
5257     ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
5258         "\"send_lowat\" is not supported, ignored");
5259
5260     *np = 0;
5261
5262 #endif
5263
5264     return NGX_CONF_OK;
5265 }
5266
5267 static char *
5268 ngx_http_core_pool_size(ngx_conf_t *cf, void *post, void *data)
5269 {
5270     size_t *sp = data;
5271
5272     if (*sp < NGX_MIN_POOL_SIZE) {
5273         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
5274             "the pool size must be no less than %uz",
5275             NGX_MIN_POOL_SIZE);
5276         return NGX_CONF_ERROR;
5277     }
5278
5279     if (*sp % NGX_POOL_ALIGNMENT) {
5280         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
5281             "the pool size must be a multiple of %uz",
5282             NGX_POOL_ALIGNMENT);
5283         return NGX_CONF_ERROR;
5284     }
5285
5286     return NGX_CONF_OK;
5287 }
5288 }

```

[One Level Up](#)

[Top Level](#)



## src/http/nginx\_http\_request\_body.c - nginx-1.7.10

### Functions defined

- [ngx\\_http\\_discard\\_request\\_body](#)
- [ngx\\_http\\_discard\\_request\\_body\\_filter](#)
- [ngx\\_http\\_discarded\\_request\\_body\\_handler](#)
- [ngx\\_http\\_do\\_read\\_client\\_request\\_body](#)
- [ngx\\_http\\_read\\_client\\_request\\_body](#)
- [ngx\\_http\\_read\\_client\\_request\\_body\\_handler](#)
- [ngx\\_http\\_read\\_discarded\\_request\\_body](#)
- [ngx\\_http\\_request\\_body\\_chunked\\_filter](#)
- [ngx\\_http\\_request\\_body\\_filter](#)
- [ngx\\_http\\_request\\_body\\_length\\_filter](#)
- [ngx\\_http\\_request\\_body\\_save\\_filter](#)
- [ngx\\_http\\_test\\_expect](#)
- [ngx\\_http\\_write\\_request\\_body](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 static void ngx_http_read_client_request_body_handler(ngx_http_request_t *r);
14 static ngx_int_t ngx_http_do_read_client_request_body(ngx_http_request_t *r);
15 static ngx_int_t ngx_http_write_request_body(ngx_http_request_t *r);
16 static ngx_int_t ngx_http_read_discarded_request_body(ngx_http_request_t *r);
17 static ngx_int_t ngx_http_discard_request_body_filter(ngx_http_request_t *r,
18     ngx_buf_t *b);
19 static ngx_int_t ngx_http_test_expect(ngx_http_request_t *r);
20
21 static ngx_int_t ngx_http_request_body_filter(ngx_http_request_t *r,
22     ngx_chain_t *in);
23 static ngx_int_t ngx_http_request_body_length_filter(ngx_http_request_t *r,
24     ngx_chain_t *in);
25 static ngx_int_t ngx_http_request_body_chunked_filter(ngx_http_request_t *r,
26     ngx_chain_t *in);
27 static ngx_int_t ngx_http_request_body_save_filter(ngx_http_request_t *r,
28     ngx_chain_t *in);
29
30
31 ngx_int_t
32 ngx_http_read_client_request_body(ngx_http_request_t *r,
33     ngx_http_client_body_handler_pt post_handler)
34 {
35     size_t          preread;
```

```

36     ssize_t             size;
37     ngx_int_t          rc;
38     ngx_buf_t          *b;
39     ngx_chain_t        out, *cl;
40     ngx_http_request_body_t *rb;
41     ngx_http_core_loc_conf_t *clcf;
42
43     r->main->count++;
44
45     #if (NGX_HTTP_SPDY)
46     if (r->spdy_stream && r == r->main) {
47         rc = ngx_http_spdy_read_request_body(r, post_handler);
48         goto done;
49     }
50     #endif
51
52     if (r != r->main || r->request_body || r->discard_body) {
53         post_handler(r);
54         return NGX_OK;
55     }
56
57     if (ngx_http_test_expect(r) != NGX_OK) {
58         rc = NGX_HTTP_INTERNAL_SERVER_ERROR;
59         goto done;
60     }
61
62     rb = ngx_palloc(r->pool, sizeof(ngx_http_request_body_t));
63     if (rb == NULL) {
64         rc = NGX_HTTP_INTERNAL_SERVER_ERROR;
65         goto done;
66     }
67
68     /*
69     * set by ngx_palloc():
70     *
71     *     rb->bufs = NULL;
72     *     rb->buf = NULL;
73     *     rb->free = NULL;
74     *     rb->busy = NULL;
75     *     rb->chunked = NULL;
76     */
77
78     rb->rest = -1;
79     rb->post_handler = post_handler;
80
81     r->request_body = rb;
82
83     if (r->headers_in.content_length_n < 0 && !r->headers_in.chunked) {
84         post_handler(r);
85         return NGX_OK;
86     }
87
88     preread = r->header_in->last - r->header_in->pos;
89
90     if (preread) {
91
92         /* there is the pre-read part of the request body */
93
94         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
95             "http client request body preread %uz", preread);
96
97         out.buf = r->header_in;
98         out.next = NULL;
99
100        rc = ngx_http_request_body_filter(r, &out);
101
102        if (rc != NGX_OK) {
103            goto done;
104        }
105
106        r->request_length += preread - (r->header_in->last - r->header_in->pos);
107
108        if (!r->headers_in.chunked
109            && rb->rest > 0
110            && rb->rest <= (off_t) (r->header_in->end - r->header_in->last))
111        {

```

```

112     /* the whole request body may be placed in r->header_in */
113
114     b = ngx_alloc_buf(r->pool);
115     if (b == NULL) {
116         rc = NGX_HTTP_INTERNAL_SERVER_ERROR;
117         goto done;
118     }
119
120     b->temporary = 1;
121     b->start = r->header_in->pos;
122     b->pos = r->header_in->pos;
123     b->last = r->header_in->last;
124     b->end = r->header_in->end;
125
126     rb->buf = b;
127
128     r->read_event_handler = ngx_http_read_client_request_body_handler;
129     r->write_event_handler = ngx_http_request_empty_handler;
130
131     rc = ngx_http_do_read_client_request_body(r);
132     goto done;
133 }
134
135 } else {
136     /* set rb->rest */
137
138     if (ngx_http_request_body_filter(r, NULL) != NGX_OK) {
139         rc = NGX_HTTP_INTERNAL_SERVER_ERROR;
140         goto done;
141     }
142 }
143
144 if (rb->rest == 0) {
145     /* the whole request body was pre-read */
146
147     if (r->request_body_in_file_only) {
148         if (ngx_http_write_request_body(r) != NGX_OK) {
149             rc = NGX_HTTP_INTERNAL_SERVER_ERROR;
150             goto done;
151         }
152
153         if (rb->temp_file->file.offset != 0) {
154
155             cl = ngx_chain_get_free_buf(r->pool, &rb->free);
156             if (cl == NULL) {
157                 rc = NGX_HTTP_INTERNAL_SERVER_ERROR;
158                 goto done;
159             }
160
161             b = cl->buf;
162
163             ngx_memzero(b, sizeof(ngx_buf_t));
164
165             b->in_file = 1;
166             b->file_last = rb->temp_file->file.offset;
167             b->file = &rb->temp_file->file;
168
169             rb->bufs = cl;
170
171         } else {
172             rb->bufs = NULL;
173         }
174     }
175
176     post_handler(r);
177
178     return NGX_OK;
179 }
180
181 if (rb->rest < 0) {
182     ngx_log_error(NGX_LOG_ALERT, r->connection->log, 0,
183                 "negative request body rest");
184     rc = NGX_HTTP_INTERNAL_SERVER_ERROR;
185     goto done;
186 }
187

```

```

188     clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
189
190     size = clcf->client_body_buffer_size;
191     size += size >> 2;
192
193     /* TODO: honor r->request_body_in_single_buf */
194
195     if (!r->headers_in.chunked && rb->rest < size) {
196         size = (ssize_t) rb->rest;
197
198         if (r->request_body_in_single_buf) {
199             size += preread;
200         }
201
202     } else {
203         size = clcf->client_body_buffer_size;
204     }
205
206     rb->buf = ngx_create_temp_buf(r->pool, size);
207     if (rb->buf == NULL) {
208         rc = NGX_HTTP_INTERNAL_SERVER_ERROR;
209         goto done;
210     }
211
212     r->read_event_handler = ngx_http_read_client_request_body_handler;
213     r->write_event_handler = ngx_http_request_empty_handler;
214
215     rc = ngx_http_do_read_client_request_body(r);
216
217 done:
218
219     if (rc >= NGX_HTTP_SPECIAL_RESPONSE) {
220         r->main->count--;
221     }
222
223     return rc;
224 }
225
226
227 static void
228 ngx_http_read_client_request_body_handler(ngx_http_request_t *r)
229 {
230     ngx_int_t rc;
231
232     if (r->connection->read->timedout) {
233         r->connection->timedout = 1;
234         ngx_http_finalize_request(r, NGX_HTTP_REQUEST_TIME_OUT);
235         return;
236     }
237
238     rc = ngx_http_do_read_client_request_body(r);
239
240     if (rc >= NGX_HTTP_SPECIAL_RESPONSE) {
241         ngx_http_finalize_request(r, rc);
242     }
243 }
244
245
246 static ngx_int_t
247 ngx_http_do_read_client_request_body(ngx_http_request_t *r)
248 {
249     off_t rest;
250     size_t size;
251     ssize_t n;
252     ngx_int_t rc;
253     ngx_buf_t *b;
254     ngx_chain_t *cl, out;
255     ngx_connection_t *c;
256     ngx_http_request_body_t *rb;
257     ngx_http_core_loc_conf_t *clcf;
258
259     c = r->connection;
260     rb = r->request_body;
261
262     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, c->log, 0,
263                  "http read client request body");

```

```

264
265 for ( ;; ) {
266     for ( ;; ) {
267         if (rb->buf->last == rb->buf->end) {
268
269             /* pass buffer to request body filter chain */
270
271             out.buf = rb->buf;
272             out.next = NULL;
273
274             rc = ngx_http_request_body_filter(r, &out);
275
276             if (rc != NGX_OK) {
277                 return rc;
278             }
279
280             /* write to file */
281
282             if (ngx_http_write_request_body(r) != NGX_OK) {
283                 return NGX_HTTP_INTERNAL_SERVER_ERROR;
284             }
285
286             /* update chains */
287
288             rc = ngx_http_request_body_filter(r, NULL);
289
290             if (rc != NGX_OK) {
291                 return rc;
292             }
293
294             if (rb->busy != NULL) {
295                 return NGX_HTTP_INTERNAL_SERVER_ERROR;
296             }
297
298             rb->buf->pos = rb->buf->start;
299             rb->buf->last = rb->buf->start;
300         }
301
302         size = rb->buf->end - rb->buf->last;
303         rest = rb->rest - (rb->buf->last - rb->buf->pos);
304
305         if ((off_t) size > rest) {
306             size = (size_t) rest;
307         }
308
309         n = c->recv(c, rb->buf->last, size);
310
311         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, c->log, 0,
312             "http client request body recv %z", n);
313
314         if (n == NGX_AGAIN) {
315             break;
316         }
317
318         if (n == 0) {
319             ngx_log_error(NGX_LOG_INFO, c->log, 0,
320                 "client prematurely closed connection");
321         }
322
323         if (n == 0 || n == NGX_ERROR) {
324             c->error = 1;
325             return NGX_HTTP_BAD_REQUEST;
326         }
327
328         rb->buf->last += n;
329         r->request_length += n;
330
331         if (n == rest) {
332             /* pass buffer to request body filter chain */
333
334             out.buf = rb->buf;
335             out.next = NULL;
336
337             rc = ngx_http_request_body_filter(r, &out);
338
339             if (rc != NGX_OK) {

```

```

340         return rc;
341     }
342 }
343
344     if (rb->rest == 0) {
345         break;
346     }
347
348     if (rb->buf->last < rb->buf->end) {
349         break;
350     }
351 }
352
353 ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, c->log, 0,
354     "http client request body rest %0", rb->rest);
355
356     if (rb->rest == 0) {
357         break;
358     }
359
360     if (!c->read->ready) {
361         clcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_core\_module);
362         ngx\_add\_timer(c->read, clcf->client_body_timeout);
363
364         if (ngx\_handle\_read\_event(c->read, 0) != NGX\_OK) {
365             return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
366         }
367
368         return NGX\_AGAIN;
369     }
370 }
371
372     if (c->read->timer_set) {
373         ngx\_del\_timer(c->read);
374     }
375
376     if (rb->temp_file || r->request_body_in_file_only) {
377
378         /* save the last part */
379
380         if (ngx\_http\_write\_request\_body(r) != NGX\_OK) {
381             return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
382         }
383
384         if (rb->temp_file->file.offset != 0) {
385
386             cl = ngx\_chain\_get\_free\_buf(r->pool, &rb->free);
387             if (cl == NULL) {
388                 return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
389             }
390
391             b = cl->buf;
392
393             ngx\_memzero(b, sizeof(ngx\_buf\_t));
394
395             b->in_file = 1;
396             b->file_last = rb->temp_file->file.offset;
397             b->file = &rb->temp_file->file;
398
399             rb->bufs = cl;
400
401         } else {
402             rb->bufs = NULL;
403         }
404     }
405
406     r->read_event_handler = ngx\_http\_block\_reading;
407
408     rb->post_handler(r);
409
410     return NGX\_OK;
411 }
412
413
414 static ngx\_int\_t
415 ngx\_http\_write\_request\_body(ngx\_http\_request\_t *r)

```

```

416 {
417     ssize_t             n;
418     ngx_chain_t        *cl;
419     ngx_temp_file_t    *tf;
420     ngx_http_request_body_t *rb;
421     ngx_http_core_loc_conf_t *clcf;
422
423     rb = r->request_body;
424
425     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
426                  "http write client request body, bufs %p", rb->bufs);
427
428     if (rb->temp_file == NULL) {
429         tf = ngx_palloc(r->pool, sizeof(ngx_temp_file_t));
430         if (tf == NULL) {
431             return NGX_ERROR;
432         }
433
434         clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
435
436         tf->file.fd = NGX_INVALID_FILE;
437         tf->file.log = r->connection->log;
438         tf->path = clcf->client_body_temp_path;
439         tf->pool = r->pool;
440         tf->warn = "a client request body is buffered to a temporary file";
441         tf->log_level = r->request_body_file_log_level;
442         tf->persistent = r->request_body_in_persistent_file;
443         tf->clean = r->request_body_in_clean_file;
444
445         if (r->request_body_file_group_access) {
446             tf->access = 0660;
447         }
448
449         rb->temp_file = tf;
450
451         if (rb->bufs == NULL) {
452             /* empty body with r->request_body_in_file_only */
453
454             if (ngx_create_temp_file(&tf->file, tf->path, tf->pool,
455                                    tf->persistent, tf->clean, tf->access)
456                 != NGX_OK)
457             {
458                 return NGX_ERROR;
459             }
460
461             return NGX_OK;
462         }
463     }
464
465     if (rb->bufs == NULL) {
466         return NGX_OK;
467     }
468
469     n = ngx_write_chain_to_temp_file(rb->temp_file, rb->bufs);
470
471     /* TODO: n == 0 or not complete and level event */
472
473     if (n == NGX_ERROR) {
474         return NGX_ERROR;
475     }
476
477     rb->temp_file->offset += n;
478
479     /* mark all buffers as written */
480
481     for (cl = rb->bufs; cl; cl = cl->next) {
482         cl->buf->pos = cl->buf->last;
483     }
484
485     rb->bufs = NULL;
486
487     return NGX_OK;
488 }
489
490
491 ngx_int_t

```

```

492 ngx_http_discard_request_body(ngx_http_request_t *r)
493 {
494     ssize_t      size;
495     ngx_int_t    rc;
496     ngx_event_t  *rev;
497
498     #if (NGX_HTTP_SPDY)
499     if (r->spdy_stream && r == r->main) {
500         r->spdy_stream->skip_data = NGX_SPDY_DATA_DISCARD;
501         return NGX_OK;
502     }
503     #endif
504
505     if (r != r->main || r->discard_body || r->request_body) {
506         return NGX_OK;
507     }
508
509     if (ngx_http_test_expect(r) != NGX_OK) {
510         return NGX_HTTP_INTERNAL_SERVER_ERROR;
511     }
512
513     rev = r->connection->read;
514
515     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, rev->log, 0, "http set discard body");
516
517     if (rev->timer_set) {
518         ngx_del_timer(rev);
519     }
520
521     if (r->headers_in.content_length_n <= 0 && !r->headers_in.chunked) {
522         return NGX_OK;
523     }
524
525     size = r->header_in->last - r->header_in->pos;
526
527     if (size || r->headers_in.chunked) {
528         rc = ngx_http_discard_request_body_filter(r, r->header_in);
529
530         if (rc != NGX_OK) {
531             return rc;
532         }
533
534         if (r->headers_in.content_length_n == 0) {
535             return NGX_OK;
536         }
537     }
538
539     rc = ngx_http_read_discarded_request_body(r);
540
541     if (rc == NGX_OK) {
542         r->lingering_close = 0;
543         return NGX_OK;
544     }
545
546     if (rc >= NGX_HTTP_SPECIAL_RESPONSE) {
547         return rc;
548     }
549
550     /* rc == NGX_AGAIN */
551
552     r->read_event_handler = ngx_http_discarded_request_body_handler;
553
554     if (ngx_handle_read_event(rev, 0) != NGX_OK) {
555         return NGX_HTTP_INTERNAL_SERVER_ERROR;
556     }
557
558     r->count++;
559     r->discard_body = 1;
560
561     return NGX_OK;
562 }
563
564
565 void
566 ngx_http_discarded_request_body_handler(ngx_http_request_t *r)
567 {

```



```

568     ngx_int_t      rc;
569     ngx_msec_t     timer;
570     ngx_event_t    *rev;
571     ngx_connection_t *c;
572     ngx_http_core_loc_conf_t *clcf;
573
574     c = r->connection;
575     rev = c->read;
576
577     if (rev->timedout) {
578         c->timedout = 1;
579         c->error = 1;
580         ngx_http_finalize_request(r, NGX_ERROR);
581         return;
582     }
583
584     if (r->lingering_time) {
585         timer = (ngx_msec_t) r->lingering_time - (ngx_msec_t) ngx_time();
586
587         if ((ngx_msec_int_t) timer <= 0) {
588             r->discard_body = 0;
589             r->lingering_close = 0;
590             ngx_http_finalize_request(r, NGX_ERROR);
591             return;
592         }
593
594     } else {
595         timer = 0;
596     }
597
598     rc = ngx_http_read_discarded_request_body(r);
599
600     if (rc == NGX_OK) {
601         r->discard_body = 0;
602         r->lingering_close = 0;
603         ngx_http_finalize_request(r, NGX_DONE);
604         return;
605     }
606
607     if (rc >= NGX_HTTP_SPECIAL_RESPONSE) {
608         c->error = 1;
609         ngx_http_finalize_request(r, NGX_ERROR);
610         return;
611     }
612
613     /* rc == NGX_AGAIN */
614
615     if (ngx_handle_read_event(rev, 0) != NGX_OK) {
616         c->error = 1;
617         ngx_http_finalize_request(r, NGX_ERROR);
618         return;
619     }
620
621     if (timer) {
622
623         clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
624
625         timer *= 1000;
626
627         if (timer > clcf->lingering_timeout) {
628             timer = clcf->lingering_timeout;
629         }
630
631         ngx_add_timer(rev, timer);
632     }
633 }
634
635
636 static ngx_int_t
637 ngx_http_read_discarded_request_body(ngx_http_request_t *r)
638 {
639     size_t      size;
640     ssize_t     n;
641     ngx_int_t   rc;
642     ngx_buf_t   b;
643     u_char      buffer[NGX_HTTP_DISCARD_BUFFER_SIZE];

```

```

644 ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
645     "http read discarded body");
646
647
648 ngx\_memzero(&b, sizeof(ngx\_buf\_t));
649
650 b.temporary = 1;
651
652 for ( ;; ) {
653     if (r->headers_in.content_length_n == 0) {
654         r->read_event_handler = ngx\_http\_block\_reading;
655         return NGX\_OK;
656     }
657
658     if (!r->connection->read->ready) {
659         return NGX\_AGAIN;
660     }
661
662     size = (size_t) ngx\_min(r->headers_in.content_length_n,
663         NGX\_HTTP\_DISCARD\_BUFFER\_SIZE);
664
665     n = r->connection->recv(r->connection, buffer, size);
666
667     if (n == NGX\_ERROR) {
668         r->connection->error = 1;
669         return NGX\_OK;
670     }
671
672     if (n == NGX\_AGAIN) {
673         return NGX\_AGAIN;
674     }
675
676     if (n == 0) {
677         return NGX\_OK;
678     }
679
680     b.pos = buffer;
681     b.last = buffer + n;
682
683     rc = ngx\_http\_discard\_request\_body\_filter(r, &b);
684
685     if (rc != NGX\_OK) {
686         return rc;
687     }
688 }
689 }
690
691
692 static ngx\_int\_t
693 ngx\_http\_discard\_request\_body\_filter(ngx\_http\_request\_t *r, ngx\_buf\_t *b)
694 {
695     size_t          size;
696     ngx\_int\_t      rc;
697     ngx\_http\_request\_body\_t *rb;
698
699     if (r->headers_in.chunked) {
700
701         rb = r->request_body;
702
703         if (rb == NULL) {
704
705             rb = ngx\_palloc(r->pool, sizeof(ngx\_http\_request\_body\_t));
706             if (rb == NULL) {
707                 return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
708             }
709
710             rb->chunked = ngx\_palloc(r->pool, sizeof(ngx\_http\_chunked\_t));
711             if (rb->chunked == NULL) {
712                 return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
713             }
714
715             r->request_body = rb;
716         }
717
718     for ( ;; ) {
719

```

```

720     rc = ngx_http_parse_chunked(r, b, rb->chunked);
721
722     if (rc == NGX_OK) {
723         /* a chunk has been parsed successfully */
724
725         size = b->last - b->pos;
726
727         if ((off_t) size > rb->chunked->size) {
728             b->pos += (size_t) rb->chunked->size;
729             rb->chunked->size = 0;
730
731         } else {
732             rb->chunked->size -= size;
733             b->pos = b->last;
734         }
735
736         continue;
737     }
738
739     if (rc == NGX_DONE) {
740         /* a whole response has been parsed successfully */
741
742         r->headers_in.content_length_n = 0;
743         break;
744     }
745
746     if (rc == NGX_AGAIN) {
747         /* set amount of data we want to see next time */
748
749         r->headers_in.content_length_n = rb->chunked->length;
750         break;
751     }
752
753     /* invalid */
754
755     ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
756                 "client sent invalid chunked body");
757
758     return NGX_HTTP_BAD_REQUEST;
759 }
760
761 } else {
762     size = b->last - b->pos;
763
764     if ((off_t) size > r->headers_in.content_length_n) {
765         b->pos += (size_t) r->headers_in.content_length_n;
766         r->headers_in.content_length_n = 0;
767
768     } else {
769         b->pos = b->last;
770         r->headers_in.content_length_n -= size;
771     }
772 }
773
774 return NGX_OK;
775 }
776
777 }
778
779
780
781 static ngx_int_t
782 ngx_http_test_expect(ngx_http_request_t *r)
783 {
784     ngx_int_t    n;
785     ngx_str_t    *expect;
786
787     if (r->expect_tested
788         || r->headers_in.expect == NULL
789         || r->http_version < NGX_HTTP_VERSION_11)
790     {
791         return NGX_OK;
792     }
793
794     r->expect_tested = 1;
795

```

```

796 expect = &r->headers_in.expect->value;
797
798 if (expect->len != sizeof("100-continue") - 1
799     || ngx_strncasecmp(expect->data, (u_char *) "100-continue",
800                        sizeof("100-continue") - 1)
801     != 0)
802 {
803     return NGX_OK;
804 }
805
806 ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
807               "send 100 Continue");
808
809 n = r->connection->send(r->connection,
810                       (u_char *) "HTTP/1.1 100 Continue" CRLF CRLF,
811                       sizeof("HTTP/1.1 100 Continue" CRLF CRLF) - 1);
812
813 if (n == sizeof("HTTP/1.1 100 Continue" CRLF CRLF) - 1) {
814     return NGX_OK;
815 }
816
817 /* we assume that such small packet should be send successfully */
818
819 return NGX_ERROR;
820 }
821
822
823 static ngx_int_t
824 ngx_http_request_body_filter(ngx_http_request_t *r, ngx_chain_t *in)
825 {
826     if (r->headers_in.chunked) {
827         return ngx_http_request_body_chunked_filter(r, in);
828     }
829     else {
830         return ngx_http_request_body_length_filter(r, in);
831     }
832 }
833
834
835 static ngx_int_t
836 ngx_http_request_body_length_filter(ngx_http_request_t *r, ngx_chain_t *in)
837 {
838     size_t          size;
839     ngx_int_t       rc;
840     ngx_buf_t       *b;
841     ngx_chain_t     *cl, *tl, *out, **ll;
842     ngx_http_request_body_t *rb;
843
844     rb = r->request_body;
845
846     if (rb->rest == -1) {
847         ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
848                       "http request body content length filter");
849
850         rb->rest = r->headers_in.content_length_n;
851     }
852
853     out = NULL;
854     ll = &out;
855
856     for (cl = in; cl; cl = cl->next) {
857
858         if (rb->rest == 0) {
859             break;
860         }
861
862         tl = ngx_chain_get_free_buf(r->pool, &rb->free);
863         if (tl == NULL) {
864             return NGX_HTTP_INTERNAL_SERVER_ERROR;
865         }
866
867         b = tl->buf;
868
869         ngx_memzero(b, sizeof(ngx_buf_t));
870
871         b->temporary = 1;

```

```

872     b->tag = (ngx\_buf\_tag\_t) &ngx\_http\_read\_client\_request\_body;
873     b->start = cl->buf->pos;
874     b->pos = cl->buf->pos;
875     b->last = cl->buf->last;
876     b->end = cl->buf->end;
877
878     size = cl->buf->last - cl->buf->pos;
879
880     if ((off\_t) size < rb->rest) {
881         cl->buf->pos = cl->buf->last;
882         rb->rest -= size;
883
884     } else {
885         cl->buf->pos += (size\_t) rb->rest;
886         rb->rest = 0;
887         b->last = cl->buf->pos;
888         b->last_buf = 1;
889     }
890
891     *ll = t1;
892     ll = &t1->next;
893 }
894
895 rc = ngx\_http\_request\_body\_save\_filter(r, out);
896
897 ngx\_chain\_update\_chains(r->pool, &rb->free, &rb->busy, &out,
898     (ngx\_buf\_tag\_t) &ngx\_http\_read\_client\_request\_body);
899
900 return rc;
901 }
902
903
904 static ngx\_int\_t
905 ngx\_http\_request\_body\_chunked\_filter(ngx\_http\_request\_t *r, ngx\_chain\_t *in)
906 {
907     size\_t                size;
908     ngx\_int\_t            rc;
909     ngx\_buf\_t            *b;
910     ngx\_chain\_t          *cl, *out, *t1, **ll;
911     ngx\_http\_request\_body\_t *rb;
912     ngx\_http\_core\_loc\_conf\_t *clcf;
913
914     rb = r->request_body;
915
916     if (rb->rest == -1) {
917
918         ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
919             "http request body chunked filter");
920
921         rb->chunked = ngx\_palloc(r->pool, sizeof(ngx\_http\_chunked\_t));
922         if (rb->chunked == NULL) {
923             return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
924         }
925
926         r->headers_in.content_length_n = 0;
927         rb->rest = 3;
928     }
929
930     out = NULL;
931     ll = &out;
932
933     for (cl = in; cl; cl = cl->next) {
934
935         for ( ;; ) {
936
937             ngx\_log\_debug7(NGX\_LOG\_DEBUG\_EVENT, r->connection->log, 0,
938                 "http body chunked buf "
939                 "t:%d f:%d %p, pos %p, size: %z file: %0, size: %z",
940                 cl->buf->temporary, cl->buf->in_file,
941                 cl->buf->start, cl->buf->pos,
942                 cl->buf->last - cl->buf->pos,
943                 cl->buf->file_pos,
944                 cl->buf->file_last - cl->buf->file_pos);
945
946             rc = ngx\_http\_parse\_chunked(r, cl->buf, rb->chunked);
947

```

```

948 if (rc == NGX_OK) {
949
950     /* a chunk has been parsed successfully */
951
952     clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
953
954     if (clcf->client_max_body_size
955         && clcf->client_max_body_size
956         - r->headers_in.content_length_n < rb->chunked->size)
957     {
958         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
959             "client intended to send too large chunked "
960             "body: %0+%0 bytes",
961             r->headers_in.content_length_n,
962             rb->chunked->size);
963
964         r->lingering_close = 1;
965
966         return NGX_HTTP_REQUEST_ENTITY_TOO_LARGE;
967     }
968
969     tl = ngx_chain_get_free_buf(r->pool, &rb->free);
970     if (tl == NULL) {
971         return NGX_HTTP_INTERNAL_SERVER_ERROR;
972     }
973
974     b = tl->buf;
975
976     ngx_memzero(b, sizeof(ngx_buf_t));
977
978     b->temporary = 1;
979     b->tag = (ngx_buf_tag_t) &ngx_http_read_client_request_body;
980     b->start = cl->buf->pos;
981     b->pos = cl->buf->pos;
982     b->last = cl->buf->last;
983     b->end = cl->buf->end;
984
985     *ll = tl;
986     ll = &tl->next;
987
988     size = cl->buf->last - cl->buf->pos;
989
990     if ((off_t) size > rb->chunked->size) {
991         cl->buf->pos += (size_t) rb->chunked->size;
992         r->headers_in.content_length_n += rb->chunked->size;
993         rb->chunked->size = 0;
994     } else {
995         rb->chunked->size -= size;
996         r->headers_in.content_length_n += size;
997         cl->buf->pos = cl->buf->last;
998     }
999
1000     b->last = cl->buf->pos;
1001
1002     continue;
1003 }
1004
1005 if (rc == NGX_DONE) {
1006
1007     /* a whole response has been parsed successfully */
1008
1009     rb->rest = 0;
1010
1011     tl = ngx_chain_get_free_buf(r->pool, &rb->free);
1012     if (tl == NULL) {
1013         return NGX_HTTP_INTERNAL_SERVER_ERROR;
1014     }
1015
1016     b = tl->buf;
1017
1018     ngx_memzero(b, sizeof(ngx_buf_t));
1019
1020     b->last_buf = 1;
1021
1022     *ll = tl;
1023

```

```

1024         ll = &t1->next;
1025
1026         break;
1027     }
1028
1029     if (rc == NGX_AGAIN) {
1030
1031         /* set rb->rest, amount of data we want to see next time */
1032
1033         rb->rest = rb->chunked->length;
1034
1035         break;
1036     }
1037
1038     /* invalid */
1039
1040     ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1041                 "client sent invalid chunked body");
1042
1043     return NGX_HTTP_BAD_REQUEST;
1044 }
1045 }
1046
1047 rc = ngx_http_request_body_save_filter(r, out);
1048
1049 ngx_chain_update_chains(r->pool, &rb->free, &rb->busy, &out,
1050                          (ngx_buf_tag_t) ngx_http_read_client_request_body);
1051
1052 return rc;
1053 }
1054
1055
1056 static ngx_int_t
1057 ngx_http_request_body_save_filter(ngx_http_request_t *r, ngx_chain_t *in)
1058 {
1059     #if (NGX_DEBUG)
1060     ngx_chain_t          *cl;
1061     #endif
1062     ngx_http_request_body_t *rb;
1063
1064     rb = r->request_body;
1065
1066     #if (NGX_DEBUG)
1067
1068     for (cl = rb->bufs; cl; cl = cl->next) {
1069         ngx_log_debug7(NGX_LOG_DEBUG_EVENT, r->connection->log, 0,
1070                       "http body old buf t:%d f:%d %p, pos %p, size: %z "
1071                       "file: %0, size: %z",
1072                       cl->buf->temporary, cl->buf->in_file,
1073                       cl->buf->start, cl->buf->pos,
1074                       cl->buf->last - cl->buf->pos,
1075                       cl->buf->file_pos,
1076                       cl->buf->file_last - cl->buf->file_pos);
1077     }
1078
1079     for (cl = in; cl; cl = cl->next) {
1080         ngx_log_debug7(NGX_LOG_DEBUG_EVENT, r->connection->log, 0,
1081                       "http body new buf t:%d f:%d %p, pos %p, size: %z "
1082                       "file: %0, size: %z",
1083                       cl->buf->temporary, cl->buf->in_file,
1084                       cl->buf->start, cl->buf->pos,
1085                       cl->buf->last - cl->buf->pos,
1086                       cl->buf->file_pos,
1087                       cl->buf->file_last - cl->buf->file_pos);
1088     }
1089
1090     #endif
1091
1092     /* TODO: coalesce neighbouring buffers */
1093
1094     if (ngx_chain_add_copy(r->pool, &rb->bufs, in) != NGX_OK) {
1095         return NGX_HTTP_INTERNAL_SERVER_ERROR;
1096     }
1097
1098     return NGX_OK;
1099 }

```

[One Level Up](#)

[Top Level](#)



# src/core/nginx\_open\_file\_cache.h - nginx-1.7.10

## Data types defined

- [ngx\\_cached\\_open\\_file\\_s](#)
- [ngx\\_cached\\_open\\_file\\_t](#)
- [ngx\\_open\\_file\\_cache\\_cleanup\\_t](#)
- [ngx\\_open\\_file\\_cache\\_event\\_t](#)
- [ngx\\_open\\_file\\_cache\\_t](#)
- [ngx\\_open\\_file\\_info\\_t](#)

## Macros defined

- [NGX\\_OPEN\\_FILE\\_DIRECTIO\\_OFF](#)
- [\\_NGX\\_OPEN\\_FILE\\_CACHE\\_H\\_INCLUDED](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10
11
12 #ifndef _NGX_OPEN_FILE_CACHE_H_INCLUDED_
13 #define _NGX_OPEN_FILE_CACHE_H_INCLUDED_
14
15
16 #define NGX_OPEN_FILE_DIRECTIO_OFF  NGX_MAX_OFF_T_VALUE
17
18
19 typedef struct {
20     ngx_fd_t          fd;
21     ngx_file_uniq_t  uniq;
22     time_t           mtime;
23     off_t            size;
24     off_t            fs_size;
25     off_t            directio;
26     size_t           read_ahead;
27
28     ngx_err_t        err;
29     char             *failed;
30
31     time_t           valid;
32
33     ngx_uint_t       min_uses;
34
35     #if (NGX_HAVE_OPENAT)
36     size_t           disable_symlinks_from;
37     unsigned         disable_symlinks:2;
38 #endif
39
40     unsigned         test_dir:1;
41     unsigned         test_only:1;
42     unsigned         log:1;
43     unsigned         errors:1;
```

```

44     unsigned                events:1;
45
46     unsigned                is_dir:1;
47     unsigned                is_file:1;
48     unsigned                is_link:1;
49     unsigned                is_exec:1;
50     unsigned                is_directio:1;
51 } ngx_open_file_info_t;
52
53
54 typedef struct ngx_cached_open_file_s ngx_cached_open_file_t;
55
56 struct ngx_cached_open_file_s {
57     ngx_rbtrees_node_t      node;
58     ngx_queue_t           queue;
59
60     u_char                  *name;
61     time_t                  created;
62     time_t                  accessed;
63
64     ngx_fd_t              fd;
65     ngx_file_uniq_t      uniq;
66     time_t                  mtime;
67     off_t                   size;
68     ngx_err_t           err;
69
70     uint32_t                uses;
71
72     #if (NGX_HAVE_OPENAT)
73     size_t                  disable_symlinks_from;
74     unsigned                disable_symlinks:2;
75     #endif
76
77     unsigned                count:24;
78     unsigned                close:1;
79     unsigned                use_event:1;
80
81     unsigned                is_dir:1;
82     unsigned                is_file:1;
83     unsigned                is_link:1;
84     unsigned                is_exec:1;
85     unsigned                is_directio:1;
86
87     ngx_event_t         *event;
88 };
89
90
91 typedef struct {
92     ngx_rbtrees_t        rbtree;
93     ngx_rbtrees_node_t  sentinel;
94     ngx_queue_t        expire_queue;
95
96     ngx_uint_t          current;
97     ngx_uint_t          max;
98     time_t               inactive;
99 } ngx_open_file_cache_t;
100
101
102 typedef struct {
103     ngx_open_file_cache_t *cache;
104     ngx_cached_open_file_t *file;
105     ngx_uint_t          min_uses;
106     ngx_log_t         *log;
107 } ngx_open_file_cache_cleanup_t;
108
109
110 typedef struct {
111
112     /* ngx_connection_t stub to allow use c->fd as event ident */
113     void                *data;
114     ngx_event_t      *read;
115     ngx_event_t      *write;
116     ngx_fd_t         fd;
117
118     ngx_cached_open_file_t *file;
119     ngx_open_file_cache_t *cache;

```

```
120 } ngx_open_file_cache_event_t;
121
122
123 ngx_open_file_cache_t *ngx_open_file_cache_init(ngx_pool_t *pool,
124     ngx_uint_t max, time_t inactive);
125 ngx_int_t ngx_open_cached_file(ngx_open_file_cache_t *cache, ngx_str_t *name,
126     ngx_open_file_info_t *of, ngx_pool_t *pool);
127
128
129 #endif /* _NGX_OPEN_FILE_CACHE_H_INCLUDED */
```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_open\_file\_cache.c - nginx-1.7.10

### Functions defined

- [ngx\\_close\\_cached\\_file](#)
- [ngx\\_expire\\_old\\_cached\\_files](#)
- [ngx\\_file\\_info\\_wrapper](#)
- [ngx\\_file\\_o\\_path\\_info](#)
- [ngx\\_open\\_and\\_stat\\_file](#)
- [ngx\\_open\\_cached\\_file](#)
- [ngx\\_open\\_file\\_add\\_event](#)
- [ngx\\_open\\_file\\_cache\\_cleanup](#)
- [ngx\\_open\\_file\\_cache\\_init](#)
- [ngx\\_open\\_file\\_cache\\_rbtrees\\_insert\\_value](#)
- [ngx\\_open\\_file\\_cache\\_remove](#)
- [ngx\\_open\\_file\\_cleanup](#)
- [ngx\\_open\\_file\\_del\\_event](#)
- [ngx\\_open\\_file\\_lookup](#)
- [ngx\\_open\\_file\\_wrapper](#)
- [ngx\\_openat\\_file\\_owner](#)

### Macros defined

- [NGX\\_MIN\\_READ\\_AHEAD](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 /*
14  * open file cache caches
15  *   open file handles with stat() info;
16  *   directories stat() info;
17  *   files and directories errors: not found, access denied, etc.
18  */
19
20
21 #define NGX_MIN_READ_AHEAD (128 * 1024)
22
23
```

```

24 static void ngx_open_file_cache_cleanup(void *data);
25 #if (NGX_HAVE_OPENAT)
26 static ngx_fd_t ngx_openat_file_owner(ngx_fd_t at_fd, const u_char *name,
27     ngx_int_t mode, ngx_int_t create, ngx_int_t access, ngx_log_t *log);
28 #if (NGX_HAVE_O_PATH)
29 static ngx_int_t ngx_file_o_path_info(ngx_fd_t fd, ngx_file_info_t *fi,
30     ngx_log_t *log);
31 #endif
32 #endif
33 static ngx_fd_t ngx_open_file_wrapper(ngx_str_t *name,
34     ngx_open_file_info_t *of, ngx_int_t mode, ngx_int_t create,
35     ngx_int_t access, ngx_log_t *log);
36 static ngx_int_t ngx_file_info_wrapper(ngx_str_t *name,
37     ngx_open_file_info_t *of, ngx_file_info_t *fi, ngx_log_t *log);
38 static ngx_int_t ngx_open_and_stat_file(ngx_str_t *name,
39     ngx_open_file_info_t *of, ngx_log_t *log);
40 static void ngx_open_file_add_event(ngx_open_file_cache_t *cache,
41     ngx_cached_open_file_t *file, ngx_open_file_info_t *of, ngx_log_t *log);
42 static void ngx_open_file_cleanup(void *data);
43 static void ngx_close_cached_file(ngx_open_file_cache_t *cache,
44     ngx_cached_open_file_t *file, ngx_uint_t min_uses, ngx_log_t *log);
45 static void ngx_open_file_del_event(ngx_cached_open_file_t *file);
46 static void ngx_expire_old_cached_files(ngx_open_file_cache_t *cache,
47     ngx_uint_t n, ngx_log_t *log);
48 static void ngx_open_file_cache_rbtrees_insert_value(ngx_rbtrees_node_t *temp,
49     ngx_rbtrees_node_t *node, ngx_rbtrees_node_t *sentinel);
50 static ngx_cached_open_file_t *
51     ngx_open_file_lookup(ngx_open_file_cache_t *cache, ngx_str_t *name,
52     uint32_t hash);
53 static void ngx_open_file_cache_remove(ngx_event_t *ev);
54
55
56 ngx_open_file_cache_t *
57 ngx_open_file_cache_init(ngx_pool_t *pool, ngx_uint_t max, time_t inactive)
58 {
59     ngx_pool_cleanup_t *cfn;
60     ngx_open_file_cache_t *cache;
61
62     cache = ngx_palloc(pool, sizeof(ngx_open_file_cache_t));
63     if (cache == NULL) {
64         return NULL;
65     }
66
67     ngx_rbtrees_init(&cache->rbtrees, &cache->sentinel,
68         ngx_open_file_cache_rbtrees_insert_value);
69
70     ngx_queue_init(&cache->expire_queue);
71
72     cache->current = 0;
73     cache->max = max;
74     cache->inactive = inactive;
75
76     cfn = ngx_pool_cleanup_add(pool, 0);
77     if (cfn == NULL) {
78         return NULL;
79     }
80
81     cfn->handler = ngx_open_file_cache_cleanup;
82     cfn->data = cache;
83
84     return cache;
85 }
86
87
88 static void
89 ngx_open_file_cache_cleanup(void *data)
90 {
91     ngx_open_file_cache_t *cache = data;
92
93     ngx_queue_t *q;
94     ngx_cached_open_file_t *file;
95
96     ngx_log_debug0(NGX_LOG_DEBUG_CORE, ngx_cycle->log, 0,
97         "open file cache cleanup");
98
99     for ( ;; ) {

```

```

100     if (ngx\_queue\_empty(&cache->expire_queue)) {
101         break;
102     }
103
104
105     q = ngx\_queue\_last(&cache->expire_queue);
106
107     file = ngx\_queue\_data(q, ngx\_cached\_open\_file\_t, queue);
108
109     ngx\_queue\_remove(q);
110
111     ngx\_rbtrees\_delete(&cache->rbtree, &file->node);
112
113     cache->current--;
114
115     ngx\_log\_debug1(NGX_LOG_DEBUG_CORE, ngx\_cycle->log, 0,
116         "delete cached open file: %s", file->name);
117
118     if (!file->err && !file->is_dir) {
119         file->close = 1;
120         file->count = 0;
121         ngx\_close\_cached\_file(cache, file, 0, ngx\_cycle->log);
122
123     } else {
124         ngx\_free(file->name);
125         ngx\_free(file);
126     }
127 }
128
129 if (cache->current) {
130     ngx\_log\_error(NGX_LOG_ALERT, ngx\_cycle->log, 0,
131         "%ui items still leave in open file cache",
132         cache->current);
133 }
134
135 if (cache->rbtree.root != cache->rbtree.sentinel) {
136     ngx\_log\_error(NGX_LOG_ALERT, ngx\_cycle->log, 0,
137         "rbtree still is not empty in open file cache");
138 }
139 }
140 }
141
142
143 ngx\_int\_t
144 ngx\_open\_cached\_file(ngx\_open\_file\_cache\_t *cache, ngx\_str\_t *name,
145     ngx\_open\_file\_info\_t *of, ngx\_pool\_t *pool)
146 {
147     time_t                now;
148     uint32_t              hash;
149     ngx\_int\_t              rc;
150     ngx\_file\_info\_t        fi;
151     ngx\_pool\_cleanup\_t     *cln;
152     ngx\_cached\_open\_file\_t *file;
153     ngx\_pool\_cleanup\_file\_t *clnf;
154     ngx\_open\_file\_cache\_cleanup\_t *ofcln;
155
156     of->fd = NGX\_INVALID\_FILE;
157     of->err = 0;
158
159     if (cache == NULL) {
160
161         if (of->test_only) {
162
163             if (ngx\_file\_info\_wrapper(name, of, &fi, pool->log)
164                 == NGX\_FILE\_ERROR)
165             {
166                 return NGX\_ERROR;
167             }
168
169             of->uniq = ngx\_file\_uniq(&fi);
170             of->mtime = ngx\_file\_mtime(&fi);
171             of->size = ngx\_file\_size(&fi);
172             of->fs_size = ngx\_file\_fs\_size(&fi);
173             of->is_dir = ngx\_is\_dir(&fi);
174             of->is_file = ngx\_is\_file(&fi);
175             of->is_link = ngx\_is\_link(&fi);

```

```

176         of->is_exec = ngx_is_exec(&fi);
177
178         return NGX_OK;
179     }
180
181     cln = ngx_pool_cleanup_add(pool, sizeof(ngx_pool_cleanup_file_t));
182     if (cln == NULL) {
183         return NGX_ERROR;
184     }
185
186     rc = ngx_open_and_stat_file(name, of, pool->log);
187
188     if (rc == NGX_OK && !of->is_dir) {
189         cln->handler = ngx_pool_cleanup_file;
190         clnf = cln->data;
191
192         clnf->fd = of->fd;
193         clnf->name = name->data;
194         clnf->log = pool->log;
195     }
196
197     return rc;
198 }
199
200 cln = ngx_pool_cleanup_add(pool, sizeof(ngx_open_file_cache_cleanup_t));
201 if (cln == NULL) {
202     return NGX_ERROR;
203 }
204
205 now = ngx_time();
206
207 hash = ngx_crc32_long(name->data, name->len);
208
209 file = ngx_open_file_lookup(cache, name, hash);
210
211 if (file) {
212
213     file->uses++;
214
215     ngx_queue_remove(&file->queue);
216
217     if (file->fd == NGX_INVALID_FILE && file->err == 0 && !file->is_dir) {
218
219         /* file was not used often enough to keep open */
220
221         rc = ngx_open_and_stat_file(name, of, pool->log);
222
223         if (rc != NGX_OK && (of->err == 0 || !of->errors)) {
224             goto failed;
225         }
226
227         goto add_event;
228     }
229
230     if (file->use_event
231         || (file->event == NULL
232             && (of->uniq == 0 || of->uniq == file->uniq)
233             && now - file->created < of->valid
234 #if (NGX_HAVE_OPENAT)
235             && of->disable_symlinks == file->disable_symlinks
236             && of->disable_symlinks_from == file->disable_symlinks_from
237 #endif
238         ))
239     {
240         if (file->err == 0) {
241
242             of->fd = file->fd;
243             of->uniq = file->uniq;
244             of->mtime = file->mtime;
245             of->size = file->size;
246
247             of->is_dir = file->is_dir;
248             of->is_file = file->is_file;
249             of->is_link = file->is_link;
250             of->is_exec = file->is_exec;
251             of->is_directio = file->is_directio;

```

```

252         if (!file->is_dir) {
253             file->count++;
254             ngx\_open\_file\_add\_event(cache, file, of, pool->log);
255         }
256     }
257 } else {
258     of->err = file->err;
259 #if (NGX_HAVE_OPENAT)
260     of->failed = file->disable_symlinks ? ngx\_openat\_file\_n
261                                         : ngx\_open\_file\_n;
262 #else
263     of->failed = ngx\_open\_file\_n;
264 #endif
265 }
266
267     goto found;
268 }
269
270 ngx\_log\_debug4(NGX_LOG_DEBUG_CORE, pool->log, 0,
271               "retest open file: %s, fd:%d, c:%d, e:%d",
272               file->name, file->fd, file->count, file->err);
273
274 if (file->is_dir) {
275     /*
276     * chances that directory became file are very small
277     * so test_dir flag allows to use a single syscall
278     * in ngx\_file\_info\(\) instead of three syscalls
279     */
280
281     of->test_dir = 1;
282 }
283
284 of->fd = file->fd;
285 of->uniq = file->uniq;
286
287 rc = ngx\_open\_and\_stat\_file(name, of, pool->log);
288
289 if (rc != NGX_OK && (of->err == 0 || !of->errors)) {
290     goto failed;
291 }
292
293 if (of->is_dir) {
294     if (file->is_dir || file->err) {
295         goto update;
296     }
297
298     /* file became directory */
299 } else if (of->err == 0) { /* file */
300
301     if (file->is_dir || file->err) {
302         goto add_event;
303     }
304
305     if (of->uniq == file->uniq) {
306         if (file->event) {
307             file->use_event = 1;
308         }
309
310         of->is_directio = file->is_directio;
311
312         goto update;
313     }
314
315     /* file was changed */
316 } else { /* error to cache */
317
318     if (file->err || file->is_dir) {
319         goto update;
320     }
321 }
322
323
324
325
326
327

```



```

328     /* file was removed, etc. */
329 }
330
331 if (file->count == 0) {
332     ngx_open_file_del_event(file);
333
334     if (ngx_close_file(file->fd) == NGX_FILE_ERROR) {
335         ngx_log_error(NGX_LOG_ALERT, pool->log, ngx_errno,
336             ngx_close_file_n "\'%\%' failed", name);
337     }
338 }
339
340 goto add_event;
341 }
342
343 ngx_rbtrees_delete(&cache->rbtree, &file->node);
344
345 cache->current--;
346
347 file->close = 1;
348
349 goto create;
350 }
351
352 /* not found */
353
354 rc = ngx_open_and_stat_file(name, of, pool->log);
355
356 if (rc != NGX_OK && (of->err == 0 || !of->errors)) {
357     goto failed;
358 }
359
360 create:
361
362 if (cache->current >= cache->max) {
363     ngx_expire_old_cached_files(cache, 0, pool->log);
364 }
365
366 file = ngx_alloc(sizeof(ngx_cached_open_file_t), pool->log);
367
368 if (file == NULL) {
369     goto failed;
370 }
371
372 file->name = ngx_alloc(name->len + 1, pool->log);
373
374 if (file->name == NULL) {
375     ngx_free(file);
376     file = NULL;
377     goto failed;
378 }
379
380 ngx_cpysrtn(file->name, name->data, name->len + 1);
381
382 file->node.key = hash;
383
384 ngx_rbtrees_insert(&cache->rbtree, &file->node);
385
386 cache->current++;
387
388 file->uses = 1;
389 file->count = 0;
390 file->use_event = 0;
391 file->event = NULL;
392
393 add_event:
394
395 ngx_open_file_add_event(cache, file, of, pool->log);
396
397 update:
398
399 file->fd = of->fd;
400 file->err = of->err;
401 #if (NGX_HAVE_OPENAT)
402 file->disable_symlinks = of->disable_symlinks;
403 file->disable_symlinks_from = of->disable_symlinks_from;

```

```

404 #endif
405
406     if (of->err == 0) {
407         file->uniq = of->uniq;
408         file->mtime = of->mtime;
409         file->size = of->size;
410
411         file->close = 0;
412
413         file->is_dir = of->is_dir;
414         file->is_file = of->is_file;
415         file->is_link = of->is_link;
416         file->is_exec = of->is_exec;
417         file->is_directio = of->is_directio;
418
419         if (!of->is_dir) {
420             file->count++;
421         }
422     }
423
424     file->created = now;
425
426 found:
427
428     file->accessed = now;
429
430     ngx\_queue\_insert\_head(&cache->expire_queue, &file->queue);
431
432     ngx\_log\_debug5(NGX_LOG_DEBUG_CORE, pool->log, 0,
433         "cached open file: %s, fd:%d, c:%d, e:%d, u:%d",
434         file->name, file->fd, file->count, file->err, file->uses);
435
436     if (of->err == 0) {
437
438         if (!of->is_dir) {
439             cln->handler = ngx\_open\_file\_cleanup;
440             ofcln = cln->data;
441
442             ofcln->cache = cache;
443             ofcln->file = file;
444             ofcln->min_uses = of->min_uses;
445             ofcln->log = pool->log;
446         }
447
448         return NGX\_OK;
449     }
450
451     return NGX\_ERROR;
452
453 failed:
454
455     if (file) {
456         ngx\_rbtree\_delete(&cache->rbtree, &file->node);
457
458         cache->current--;
459
460         if (file->count == 0) {
461
462             if (file->fd != NGX\_INVALID\_FILE) {
463                 if (ngx\_close\_file(file->fd) == NGX\_FILE\_ERROR) {
464                     ngx\_log\_error(NGX_LOG_ALERT, pool->log, ngx\_errno,
465                         ngx\_close\_file\_n " \"%s\" failed",
466                         file->name);
467                 }
468             }
469
470             ngx\_free(file->name);
471             ngx\_free(file);
472
473         } else {
474             file->close = 1;
475         }
476     }
477
478     if (of->fd != NGX\_INVALID\_FILE) {
479         if (ngx\_close\_file(of->fd) == NGX\_FILE\_ERROR) {

```

```

480         ngx_log_error(NGX_LOG_ALERT, pool->log, ngx_errno,
481                     ngx_close_file_n "\'%V\' failed", name);
482     }
483 }
484
485 return NGX_ERROR;
486 }
487
488
489 #if (NGX_HAVE_OPENAT)
490
491 static ngx_fd_t
492 ngx_openat_file_owner(ngx_fd_t at_fd, const u_char *name,
493                      ngx_int_t mode, ngx_int_t create, ngx_int_t access, ngx_log_t *log)
494 {
495     ngx_fd_t      fd;
496     ngx_err_t     err;
497     ngx_file_info_t fi, atfi;
498
499     /*
500      * To allow symlinks with the same owner, use openat() (followed
501      * by fstat()) and fstatat(AT_SYMLINK_NOFOLLOW), and then compare
502      * uids between fstat() and fstatat().
503      *
504      * As there is a race between openat() and fstatat() we don't
505      * know if openat() in fact opened symlink or not. Therefore,
506      * we have to compare uids even if fstatat() reports the opened
507      * component isn't a symlink (as we don't know whether it was
508      * symlink during openat() or not).
509      */
510
511     fd = ngx_openat_file(at_fd, name, mode, create, access);
512
513     if (fd == NGX_INVALID_FILE) {
514         return NGX_INVALID_FILE;
515     }
516
517     if (ngx_file_at_info(at_fd, name, &atfi, AT_SYMLINK_NOFOLLOW)
518         == NGX_FILE_ERROR)
519     {
520         err = ngx_errno;
521         goto failed;
522     }
523
524     #if (NGX_HAVE_O_PATH)
525     if (ngx_file_o_path_info(fd, &fi, log) == NGX_ERROR) {
526         err = ngx_errno;
527         goto failed;
528     }
529     #else
530     if (ngx_fd_info(fd, &fi) == NGX_FILE_ERROR) {
531         err = ngx_errno;
532         goto failed;
533     }
534     #endif
535
536     if (fi.st_uid != atfi.st_uid) {
537         err = NGX_ELOOP;
538         goto failed;
539     }
540
541     return fd;
542
543 failed:
544
545     if (ngx_close_file(fd) == NGX_FILE_ERROR) {
546         ngx_log_error(NGX_LOG_ALERT, log, ngx_errno,
547                     ngx_close_file_n "\'%V\' failed", name);
548     }
549
550     ngx_set_errno(err);
551
552     return NGX_INVALID_FILE;
553 }
554
555

```

```

556 #if (NGX_HAVE_O_PATH)
557
558 static ngx_int_t
559 ngx_file_o_path_info(ngx_fd_t fd, ngx_file_info_t *fi, ngx_log_t *log)
560 {
561     static ngx_uint_t use_fstat = 1;
562
563     /*
564      * In Linux 2.6.39 the O_PATH flag was introduced that allows to obtain
565      * a descriptor without actually opening file or directory. It requires
566      * less permissions for path components, but till Linux 3.6 fstat() returns
567      * EBADF on such descriptors, and fstatat() with the AT_EMPTY_PATH flag
568      * should be used instead.
569      *
570      * Three scenarios are handled in this function:
571      *
572      * 1) The kernel is newer than 3.6 or fstat() with O_PATH support was
573      *    backported by vendor. Then fstat() is used.
574      *
575      * 2) The kernel is newer than 2.6.39 but older than 3.6. In this case
576      *    the first call of fstat() returns EBADF and we fallback to fstatat()
577      *    with AT_EMPTY_PATH which was introduced at the same time as O_PATH.
578      *
579      * 3) The kernel is older than 2.6.39 but nginx was build with O_PATH
580      *    support. Since descriptors are opened with O_PATH|O_RDONLY flags
581      *    and O_PATH is ignored by the kernel then the O_RDONLY flag is
582      *    actually used. In this case fstat() just works.
583      */
584
585     if (use_fstat) {
586         if (ngx_fd_info(fd, fi) != NGX_FILE_ERROR) {
587             return NGX_OK;
588         }
589
590         if (ngx_errno != NGX_EBADF) {
591             return NGX_ERROR;
592         }
593
594         ngx_log_error(NGX_LOG_NOTICE, log, 0,
595                     "fstat(O_PATH) failed with EBADF, "
596                     "switching to fstatat(AT_EMPTY_PATH)");
597
598         use_fstat = 0;
599     }
600
601     if (ngx_file_at_info(fd, "", fi, AT_EMPTY_PATH) != NGX_FILE_ERROR) {
602         return NGX_OK;
603     }
604
605     return NGX_ERROR;
606 }
607
608 #endif
609
610 #endif /* NGX_HAVE_OPENAT */
611
612
613 static ngx_fd_t
614 ngx_open_file_wrapper(ngx_str_t *name, ngx_open_file_info_t *of,
615                      ngx_int_t mode, ngx_int_t create, ngx_int_t access, ngx_log_t *log)
616 {
617     ngx_fd_t fd;
618
619     #if !(NGX_HAVE_OPENAT)
620
621     fd = ngx_open_file(name->data, mode, create, access);
622
623     if (fd == NGX_INVALID_FILE) {
624         of->err = ngx_errno;
625         of->failed = ngx_open_file_n;
626         return NGX_INVALID_FILE;
627     }
628
629     return fd;
630
631 #else

```

```

632 u_char          *p, *cp, *end;
633 ngx_fd_t       at_fd;
634 ngx_str_t      at_name;
635
636
637 if (of->disable_symlinks == NGX\_DISABLE\_SYMLINKS\_OFF) {
638     fd = ngx\_open\_file(name->data, mode, create, access);
639
640     if (fd == NGX\_INVALID\_FILE) {
641         of->err = ngx\_errno;
642         of->failed = ngx\_open\_file\_n;
643         return NGX\_INVALID\_FILE;
644     }
645
646     return fd;
647 }
648
649 p = name->data;
650 end = p + name->len;
651
652 at_name = *name;
653
654 if (of->disable_symlinks_from) {
655
656     cp = p + of->disable_symlinks_from;
657
658     *cp = '\0';
659
660     at_fd = ngx\_open\_file(p, NGX\_FILE\_SEARCH|NGX\_FILE\_NONBLOCK,
661                          NGX\_FILE\_OPEN, 0);
662
663     *cp = '/';
664
665     if (at_fd == NGX\_INVALID\_FILE) {
666         of->err = ngx\_errno;
667         of->failed = ngx\_open\_file\_n;
668         return NGX\_INVALID\_FILE;
669     }
670
671     at_name.len = of->disable_symlinks_from;
672     p = cp + 1;
673
674 } else if (*p == '/') {
675
676     at_fd = ngx\_open\_file("/",
677                          NGX\_FILE\_SEARCH|NGX\_FILE\_NONBLOCK,
678                          NGX\_FILE\_OPEN, 0);
679
680     if (at_fd == NGX\_INVALID\_FILE) {
681         of->err = ngx\_errno;
682         of->failed = ngx\_openat\_file\_n;
683         return NGX\_INVALID\_FILE;
684     }
685
686     at_name.len = 1;
687     p++;
688
689 } else {
690     at_fd = NGX\_AT\_FDCWD;
691 }
692
693 for ( ;; ) {
694     cp = ngx\_strlchr(p, end, '/');
695     if (cp == NULL) {
696         break;
697     }
698
699     if (cp == p) {
700         p++;
701         continue;
702     }
703
704     *cp = '\0';
705
706     if (of->disable_symlinks == NGX\_DISABLE\_SYMLINKS\_NOTOWNER) {
707         fd = ngx\_openat\_file\_owner(at_fd, p,

```

```

708                                     NGX_FILE_SEARCH|NGX_FILE_NONBLOCK,
709                                     NGX_FILE_OPEN, 0, log);
710
711     } else {
712         fd = ngx_openat_file(at_fd, p,
713                               NGX_FILE_SEARCH|NGX_FILE_NONBLOCK|NGX_FILE_NOFOLLOW,
714                               NGX_FILE_OPEN, 0);
715     }
716
717     *cp = '/';
718
719     if (fd == NGX_INVALID_FILE) {
720         of->err = ngx_errno;
721         of->failed = ngx_openat_file_n;
722         goto failed;
723     }
724
725     if (at_fd != NGX_AT_FDCWD && ngx_close_file(at_fd) == NGX_FILE_ERROR) {
726         ngx_log_error(NGX_LOG_ALERT, log, ngx_errno,
727                     ngx_close_file_n " \"%V\" failed", &at_name);
728     }
729
730     p = cp + 1;
731     at_fd = fd;
732     at_name.len = cp - at_name.data;
733 }
734
735 if (p == end) {
736
737     /*
738     * If pathname ends with a trailing slash, assume the last path
739     * component is a directory and reopen it with requested flags;
740     * if not, fail with ENOTDIR as per POSIX.
741     *
742     * We cannot rely on O_DIRECTORY in the loop above to check
743     * that the last path component is a directory because
744     * O_DIRECTORY doesn't work on FreeBSD 8. Fortunately, by
745     * reopening a directory, we don't depend on it at all.
746     */
747
748     fd = ngx_openat_file(at_fd, ".", mode, create, access);
749     goto done;
750 }
751
752 if (of->disable_symlinks == NGX_DISABLE_SYMLINKS_NOTOWNER
753     && !(create & (NGX_FILE_CREATE_OR_OPEN|NGX_FILE_TRUNCATE)))
754 {
755     fd = ngx_openat_file_owner(at_fd, p, mode, create, access, log);
756 }
757 else {
758     fd = ngx_openat_file(at_fd, p, mode|NGX_FILE_NOFOLLOW, create, access);
759 }
760
761 done:
762
763 if (fd == NGX_INVALID_FILE) {
764     of->err = ngx_errno;
765     of->failed = ngx_openat_file_n;
766 }
767
768 failed:
769
770 if (at_fd != NGX_AT_FDCWD && ngx_close_file(at_fd) == NGX_FILE_ERROR) {
771     ngx_log_error(NGX_LOG_ALERT, log, ngx_errno,
772                 ngx_close_file_n " \"%V\" failed", &at_name);
773 }
774
775 return fd;
776 #endif
777 }
778
779
780 static ngx_int_t
781 ngx_file_info_wrapper(ngx_str_t *name, ngx_open_file_info_t *of,
782                       ngx_file_info_t *fi, ngx_log_t *log)
783 {

```

```

784     ngx_int_t rc;
785
786     #if !(NGX_HAVE_OPENAT)
787
788     rc = ngx_file_info(name->data, fi);
789
790     if (rc == NGX_FILE_ERROR) {
791         of->err = ngx_errno;
792         of->failed = ngx_file_info_n;
793         return NGX_FILE_ERROR;
794     }
795
796     return rc;
797
798 #else
799
800     ngx_fd_t fd;
801
802     if (of->disable_symlinks == NGX_DISABLE_SYMLINKS_OFF) {
803
804         rc = ngx_file_info(name->data, fi);
805
806         if (rc == NGX_FILE_ERROR) {
807             of->err = ngx_errno;
808             of->failed = ngx_file_info_n;
809             return NGX_FILE_ERROR;
810         }
811
812         return rc;
813     }
814
815     fd = ngx_open_file_wrapper(name, of, NGX_FILE_RDONLY|NGX_FILE_NONBLOCK,
816                               NGX_FILE_OPEN, 0, log);
817
818     if (fd == NGX_INVALID_FILE) {
819         return NGX_FILE_ERROR;
820     }
821
822     rc = ngx_fd_info(fd, fi);
823
824     if (rc == NGX_FILE_ERROR) {
825         of->err = ngx_errno;
826         of->failed = ngx_fd_info_n;
827     }
828
829     if (ngx_close_file(fd) == NGX_FILE_ERROR) {
830         ngx_log_error(NGX_LOG_ALERT, log, ngx_errno,
831                     ngx_close_file_n " \"%V\" failed", name);
832     }
833
834     return rc;
835 #endif
836 }
837
838
839 static ngx_int_t
840 ngx_open_and_stat_file(ngx_str_t *name, ngx_open_file_info_t *of,
841                       ngx_log_t *log)
842 {
843     ngx_fd_t fd;
844     ngx_file_info_t fi;
845
846     if (of->fd != NGX_INVALID_FILE) {
847
848         if (ngx_file_info_wrapper(name, of, &fi, log) == NGX_FILE_ERROR) {
849             of->fd = NGX_INVALID_FILE;
850             return NGX_ERROR;
851         }
852
853         if (of->uniq == ngx_file_uniq(&fi)) {
854             goto done;
855         }
856
857     } else if (of->test_dir) {
858
859         if (ngx_file_info_wrapper(name, of, &fi, log) == NGX_FILE_ERROR) {

```

```

860         of->fd = NGX\_INVALID\_FILE;
861         return NGX\_ERROR;
862     }
863
864     if (ngx\_is\_dir(&fi)) {
865         goto done;
866     }
867 }
868
869 if (!of->log) {
870
871     /*
872     * Use non-blocking open() not to hang on FIFO files, etc.
873     * This flag has no effect on a regular files.
874     */
875
876     fd = ngx\_open\_file\_wrapper(name, of, NGX\_FILE\_RDONLY|NGX\_FILE\_NONBLOCK,
877                               NGX\_FILE\_OPEN, 0, log);
878
879 } else {
880     fd = ngx\_open\_file\_wrapper(name, of, NGX\_FILE\_APPEND,
881                               NGX\_FILE\_CREATE\_OR\_OPEN,
882                               NGX\_FILE\_DEFAULT\_ACCESS, log);
883 }
884
885 if (fd == NGX\_INVALID\_FILE) {
886     of->fd = NGX\_INVALID\_FILE;
887     return NGX\_ERROR;
888 }
889
890 if (ngx\_fd\_info(fd, &fi) == NGX\_FILE\_ERROR) {
891     ngx\_log\_error(NGX\_LOG\_CRIT, log, ngx\_errno,
892                 ngx\_fd\_info\_n " \"%V\" failed", name);
893
894     if (ngx\_close\_file(fd) == NGX\_FILE\_ERROR) {
895         ngx\_log\_error(NGX\_LOG\_ALERT, log, ngx\_errno,
896                     ngx\_close\_file\_n " \"%V\" failed", name);
897     }
898
899     of->fd = NGX\_INVALID\_FILE;
900
901     return NGX\_ERROR;
902 }
903
904 if (ngx\_is\_dir(&fi)) {
905     if (ngx\_close\_file(fd) == NGX\_FILE\_ERROR) {
906         ngx\_log\_error(NGX\_LOG\_ALERT, log, ngx\_errno,
907                     ngx\_close\_file\_n " \"%V\" failed", name);
908     }
909
910     of->fd = NGX\_INVALID\_FILE;
911
912 } else {
913     of->fd = fd;
914
915     if (of->read_ahead && ngx\_file\_size(&fi) > NGX\_MIN\_READ\_AHEAD) {
916         if (ngx\_read\_ahead(fd, of->read_ahead) == NGX\_ERROR) {
917             ngx\_log\_error(NGX\_LOG\_ALERT, log, ngx\_errno,
918                         ngx\_read\_ahead\_n " \"%V\" failed", name);
919         }
920     }
921
922     if (of->directio <= ngx\_file\_size(&fi)) {
923         if (ngx\_directio\_on(fd) == NGX\_FILE\_ERROR) {
924             ngx\_log\_error(NGX\_LOG\_ALERT, log, ngx\_errno,
925                         ngx\_directio\_on\_n " \"%V\" failed", name);
926
927         } else {
928             of->is_directio = 1;
929         }
930     }
931 }
932
933 done:
934
935 of->uniq = ngx\_file\_uniq(&fi);

```



```

936 of->mtime = ngx_file_mtime(&fi);
937 of->size = ngx_file_size(&fi);
938 of->fs_size = ngx_file_fs_size(&fi);
939 of->is_dir = ngx_is_dir(&fi);
940 of->is_file = ngx_is_file(&fi);
941 of->is_link = ngx_is_link(&fi);
942 of->is_exec = ngx_is_exec(&fi);
943
944 return NGX_OK;
945 }
946
947
948 /*
949  * we ignore any possible event setting error and
950  * fallback to usual periodic file retests
951  */
952
953 static void
954 ngx_open_file_add_event(ngx_open_file_cache_t *cache,
955 ngx_cached_open_file_t *file, ngx_open_file_info_t *of, ngx_log_t *log)
956 {
957 ngx_open_file_cache_event_t *fev;
958
959 if (!(ngx_event_flags & NGX_USE_VNODE_EVENT)
960     || !of->events
961     || file->event
962     || of->fd == NGX_INVALID_FILE
963     || file->uses < of->min_uses)
964 {
965     return;
966 }
967
968 file->use_event = 0;
969
970 file->event = ngx_calloc(sizeof(ngx_event_t), log);
971 if (file->event == NULL) {
972     return;
973 }
974
975 fev = ngx_alloc(sizeof(ngx_open_file_cache_event_t), log);
976 if (fev == NULL) {
977     ngx_free(file->event);
978     file->event = NULL;
979     return;
980 }
981
982 fev->fd = of->fd;
983 fev->file = file;
984 fev->cache = cache;
985
986 file->event->handler = ngx_open_file_cache_remove;
987 file->event->data = fev;
988
989 /*
990  * although vnode event may be called while ngx_cycle->poll
991  * destruction, however, cleanup procedures are run before any
992  * memory freeing and events will be canceled.
993  */
994
995 file->event->log = ngx_cycle->log;
996
997 if (ngx_add_event(file->event, NGX_VNODE_EVENT, NGX_ONESHOT_EVENT)
998     != NGX_OK)
999 {
1000     ngx_free(file->event->data);
1001     ngx_free(file->event);
1002     file->event = NULL;
1003     return;
1004 }
1005
1006 /*
1007  * we do not set file->use_event here because there may be a race
1008  * condition: a file may be deleted between opening the file and
1009  * adding event, so we rely upon event notification only after
1010  * one file revalidation on next file access
1011  */

```

```

1012     return;
1013 }
1014 }
1015
1016
1017 static void
1018 ngx_open_file_cleanup(void *data)
1019 {
1020     ngx_open_file_cache_cleanup_t *c = data;
1021
1022     c->file->count--;
1023
1024     ngx_close_cached_file(c->cache, c->file, c->min_uses, c->log);
1025
1026     /* drop one or two expired open files */
1027     ngx_expire_old_cached_files(c->cache, 1, c->log);
1028 }
1029
1030
1031 static void
1032 ngx_close_cached_file(ngx_open_file_cache_t *cache,
1033     ngx_cached_open_file_t *file, ngx_uint_t min_uses, ngx_log_t *log)
1034 {
1035     ngx_log_debug5(NGX_LOG_DEBUG_CORE, log, 0,
1036         "close cached open file: %s, fd:%d, c:%d, u:%d, %d",
1037         file->name, file->fd, file->count, file->uses, file->close);
1038
1039     if (!file->close) {
1040
1041         file->accessed = ngx_time();
1042
1043         ngx_queue_remove(&file->queue);
1044
1045         ngx_queue_insert_head(&cache->expire_queue, &file->queue);
1046
1047         if (file->uses >= min_uses || file->count) {
1048             return;
1049         }
1050     }
1051
1052     ngx_open_file_del_event(file);
1053
1054     if (file->count) {
1055         return;
1056     }
1057
1058     if (file->fd != NGX_INVALID_FILE) {
1059
1060         if (ngx_close_file(file->fd) == NGX_FILE_ERROR) {
1061             ngx_log_error(NGX_LOG_ALERT, log, ngx_errno,
1062                 ngx_close_file_n " \"%s\" failed", file->name);
1063         }
1064
1065         file->fd = NGX_INVALID_FILE;
1066     }
1067
1068     if (!file->close) {
1069         return;
1070     }
1071
1072     ngx_free(file->name);
1073     ngx_free(file);
1074 }
1075
1076
1077 static void
1078 ngx_open_file_del_event(ngx_cached_open_file_t *file)
1079 {
1080     if (file->event == NULL) {
1081         return;
1082     }
1083
1084     (void) ngx_del_event(file->event, NGX_VNODE_EVENT,
1085         file->count ? NGX_FLUSH_EVENT : NGX_CLOSE_EVENT);
1086
1087     ngx_free(file->event->data);

```

```

1088     ngx_free(file->event);
1089     file->event = NULL;
1090     file->use_event = 0;
1091 }
1092
1093
1094 static void
1095 ngx_expire_old_cached_files(ngx_open_file_cache_t *cache, ngx_uint_t n,
1096     ngx_log_t *log)
1097 {
1098     time_t          now;
1099     ngx_queue_t     *q;
1100     ngx_cached_open_file_t *file;
1101
1102     now = ngx_time();
1103
1104     /*
1105      * n == 1 deletes one or two inactive files
1106      * n == 0 deletes least recently used file by force
1107      *         and one or two inactive files
1108      */
1109
1110     while (n < 3) {
1111         if (ngx_queue_empty(&cache->expire_queue)) {
1112             return;
1113         }
1114
1115         q = ngx_queue_last(&cache->expire_queue);
1116
1117         file = ngx_queue_data(q, ngx_cached_open_file_t, queue);
1118
1119         if (n++ != 0 && now - file->accessed <= cache->inactive) {
1120             return;
1121         }
1122     }
1123
1124     ngx_queue_remove(q);
1125
1126     ngx_rbtree_delete(&cache->rbtree, &file->node);
1127
1128     cache->current--;
1129
1130     ngx_log_debug1(NGX_LOG_DEBUG_CORE, log, 0,
1131         "expire cached open file: %s", file->name);
1132
1133     if (!file->err && !file->is_dir) {
1134         file->close = 1;
1135         ngx_close_cached_file(cache, file, 0, log);
1136     } else {
1137         ngx_free(file->name);
1138         ngx_free(file);
1139     }
1140 }
1141 }
1142 }
1143
1144
1145 static void
1146 ngx_open_file_cache_rbtree_insert_value(ngx_rbtree_node_t *temp,
1147     ngx_rbtree_node_t *node, ngx_rbtree_node_t *sentinel)
1148 {
1149     ngx_rbtree_node_t **p;
1150     ngx_cached_open_file_t *file, *file_temp;
1151
1152     for ( ;; ) {
1153         if (node->key < temp->key) {
1154             p = &temp->left;
1155         } else if (node->key > temp->key) {
1156             p = &temp->right;
1157         } else { /* node->key == temp->key */

```

```

1164         file = (ngx_cached_open_file_t *) node;
1165         file_temp = (ngx_cached_open_file_t *) temp;
1166
1167         p = (ngx_strcmp(file->name, file_temp->name) < 0)
1168             ? &temp->left : &temp->right;
1169     }
1170
1171     if (*p == sentinel) {
1172         break;
1173     }
1174
1175     temp = *p;
1176 }
1177
1178 *p = node;
1179 node->parent = temp;
1180 node->left = sentinel;
1181 node->right = sentinel;
1182 ngx_rbt_red(node);
1183 }
1184
1185
1186 static ngx_cached_open_file_t *
1187 ngx_open_file_lookup(ngx_open_file_cache_t *cache, ngx_str_t *name,
1188                     uint32_t hash)
1189 {
1190     ngx_int_t          rc;
1191     ngx_rbtree_node_t *node, *sentinel;
1192     ngx_cached_open_file_t *file;
1193
1194     node = cache->rbtree.root;
1195     sentinel = cache->rbtree.sentinel;
1196
1197     while (node != sentinel) {
1198
1199         if (hash < node->key) {
1200             node = node->left;
1201             continue;
1202         }
1203
1204         if (hash > node->key) {
1205             node = node->right;
1206             continue;
1207         }
1208
1209         /* hash == node->key */
1210
1211         file = (ngx_cached_open_file_t *) node;
1212
1213         rc = ngx_strcmp(name->data, file->name);
1214
1215         if (rc == 0) {
1216             return file;
1217         }
1218
1219         node = (rc < 0) ? node->left : node->right;
1220     }
1221
1222     return NULL;
1223 }
1224
1225
1226 static void
1227 ngx_open_file_cache_remove(ngx_event_t *ev)
1228 {
1229     ngx_cached_open_file_t *file;
1230     ngx_open_file_cache_event_t *fev;
1231
1232     fev = ev->data;
1233     file = fev->file;
1234
1235     ngx_queue_remove(&file->queue);
1236
1237     ngx_rbtree_delete(&fev->cache->rbtree, &file->node);
1238
1239     fev->cache->current--;

```

```
1240
1241 /* NGX_ONESHOT_EVENT was already deleted */
1242 file->event = NULL;
1243 file->use_event = 0;
1244
1245 file->close = 1;
1246
1247 ngx\_close\_cached\_file(fev->cache, file, 0, ev->log);
1248
1249 /* free memory only when fev->cache and fev->file are already not needed */
1250
1251 ngx\_free(ev->data);
1252 ngx\_free(ev);
1253 }
```

[One Level Up](#)

[Top Level](#)

# src/http/nginx\_http\_script.h - nginx-1.7.10

## Data types defined

- [ngx\\_http\\_compile\\_complex\\_value\\_t](#)
- [ngx\\_http\\_complex\\_value\\_t](#)
- [ngx\\_http\\_script\\_code\\_pt](#)
- [ngx\\_http\\_script\\_compile\\_t](#)
- [ngx\\_http\\_script\\_complex\\_value\\_code\\_t](#)
- [ngx\\_http\\_script\\_copy\\_capture\\_code\\_t](#)
- [ngx\\_http\\_script\\_copy\\_code\\_t](#)
- [ngx\\_http\\_script\\_engine\\_t](#)
- [ngx\\_http\\_script\\_file\\_code\\_t](#)
- [ngx\\_http\\_script\\_file\\_op\\_e](#)
- [ngx\\_http\\_script\\_full\\_name\\_code\\_t](#)
- [ngx\\_http\\_script\\_if\\_code\\_t](#)
- [ngx\\_http\\_script\\_len\\_code\\_pt](#)
- [ngx\\_http\\_script\\_regex\\_code\\_t](#)
- [ngx\\_http\\_script\\_regex\\_end\\_code\\_t](#)
- [ngx\\_http\\_script\\_return\\_code\\_t](#)
- [ngx\\_http\\_script\\_value\\_code\\_t](#)
- [ngx\\_http\\_script\\_var\\_code\\_t](#)
- [ngx\\_http\\_script\\_var\\_handler\\_code\\_t](#)

## Macros defined

- [\\_NGX\\_HTTP\\_SCRIPT\\_H\\_INCLUDED](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_HTTP\_SCRIPT\_H\_INCLUDED
9 #define \_NGX\_HTTP\_SCRIPT\_H\_INCLUDED
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14 #include <ngx_http.h>
15
16
```

```

17 typedef struct {
18     u_char                *ip;
19     u_char                *pos;
20     ngx_http_variable_value_t *sp;
21
22     ngx_str_t             buf;
23     ngx_str_t             line;
24
25     /* the start of the rewritten arguments */
26     u_char                *args;
27
28     unsigned               flushed:1;
29     unsigned               skip:1;
30     unsigned               quote:1;
31     unsigned               is_args:1;
32     unsigned               log:1;
33
34     ngx_int_t              status;
35     ngx_http_request_t     *request;
36 } ngx_http_script_engine_t;
37
38
39 typedef struct {
40     ngx_conf_t             *cf;
41     ngx_str_t              *source;
42
43     ngx_array_t             **flushes;
44     ngx_array_t             **lengths;
45     ngx_array_t             **values;
46
47     ngx_uint_t              variables;
48     ngx_uint_t              ncaptures;
49     ngx_uint_t              captures_mask;
50     ngx_uint_t              size;
51
52     void                    *main;
53
54     unsigned                compile_args:1;
55     unsigned                complete_lengths:1;
56     unsigned                complete_values:1;
57     unsigned                zero:1;
58     unsigned                conf_prefix:1;
59     unsigned                root_prefix:1;
60
61     unsigned                dup_capture:1;
62     unsigned                args:1;
63 } ngx_http_script_compile_t;
64
65
66 typedef struct {
67     ngx_str_t              value;
68     ngx_uint_t             *flushes;
69     void                    *lengths;
70     void                    *values;
71 } ngx_http_complex_value_t;
72
73
74 typedef struct {
75     ngx_conf_t             *cf;
76     ngx_str_t              *value;
77     ngx_http_complex_value_t *complex_value;
78
79     unsigned                zero:1;
80     unsigned                conf_prefix:1;
81     unsigned                root_prefix:1;
82 } ngx_http_compile_complex_value_t;
83
84
85 typedef void (*ngx_http_script_code_pt) (ngx_http_script_engine_t *e);
86 typedef size_t (*ngx_http_script_len_code_pt) (ngx_http_script_engine_t *e);
87
88
89 typedef struct {
90     ngx_http_script_code_pt code;
91     uintptr_t               len;
92 } ngx_http_script_copy_code_t;

```

```

93
94
95 typedef struct {
96     ngx_http_script_code_pt    code;
97     uintptr_t                  index;
98 } ngx_http_script_var_code_t;
99
100
101 typedef struct {
102     ngx_http_script_code_pt    code;
103     ngx_http_set_variable_pt   handler;
104     uintptr_t                   data;
105 } ngx_http_script_var_handler_code_t;
106
107
108 typedef struct {
109     ngx_http_script_code_pt    code;
110     uintptr_t                   n;
111 } ngx_http_script_copy_capture_code_t;
112
113
114 #if (NGX_PCRE)
115
116 typedef struct {
117     ngx_http_script_code_pt    code;
118     ngx_http_regex_t           *regex;
119     ngx_array_t                 *lengths;
120     uintptr_t                   size;
121     uintptr_t                   status;
122     uintptr_t                   next;
123
124     uintptr_t                   test:1;
125     uintptr_t                   negative_test:1;
126     uintptr_t                   uri:1;
127     uintptr_t                   args:1;
128
129     /* add the r->args to the new arguments */
130     uintptr_t                   add_args:1;
131
132     uintptr_t                   redirect:1;
133     uintptr_t                   break_cycle:1;
134
135     ngx_str_t                   name;
136 } ngx_http_script_regex_code_t;
137
138
139 typedef struct {
140     ngx_http_script_code_pt    code;
141
142     uintptr_t                   uri:1;
143     uintptr_t                   args:1;
144
145     /* add the r->args to the new arguments */
146     uintptr_t                   add_args:1;
147
148     uintptr_t                   redirect:1;
149 } ngx_http_script_regex_end_code_t;
150
151 #endif
152
153
154 typedef struct {
155     ngx_http_script_code_pt    code;
156     uintptr_t                   conf_prefix;
157 } ngx_http_script_full_name_code_t;
158
159
160 typedef struct {
161     ngx_http_script_code_pt    code;
162     uintptr_t                   status;
163     ngx_http_complex_value_t    text;
164 } ngx_http_script_return_code_t;
165
166
167 typedef enum {
168     ngx_http_script_file_plain = 0,

```



```

169     ngx_http_script_file_not_plain,
170     ngx_http_script_file_dir,
171     ngx_http_script_file_not_dir,
172     ngx_http_script_file_exists,
173     ngx_http_script_file_not_exists,
174     ngx_http_script_file_exec,
175     ngx_http_script_file_not_exec
176 } ngx_http_script_file_op_e;
177
178
179 typedef struct {
180     ngx_http_script_code_pt    code;
181     uintptr_t                  op;
182 } ngx_http_script_file_code_t;
183
184
185 typedef struct {
186     ngx_http_script_code_pt    code;
187     uintptr_t                  next;
188     void                       **loc_conf;
189 } ngx_http_script_if_code_t;
190
191
192 typedef struct {
193     ngx_http_script_code_pt    code;
194     ngx_array_t                *lengths;
195 } ngx_http_script_complex_value_code_t;
196
197
198 typedef struct {
199     ngx_http_script_code_pt    code;
200     uintptr_t                  value;
201     uintptr_t                  text_len;
202     uintptr_t                  text_data;
203 } ngx_http_script_value_code_t;
204
205
206 void ngx_http_script_flush_complex_value(ngx_http_request_t *r,
207     ngx_http_complex_value_t *val);
208 ngx_int_t ngx_http_complex_value(ngx_http_request_t *r,
209     ngx_http_complex_value_t *val, ngx_str_t *value);
210 ngx_int_t ngx_http_compile_complex_value(ngx_http_compile_complex_value_t *ccv);
211 char *ngx_http_set_complex_value_slot(ngx_conf_t *cf, ngx_command_t *cmd,
212     void *conf);
213
214
215 ngx_int_t ngx_http_test_predicates(ngx_http_request_t *r,
216     ngx_array_t *predicates);
217 char *ngx_http_set_predicate_slot(ngx_conf_t *cf, ngx_command_t *cmd,
218     void *conf);
219
220 ngx_uint_t ngx_http_script_variables_count(ngx_str_t *value);
221 ngx_int_t ngx_http_script_compile(ngx_http_script_compile_t *sc);
222 u_char *ngx_http_script_run(ngx_http_request_t *r, ngx_str_t *value,
223     void *code_lengths, size_t reserved, void *code_values);
224 void ngx_http_script_flush_no_cacheable_variables(ngx_http_request_t *r,
225     ngx_array_t *indices);
226
227 void *ngx_http_script_start_code(ngx_pool_t *pool, ngx_array_t **codes,
228     size_t size);
229 void *ngx_http_script_add_code(ngx_array_t *codes, size_t size, void *code);
230
231 size_t ngx_http_script_copy_len_code(ngx_http_script_engine_t *e);
232 void ngx_http_script_copy_code(ngx_http_script_engine_t *e);
233 size_t ngx_http_script_copy_var_len_code(ngx_http_script_engine_t *e);
234 void ngx_http_script_copy_var_code(ngx_http_script_engine_t *e);
235 size_t ngx_http_script_copy_capture_len_code(ngx_http_script_engine_t *e);
236 void ngx_http_script_copy_capture_code(ngx_http_script_engine_t *e);
237 size_t ngx_http_script_mark_args_code(ngx_http_script_engine_t *e);
238 void ngx_http_script_start_args_code(ngx_http_script_engine_t *e);
239 #if (NGX_PCRE)
240 void ngx_http_script_regex_start_code(ngx_http_script_engine_t *e);
241 void ngx_http_script_regex_end_code(ngx_http_script_engine_t *e);
242 #endif
243 void ngx_http_script_return_code(ngx_http_script_engine_t *e);
244 void ngx_http_script_break_code(ngx_http_script_engine_t *e);

```

```
245 void ngx\_http\_script\_if\_code(ngx_http_script_engine_t *e);
246 void ngx\_http\_script\_equal\_code(ngx_http_script_engine_t *e);
247 void ngx\_http\_script\_not\_equal\_code(ngx_http_script_engine_t *e);
248 void ngx\_http\_script\_file\_code(ngx_http_script_engine_t *e);
249 void ngx\_http\_script\_complex\_value\_code(ngx_http_script_engine_t *e);
250 void ngx\_http\_script\_value\_code(ngx_http_script_engine_t *e);
251 void ngx\_http\_script\_set\_var\_code(ngx_http_script_engine_t *e);
252 void ngx\_http\_script\_var\_set\_handler\_code(ngx_http_script_engine_t *e);
253 void ngx\_http\_script\_var\_code(ngx_http_script_engine_t *e);
254 void ngx\_http\_script\_nop\_code(ngx_http_script_engine_t *e);
255
256
257 #endif /* \_NGX\_HTTP\_SCRIPT\_H\_INCLUDED\_ */
```

[One Level Up](#)

[Top Level](#)

## src/http/nginx\_http\_script.c - nginx-1.7.10

### Global variables defined

- [ngx\\_http\\_script\\_exit\\_code](#)

### Functions defined

- [ngx\\_http\\_compile\\_complex\\_value](#)
- [ngx\\_http\\_complex\\_value](#)
- [ngx\\_http\\_script\\_add\\_args\\_code](#)
- [ngx\\_http\\_script\\_add\\_capture\\_code](#)
- [ngx\\_http\\_script\\_add\\_code](#)
- [ngx\\_http\\_script\\_add\\_copy\\_code](#)
- [ngx\\_http\\_script\\_add\\_full\\_name\\_code](#)
- [ngx\\_http\\_script\\_add\\_var\\_code](#)
- [ngx\\_http\\_script\\_break\\_code](#)
- [ngx\\_http\\_script\\_compile](#)
- [ngx\\_http\\_script\\_complex\\_value\\_code](#)
- [ngx\\_http\\_script\\_copy\\_capture\\_code](#)
- [ngx\\_http\\_script\\_copy\\_capture\\_len\\_code](#)
- [ngx\\_http\\_script\\_copy\\_code](#)
- [ngx\\_http\\_script\\_copy\\_len\\_code](#)
- [ngx\\_http\\_script\\_copy\\_var\\_code](#)
- [ngx\\_http\\_script\\_copy\\_var\\_len\\_code](#)
- [ngx\\_http\\_script\\_done](#)
- [ngx\\_http\\_script\\_equal\\_code](#)
- [ngx\\_http\\_script\\_file\\_code](#)
- [ngx\\_http\\_script\\_flush\\_complex\\_value](#)
- [ngx\\_http\\_script\\_flush\\_no\\_cacheable\\_variables](#)
- [ngx\\_http\\_script\\_full\\_name\\_code](#)
- [ngx\\_http\\_script\\_full\\_name\\_len\\_code](#)
- [ngx\\_http\\_script\\_if\\_code](#)
- [ngx\\_http\\_script\\_init\\_arrays](#)
- [ngx\\_http\\_script\\_mark\\_args\\_code](#)

- [ngx http script nop code](#)
- [ngx http script not equal code](#)
- [ngx http script regex end code](#)
- [ngx http script regex start code](#)
- [ngx http script return code](#)
- [ngx http script run](#)
- [ngx http script set var code](#)
- [ngx http script start args code](#)
- [ngx http script start code](#)
- [ngx http script value code](#)
- [ngx http script var code](#)
- [ngx http script var set handler code](#)
- [ngx http script variables count](#)
- [ngx http set complex value slot](#)
- [ngx http set predicate slot](#)
- [ngx http test predicates](#)

## Macros defined

- [ngx http script exit](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 static ngx_int_t ngx_http_script_init_arrays(ngx_http_script_compile_t *sc);
14 static ngx_int_t ngx_http_script_done(ngx_http_script_compile_t *sc);
15 static ngx_int_t ngx_http_script_add_copy_code(ngx_http_script_compile_t *sc,
16     ngx_str_t *value, ngx_uint_t last);
17 static ngx_int_t ngx_http_script_add_var_code(ngx_http_script_compile_t *sc,
18     ngx_str_t *name);
19 static ngx_int_t ngx_http_script_add_args_code(ngx_http_script_compile_t *sc);
20 #if (NGX_PCRE)
21 static ngx_int_t ngx_http_script_add_capture_code(ngx_http_script_compile_t *sc,
22     ngx_uint_t n);
23 #endif
24 static ngx_int_t
25     ngx_http_script_add_full_name_code(ngx_http_script_compile_t *sc);
26 static size_t ngx_http_script_full_name_len_code(ngx_http_script_engine_t *e);
27 static void ngx_http_script_full_name_code(ngx_http_script_engine_t *e);
28
29
30 #define ngx_http_script_exit (u_char *) &ngx_http_script_exit_code

```

```

31
32 static uintptr_t ngx_http_script_exit_code = (uintptr_t) NULL;
33
34
35 void
36 ngx_http_script_flush_complex_value(ngx_http_request_t *r,
37     ngx_http_complex_value_t *val)
38 {
39     ngx_uint_t *index;
40
41     index = val->flushes;
42
43     if (index) {
44         while (*index != (ngx_uint_t) -1) {
45
46             if (r->variables[*index].no_cacheable) {
47                 r->variables[*index].valid = 0;
48                 r->variables[*index].not_found = 0;
49             }
50
51             index++;
52         }
53     }
54 }
55
56
57 ngx_int_t
58 ngx_http_complex_value(ngx_http_request_t *r, ngx_http_complex_value_t *val,
59     ngx_str_t *value)
60 {
61     size_t len;
62     ngx_http_script_code_pt code;
63     ngx_http_script_len_code_pt lcode;
64     ngx_http_script_engine_t e;
65
66     if (val->lengths == NULL) {
67         *value = val->value;
68         return NGX_OK;
69     }
70
71     ngx_http_script_flush_complex_value(r, val);
72
73     ngx_memzero(&e, sizeof(ngx_http_script_engine_t));
74
75     e.ip = val->lengths;
76     e.request = r;
77     e.flushed = 1;
78
79     len = 0;
80
81     while (*(uintptr_t *) e.ip) {
82         lcode = *(ngx_http_script_len_code_pt *) e.ip;
83         len += lcode(&e);
84     }
85
86     value->len = len;
87     value->data = ngx_pnalloc(r->pool, len);
88     if (value->data == NULL) {
89         return NGX_ERROR;
90     }
91
92     e.ip = val->values;
93     e.pos = value->data;
94     e.buf = *value;
95
96     while (*(uintptr_t *) e.ip) {
97         code = *(ngx_http_script_code_pt *) e.ip;
98         code((ngx_http_script_engine_t *) &e);
99     }
100
101     *value = e.buf;
102
103     return NGX_OK;
104 }
105
106

```

```

107 ngx_int_t
108 ngx_http_compile_complex_value(ngx_http_compile_complex_value_t *ccv)
109 {
110     ngx_str_t          *v;
111     ngx_uint_t         i, n, nv, nc;
112     ngx_array_t        flushes, lengths, values, *pf, *pl, *pv;
113     ngx_http_script_compile_t sc;
114
115     v = ccv->value;
116
117     nv = 0;
118     nc = 0;
119
120     for (i = 0; i < v->len; i++) {
121         if (v->data[i] == '$') {
122             if (v->data[i + 1] >= '1' && v->data[i + 1] <= '9') {
123                 nc++;
124             } else {
125                 nv++;
126             }
127         }
128     }
129 }
130
131 if ((v->len == 0 || v->data[0] != '$')
132     && (ccv->conf_prefix || ccv->root_prefix))
133 {
134     if (ngx_conf_full_name(ccv->cf->cycle, v, ccv->conf_prefix) != NGX_OK) {
135         return NGX_ERROR;
136     }
137
138     ccv->conf_prefix = 0;
139     ccv->root_prefix = 0;
140 }
141
142 ccv->complex_value->value = *v;
143 ccv->complex_value->flushes = NULL;
144 ccv->complex_value->lengths = NULL;
145 ccv->complex_value->values = NULL;
146
147 if (nv == 0 && nc == 0) {
148     return NGX_OK;
149 }
150
151 n = nv + 1;
152
153 if (ngx_array_init(&flushes, ccv->cf->pool, n, sizeof(ngx_uint_t))
154     != NGX_OK)
155 {
156     return NGX_ERROR;
157 }
158
159 n = nv * (2 * sizeof(ngx_http_script_copy_code_t)
160          + sizeof(ngx_http_script_var_code_t))
161     + sizeof(uintptr_t);
162
163 if (ngx_array_init(&lengths, ccv->cf->pool, n, 1) != NGX_OK) {
164     return NGX_ERROR;
165 }
166
167 n = (nv * (2 * sizeof(ngx_http_script_copy_code_t)
168          + sizeof(ngx_http_script_var_code_t))
169     + sizeof(uintptr_t)
170     + v->len
171     + sizeof(uintptr_t) - 1)
172     & ~(sizeof(uintptr_t) - 1);
173
174 if (ngx_array_init(&values, ccv->cf->pool, n, 1) != NGX_OK) {
175     return NGX_ERROR;
176 }
177
178 pf = &flushes;
179 pl = &lengths;
180 pv = &values;
181
182 ngx_memzero(&sc, sizeof(ngx_http_script_compile_t));

```

```

183     sc.cf = ccv->cf;
184     sc.source = v;
185     sc.flushes = &pf;
186     sc.lengths = &pl;
187     sc.values = &pv;
188     sc.complete_lengths = 1;
189     sc.complete_values = 1;
190     sc.zero = ccv->zero;
191     sc.conf_prefix = ccv->conf_prefix;
192     sc.root_prefix = ccv->root_prefix;
193
194     if (ngx_http_script_compile(&sc) != NGX_OK) {
195         return NGX_ERROR;
196     }
197
198     if (flushes.nelts) {
199         ccv->complex_value->flushes = flushes.elts;
200         ccv->complex_value->flushes[flushes.nelts] = (ngx_uint_t) -1;
201     }
202
203     ccv->complex_value->lengths = lengths.elts;
204     ccv->complex_value->values = values.elts;
205
206     return NGX_OK;
207 }
208
209
210
211 char *
212 ngx_http_set_complex_value_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
213 {
214     char *p = conf;
215
216     ngx_str_t          *value;
217     ngx_http_complex_value_t **cv;
218     ngx_http_compile_complex_value_t ccv;
219
220     cv = (ngx_http_complex_value_t **) (p + cmd->offset);
221
222     if (*cv != NULL) {
223         return "duplicate";
224     }
225
226     *cv = ngx_palloc(cf->pool, sizeof(ngx_http_complex_value_t));
227     if (*cv == NULL) {
228         return NGX_CONF_ERROR;
229     }
230
231     value = cf->args->elts;
232
233     ngx_memzero(&ccv, sizeof(ngx_http_compile_complex_value_t));
234
235     ccv.cf = cf;
236     ccv.value = &value[1];
237     ccv.complex_value = *cv;
238
239     if (ngx_http_compile_complex_value(&ccv) != NGX_OK) {
240         return NGX_CONF_ERROR;
241     }
242
243     return NGX_CONF_OK;
244 }
245
246
247 ngx_int_t
248 ngx_http_test_predicates(ngx_http_request_t *r, ngx_array_t *predicates)
249 {
250     ngx_str_t          val;
251     ngx_uint_t         i;
252     ngx_http_complex_value_t *cv;
253
254     if (predicates == NULL) {
255         return NGX_OK;
256     }
257
258     cv = predicates->elts;

```

```

259
260     for (i = 0; i < predicates->nelts; i++) {
261         if (ngx\_http\_complex\_value(r, &cv[i], &val) != NGX\_OK) {
262             return NGX\_ERROR;
263         }
264
265         if (val.len && (val.len != 1 || val.data[0] != '0')) {
266             return NGX\_DECLINED;
267         }
268     }
269
270     return NGX\_OK;
271 }
272
273
274 char *
275 ngx\_http\_set\_predicate\_slot(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
276 {
277     char *p = conf;
278
279     ngx\_str\_t          *value;
280     ngx\_uint\_t        i;
281     ngx\_array\_t       **a;
282     ngx\_http\_complex\_value\_t *cv;
283     ngx\_http\_compile\_complex\_value\_t ccv;
284
285     a = (ngx\_array\_t **) (p + cmd->offset);
286
287     if (*a == NGX\_CONF\_UNSET\_PTR) {
288         *a = ngx\_array\_create(cf->pool, 1, sizeof(ngx\_http\_complex\_value\_t));
289         if (*a == NULL) {
290             return NGX\_CONF\_ERROR;
291         }
292     }
293
294     value = cf->args->elts;
295
296     for (i = 1; i < cf->args->nelts; i++) {
297         cv = ngx\_array\_push(*a);
298         if (cv == NULL) {
299             return NGX\_CONF\_ERROR;
300         }
301
302         ngx\_memzero(&ccv, sizeof(ngx\_http\_compile\_complex\_value\_t));
303
304         ccv.cf = cf;
305         ccv.value = &value[i];
306         ccv.complex_value = cv;
307
308         if (ngx\_http\_compile\_complex\_value(&ccv) != NGX\_OK) {
309             return NGX\_CONF\_ERROR;
310         }
311     }
312
313     return NGX\_CONF\_OK;
314 }
315
316
317 ngx\_uint\_t
318 ngx\_http\_script\_variables\_count(ngx\_str\_t *value)
319 {
320     ngx\_uint\_t i, n;
321
322     for (n = 0, i = 0; i < value->len; i++) {
323         if (value->data[i] == '$') {
324             n++;
325         }
326     }
327
328     return n;
329 }
330
331
332 ngx\_int\_t
333 ngx\_http\_script\_compile(ngx\_http\_script\_compile\_t *sc)
334 {

```



```

335     u_char      ch;
336     ngx_str_t   name;
337     ngx_uint_t  i, bracket;
338
339     if (ngx_http_script_init_arrays(sc) != NGX_OK) {
340         return NGX_ERROR;
341     }
342
343     for (i = 0; i < sc->source->len; /* void */) {
344
345         name.len = 0;
346
347         if (sc->source->data[i] == '$') {
348
349             if (++i == sc->source->len) {
350                 goto invalid_variable;
351             }
352
353             #if (NGX_PCRE)
354             {
355                 ngx_uint_t  n;
356
357                 if (sc->source->data[i] >= '1' && sc->source->data[i] <= '9') {
358
359                     n = sc->source->data[i] - '0';
360
361                     if (sc->captures_mask & (1 << n)) {
362                         sc->dup_capture = 1;
363                     }
364
365                     sc->captures_mask |= 1 << n;
366
367                     if (ngx_http_script_add_capture_code(sc, n) != NGX_OK) {
368                         return NGX_ERROR;
369                     }
370
371                     i++;
372
373                     continue;
374                 }
375             }
376             #endif
377
378             if (sc->source->data[i] == '{') {
379                 bracket = 1;
380
381                 if (++i == sc->source->len) {
382                     goto invalid_variable;
383                 }
384
385                 name.data = &sc->source->data[i];
386
387             } else {
388                 bracket = 0;
389                 name.data = &sc->source->data[i];
390             }
391
392             for ( /* void */ ; i < sc->source->len; i++, name.len++) {
393                 ch = sc->source->data[i];
394
395                 if (ch == '}' && bracket) {
396                     i++;
397                     bracket = 0;
398                     break;
399                 }
400
401                 if ((ch >= 'A' && ch <= 'Z')
402                     || (ch >= 'a' && ch <= 'z')
403                     || (ch >= '0' && ch <= '9')
404                     || ch == '_')
405                 {
406                     continue;
407                 }
408
409                 break;
410             }

```

```

411     if (bracket) {
412         ngx_conf_log_error(NGX_LOG_EMERG, sc->cf, 0,
413             "the closing bracket in \"%V\" "
414             "variable is missing", &name);
415         return NGX_ERROR;
416     }
417
418     if (name.len == 0) {
419         goto invalid_variable;
420     }
421
422     sc->variables++;
423
424     if (ngx_http_script_add_var_code(sc, &name) != NGX_OK) {
425         return NGX_ERROR;
426     }
427
428     continue;
429 }
430
431 if (sc->source->data[i] == '?' && sc->compile_args) {
432     sc->args = 1;
433     sc->compile_args = 0;
434
435     if (ngx_http_script_add_args_code(sc) != NGX_OK) {
436         return NGX_ERROR;
437     }
438
439     i++;
440
441     continue;
442 }
443
444 name.data = &sc->source->data[i];
445
446 while (i < sc->source->len) {
447     if (sc->source->data[i] == '$') {
448         break;
449     }
450
451     if (sc->source->data[i] == '?') {
452         sc->args = 1;
453
454         if (sc->compile_args) {
455             break;
456         }
457     }
458
459     i++;
460     name.len++;
461 }
462
463 sc->size += name.len;
464
465 if (ngx_http_script_add_copy_code(sc, &name, (i == sc->source->len))
466     != NGX_OK)
467 {
468     return NGX_ERROR;
469 }
470
471 return ngx_http_script_done(sc);
472
473 invalid_variable:
474
475     ngx_conf_log_error(NGX_LOG_EMERG, sc->cf, 0, "invalid variable name");
476
477     return NGX_ERROR;
478 }
479
480 u_char *
481 ngx_http_script_run(ngx_http_request_t *r, ngx_str_t *value,

```

```

487 void *code_lengths, size_t len, void *code_values)
488 {
489     ngx_uint_t          i;
490     ngx_http_script_code_pt  code;
491     ngx_http_script_len_code_pt  lcode;
492     ngx_http_script_engine_t  e;
493     ngx_http_core_main_conf_t  *cmcf;
494
495     cmcf = ngx_http_get_module_main_conf(r, ngx_http_core_module);
496
497     for (i = 0; i < cmcf->variables.nelts; i++) {
498         if (r->variables[i].no_cacheable) {
499             r->variables[i].valid = 0;
500             r->variables[i].not_found = 0;
501         }
502     }
503
504     ngx_memzero(&e, sizeof(ngx_http_script_engine_t));
505
506     e.ip = code_lengths;
507     e.request = r;
508     e.flushed = 1;
509
510     while (*(uintptr_t *) e.ip) {
511         lcode = *(ngx_http_script_len_code_pt *) e.ip;
512         len += lcode(&e);
513     }
514
515     value->len = len;
516     value->data = ngx_pnalloc(r->pool, len);
517     if (value->data == NULL) {
518         return NULL;
519     }
520
521     e.ip = code_values;
522     e.pos = value->data;
523
524     while (*(uintptr_t *) e.ip) {
525         code = *(ngx_http_script_code_pt *) e.ip;
526         code((ngx_http_script_engine_t *) &e);
527     }
528
529     return e.pos;
530 }
531
532
533
534 void
535 ngx_http_script_flush_no_cacheable_variables(ngx_http_request_t *r,
536     ngx_array_t *indices)
537 {
538     ngx_uint_t  n, *index;
539
540     if (indices) {
541         index = indices->elts;
542         for (n = 0; n < indices->nelts; n++) {
543             if (r->variables[index[n]].no_cacheable) {
544                 r->variables[index[n]].valid = 0;
545                 r->variables[index[n]].not_found = 0;
546             }
547         }
548     }
549 }
550
551
552 static ngx_int_t
553 ngx_http_script_init_arrays(ngx_http_script_compile_t *sc)
554 {
555     ngx_uint_t  n;
556
557     if (sc->flushes && *sc->flushes == NULL) {
558         n = sc->variables ? sc->variables : 1;
559         *sc->flushes = ngx_array_create(sc->cf->pool, n, sizeof(ngx_uint_t));
560         if (*sc->flushes == NULL) {
561             return NGX_ERROR;
562         }
563     }

```

```

563 }
564
565 if (*sc->lengths == NULL) {
566     n = sc->variables * (2 * sizeof(ngx\_http\_script\_copy\_code\_t)
567         + sizeof(ngx\_http\_script\_var\_code\_t))
568         + sizeof(uintptr\_t);
569
570     *sc->lengths = ngx\_array\_create(sc->cf->pool, n, 1);
571     if (*sc->lengths == NULL) {
572         return NGX\_ERROR;
573     }
574 }
575
576 if (*sc->values == NULL) {
577     n = (sc->variables * (2 * sizeof(ngx\_http\_script\_copy\_code\_t)
578         + sizeof(ngx\_http\_script\_var\_code\_t))
579         + sizeof(uintptr\_t)
580         + sc->source->len
581         + sizeof(uintptr\_t) - 1)
582         & ~(sizeof(uintptr\_t) - 1);
583
584     *sc->values = ngx\_array\_create(sc->cf->pool, n, 1);
585     if (*sc->values == NULL) {
586         return NGX\_ERROR;
587     }
588 }
589
590 sc->variables = 0;
591
592 return NGX\_OK;
593 }
594
595
596 static ngx\_int\_t
597 ngx\_http\_script\_done(ngx\_http\_script\_compile\_t *sc)
598 {
599     ngx\_str\_t    zero;
600     uintptr\_t   *code;
601
602     if (sc->zero) {
603
604         zero.len = 1;
605         zero.data = (u_char *) "\0";
606
607         if (ngx\_http\_script\_add\_copy\_code(sc, &zero, 0) != NGX\_OK) {
608             return NGX\_ERROR;
609         }
610     }
611
612     if (sc->conf_prefix || sc->root_prefix) {
613         if (ngx\_http\_script\_add\_full\_name\_code(sc) != NGX\_OK) {
614             return NGX\_ERROR;
615         }
616     }
617
618     if (sc->complete_lengths) {
619         code = ngx\_http\_script\_add\_code(*sc->lengths, sizeof(uintptr\_t), NULL);
620         if (code == NULL) {
621             return NGX\_ERROR;
622         }
623
624         *code = (uintptr\_t) NULL;
625     }
626
627     if (sc->complete_values) {
628         code = ngx\_http\_script\_add\_code(*sc->values, sizeof(uintptr\_t),
629             &sc->main);
630         if (code == NULL) {
631             return NGX\_ERROR;
632         }
633
634         *code = (uintptr\_t) NULL;
635     }
636
637     return NGX\_OK;
638 }

```

```

639
640
641 void *
642 ngx_http_script_start_code(ngx_pool_t *pool, ngx_array_t **codes, size_t size)
643 {
644     if (*codes == NULL) {
645         *codes = ngx_array_create(pool, 256, 1);
646         if (*codes == NULL) {
647             return NULL;
648         }
649     }
650
651     return ngx_array_push_n(*codes, size);
652 }
653
654
655 void *
656 ngx_http_script_add_code(ngx_array_t *codes, size_t size, void *code)
657 {
658     u_char *elts, **p;
659     void *new;
660
661     elts = codes->elts;
662
663     new = ngx_array_push_n(codes, size);
664     if (new == NULL) {
665         return NULL;
666     }
667
668     if (code) {
669         if (elts != codes->elts) {
670             p = code;
671             *p += (u_char *) codes->elts - elts;
672         }
673     }
674
675     return new;
676 }
677
678
679 static ngx_int_t
680 ngx_http_script_add_copy_code(ngx_http_script_compile_t *sc, ngx_str_t *value,
681 ngx_uint_t last)
682 {
683     u_char *p;
684     size_t size, len, zero;
685     ngx_http_script_copy_code_t *code;
686
687     zero = (sc->zero && last);
688     len = value->len + zero;
689
690     code = ngx_http_script_add_code(*sc->lengths,
691 sizeof(ngx_http_script_copy_code_t), NULL);
692     if (code == NULL) {
693         return NGX_ERROR;
694     }
695
696     code->code = (ngx_http_script_code_pt) ngx_http_script_copy_len_code;
697     code->len = len;
698
699     size = (sizeof(ngx_http_script_copy_code_t) + len + sizeof(uintptr_t) - 1)
700 & ~(sizeof(uintptr_t) - 1);
701
702     code = ngx_http_script_add_code(*sc->values, size, &sc->main);
703     if (code == NULL) {
704         return NGX_ERROR;
705     }
706
707     code->code = ngx_http_script_copy_code;
708     code->len = len;
709
710     p = ngx_cpymem((u_char *) code + sizeof(ngx_http_script_copy_code_t),
711 value->data, value->len);
712
713     if (zero) {
714         *p = '\\0';

```



```

791     if (code == NULL) {
792         return NGX\_ERROR;
793     }
794
795     code->code = ngx\_http\_script\_copy\_var\_code;
796     code->index = (uintptr_t) index;
797
798     return NGX\_OK;
799 }
800
801
802 size_t
803 ngx\_http\_script\_copy\_var\_len\_code(ngx\_http\_script\_engine\_t *e)
804 {
805     ngx\_http\_variable\_value\_t    *value;
806     ngx\_http\_script\_var\_code\_t    *code;
807
808     code = (ngx\_http\_script\_var\_code\_t *) e->ip;
809
810     e->ip += sizeof(ngx\_http\_script\_var\_code\_t);
811
812     if (e->flushed) {
813         value = ngx\_http\_get\_indexed\_variable(e->request, code->index);
814
815     } else {
816         value = ngx\_http\_get\_flushed\_variable(e->request, code->index);
817     }
818
819     if (value && !value->not_found) {
820         return value->len;
821     }
822
823     return 0;
824 }
825
826
827 void
828 ngx\_http\_script\_copy\_var\_code(ngx\_http\_script\_engine\_t *e)
829 {
830     u_char                *p;
831     ngx\_http\_variable\_value\_t    *value;
832     ngx\_http\_script\_var\_code\_t    *code;
833
834     code = (ngx\_http\_script\_var\_code\_t *) e->ip;
835
836     e->ip += sizeof(ngx\_http\_script\_var\_code\_t);
837
838     if (!e->skip) {
839
840         if (e->flushed) {
841             value = ngx\_http\_get\_indexed\_variable(e->request, code->index);
842
843         } else {
844             value = ngx\_http\_get\_flushed\_variable(e->request, code->index);
845         }
846
847         if (value && !value->not_found) {
848             p = e->pos;
849             e->pos = ngx\_copy(p, value->data, value->len);
850
851             ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP,
852                 e->request->connection->log, 0,
853                 "http script var: \"%s\"", e->pos - p, p);
854         }
855     }
856 }
857
858
859 static ngx\_int\_t
860 ngx\_http\_script\_add\_args\_code(ngx\_http\_script\_compile\_t *sc)
861 {
862     uintptr_t    *code;
863
864     code = ngx\_http\_script\_add\_code(*sc->lengths, sizeof(uintptr_t), NULL);
865     if (code == NULL) {
866         return NGX\_ERROR;

```

```

867     }
868
869     *code = (uintptr_t) ngx_http_script_mark_args_code;
870
871     code = ngx_http_script_add_code(*sc->values, sizeof(uintptr_t), &sc->main);
872     if (code == NULL) {
873         return NGX_ERROR;
874     }
875
876     *code = (uintptr_t) ngx_http_script_start_args_code;
877
878     return NGX_OK;
879 }
880
881 size_t
882 ngx_http_script_mark_args_code(ngx_http_script_engine_t *e)
883 {
884     e->is_args = 1;
885     e->ip += sizeof(uintptr_t);
886
887     return 1;
888 }
889
890
891 void
892 ngx_http_script_start_args_code(ngx_http_script_engine_t *e)
893 {
894     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, e->request->connection->log, 0,
895                  "http script args");
896
897     e->is_args = 1;
898     e->args = e->pos;
899     e->ip += sizeof(uintptr_t);
900 }
901
902
903
904 #if (NGX_PCRE)
905
906 void
907 ngx_http_script_regex_start_code(ngx_http_script_engine_t *e)
908 {
909     size_t          len;
910     ngx_int_t      rc;
911     ngx_uint_t     n;
912     ngx_http_request_t *r;
913     ngx_http_script_engine_t le;
914     ngx_http_script_len_code_pt lcode;
915     ngx_http_script_regex_code_t *code;
916
917     code = (ngx_http_script_regex_code_t *) e->ip;
918
919     r = e->request;
920
921     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
922                  "http script regex: \"%V\"", &code->name);
923
924     if (code->uri) {
925         e->line = r->uri;
926     } else {
927         e->sp--;
928         e->line.len = e->sp->len;
929         e->line.data = e->sp->data;
930     }
931
932     rc = ngx_http_regex_exec(r, code->regex, &e->line);
933
934     if (rc == NGX_DECLINED) {
935         if (e->log || (r->connection->log->log_level & NGX_LOG_DEBUG_HTTP)) {
936             ngx_log_error(NGX_LOG_NOTICE, r->connection->log, 0,
937                          "\"%V\" does not match \"%V\"",
938                          &code->name, &e->line);
939         }
940
941         r->ncaptures = 0;
942

```





```

1019     }
1020 }
1021
1022     for (n = 2; n < r->ncaptures; n += 2) {
1023         e->buf.len += r->captures[n + 1] - r->captures[n];
1024     }
1025
1026 } else {
1027     ngx_memzero(&le, sizeof(ngx_http_script_engine_t));
1028
1029     le.ip = code->lengths->elts;
1030     le.line = e->line;
1031     le.request = r;
1032     le.quote = code->redirect;
1033
1034     len = 0;
1035
1036     while (*(uintptr_t *) le.ip) {
1037         lcode = *(ngx_http_script_len_code_pt *) le.ip;
1038         len += lcode(&le);
1039     }
1040
1041     e->buf.len = len;
1042 }
1043
1044 if (code->add_args && r->args.len) {
1045     e->buf.len += r->args.len + 1;
1046 }
1047
1048 e->buf.data = ngx_pnalloc(r->pool, e->buf.len);
1049 if (e->buf.data == NULL) {
1050     e->ip = ngx_http_script_exit;
1051     e->status = NGX_HTTP_INTERNAL_SERVER_ERROR;
1052     return;
1053 }
1054
1055 e->quote = code->redirect;
1056
1057 e->pos = e->buf.data;
1058
1059 e->ip += sizeof(ngx_http_script_regex_code_t);
1060 }
1061
1062
1063 void
1064 ngx_http_script_regex_end_code(ngx_http_script_engine_t *e)
1065 {
1066     u_char          *dst, *src;
1067     ngx_http_request_t *r;
1068     ngx_http_script_regex_end_code_t *code;
1069
1070     code = (ngx_http_script_regex_end_code_t *) e->ip;
1071
1072     r = e->request;
1073
1074     e->quote = 0;
1075
1076     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1077                  "http script regex end");
1078
1079     if (code->redirect) {
1080
1081         dst = e->buf.data;
1082         src = e->buf.data;
1083
1084         ngx_unescape_uri(&dst, &src, e->pos - e->buf.data,
1085                         NGX_UNESCAPE_REDIRECT);
1086
1087         if (src < e->pos) {
1088             dst = ngx_movemem(dst, src, e->pos - src);
1089         }
1090
1091         e->pos = dst;
1092
1093         if (code->add_args && r->args.len) {
1094             *e->pos++ = (u_char) (code->args ? '&' : '?');

```

```

1095     e->pos = ngx_copy(e->pos, r->args.data, r->args.len);
1096 }
1097
1098 e->buf.len = e->pos - e->buf.data;
1099
1100 if (e->log || (r->connection->log->log_level & NGX_LOG_DEBUG_HTTP)) {
1101     ngx_log_error(NGX_LOG_NOTICE, r->connection->log, 0,
1102                 "rewritten redirect: \"%V\"", &e->buf);
1103 }
1104
1105 ngx_http_clear_location(r);
1106
1107 r->headers_out.location = ngx_list_push(&r->headers_out.headers);
1108 if (r->headers_out.location == NULL) {
1109     e->ip = ngx_http_script_exit;
1110     e->status = NGX_HTTP_INTERNAL_SERVER_ERROR;
1111     return;
1112 }
1113
1114 r->headers_out.location->hash = 1;
1115 ngx_str_set(&r->headers_out.location->key, "Location");
1116 r->headers_out.location->value = e->buf;
1117
1118 e->ip += sizeof(ngx_http_script_regex_end_code_t);
1119 return;
1120 }
1121
1122 if (e->args) {
1123     e->buf.len = e->args - e->buf.data;
1124
1125     if (code->add_args && r->args.len) {
1126         *e->pos++ = '&';
1127         e->pos = ngx_copy(e->pos, r->args.data, r->args.len);
1128     }
1129
1130     r->args.len = e->pos - e->args;
1131     r->args.data = e->args;
1132
1133     e->args = NULL;
1134 } else {
1135     e->buf.len = e->pos - e->buf.data;
1136
1137     if (!code->add_args) {
1138         r->args.len = 0;
1139     }
1140 }
1141
1142 if (e->log || (r->connection->log->log_level & NGX_LOG_DEBUG_HTTP)) {
1143     ngx_log_error(NGX_LOG_NOTICE, r->connection->log, 0,
1144                 "rewritten data: \"%V\"", args: \"%V\"",
1145                 &e->buf, &r->args);
1146 }
1147
1148 if (code->uri) {
1149     r->uri = e->buf;
1150
1151     if (r->uri.len == 0) {
1152         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1153                     "the rewritten URI has a zero length");
1154         e->ip = ngx_http_script_exit;
1155         e->status = NGX_HTTP_INTERNAL_SERVER_ERROR;
1156         return;
1157     }
1158
1159     ngx_http_set_exten(r);
1160 }
1161
1162 e->ip += sizeof(ngx_http_script_regex_end_code_t);
1163 }
1164
1165
1166
1167 static ngx_int_t
1168 ngx_http_script_add_capture_code(ngx_http_script_compile_t *sc, ngx_uint_t n)
1169 {
1170     ngx_http_script_copy_capture_code_t *code;

```

```

1171 code = ngx_http_script_add_code(*sc->lengths,
1172                               sizeof(ngx_http_script_copy_capture_code_t),
1173                               NULL);
1174
1175 if (code == NULL) {
1176     return NGX_ERROR;
1177 }
1178
1179 code->code = (ngx_http_script_code_pt)
1180             ngx_http_script_copy_capture_len_code;
1181 code->n = 2 * n;
1182
1183
1184 code = ngx_http_script_add_code(*sc->values,
1185                               sizeof(ngx_http_script_copy_capture_code_t),
1186                               &sc->main);
1187
1188 if (code == NULL) {
1189     return NGX_ERROR;
1190 }
1191
1192 code->code = ngx_http_script_copy_capture_code;
1193 code->n = 2 * n;
1194
1195 if (sc->ncaptures < n) {
1196     sc->ncaptures = n;
1197 }
1198
1199 return NGX_OK;
1200 }
1201
1202 size_t
1203 ngx_http_script_copy_capture_len_code(ngx_http_script_engine_t *e)
1204 {
1205     int *cap;
1206     u_char *p;
1207     ngx_uint_t n;
1208     ngx_http_request_t *r;
1209     ngx_http_script_copy_capture_code_t *code;
1210
1211     r = e->request;
1212
1213     code = (ngx_http_script_copy_capture_code_t *) e->ip;
1214
1215     e->ip += sizeof(ngx_http_script_copy_capture_code_t);
1216
1217     n = code->n;
1218
1219     if (n < r->ncaptures) {
1220         cap = r->captures;
1221
1222         if ((e->is_args || e->quote)
1223             && (e->request->quoted_uri || e->request->plus_in_uri))
1224         {
1225             p = r->captures_data;
1226
1227             return cap[n + 1] - cap[n]
1228                    + 2 * ngx_escape_uri(NULL, &p[cap[n]], cap[n + 1] - cap[n],
1229                                       NGX_ESCAPE_ARGS);
1230         } else {
1231             return cap[n + 1] - cap[n];
1232         }
1233     }
1234
1235     return 0;
1236 }
1237
1238
1239 void
1240 ngx_http_script_copy_capture_code(ngx_http_script_engine_t *e)
1241 {
1242     int *cap;
1243     u_char *p, *pos;
1244     ngx_uint_t n;
1245     ngx_http_request_t *r;

```

```

1247     ngx\_http\_script\_copy\_capture\_code\_t *code;
1248
1249     r = e->request;
1250
1251     code = (ngx\_http\_script\_copy\_capture\_code\_t *) e->ip;
1252
1253     e->ip += sizeof(ngx\_http\_script\_copy\_capture\_code\_t);
1254
1255     n = code->n;
1256
1257     pos = e->pos;
1258
1259     if (n < r->ncaptures) {
1260
1261         cap = r->captures;
1262         p = r->captures_data;
1263
1264         if ((e->is_args || e->quote)
1265             && (e->request->quoted_uri || e->request->plus_in_uri))
1266         {
1267             e->pos = (u_char *) ngx\_escape\_uri(pos, &p[cap[n]],
1268                                             cap[n + 1] - cap[n],
1269                                             NGX\_ESCAPE\_ARGS);
1270         } else {
1271             e->pos = ngx\_copy(pos, &p[cap[n]], cap[n + 1] - cap[n]);
1272         }
1273     }
1274
1275     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, e->request->connection->log, 0,
1276                  "http script capture: \"%*s\"", e->pos - pos, pos);
1277 }
1278
1279 #endif
1280
1281
1282 static ngx\_int\_t
1283 ngx\_http\_script\_add\_full\_name\_code(ngx\_http\_script\_compile\_t *sc)
1284 {
1285     ngx\_http\_script\_full\_name\_code\_t *code;
1286
1287     code = ngx\_http\_script\_add\_code(*sc->lengths,
1288                                   sizeof(ngx\_http\_script\_full\_name\_code\_t),
1289                                   NULL);
1290
1291     if (code == NULL) {
1292         return NGX\_ERROR;
1293     }
1294
1295     code->code = (ngx\_http\_script\_code\_pt) ngx\_http\_script\_full\_name\_len\_code;
1296     code->conf_prefix = sc->conf_prefix;
1297
1298     code = ngx\_http\_script\_add\_code(*sc->values,
1299                                   sizeof(ngx\_http\_script\_full\_name\_code\_t),
1300                                   &sc->main);
1301
1302     if (code == NULL) {
1303         return NGX\_ERROR;
1304     }
1305
1306     code->code = ngx\_http\_script\_full\_name\_code;
1307     code->conf_prefix = sc->conf_prefix;
1308
1309     return NGX\_OK;
1310 }
1311
1312 static size\_t
1313 ngx\_http\_script\_full\_name\_len\_code(ngx\_http\_script\_engine\_t *e)
1314 {
1315     ngx\_http\_script\_full\_name\_code\_t *code;
1316
1317     code = (ngx\_http\_script\_full\_name\_code\_t *) e->ip;
1318
1319     e->ip += sizeof(ngx\_http\_script\_full\_name\_code\_t);
1320
1321     return code->conf_prefix ? ngx\_cycle->conf_prefix.len:
1322                               ngx\_cycle->prefix.len;
1323 }

```

```

1323
1324
1325 static void
1326 ngx_http_script_full_name_code(ngx_http_script_engine_t *e)
1327 {
1328     ngx_http_script_full_name_code_t *code;
1329
1330     ngx_str_t value, *prefix;
1331
1332     code = (ngx_http_script_full_name_code_t *) e->ip;
1333
1334     value.data = e->buf.data;
1335     value.len = e->pos - e->buf.data;
1336
1337     prefix = code->conf_prefix ? (ngx_str_t *) &ngx_cycle->conf_prefix:
1338                               (ngx_str_t *) &ngx_cycle->prefix;
1339
1340     if (ngx_get_full_name(e->request->pool, prefix, &value) != NGX_OK) {
1341         e->ip = ngx_http_script_exit;
1342         e->status = NGX_HTTP_INTERNAL_SERVER_ERROR;
1343         return;
1344     }
1345
1346     e->buf = value;
1347
1348     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, e->request->connection->log, 0,
1349                  "http script fullname: \"%V\"", &value);
1350
1351     e->ip += sizeof(ngx_http_script_full_name_code_t);
1352 }
1353
1354
1355 void
1356 ngx_http_script_return_code(ngx_http_script_engine_t *e)
1357 {
1358     ngx_http_script_return_code_t *code;
1359
1360     code = (ngx_http_script_return_code_t *) e->ip;
1361
1362     if (code->status < NGX_HTTP_BAD_REQUEST
1363         || code->text.value.len
1364         || code->text.lengths)
1365     {
1366         e->status = ngx_http_send_response(e->request, code->status, NULL,
1367                                         &code->text);
1368     } else {
1369         e->status = code->status;
1370     }
1371
1372     e->ip = ngx_http_script_exit;
1373 }
1374
1375
1376 void
1377 ngx_http_script_break_code(ngx_http_script_engine_t *e)
1378 {
1379     e->request->uri_changed = 0;
1380
1381     e->ip = ngx_http_script_exit;
1382 }
1383
1384
1385 void
1386 ngx_http_script_if_code(ngx_http_script_engine_t *e)
1387 {
1388     ngx_http_script_if_code_t *code;
1389
1390     code = (ngx_http_script_if_code_t *) e->ip;
1391
1392     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, e->request->connection->log, 0,
1393                  "http script if");
1394
1395     e->sp--;
1396
1397     if (e->sp->len && (e->sp->len != 1 || e->sp->data[0] != '0')) {
1398         if (code->loc_conf) {

```

```

1399         e->request->loc_conf = code->loc_conf;
1400         ngx\_http\_update\_location\_config(e->request);
1401     }
1402
1403     e->ip += sizeof\(ngx\_http\_script\_if\_code\_t\);
1404     return;
1405 }
1406
1407 ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, e->request->connection->log, 0,
1408     "http script if: false");
1409
1410 e->ip += code->next;
1411 }
1412
1413
1414 void
1415 ngx\_http\_script\_equal\_code(ngx\_http\_script\_engine\_t *e)
1416 {
1417     ngx\_http\_variable\_value\_t *val, *res;
1418
1419     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, e->request->connection->log, 0,
1420         "http script equal");
1421
1422     e->sp--;
1423     val = e->sp;
1424     res = e->sp - 1;
1425
1426     e->ip += sizeof\(uintptr\_t\);
1427
1428     if (val->len == res->len
1429         && ngx\_strncmp(val->data, res->data, res->len) == 0)
1430     {
1431         *res = ngx\_http\_variable\_true\_value;
1432         return;
1433     }
1434
1435     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, e->request->connection->log, 0,
1436         "http script equal: no");
1437
1438     *res = ngx\_http\_variable\_null\_value;
1439 }
1440
1441
1442 void
1443 ngx\_http\_script\_not\_equal\_code(ngx\_http\_script\_engine\_t *e)
1444 {
1445     ngx\_http\_variable\_value\_t *val, *res;
1446
1447     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, e->request->connection->log, 0,
1448         "http script not equal");
1449
1450     e->sp--;
1451     val = e->sp;
1452     res = e->sp - 1;
1453
1454     e->ip += sizeof\(uintptr\_t\);
1455
1456     if (val->len == res->len
1457         && ngx\_strncmp(val->data, res->data, res->len) == 0)
1458     {
1459         ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, e->request->connection->log, 0,
1460             "http script not equal: no");
1461
1462         *res = ngx\_http\_variable\_null\_value;
1463         return;
1464     }
1465
1466     *res = ngx\_http\_variable\_true\_value;
1467 }
1468
1469
1470 void
1471 ngx\_http\_script\_file\_code(ngx\_http\_script\_engine\_t *e)
1472 {
1473     ngx\_str\_t                path;
1474     ngx\_http\_request\_t      *r;

```

```

1475 ngx\_open\_file\_info\_t of;
1476 ngx\_http\_core\_loc\_conf\_t *clcf;
1477 ngx\_http\_variable\_value\_t *value;
1478 ngx\_http\_script\_file\_code\_t *code;
1479
1480 value = e->sp - 1;
1481
1482 code = (ngx\_http\_script\_file\_code\_t *) e->ip;
1483 e->ip += sizeof(ngx\_http\_script\_file\_code\_t);
1484
1485 path.len = value->len - 1;
1486 path.data = value->data;
1487
1488 r = e->request;
1489
1490 ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
1491 "http script file op %p \"%V\"", code->op, &path);
1492
1493 clcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_core\_module);
1494
1495 ngx\_memzero(&of, sizeof(ngx\_open\_file\_info\_t));
1496
1497 of.read_ahead = clcf->read_ahead;
1498 of.directio = clcf->directio;
1499 of.valid = clcf->open_file_cache_valid;
1500 of.min_uses = clcf->open_file_cache_min_uses;
1501 of.test_only = 1;
1502 of.errors = clcf->open_file_cache_errors;
1503 of.events = clcf->open_file_cache_events;
1504
1505 if (ngx\_http\_set\_disable\_symlinks(r, clcf, &path, &of) != NGX\_OK) {
1506     e->ip = ngx\_http\_script\_exit;
1507     e->status = NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
1508     return;
1509 }
1510
1511 if (ngx\_open\_cached\_file(clcf->open_file_cache, &path, &of, r->pool)
1512     != NGX\_OK)
1513 {
1514     if (of.err != NGX\_ENOENT
1515         && of.err != NGX\_ENOTDIR
1516         && of.err != NGX\_ENAMETOOLONG)
1517     {
1518         ngx\_log\_error(NGX\_LOG\_CRIT, r->connection->log, of.err,
1519 "%s \"%s\" failed", of.failed, value->data);
1520     }
1521
1522     switch (code->op) {
1523
1524     case ngx_http_script_file_plain:
1525     case ngx_http_script_file_dir:
1526     case ngx_http_script_file_exists:
1527     case ngx_http_script_file_exec:
1528         goto false_value;
1529
1530     case ngx_http_script_file_not_plain:
1531     case ngx_http_script_file_not_dir:
1532     case ngx_http_script_file_not_exists:
1533     case ngx_http_script_file_not_exec:
1534         goto true_value;
1535     }
1536
1537     goto false_value;
1538 }
1539
1540 switch (code->op) {
1541 case ngx_http_script_file_plain:
1542     if (of.is_file) {
1543         goto true_value;
1544     }
1545     goto false_value;
1546
1547 case ngx_http_script_file_not_plain:
1548     if (of.is_file) {
1549         goto false_value;
1550     }

```



```

1551     goto true_value;
1552
1553 case ngx_http_script_file_dir:
1554     if (of.is_dir) {
1555         goto true_value;
1556     }
1557     goto false_value;
1558
1559 case ngx_http_script_file_not_dir:
1560     if (of.is_dir) {
1561         goto false_value;
1562     }
1563     goto true_value;
1564
1565 case ngx_http_script_file_exists:
1566     if (of.is_file || of.is_dir || of.is_link) {
1567         goto true_value;
1568     }
1569     goto false_value;
1570
1571 case ngx_http_script_file_not_exists:
1572     if (of.is_file || of.is_dir || of.is_link) {
1573         goto false_value;
1574     }
1575     goto true_value;
1576
1577 case ngx_http_script_file_exec:
1578     if (of.is_exec) {
1579         goto true_value;
1580     }
1581     goto false_value;
1582
1583 case ngx_http_script_file_not_exec:
1584     if (of.is_exec) {
1585         goto false_value;
1586     }
1587     goto true_value;
1588 }
1589
1590 false_value:
1591
1592     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1593                 "http script file op false");
1594
1595     *value = ngx_http_variable_null_value;
1596     return;
1597
1598 true_value:
1599
1600     *value = ngx_http_variable_true_value;
1601     return;
1602 }
1603
1604
1605 void
1606 ngx_http_script_complex_value_code(ngx_http_script_engine_t *e)
1607 {
1608     size_t                len;
1609     ngx_http_script_engine_t  le;
1610     ngx_http_script_len_code_pt lcode;
1611     ngx_http_script_complex_value_code_t *code;
1612
1613     code = (ngx_http_script_complex_value_code_t *) e->ip;
1614
1615     e->ip += sizeof(ngx_http_script_complex_value_code_t);
1616
1617     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, e->request->connection->log, 0,
1618                 "http script complex value");
1619
1620     ngx_memzero(&le, sizeof(ngx_http_script_engine_t));
1621
1622     le.ip = code->lengths->elts;
1623     le.line = e->line;
1624     le.request = e->request;
1625     le.quote = e->quote;
1626

```

```

1627     for (len = 0; *(uintptr_t *) le.ip; len += lcode(&le)) {
1628         lcode = *(ngx_http_script_len_code_pt *) le.ip;
1629     }
1630
1631     e->buf.len = len;
1632     e->buf.data = ngx_pnalloc(e->request->pool, len);
1633     if (e->buf.data == NULL) {
1634         e->ip = ngx_http_script_exit;
1635         e->status = NGX_HTTP_INTERNAL_SERVER_ERROR;
1636         return;
1637     }
1638
1639     e->pos = e->buf.data;
1640
1641     e->sp->len = e->buf.len;
1642     e->sp->data = e->buf.data;
1643     e->sp++;
1644 }
1645
1646
1647 void
1648 ngx_http_script_value_code(ngx_http_script_engine_t *e)
1649 {
1650     ngx_http_script_value_code_t *code;
1651
1652     code = (ngx_http_script_value_code_t *) e->ip;
1653
1654     e->ip += sizeof(ngx_http_script_value_code_t);
1655
1656     e->sp->len = code->text_len;
1657     e->sp->data = (u_char *) code->text_data;
1658
1659     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, e->request->connection->log, 0,
1660                  "http script value: \"%v\"", e->sp);
1661
1662     e->sp++;
1663 }
1664
1665
1666 void
1667 ngx_http_script_set_var_code(ngx_http_script_engine_t *e)
1668 {
1669     ngx_http_request_t *r;
1670     ngx_http_script_var_code_t *code;
1671
1672     code = (ngx_http_script_var_code_t *) e->ip;
1673
1674     e->ip += sizeof(ngx_http_script_var_code_t);
1675
1676     r = e->request;
1677
1678     e->sp--;
1679
1680     r->variables[code->index].len = e->sp->len;
1681     r->variables[code->index].valid = 1;
1682     r->variables[code->index].no_cacheable = 0;
1683     r->variables[code->index].not_found = 0;
1684     r->variables[code->index].data = e->sp->data;
1685
1686     #if (NGX_DEBUG)
1687     {
1688         ngx_http_variable_t *v;
1689         ngx_http_core_main_conf_t *cmcf;
1690
1691         cmcf = ngx_http_get_module_main_conf(r, ngx_http_core_module);
1692
1693         v = cmcf->variables.elts;
1694
1695         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, e->request->connection->log, 0,
1696                      "http script set %v", &v[code->index].name);
1697     }
1698     #endif
1699 }
1700
1701 void

```

```

1703 ngx_http_script_var_set_handler_code(ngx_http_script_engine_t *e)
1704 {
1705     ngx_http_script_var_handler_code_t *code;
1706
1707     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, e->request->connection->log, 0,
1708         "http script set var handler");
1709
1710     code = (ngx_http_script_var_handler_code_t *) e->ip;
1711
1712     e->ip += sizeof(ngx_http_script_var_handler_code_t);
1713
1714     e->sp--;
1715
1716     code->handler(e->request, e->sp, code->data);
1717 }
1718
1719
1720 void
1721 ngx_http_script_var_code(ngx_http_script_engine_t *e)
1722 {
1723     ngx_http_variable_value_t *value;
1724     ngx_http_script_var_code_t *code;
1725
1726     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, e->request->connection->log, 0,
1727         "http script var");
1728
1729     code = (ngx_http_script_var_code_t *) e->ip;
1730
1731     e->ip += sizeof(ngx_http_script_var_code_t);
1732
1733     value = ngx_http_get_flushed_variable(e->request, code->index);
1734
1735     if (value && !value->not_found) {
1736         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, e->request->connection->log, 0,
1737             "http script var: \"%v\"", value);
1738
1739         *e->sp = *value;
1740         e->sp++;
1741
1742         return;
1743     }
1744
1745     *e->sp = ngx_http_variable_null_value;
1746     e->sp++;
1747 }
1748
1749
1750 void
1751 ngx_http_script_nop_code(ngx_http_script_engine_t *e)
1752 {
1753     e->ip += sizeof(uintptr_t);
1754 }

```

[One Level Up](#)

[Top Level](#)

## src/http/modules/nginx\_http\_userid\_filter\_module.c - nginx-1.7.10

### Global variables defined

- [expires](#)
- [ngx\\_http\\_next\\_header\\_filter](#)
- [ngx\\_http\\_userid\\_commands](#)
- [ngx\\_http\\_userid\\_domain\\_p](#)
- [ngx\\_http\\_userid\\_filter\\_module](#)
- [ngx\\_http\\_userid\\_filter\\_module\\_ctx](#)
- [ngx\\_http\\_userid\\_got](#)
- [ngx\\_http\\_userid\\_p3p\\_p](#)
- [ngx\\_http\\_userid\\_path\\_p](#)
- [ngx\\_http\\_userid\\_reset](#)
- [ngx\\_http\\_userid\\_reset\\_index](#)
- [ngx\\_http\\_userid\\_set](#)
- [ngx\\_http\\_userid\\_state](#)
- [sequencer\\_v1](#)
- [sequencer\\_v2](#)
- [start\\_value](#)

### Data types defined

- [ngx\\_http\\_userid\\_conf\\_t](#)
- [ngx\\_http\\_userid\\_ctx\\_t](#)

### Functions defined

- [ngx\\_http\\_userid\\_add\\_variables](#)
- [ngx\\_http\\_userid\\_create\\_conf](#)
- [ngx\\_http\\_userid\\_create\\_uid](#)
- [ngx\\_http\\_userid\\_domain](#)
- [ngx\\_http\\_userid\\_expires](#)
- [ngx\\_http\\_userid\\_filter](#)
- [ngx\\_http\\_userid\\_get\\_uid](#)
- [ngx\\_http\\_userid\\_got\\_variable](#)
- [ngx\\_http\\_userid\\_init](#)

- [ngx\\_http\\_userid\\_init\\_worker](#)
- [ngx\\_http\\_userid\\_mark](#)
- [ngx\\_http\\_userid\\_merge\\_conf](#)
- [ngx\\_http\\_userid\\_p3p](#)
- [ngx\\_http\\_userid\\_path](#)
- [ngx\\_http\\_userid\\_reset\\_variable](#)
- [ngx\\_http\\_userid\\_set\\_uid](#)
- [ngx\\_http\\_userid\\_set\\_variable](#)
- [ngx\\_http\\_userid\\_variable](#)

## Macros defined

- [NGX\\_HTTP\\_USERID\\_LOG](#)
- [NGX\\_HTTP\\_USERID\\_MAX\\_EXPIRES](#)
- [NGX\\_HTTP\\_USERID\\_OFF](#)
- [NGX\\_HTTP\\_USERID\\_ON](#)
- [NGX\\_HTTP\\_USERID\\_V1](#)

## Source code

```

1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 #define NGX_HTTP_USERID_OFF    0
14 #define NGX_HTTP_USERID_LOG    1
15 #define NGX_HTTP_USERID_V1    2
16 #define NGX_HTTP_USERID_ON    3
17
18 /* 31 Dec 2037 23:55:55 GMT */
19 #define NGX_HTTP_USERID_MAX_EXPIRES  2145916555
20
21
22 typedef struct {
23     ngx_uint_t  enable;
24
25     ngx_int_t   service;
26
27     ngx_str_t   name;
28     ngx_str_t   domain;
29     ngx_str_t   path;
30     ngx_str_t   p3p;
31
32     time_t      expires;
33
34     u_char      mark;
35 } ngx_http_userid_conf_t;
36
37

```

```

38 typedef struct {
39     uint32_t    uid_got[4];
40     uint32_t    uid_set[4];
41     ngx_str_t   cookie;
42     ngx_uint_t  reset;
43 } ngx_http_userid_ctx_t;
44
45
46 static ngx_http_userid_ctx_t *ngx_http_userid_get_uid(ngx_http_request_t *r,
47     ngx_http_userid_conf_t *conf);
48 static ngx_int_t ngx_http_userid_variable(ngx_http_request_t *r,
49     ngx_http_variable_value_t *v, ngx_str_t *name, uint32_t *uid);
50 static ngx_int_t ngx_http_userid_set_uid(ngx_http_request_t *r,
51     ngx_http_userid_ctx_t *ctx, ngx_http_userid_conf_t *conf);
52 static ngx_int_t ngx_http_userid_create_uid(ngx_http_request_t *r,
53     ngx_http_userid_ctx_t *ctx, ngx_http_userid_conf_t *conf);
54
55 static ngx_int_t ngx_http_userid_add_variables(ngx_conf_t *cf);
56 static ngx_int_t ngx_http_userid_init(ngx_conf_t *cf);
57 static void *ngx_http_userid_create_conf(ngx_conf_t *cf);
58 static char *ngx_http_userid_merge_conf(ngx_conf_t *cf, void *parent,
59     void *child);
60 static char *ngx_http_userid_domain(ngx_conf_t *cf, void *post, void *data);
61 static char *ngx_http_userid_path(ngx_conf_t *cf, void *post, void *data);
62 static char *ngx_http_userid_expires(ngx_conf_t *cf, ngx_command_t *cmd,
63     void *conf);
64 static char *ngx_http_userid_p3p(ngx_conf_t *cf, void *post, void *data);
65 static char *ngx_http_userid_mark(ngx_conf_t *cf, ngx_command_t *cmd,
66     void *conf);
67 static ngx_int_t ngx_http_userid_init_worker(ngx_cycle_t *cycle);
68
69
70
71 static uint32_t  start_value;
72 static uint32_t  sequencer_v1 = 1;
73 static uint32_t  sequencer_v2 = 0x03030302;
74
75
76 static u_char expires[] = "; expires=Thu, 31-Dec-37 23:55:55 GMT";
77
78
79 static ngx_http_output_header_filter_pt  ngx_http_next_header_filter;
80
81
82 static ngx_conf_enum_t  ngx_http_userid_state[] = {
83     { ngx_string("off"),  NGX_HTTP_USERID_OFF },
84     { ngx_string("log"),  NGX_HTTP_USERID_LOG },
85     { ngx_string("v1"),   NGX_HTTP_USERID_V1 },
86     { ngx_string("on"),   NGX_HTTP_USERID_ON },
87     { ngx_null_string, 0 }
88 };
89
90
91 static ngx_conf_post_handler_pt  ngx_http_userid_domain_p =
92     ngx_http_userid_domain;
93 static ngx_conf_post_handler_pt  ngx_http_userid_path_p = ngx_http_userid_path;
94 static ngx_conf_post_handler_pt  ngx_http_userid_p3p_p = ngx_http_userid_p3p;
95
96
97 static ngx_command_t  ngx_http_userid_commands[] = {
98
99     { ngx_string("userid"),
100     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
101     ngx_conf_set_enum_slot,
102     NGX_HTTP_LOC_CONF_OFFSET,
103     offsetof(ngx_http_userid_conf_t, enable),
104     ngx_http_userid_state },
105
106     { ngx_string("userid_service"),
107     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
108     ngx_conf_set_num_slot,
109     NGX_HTTP_LOC_CONF_OFFSET,
110     offsetof(ngx_http_userid_conf_t, service),
111     NULL },
112
113     { ngx_string("userid_name"),

```

```

114     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF TAKE1,
115     ngx\_conf\_set\_str\_slot,
116     NGX\_HTTP\_LOC\_CONF\_OFFSET,
117     offsetof(ngx\_http\_userid\_conf\_t, name),
118     NULL },
119
120     { ngx\_string\("userid\_domain"\),
121     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF TAKE1,
122     ngx\_conf\_set\_str\_slot,
123     NGX\_HTTP\_LOC\_CONF\_OFFSET,
124     offsetof(ngx\_http\_userid\_conf\_t, domain),
125     &ngx\_http\_userid\_domain\_p },
126
127     { ngx\_string\("userid\_path"\),
128     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF TAKE1,
129     ngx\_conf\_set\_str\_slot,
130     NGX\_HTTP\_LOC\_CONF\_OFFSET,
131     offsetof(ngx\_http\_userid\_conf\_t, path),
132     &ngx\_http\_userid\_path\_p },
133
134     { ngx\_string\("userid\_expires"\),
135     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF TAKE1,
136     ngx\_http\_userid\_expires,
137     NGX\_HTTP\_LOC\_CONF\_OFFSET,
138     0,
139     NULL },
140
141     { ngx\_string\("userid\_p3p"\),
142     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF TAKE1,
143     ngx\_conf\_set\_str\_slot,
144     NGX\_HTTP\_LOC\_CONF\_OFFSET,
145     offsetof(ngx\_http\_userid\_conf\_t, p3p),
146     &ngx\_http\_userid\_p3p\_p },
147
148     { ngx\_string\("userid\_mark"\),
149     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF TAKE1,
150     ngx\_http\_userid\_mark,
151     NGX\_HTTP\_LOC\_CONF\_OFFSET,
152     0,
153     NULL },
154
155     ngx\_null\_command
156 };
157
158
159 static ngx\_http\_module\_t ngx\_http\_userid\_filter\_module\_ctx = {
160     ngx\_http\_userid\_add\_variables,          /* preconfiguration */
161     ngx\_http\_userid\_init,                  /* postconfiguration */
162
163     NULL,                                    /* create main configuration */
164     NULL,                                    /* init main configuration */
165
166     NULL,                                    /* create server configuration */
167     NULL,                                    /* merge server configuration */
168
169     ngx\_http\_userid\_create\_conf,           /* create location configuration */
170     ngx\_http\_userid\_merge\_conf            /* merge location configuration */
171 };
172
173
174 ngx\_module\_t ngx\_http\_userid\_filter\_module = {
175     NGX\_MODULE\_V1,
176     &ngx\_http\_userid\_filter\_module\_ctx,    /* module context */
177     ngx\_http\_userid\_commands,          /* module directives */
178     NGX\_HTTP\_MODULE,                    /* module type */
179     NULL,                                  /* init master */
180     NULL,                                  /* init module */
181     ngx\_http\_userid\_init\_worker,        /* init process */
182     NULL,                                  /* init thread */
183     NULL,                                  /* exit thread */
184     NULL,                                  /* exit process */
185     NULL,                                  /* exit master */
186     NGX\_MODULE\_V1\_PADDING
187 };
188
189

```

```

190 static ngx_str_t ngx_http_userid_got = ngx_string("uid_got");
191 static ngx_str_t ngx_http_userid_set = ngx_string("uid_set");
192 static ngx_str_t ngx_http_userid_reset = ngx_string("uid_reset");
193 static ngx_uint_t ngx_http_userid_reset_index;
194
195
196 static ngx_int_t
197 ngx_http_userid_filter(ngx_http_request_t *r)
198 {
199     ngx_http_userid_ctx_t *ctx;
200     ngx_http_userid_conf_t *conf;
201
202     if (r != r->main) {
203         return ngx_http_next_header_filter(r);
204     }
205
206     conf = ngx_http_get_module_loc_conf(r, ngx_http_userid_filter_module);
207
208     if (conf->enable < NGX_HTTP_USERID_V1) {
209         return ngx_http_next_header_filter(r);
210     }
211
212     ctx = ngx_http_userid_get_uid(r, conf);
213
214     if (ctx == NULL) {
215         return NGX_ERROR;
216     }
217
218     if (ngx_http_userid_set_uid(r, ctx, conf) == NGX_OK) {
219         return ngx_http_next_header_filter(r);
220     }
221
222     return NGX_ERROR;
223 }
224
225
226 static ngx_int_t
227 ngx_http_userid_got_variable(ngx_http_request_t *r,
228     ngx_http_variable_value_t *v, uintptr_t data)
229 {
230     ngx_http_userid_ctx_t *ctx;
231     ngx_http_userid_conf_t *conf;
232
233     conf = ngx_http_get_module_loc_conf(r->main, ngx_http_userid_filter_module);
234
235     if (conf->enable == NGX_HTTP_USERID_OFF) {
236         v->not_found = 1;
237         return NGX_OK;
238     }
239
240     ctx = ngx_http_userid_get_uid(r->main, conf);
241
242     if (ctx == NULL) {
243         return NGX_ERROR;
244     }
245
246     if (ctx->uid_got[3] != 0) {
247         return ngx_http_userid_variable(r->main, v, &conf->name, ctx->uid_got);
248     }
249
250     v->not_found = 1;
251
252     return NGX_OK;
253 }
254
255
256 static ngx_int_t
257 ngx_http_userid_set_variable(ngx_http_request_t *r,
258     ngx_http_variable_value_t *v, uintptr_t data)
259 {
260     ngx_http_userid_ctx_t *ctx;
261     ngx_http_userid_conf_t *conf;
262
263     conf = ngx_http_get_module_loc_conf(r->main, ngx_http_userid_filter_module);
264
265     if (conf->enable < NGX_HTTP_USERID_V1) {

```



```

266     v->not_found = 1;
267     return NGX_OK;
268 }
269
270 ctx = ngx_http_userid_get_uid(r->main, conf);
271
272 if (ctx == NULL) {
273     return NGX_ERROR;
274 }
275
276 if (ngx_http_userid_create_uid(r->main, ctx, conf) != NGX_OK) {
277     return NGX_ERROR;
278 }
279
280 if (ctx->uid_set[3] == 0) {
281     v->not_found = 1;
282     return NGX_OK;
283 }
284
285 return ngx_http_userid_variable(r->main, v, &conf->name, ctx->uid_set);
286 }
287
288
289 static ngx_http_userid_ctx_t *
290 ngx_http_userid_get_uid(ngx_http_request_t *r, ngx_http_userid_conf_t *conf)
291 {
292     ngx_int_t          n;
293     ngx_str_t          src, dst;
294     ngx_table_elt_t    **cookies;
295     ngx_http_userid_ctx_t *ctx;
296
297     ctx = ngx_http_get_module_ctx(r, ngx_http_userid_filter_module);
298
299     if (ctx) {
300         return ctx;
301     }
302
303     if (ctx == NULL) {
304         ctx = ngx_palloc(r->pool, sizeof(ngx_http_userid_ctx_t));
305         if (ctx == NULL) {
306             return NULL;
307         }
308
309         ngx_http_set_ctx(r, ctx, ngx_http_userid_filter_module);
310     }
311
312     n = ngx_http_parse_multi_header_lines(&r->headers_in.cookies, &conf->name,
313                                         &ctx->cookie);
314     if (n == NGX_DECLINED) {
315         return ctx;
316     }
317
318     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
319                  "uid cookie: \"%V\"", &ctx->cookie);
320
321     if (ctx->cookie.len < 22) {
322         cookies = r->headers_in.cookies.elts;
323         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
324                      "client sent too short userid cookie \"%V\"",
325                      &cookies[n]->value);
326         return ctx;
327     }
328
329     src = ctx->cookie;
330
331     /*
332     * we have to limit the encoded string to 22 characters because
333     * 1) cookie may be marked by "userid_mark",
334     * 2) and there are already the millions cookies with a garbage
335     *    instead of the correct base64 trail "=="
336     */
337
338     src.len = 22;
339
340     dst.data = (u_char *) ctx->uid_got;
341

```

```

342 if (ngx_decode_base64(&dst, &src) == NGX_ERROR) {
343     cookies = r->headers_in.cookies.elts;
344     ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
345         "client sent invalid userid cookie \"%V\"",
346         &cookies[n]->value);
347     return ctx;
348 }
349
350 ngx_log_debug4(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
351     "uid: %08XD%08XD%08XD%08XD",
352     ctx->uid_got[0], ctx->uid_got[1],
353     ctx->uid_got[2], ctx->uid_got[3]);
354
355 return ctx;
356 }
357
358
359 static ngx_int_t
360 ngx_http_userid_set_uid(ngx_http_request_t *r, ngx_http_userid_ctx_t *ctx,
361     ngx_http_userid_conf_t *conf)
362 {
363     u_char      *cookie, *p;
364     size_t      len;
365     ngx_str_t    src, dst;
366     ngx_table_elt_t *set_cookie, *p3p;
367
368     if (ngx_http_userid_create_uid(r, ctx, conf) != NGX_OK) {
369         return NGX_ERROR;
370     }
371
372     if (ctx->uid_set[3] == 0) {
373         return NGX_OK;
374     }
375
376     len = conf->name.len + 1 + ngx_base64_encoded_length(16) + conf->path.len;
377
378     if (conf->expires) {
379         len += sizeof(expires) - 1 + 2;
380     }
381
382     if (conf->domain.len) {
383         len += conf->domain.len;
384     }
385
386     cookie = ngx_pnalloc(r->pool, len);
387     if (cookie == NULL) {
388         return NGX_ERROR;
389     }
390
391     p = ngx_copy(cookie, conf->name.data, conf->name.len);
392     *p++ = '=';
393
394     if (ctx->uid_got[3] == 0 || ctx->reset) {
395         src.len = 16;
396         src.data = (u_char *) ctx->uid_set;
397         dst.data = p;
398
399         ngx_encode_base64(&dst, &src);
400
401         p += dst.len;
402
403         if (conf->mark) {
404             *(p - 2) = conf->mark;
405         }
406
407     } else {
408         p = ngx_cpymem(p, ctx->cookie.data, 22);
409         *p++ = conf->mark;
410         *p++ = '=';
411     }
412
413     if (conf->expires == NGX_HTTP_USERID_MAX_EXPIRES) {
414         p = ngx_cpymem(p, expires, sizeof(expires) - 1);
415
416     } else if (conf->expires) {
417         p = ngx_cpymem(p, expires, sizeof("; expires=") - 1);

```

```

418     p = ngx_http_cookie_time(p, ngx_time() + conf->expires);
419 }
420
421 p = ngx_copy(p, conf->domain.data, conf->domain.len);
422
423 p = ngx_copy(p, conf->path.data, conf->path.len);
424
425 set_cookie = ngx_list_push(&r->headers_out.headers);
426 if (set_cookie == NULL) {
427     return NGX_ERROR;
428 }
429
430 set_cookie->hash = 1;
431 ngx_str_set(&set_cookie->key, "Set-Cookie");
432 set_cookie->value.len = p - cookie;
433 set_cookie->value.data = cookie;
434
435 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
436               "uid cookie: \"%V\"", &set_cookie->value);
437
438 if (conf->p3p.len == 0) {
439     return NGX_OK;
440 }
441
442 p3p = ngx_list_push(&r->headers_out.headers);
443 if (p3p == NULL) {
444     return NGX_ERROR;
445 }
446
447 p3p->hash = 1;
448 ngx_str_set(&p3p->key, "P3P");
449 p3p->value = conf->p3p;
450
451 return NGX_OK;
452 }
453
454
455 static ngx_int_t
456 ngx_http_userid_create_uid(ngx_http_request_t *r, ngx_http_userid_ctx_t *ctx,
457                            ngx_http_userid_conf_t *conf)
458 {
459     ngx_connection_t      *c;
460     struct sockaddr_in     *sin;
461     ngx_http_variable_value_t *vv;
462     #if (NGX_HAVE_INET6)
463     u_char                 *p;
464     struct sockaddr_in6    *sin6;
465     #endif
466
467     if (ctx->uid_set[3] != 0) {
468         return NGX_OK;
469     }
470
471     if (ctx->uid_got[3] != 0) {
472
473         vv = ngx_http_get_indexed_variable(r, ngx_http_userid_reset_index);
474
475         if (vv->len == 0 || (vv->len == 1 && vv->data[0] == '0')) {
476
477             if (conf->mark == '\0'
478                 || (ctx->cookie.len > 23
479                     && ctx->cookie.data[22] == conf->mark
480                     && ctx->cookie.data[23] == '='))
481             {
482                 return NGX_OK;
483             }
484
485             ctx->uid_set[0] = ctx->uid_got[0];
486             ctx->uid_set[1] = ctx->uid_got[1];
487             ctx->uid_set[2] = ctx->uid_got[2];
488             ctx->uid_set[3] = ctx->uid_got[3];
489
490             return NGX_OK;
491         }
492     } else {
493         ctx->reset = 1;

```

```

494     if (vv->len == 3 && ngx_strcmp(vv->data, "log", 3) == 0) {
495         ngx_log_error(NGX_LOG_NOTICE, r->connection->log, 0,
496             "userid cookie \"%V=%08XD%08XD%08XD%08XD\" was reset",
497             &conf->name, ctx->uid_got[0], ctx->uid_got[1],
498             ctx->uid_got[2], ctx->uid_got[3]);
499     }
500 }
501 }
502 }
503
504 /*
505  * TODO: in the threaded mode the sequencers should be in TLS and their
506  * ranges should be divided between threads
507  */
508
509 if (conf->enable == NGX_HTTP_USERID_V1) {
510     if (conf->service == NGX_CONF_UNSET) {
511         ctx->uid_set[0] = 0;
512     } else {
513         ctx->uid_set[0] = conf->service;
514     }
515     ctx->uid_set[1] = (uint32_t) ngx_time();
516     ctx->uid_set[2] = start_value;
517     ctx->uid_set[3] = sequencer_v1;
518     sequencer_v1 += 0x100;
519 } else {
520     if (conf->service == NGX_CONF_UNSET) {
521
522         c = r->connection;
523
524         if (ngx_connection_local_sockaddr(c, NULL, 0) != NGX_OK) {
525             return NGX_ERROR;
526         }
527
528         switch (c->local_sockaddr->sa_family) {
529
530             #if (NGX_HAVE_INET6)
531             case AF_INET6:
532                 sin6 = (struct sockaddr_in6 *) c->local_sockaddr;
533
534                 p = (u_char *) &ctx->uid_set[0];
535
536                 *p++ = sin6->sin6_addr.s6_addr[12];
537                 *p++ = sin6->sin6_addr.s6_addr[13];
538                 *p++ = sin6->sin6_addr.s6_addr[14];
539                 *p = sin6->sin6_addr.s6_addr[15];
540
541                 break;
542             #endif
543             default: /* AF_INET */
544                 sin = (struct sockaddr_in *) c->local_sockaddr;
545                 ctx->uid_set[0] = sin->sin_addr.s_addr;
546                 break;
547         }
548     } else {
549         ctx->uid_set[0] = htonl(conf->service);
550     }
551
552     ctx->uid_set[1] = htonl((uint32_t) ngx_time());
553     ctx->uid_set[2] = htonl(start_value);
554     ctx->uid_set[3] = htonl(sequencer_v2);
555     sequencer_v2 += 0x100;
556     if (sequencer_v2 < 0x03030302) {
557         sequencer_v2 = 0x03030302;
558     }
559 }
560
561 return NGX_OK;
562 }
563
564 static ngx_int_t
565 ngx_http_userid_variable(ngx_http_request_t *r, ngx_http_variable_value_t *v,
566     ngx_str_t *name, uint32_t *uid)

```

```

570 {
571     v->len = name->len + sizeof("=00001111222233334444555566667777") - 1;
572     v->data = ngx_pnalloc(r->pool, v->len);
573     if (v->data == NULL) {
574         return NGX_ERROR;
575     }
576
577     v->valid = 1;
578     v->no_cacheable = 0;
579     v->not_found = 0;
580
581     ngx_sprintf(v->data, "%V=%08XD%08XD%08XD%08XD",
582               name, uid[0], uid[1], uid[2], uid[3]);
583
584     return NGX_OK;
585 }
586
587
588 static ngx_int_t
589 ngx_http_userid_reset_variable(ngx_http_request_t *r,
590                               ngx_http_variable_value_t *v, uintptr_t data)
591 {
592     *v = ngx_http_variable_null_value;
593
594     return NGX_OK;
595 }
596
597
598 static ngx_int_t
599 ngx_http_userid_add_variables(ngx_conf_t *cf)
600 {
601     ngx_int_t      n;
602     ngx_http_variable_t *var;
603
604     var = ngx_http_add_variable(cf, &ngx_http_userid_get, 0);
605     if (var == NULL) {
606         return NGX_ERROR;
607     }
608
609     var->get_handler = ngx_http_userid_get_variable;
610
611     var = ngx_http_add_variable(cf, &ngx_http_userid_set, 0);
612     if (var == NULL) {
613         return NGX_ERROR;
614     }
615
616     var->get_handler = ngx_http_userid_set_variable;
617
618     var = ngx_http_add_variable(cf, &ngx_http_userid_reset,
619                               NGX_HTTP_VAR_CHANGEABLE);
620     if (var == NULL) {
621         return NGX_ERROR;
622     }
623
624     var->get_handler = ngx_http_userid_reset_variable;
625
626     n = ngx_http_get_variable_index(cf, &ngx_http_userid_reset);
627     if (n == NGX_ERROR) {
628         return NGX_ERROR;
629     }
630
631     ngx_http_userid_reset_index = n;
632
633     return NGX_OK;
634 }
635
636
637 static void *
638 ngx_http_userid_create_conf(ngx_conf_t *cf)
639 {
640     ngx_http_userid_conf_t *conf;
641
642     conf = ngx_pcalloc(cf->pool, sizeof(ngx_http_userid_conf_t));
643     if (conf == NULL) {
644         return NULL;
645     }

```

```

646 /*
647  * set by ngx_palloc():
648  *
649  *     conf->name = { 0, NULL };
650  *     conf->domain = { 0, NULL };
651  *     conf->path = { 0, NULL };
652  *     conf->p3p = { 0, NULL };
653  */
654
655 conf->enable = NGX\_CONF\_UNSET\_UINT;
656 conf->service = NGX\_CONF\_UNSET;
657 conf->expires = NGX\_CONF\_UNSET;
658 conf->mark = (u_char) '\xFF';
659
660
661 return conf;
662 }
663
664
665 static char *
666 ngx_http_userid_merge_conf(ngx\_conf\_t *cf, void *parent, void *child)
667 {
668     ngx\_http\_userid\_conf\_t *prev = parent;
669     ngx\_http\_userid\_conf\_t *conf = child;
670
671     ngx\_conf\_merge\_uint\_value(conf->enable, prev->enable,
672                               NGX\_HTTP\_USERID\_OFF);
673
674     ngx\_conf\_merge\_str\_value(conf->name, prev->name, "uid");
675     ngx\_conf\_merge\_str\_value(conf->domain, prev->domain, "");
676     ngx\_conf\_merge\_str\_value(conf->path, prev->path, "; path=/");
677     ngx\_conf\_merge\_str\_value(conf->p3p, prev->p3p, "");
678
679     ngx\_conf\_merge\_value(conf->service, prev->service, NGX\_CONF\_UNSET);
680     ngx\_conf\_merge\_sec\_value(conf->expires, prev->expires, 0);
681
682     if (conf->mark == (u_char) '\xFF') {
683         if (prev->mark == (u_char) '\xFF') {
684             conf->mark = '\0';
685         } else {
686             conf->mark = prev->mark;
687         }
688     }
689
690     return NGX\_CONF\_OK;
691 }
692
693
694 static ngx\_int\_t
695 ngx_http_userid_init(ngx\_conf\_t *cf)
696 {
697     ngx\_http\_next\_header\_filter = ngx\_http\_top\_header\_filter;
698     ngx\_http\_top\_header\_filter = ngx\_http\_userid\_filter;
699
700     return NGX\_OK;
701 }
702
703
704 static char *
705 ngx_http_userid_domain(ngx\_conf\_t *cf, void *post, void *data)
706 {
707     ngx\_str\_t *domain = data;
708
709     u_char *p, *new;
710
711     if (ngx\_strcmp(domain->data, "none") == 0) {
712         ngx\_str\_set(domain, "");
713         return NGX\_CONF\_OK;
714     }
715
716     new = ngx\_pnalloc(cf->pool, sizeof("; domain=") - 1 + domain->len);
717     if (new == NULL) {
718         return NGX\_CONF\_ERROR;
719     }
720
721     p = ngx\_cpymem(new, "; domain=", sizeof("; domain=") - 1);

```

```

722     ngx_memcpy(p, domain->data, domain->len);
723
724     domain->len += sizeof("; domain=") - 1;
725     domain->data = new;
726
727     return NGX_CONF_OK;
728 }
729
730
731 static char *
732 ngx_http_userid_path(ngx_conf_t *cf, void *post, void *data)
733 {
734     ngx_str_t *path = data;
735
736     u_char *p, *new;
737
738     new = ngx_pnalloc(cf->pool, sizeof("; path=") - 1 + path->len);
739     if (new == NULL) {
740         return NGX_CONF_ERROR;
741     }
742
743     p = ngx_cpymem(new, "; path=", sizeof("; path=") - 1);
744     ngx_memcpy(p, path->data, path->len);
745
746     path->len += sizeof("; path=") - 1;
747     path->data = new;
748
749     return NGX_CONF_OK;
750 }
751
752
753 static char *
754 ngx_http_userid_expires(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
755 {
756     ngx_http_userid_conf_t *ucf = conf;
757
758     ngx_str_t *value;
759
760     if (ucf->expires != NGX_CONF_UNSET) {
761         return "is duplicate";
762     }
763
764     value = cf->args->elts;
765
766     if (ngx_strcmp(value[1].data, "max") == 0) {
767         ucf->expires = NGX_HTTP_USERID_MAX_EXPIRES;
768         return NGX_CONF_OK;
769     }
770
771     if (ngx_strcmp(value[1].data, "off") == 0) {
772         ucf->expires = 0;
773         return NGX_CONF_OK;
774     }
775
776     ucf->expires = ngx_parse_time(&value[1], 1);
777     if (ucf->expires == (time_t) NGX_ERROR) {
778         return "invalid value";
779     }
780
781     return NGX_CONF_OK;
782 }
783
784
785 static char *
786 ngx_http_userid_p3p(ngx_conf_t *cf, void *post, void *data)
787 {
788     ngx_str_t *p3p = data;
789
790     if (ngx_strcmp(p3p->data, "none") == 0) {
791         ngx_str_set(p3p, "");
792     }
793
794     return NGX_CONF_OK;
795 }
796
797

```

```

798 static char *
799 ngx_http_userid_mark(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
800 {
801     ngx_http_userid_conf_t *ucf = conf;
802
803     ngx_str_t *value;
804
805     if (ucf->mark != (u_char) '\xFF') {
806         return "is duplicate";
807     }
808
809     value = cf->args->elts;
810
811     if (ngx_strcmp(value[1].data, "off") == 0) {
812         ucf->mark = '\0';
813         return NGX_CONF_OK;
814     }
815
816     if (value[1].len != 1
817         || !((value[1].data[0] >= '0' && value[1].data[0] <= '9')
818             || (value[1].data[0] >= 'A' && value[1].data[0] <= 'Z')
819             || (value[1].data[0] >= 'a' && value[1].data[0] <= 'z')
820             || value[1].data[0] == '='))
821     {
822         return "value must be \"off\" or a single letter, digit or \"=\";";
823     }
824
825     ucf->mark = value[1].data[0];
826
827     return NGX_CONF_OK;
828 }
829
830
831 static ngx_int_t
832 ngx_http_userid_init_worker(ngx_cycle_t *cycle)
833 {
834     struct timeval tp;
835
836     ngx_gettimeofday(&tp);
837
838     /* use the most significant usec part that fits to 16 bits */
839     start_value = ((tp.tv_usec / 20) << 16) | ngx_pid;
840
841     return NGX_OK;
842 }

```

[One Level Up](#)

[Top Level](#)



## src/http/nginx\_http\_spdy\_filter\_module.c - nginx-1.7.10

### Global variables defined

- [ngx\\_http\\_next\\_header\\_filter](#)
- [ngx\\_http\\_spdy\\_filter\\_module](#)
- [ngx\\_http\\_spdy\\_filter\\_module\\_ctx](#)

### Functions defined

- [ngx\\_http\\_spdy\\_data\\_frame\\_handler](#)
- [ngx\\_http\\_spdy\\_filter\\_cleanup](#)
- [ngx\\_http\\_spdy\\_filter\\_get\\_data\\_frame](#)
- [ngx\\_http\\_spdy\\_filter\\_get\\_shadow](#)
- [ngx\\_http\\_spdy\\_filter\\_init](#)
- [ngx\\_http\\_spdy\\_filter\\_send](#)
- [ngx\\_http\\_spdy\\_flow\\_control](#)
- [ngx\\_http\\_spdy\\_handle\\_frame](#)
- [ngx\\_http\\_spdy\\_handle\\_stream](#)
- [ngx\\_http\\_spdy\\_header\\_filter](#)
- [ngx\\_http\\_spdy\\_send\\_chain](#)
- [ngx\\_http\\_spdy\\_syn\\_frame\\_handler](#)
- [ngx\\_http\\_spdy\\_waiting\\_queue](#)

### Macros defined

- [ngx\\_http\\_spdy\\_nv\\_nsize](#)
- [ngx\\_http\\_spdy\\_nv\\_vsize](#)
- [ngx\\_http\\_spdy\\_nv\\_write\\_name](#)
- [ngx\\_http\\_spdy\\_nv\\_write\\_nlen](#)
- [ngx\\_http\\_spdy\\_nv\\_write\\_num](#)
- [ngx\\_http\\_spdy\\_nv\\_write\\_val](#)
- [ngx\\_http\\_spdy\\_nv\\_write\\_vlen](#)

### Source code

```
1
2 /*
3  * Copyright (C) Nginx, Inc.
4  * Copyright (C) Valentin V. Bartenev
5  */
```

```

6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11 #include <nginx.h>
12 #include <ngx_http_spdy_module.h>
13
14 #include <zlib.h>
15
16
17 #define ngx_http_spdy_nv_nsize(h) (NGX_SPDY_NV_NLEN_SIZE + sizeof(h) - 1)
18 #define ngx_http_spdy_nv_vsize(h) (NGX_SPDY_NV_VLEN_SIZE + sizeof(h) - 1)
19
20 #define ngx_http_spdy_nv_write_num ngx_spdy_frame_write_uint32
21 #define ngx_http_spdy_nv_write_nlen ngx_spdy_frame_write_uint32
22 #define ngx_http_spdy_nv_write_vlen ngx_spdy_frame_write_uint32
23
24 #define ngx_http_spdy_nv_write_name(p, h) \
25     ngx_cpymem(ngx_http_spdy_nv_write_nlen(p, sizeof(h) - 1), h, sizeof(h) - 1)
26
27 #define ngx_http_spdy_nv_write_val(p, h) \
28     ngx_cpymem(ngx_http_spdy_nv_write_vlen(p, sizeof(h) - 1), h, sizeof(h) - 1)
29
30
31 static ngx_chain_t *ngx_http_spdy_send_chain(ngx_connection_t *fc,
32     ngx_chain_t *in, off_t limit);
33
34 static ngx_inline ngx_int_t ngx_http_spdy_filter_send(
35     ngx_connection_t *fc, ngx_http_spdy_stream_t *stream);
36 static ngx_inline ngx_int_t ngx_http_spdy_flow_control(
37     ngx_http_spdy_connection_t *sc, ngx_http_spdy_stream_t *stream);
38 static void ngx_http_spdy_waiting_queue(ngx_http_spdy_connection_t *sc,
39     ngx_http_spdy_stream_t *stream);
40
41 static ngx_chain_t *ngx_http_spdy_filter_get_shadow(
42     ngx_http_spdy_stream_t *stream, ngx_buf_t *buf, off_t offset, off_t size);
43 static ngx_http_spdy_out_frame_t *ngx_http_spdy_filter_get_data_frame(
44     ngx_http_spdy_stream_t *stream, size_t len, ngx_chain_t *first,
45     ngx_chain_t *last);
46
47 static ngx_int_t ngx_http_spdy_syn_frame_handler(
48     ngx_http_spdy_connection_t *sc, ngx_http_spdy_out_frame_t *frame);
49 static ngx_int_t ngx_http_spdy_data_frame_handler(
50     ngx_http_spdy_connection_t *sc, ngx_http_spdy_out_frame_t *frame);
51 static ngx_inline void ngx_http_spdy_handle_frame(
52     ngx_http_spdy_stream_t *stream, ngx_http_spdy_out_frame_t *frame);
53 static ngx_inline void ngx_http_spdy_handle_stream(
54     ngx_http_spdy_connection_t *sc, ngx_http_spdy_stream_t *stream);
55
56 static void ngx_http_spdy_filter_cleanup(void *data);
57
58 static ngx_int_t ngx_http_spdy_filter_init(ngx_conf_t *cf);
59
60
61 static ngx_http_module_t ngx_http_spdy_filter_module_ctx = {
62     NULL, /* preconfiguration */
63     ngx_http_spdy_filter_init, /* postconfiguration */
64
65     NULL, /* create main configuration */
66     NULL, /* init main configuration */
67
68     NULL, /* create server configuration */
69     NULL, /* merge server configuration */
70
71     NULL, /* create location configuration */
72     NULL, /* merge location configuration */
73 };
74
75
76 ngx_module_t ngx_http_spdy_filter_module = {
77     NGX_MODULE_V1,
78     &ngx_http_spdy_filter_module_ctx, /* module context */
79     NULL, /* module directives */
80     NGX_HTTP_MODULE, /* module type */
81     NULL, /* init master */

```

```

82     NULL,                                /* init module */
83     NULL,                                /* init process */
84     NULL,                                /* init thread */
85     NULL,                                /* exit thread */
86     NULL,                                /* exit process */
87     NULL,                                /* exit master */
88     NGX_MODULE_V1_PADDING
89 };
90
91
92 static ngx_http_output_header_filter_pt ngx_http_next_header_filter;
93
94
95 static ngx_int_t
96 ngx_http_spdy_header_filter(ngx_http_request_t *r)
97 {
98     int rc;
99     size_t len;
100    u_char *p, *buf, *last;
101    ngx_buf_t *b;
102    ngx_str_t host;
103    ngx_uint_t i, j, count, port;
104    ngx_chain_t *cl;
105    ngx_list_part_t *part, *pt;
106    ngx_table_elt_t *header, *h;
107    ngx_connection_t *c;
108    ngx_http_cleanup_t *cln;
109    ngx_http_core_loc_conf_t *clcf;
110    ngx_http_core_srv_conf_t *cscf;
111    ngx_http_spdy_stream_t *stream;
112    ngx_http_spdy_out_frame_t *frame;
113    ngx_http_spdy_connection_t *sc;
114    struct sockaddr_in *sin;
115    #if (NGX_HAVE_INET6)
116    struct sockaddr_in6 *sin6;
117    #endif
118    u_char addr[NGX_SOCKADDR_STRLEN];
119
120    if (!r->spdy_stream) {
121        return ngx_http_next_header_filter(r);
122    }
123
124    ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
125        "spdy header filter");
126
127    if (r->header_sent) {
128        return NGX_OK;
129    }
130
131    r->header_sent = 1;
132
133    if (r != r->main) {
134        return NGX_OK;
135    }
136
137    c = r->connection;
138
139    if (r->method == NGX_HTTP_HEAD) {
140        r->header_only = 1;
141    }
142
143    switch (r->headers_out.status) {
144
145    case NGX_HTTP_OK:
146    case NGX_HTTP_PARTIAL_CONTENT:
147        break;
148
149    case NGX_HTTP_NOT_MODIFIED:
150        r->header_only = 1;
151        break;
152
153    case NGX_HTTP_NO_CONTENT:
154        r->header_only = 1;
155
156        ngx_str_null(&r->headers_out.content_type);
157

```

```

158     r->headers_out.content_length = NULL;
159     r->headers_out.content_length_n = -1;
160
161     /* fall through */
162
163     default:
164         r->headers_out.last_modified_time = -1;
165         r->headers_out.last_modified = NULL;
166     }
167
168     len = NGX_SPDY_NV_NUM_SIZE
169         + ngx_http_spdy_nv_nsize(":version")
170         + ngx_http_spdy_nv_vsize("HTTP/1.1")
171         + ngx_http_spdy_nv_nsize(":status")
172         + (r->headers_out.status_line.len
173           ? NGX_SPDY_NV_VLEN_SIZE + r->headers_out.status_line.len
174           : ngx_http_spdy_nv_vsize("418"));
175
176     clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
177
178     if (r->headers_out.server == NULL) {
179         len += ngx_http_spdy_nv_nsize("server");
180         len += clcf->server_tokens ? ngx_http_spdy_nv_vsize(NGINX_VER)
181             : ngx_http_spdy_nv_vsize("nginx");
182     }
183
184     if (r->headers_out.date == NULL) {
185         len += ngx_http_spdy_nv_nsize("date")
186             + ngx_http_spdy_nv_vsize("Wed, 31 Dec 1986 10:00:00 GMT");
187     }
188
189     if (r->headers_out.content_type.len) {
190         len += ngx_http_spdy_nv_nsize("content-type")
191             + NGX_SPDY_NV_VLEN_SIZE + r->headers_out.content_type.len;
192
193         if (r->headers_out.content_type_len == r->headers_out.content_type.len
194             && r->headers_out.charset.len)
195             {
196                 len += sizeof("; charset=") - 1 + r->headers_out.charset.len;
197             }
198     }
199
200     if (r->headers_out.content_length == NULL
201         && r->headers_out.content_length_n >= 0)
202     {
203         len += ngx_http_spdy_nv_nsize("content-length")
204             + NGX_SPDY_NV_VLEN_SIZE + NGX_OFF_T_LEN;
205     }
206
207     if (r->headers_out.last_modified == NULL
208         && r->headers_out.last_modified_time != -1)
209     {
210         len += ngx_http_spdy_nv_nsize("last-modified")
211             + ngx_http_spdy_nv_vsize("Wed, 31 Dec 1986 10:00:00 GMT");
212     }
213
214     if (r->headers_out.location
215         && r->headers_out.location->value.len
216         && r->headers_out.location->value.data[0] == '/')
217     {
218         r->headers_out.location->hash = 0;
219
220         if (clcf->server_name_in_redirect) {
221             cscf = ngx_http_get_module_srv_conf(r, ngx_http_core_module);
222             host = cscf->server_name;
223
224         } else if (r->headers_in.server.len) {
225             host = r->headers_in.server;
226
227         } else {
228             host.len = NGX_SOCKADDR_STRLEN;
229             host.data = addr;
230
231             if (ngx_connection_local_sockaddr(c, &host, 0) != NGX_OK) {
232                 return NGX_ERROR;
233             }

```

```

234     }
235
236     switch (c->local_sockaddr->sa_family) {
237
238     #if (NGX_HAVE_INET6)
239         case AF_INET6:
240             sin6 = (struct sockaddr_in6 *) c->local_sockaddr;
241             port = ntohs(sin6->sin6_port);
242             break;
243     #endif
244     #if (NGX_HAVE_UNIX_DOMAIN)
245         case AF_UNIX:
246             port = 0;
247             break;
248     #endif
249     default: /* AF_INET */
250         sin = (struct sockaddr_in *) c->local_sockaddr;
251         port = ntohs(sin->sin_port);
252         break;
253     }
254
255     len += ngx_http_spdy_nv_nsize("location")
256           + ngx_http_spdy_nv_vsize("https://")
257           + host.len
258           + r->headers_out.location->value.len;
259
260     if (clcf->port_in_redirect) {
261
262     #if (NGX_HTTP_SSL)
263         if (c->ssl)
264             port = (port == 443) ? 0 : port;
265         else
266     #endif
267             port = (port == 80) ? 0 : port;
268
269     } else {
270         port = 0;
271     }
272
273     if (port) {
274         len += sizeof(":65535") - 1;
275     }
276
277 } else {
278     ngx_str_null(&host);
279     port = 0;
280 }
281
282 part = &r->headers_out.headers.part;
283 header = part->elts;
284
285 for (i = 0; /* void */; i++) {
286
287     if (i >= part->nelts) {
288         if (part->next == NULL) {
289             break;
290         }
291
292         part = part->next;
293         header = part->elts;
294         i = 0;
295     }
296
297     if (header[i].hash == 0) {
298         continue;
299     }
300
301     len += NGX_SPDY_NV_NLEN_SIZE + header[i].key.len
302           + NGX_SPDY_NV_VLEN_SIZE + header[i].value.len;
303 }
304
305 buf = ngx_alloc(len, r->pool->log);
306 if (buf == NULL) {
307     return NGX_ERROR;
308 }
309

```

```

310 last = buf + NGX_SPDY_NV_NUM_SIZE;
311
312 last = ngx_http_spdy_nv_write_name(last, ":version");
313 last = ngx_http_spdy_nv_write_val(last, "HTTP/1.1");
314
315 last = ngx_http_spdy_nv_write_name(last, ":status");
316
317 if (r->headers_out.status_line.len) {
318     last = ngx_http_spdy_nv_write_vlen(last,
319                                         r->headers_out.status_line.len);
320     last = ngx_cpymem(last, r->headers_out.status_line.data,
321                       r->headers_out.status_line.len);
322 } else {
323     last = ngx_http_spdy_nv_write_vlen(last, 3);
324     last = ngx_sprintf(last, "%03ui", r->headers_out.status);
325 }
326
327 count = 2;
328
329 if (r->headers_out.server == NULL) {
330     last = ngx_http_spdy_nv_write_name(last, "server");
331     last = clcf->server_tokens
332         ? ngx_http_spdy_nv_write_val(last, NGINX_VER)
333         : ngx_http_spdy_nv_write_val(last, "nginx");
334
335     count++;
336 }
337
338 if (r->headers_out.date == NULL) {
339     last = ngx_http_spdy_nv_write_name(last, "date");
340
341     last = ngx_http_spdy_nv_write_vlen(last, ngx_cached_http_time.len);
342
343     last = ngx_cpymem(last, ngx_cached_http_time.data,
344                       ngx_cached_http_time.len);
345
346     count++;
347 }
348
349 if (r->headers_out.content_type.len) {
350
351     last = ngx_http_spdy_nv_write_name(last, "content-type");
352
353     p = last + NGX_SPDY_NV_VLEN_SIZE;
354
355     last = ngx_cpymem(p, r->headers_out.content_type.data,
356                       r->headers_out.content_type.len);
357
358     if (r->headers_out.content_type_len == r->headers_out.content_type.len
359         && r->headers_out.charset.len)
360     {
361         last = ngx_cpymem(last, "; charset=", sizeof("; charset=") - 1);
362
363         last = ngx_cpymem(last, r->headers_out.charset.data,
364                           r->headers_out.charset.len);
365
366         /* update r->headers_out.content_type for possible logging */
367
368         r->headers_out.content_type.len = last - p;
369         r->headers_out.content_type.data = p;
370     }
371
372     (void) ngx_http_spdy_nv_write_vlen(p - NGX_SPDY_NV_VLEN_SIZE,
373                                         r->headers_out.content_type.len);
374
375     count++;
376 }
377
378 if (r->headers_out.content_length == NULL
379     && r->headers_out.content_length_n >= 0)
380 {
381     last = ngx_http_spdy_nv_write_name(last, "content-length");
382
383     p = last + NGX_SPDY_NV_VLEN_SIZE;
384
385     last = ngx_sprintf(p, "%0", r->headers_out.content_length_n);

```

```

386     (void) ngx_http_spdy_nv_write_vlen(p - NGX_SPDY_NV_VLEN_SIZE,
387                                       last - p);
388
389     count++;
390 }
391
392
393 if (r->headers_out.last_modified == NULL
394     && r->headers_out.last_modified_time != -1)
395 {
396     last = ngx_http_spdy_nv_write_name(last, "last-modified");
397
398     p = last + NGX_SPDY_NV_VLEN_SIZE;
399
400     last = ngx_http_time(p, r->headers_out.last_modified_time);
401
402     (void) ngx_http_spdy_nv_write_vlen(p - NGX_SPDY_NV_VLEN_SIZE,
403                                       last - p);
404
405     count++;
406 }
407
408 if (host.data) {
409
410     last = ngx_http_spdy_nv_write_name(last, "location");
411
412     p = last + NGX_SPDY_NV_VLEN_SIZE;
413
414     last = ngx_cpymem(p, "http", sizeof("http") - 1);
415
416 #if (NGX_HTTP_SSL)
417     if (c->ssl) {
418         *last++ = 's';
419     }
420 #endif
421
422     *last++ = ':'; *last++ = '/'; *last++ = '/';
423
424     last = ngx_cpymem(last, host.data, host.len);
425
426     if (port) {
427         last = ngx_sprintf(last, ":%ui", port);
428     }
429
430     last = ngx_cpymem(last, r->headers_out.location->value.data,
431                       r->headers_out.location->value.len);
432
433     /* update r->headers_out.location->value for possible logging */
434
435     r->headers_out.location->value.len = last - p;
436     r->headers_out.location->value.data = p;
437     ngx_str_set(&r->headers_out.location->key, "location");
438
439     (void) ngx_http_spdy_nv_write_vlen(p - NGX_SPDY_NV_VLEN_SIZE,
440                                       r->headers_out.location->value.len);
441
442     count++;
443 }
444
445 part = &r->headers_out.headers.part;
446 header = part->elts;
447
448 for (i = 0; /* void */; i++) {
449
450     if (i >= part->nelts) {
451         if (part->next == NULL) {
452             break;
453         }
454
455         part = part->next;
456         header = part->elts;
457         i = 0;
458     }
459
460     if (header[i].hash == 0 || header[i].hash == 2) {
461         continue;

```

```

462     }
463
464     last = ngx_http_spdy_nv_write_nlen(last, header[i].key.len);
465
466     ngx_strlow(last, header[i].key.data, header[i].key.len);
467     last += header[i].key.len;
468
469     p = last + NGX_SPDY_NV_VLEN_SIZE;
470
471     last = ngx_cpymem(p, header[i].value.data, header[i].value.len);
472
473     pt = part;
474     h = header;
475
476     for (j = i + 1; /* void */; j++) {
477
478         if (j >= pt->nelts) {
479             if (pt->next == NULL) {
480                 break;
481             }
482
483             pt = pt->next;
484             h = pt->elts;
485             j = 0;
486         }
487
488         if (h[j].hash == 0 || h[j].hash == 2
489             || h[j].key.len != header[i].key.len
490             || ngx_strncasecmp(header[i].key.data, h[j].key.data,
491                               header[i].key.len))
492         {
493             continue;
494         }
495
496         if (h[j].value.len) {
497             if (last != p) {
498                 *last++ = '\\0';
499             }
500
501             last = ngx_cpymem(last, h[j].value.data, h[j].value.len);
502         }
503
504         h[j].hash = 2;
505     }
506
507     (void) ngx_http_spdy_nv_write_vlen(p - NGX_SPDY_NV_VLEN_SIZE,
508                                       last - p);
509
510     count++;
511 }
512
513 (void) ngx_http_spdy_nv_write_num(buf, count);
514
515 stream = r->spdy_stream;
516 sc = stream->connection;
517
518 len = last - buf;
519
520 b = ngx_create_temp_buf(r->pool, NGX_SPDY_FRAME_HEADER_SIZE
521                        + NGX_SPDY_SYN_REPLY_SIZE
522                        + deflateBound(&sc->zstream_out, len));
523
524 if (b == NULL) {
525     ngx_free(buf);
526     return NGX_ERROR;
527 }
528
529 b->last += NGX_SPDY_FRAME_HEADER_SIZE + NGX_SPDY_SYN_REPLY_SIZE;
530
531 sc->zstream_out.next_in = buf;
532 sc->zstream_out.avail_in = len;
533 sc->zstream_out.next_out = b->last;
534 sc->zstream_out.avail_out = b->end - b->last;
535
536 rc = deflate(&sc->zstream_out, Z_SYNC_FLUSH);
537
538 ngx_free(buf);

```



```

538
539 if (rc != Z_OK) {
540     ngx_log_error(NGX_LOG_ALERT, c->log, 0, "deflate() failed: %d", rc);
541     return NGX_ERROR;
542 }
543
544 ngx_log_debug5(NGX_LOG_DEBUG_HTTP, c->log, 0,
545     "spdy deflate out: ni:%p no:%p ai:%ud ao:%ud rc:%d",
546     sc->zstream_out.next_in, sc->zstream_out.next_out,
547     sc->zstream_out.avail_in, sc->zstream_out.avail_out,
548     rc);
549
550 b->last = sc->zstream_out.next_out;
551
552 p = b->pos;
553 p = ngx_spdy_frame_write_head(p, NGX_SPDY_SYN_REPLY);
554
555 len = b->last - b->pos;
556
557 r->header_size = len;
558
559 len -= NGX_SPDY_FRAME_HEADER_SIZE;
560
561 if (r->header_only) {
562     b->last_buf = 1;
563     p = ngx_spdy_frame_write_flags_and_len(p, NGX_SPDY_FLAG_FIN, len);
564 } else {
565     p = ngx_spdy_frame_write_flags_and_len(p, 0, len);
566 }
567
568 (void) ngx_spdy_frame_write_sid(p, stream->id);
569
570
571 cl = ngx_alloc_chain_link(r->pool);
572 if (cl == NULL) {
573     return NGX_ERROR;
574 }
575
576 cl->buf = b;
577 cl->next = NULL;
578
579 frame = ngx_palloc(r->pool, sizeof(ngx_http_spdy_out_frame_t));
580 if (frame == NULL) {
581     return NGX_ERROR;
582 }
583
584 frame->first = cl;
585 frame->last = cl;
586 frame->handler = ngx_http_spdy_syn_frame_handler;
587 frame->stream = stream;
588 frame->length = len;
589 frame->priority = stream->priority;
590 frame->blocked = 1;
591 frame->fin = r->header_only;
592
593 ngx_log_debug3(NGX_LOG_DEBUG_HTTP, stream->request->connection->log, 0,
594     "spdy:%ui create SYN_REPLY frame %p: len:%uz",
595     stream->id, frame, frame->length);
596
597 ngx_http_spdy_queue_blocked_frame(sc, frame);
598
599 cIn = ngx_http_cleanup_add(r, 0);
600 if (cIn == NULL) {
601     return NGX_ERROR;
602 }
603
604 cIn->handler = ngx_http_spdy_filter_cleanup;
605 cIn->data = stream;
606
607 stream->queued = 1;
608
609 c->send_chain = ngx_http_spdy_send_chain;
610 c->need_last_buf = 1;
611
612 return ngx_http_spdy_filter_send(c, stream);
613 }

```

```

614
615
616 static ngx_chain_t *
617 ngx_http_spdy_send_chain(ngx_connection_t *fc, ngx_chain_t *in, off_t limit)
618 {
619     off_t          size, offset;
620     size_t         rest, frame_size;
621     ngx_chain_t   *cl, *out, **ln;
622     ngx_http_request_t *r;
623     ngx_http_spdy_stream_t *stream;
624     ngx_http_spdy_loc_conf_t *slcf;
625     ngx_http_spdy_out_frame_t *frame;
626     ngx_http_spdy_connection_t *sc;
627
628     r = fc->data;
629     stream = r->spdy_stream;
630
631     #if (NGX_SUPPRESS_WARN)
632     size = 0;
633     #endif
634
635     while (in) {
636         size = ngx_buf_size(in->buf);
637
638         if (size || in->buf->last_buf) {
639             break;
640         }
641
642         in = in->next;
643     }
644
645     if (in == NULL) {
646
647         if (stream->queued) {
648             fc->write->delayed = 1;
649         } else {
650             fc->buffered &= ~NGX_SPDY_BUFFERED;
651         }
652
653         return NULL;
654     }
655
656     sc = stream->connection;
657
658     if (size && ngx_http_spdy_flow_control(sc, stream) == NGX_DECLINED) {
659         fc->write->delayed = 1;
660         return in;
661     }
662
663     if (limit == 0 || limit > (off_t) sc->send_window) {
664         limit = sc->send_window;
665     }
666
667     if (limit > stream->send_window) {
668         limit = (stream->send_window > 0) ? stream->send_window : 0;
669     }
670
671     if (in->buf->tag == (ngx_buf_tag_t) &ngx_http_spdy_filter_get_shadow) {
672         cl = ngx_alloc_chain_link(r->pool);
673         if (cl == NULL) {
674             return NGX_CHAIN_ERROR;
675         }
676
677         cl->buf = in->buf;
678         in->buf = cl->buf->shadow;
679
680         offset = ngx_buf_in_memory(in->buf)
681             ? (cl->buf->pos - in->buf->pos)
682             : (cl->buf->file_pos - in->buf->file_pos);
683
684         cl->next = stream->free_bufs;
685         stream->free_bufs = cl;
686
687     } else {
688         offset = 0;
689     }

```

```

690
691 #if (NGX_SUPPRESS_WARN)
692     cl = NULL;
693 #endif
694
695     slcf = ngx_http_get_module_loc_conf(r, ngx_http_spdy_module);
696
697     frame_size = (limit <= (off_t) slcf->chunk_size) ? (size_t) limit
698                 : slcf->chunk_size;
699
700     for ( ;; ) {
701         ln = &out;
702         rest = frame_size;
703
704         while ((off_t) rest >= size) {
705
706             if (offset) {
707                 cl = ngx_http_spdy_filter_get_shadow(stream, in->buf,
708                                                     offset, size);
709
710                 if (cl == NULL) {
711                     return NGX_CHAIN_ERROR;
712                 }
713
714                 offset = 0;
715
716             } else {
717                 cl = ngx_alloc_chain_link(r->pool);
718                 if (cl == NULL) {
719                     return NGX_CHAIN_ERROR;
720                 }
721
722                 cl->buf = in->buf;
723             }
724
725             *ln = cl;
726             ln = &cl->next;
727
728             rest -= (size_t) size;
729             in = in->next;
730
731             if (in == NULL) {
732                 frame_size -= rest;
733                 rest = 0;
734                 break;
735             }
736
737             size = ngx_buf_size(in->buf);
738         }
739
740         if (rest) {
741             cl = ngx_http_spdy_filter_get_shadow(stream, in->buf,
742                                                 offset, rest);
743
744             if (cl == NULL) {
745                 return NGX_CHAIN_ERROR;
746             }
747
748             cl->buf->flush = 0;
749             cl->buf->last_buf = 0;
750
751             *ln = cl;
752
753             offset += rest;
754             size -= rest;
755         }
756
757         frame = ngx_http_spdy_filter_get_data_frame(stream, frame_size,
758                                                    out, cl);
759
760         if (frame == NULL) {
761             return NGX_CHAIN_ERROR;
762         }
763
764         ngx_http_spdy_queue_frame(sc, frame);
765
766         sc->send_window -= frame_size;
767
768         stream->send_window -= frame_size;

```

```

766     stream->queued++;
767
768     if (in == NULL) {
769         break;
770     }
771
772     limit -= frame_size;
773
774     if (limit == 0) {
775         break;
776     }
777
778     if (limit < (off_t) slcf->chunk_size) {
779         frame_size = (size_t) limit;
780     }
781 }
782
783 if (offset) {
784     cl = ngx\_http\_spdy\_filter\_get\_shadow(stream, in->buf, offset, size);
785     if (cl == NULL) {
786         return NGX\_CHAIN\_ERROR;
787     }
788
789     in->buf = cl->buf;
790     ngx\_free\_chain(r->pool, cl);
791 }
792
793 if (ngx\_http\_spdy\_filter\_send(fc, stream) == NGX\_ERROR) {
794     return NGX\_CHAIN\_ERROR;
795 }
796
797 if (in && ngx\_http\_spdy\_flow\_control(sc, stream) == NGX\_DECLINED) {
798     fc->write->delayed = 1;
799 }
800
801 return in;
802 }
803
804
805 static ngx\_chain\_t *
806 ngx\_http\_spdy\_filter\_get\_shadow(ngx\_http\_spdy\_stream\_t *stream, ngx\_buf\_t *buf,
807     off_t offset, off_t size)
808 {
809     ngx\_buf\_t     *chunk;
810     ngx\_chain\_t  *cl;
811
812     cl = ngx\_chain\_get\_free\_buf(stream->request->pool, &stream->free_bufs);
813     if (cl == NULL) {
814         return NULL;
815     }
816
817     chunk = cl->buf;
818
819     ngx\_memcpy(chunk, buf, sizeof(ngx\_buf\_t));
820
821     chunk->tag = (ngx\_buf\_tag\_t) &ngx\_http\_spdy\_filter\_get\_shadow;
822     chunk->shadow = buf;
823
824     if (ngx\_buf\_in\_memory(chunk)) {
825         chunk->pos += offset;
826         chunk->last = chunk->pos + size;
827     }
828
829     if (chunk->in_file) {
830         chunk->file_pos += offset;
831         chunk->file_last = chunk->file_pos + size;
832     }
833
834     return cl;
835 }
836
837
838 static ngx\_http\_spdy\_out\_frame\_t *
839 ngx\_http\_spdy\_filter\_get\_data\_frame(ngx\_http\_spdy\_stream\_t *stream,
840     size_t len, ngx\_chain\_t *first, ngx\_chain\_t *last)
841 {

```

```

842     u_char                *p;
843     ngx_buf_t            *buf;
844     ngx_uint_t           flags;
845     ngx_chain_t          *cl;
846     ngx_http_spdy_out_frame_t *frame;
847
848
849     frame = stream->free_frames;
850
851     if (frame) {
852         stream->free_frames = frame->next;
853
854     } else {
855         frame = ngx_palloc(stream->request->pool,
856             sizeof(ngx_http_spdy_out_frame_t));
857         if (frame == NULL) {
858             return NULL;
859         }
860     }
861
862     flags = last->buf->last_buf ? NGX_SPDY_FLAG_FIN : 0;
863
864     ngx_log_debug4(NGX_LOG_DEBUG_HTTP, stream->request->connection->log, 0,
865         "spdy:%ui create DATA frame %p: len:%uz flags:%ui",
866         stream->id, frame, len, flags);
867
868     cl = ngx_chain_get_free_buf(stream->request->pool,
869         &stream->free_data_headers);
870     if (cl == NULL) {
871         return NULL;
872     }
873
874     buf = cl->buf;
875
876     if (buf->start) {
877         p = buf->start;
878         buf->pos = p;
879
880         p += NGX_SPDY_SID_SIZE;
881
882         (void) ngx_spdy_frame_write_flags_and_len(p, flags, len);
883
884     } else {
885         p = ngx_palloc(stream->request->pool, NGX_SPDY_FRAME_HEADER_SIZE);
886         if (p == NULL) {
887             return NULL;
888         }
889
890         buf->pos = p;
891         buf->start = p;
892
893         p = ngx_spdy_frame_write_sid(p, stream->id);
894         p = ngx_spdy_frame_write_flags_and_len(p, flags, len);
895
896         buf->last = p;
897         buf->end = p;
898
899         buf->tag = (ngx_buf_tag_t) &ngx_http_spdy_filter_get_data_frame;
900         buf->memory = 1;
901     }
902
903     cl->next = first;
904     first = cl;
905
906     last->buf->flush = 1;
907
908     frame->first = first;
909     frame->last = last;
910     frame->handler = ngx_http_spdy_data_frame_handler;
911     frame->stream = stream;
912     frame->length = len;
913     frame->priority = stream->priority;
914     frame->blocked = 0;
915     frame->fin = last->buf->last_buf;
916
917     return frame;

```

```

918 }
919
920
921 static ngx_inline ngx_int_t
922 ngx_http_spdy_filter_send(ngx_connection_t *fc, ngx_http_spdy_stream_t *stream)
923 {
924     stream->blocked = 1;
925
926     if (ngx_http_spdy_send_output_queue(stream->connection) == NGX_ERROR) {
927         fc->error = 1;
928         return NGX_ERROR;
929     }
930
931     stream->blocked = 0;
932
933     if (stream->queued) {
934         fc->buffered |= NGX_SPDY_BUFFERED;
935         fc->write->delayed = 1;
936         return NGX_AGAIN;
937     }
938
939     fc->buffered &= ~NGX_SPDY_BUFFERED;
940
941     return NGX_OK;
942 }
943
944
945 static ngx_inline ngx_int_t
946 ngx_http_spdy_flow_control(ngx_http_spdy_connection_t *sc,
947     ngx_http_spdy_stream_t *stream)
948 {
949     if (stream->send_window <= 0) {
950         stream->exhausted = 1;
951         return NGX_DECLINED;
952     }
953
954     if (sc->send_window == 0) {
955         ngx_http_spdy_waiting_queue(sc, stream);
956         return NGX_DECLINED;
957     }
958
959     return NGX_OK;
960 }
961
962
963 static void
964 ngx_http_spdy_waiting_queue(ngx_http_spdy_connection_t *sc,
965     ngx_http_spdy_stream_t *stream)
966 {
967     ngx_queue_t *q;
968     ngx_http_spdy_stream_t *s;
969
970     if (stream->handled) {
971         return;
972     }
973
974     stream->handled = 1;
975
976     for (q = ngx_queue_last(&sc->waiting);
977         q != ngx_queue_sentinel(&sc->waiting);
978         q = ngx_queue_prev(q))
979     {
980         s = ngx_queue_data(q, ngx_http_spdy_stream_t, queue);
981
982         /*
983          * NB: higher values represent lower priorities.
984          */
985         if (stream->priority >= s->priority) {
986             break;
987         }
988     }
989
990     ngx_queue_insert_after(q, &stream->queue);
991 }
992
993

```

```

994 static ngx_int_t
995 ngx_http_spdy_syn_frame_handler(ngx_http_spdy_connection_t *sc,
996     ngx_http_spdy_out_frame_t *frame)
997 {
998     ngx_buf_t          *buf;
999     ngx_http_spdy_stream_t *stream;
1000
1001     buf = frame->first->buf;
1002
1003     if (buf->pos != buf->last) {
1004         return NGX_AGAIN;
1005     }
1006
1007     stream = frame->stream;
1008
1009     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, sc->connection->log, 0,
1010         "spdy:%ui SYN_REPLY frame %p was sent", stream->id, frame);
1011
1012     ngx_free_chain(stream->request->pool, frame->first);
1013
1014     ngx_http_spdy_handle_frame(stream, frame);
1015
1016     ngx_http_spdy_handle_stream(sc, stream);
1017
1018     return NGX_OK;
1019 }
1020
1021
1022 static ngx_int_t
1023 ngx_http_spdy_data_frame_handler(ngx_http_spdy_connection_t *sc,
1024     ngx_http_spdy_out_frame_t *frame)
1025 {
1026     ngx_buf_t          *buf;
1027     ngx_chain_t        *cl, *ln;
1028     ngx_http_spdy_stream_t *stream;
1029
1030     stream = frame->stream;
1031
1032     cl = frame->first;
1033
1034     if (cl->buf->tag == (ngx_buf_tag_t) &ngx_http_spdy_filter_get_data_frame) {
1035
1036         if (cl->buf->pos != cl->buf->last) {
1037             ngx_log_debug2(NGX_LOG_DEBUG_HTTP, sc->connection->log, 0,
1038                 "spdy:%ui DATA frame %p was sent partially",
1039                 stream->id, frame);
1040
1041             return NGX_AGAIN;
1042         }
1043
1044         ln = cl->next;
1045
1046         cl->next = stream->free_data_headers;
1047         stream->free_data_headers = cl;
1048
1049         if (cl == frame->last) {
1050             goto done;
1051         }
1052
1053         cl = ln;
1054     }
1055
1056     for ( ;; ) {
1057         if (cl->buf->tag == (ngx_buf_tag_t) &ngx_http_spdy_filter_get_shadow) {
1058             buf = cl->buf->shadow;
1059
1060             if (ngx_buf_in_memory(buf)) {
1061                 buf->pos = cl->buf->pos;
1062             }
1063
1064             if (buf->in_file) {
1065                 buf->file_pos = cl->buf->file_pos;
1066             }
1067         }
1068
1069         if (ngx_buf_size(cl->buf) != 0) {

```

```

1070         if (cl != frame->first) {
1071             frame->first = cl;
1072             ngx\_http\_spdy\_handle\_stream(sc, stream);
1073         }
1074
1075         ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, sc->connection->log, 0,
1076             "spdy:%ui DATA frame %p was sent partially",
1077             stream->id, frame);
1078
1079         return NGX\_AGAIN;
1080     }
1081
1082     ln = cl->next;
1083
1084     if (cl->buf->tag == (ngx\_buf\_tag\_t) &ngx\_http\_spdy\_filter\_get\_shadow) {
1085         cl->next = stream->free_bufs;
1086         stream->free_bufs = cl;
1087     } else {
1088         ngx\_free\_chain(stream->request->pool, cl);
1089     }
1090
1091     if (cl == frame->last) {
1092         goto done;
1093     }
1094
1095     cl = ln;
1096 }
1097
1098 done:
1099
1100     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, sc->connection->log, 0,
1101         "spdy:%ui DATA frame %p was sent", stream->id, frame);
1102
1103     stream->request->header_size += NGX\_SPDY\_FRAME\_HEADER\_SIZE;
1104
1105     ngx\_http\_spdy\_handle\_frame(stream, frame);
1106
1107     ngx\_http\_spdy\_handle\_stream(sc, stream);
1108
1109     return NGX\_OK;
1110 }
1111
1112 static ngx\_inline void
1113 ngx\_http\_spdy\_handle\_frame(ngx\_http\_spdy\_stream\_t *stream,
1114     ngx\_http\_spdy\_out\_frame\_t *frame)
1115 {
1116     ngx\_http\_request\_t *r;
1117
1118     r = stream->request;
1119
1120     r->connection->sent += NGX\_SPDY\_FRAME\_HEADER\_SIZE + frame->length;
1121
1122     if (frame->fin) {
1123         stream->out_closed = 1;
1124     }
1125
1126     frame->next = stream->free_frames;
1127     stream->free_frames = frame;
1128
1129     stream->queued--;
1130 }
1131
1132 static ngx\_inline void
1133 ngx\_http\_spdy\_handle\_stream(ngx\_http\_spdy\_connection\_t *sc,
1134     ngx\_http\_spdy\_stream\_t *stream)
1135 {
1136     ngx\_event\_t *wev;
1137
1138     if (stream->handled || stream->blocked || stream->exhausted) {
1139         return;
1140     }
1141 }
1142
1143
1144
1145

```



```

1146     wev = stream->request->connection->write;
1147
1148     /*
1149     * This timer can only be set if the stream was delayed because of rate
1150     * limit. In that case the event should be triggered by the timer.
1151     */
1152
1153     if (!wev->timer_set) {
1154         wev->delayed = 0;
1155
1156         stream->handled = 1;
1157         ngx\_queue\_insert\_tail(&sc->posted, &stream->queue);
1158     }
1159 }
1160
1161
1162 static void
1163 ngx_http_spdy_filter_cleanup(void *data)
1164 {
1165     ngx\_http\_spdy\_stream\_t *stream = data;
1166
1167     size_t                delta;
1168     ngx\_http\_spdy\_out\_frame\_t *frame, **fn;
1169     ngx\_http\_spdy\_connection\_t *sc;
1170
1171     if (stream->handled) {
1172         stream->handled = 0;
1173         ngx\_queue\_remove(&stream->queue);
1174     }
1175
1176     if (stream->queued == 0) {
1177         return;
1178     }
1179
1180     delta = 0;
1181     sc = stream->connection;
1182     fn = &sc->last_out;
1183
1184     for ( ;; ) {
1185         frame = *fn;
1186
1187         if (frame == NULL) {
1188             break;
1189         }
1190
1191         if (frame->stream == stream && !frame->blocked) {
1192             *fn = frame->next;
1193
1194             delta += frame->length;
1195
1196             if (--stream->queued == 0) {
1197                 break;
1198             }
1199
1200             continue;
1201         }
1202
1203         fn = &frame->next;
1204     }
1205
1206     if (sc->send_window == 0 && delta && !ngx\_queue\_empty(&sc->waiting)) {
1207         ngx\_queue\_add(&sc->posted, &sc->waiting);
1208         ngx\_queue\_init(&sc->waiting);
1209     }
1210
1211     sc->send_window += delta;
1212 }
1213
1214
1215 static ngx\_int\_t
1216 ngx_http_spdy_filter_init(ngx\_conf\_t *cf)
1217 {
1218     ngx\_http\_next\_header\_filter = ngx\_http\_top\_header\_filter;
1219     ngx\_http\_top\_header\_filter = ngx\_http\_spdy\_header\_filter;
1220
1221     return NGX\_OK;

```

[One Level Up](#)

[Top Level](#)

# src/http/nginx\_http\_parse\_time.c - nginx-1.7.10

## Global variables defined

- [mday](#)

## Functions defined

- [ngx\\_http\\_parse\\_time](#)

## Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 static ngx_uint_t mday[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
14
15 time_t
16 ngx_http_parse_time(u_char *value, size_t len)
17 {
18     u_char      *p, *end;
19     ngx_int_t    month;
20     ngx_uint_t   day, year, hour, min, sec;
21     uint64_t     time;
22     enum {
23         no = 0,
24         rfc822, /* Tue, 10 Nov 2002 23:50:13 */
25         rfc850, /* Tuesday, 10-Dec-02 23:50:13 */
26         isoc    /* Tue Dec 10 23:50:13 2002 */
27     } fmt;
28
29     fmt = 0;
30     end = value + len;
31
32     #if (NGX_SUPPRESS_WARN)
33         day = 32;
34         year = 2038;
35     #endif
36
37     for (p = value; p < end; p++) {
38         if (*p == ',') {
39             break;
40         }
41
42         if (*p == ' ') {
43             fmt = isoc;
44             break;
45         }
46     }
47
48     for (p++; p < end; p++)
49         if (*p != ' ') {
50             break;
51         }
52
53     if (end - p < 18) {
54         return NGX_ERROR;
55     }
56
57     if (fmt != isoc) {
```

```

58     if (*p < '0' || *p > '9' || *(p + 1) < '0' || *(p + 1) > '9') {
59         return NGX\_ERROR;
60     }
61
62     day = (*p - '0') * 10 + *(p + 1) - '0';
63     p += 2;
64
65     if (*p == ' ') {
66         if (end - p < 18) {
67             return NGX\_ERROR;
68         }
69         fmt = rfc822;
70
71     } else if (*p == '-') {
72         fmt = rfc850;
73
74     } else {
75         return NGX\_ERROR;
76     }
77
78     p++;
79 }
80
81 switch (*p) {
82
83 case 'J':
84     month = *(p + 1) == 'a' ? 0 : *(p + 2) == 'n' ? 5 : 6;
85     break;
86
87 case 'F':
88     month = 1;
89     break;
90
91 case 'M':
92     month = *(p + 2) == 'r' ? 2 : 4;
93     break;
94
95 case 'A':
96     month = *(p + 1) == 'p' ? 3 : 7;
97     break;
98
99 case 'S':
100    month = 8;
101    break;
102
103 case '0':
104    month = 9;
105    break;
106
107 case 'N':
108    month = 10;
109    break;
110
111 case 'D':
112    month = 11;
113    break;
114
115 default:
116    return NGX\_ERROR;
117 }
118
119 p += 3;
120
121 if ((fmt == rfc822 && *p != ' ') || (fmt == rfc850 && *p != '-')) {
122     return NGX\_ERROR;
123 }
124
125 p++;
126
127 if (fmt == rfc822) {
128     if (*p < '0' || *p > '9' || *(p + 1) < '0' || *(p + 1) > '9'
129         || *(p + 2) < '0' || *(p + 2) > '9'
130         || *(p + 3) < '0' || *(p + 3) > '9')
131     {
132         return NGX\_ERROR;
133     }

```

```

134     year = (*p - '0') * 1000 + (*(p + 1) - '0') * 100
135           + (*(p + 2) - '0') * 10 + *(p + 3) - '0';
136     p += 4;
137
138
139 } else if (fmt == rfc850) {
140     if (*p < '0' || *p > '9' || *(p + 1) < '0' || *(p + 1) > '9') {
141         return NGX_ERROR;
142     }
143
144     year = (*p - '0') * 10 + *(p + 1) - '0';
145     year += (year < 70) ? 2000 : 1900;
146     p += 2;
147 }
148
149 if (fmt == isoc) {
150     if (*p == ' ') {
151         p++;
152     }
153
154     if (*p < '0' || *p > '9') {
155         return NGX_ERROR;
156     }
157
158     day = *p++ - '0';
159
160     if (*p != ' ') {
161         if (*p < '0' || *p > '9') {
162             return NGX_ERROR;
163         }
164
165         day = day * 10 + *p++ - '0';
166     }
167
168     if (end - p < 14) {
169         return NGX_ERROR;
170     }
171 }
172
173 if (*p++ != ' ') {
174     return NGX_ERROR;
175 }
176
177 if (*p < '0' || *p > '9' || *(p + 1) < '0' || *(p + 1) > '9') {
178     return NGX_ERROR;
179 }
180
181 hour = (*p - '0') * 10 + *(p + 1) - '0';
182 p += 2;
183
184 if (*p++ != ':') {
185     return NGX_ERROR;
186 }
187
188 if (*p < '0' || *p > '9' || *(p + 1) < '0' || *(p + 1) > '9') {
189     return NGX_ERROR;
190 }
191
192 min = (*p - '0') * 10 + *(p + 1) - '0';
193 p += 2;
194
195 if (*p++ != ':') {
196     return NGX_ERROR;
197 }
198
199 if (*p < '0' || *p > '9' || *(p + 1) < '0' || *(p + 1) > '9') {
200     return NGX_ERROR;
201 }
202
203 sec = (*p - '0') * 10 + *(p + 1) - '0';
204
205 if (fmt == isoc) {
206     p += 2;
207
208     if (*p++ != ' ') {
209         return NGX_ERROR;

```

```

210     }
211
212     if (*p < '0' || *p > '9' || *(p + 1) < '0' || *(p + 1) > '9'
213         || *(p + 2) < '0' || *(p + 2) > '9'
214         || *(p + 3) < '0' || *(p + 3) > '9')
215     {
216         return NGX\_ERROR;
217     }
218
219     year = (*p - '0') * 1000 + *(p + 1) - '0' * 100
220         + *(p + 2) - '0' * 10 + *(p + 3) - '0';
221 }
222
223 if (hour > 23 || min > 59 || sec > 59) {
224     return NGX\_ERROR;
225 }
226
227 if (day == 29 && month == 1) {
228     if ((year & 3) || ((year % 100 == 0) && (year % 400) != 0)) {
229         return NGX\_ERROR;
230     }
231 }
232 } else if (day > mday[month]) {
233     return NGX\_ERROR;
234 }
235
236 /*
237  * shift new year to March 1 and start months from 1 (not 0),
238  * it is needed for Gauss' formula
239  */
240
241 if (--month <= 0) {
242     month += 12;
243     year -= 1;
244 }
245
246 /* Gauss' formula for Gregorian days since March 1, 1 BC */
247
248 time = (uint64\_t) (
249     /* days in years including leap years since March 1, 1 BC */
250
251     365 * year + year / 4 - year / 100 + year / 400
252
253     /* days before the month */
254
255     + 367 * month / 12 - 30
256
257     /* days before the day */
258
259     + day - 1
260
261     /*
262      * 719527 days were between March 1, 1 BC and March 1, 1970,
263      * 31 and 28 days were in January and February 1970
264      */
265
266     - 719527 + 31 + 28) * 86400 + hour * 3600 + min * 60 + sec;
267
268 #if (NGX\_TIME\_T\_SIZE <= 4)
269
270     if (time > 0x7fffffff) {
271         return NGX\_ERROR;
272     }
273
274 #endif
275
276     return (time\_t) time;
277 }

```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_regex.c - nginx-1.7.10

### Global variables defined

- [ngx\\_pcre\\_pool](#)
- [ngx\\_pcre\\_studies](#)
- [ngx\\_regex\\_commands](#)
- [ngx\\_regex\\_module](#)
- [ngx\\_regex\\_module\\_ctx](#)
- [ngx\\_regex\\_pcre\\_jit\\_post](#)

### Data types defined

- [ngx\\_regex\\_conf\\_t](#)

### Functions defined

- [ngx\\_pcre\\_free\\_studies](#)
- [ngx\\_regex\\_compile](#)
- [ngx\\_regex\\_create\\_conf](#)
- [ngx\\_regex\\_exec\\_array](#)
- [ngx\\_regex\\_free](#)
- [ngx\\_regex\\_init](#)
- [ngx\\_regex\\_init\\_conf](#)
- [ngx\\_regex\\_malloc](#)
- [ngx\\_regex\\_malloc\\_done](#)
- [ngx\\_regex\\_malloc\\_init](#)
- [ngx\\_regex\\_module\\_init](#)
- [ngx\\_regex\\_pcre\\_jit](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10
11
12 typedef struct {
13     ngx_flag_t pcre_jit;
14 } ngx_regex_conf_t;
```

```

15
16
17 static void * ngx_libc cdecl ngx_regex_malloc(size_t size);
18 static void ngx_libc cdecl ngx_regex_free(void *p);
19 #if (NGX_HAVE_PCRE_JIT)
20 static void ngx_pcre_free_studies(void *data);
21 #endif
22
23 static ngx_int_t ngx_regex_module_init(ngx_cycle_t *cycle);
24
25 static void *ngx_regex_create_conf(ngx_cycle_t *cycle);
26 static char *ngx_regex_init_conf(ngx_cycle_t *cycle, void *conf);
27
28 static char *ngx_regex_pcre_jit(ngx_conf_t *cf, void *post, void *data);
29 static ngx_conf_post_t ngx_regex_pcre_jit_post = { ngx_regex_pcre_jit };
30
31
32 static ngx_command_t ngx_regex_commands[] = {
33
34     { ngx_string("pcre_jit"),
35       NGX_MAIN_CONF|NGX_DIRECT_CONF|NGX_CONF_TAKE1,
36       ngx_conf_set_flag_slot,
37       0,
38       offsetof(ngx_regex_conf_t, pcre_jit),
39       &ngx_regex_pcre_jit_post },
40
41     ngx_null_command
42 };
43
44
45 static ngx_core_module_t ngx_regex_module_ctx = {
46     ngx_string("regex"),
47     ngx_regex_create_conf,
48     ngx_regex_init_conf
49 };
50
51
52 ngx_module_t ngx_regex_module = {
53     NGX_MODULE_V1,
54     &ngx_regex_module_ctx,          /* module context */
55     ngx_regex_commands,            /* module directives */
56     NGX_CORE_MODULE,               /* module type */
57     NULL,                            /* init master */
58     ngx_regex_module_init,          /* init module */
59     NULL,                            /* init process */
60     NULL,                            /* init thread */
61     NULL,                            /* exit thread */
62     NULL,                            /* exit process */
63     NULL,                            /* exit master */
64     NGX_MODULE_V1_PADDING
65 };
66
67
68 static ngx_pool_t *ngx_pcre_pool;
69 static ngx_list_t *ngx_pcre_studies;
70
71
72 void
73 ngx_regex_init(void)
74 {
75     pcre_malloc = ngx_regex_malloc;
76     pcre_free = ngx_regex_free;
77 }
78
79
80 static ngx_inline void
81 ngx_regex_malloc_init(ngx_pool_t *pool)
82 {
83     #if (NGX_THREADS)
84         ngx_core_tls_t *tls;
85
86         if (ngx_threaded) {
87             tls = ngx_thread_get_tls(ngx_core_tls_key);
88             tls->pool = pool;
89             return;
90         }

```



```

91
92 #endif
93
94     ngx_pcre_pool = pool;
95 }
96
97
98 static ngx_inline void
99 ngx_regex_malloc_done(void)
100 {
101     #if (NGX_THREADS)
102         ngx_core_tls_t *tls;
103
104         if (ngx_threaded) {
105             tls = ngx_thread_get_tls(ngx_core_tls_key);
106             tls->pool = NULL;
107             return;
108         }
109
110     #endif
111
112     ngx_pcre_pool = NULL;
113 }
114
115
116 ngx_int_t
117 ngx_regex_compile(ngx_regex_compile_t *rc)
118 {
119     int             n, erroff;
120     char           *p;
121     pcre           *re;
122     const char     *errstr;
123     ngx_regex_elt_t *elt;
124
125     ngx_regex_malloc_init(rc->pool);
126
127     re = pcre_compile((const char *) rc->pattern.data, (int) rc->options,
128                     &errstr, &erroff, NULL);
129
130     /* ensure that there is no current pool */
131     ngx_regex_malloc_done();
132
133     if (re == NULL) {
134         if ((size_t) erroff == rc->pattern.len) {
135             rc->err.len = ngx_snprintf(rc->err.data, rc->err.len,
136                                     "pcre_compile() failed: %s in \"%V\"",
137                                     errstr, &rc->pattern)
138                 - rc->err.data;
139         } else {
140             rc->err.len = ngx_snprintf(rc->err.data, rc->err.len,
141                                     "pcre_compile() failed: %s in \"%V\" at \"%s\"",
142                                     errstr, &rc->pattern, rc->pattern.data + erroff)
143                 - rc->err.data;
144         }
145     }
146
147     return NGX_ERROR;
148 }
149
150 rc->regex = ngx_pccalloc(rc->pool, sizeof(ngx_regex_t));
151 if (rc->regex == NULL) {
152     goto nomem;
153 }
154
155 rc->regex->code = re;
156
157 /* do not study at runtime */
158
159 if (ngx_pcre_studies != NULL) {
160     elt = ngx_list_push(ngx_pcre_studies);
161     if (elt == NULL) {
162         goto nomem;
163     }
164
165     elt->regex = rc->regex;
166     elt->name = rc->pattern.data;

```

```

167     }
168
169     n = pcre_fullinfo(re, NULL, PCRE_INFO_CAPTURECOUNT, &rc->captures);
170     if (n < 0) {
171         p = "pcre_fullinfo(\"%V\", PCRE_INFO_CAPTURECOUNT) failed: %d";
172         goto failed;
173     }
174
175     if (rc->captures == 0) {
176         return NGX_OK;
177     }
178
179     n = pcre_fullinfo(re, NULL, PCRE_INFO_NAMECOUNT, &rc->named_captures);
180     if (n < 0) {
181         p = "pcre_fullinfo(\"%V\", PCRE_INFO_NAMECOUNT) failed: %d";
182         goto failed;
183     }
184
185     if (rc->named_captures == 0) {
186         return NGX_OK;
187     }
188
189     n = pcre_fullinfo(re, NULL, PCRE_INFO_NAMEENTRYSIZE, &rc->name_size);
190     if (n < 0) {
191         p = "pcre_fullinfo(\"%V\", PCRE_INFO_NAMEENTRYSIZE) failed: %d";
192         goto failed;
193     }
194
195     n = pcre_fullinfo(re, NULL, PCRE_INFO_NAMETABLE, &rc->names);
196     if (n < 0) {
197         p = "pcre_fullinfo(\"%V\", PCRE_INFO_NAMETABLE) failed: %d";
198         goto failed;
199     }
200
201     return NGX_OK;
202
203 failed:
204
205     rc->err.len = ngx_snprintf(rc->err.data, rc->err.len, p, &rc->pattern, n)
206                 - rc->err.data;
207     return NGX_ERROR;
208
209 nomem:
210
211     rc->err.len = ngx_snprintf(rc->err.data, rc->err.len,
212                             "regex \"%V\" compilation failed: no memory",
213                             &rc->pattern)
214                 - rc->err.data;
215     return NGX_ERROR;
216 }
217
218
219 ngx_int_t
220 ngx_regex_exec_array(ngx_array_t *a, ngx_str_t *s, ngx_log_t *log)
221 {
222     ngx_int_t     n;
223     ngx_uint_t    i;
224     ngx_regex_elt_t *re;
225
226     re = a->elts;
227
228     for (i = 0; i < a->nelts; i++) {
229
230         n = ngx_regex_exec(re[i].regex, s, NULL, 0);
231
232         if (n == NGX_REGEX_NO_MATCHED) {
233             continue;
234         }
235
236         if (n < 0) {
237             ngx_log_error(NGX_LOG_ALERT, log, 0,
238                         ngx_regex_exec_n " failed: %i on \"%V\" using \"%s\"",
239                         n, s, re[i].name);
240             return NGX_ERROR;
241         }
242

```

```

243     /* match */
244
245     return NGX_OK;
246 }
247
248 return NGX_DECLINED;
249 }
250
251
252 static void * ngx_libc_cdecl
253 ngx_regex_malloc(size_t size)
254 {
255     ngx_pool_t      *pool;
256 #if (NGX_THREADS)
257     ngx_core_tls_t  *tls;
258
259     if (ngx_threaded) {
260         tls = ngx_thread_get_tls(ngx_core_tls_key);
261         pool = tls->pool;
262     } else {
263         pool = ngx_pcre_pool;
264     }
265 }
266 #else
267     pool = ngx_pcre_pool;
268 #endif
269
270     if (pool) {
271         return ngx_palloc(pool, size);
272     }
273
274     return NULL;
275 }
276
277
278 static void ngx_libc_cdecl
279 ngx_regex_free(void *p)
280 {
281     return;
282 }
283
284 #if (NGX_HAVE_PCRE_JIT)
285
286 static void
287 ngx_pcre_free_studies(void *data)
288 {
289     ngx_list_t  *studies = data;
290
291     ngx_uint_t      i;
292     ngx_list_part_t *part;
293     ngx_regex_elt_t *elts;
294
295     part = &studies->part;
296     elts = part->elts;
297
298     for (i = 0 ; /* void */ ; i++) {
299         if (i >= part->nelts) {
300             if (part->next == NULL) {
301                 break;
302             }
303
304             part = part->next;
305             elts = part->elts;
306             i = 0;
307         }
308
309         if (elts[i].regex->extra != NULL) {
310             pcre_free_study(elts[i].regex->extra);
311         }
312     }
313 }
314
315 }
316
317 }
318

```

```

319
320 #endif
321
322
323 static ngx_int_t
324 ngx_regex_module_init(ngx_cycle_t *cycle)
325 {
326     int                opt;
327     const char        *errstr;
328     ngx_uint_t        i;
329     ngx_list_part_t   *part;
330     ngx_regex_elt_t   *elts;
331
332     opt = 0;
333
334     #if (NGX_HAVE_PCRE_JIT)
335     {
336         ngx_regex_conf_t   *rcf;
337         ngx_pool_cleanup_t   *cfn;
338
339         rcf = (ngx_regex_conf_t *) ngx_get_conf(cycle->conf_ctx, ngx_regex_module);
340
341         if (rcf->pcre_jit) {
342             opt = PCRE_STUDY_JIT_COMPILE;
343
344             /*
345              * The PCRE JIT compiler uses mmap for its executable codes, so we
346              * have to explicitly call the pcre_free_study() function to free
347              * this memory.
348              */
349
350             cfn = ngx_pool_cleanup_add(cycle->pool, 0);
351             if (cfn == NULL) {
352                 return NGX_ERROR;
353             }
354
355             cfn->handler = ngx_pcre_free_studies;
356             cfn->data = ngx_pcre_studies;
357         }
358     }
359     #endif
360
361     ngx_regex_malloc_init(cycle->pool);
362
363     part = &ngx_pcre_studies->part;
364     elts = part->elts;
365
366     for (i = 0 ; /* void */ ; i++) {
367
368         if (i >= part->nelts) {
369             if (part->next == NULL) {
370                 break;
371             }
372
373             part = part->next;
374             elts = part->elts;
375             i = 0;
376         }
377
378         elts[i].regex->extra = pcre_study(elts[i].regex->code, opt, &errstr);
379
380         if (errstr != NULL) {
381             ngx_log_error(NGX_LOG_ALERT, cycle->log, 0,
382                 "pcre_study() failed: %s in \"%s\"",
383                 errstr, elts[i].name);
384         }
385
386     #if (NGX_HAVE_PCRE_JIT)
387         if (opt & PCRE_STUDY_JIT_COMPILE) {
388             int jit, n;
389
390             jit = 0;
391             n = pcre_fullinfo(elts[i].regex->code, elts[i].regex->extra,
392                 PCRE_INFO_JIT, &jit);
393
394             if (n != 0 || jit != 1) {

```

```

395         ngx_log_error(NGX_LOG_INFO, cycle->log, 0,
396             "JIT compiler does not support pattern: \"%s\"",
397             elts[i].name);
398     }
399 }
400 #endif
401 }
402
403 ngx_regex_malloc done();
404
405 ngx_pcre_studies = NULL;
406
407 return NGX_OK;
408 }
409
410
411 static void *
412 ngx_regex_create_conf(ngx_cycle_t *cycle)
413 {
414     ngx_regex_conf_t *rcf;
415
416     rcf = ngx_palloc(cycle->pool, sizeof(ngx_regex_conf_t));
417     if (rcf == NULL) {
418         return NULL;
419     }
420
421     rcf->pcre_jit = NGX_CONF_UNSET;
422
423     ngx_pcre_studies = ngx_list_create(cycle->pool, 8, sizeof(ngx_regex_elt_t));
424     if (ngx_pcre_studies == NULL) {
425         return NULL;
426     }
427
428     return rcf;
429 }
430
431
432 static char *
433 ngx_regex_init_conf(ngx_cycle_t *cycle, void *conf)
434 {
435     ngx_regex_conf_t *rcf = conf;
436
437     ngx_conf_init_value(rcf->pcre_jit, 0);
438
439     return NGX_CONF_OK;
440 }
441
442
443 static char *
444 ngx_regex_pcre_jit(ngx_conf_t *cf, void *post, void *data)
445 {
446     ngx_flag_t *fp = data;
447
448     if (*fp == 0) {
449         return NGX_CONF_OK;
450     }
451
452     #if (NGX_HAVE_PCRE_JIT)
453     {
454         int jit, r;
455
456         jit = 0;
457         r = pcre_config(PCRE_CONFIG_JIT, &jit);
458
459         if (r != 0 || jit != 1) {
460             ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
461                 "PCRE library does not support JIT");
462             *fp = 0;
463         }
464     }
465     #else
466     ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
467         "nginx was built without PCRE JIT support");
468     *fp = 0;
469     #endif
470

```

```
471 return NGX\_CONF\_OK;  
472 }
```

[One Level Up](#)

[Top Level](#)

# src/core/nginx\_regex.h - nginx-1.7.10

## Data types defined

- [ngx\\_regex\\_compile\\_t](#)
- [ngx\\_regex\\_elt\\_t](#)
- [ngx\\_regex\\_t](#)

## Macros defined

- [NGX\\_REGEX\\_CASELESS](#)
- [NGX\\_REGEX\\_NO\\_MATCHED](#)
- [\\_NGX\\_REGEX\\_H\\_INCLUDED](#)
- [ngx\\_regex\\_exec](#)
- [ngx\\_regex\\_exec\\_n](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_REGEX_H_INCLUDED_
9 #define _NGX_REGEX_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15 #include <pcre.h>
16
17
18 #define NGX_REGEX_NO_MATCHED  PCRE_ERROR_NOMATCH  /* -1 */
19
20 #define NGX_REGEX_CASELESS    PCRE_CASELESS
21
22
23 typedef struct {
24     pcre      *code;
25     pcre_extra *extra;
26 } ngx_regex_t;
27
28
29 typedef struct {
30     ngx_str_t  pattern;
31     ngx_pool_t *pool;
32     ngx_int_t  options;
33
34     ngx_regex_t *regex;
35     int         captures;
36     int         named_captures;
37     int         name_size;
38     u_char     *names;
39     ngx_str_t  err;
40 } ngx_regex_compile_t;
41
42
43 typedef struct {
```

```
44     ngx_regex_t *regex;
45     u_char      *name;
46 } ngx_regex_elt_t;
47
48
49 void ngx_regex_init(void);
50 ngx_int_t ngx_regex_compile(ngx_regex_compile_t *rc);
51
52 #define ngx_regex_exec(re, s, captures, size) \
53     pcre_exec(re->code, re->extra, (const char *) (s)->data, (s)->len, 0, 0, \
54     captures, size)
55 #define ngx_regex_exec_n      "pcre_exec()"
56
57 ngx_int_t ngx_regex_exec_array(ngx_array_t *a, ngx_str_t *s, ngx_log_t *log);
58
59
60 #endif /* NGX_REGEX_H_INCLUDED */
```

[One Level Up](#)

[Top Level](#)



# src/core/nginx\_proxy\_protocol.c - nginx-1.7.10

## Functions defined

- [ngx\\_proxy\\_protocol\\_parse](#)

## Source code

```
1
2  /*
3  * Copyright (C) Roman Arutyunyan
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10
11
12  u_char *
13  ngx_proxy_protocol_parse(ngx_connection_t *c, u_char *buf, u_char *last)
14  {
15      size_t  len;
16      u_char  ch, *p, *addr;
17
18      p = buf;
19      len = last - buf;
20
21      if (len < 8 || ngx_strncmp(p, "PROXY ", 6) != 0) {
22          goto invalid;
23      }
24
25      p += 6;
26      len -= 6;
27
28      if (len >= 7 && ngx_strncmp(p, "UNKNOWN", 7) == 0) {
29          ngx_log_debug0(NGX_LOG_DEBUG_CORE, c->log, 0,
30                       "PROXY protocol unknown protocol");
31          p += 7;
32          goto skip;
33      }
34
35      if (len < 5 || ngx_strncmp(p, "TCP", 3) != 0
36          || (p[3] != '4' && p[3] != '6') || p[4] != ' ')
37      {
38          goto invalid;
39      }
40
41      p += 5;
42      addr = p;
43
44      for ( ;; ) {
45          if (p == last) {
46              goto invalid;
47          }
48
49          ch = *p++;
50
51          if (ch == ' ') {
52              break;
53          }
54
55          if (ch != ':' && ch != '.'
56              && (ch < 'a' || ch > 'f')
57              && (ch < 'A' || ch > 'F')
58              && (ch < '0' || ch > '9'))
59          {
60              goto invalid;
61          }
62      }
```

```

63 len = p - addr - 1;
64 c->proxy_protocol_addr.data = ngx_pnalloc(c->pool, len);
65
66
67 if (c->proxy_protocol_addr.data == NULL) {
68     return NULL;
69 }
70
71 ngx_memcpy(c->proxy_protocol_addr.data, addr, len);
72 c->proxy_protocol_addr.len = len;
73
74 ngx_log_debug1(NGX_LOG_DEBUG_CORE, c->log, 0,
75     "PROXY protocol address: \"%V\"", &c->proxy_protocol_addr);
76
77 skip:
78
79 for ( /* void */ ; p < last - 1; p++) {
80     if (p[0] == CR && p[1] == LF) {
81         return p + 2;
82     }
83 }
84
85 invalid:
86
87 ngx_log_error(NGX_LOG_ERR, c->log, 0,
88     "broken header: \"%*s\"", (size_t) (last - buf), buf);
89
90 return NULL;
91 }

```

[One Level Up](#)

[Top Level](#)

# src/core/nginx\_proxy\_protocol.h - nginx-1.7.10

## Macros defined

- [NGX\\_PROXY\\_PROTOCOL\\_MAX\\_HEADER](#)
- [\\_NGX\\_PROXY\\_PROTOCOL\\_H\\_INCLUDED](#)

## Source code

```
1
2 /*
3  * Copyright (C) Roman Arutyunyan
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_PROXY\_PROTOCOL\_H\_INCLUDED
9 #define \_NGX\_PROXY\_PROTOCOL\_H\_INCLUDED
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 #define NGX\_PROXY\_PROTOCOL\_MAX\_HEADER 107
17
18
19 u_char *ngx\_proxy\_protocol\_parse(ngx\_connection\_t *c, u_char *buf,
20     u_char *last);
21
22
23 #endif /* \_NGX\_PROXY\_PROTOCOL\_H\_INCLUDED */
```

## src/http/nginx\_http\_special\_response.c - nginx-1.7.10

### Global variables defined

- [ngx\\_http\\_error\\_301\\_page](#)
- [ngx\\_http\\_error\\_302\\_page](#)
- [ngx\\_http\\_error\\_303\\_page](#)
- [ngx\\_http\\_error\\_307\\_page](#)
- [ngx\\_http\\_error\\_400\\_page](#)
- [ngx\\_http\\_error\\_401\\_page](#)
- [ngx\\_http\\_error\\_402\\_page](#)
- [ngx\\_http\\_error\\_403\\_page](#)
- [ngx\\_http\\_error\\_404\\_page](#)
- [ngx\\_http\\_error\\_405\\_page](#)
- [ngx\\_http\\_error\\_406\\_page](#)
- [ngx\\_http\\_error\\_408\\_page](#)
- [ngx\\_http\\_error\\_409\\_page](#)
- [ngx\\_http\\_error\\_410\\_page](#)
- [ngx\\_http\\_error\\_411\\_page](#)
- [ngx\\_http\\_error\\_412\\_page](#)
- [ngx\\_http\\_error\\_413\\_page](#)
- [ngx\\_http\\_error\\_414\\_page](#)
- [ngx\\_http\\_error\\_415\\_page](#)
- [ngx\\_http\\_error\\_416\\_page](#)
- [ngx\\_http\\_error\\_494\\_page](#)
- [ngx\\_http\\_error\\_495\\_page](#)
- [ngx\\_http\\_error\\_496\\_page](#)
- [ngx\\_http\\_error\\_497\\_page](#)
- [ngx\\_http\\_error\\_500\\_page](#)
- [ngx\\_http\\_error\\_501\\_page](#)
- [ngx\\_http\\_error\\_502\\_page](#)
- [ngx\\_http\\_error\\_503\\_page](#)
- [ngx\\_http\\_error\\_504\\_page](#)
- [ngx\\_http\\_error\\_507\\_page](#)

- [ngx\\_http\\_error\\_full\\_tail](#)
- [ngx\\_http\\_error\\_pages](#)
- [ngx\\_http\\_error\\_tail](#)
- [ngx\\_http\\_get\\_name](#)
- [ngx\\_http\\_msie\\_padding](#)
- [ngx\\_http\\_msie\\_refresh\\_head](#)
- [ngx\\_http\\_msie\\_refresh\\_tail](#)

## Functions defined

- [ngx\\_http\\_clean\\_header](#)
- [ngx\\_http\\_filter\\_finalize\\_request](#)
- [ngx\\_http\\_send\\_error\\_page](#)
- [ngx\\_http\\_send\\_refresh](#)
- [ngx\\_http\\_send\\_special\\_response](#)
- [ngx\\_http\\_special\\_response\\_handler](#)

## Macros defined

- [NGX\\_HTTP\\_LAST\\_2XX](#)
- [NGX\\_HTTP\\_LAST\\_3XX](#)
- [NGX\\_HTTP\\_LAST\\_4XX](#)
- [NGX\\_HTTP\\_LAST\\_5XX](#)
- [NGX\\_HTTP\\_OFF\\_3XX](#)
- [NGX\\_HTTP\\_OFF\\_4XX](#)
- [NGX\\_HTTP\\_OFF\\_5XX](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11 #include <nginx.h>
12
13
14 static ngx_int_t ngx_http_send_error_page(ngx_http_request_t *r,
15     ngx_http_err_page_t *err_page);
16 static ngx_int_t ngx_http_send_special_response(ngx_http_request_t *r,
17     ngx_http_core_loc_conf_t *clcf, ngx_uint_t err);
18 static ngx_int_t ngx_http_send_refresh(ngx_http_request_t *r);
19
20

```

```

21 static u_char ngx_http_error_full_tail[] =
22 "<hr><center>" NGINX\_VER "</center>" CRLF
23 "</body>" CRLF
24 "</html>" CRLF
25 ;
26
27
28 static u_char ngx_http_error_tail[] =
29 "<hr><center>nginx</center>" CRLF
30 "</body>" CRLF
31 "</html>" CRLF
32 ;
33
34
35 static u_char ngx_http_msie_padding[] =
36 "<!-- a padding to disable MSIE and Chrome friendly error page -->" CRLF
37 "<!-- a padding to disable MSIE and Chrome friendly error page -->" CRLF
38 "<!-- a padding to disable MSIE and Chrome friendly error page -->" CRLF
39 "<!-- a padding to disable MSIE and Chrome friendly error page -->" CRLF
40 "<!-- a padding to disable MSIE and Chrome friendly error page -->" CRLF
41 "<!-- a padding to disable MSIE and Chrome friendly error page -->" CRLF
42 ;
43
44
45 static u_char ngx_http_msie_refresh_head[] =
46 "<html><head><meta http-equiv=\"Refresh\" content=\"0; URL=\"";
47
48
49 static u_char ngx_http_msie_refresh_tail[] =
50 "\"></head><body></body></html>" CRLF;
51
52
53 static char ngx_http_error_301_page[] =
54 "<html>" CRLF
55 "<head><title>301 Moved Permanently</title></head>" CRLF
56 "<body bgcolor=\"white\">" CRLF
57 "<center><h1>301 Moved Permanently</h1></center>" CRLF
58 ;
59
60
61 static char ngx_http_error_302_page[] =
62 "<html>" CRLF
63 "<head><title>302 Found</title></head>" CRLF
64 "<body bgcolor=\"white\">" CRLF
65 "<center><h1>302 Found</h1></center>" CRLF
66 ;
67
68
69 static char ngx_http_error_303_page[] =
70 "<html>" CRLF
71 "<head><title>303 See Other</title></head>" CRLF
72 "<body bgcolor=\"white\">" CRLF
73 "<center><h1>303 See Other</h1></center>" CRLF
74 ;
75
76
77 static char ngx_http_error_307_page[] =
78 "<html>" CRLF
79 "<head><title>307 Temporary Redirect</title></head>" CRLF
80 "<body bgcolor=\"white\">" CRLF
81 "<center><h1>307 Temporary Redirect</h1></center>" CRLF
82 ;
83
84
85 static char ngx_http_error_400_page[] =
86 "<html>" CRLF
87 "<head><title>400 Bad Request</title></head>" CRLF
88 "<body bgcolor=\"white\">" CRLF
89 "<center><h1>400 Bad Request</h1></center>" CRLF
90 ;
91
92
93 static char ngx_http_error_401_page[] =
94 "<html>" CRLF
95 "<head><title>401 Authorization Required</title></head>" CRLF
96 "<body bgcolor=\"white\">" CRLF

```

```
97 "<center><h1>401 Authorization Required</h1></center>" CRLF
98 ;
99
100
101 static char ngx_http_error_402_page[] =
102 "<html>" CRLF
103 "<head><title>402 Payment Required</title></head>" CRLF
104 "<body bgcolor=\"white\">" CRLF
105 "<center><h1>402 Payment Required</h1></center>" CRLF
106 ;
107
108
109 static char ngx_http_error_403_page[] =
110 "<html>" CRLF
111 "<head><title>403 Forbidden</title></head>" CRLF
112 "<body bgcolor=\"white\">" CRLF
113 "<center><h1>403 Forbidden</h1></center>" CRLF
114 ;
115
116
117 static char ngx_http_error_404_page[] =
118 "<html>" CRLF
119 "<head><title>404 Not Found</title></head>" CRLF
120 "<body bgcolor=\"white\">" CRLF
121 "<center><h1>404 Not Found</h1></center>" CRLF
122 ;
123
124
125 static char ngx_http_error_405_page[] =
126 "<html>" CRLF
127 "<head><title>405 Not Allowed</title></head>" CRLF
128 "<body bgcolor=\"white\">" CRLF
129 "<center><h1>405 Not Allowed</h1></center>" CRLF
130 ;
131
132
133 static char ngx_http_error_406_page[] =
134 "<html>" CRLF
135 "<head><title>406 Not Acceptable</title></head>" CRLF
136 "<body bgcolor=\"white\">" CRLF
137 "<center><h1>406 Not Acceptable</h1></center>" CRLF
138 ;
139
140
141 static char ngx_http_error_408_page[] =
142 "<html>" CRLF
143 "<head><title>408 Request Time-out</title></head>" CRLF
144 "<body bgcolor=\"white\">" CRLF
145 "<center><h1>408 Request Time-out</h1></center>" CRLF
146 ;
147
148
149 static char ngx_http_error_409_page[] =
150 "<html>" CRLF
151 "<head><title>409 Conflict</title></head>" CRLF
152 "<body bgcolor=\"white\">" CRLF
153 "<center><h1>409 Conflict</h1></center>" CRLF
154 ;
155
156
157 static char ngx_http_error_410_page[] =
158 "<html>" CRLF
159 "<head><title>410 Gone</title></head>" CRLF
160 "<body bgcolor=\"white\">" CRLF
161 "<center><h1>410 Gone</h1></center>" CRLF
162 ;
163
164
165 static char ngx_http_error_411_page[] =
166 "<html>" CRLF
167 "<head><title>411 Length Required</title></head>" CRLF
168 "<body bgcolor=\"white\">" CRLF
169 "<center><h1>411 Length Required</h1></center>" CRLF
170 ;
171
172
```

```
173 static char ngx_http_error_412_page[] =
174 "<html>" CRLF
175 "<head><title>412 Precondition Failed</title></head>" CRLF
176 "<body bgcolor=\"white\">" CRLF
177 "<center><h1>412 Precondition Failed</h1></center>" CRLF
178 ;
179
180
181 static char ngx_http_error_413_page[] =
182 "<html>" CRLF
183 "<head><title>413 Request Entity Too Large</title></head>" CRLF
184 "<body bgcolor=\"white\">" CRLF
185 "<center><h1>413 Request Entity Too Large</h1></center>" CRLF
186 ;
187
188
189 static char ngx_http_error_414_page[] =
190 "<html>" CRLF
191 "<head><title>414 Request-URI Too Large</title></head>" CRLF
192 "<body bgcolor=\"white\">" CRLF
193 "<center><h1>414 Request-URI Too Large</h1></center>" CRLF
194 ;
195
196
197 static char ngx_http_error_415_page[] =
198 "<html>" CRLF
199 "<head><title>415 Unsupported Media Type</title></head>" CRLF
200 "<body bgcolor=\"white\">" CRLF
201 "<center><h1>415 Unsupported Media Type</h1></center>" CRLF
202 ;
203
204
205 static char ngx_http_error_416_page[] =
206 "<html>" CRLF
207 "<head><title>416 Requested Range Not Satisfiable</title></head>" CRLF
208 "<body bgcolor=\"white\">" CRLF
209 "<center><h1>416 Requested Range Not Satisfiable</h1></center>" CRLF
210 ;
211
212
213 static char ngx_http_error_494_page[] =
214 "<html>" CRLF
215 "<head><title>400 Request Header Or Cookie Too Large</title></head>"
216 CRLF
217 "<body bgcolor=\"white\">" CRLF
218 "<center><h1>400 Bad Request</h1></center>" CRLF
219 "<center>Request Header Or Cookie Too Large</center>" CRLF
220 ;
221
222
223 static char ngx_http_error_495_page[] =
224 "<html>" CRLF
225 "<head><title>400 The SSL certificate error</title></head>"
226 CRLF
227 "<body bgcolor=\"white\">" CRLF
228 "<center><h1>400 Bad Request</h1></center>" CRLF
229 "<center>The SSL certificate error</center>" CRLF
230 ;
231
232
233 static char ngx_http_error_496_page[] =
234 "<html>" CRLF
235 "<head><title>400 No required SSL certificate was sent</title></head>"
236 CRLF
237 "<body bgcolor=\"white\">" CRLF
238 "<center><h1>400 Bad Request</h1></center>" CRLF
239 "<center>No required SSL certificate was sent</center>" CRLF
240 ;
241
242
243 static char ngx_http_error_497_page[] =
244 "<html>" CRLF
245 "<head><title>400 The plain HTTP request was sent to HTTPS port</title></head>"
246 CRLF
247 "<body bgcolor=\"white\">" CRLF
248 "<center><h1>400 Bad Request</h1></center>" CRLF
```



```

249 "<center>The plain HTTP request was sent to HTTPS port</center>" CRLF
250 ;
251
252
253 static char ngx_http_error_500_page[] =
254 "<html>" CRLF
255 "<head><title>500 Internal Server Error</title></head>" CRLF
256 "<body bgcolor=\"white\">" CRLF
257 "<center><h1>500 Internal Server Error</h1></center>" CRLF
258 ;
259
260
261 static char ngx_http_error_501_page[] =
262 "<html>" CRLF
263 "<head><title>501 Not Implemented</title></head>" CRLF
264 "<body bgcolor=\"white\">" CRLF
265 "<center><h1>501 Not Implemented</h1></center>" CRLF
266 ;
267
268
269 static char ngx_http_error_502_page[] =
270 "<html>" CRLF
271 "<head><title>502 Bad Gateway</title></head>" CRLF
272 "<body bgcolor=\"white\">" CRLF
273 "<center><h1>502 Bad Gateway</h1></center>" CRLF
274 ;
275
276
277 static char ngx_http_error_503_page[] =
278 "<html>" CRLF
279 "<head><title>503 Service Temporarily Unavailable</title></head>" CRLF
280 "<body bgcolor=\"white\">" CRLF
281 "<center><h1>503 Service Temporarily Unavailable</h1></center>" CRLF
282 ;
283
284
285 static char ngx_http_error_504_page[] =
286 "<html>" CRLF
287 "<head><title>504 Gateway Time-out</title></head>" CRLF
288 "<body bgcolor=\"white\">" CRLF
289 "<center><h1>504 Gateway Time-out</h1></center>" CRLF
290 ;
291
292
293 static char ngx_http_error_507_page[] =
294 "<html>" CRLF
295 "<head><title>507 Insufficient Storage</title></head>" CRLF
296 "<body bgcolor=\"white\">" CRLF
297 "<center><h1>507 Insufficient Storage</h1></center>" CRLF
298 ;
299
300
301 static ngx_str_t ngx_http_error_pages[] = {
302     ngx_null_string,                /* 201, 204 */
303
304     #define NGX_HTTP_LAST_2XX 202
305     #define NGX_HTTP_OFF_3XX (NGX_HTTP_LAST_2XX - 201)
306
307     /* ngx_null_string, */ /* 300 */
308     ngx_string(ngx_http_error_301_page),
309     ngx_string(ngx_http_error_302_page),
310     ngx_string(ngx_http_error_303_page),
311     ngx_null_string,            /* 304 */
312     ngx_null_string,            /* 305 */
313     ngx_null_string,            /* 306 */
314     ngx_string(ngx_http_error_307_page),
315
316     #define NGX_HTTP_LAST_3XX 308
317     #define NGX_HTTP_OFF_4XX (NGX_HTTP_LAST_3XX - 301 + NGX_HTTP_OFF_3XX)
318
319     ngx_string(ngx_http_error_400_page),
320     ngx_string(ngx_http_error_401_page),
321     ngx_string(ngx_http_error_402_page),
322     ngx_string(ngx_http_error_403_page),
323     ngx_string(ngx_http_error_404_page),
324

```

```

325     ngx_string(ngx_http_error_405_page),
326     ngx_string(ngx_http_error_406_page),
327     ngx_null_string, /* 407 */
328     ngx_string(ngx_http_error_408_page),
329     ngx_string(ngx_http_error_409_page),
330     ngx_string(ngx_http_error_410_page),
331     ngx_string(ngx_http_error_411_page),
332     ngx_string(ngx_http_error_412_page),
333     ngx_string(ngx_http_error_413_page),
334     ngx_string(ngx_http_error_414_page),
335     ngx_string(ngx_http_error_415_page),
336     ngx_string(ngx_http_error_416_page),
337
338 #define NGX_HTTP_LAST_4XX 417
339 #define NGX_HTTP_OFF_5XX (NGX_HTTP_LAST_4XX - 400 + NGX_HTTP_OFF_4XX)
340
341     ngx_string(ngx_http_error_494_page), /* 494, request header too large */
342     ngx_string(ngx_http_error_495_page), /* 495, https certificate error */
343     ngx_string(ngx_http_error_496_page), /* 496, https no certificate */
344     ngx_string(ngx_http_error_497_page), /* 497, http to https */
345     ngx_string(ngx_http_error_404_page), /* 498, canceled */
346     ngx_null_string, /* 499, client has closed connection */
347
348     ngx_string(ngx_http_error_500_page),
349     ngx_string(ngx_http_error_501_page),
350     ngx_string(ngx_http_error_502_page),
351     ngx_string(ngx_http_error_503_page),
352     ngx_string(ngx_http_error_504_page),
353     ngx_null_string, /* 505 */
354     ngx_null_string, /* 506 */
355     ngx_string(ngx_http_error_507_page)
356
357 #define NGX_HTTP_LAST_5XX 508
358
359 };
360
361
362 static ngx_str_t ngx_http_get_name = { 3, (u_char *) "GET " };
363
364
365 ngx_int_t
366 ngx_http_special_response_handler(ngx_http_request_t *r, ngx_int_t error)
367 {
368     ngx_uint_t i, err;
369     ngx_http_err_page_t *err_page;
370     ngx_http_core_loc_conf_t *clcf;
371
372     ngx_log_debug3(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
373                  "http special response: %i, \"%V?%V\"",
374                  error, &r->uri, &r->args);
375
376     r->err_status = error;
377
378     if (r->keepalive) {
379         switch (error) {
380             case NGX_HTTP_BAD_REQUEST:
381             case NGX_HTTP_REQUEST_ENTITY_TOO_LARGE:
382             case NGX_HTTP_REQUEST_URI_TOO_LARGE:
383             case NGX_HTTP_TO_HTTPS:
384             case NGX_HTTPS_CERT_ERROR:
385             case NGX_HTTPS_NO_CERT:
386             case NGX_HTTP_INTERNAL_SERVER_ERROR:
387             case NGX_HTTP_NOT_IMPLEMENTED:
388                 r->keepalive = 0;
389         }
390     }
391
392     if (r->lingering_close) {
393         switch (error) {
394             case NGX_HTTP_BAD_REQUEST:
395             case NGX_HTTP_TO_HTTPS:
396             case NGX_HTTPS_CERT_ERROR:
397             case NGX_HTTPS_NO_CERT:
398                 r->lingering_close = 0;
399         }
400     }

```

```

401 r->headers_out.content_type.len = 0;
402
403
404 clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
405
406 if (!r->error_page && clcf->error_pages && r->uri_changes != 0) {
407
408     if (clcf->recursive_error_pages == 0) {
409         r->error_page = 1;
410     }
411
412     err_page = clcf->error_pages->elts;
413
414     for (i = 0; i < clcf->error_pages->nelts; i++) {
415         if (err_page[i].status == error) {
416             return ngx_http_send_error_page(r, &err_page[i]);
417         }
418     }
419 }
420
421 r->expect_tested = 1;
422
423 if (ngx_http_discard_request_body(r) != NGX_OK) {
424     r->keepalive = 0;
425 }
426
427 if (clcf->msie_refresh
428     && r->headers_in.msie
429     && (error == NGX_HTTP_MOVED_PERMANENTLY
430         || error == NGX_HTTP_MOVED_TEMPORARILY))
431 {
432     return ngx_http_send_refresh(r);
433 }
434
435 if (error == NGX_HTTP_CREATED) {
436     /* 201 */
437     err = 0;
438 }
439 else if (error == NGX_HTTP_NO_CONTENT) {
440     /* 204 */
441     err = 0;
442 }
443 else if (error >= NGX_HTTP_MOVED_PERMANENTLY
444         && error < NGX_HTTP_LAST_3XX)
445 {
446     /* 3XX */
447     err = error - NGX_HTTP_MOVED_PERMANENTLY + NGX_HTTP_OFF_3XX;
448 }
449 else if (error >= NGX_HTTP_BAD_REQUEST
450         && error < NGX_HTTP_LAST_4XX)
451 {
452     /* 4XX */
453     err = error - NGX_HTTP_BAD_REQUEST + NGX_HTTP_OFF_4XX;
454 }
455 else if (error >= NGX_HTTP_NGINX_CODES
456         && error < NGX_HTTP_LAST_5XX)
457 {
458     /* 49X, 5XX */
459     err = error - NGX_HTTP_NGINX_CODES + NGX_HTTP_OFF_5XX;
460     switch (error) {
461         case NGX_HTTP_TO_HTTPS:
462         case NGX_HTTPS_CERT_ERROR:
463         case NGX_HTTPS_NO_CERT:
464         case NGX_HTTP_REQUEST_HEADER_TOO_LARGE:
465             r->err_status = NGX_HTTP_BAD_REQUEST;
466             break;
467     }
468 }
469 else {
470     /* unknown code, zero body */
471     err = 0;
472 }
473
474 return ngx_http_send_special_response(r, clcf, err);
475 }
476

```

```

477 ngx_int_t
478 ngx_http_filter_finalize_request(ngx_http_request_t *r, ngx_module_t *m,
479 ngx_int_t error)
480 {
481     void *ctx;
482     ngx_int_t rc;
483
484     ngx_http_clean_header(r);
485
486     ctx = NULL;
487
488     if (m) {
489         ctx = r->ctx[m->ctx_index];
490     }
491
492     /* clear the modules contexts */
493     ngx_memzero(r->ctx, sizeof(void *) * ngx_http_max_module);
494
495     if (m) {
496         r->ctx[m->ctx_index] = ctx;
497     }
498
499     r->filter_finalize = 1;
500
501     rc = ngx_http_special_response_handler(r, error);
502
503     /* NGX_ERROR resets any pending data */
504
505     switch (rc) {
506     case NGX_OK:
507     case NGX_DONE:
508         return NGX_ERROR;
509
510     default:
511         return rc;
512     }
513 }
514
515 void
516 ngx_http_clean_header(ngx_http_request_t *r)
517 {
518     ngx_memzero(&r->headers_out.status,
519                 sizeof(ngx_http_headers_out_t)
520                 - offsetof(ngx_http_headers_out_t, status));
521
522     r->headers_out.headers.part.nelts = 0;
523     r->headers_out.headers.part.next = NULL;
524     r->headers_out.headers.last = &r->headers_out.headers.part;
525
526     r->headers_out.content_length_n = -1;
527     r->headers_out.last_modified_time = -1;
528 }
529
530 static ngx_int_t
531 ngx_http_send_error_page(ngx_http_request_t *r, ngx_http_err_page_t *err_page)
532 {
533     ngx_int_t overwrite;
534     ngx_str_t uri, args;
535     ngx_table_elt_t *location;
536     ngx_http_core_loc_conf_t *clcf;
537
538     overwrite = err_page->overwrite;
539
540     if (overwrite && overwrite != NGX_HTTP_OK) {
541         r->expect_tested = 1;
542     }
543
544     if (overwrite >= 0) {
545         r->err_status = overwrite;
546     }
547
548     if (ngx_http_complex_value(r, &err_page->value, &uri) != NGX_OK) {

```

```

553     return NGX\_ERROR;
554 }
555
556 if (uri.data[0] == '/') {
557     if (err_page->value.lengths) {
558         ngx\_http\_split\_args(r, &uri, &args);
559     } else {
560         args = err_page->args;
561     }
562
563     if (r->method != NGX\_HTTP\_HEAD) {
564         r->method = NGX\_HTTP\_GET;
565         r->method_name = ngx\_http\_get\_name;
566     }
567
568     return ngx\_http\_internal\_redirect(r, &uri, &args);
569 }
570
571 if (uri.data[0] == '@') {
572     return ngx\_http\_named\_location(r, &uri);
573 }
574
575 location = ngx\_list\_push(&r->headers_out.headers);
576
577 if (location == NULL) {
578     return NGX\_ERROR;
579 }
580
581 if (overwrite != NGX\_HTTP\_MOVED\_PERMANENTLY
582     && overwrite != NGX\_HTTP\_MOVED\_TEMPORARILY
583     && overwrite != NGX\_HTTP\_SEE\_OTHER
584     && overwrite != NGX\_HTTP\_TEMPORARY\_REDIRECT)
585 {
586     r->err_status = NGX\_HTTP\_MOVED\_TEMPORARILY;
587 }
588
589 location->hash = 1;
590 ngx\_str\_set(&location->key, "Location");
591 location->value = uri;
592
593 ngx\_http\_clear\_location(r);
594
595 r->headers_out.location = location;
596
597 clcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_core\_module);
598
599 if (clcf->msie_refresh && r->headers_in.msie) {
600     return ngx\_http\_send\_refresh(r);
601 }
602
603 return ngx\_http\_send\_special\_response(r, clcf, r->err_status
604     - NGX\_HTTP\_MOVED\_PERMANENTLY
605     + NGX\_HTTP\_OFF\_3XX);
606 }
607
608
609
610
611 static ngx\_int\_t
612 ngx\_http\_send\_special\_response(ngx\_http\_request\_t *r,
613     ngx\_http\_core\_loc\_conf\_t *clcf, ngx\_uint\_t err)
614 {
615     u_char        *tail;
616     size_t        len;
617     ngx\_int\_t     rc;
618     ngx\_buf\_t    *b;
619     ngx\_uint\_t   msie_padding;
620     ngx\_chain\_t  out[3];
621
622     if (clcf->server_tokens) {
623         len = sizeof(ngx\_http\_error\_full\_tail) - 1;
624         tail = ngx\_http\_error\_full\_tail;
625     } else {
626         len = sizeof(ngx\_http\_error\_tail) - 1;
627         tail = ngx\_http\_error\_tail;
628     }

```

```

629     }
630
631     msie_padding = 0;
632
633     if (ngx_http_error_pages[err].len) {
634         r->headers_out.content_length_n = ngx_http_error_pages[err].len + len;
635         if (clcf->msie_padding
636             && (r->headers_in.msie || r->headers_in.chrome)
637             && r->http_version >= NGX_HTTP_VERSION_10
638             && err >= NGX_HTTP_OFF_4XX)
639             {
640                 r->headers_out.content_length_n +=
641                     sizeof(ngx_http_msie_padding) - 1;
642                 msie_padding = 1;
643             }
644
645         r->headers_out.content_type_len = sizeof("text/html") - 1;
646         ngx_str_set(&r->headers_out.content_type, "text/html");
647         r->headers_out.content_type_lowcase = NULL;
648     } else {
649         r->headers_out.content_length_n = 0;
650     }
651 }
652
653 if (r->headers_out.content_length) {
654     r->headers_out.content_length->hash = 0;
655     r->headers_out.content_length = NULL;
656 }
657
658 ngx_http_clear_accept_ranges(r);
659 ngx_http_clear_last_modified(r);
660 ngx_http_clear_etag(r);
661
662 rc = ngx_http_send_header(r);
663
664 if (rc == NGX_ERROR || r->header_only) {
665     return rc;
666 }
667
668 if (ngx_http_error_pages[err].len == 0) {
669     return ngx_http_send_special(r, NGX_HTTP_LAST);
670 }
671
672 b = ngx_calloc_buf(r->pool);
673 if (b == NULL) {
674     return NGX_ERROR;
675 }
676
677 b->memory = 1;
678 b->pos = ngx_http_error_pages[err].data;
679 b->last = ngx_http_error_pages[err].data + ngx_http_error_pages[err].len;
680
681 out[0].buf = b;
682 out[0].next = &out[1];
683
684 b = ngx_calloc_buf(r->pool);
685 if (b == NULL) {
686     return NGX_ERROR;
687 }
688
689 b->memory = 1;
690
691 b->pos = tail;
692 b->last = tail + len;
693
694 out[1].buf = b;
695 out[1].next = NULL;
696
697 if (msie_padding) {
698     b = ngx_calloc_buf(r->pool);
699     if (b == NULL) {
700         return NGX_ERROR;
701     }
702
703     b->memory = 1;
704     b->pos = ngx_http_msie_padding;

```

```

705     b->last = ngx_http_msie_padding + sizeof(ngx_http_msie_padding) - 1;
706
707     out[1].next = &out[2];
708     out[2].buf = b;
709     out[2].next = NULL;
710 }
711
712 if (r == r->main) {
713     b->last_buf = 1;
714 }
715
716 b->last_in_chain = 1;
717
718 return ngx_http_output_filter(r, &out[0]);
719 }
720
721
722 static ngx_int_t
723 ngx_http_send_refresh(ngx_http_request_t *r)
724 {
725     u_char      *p, *location;
726     size_t      len, size;
727     uintptr_t   escape;
728     ngx_int_t   rc;
729     ngx_buf_t   *b;
730     ngx_chain_t out;
731
732     len = r->headers_out.location->value.len;
733     location = r->headers_out.location->value.data;
734
735     escape = 2 * ngx_escape_uri(NULL, location, len, NGX_ESCAPE_REFRESH);
736
737     size = sizeof(ngx_http_msie_refresh_head) - 1
738           + escape + len
739           + sizeof(ngx_http_msie_refresh_tail) - 1;
740
741     r->err_status = NGX_HTTP_OK;
742
743     r->headers_out.content_type_len = sizeof("text/html") - 1;
744     ngx_str_set(&r->headers_out.content_type, "text/html");
745     r->headers_out.content_type_lowercase = NULL;
746
747     r->headers_out.location->hash = 0;
748     r->headers_out.location = NULL;
749
750     r->headers_out.content_length_n = size;
751
752     if (r->headers_out.content_length) {
753         r->headers_out.content_length->hash = 0;
754         r->headers_out.content_length = NULL;
755     }
756
757     ngx_http_clear_accept_ranges(r);
758     ngx_http_clear_last_modified(r);
759     ngx_http_clear_etag(r);
760
761     rc = ngx_http_send_header(r);
762
763     if (rc == NGX_ERROR || r->header_only) {
764         return rc;
765     }
766
767     b = ngx_create_temp_buf(r->pool, size);
768     if (b == NULL) {
769         return NGX_ERROR;
770     }
771
772     p = ngx_cpymem(b->pos, ngx_http_msie_refresh_head,
773                  sizeof(ngx_http_msie_refresh_head) - 1);
774
775     if (escape == 0) {
776         p = ngx_cpymem(p, location, len);
777     } else {
778         p = (u_char *) ngx_escape_uri(p, location, len, NGX_ESCAPE_REFRESH);
779     }
780 }

```

```
781
782 b->last = ngx_cpymem(p, ngx_http_msie_refresh_tail,
783                     sizeof(ngx_http_msie_refresh_tail) - 1);
784
785 b->last_buf = 1;
786 b->last_in_chain = 1;
787
788 out.buf = b;
789 out.next = NULL;
790
791 return ngx_http_output_filter(r, &out);
792 }
```

[One Level Up](#)

[Top Level](#)





```

25     NULL,                                /* create server configuration */
26     NULL,                                /* merge server configuration */
27
28     NULL,                                /* create location configuration */
29     NULL,                                /* merge location configuration */
30 };
31
32
33 ngx_module_t ngx_http_header_filter_module = {
34     NGX_MODULE_V1,
35     &ngx_http_header_filter_module_ctx, /* module context */
36     NULL,                                /* module directives */
37     NGX_HTTP_MODULE,                    /* module type */
38     NULL,                                /* init master */
39     NULL,                                /* init module */
40     NULL,                                /* init process */
41     NULL,                                /* init thread */
42     NULL,                                /* exit thread */
43     NULL,                                /* exit process */
44     NULL,                                /* exit master */
45     NGX_MODULE_V1_PADDING
46 };
47
48
49 static char ngx_http_server_string[] = "Server: nginx" CRLF;
50 static char ngx_http_server_full_string[] = "Server: " NGINX_VER CRLF;
51
52
53 static ngx_str_t ngx_http_status_lines[] = {
54
55     ngx_string("200 OK"),
56     ngx_string("201 Created"),
57     ngx_string("202 Accepted"),
58     ngx_null_string, /* "203 Non-Authoritative Information" */
59     ngx_string("204 No Content"),
60     ngx_null_string, /* "205 Reset Content" */
61     ngx_string("206 Partial Content"),
62
63     /* ngx_null_string, */ /* "207 Multi-Status" */
64
65     #define NGX_HTTP_LAST_2XX 207
66     #define NGX_HTTP_OFF_3XX (NGX_HTTP_LAST_2XX - 200)
67
68     /* ngx_null_string, */ /* "300 Multiple Choices" */
69
70     ngx_string("301 Moved Permanently"),
71     ngx_string("302 Moved Temporarily"),
72     ngx_string("303 See Other"),
73     ngx_string("304 Not Modified"),
74     ngx_null_string, /* "305 Use Proxy" */
75     ngx_null_string, /* "306 unused" */
76     ngx_string("307 Temporary Redirect"),
77
78     #define NGX_HTTP_LAST_3XX 308
79     #define NGX_HTTP_OFF_4XX (NGX_HTTP_LAST_3XX - 301 + NGX_HTTP_OFF_3XX)
80
81     ngx_string("400 Bad Request"),
82     ngx_string("401 Unauthorized"),
83     ngx_string("402 Payment Required"),
84     ngx_string("403 Forbidden"),
85     ngx_string("404 Not Found"),
86     ngx_string("405 Not Allowed"),
87     ngx_string("406 Not Acceptable"),
88     ngx_null_string, /* "407 Proxy Authentication Required" */
89     ngx_string("408 Request Time-out"),
90     ngx_string("409 Conflict"),
91     ngx_string("410 Gone"),
92     ngx_string("411 Length Required"),
93     ngx_string("412 Precondition Failed"),
94     ngx_string("413 Request Entity Too Large"),
95     ngx_string("414 Request-URI Too Large"),
96     ngx_string("415 Unsupported Media Type"),
97     ngx_string("416 Requested Range Not Satisfiable"),
98
99     /* ngx_null_string, */ /* "417 Expectation Failed" */
100    /* ngx_null_string, */ /* "418 unused" */

```

```

101 /* ngx_null_string, */ /* "419 unused" */
102 /* ngx_null_string, */ /* "420 unused" */
103 /* ngx_null_string, */ /* "421 unused" */
104 /* ngx_null_string, */ /* "422 Unprocessable Entity" */
105 /* ngx_null_string, */ /* "423 Locked" */
106 /* ngx_null_string, */ /* "424 Failed Dependency" */
107
108 #define NGX_HTTP_LAST_4XX 417
109 #define NGX_HTTP_OFF_5XX (NGX_HTTP_LAST_4XX - 400 + NGX_HTTP_OFF_4XX)
110
111 ngx_string("500 Internal Server Error"),
112 ngx_string("501 Not Implemented"),
113 ngx_string("502 Bad Gateway"),
114 ngx_string("503 Service Temporarily Unavailable"),
115 ngx_string("504 Gateway Time-out"),
116
117 ngx_null_string, /* "505 HTTP Version Not Supported" */
118 ngx_null_string, /* "506 Variant Also Negotiates" */
119 ngx_string("507 Insufficient Storage"),
120 /* ngx_null_string, */ /* "508 unused" */
121 /* ngx_null_string, */ /* "509 unused" */
122 /* ngx_null_string, */ /* "510 Not Extended" */
123
124 #define NGX_HTTP_LAST_5XX 508
125
126 };
127
128
129 ngx_http_header_out_t ngx_http_headers_out[] = {
130 { ngx_string("Server"), offsetof(ngx_http_headers_out_t, server) },
131 { ngx_string("Date"), offsetof(ngx_http_headers_out_t, date) },
132 { ngx_string("Content-Length"),
133   offsetof(ngx_http_headers_out_t, content_length) },
134 { ngx_string("Content-Encoding"),
135   offsetof(ngx_http_headers_out_t, content_encoding) },
136 { ngx_string("Location"), offsetof(ngx_http_headers_out_t, location) },
137 { ngx_string("Last-Modified"),
138   offsetof(ngx_http_headers_out_t, last_modified) },
139 { ngx_string("Accept-Ranges"),
140   offsetof(ngx_http_headers_out_t, accept_ranges) },
141 { ngx_string("Expires"), offsetof(ngx_http_headers_out_t, expires) },
142 { ngx_string("Cache-Control"),
143   offsetof(ngx_http_headers_out_t, cache_control) },
144 { ngx_string("ETag"), offsetof(ngx_http_headers_out_t, etag) },
145
146 { ngx_null_string, 0 }
147 };
148
149
150 static ngx_int_t
151 ngx_http_header_filter(ngx_http_request_t *r)
152 {
153     u_char          *p;
154     size_t          len;
155     ngx_str_t       host, *status_line;
156     ngx_buf_t       *b;
157     ngx_uint_t      status, i, port;
158     ngx_chain_t     out;
159     ngx_list_part_t *part;
160     ngx_table_elt_t *header;
161     ngx_connection_t *c;
162     ngx_http_core_loc_conf_t *clcf;
163     ngx_http_core_srv_conf_t *cscf;
164     struct sockaddr_in *sin;
165     #if (NGX_HAVE_INET6)
166     struct sockaddr_in6 *sin6;
167     #endif
168     u_char          addr[NGX_SOCKADDR_STRLEN];
169
170     if (r->header_sent) {
171         return NGX_OK;
172     }
173
174     r->header_sent = 1;
175
176     if (r != r->main) {

```

```

177     return NGX\_OK;
178 }
179
180 if (r->http_version < NGX\_HTTP\_VERSION\_10) {
181     return NGX\_OK;
182 }
183
184 if (r->method == NGX\_HTTP\_HEAD) {
185     r->header_only = 1;
186 }
187
188 if (r->headers_out.last_modified_time != -1) {
189     if (r->headers_out.status != NGX\_HTTP\_OK
190         && r->headers_out.status != NGX\_HTTP\_PARTIAL\_CONTENT
191         && r->headers_out.status != NGX\_HTTP\_NOT\_MODIFIED)
192     {
193         r->headers_out.last_modified_time = -1;
194         r->headers_out.last_modified = NULL;
195     }
196 }
197
198 len = sizeof("HTTP/1.x ") - 1 + sizeof(CRLF) - 1
199     /* the end of the header */
200     + sizeof(CRLF) - 1;
201
202     /* status line */
203
204     if (r->headers_out.status_line.len) {
205         len += r->headers_out.status_line.len;
206         status_line = &r->headers_out.status_line;
207 #if (NGX\_SUPPRESS\_WARN)
208         status = 0;
209 #endif
210
211     } else {
212
213         status = r->headers_out.status;
214
215         if (status >= NGX\_HTTP\_OK
216             && status < NGX\_HTTP\_LAST\_2XX)
217         {
218             /* 2XX */
219
220             if (status == NGX\_HTTP\_NO\_CONTENT) {
221                 r->header_only = 1;
222                 ngx\_str\_null(&r->headers_out.content_type);
223                 r->headers_out.last_modified_time = -1;
224                 r->headers_out.last_modified = NULL;
225                 r->headers_out.content_length = NULL;
226                 r->headers_out.content_length_n = -1;
227             }
228
229             status -= NGX\_HTTP\_OK;
230             status_line = &ngx\_http\_status\_lines[status];
231             len += ngx\_http\_status\_lines[status].len;
232
233         } else if (status >= NGX\_HTTP\_MOVED\_PERMANENTLY
234                 && status < NGX\_HTTP\_LAST\_3XX)
235         {
236             /* 3XX */
237
238             if (status == NGX\_HTTP\_NOT\_MODIFIED) {
239                 r->header_only = 1;
240             }
241
242             status = status - NGX\_HTTP\_MOVED\_PERMANENTLY + NGX\_HTTP\_OFF\_3XX;
243             status_line = &ngx\_http\_status\_lines[status];
244             len += ngx\_http\_status\_lines[status].len;
245
246         } else if (status >= NGX\_HTTP\_BAD\_REQUEST
247                 && status < NGX\_HTTP\_LAST\_4XX)
248         {
249             /* 4XX */
250             status = status - NGX\_HTTP\_BAD\_REQUEST
251                 + NGX\_HTTP\_OFF\_4XX;
252

```

```

253     status_line = &ngx_http_status_lines[status];
254     len += ngx_http_status_lines[status].len;
255
256 } else if (status >= NGX_HTTP_INTERNAL_SERVER_ERROR
257           && status < NGX_HTTP_LAST_5XX)
258 {
259     /* 5XX */
260     status = status - NGX_HTTP_INTERNAL_SERVER_ERROR
261               + NGX_HTTP_OFF_5XX;
262
263     status_line = &ngx_http_status_lines[status];
264     len += ngx_http_status_lines[status].len;
265
266 } else {
267     len += NGX_INT_T_LEN + 1 /* SP */;
268     status_line = NULL;
269 }
270
271 if (status_line && status_line->len == 0) {
272     status = r->headers_out.status;
273     len += NGX_INT_T_LEN + 1 /* SP */;
274     status_line = NULL;
275 }
276 }
277
278 clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
279
280 if (r->headers_out.server == NULL) {
281     len += clcf->server_tokens ? sizeof(ngx_http_server_full_string) - 1:
282                               sizeof(ngx_http_server_string) - 1;
283 }
284
285 if (r->headers_out.date == NULL) {
286     len += sizeof("Date: Mon, 28 Sep 1970 06:00:00 GMT" CRLF) - 1;
287 }
288
289 if (r->headers_out.content_type.len) {
290     len += sizeof("Content-Type: ") - 1
291           + r->headers_out.content_type.len + 2;
292
293     if (r->headers_out.content_type_len == r->headers_out.content_type.len
294         && r->headers_out.charset.len)
295     {
296         len += sizeof("; charset=") - 1 + r->headers_out.charset.len;
297     }
298 }
299
300 if (r->headers_out.content_length == NULL
301     && r->headers_out.content_length_n >= 0)
302 {
303     len += sizeof("Content-Length: ") - 1 + NGX_OFF_T_LEN + 2;
304 }
305
306 if (r->headers_out.last_modified == NULL
307     && r->headers_out.last_modified_time != -1)
308 {
309     len += sizeof("Last-Modified: Mon, 28 Sep 1970 06:00:00 GMT" CRLF) - 1;
310 }
311
312 c = r->connection;
313
314 if (r->headers_out.location
315     && r->headers_out.location->value.len
316     && r->headers_out.location->value.data[0] == '/')
317 {
318     r->headers_out.location->hash = 0;
319
320     if (clcf->server_name_in_redirect) {
321         cscf = ngx_http_get_module_srv_conf(r, ngx_http_core_module);
322         host = cscf->server_name;
323     }
324     else if (r->headers_in.server.len) {
325         host = r->headers_in.server;
326     }
327     else {
328         host.len = NGX_SOCKADDR_STRLEN;

```

```

329         host.data = addr;
330
331         if (ngx\_connection\_local\_sockaddr(c, &host, 0) != NGX\_OK) {
332             return NGX\_ERROR;
333         }
334     }
335
336     switch (c->local_sockaddr->sa_family) {
337
338     #if (NGX\_HAVE\_INET6)
339         case AF_INET6:
340             sin6 = (struct sockaddr\_in6 \*) c->local_sockaddr;
341             port = ntohs(sin6->sin6_port);
342             break;
343     #endif
344     #if (NGX\_HAVE\_UNIX\_DOMAIN)
345         case AF_UNIX:
346             port = 0;
347             break;
348     #endif
349     default: /* AF\_INET */
350         sin = (struct sockaddr\_in \*) c->local_sockaddr;
351         port = ntohs(sin->sin_port);
352         break;
353     }
354
355     len += sizeof("Location: https://") - 1
356           + host.len
357           + r->headers_out.location->value.len + 2;
358
359     if (clcf->port_in_redirect) {
360
361     #if (NGX\_HTTP\_SSL)
362         if (c->ssl)
363             port = (port == 443) ? 0 : port;
364         else
365     #endif
366             port = (port == 80) ? 0 : port;
367
368     } else {
369         port = 0;
370     }
371
372     if (port) {
373         len += sizeof(":65535") - 1;
374     }
375
376 } else {
377     ngx\_str\_null(&host);
378     port = 0;
379 }
380
381 if (r->chunked) {
382     len += sizeof("Transfer-Encoding: chunked" CRLF) - 1;
383 }
384
385 if (r->headers_out.status == NGX\_HTTP\_SWITCHING\_PROTOCOLS) {
386     len += sizeof("Connection: upgrade" CRLF) - 1;
387
388 } else if (r->keepalive) {
389     len += sizeof("Connection: keep-alive" CRLF) - 1;
390
391     /*
392     * MSIE and Opera ignore the "Keep-Alive: timeout=<N>" header.
393     * MSIE keeps the connection alive for about 60-65 seconds.
394     * Opera keeps the connection alive very long.
395     * Mozilla keeps the connection alive for N plus about 1-10 seconds.
396     * Konqueror keeps the connection alive for about N seconds.
397     */
398
399     if (clcf->keepalive_header) {
400         len += sizeof("Keep-Alive: timeout=") - 1 + NGX\_TIME\_T\_LEN + 2;
401     }
402
403 } else {
404     len += sizeof("Connection: close" CRLF) - 1;

```

```

405     }
406
407     #if (NGX_HTTP_GZIP)
408     if (r->gzip_vary) {
409         if (clcf->gzip_vary) {
410             len += sizeof("Vary: Accept-Encoding" CRLF) - 1;
411
412             } else {
413                 r->gzip_vary = 0;
414             }
415         }
416     #endif
417
418     part = &r->headers_out.headers.part;
419     header = part->elts;
420
421     for (i = 0; /* void */; i++) {
422
423         if (i >= part->nelts) {
424             if (part->next == NULL) {
425                 break;
426             }
427
428             part = part->next;
429             header = part->elts;
430             i = 0;
431         }
432
433         if (header[i].hash == 0) {
434             continue;
435         }
436
437         len += header[i].key.len + sizeof(": ") - 1 + header[i].value.len
438             + sizeof(CRLF) - 1;
439     }
440
441     b = ngx_create_temp_buf(r->pool, len);
442     if (b == NULL) {
443         return NGX_ERROR;
444     }
445
446     /* "HTTP/1.x " */
447     b->last = ngx_cpymem(b->last, "HTTP/1.1 ", sizeof("HTTP/1.x ") - 1);
448
449     /* status line */
450     if (status_line) {
451         b->last = ngx_copy(b->last, status_line->data, status_line->len);
452     }
453     else {
454         b->last = ngx_sprintf(b->last, "%03ui ", status);
455     }
456     *b->last++ = CR; *b->last++ = LF;
457
458     if (r->headers_out.server == NULL) {
459         if (clcf->server_tokens) {
460             p = (u_char *) ngx_http_server_full_string;
461             len = sizeof(ngx_http_server_full_string) - 1;
462
463             } else {
464                 p = (u_char *) ngx_http_server_string;
465                 len = sizeof(ngx_http_server_string) - 1;
466             }
467
468             b->last = ngx_cpymem(b->last, p, len);
469         }
470
471         if (r->headers_out.date == NULL) {
472             b->last = ngx_cpymem(b->last, "Date: ", sizeof("Date: ") - 1);
473             b->last = ngx_cpymem(b->last, ngx_cached_http_time.data,
474                 ngx_cached_http_time.len);
475
476             *b->last++ = CR; *b->last++ = LF;
477         }
478
479         if (r->headers_out.content_type.len) {
480             b->last = ngx_cpymem(b->last, "Content-Type: ",

```

```

481         sizeof("Content-Type: ") - 1);
482     p = b->last;
483     b->last = ngx_copy(b->last, r->headers_out.content_type.data,
484                     r->headers_out.content_type.len);
485
486     if (r->headers_out.content_type_len == r->headers_out.content_type.len
487         && r->headers_out.charset.len)
488     {
489         b->last = ngx_cpymem(b->last, "; charset=",
490                             sizeof("; charset=") - 1);
491         b->last = ngx_copy(b->last, r->headers_out.charset.data,
492                             r->headers_out.charset.len);
493
494         /* update r->headers_out.content_type for possible logging */
495
496         r->headers_out.content_type.len = b->last - p;
497         r->headers_out.content_type.data = p;
498     }
499
500     *b->last++ = CR; *b->last++ = LF;
501 }
502
503 if (r->headers_out.content_length == NULL
504     && r->headers_out.content_length_n >= 0)
505 {
506     b->last = ngx_sprintf(b->last, "Content-Length: %0" CRLF,
507                           r->headers_out.content_length_n);
508 }
509
510 if (r->headers_out.last_modified == NULL
511     && r->headers_out.last_modified_time != -1)
512 {
513     b->last = ngx_cpymem(b->last, "Last-Modified: ",
514                         sizeof("Last-Modified: ") - 1);
515     b->last = ngx_http_time(b->last, r->headers_out.last_modified_time);
516
517     *b->last++ = CR; *b->last++ = LF;
518 }
519
520 if (host.data) {
521
522     p = b->last + sizeof("Location: ") - 1;
523
524     b->last = ngx_cpymem(b->last, "Location: http",
525                         sizeof("Location: http") - 1);
526
527     #if (NGX_HTTP_SSL)
528     if (c->ssl) {
529         *b->last++ = 's';
530     }
531     #endif
532
533     *b->last++ = ':'; *b->last++ = '/'; *b->last++ = '/';
534     b->last = ngx_copy(b->last, host.data, host.len);
535
536     if (port) {
537         b->last = ngx_sprintf(b->last, ":%ui", port);
538     }
539
540     b->last = ngx_copy(b->last, r->headers_out.location->value.data,
541                       r->headers_out.location->value.len);
542
543     /* update r->headers_out.location->value for possible logging */
544
545     r->headers_out.location->value.len = b->last - p;
546     r->headers_out.location->value.data = p;
547     ngx_str_set(&r->headers_out.location->key, "Location");
548
549     *b->last++ = CR; *b->last++ = LF;
550 }
551
552 if (r->chunked) {
553     b->last = ngx_cpymem(b->last, "Transfer-Encoding: chunked" CRLF,
554                         sizeof("Transfer-Encoding: chunked" CRLF) - 1);
555 }
556

```



```

557 if (r->headers_out.status == NGX_HTTP_SWITCHING_PROTOCOLS) {
558     b->last = ngx_cpymem(b->last, "Connection: upgrade" CRLF,
559                         sizeof("Connection: upgrade" CRLF) - 1);
560
561 } else if (r->keepalive) {
562     b->last = ngx_cpymem(b->last, "Connection: keep-alive" CRLF,
563                         sizeof("Connection: keep-alive" CRLF) - 1);
564
565     if (clcf->keepalive_header) {
566         b->last = ngx_sprintf(b->last, "Keep-Alive: timeout=%T" CRLF,
567                               clcf->keepalive_header);
568     }
569
570 } else {
571     b->last = ngx_cpymem(b->last, "Connection: close" CRLF,
572                         sizeof("Connection: close" CRLF) - 1);
573 }
574
575 #if (NGX_HTTP_GZIP)
576 if (r->gzip_vary) {
577     b->last = ngx_cpymem(b->last, "Vary: Accept-Encoding" CRLF,
578                         sizeof("Vary: Accept-Encoding" CRLF) - 1);
579 }
580 #endif
581
582 part = &r->headers_out.headers.part;
583 header = part->elts;
584
585 for (i = 0; /* void */; i++) {
586
587     if (i >= part->nelts) {
588         if (part->next == NULL) {
589             break;
590         }
591
592         part = part->next;
593         header = part->elts;
594         i = 0;
595     }
596
597     if (header[i].hash == 0) {
598         continue;
599     }
600
601     b->last = ngx_copy(b->last, header[i].key.data, header[i].key.len);
602     *b->last++ = ':'; *b->last++ = ' ';
603
604     b->last = ngx_copy(b->last, header[i].value.data, header[i].value.len);
605     *b->last++ = CR; *b->last++ = LF;
606 }
607
608 ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
609               "%*s", (size_t) (b->last - b->pos), b->pos);
610
611 /* the end of HTTP header */
612 *b->last++ = CR; *b->last++ = LF;
613
614 r->header_size = b->last - b->pos;
615
616 if (r->header_only) {
617     b->last_buf = 1;
618 }
619
620 out.buf = b;
621 out.next = NULL;
622
623 return ngx_http_write_filter(r, &out);
624 }
625
626
627 static ngx_int_t
628 ngx_http_header_filter_init(ngx_conf_t *cf)
629 {
630     ngx_http_top_header_filter = ngx_http_header_filter;
631
632     return NGX_OK;

```

[One Level Up](#)

[Top Level](#)

# src/http/nginx\_http\_write\_filter\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_write\\_filter\\_module](#)
- [ngx\\_http\\_write\\_filter\\_module\\_ctx](#)

## Functions defined

- [ngx\\_http\\_write\\_filter](#)
- [ngx\\_http\\_write\\_filter\\_init](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 static ngx_int_t ngx_http_write_filter_init(ngx_conf_t *cf);
14
15
16 static ngx_http_module_t ngx_http_write_filter_module_ctx = {
17     NULL, /* preconfiguration */
18     ngx_http_write_filter_init, /* postconfiguration */
19
20     NULL, /* create main configuration */
21     NULL, /* init main configuration */
22
23     NULL, /* create server configuration */
24     NULL, /* merge server configuration */
25
26     NULL, /* create location configuration */
27     NULL, /* merge location configuration */
28 };
29
30
31 ngx_module_t ngx_http_write_filter_module = {
32     NGX_MODULE_V1,
33     &ngx_http_write_filter_module_ctx, /* module context */
34     NULL, /* module directives */
35     NGX_HTTP_MODULE, /* module type */
36     NULL, /* init master */
37     NULL, /* init module */
38     NULL, /* init process */
39     NULL, /* init thread */
40     NULL, /* exit thread */
41     NULL, /* exit process */
42     NULL, /* exit master */
43     NGX_MODULE_V1_PADDING
44 };
45
46
47 ngx_int_t
48 ngx_http_write_filter(ngx_http_request_t *r, ngx_chain_t *in)
49 {
50     off_t size, sent, nsent, limit;
51     ngx_uint_t last, flush, sync;
52     ngx_msec_t delay;
```

```

53     ngx_chain_t      *cl, *ln, **ll, *chain;
54     ngx_connection_t *c;
55     ngx_http_core_loc_conf_t *clcf;
56
57     c = r->connection;
58
59     if (c->error) {
60         return NGX_ERROR;
61     }
62
63     size = 0;
64     flush = 0;
65     sync = 0;
66     last = 0;
67     ll = &r->out;
68
69     /* find the size, the flush point and the last link of the saved chain */
70
71     for (cl = r->out; cl; cl = cl->next) {
72         ll = &cl->next;
73
74         ngx_log_debug7(NGX_LOG_DEBUG_EVENT, c->log, 0,
75             "write old buf t:%d f:%d %p, pos %p, size: %z "
76             "file: %0, size: %z",
77             cl->buf->temporary, cl->buf->in_file,
78             cl->buf->start, cl->buf->pos,
79             cl->buf->last - cl->buf->pos,
80             cl->buf->file_pos,
81             cl->buf->file_last - cl->buf->file_pos);
82
83     #if 1
84         if (ngx_buf_size(cl->buf) == 0 && !ngx_buf_special(cl->buf)) {
85             ngx_log_error(NGX_LOG_ALERT, c->log, 0,
86                 "zero size buf in writer "
87                 "t:%d r:%d f:%d %p %p-%p %p %0-%0",
88                 cl->buf->temporary,
89                 cl->buf->recycled,
90                 cl->buf->in_file,
91                 cl->buf->start,
92                 cl->buf->pos,
93                 cl->buf->last,
94                 cl->buf->file,
95                 cl->buf->file_pos,
96                 cl->buf->file_last);
97
98             ngx_debug_point();
99             return NGX_ERROR;
100         }
101     #endif
102
103     size += ngx_buf_size(cl->buf);
104
105     if (cl->buf->flush || cl->buf->recycled) {
106         flush = 1;
107     }
108
109     if (cl->buf->sync) {
110         sync = 1;
111     }
112
113     if (cl->buf->last_buf) {
114         last = 1;
115     }
116 }
117
118 /* add the new chain to the existent one */
119
120 for (ln = in; ln; ln = ln->next) {
121     cl = ngx_alloc_chain_link(r->pool);
122     if (cl == NULL) {
123         return NGX_ERROR;
124     }
125
126     cl->buf = ln->buf;
127     *ll = cl;
128     ll = &cl->next;

```

```

129
130     ngx_log_debug7(NGX_LOG_DEBUG_EVENT, c->log, 0,
131         "write new buf t:%d f:%d %p, pos %p, size: %z "
132         "file: %O, size: %z",
133         cl->buf->temporary, cl->buf->in_file,
134         cl->buf->start, cl->buf->pos,
135         cl->buf->last - cl->buf->pos,
136         cl->buf->file_pos,
137         cl->buf->file_last - cl->buf->file_pos);
138
139 #if 1
140     if (ngx_buf_size(cl->buf) == 0 && !ngx_buf_special(cl->buf)) {
141         ngx_log_error(NGX_LOG_ALERT, c->log, 0,
142             "zero size buf in writer "
143             "t:%d r:%d f:%d %p %p-%p %p %O-%O",
144             cl->buf->temporary,
145             cl->buf->recycled,
146             cl->buf->in_file,
147             cl->buf->start,
148             cl->buf->pos,
149             cl->buf->last,
150             cl->buf->file,
151             cl->buf->file_pos,
152             cl->buf->file_last);
153
154         ngx_debug_point();
155         return NGX_ERROR;
156     }
157 #endif
158
159     size += ngx_buf_size(cl->buf);
160
161     if (cl->buf->flush || cl->buf->recycled) {
162         flush = 1;
163     }
164
165     if (cl->buf->sync) {
166         sync = 1;
167     }
168
169     if (cl->buf->last_buf) {
170         last = 1;
171     }
172 }
173
174 *ll = NULL;
175
176 ngx_log_debug3(NGX_LOG_DEBUG_HTTP, c->log, 0,
177     "http write filter: l:%d f:%d s:%O", last, flush, size);
178
179 clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
180
181 /*
182  * avoid the output if there are no last buf, no flush point,
183  * there are the incoming bufs and the size of all bufs
184  * is smaller than "postpone_output" directive
185  */
186
187 if (!last && !flush && in && size < (off_t) clcf->postpone_output) {
188     return NGX_OK;
189 }
190
191 if (c->write->delayed) {
192     c->buffered |= NGX_HTTP_WRITE_BUFFERED;
193     return NGX_AGAIN;
194 }
195
196 if (size == 0
197     && !(c->buffered & NGX_LOWLEVEL_BUFFERED)
198     && !(last && c->need_last_buf))
199 {
200     if (last || flush || sync) {
201         for (cl = r->out; cl; /* void */) {
202             ln = cl;
203             cl = cl->next;
204             ngx_free_chain(r->pool, ln);

```

```

205     }
206
207     r->out = NULL;
208     c->buffered &= ~NGX_HTTP_WRITE_BUFFERED;
209
210     return NGX_OK;
211 }
212
213 ngx_log_error(NGX_LOG_ALERT, c->log, 0,
214     "the http output chain is empty");
215
216 ngx_debug_point();
217
218 return NGX_ERROR;
219 }
220
221 if (r->limit_rate) {
222     if (r->limit_rate_after == 0) {
223         r->limit_rate_after = clcf->limit_rate_after;
224     }
225
226     limit = (off_t) r->limit_rate * (ngx_time() - r->start_sec + 1)
227         - (c->sent - r->limit_rate_after);
228
229     if (limit <= 0) {
230         c->write->delayed = 1;
231         delay = (ngx_msec_t) (- limit * 1000 / r->limit_rate + 1);
232         ngx_add_timer(c->write, delay);
233
234         c->buffered |= NGX_HTTP_WRITE_BUFFERED;
235
236         return NGX_AGAIN;
237     }
238
239     if (clcf->sendfile_max_chunk
240         && (off_t) clcf->sendfile_max_chunk < limit)
241     {
242         limit = clcf->sendfile_max_chunk;
243     }
244 } else {
245     limit = clcf->sendfile_max_chunk;
246 }
247
248 sent = c->sent;
249
250 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, c->log, 0,
251     "http write filter limit %0", limit);
252
253 chain = c->send_chain(c, r->out, limit);
254
255 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, c->log, 0,
256     "http write filter %p", chain);
257
258 if (chain == NGX_CHAIN_ERROR) {
259     c->error = 1;
260     return NGX_ERROR;
261 }
262
263 if (r->limit_rate) {
264     nsent = c->sent;
265
266     if (r->limit_rate_after) {
267         sent -= r->limit_rate_after;
268         if (sent < 0) {
269             sent = 0;
270         }
271
272         nsent -= r->limit_rate_after;
273         if (nsent < 0) {
274             nsent = 0;
275         }
276     }
277 }
278
279 }
280

```

```

281     delay = (ngx\_msec\_t) ((nsent - sent) * 1000 / r->limit_rate);
282
283     if (delay > 0) {
284         limit = 0;
285         c->write->delayed = 1;
286         ngx\_add\_timer(c->write, delay);
287     }
288 }
289
290 if (limit
291     && c->write->ready
292     && c->sent - sent >= limit - (off\_t) (2 * ngx\_pagesize))
293 {
294     c->write->delayed = 1;
295     ngx\_add\_timer(c->write, 1);
296 }
297
298 for (cl = r->out; cl && cl != chain; /* void */) {
299     ln = cl;
300     cl = cl->next;
301     ngx\_free\_chain(r->pool, ln);
302 }
303
304 r->out = chain;
305
306 if (chain) {
307     c->buffered |= NGX\_HTTP\_WRITE\_BUFFERED;
308     return NGX\_AGAIN;
309 }
310
311 c->buffered &= ~NGX\_HTTP\_WRITE\_BUFFERED;
312
313 if ((c->buffered & NGX\_LOWLEVEL\_BUFFERED) && r->postponed == NULL) {
314     return NGX\_AGAIN;
315 }
316
317 return NGX\_OK;
318 }
319
320
321 static ngx\_int\_t
322 ngx\_http\_write\_filter\_init(ngx\_conf\_t *cf)
323 {
324     ngx\_http\_top\_body\_filter = ngx\_http\_write\_filter;
325
326     return NGX\_OK;
327 }

```

[One Level Up](#)

[Top Level](#)

# src/os/unix/nginx\_file\_aio\_read.c - nginx-1.7.10

## Functions defined

- [ngx\\_file\\_aio\\_event\\_handler](#)
- [ngx\\_file\\_aio\\_read](#)
- [ngx\\_file\\_aio\\_result](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 /*
14  * FreeBSD file AIO features and quirks:
15  *
16  *   if an asked data are already in VM cache, then aio_error() returns 0,
17  *   and the data are already copied in buffer;
18  *
19  *   aio_read() pre-read in VM cache as minimum 16K (probably BKVASIZE);
20  *   the first AIO preload may be up to 128K;
21  *
22  *   aio_read/aio_error() may return EINPROGRESS for just written data;
23  *
24  *   kqueue EVFILT_AIO filter is level triggered only: an event repeats
25  *   until aio_return() will be called;
26  *
27  *   aio_cancel() cannot cancel file AIO: it returns AIO_NOTCANCELED always.
28  */
29
30
31 extern int ngx_kqueue;
32
33
34 static ssize_t ngx_file_aio_result(ngx_file_t *file, ngx_event_aio_t *aio,
35     ngx_event_t *ev);
36 static void ngx_file_aio_event_handler(ngx_event_t *ev);
37
38
39 ssize_t
40 ngx_file_aio_read(ngx_file_t *file, u_char *buf, size_t size, off_t offset,
41     ngx_pool_t *pool)
42 {
43     int n;
44     ngx_event_t *ev;
45     ngx_event_aio_t *aio;
46
47     if (!ngx_file_aio) {
48         return ngx_read_file(file, buf, size, offset);
49     }
50
51     aio = file->aio;
52
53     if (aio == NULL) {
54         aio = ngx_palloc(pool, sizeof(ngx_event_aio_t));
55         if (aio == NULL) {
56             return NGX_ERROR;
57         }
58     }
```



```

59     aio->file = file;
60     aio->fd = file->fd;
61     aio->event.data = aio;
62     aio->event.ready = 1;
63     aio->event.log = file->log;
64     #if (NGX_HAVE_AIO_SENDFILE)
65         aio->last_offset = -1;
66     #endif
67     file->aio = aio;
68 }
69
70 ev = &aio->event;
71
72 if (!ev->ready) {
73     ngx_log_error(NGX_LOG_ALERT, file->log, 0,
74                 "second aio post for \"%V\"", &file->name);
75     return NGX_AGAIN;
76 }
77
78 ngx_log_debug4(NGX_LOG_DEBUG_CORE, file->log, 0,
79               "aio complete:%d @%O:%z %V",
80               ev->complete, offset, size, &file->name);
81
82 if (ev->complete) {
83     ev->complete = 0;
84     ngx_set_errno(aio->err);
85
86     if (aio->err == 0) {
87         return aio->nbytes;
88     }
89
90     ngx_log_error(NGX_LOG_CRIT, file->log, ngx_errno,
91                 "aio read \"%s\" failed", file->name.data);
92
93     return NGX_ERROR;
94 }
95
96 ngx_memzero(&aio->aiocb, sizeof(struct aiocb));
97
98 aio->aiocb.aio_fildes = file->fd;
99 aio->aiocb.aio_offset = offset;
100 aio->aiocb.aio_buf = buf;
101 aio->aiocb.aio_nbytes = size;
102 #if (NGX_HAVE_KQUEUE)
103 aio->aiocb.aio_sigevent.sigev_notify_kqueue = ngx_kqueue;
104 aio->aiocb.aio_sigevent.sigev_notify = SIGEV_KEVENT;
105 aio->aiocb.aio_sigevent.sigev_value.sigval_ptr = ev;
106 #endif
107 ev->handler = ngx_file_aio_event_handler;
108
109 n = aio_read(&aio->aiocb);
110
111 if (n == -1) {
112     n = ngx_errno;
113
114     if (n == NGX_EAGAIN) {
115         return ngx_read_file(file, buf, size, offset);
116     }
117
118     ngx_log_error(NGX_LOG_CRIT, file->log, n,
119                 "aio_read(\"%V\") failed", &file->name);
120
121     if (n == NGX_ENOSYS) {
122         ngx_file_aio = 0;
123         return ngx_read_file(file, buf, size, offset);
124     }
125
126     return NGX_ERROR;
127 }
128
129 ngx_log_debug2(NGX_LOG_DEBUG_CORE, file->log, 0,
130               "aio_read: fd:%d %d", file->fd, n);
131
132 ev->active = 1;
133 ev->ready = 0;
134 ev->complete = 0;

```

```

135     return ngx\_file\_aio\_result(aio->file, aio, ev);
136 }
137
138
139
140 static ssize_t
141 ngx\_file\_aio\_result(ngx\_file\_t *file, ngx\_event\_aio\_t *aio, ngx\_event\_t *ev)
142 {
143     int n;
144     ngx\_err\_t err;
145
146     n = aio_error(&aio->aiocb);
147
148     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_CORE, file->log, 0,
149                 "aio_error: fd:%d %d", file->fd, n);
150
151     if (n == -1) {
152         err = ngx\_errno;
153         aio->err = err;
154
155         ngx\_log\_error(NGX\_LOG\_ALERT, file->log, err,
156                     "aio_error(\"%V\") failed", &file->name);
157         return NGX\_ERROR;
158     }
159
160     if (n == NGX\_EINPROGRESS) {
161         if (ev->ready) {
162             ev->ready = 0;
163             ngx\_log\_error(NGX\_LOG\_ALERT, file->log, n,
164                         "aio_read(\"%V\") still in progress",
165                         &file->name);
166         }
167
168         return NGX\_AGAIN;
169     }
170
171     n = aio_return(&aio->aiocb);
172
173     if (n == -1) {
174         err = ngx\_errno;
175         aio->err = err;
176         ev->ready = 1;
177
178         ngx\_log\_error(NGX\_LOG\_CRIT, file->log, err,
179                     "aio_return(\"%V\") failed", &file->name);
180         return NGX\_ERROR;
181     }
182
183     aio->err = 0;
184     aio->nbytes = n;
185     ev->ready = 1;
186     ev->active = 0;
187
188     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_CORE, file->log, 0,
189                 "aio_return: fd:%d %d", file->fd, n);
190
191     return n;
192 }
193
194
195 static void
196 ngx\_file\_aio\_event\_handler(ngx\_event\_t *ev)
197 {
198     ngx\_event\_aio\_t *aio;
199
200     aio = ev->data;
201
202     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_CORE, ev->log, 0,
203                 "aio event handler fd:%d %V", aio->fd, &aio->file->name);
204
205     if (ngx\_file\_aio\_result(aio->file, aio, ev) != NGX\_AGAIN) {
206         aio->handler(ev);
207     }
208 }

```

## src/http/nginx\_http\_upstream.c - nginx-1.7.10

### Global variables defined

- [ngx\\_http\\_upstream\\_cache\\_method\\_mask](#)
- [ngx\\_http\\_upstream\\_commands](#)
- [ngx\\_http\\_upstream\\_headers\\_in](#)
- [ngx\\_http\\_upstream\\_ignore\\_headers\\_masks](#)
- [ngx\\_http\\_upstream\\_module](#)
- [ngx\\_http\\_upstream\\_module\\_ctx](#)
- [ngx\\_http\\_upstream\\_next\\_errors](#)
- [ngx\\_http\\_upstream\\_vars](#)

### Functions defined

- [ngx\\_http\\_upstream](#)
- [ngx\\_http\\_upstream\\_add](#)
- [ngx\\_http\\_upstream\\_add\\_variables](#)
- [ngx\\_http\\_upstream\\_addr\\_variable](#)
- [ngx\\_http\\_upstream\\_bind\\_set\\_slot](#)
- [ngx\\_http\\_upstream\\_cache](#)
- [ngx\\_http\\_upstream\\_cache\\_etag](#)
- [ngx\\_http\\_upstream\\_cache\\_get](#)
- [ngx\\_http\\_upstream\\_cache\\_last\\_modified](#)
- [ngx\\_http\\_upstream\\_cache\\_send](#)
- [ngx\\_http\\_upstream\\_cache\\_status](#)
- [ngx\\_http\\_upstream\\_check\\_broken\\_connection](#)
- [ngx\\_http\\_upstream\\_cleanup](#)
- [ngx\\_http\\_upstream\\_connect](#)
- [ngx\\_http\\_upstream\\_cookie\\_variable](#)
- [ngx\\_http\\_upstream\\_copy\\_allow\\_ranges](#)
- [ngx\\_http\\_upstream\\_copy\\_content\\_encoding](#)
- [ngx\\_http\\_upstream\\_copy\\_content\\_type](#)
- [ngx\\_http\\_upstream\\_copy\\_header\\_line](#)
- [ngx\\_http\\_upstream\\_copy\\_last\\_modified](#)

- [ngx http upstream copy multi header lines](#)
- [ngx http upstream create](#)
- [ngx http upstream create main conf](#)
- [ngx http upstream dummy handler](#)
- [ngx http upstream finalize request](#)
- [ngx http upstream get local](#)
- [ngx http upstream handler](#)
- [ngx http upstream header variable](#)
- [ngx http upstream hide headers hash](#)
- [ngx http upstream ignore header line](#)
- [ngx http upstream init](#)
- [ngx http upstream init main conf](#)
- [ngx http upstream init request](#)
- [ngx http upstream intercept errors](#)
- [ngx http upstream next](#)
- [ngx http upstream non buffered filter](#)
- [ngx http upstream non buffered filter init](#)
- [ngx http upstream param set slot](#)
- [ngx http upstream process accel expires](#)
- [ngx http upstream process body in memory](#)
- [ngx http upstream process buffering](#)
- [ngx http upstream process cache control](#)
- [ngx http upstream process charset](#)
- [ngx http upstream process connection](#)
- [ngx http upstream process content length](#)
- [ngx http upstream process downstream](#)
- [ngx http upstream process expires](#)
- [ngx http upstream process header](#)
- [ngx http upstream process header line](#)
- [ngx http upstream process headers](#)
- [ngx http upstream process last modified](#)
- [ngx http upstream process limit rate](#)
- [ngx http upstream process non buffered downstream](#)

- [ngx http upstream process non buffered request](#)
- [ngx http upstream process non buffered upstream](#)
- [ngx http upstream process request](#)
- [ngx http upstream process set cookie](#)
- [ngx http upstream process transfer encoding](#)
- [ngx http upstream process upgraded](#)
- [ngx http upstream process upstream](#)
- [ngx http upstream process vary](#)
- [ngx http upstream rd check broken connection](#)
- [ngx http upstream reinit](#)
- [ngx http upstream resolve handler](#)
- [ngx http upstream response length variable](#)
- [ngx http upstream response time variable](#)
- [ngx http upstream rewrite location](#)
- [ngx http upstream rewrite refresh](#)
- [ngx http upstream rewrite set cookie](#)
- [ngx http upstream send request](#)
- [ngx http upstream send request handler](#)
- [ngx http upstream send response](#)
- [ngx http upstream server](#)
- [ngx http upstream ssl handshake](#)
- [ngx http upstream ssl init connection](#)
- [ngx http upstream ssl name](#)
- [ngx http upstream status variable](#)
- [ngx http upstream store](#)
- [ngx http upstream test connect](#)
- [ngx http upstream test next](#)
- [ngx http upstream upgrade](#)
- [ngx http upstream upgraded read downstream](#)
- [ngx http upstream upgraded read upstream](#)
- [ngx http upstream upgraded write downstream](#)
- [ngx http upstream upgraded write upstream](#)
- [ngx http upstream wr check broken connection](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 #if (NGX_HTTP_CACHE)
14 static ngx_int_t ngx_http_upstream_cache(ngx_http_request_t *r,
15     ngx_http_upstream_t *u);
16 static ngx_int_t ngx_http_upstream_cache_get(ngx_http_request_t *r,
17     ngx_http_upstream_t *u, ngx_http_file_cache_t **cache);
18 static ngx_int_t ngx_http_upstream_cache_send(ngx_http_request_t *r,
19     ngx_http_upstream_t *u);
20 static ngx_int_t ngx_http_upstream_cache_status(ngx_http_request_t *r,
21     ngx_http_variable_value_t *v, uintptr_t data);
22 static ngx_int_t ngx_http_upstream_cache_last_modified(ngx_http_request_t *r,
23     ngx_http_variable_value_t *v, uintptr_t data);
24 static ngx_int_t ngx_http_upstream_cache_etag(ngx_http_request_t *r,
25     ngx_http_variable_value_t *v, uintptr_t data);
26 #endif
27
28 static void ngx_http_upstream_init_request(ngx_http_request_t *r);
29 static void ngx_http_upstream_resolve_handler(ngx_resolver_ctx_t *ctx);
30 static void ngx_http_upstream_rd_check_broken_connection(ngx_http_request_t *r);
31 static void ngx_http_upstream_wr_check_broken_connection(ngx_http_request_t *r);
32 static void ngx_http_upstream_check_broken_connection(ngx_http_request_t *r,
33     ngx_event_t *ev);
34 static void ngx_http_upstream_connect(ngx_http_request_t *r,
35     ngx_http_upstream_t *u);
36 static ngx_int_t ngx_http_upstream_reinit(ngx_http_request_t *r,
37     ngx_http_upstream_t *u);
38 static void ngx_http_upstream_send_request(ngx_http_request_t *r,
39     ngx_http_upstream_t *u);
40 static void ngx_http_upstream_send_request_handler(ngx_http_request_t *r,
41     ngx_http_upstream_t *u);
42 static void ngx_http_upstream_process_header(ngx_http_request_t *r,
43     ngx_http_upstream_t *u);
44 static ngx_int_t ngx_http_upstream_test_next(ngx_http_request_t *r,
45     ngx_http_upstream_t *u);
46 static ngx_int_t ngx_http_upstream_intercept_errors(ngx_http_request_t *r,
47     ngx_http_upstream_t *u);
48 static ngx_int_t ngx_http_upstream_test_connect(ngx_connection_t *c);
49 static ngx_int_t ngx_http_upstream_process_headers(ngx_http_request_t *r,
50     ngx_http_upstream_t *u);
51 static void ngx_http_upstream_process_body_in_memory(ngx_http_request_t *r,
52     ngx_http_upstream_t *u);
53 static void ngx_http_upstream_send_response(ngx_http_request_t *r,
54     ngx_http_upstream_t *u);
55 static void ngx_http_upstream_upgrade(ngx_http_request_t *r,
56     ngx_http_upstream_t *u);
57 static void ngx_http_upstream_upgraded_read_downstream(ngx_http_request_t *r);
58 static void ngx_http_upstream_upgraded_write_downstream(ngx_http_request_t *r);
59 static void ngx_http_upstream_upgraded_read_upstream(ngx_http_request_t *r,
60     ngx_http_upstream_t *u);
61 static void ngx_http_upstream_upgraded_write_upstream(ngx_http_request_t *r,
62     ngx_http_upstream_t *u);
63 static void ngx_http_upstream_process_upgraded(ngx_http_request_t *r,
64     ngx_uint_t from_upstream, ngx_uint_t do_write);
65 static void
66     ngx_http_upstream_process_non_buffered_downstream(ngx_http_request_t *r);
67 static void
68     ngx_http_upstream_process_non_buffered_upstream(ngx_http_request_t *r,
69     ngx_http_upstream_t *u);
70 static void
71     ngx_http_upstream_process_non_buffered_request(ngx_http_request_t *r,
72     ngx_uint_t do_write);
73 static ngx_int_t ngx_http_upstream_non_buffered_filter_init(void *data);
```

```

74 static ngx_int_t ngx_http_upstream_non_buffered_filter(void *data,
75     ssize_t bytes);
76 static void ngx_http_upstream_process_downstream(ngx_http_request_t *r);
77 static void ngx_http_upstream_process_upstream(ngx_http_request_t *r,
78     ngx_http_upstream_t *u);
79 static void ngx_http_upstream_process_request(ngx_http_request_t *r);
80 static void ngx_http_upstream_store(ngx_http_request_t *r,
81     ngx_http_upstream_t *u);
82 static void ngx_http_upstream_dummy_handler(ngx_http_request_t *r,
83     ngx_http_upstream_t *u);
84 static void ngx_http_upstream_next(ngx_http_request_t *r,
85     ngx_http_upstream_t *u, ngx_uint_t ft_type);
86 static void ngx_http_upstream_cleanup(void *data);
87 static void ngx_http_upstream_finalize_request(ngx_http_request_t *r,
88     ngx_http_upstream_t *u, ngx_int_t rc);
89
90 static ngx_int_t ngx_http_upstream_process_header_line(ngx_http_request_t *r,
91     ngx_table_elt_t *h, ngx_uint_t offset);
92 static ngx_int_t ngx_http_upstream_process_content_length(ngx_http_request_t *r,
93     ngx_table_elt_t *h, ngx_uint_t offset);
94 static ngx_int_t ngx_http_upstream_process_last_modified(ngx_http_request_t *r,
95     ngx_table_elt_t *h, ngx_uint_t offset);
96 static ngx_int_t ngx_http_upstream_process_set_cookie(ngx_http_request_t *r,
97     ngx_table_elt_t *h, ngx_uint_t offset);
98 static ngx_int_t
99     ngx_http_upstream_process_cache_control(ngx_http_request_t *r,
100     ngx_table_elt_t *h, ngx_uint_t offset);
101 static ngx_int_t ngx_http_upstream_ignore_header_line(ngx_http_request_t *r,
102     ngx_table_elt_t *h, ngx_uint_t offset);
103 static ngx_int_t ngx_http_upstream_process_expires(ngx_http_request_t *r,
104     ngx_table_elt_t *h, ngx_uint_t offset);
105 static ngx_int_t ngx_http_upstream_process_accel_expires(ngx_http_request_t *r,
106     ngx_table_elt_t *h, ngx_uint_t offset);
107 static ngx_int_t ngx_http_upstream_process_limit_rate(ngx_http_request_t *r,
108     ngx_table_elt_t *h, ngx_uint_t offset);
109 static ngx_int_t ngx_http_upstream_process_buffering(ngx_http_request_t *r,
110     ngx_table_elt_t *h, ngx_uint_t offset);
111 static ngx_int_t ngx_http_upstream_process_charset(ngx_http_request_t *r,
112     ngx_table_elt_t *h, ngx_uint_t offset);
113 static ngx_int_t ngx_http_upstream_process_connection(ngx_http_request_t *r,
114     ngx_table_elt_t *h, ngx_uint_t offset);
115 static ngx_int_t
116     ngx_http_upstream_process_transfer_encoding(ngx_http_request_t *r,
117     ngx_table_elt_t *h, ngx_uint_t offset);
118 static ngx_int_t ngx_http_upstream_process_vary(ngx_http_request_t *r,
119     ngx_table_elt_t *h, ngx_uint_t offset);
120 static ngx_int_t ngx_http_upstream_copy_header_line(ngx_http_request_t *r,
121     ngx_table_elt_t *h, ngx_uint_t offset);
122 static ngx_int_t
123     ngx_http_upstream_copy_multi_header_lines(ngx_http_request_t *r,
124     ngx_table_elt_t *h, ngx_uint_t offset);
125 static ngx_int_t ngx_http_upstream_copy_content_type(ngx_http_request_t *r,
126     ngx_table_elt_t *h, ngx_uint_t offset);
127 static ngx_int_t ngx_http_upstream_copy_last_modified(ngx_http_request_t *r,
128     ngx_table_elt_t *h, ngx_uint_t offset);
129 static ngx_int_t ngx_http_upstream_rewrite_location(ngx_http_request_t *r,
130     ngx_table_elt_t *h, ngx_uint_t offset);
131 static ngx_int_t ngx_http_upstream_rewrite_refresh(ngx_http_request_t *r,
132     ngx_table_elt_t *h, ngx_uint_t offset);
133 static ngx_int_t ngx_http_upstream_rewrite_set_cookie(ngx_http_request_t *r,
134     ngx_table_elt_t *h, ngx_uint_t offset);
135 static ngx_int_t ngx_http_upstream_copy_allow_ranges(ngx_http_request_t *r,
136     ngx_table_elt_t *h, ngx_uint_t offset);
137
138 #if (NGX_HTTP_GZIP)
139 static ngx_int_t ngx_http_upstream_copy_content_encoding(ngx_http_request_t *r,
140     ngx_table_elt_t *h, ngx_uint_t offset);
141 #endif
142
143 static ngx_int_t ngx_http_upstream_add_variables(ngx_conf_t *cf);
144 static ngx_int_t ngx_http_upstream_addr_variable(ngx_http_request_t *r,
145     ngx_http_variable_value_t *v, uintptr_t data);
146 static ngx_int_t ngx_http_upstream_status_variable(ngx_http_request_t *r,
147     ngx_http_variable_value_t *v, uintptr_t data);
148 static ngx_int_t ngx_http_upstream_response_time_variable(ngx_http_request_t *r,
149     ngx_http_variable_value_t *v, uintptr_t data);

```

```

150 static ngx_int_t ngx_http_upstream_response_length_variable(
151     ngx_http_request_t *r, ngx_http_variable_value_t *v, uintptr_t data);
152
153 static char *ngx_http_upstream(ngx_conf_t *cf, ngx_command_t *cmd, void *dummy);
154 static char *ngx_http_upstream_server(ngx_conf_t *cf, ngx_command_t *cmd,
155     void *conf);
156
157 static ngx_addr_t *ngx_http_upstream_get_local(ngx_http_request_t *r,
158     ngx_http_upstream_local_t *local);
159
160 static void *ngx_http_upstream_create_main_conf(ngx_conf_t *cf);
161 static char *ngx_http_upstream_init_main_conf(ngx_conf_t *cf, void *conf);
162
163 #if (NGX_HTTP_SSL)
164 static void ngx_http_upstream_ssl_init_connection(ngx_http_request_t *,
165     ngx_http_upstream_t *u, ngx_connection_t *c);
166 static void ngx_http_upstream_ssl_handshake(ngx_connection_t *c);
167 static ngx_int_t ngx_http_upstream_ssl_name(ngx_http_request_t *r,
168     ngx_http_upstream_t *u, ngx_connection_t *c);
169 #endif
170
171
172 ngx_http_upstream_header_t ngx_http_upstream_headers_in[] = {
173
174     { ngx_string("Status"),
175         ngx_http_upstream_process_header_line,
176         offsetof(ngx_http_upstream_headers_in_t, status),
177         ngx_http_upstream_copy_header_line, 0, 0 },
178
179     { ngx_string("Content-Type"),
180         ngx_http_upstream_process_header_line,
181         offsetof(ngx_http_upstream_headers_in_t, content_type),
182         ngx_http_upstream_copy_content_type, 0, 1 },
183
184     { ngx_string("Content-Length"),
185         ngx_http_upstream_process_content_length, 0,
186         ngx_http_upstream_ignore_header_line, 0, 0 },
187
188     { ngx_string("Date"),
189         ngx_http_upstream_process_header_line,
190         offsetof(ngx_http_upstream_headers_in_t, date),
191         ngx_http_upstream_copy_header_line,
192         offsetof(ngx_http_headers_out_t, date), 0 },
193
194     { ngx_string("Last-Modified"),
195         ngx_http_upstream_process_last_modified, 0,
196         ngx_http_upstream_copy_last_modified, 0, 0 },
197
198     { ngx_string("ETag"),
199         ngx_http_upstream_process_header_line,
200         offsetof(ngx_http_upstream_headers_in_t, etag),
201         ngx_http_upstream_copy_header_line,
202         offsetof(ngx_http_headers_out_t, etag), 0 },
203
204     { ngx_string("Server"),
205         ngx_http_upstream_process_header_line,
206         offsetof(ngx_http_upstream_headers_in_t, server),
207         ngx_http_upstream_copy_header_line,
208         offsetof(ngx_http_headers_out_t, server), 0 },
209
210     { ngx_string("WWW-Authenticate"),
211         ngx_http_upstream_process_header_line,
212         offsetof(ngx_http_upstream_headers_in_t, www_authenticate),
213         ngx_http_upstream_copy_header_line, 0, 0 },
214
215     { ngx_string("Location"),
216         ngx_http_upstream_process_header_line,
217         offsetof(ngx_http_upstream_headers_in_t, location),
218         ngx_http_upstream_rewrite_location, 0, 0 },
219
220     { ngx_string("Refresh"),
221         ngx_http_upstream_ignore_header_line, 0,
222         ngx_http_upstream_rewrite_refresh, 0, 0 },
223
224     { ngx_string("Set-Cookie"),
225         ngx_http_upstream_process_set_cookie,

```



```

226         offsetof(ngx_http_upstream_headers_in_t, cookies),
227         ngx_http_upstream_rewrite_set_cookie, 0, 1 },
228
229     { ngx_string("Content-Disposition"),
230       ngx_http_upstream_ignore_header_line, 0,
231       ngx_http_upstream_copy_header_line, 0, 1 },
232
233     { ngx_string("Cache-Control"),
234       ngx_http_upstream_process_cache_control, 0,
235       ngx_http_upstream_copy_multi_header_lines,
236       offsetof(ngx_http_headers_out_t, cache_control), 1 },
237
238     { ngx_string("Expires"),
239       ngx_http_upstream_process_expires, 0,
240       ngx_http_upstream_copy_header_line,
241       offsetof(ngx_http_headers_out_t, expires), 1 },
242
243     { ngx_string("Accept-Ranges"),
244       ngx_http_upstream_process_header_line,
245       offsetof(ngx_http_upstream_headers_in_t, accept_ranges),
246       ngx_http_upstream_copy_allow_ranges,
247       offsetof(ngx_http_headers_out_t, accept_ranges), 1 },
248
249     { ngx_string("Connection"),
250       ngx_http_upstream_process_connection, 0,
251       ngx_http_upstream_ignore_header_line, 0, 0 },
252
253     { ngx_string("Keep-Alive"),
254       ngx_http_upstream_ignore_header_line, 0,
255       ngx_http_upstream_ignore_header_line, 0, 0 },
256
257     { ngx_string("Vary"),
258       ngx_http_upstream_process_vary, 0,
259       ngx_http_upstream_copy_header_line, 0, 0 },
260
261     { ngx_string("X-Powered-By"),
262       ngx_http_upstream_ignore_header_line, 0,
263       ngx_http_upstream_copy_header_line, 0, 0 },
264
265     { ngx_string("X-Accel-Expires"),
266       ngx_http_upstream_process_accel_expires, 0,
267       ngx_http_upstream_copy_header_line, 0, 0 },
268
269     { ngx_string("X-Accel-Redirect"),
270       ngx_http_upstream_process_header_line,
271       offsetof(ngx_http_upstream_headers_in_t, x_accel_redirect),
272       ngx_http_upstream_copy_header_line, 0, 0 },
273
274     { ngx_string("X-Accel-Limit-Rate"),
275       ngx_http_upstream_process_limit_rate, 0,
276       ngx_http_upstream_copy_header_line, 0, 0 },
277
278     { ngx_string("X-Accel-Buffering"),
279       ngx_http_upstream_process_buffering, 0,
280       ngx_http_upstream_copy_header_line, 0, 0 },
281
282     { ngx_string("X-Accel-Charset"),
283       ngx_http_upstream_process_charset, 0,
284       ngx_http_upstream_copy_header_line, 0, 0 },
285
286     { ngx_string("Transfer-Encoding"),
287       ngx_http_upstream_process_transfer_encoding, 0,
288       ngx_http_upstream_ignore_header_line, 0, 0 },
289
290     #if (NGX_HTTP_GZIP)
291     { ngx_string("Content-Encoding"),
292       ngx_http_upstream_process_header_line,
293       offsetof(ngx_http_upstream_headers_in_t, content_encoding),
294       ngx_http_upstream_copy_content_encoding, 0, 0 },
295     #endif
296
297     { ngx_null_string, NULL, 0, NULL, 0, 0 }
298 };
299
300
301 static ngx_command_t ngx_http_upstream_commands[] = {

```

```

302     { ngx_string("upstream"),
303       ngx_http_main_conf|ngx_conf_block|ngx_conf_take1,
304       ngx_http_upstream,
305       0,
306       0,
307       NULL },
308
309
310     { ngx_string("server"),
311       ngx_http_ups_conf|ngx_conf_1more,
312       ngx_http_upstream_server,
313       ngx_http_srv_conf_offset,
314       0,
315       NULL },
316
317     ngx_null_command
318 };
319
320
321 static ngx_http_module_t  ngx_http_upstream_module_ctx = {
322     ngx_http_upstream_add_variables,      /* preconfiguration */
323     NULL,                                 /* postconfiguration */
324
325     ngx_http_upstream_create_main_conf,  /* create main configuration */
326     ngx_http_upstream_init_main_conf,    /* init main configuration */
327
328     NULL,                                 /* create server configuration */
329     NULL,                                 /* merge server configuration */
330
331     NULL,                                 /* create location configuration */
332     NULL,                                 /* merge location configuration */
333 };
334
335
336 ngx_module_t  ngx_http_upstream_module = {
337     NGX_MODULE_V1,
338     &ngx_http_upstream_module_ctx,      /* module context */
339     ngx_http_upstream_commands,        /* module directives */
340     NGX_HTTP_MODULE,                   /* module type */
341     NULL,                               /* init master */
342     NULL,                               /* init module */
343     NULL,                               /* init process */
344     NULL,                               /* init thread */
345     NULL,                               /* exit thread */
346     NULL,                               /* exit process */
347     NULL,                               /* exit master */
348     NGX_MODULE_V1_PADDING
349 };
350
351
352 static ngx_http_variable_t  ngx_http_upstream_vars[] = {
353
354     { ngx_string("upstream_addr"), NULL,
355       ngx_http_upstream_addr_variable, 0,
356       NGX_HTTP_VAR_NOCACHEABLE, 0 },
357
358     { ngx_string("upstream_status"), NULL,
359       ngx_http_upstream_status_variable, 0,
360       NGX_HTTP_VAR_NOCACHEABLE, 0 },
361
362     { ngx_string("upstream_header_time"), NULL,
363       ngx_http_upstream_response_time_variable, 1,
364       NGX_HTTP_VAR_NOCACHEABLE, 0 },
365
366     { ngx_string("upstream_response_time"), NULL,
367       ngx_http_upstream_response_time_variable, 0,
368       NGX_HTTP_VAR_NOCACHEABLE, 0 },
369
370     { ngx_string("upstream_response_length"), NULL,
371       ngx_http_upstream_response_length_variable, 0,
372       NGX_HTTP_VAR_NOCACHEABLE, 0 },
373
374     #if (NGX_HTTP_CACHE)
375
376     { ngx_string("upstream_cache_status"), NULL,
377       ngx_http_upstream_cache_status, 0,

```

```

378     NGX\_HTTP\_VAR\_NOCACHEABLE, 0 },
379
380     { ngx\_string\("upstream\_cache\_last\_modified"\), NULL,
381       ngx\_http\_upstream\_cache\_last\_modified, 0,
382       NGX\_HTTP\_VAR\_NOCACHEABLE|NGX\_HTTP\_VAR\_NOHASH, 0 },
383
384     { ngx\_string\("upstream\_cache\_etag"\), NULL,
385       ngx\_http\_upstream\_cache\_etag, 0,
386       NGX\_HTTP\_VAR\_NOCACHEABLE|NGX\_HTTP\_VAR\_NOHASH, 0 },
387
388 #endif
389
390     { ngx\_null\_string, NULL, NULL, 0, 0, 0 }
391 };
392
393
394 static ngx\_http\_upstream\_next\_t ngx\_http\_upstream\_next\_errors[] = {
395     { 500, NGX\_HTTP\_UPSTREAM\_FT\_HTTP\_500 },
396     { 502, NGX\_HTTP\_UPSTREAM\_FT\_HTTP\_502 },
397     { 503, NGX\_HTTP\_UPSTREAM\_FT\_HTTP\_503 },
398     { 504, NGX\_HTTP\_UPSTREAM\_FT\_HTTP\_504 },
399     { 403, NGX\_HTTP\_UPSTREAM\_FT\_HTTP\_403 },
400     { 404, NGX\_HTTP\_UPSTREAM\_FT\_HTTP\_404 },
401     { 0, 0 }
402 };
403
404
405 ngx\_conf\_bitmask\_t ngx\_http\_upstream\_cache\_method\_mask[] = {
406     { ngx\_string\("GET"\), NGX\_HTTP\_GET },
407     { ngx\_string\("HEAD"\), NGX\_HTTP\_HEAD },
408     { ngx\_string\("POST"\), NGX\_HTTP\_POST },
409     { ngx\_null\_string, 0 }
410 };
411
412
413 ngx\_conf\_bitmask\_t ngx\_http\_upstream\_ignore\_headers\_masks[] = {
414     { ngx\_string\("X-Accel-Redirect"\), NGX\_HTTP\_UPSTREAM\_IGN\_XA\_REDIRECT },
415     { ngx\_string\("X-Accel-Expires"\), NGX\_HTTP\_UPSTREAM\_IGN\_XA\_EXPIRES },
416     { ngx\_string\("X-Accel-Limit-Rate"\), NGX\_HTTP\_UPSTREAM\_IGN\_XA\_LIMIT\_RATE },
417     { ngx\_string\("X-Accel-Buffering"\), NGX\_HTTP\_UPSTREAM\_IGN\_XA\_BUFFERING },
418     { ngx\_string\("X-Accel-Charset"\), NGX\_HTTP\_UPSTREAM\_IGN\_XA\_CHARSET },
419     { ngx\_string\("Expires"\), NGX\_HTTP\_UPSTREAM\_IGN\_EXPIRES },
420     { ngx\_string\("Cache-Control"\), NGX\_HTTP\_UPSTREAM\_IGN\_CACHE\_CONTROL },
421     { ngx\_string\("Set-Cookie"\), NGX\_HTTP\_UPSTREAM\_IGN\_SET\_COOKIE },
422     { ngx\_string\("Vary"\), NGX\_HTTP\_UPSTREAM\_IGN\_VARY },
423     { ngx\_null\_string, 0 }
424 };
425
426
427 ngx\_int\_t
428 ngx\_http\_upstream\_create(ngx\_http\_request\_t *r)
429 {
430     ngx\_http\_upstream\_t *u;
431
432     u = r->upstream;
433
434     if (u && u->cleanup) {
435         r->main->count++;
436         ngx\_http\_upstream\_cleanup(r);
437     }
438
439     u = ngx\_palloc(r->pool, sizeof(ngx\_http\_upstream\_t));
440     if (u == NULL) {
441         return NGX\_ERROR;
442     }
443
444     r->upstream = u;
445
446     u->peer.log = r->connection->log;
447     u->peer.log_error = NGX\_ERROR\_ERR;
448 #if (NGX\_THREADS)
449     u->peer.lock = &r->connection->lock;
450 #endif
451
452 #if (NGX\_HTTP\_CACHE)
453     r->cache = NULL;

```

```

454 #endif
455
456     u->headers_in.content_length_n = -1;
457     u->headers_in.last_modified_time = -1;
458
459     return NGX_OK;
460 }
461
462
463 void
464 ngx_http_upstream_init(ngx_http_request_t *r)
465 {
466     ngx_connection_t *c;
467
468     c = r->connection;
469
470     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, c->log, 0,
471                  "http init upstream, client timer: %d", c->read->timer_set);
472
473 #if (NGX_HTTP_SPDY)
474     if (r->spdy_stream) {
475         ngx_http_upstream_init_request(r);
476         return;
477     }
478 #endif
479
480     if (c->read->timer_set) {
481         ngx_del_timer(c->read);
482     }
483
484     if (ngx_event_flags & NGX_USE_CLEAR_EVENT) {
485
486         if (!c->write->active) {
487             if (ngx_add_event(c->write, NGX_WRITE_EVENT, NGX_CLEAR_EVENT)
488                 == NGX_ERROR)
489             {
490                 ngx_http_finalize_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
491                 return;
492             }
493         }
494     }
495
496     ngx_http_upstream_init_request(r);
497 }
498
499
500 static void
501 ngx_http_upstream_init_request(ngx_http_request_t *r)
502 {
503     ngx_str_t *host;
504     ngx_uint_t i;
505     ngx_resolver_ctx_t *ctx, temp;
506     ngx_http_cleanup_t *cln;
507     ngx_http_upstream_t *u;
508     ngx_http_core_loc_conf_t *clcf;
509     ngx_http_upstream_srv_conf_t *uscf, **uscfp;
510     ngx_http_upstream_main_conf_t *umcf;
511
512     if (r->aio) {
513         return;
514     }
515
516     u = r->upstream;
517
518 #if (NGX_HTTP_CACHE)
519
520     if (u->conf->cache) {
521         ngx_int_t rc;
522
523         rc = ngx_http_upstream_cache(r, u);
524
525         if (rc == NGX_BUSY) {
526             r->write_event_handler = ngx_http_upstream_init_request;
527             return;
528         }
529

```

```

530     r->write_event_handler = ngx_http_request_empty_handler;
531
532     if (rc == NGX\_DONE) {
533         return;
534     }
535
536     if (rc == NGX\_ERROR) {
537         ngx\_http\_finalize\_request(r, NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);
538         return;
539     }
540
541     if (rc != NGX\_DECLINED) {
542         ngx\_http\_finalize\_request(r, rc);
543         return;
544     }
545 }
546
547 #endif
548
549 u->store = u->conf->store;
550
551 if (!u->store && !r->post_action && !u->conf->ignore_client_abort) {
552     r->read_event_handler = ngx\_http\_upstream\_rd\_check\_broken\_connection;
553     r->write_event_handler = ngx\_http\_upstream\_wr\_check\_broken\_connection;
554 }
555
556 if (r->request_body) {
557     u->request_bufs = r->request_body->bufs;
558 }
559
560 if (u->create_request(r) != NGX\_OK) {
561     ngx\_http\_finalize\_request(r, NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);
562     return;
563 }
564
565 u->peer.local = ngx\_http\_upstream\_get\_local(r, u->conf->local);
566
567 clcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_core\_module);
568
569 u->output.alignment = clcf->directio_alignment;
570 u->output.pool = r->pool;
571 u->output.bufs.num = 1;
572 u->output.bufs.size = clcf->client_body_buffer_size;
573 u->output.output_filter = ngx\_chain\_writer;
574 u->output.filter_ctx = &u->writer;
575
576 u->writer.pool = r->pool;
577
578 if (r->upstream_states == NULL) {
579
580     r->upstream_states = ngx\_array\_create(r->pool, 1,
581                                         sizeof(ngx\_http\_upstream\_state\_t));
582     if (r->upstream_states == NULL) {
583         ngx\_http\_finalize\_request(r, NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);
584         return;
585     }
586 } else {
587
588     u->state = ngx\_array\_push(r->upstream_states);
589     if (u->state == NULL) {
590         ngx\_http\_upstream\_finalize\_request(r, u,
591                                         NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);
592         return;
593     }
594
595     ngx\_memzero(u->state, sizeof(ngx\_http\_upstream\_state\_t));
596 }
597
598
599 cln = ngx\_http\_cleanup\_add(r, 0);
600 if (cln == NULL) {
601     ngx\_http\_finalize\_request(r, NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);
602     return;
603 }
604
605 cln->handler = ngx\_http\_upstream\_cleanup;

```

```

606     cln->data = r;
607     u->cleanup = &cln->handler;
608
609     if (u->resolved == NULL) {
610         uscf = u->conf->upstream;
611     } else {
612
613     #if (NGX_HTTP_SSL)
614         u->ssl_name = u->resolved->host;
615     #endif
616
617     if (u->resolved->sockaddr) {
618
619         if (ngx_http_upstream_create_round_robin_peer(r, u->resolved)
620             != NGX_OK)
621         {
622             ngx_http_upstream_finalize_request(r, u,
623                 NGX_HTTP_INTERNAL_SERVER_ERROR);
624             return;
625         }
626
627         ngx_http_upstream_connect(r, u);
628
629         return;
630     }
631
632     host = &u->resolved->host;
633
634     umcf = ngx_http_get_module_main_conf(r, ngx_http_upstream_module);
635
636     uscfp = umcf->upstreams.elts;
637
638     for (i = 0; i < umcf->upstreams.nelts; i++) {
639
640         uscf = uscfp[i];
641
642         if (uscf->host.len == host->len
643             && ((uscf->port == 0 && u->resolved->no_port)
644                 || uscf->port == u->resolved->port)
645             && ngx_strncasecmp(uscf->host.data, host->data, host->len) == 0)
646         {
647             goto found;
648         }
649     }
650
651     if (u->resolved->port == 0) {
652         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
653             "no port in upstream \"%V\"", host);
654         ngx_http_upstream_finalize_request(r, u,
655             NGX_HTTP_INTERNAL_SERVER_ERROR);
656         return;
657     }
658
659     temp.name = *host;
660
661     ctx = ngx_resolve_start(clcf->resolver, &temp);
662     if (ctx == NULL) {
663         ngx_http_upstream_finalize_request(r, u,
664             NGX_HTTP_INTERNAL_SERVER_ERROR);
665         return;
666     }
667
668     if (ctx == NGX_NO_RESOLVER) {
669         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
670             "no resolver defined to resolve %V", host);
671
672         ngx_http_upstream_finalize_request(r, u, NGX_HTTP_BAD_GATEWAY);
673         return;
674     }
675
676     ctx->name = *host;
677     ctx->handler = ngx_http_upstream_resolve_handler;
678     ctx->data = r;
679     ctx->timeout = clcf->resolver_timeout;

```

```

682     u->resolved->ctx = ctx;
683
684
685     if (ngx_resolve_name(ctx) != NGX_OK) {
686         u->resolved->ctx = NULL;
687         ngx_http_upstream_finalize_request(r, u,
688                                           NGX_HTTP_INTERNAL_SERVER_ERROR);
689         return;
690     }
691
692     return;
693 }
694
695 found:
696
697     if (uscf == NULL) {
698         ngx_log_error(NGX_LOG_ALERT, r->connection->log, 0,
699                     "no upstream configuration");
700         ngx_http_upstream_finalize_request(r, u,
701                                           NGX_HTTP_INTERNAL_SERVER_ERROR);
702         return;
703     }
704
705     #if (NGX_HTTP_SSL)
706     u->ssl_name = uscf->host;
707     #endif
708
709     if (uscf->peer.init(r, uscf) != NGX_OK) {
710         ngx_http_upstream_finalize_request(r, u,
711                                           NGX_HTTP_INTERNAL_SERVER_ERROR);
712         return;
713     }
714
715     u->peer.start_time = ngx_current_msec;
716
717     if (u->conf->next_upstream_tries
718         && u->peer.tries > u->conf->next_upstream_tries)
719     {
720         u->peer.tries = u->conf->next_upstream_tries;
721     }
722
723     ngx_http_upstream_connect(r, u);
724 }
725
726
727 #if (NGX_HTTP_CACHE)
728
729 static ngx_int_t
730 ngx_http_upstream_cache(ngx_http_request_t *r, ngx_http_upstream_t *u)
731 {
732     ngx_int_t          rc;
733     ngx_http_cache_t  *c;
734     ngx_http_file_cache_t *cache;
735
736     c = r->cache;
737
738     if (c == NULL) {
739
740         if (!(r->method & u->conf->cache_methods)) {
741             return NGX_DECLINED;
742         }
743
744         rc = ngx_http_upstream_cache_get(r, u, &cache);
745
746         if (rc != NGX_OK) {
747             return rc;
748         }
749
750         if (r->method & NGX_HTTP_HEAD) {
751             u->method = ngx_http_core_get_method;
752         }
753
754         if (ngx_http_file_cache_new(r) != NGX_OK) {
755             return NGX_ERROR;
756         }
757     }

```

```

758     if (u->create_key(r) != NGX_OK) {
759         return NGX_ERROR;
760     }
761
762     /* TODO: add keys */
763
764     ngx_http_file_cache_create_key(r);
765
766     if (r->cache->header_start + 256 >= u->conf->buffer_size) {
767         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
768             "%V_buffer_size %uz is not enough for cache key, "
769             "it should be increased to at least %uz",
770             &u->conf->module, u->conf->buffer_size,
771             ngx_align(r->cache->header_start + 256, 1024));
772
773         r->cache = NULL;
774         return NGX_DECLINED;
775     }
776
777     u->cacheable = 1;
778
779     c = r->cache;
780
781     c->body_start = u->conf->buffer_size;
782     c->min_uses = u->conf->cache_min_uses;
783     c->file_cache = cache;
784
785     switch (ngx_http_test_predicates(r, u->conf->cache_bypass)) {
786
787     case NGX_ERROR:
788         return NGX_ERROR;
789
790     case NGX_DECLINED:
791         u->cache_status = NGX_HTTP_CACHE_BYPASS;
792         return NGX_DECLINED;
793
794     default: /* NGX_OK */
795         break;
796     }
797
798     c->lock = u->conf->cache_lock;
799     c->lock_timeout = u->conf->cache_lock_timeout;
800     c->lock_age = u->conf->cache_lock_age;
801
802     u->cache_status = NGX_HTTP_CACHE_MISS;
803 }
804
805 rc = ngx_http_file_cache_open(r);
806
807 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
808     "http upstream cache: %i", rc);
809
810 switch (rc) {
811
812     case NGX_HTTP_CACHE_UPDATING:
813
814         if (u->conf->cache_use_stale & NGX_HTTP_UPSTREAM_FT_UPDATING) {
815             u->cache_status = rc;
816             rc = NGX_OK;
817         } else {
818             rc = NGX_HTTP_CACHE_STALE;
819         }
820     }
821
822     break;
823
824     case NGX_OK:
825         u->cache_status = NGX_HTTP_CACHE_HIT;
826     }
827
828     switch (rc) {
829
830     case NGX_OK:
831
832         rc = ngx_http_upstream_cache_send(r, u);
833

```



```

834     if (rc != NGX\_HTTP\_UPSTREAM\_INVALID\_HEADER) {
835         return rc;
836     }
837
838     break;
839
840 case NGX\_HTTP\_CACHE\_STALE:
841
842     c->valid_sec = 0;
843     u->buffer.start = NULL;
844     u->cache_status = NGX\_HTTP\_CACHE\_EXPIRED;
845
846     break;
847
848 case NGX\_DECLINED:
849
850     if ((size\_t) (u->buffer.end - u->buffer.start) < u->conf->buffer_size) {
851         u->buffer.start = NULL;
852
853     } else {
854         u->buffer.pos = u->buffer.start + c->header_start;
855         u->buffer.last = u->buffer.pos;
856     }
857
858     break;
859
860 case NGX\_HTTP\_CACHE\_SCARCE:
861
862     u->cacheable = 0;
863
864     break;
865
866 case NGX\_AGAIN:
867
868     return NGX\_BUSY;
869
870 case NGX\_ERROR:
871
872     return NGX\_ERROR;
873
874 default:
875
876     /* cached NGX\_HTTP\_BAD\_GATEWAY, NGX\_HTTP\_GATEWAY\_TIME\_OUT, etc. */
877
878     u->cache_status = NGX\_HTTP\_CACHE\_HIT;
879
880     return rc;
881 }
882
883 r->cached = 0;
884
885 return NGX\_DECLINED;
886 }
887
888
889 static ngx\_int\_t
890 ngx_http_upstream_cache_get(ngx\_http\_request\_t *r, ngx\_http\_upstream\_t *u,
891 ngx\_http\_file\_cache\_t **cache)
892 {
893     ngx\_str\_t          *name, val;
894     ngx\_uint\_t       i;
895     ngx\_http\_file\_cache\_t **caches;
896
897     if (u->conf->cache_zone) {
898         *cache = u->conf->cache_zone->data;
899         return NGX\_OK;
900     }
901
902     if (ngx\_http\_complex\_value(r, u->conf->cache_value, &val) != NGX\_OK) {
903         return NGX\_ERROR;
904     }
905
906     if (val.len == 0
907         || (val.len == 3 && ngx\_strncmp(val.data, "off", 3) == 0))
908     {
909         return NGX\_DECLINED;

```

```

910 }
911
912 caches = u->caches->elts;
913
914 for (i = 0; i < u->caches->nelts; i++) {
915     name = &caches[i]->shm_zone->shm.name;
916
917     if (name->len == val.len
918         && ngx_strncmp(name->data, val.data, val.len) == 0)
919     {
920         *cache = caches[i];
921         return NGX_OK;
922     }
923 }
924
925 ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
926     "cache \"%V\" not found", &val);
927
928 return NGX_ERROR;
929 }
930
931
932 static ngx_int_t
933 ngx_http_upstream_cache_send(ngx_http_request_t *r, ngx_http_upstream_t *u)
934 {
935     ngx_int_t rc;
936     ngx_http_cache_t *c;
937
938     r->cached = 1;
939     c = r->cache;
940
941     if (c->header_start == c->body_start) {
942         r->http_version = NGX_HTTP_VERSION_9;
943         return ngx_http_cache_send(r);
944     }
945
946     /* TODO: cache stack */
947
948     u->buffer = *c->buf;
949     u->buffer.pos += c->header_start;
950
951     ngx_memzero(&u->headers_in, sizeof(ngx_http_upstream_headers_in_t));
952     u->headers_in.content_length_n = -1;
953     u->headers_in.last_modified_time = -1;
954
955     if (ngx_list_init(&u->headers_in.headers, r->pool, 8,
956         sizeof(ngx_table_elt_t))
957         != NGX_OK)
958     {
959         return NGX_ERROR;
960     }
961
962     rc = u->process_header(r);
963
964     if (rc == NGX_OK) {
965         if (ngx_http_upstream_process_headers(r, u) != NGX_OK) {
966             return NGX_DONE;
967         }
968
969         return ngx_http_cache_send(r);
970     }
971
972     if (rc == NGX_ERROR) {
973         return NGX_ERROR;
974     }
975
976     /* rc == NGX_HTTP_UPSTREAM_INVALID_HEADER */
977
978     /* TODO: delete file */
979
980     return rc;
981 }
982
983 #endif
984
985

```

```

986 static void
987 ngx_http_upstream_resolve_handler(ngx_resolver_ctx_t *ctx)
988 {
989     ngx_connection_t      *c;
990     ngx_http_request_t    *r;
991     ngx_http_upstream_t   *u;
992     ngx_http_upstream_resolved_t *ur;
993
994     r = ctx->data;
995     c = r->connection;
996
997     u = r->upstream;
998     ur = u->resolved;
999
1000     ngx_http_set_log_request(c->log, r);
1001
1002     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
1003                  "http upstream resolve: \"%V?%V\"", &r->uri, &r->args);
1004
1005     if (ctx->state) {
1006         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1007                     "%V could not be resolved (%i: %s)",
1008                     &ctx->name, ctx->state,
1009                     ngx_resolver_strerror(ctx->state));
1010
1011         ngx_http_upstream_finalize_request(r, u, NGX_HTTP_BAD_GATEWAY);
1012         goto failed;
1013     }
1014
1015     ur->naddrs = ctx->naddrs;
1016     ur->addrs = ctx->addrs;
1017
1018     #if (NGX_DEBUG)
1019     {
1020         u_char      text[NGX_SOCKADDR_STRLEN];
1021         ngx_str_t   addr;
1022         ngx_uint_t  i;
1023
1024         addr.data = text;
1025
1026         for (i = 0; i < ctx->naddrs; i++) {
1027             addr.len = ngx_sock_ntop(ur->addrs[i].sockaddr, ur->addrs[i].socklen,
1028                                     text, NGX_SOCKADDR_STRLEN, 0);
1029
1030             ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1031                           "name was resolved to %V", &addr);
1032         }
1033     }
1034     #endif
1035
1036     if (ngx_http_upstream_create_round_robin_peer(r, ur) != NGX_OK) {
1037         ngx_http_upstream_finalize_request(r, u,
1038                                           NGX_HTTP_INTERNAL_SERVER_ERROR);
1039         goto failed;
1040     }
1041
1042     ngx_resolve_name_done(ctx);
1043     ur->ctx = NULL;
1044
1045     u->peer.start_time = ngx_current_msec;
1046
1047     if (u->conf->next_upstream_tries
1048         && u->peer.tries > u->conf->next_upstream_tries)
1049     {
1050         u->peer.tries = u->conf->next_upstream_tries;
1051     }
1052
1053     ngx_http_upstream_connect(r, u);
1054
1055 failed:
1056     ngx_http_run_posted_requests(c);
1057 }
1058
1059 }
1060
1061

```

```

1062 static void
1063 ngx_http_upstream_handler(ngx_event_t *ev)
1064 {
1065     ngx_connection_t    *c;
1066     ngx_http_request_t  *r;
1067     ngx_http_upstream_t *u;
1068
1069     c = ev->data;
1070     r = c->data;
1071
1072     u = r->upstream;
1073     c = r->connection;
1074
1075     ngx_http_set_log_request(c->log, r);
1076
1077     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
1078                 "http upstream request: \"%V?%V\"", &r->uri, &r->args);
1079
1080     if (ev->write) {
1081         u->write_event_handler(r, u);
1082     } else {
1083         u->read_event_handler(r, u);
1084     }
1085
1086     ngx_http_run_posted_requests(c);
1087 }
1088
1089
1090
1091 static void
1092 ngx_http_upstream_rd_check_broken_connection(ngx_http_request_t *r)
1093 {
1094     ngx_http_upstream_check_broken_connection(r, r->connection->read);
1095 }
1096
1097
1098 static void
1099 ngx_http_upstream_wr_check_broken_connection(ngx_http_request_t *r)
1100 {
1101     ngx_http_upstream_check_broken_connection(r, r->connection->write);
1102 }
1103
1104
1105 static void
1106 ngx_http_upstream_check_broken_connection(ngx_http_request_t *r,
1107     ngx_event_t *ev)
1108 {
1109     int                n;
1110     char               buf[1];
1111     ngx_err_t         err;
1112     ngx_int_t         event;
1113     ngx_connection_t  *c;
1114     ngx_http_upstream_t *u;
1115
1116     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, ev->log, 0,
1117                 "http upstream check client, write event:%d, \"%V\"",
1118                 ev->write, &r->uri);
1119
1120     c = r->connection;
1121     u = r->upstream;
1122
1123     if (c->error) {
1124         if ((ngx_event_flags & NGX_USE_LEVEL_EVENT) && ev->active) {
1125
1126             event = ev->write ? NGX_WRITE_EVENT : NGX_READ_EVENT;
1127
1128             if (ngx_del_event(ev, event, 0) != NGX_OK) {
1129                 ngx_http_upstream_finalize_request(r, u,
1130                     NGX_HTTP_INTERNAL_SERVER_ERROR);
1131                 return;
1132             }
1133         }
1134
1135         if (!u->cacheable) {
1136             ngx_http_upstream_finalize_request(r, u,
1137                 NGX_HTTP_CLIENT_CLOSED_REQUEST);

```

```

1138     }
1139
1140     return;
1141 }
1142
1143 #if (NGX_HTTP_SPDY)
1144     if (r->spdy_stream) {
1145         return;
1146     }
1147 #endif
1148
1149 #if (NGX_HAVE_KQUEUE)
1150
1151     if (ngx_event_flags & NGX_USE_KQUEUE_EVENT) {
1152
1153         if (!ev->pending_eof) {
1154             return;
1155         }
1156
1157         ev->eof = 1;
1158         c->error = 1;
1159
1160         if (ev->kq_errno) {
1161             ev->error = 1;
1162         }
1163
1164         if (!u->cacheable && u->peer.connection) {
1165             ngx_log_error(NGX_LOG_INFO, ev->log, ev->kq_errno,
1166                 "kevent() reported that client prematurely closed "
1167                 "connection, so upstream connection is closed too");
1168             ngx_http_upstream_finalize_request(r, u,
1169                 NGX_HTTP_CLIENT_CLOSED_REQUEST);
1170             return;
1171         }
1172
1173         ngx_log_error(NGX_LOG_INFO, ev->log, ev->kq_errno,
1174             "kevent() reported that client prematurely closed "
1175             "connection");
1176
1177         if (u->peer.connection == NULL) {
1178             ngx_http_upstream_finalize_request(r, u,
1179                 NGX_HTTP_CLIENT_CLOSED_REQUEST);
1180         }
1181
1182         return;
1183     }
1184 #endif
1185
1186 #if (NGX_HAVE_EPOLLRDHUP)
1187
1188     if ((ngx_event_flags & NGX_USE_EPOLL_EVENT) && ev->pending_eof) {
1189         socklen_t len;
1190
1191         ev->eof = 1;
1192         c->error = 1;
1193
1194         err = 0;
1195         len = sizeof(ngx_err_t);
1196
1197         /*
1198          * BSDs and Linux return 0 and set a pending error in err
1199          * Solaris returns -1 and sets errno
1200          */
1201
1202         if (getsockopt(c->fd, SOL_SOCKET, SO_ERROR, (void *) &err, &len)
1203             == -1)
1204         {
1205             err = ngx_socket_errno;
1206         }
1207
1208         if (err) {
1209             ev->error = 1;
1210         }
1211
1212         if (!u->cacheable && u->peer.connection) {

```

```

1214     ngx_log_error(NGX_LOG_INFO, ev->log, err,
1215                 "epoll_wait() reported that client prematurely closed "
1216                 "connection, so upstream connection is closed too");
1217     ngx_http_upstream_finalize_request(r, u,
1218                                     NGX_HTTP_CLIENT_CLOSED_REQUEST);
1219     return;
1220 }
1221
1222 ngx_log_error(NGX_LOG_INFO, ev->log, err,
1223             "epoll_wait() reported that client prematurely closed "
1224             "connection");
1225
1226 if (u->peer.connection == NULL) {
1227     ngx_http_upstream_finalize_request(r, u,
1228                                     NGX_HTTP_CLIENT_CLOSED_REQUEST);
1229 }
1230
1231 return;
1232 }
1233
1234 #endif
1235
1236 n = recv(c->fd, buf, 1, MSG_PEEK);
1237
1238 err = ngx_socket_errno;
1239
1240 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, ev->log, err,
1241              "http upstream recv(): %d", n);
1242
1243 if (ev->write && (n >= 0 || err == NGX_EAGAIN)) {
1244     return;
1245 }
1246
1247 if ((ngx_event_flags & NGX_USE_LEVEL_EVENT) && ev->active) {
1248     event = ev->write ? NGX_WRITE_EVENT : NGX_READ_EVENT;
1249
1250     if (ngx_del_event(ev, event, 0) != NGX_OK) {
1251         ngx_http_upstream_finalize_request(r, u,
1252                                     NGX_HTTP_INTERNAL_SERVER_ERROR);
1253         return;
1254     }
1255 }
1256
1257 if (n > 0) {
1258     return;
1259 }
1260
1261 if (n == -1) {
1262     if (err == NGX_EAGAIN) {
1263         return;
1264     }
1265
1266     ev->error = 1;
1267
1268 } else { /* n == 0 */
1269     err = 0;
1270 }
1271
1272
1273 ev->eof = 1;
1274 c->error = 1;
1275
1276 if (!u->cacheable && u->peer.connection) {
1277     ngx_log_error(NGX_LOG_INFO, ev->log, err,
1278                 "client prematurely closed connection, "
1279                 "so upstream connection is closed too");
1280     ngx_http_upstream_finalize_request(r, u,
1281                                     NGX_HTTP_CLIENT_CLOSED_REQUEST);
1282     return;
1283 }
1284
1285 ngx_log_error(NGX_LOG_INFO, ev->log, err,
1286             "client prematurely closed connection");
1287
1288 if (u->peer.connection == NULL) {
1289     ngx_http_upstream_finalize_request(r, u,

```

```

1290         NGX\_HTTP\_CLIENT\_CLOSED\_REQUEST);
1291     }
1292 }
1293
1294
1295 static void
1296 ngx_http_upstream_connect(ngx\_http\_request\_t *r, ngx\_http\_upstream\_t *u)
1297 {
1298     ngx\_int\_t         rc;
1299     ngx\_time\_t       *tp;
1300     ngx\_connection\_t *c;
1301
1302     r->connection->log->action = "connecting to upstream";
1303
1304     if (u->state && u->state->response_sec) {
1305         tp = ngx\_timeofday();
1306         u->state->response_sec = tp->sec - u->state->response_sec;
1307         u->state->response_msec = tp->msec - u->state->response_msec;
1308     }
1309
1310     u->state = ngx\_array\_push(r->upstream_states);
1311     if (u->state == NULL) {
1312         ngx\_http\_upstream\_finalize\_request(r, u,
1313                                         NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);
1314         return;
1315     }
1316
1317     ngx\_memzero(u->state, sizeof(ngx\_http\_upstream\_state\_t));
1318
1319     tp = ngx\_timeofday();
1320     u->state->response_sec = tp->sec;
1321     u->state->response_msec = tp->msec;
1322     u->state->header_sec = (time_t) NGX\_ERROR;
1323
1324     rc = ngx\_event\_connect\_peer(&u->peer);
1325
1326     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
1327                  "http upstream connect: %i", rc);
1328
1329     if (rc == NGX\_ERROR) {
1330         ngx\_http\_upstream\_finalize\_request(r, u,
1331                                         NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);
1332         return;
1333     }
1334
1335     u->state->peer = u->peer.name;
1336
1337     if (rc == NGX\_BUSY) {
1338         ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, 0, "no live upstreams");
1339         ngx\_http\_upstream\_next(r, u, NGX\_HTTP\_UPSTREAM\_FT\_NOLIVE);
1340         return;
1341     }
1342
1343     if (rc == NGX\_DECLINED) {
1344         ngx\_http\_upstream\_next(r, u, NGX\_HTTP\_UPSTREAM\_FT\_ERROR);
1345         return;
1346     }
1347
1348     /* rc == NGX\_OK || rc == NGX\_AGAIN || rc == NGX\_DONE */
1349
1350     c = u->peer.connection;
1351
1352     c->data = r;
1353
1354     c->write->handler = ngx\_http\_upstream\_handler;
1355     c->read->handler = ngx\_http\_upstream\_handler;
1356
1357     u->write_event_handler = ngx\_http\_upstream\_send\_request\_handler;
1358     u->read_event_handler = ngx\_http\_upstream\_process\_header;
1359
1360     c->sendfile &= r->connection->sendfile;
1361     u->output.sendfile = c->sendfile;
1362
1363     if (c->pool == NULL) {
1364
1365         /* we need separate pool here to be able to cache SSL connections */

```

```

1366     c->pool = ngx_create_pool(128, r->connection->log);
1367     if (c->pool == NULL) {
1368         ngx_http_upstream_finalize_request(r, u,
1369                                         NGX_HTTP_INTERNAL_SERVER_ERROR);
1370     }
1371     return;
1372 }
1373 }
1374
1375 c->log = r->connection->log;
1376 c->pool->log = c->log;
1377 c->read->log = c->log;
1378 c->write->log = c->log;
1379
1380 /* init or reinit the ngx_output_chain() and ngx_chain_writer() contexts */
1381
1382 u->writer.out = NULL;
1383 u->writer.last = &u->writer.out;
1384 u->writer.connection = c;
1385 u->writer.limit = 0;
1386
1387 if (u->request_sent) {
1388     if (ngx_http_upstream_reinit(r, u) != NGX_OK) {
1389         ngx_http_upstream_finalize_request(r, u,
1390                                         NGX_HTTP_INTERNAL_SERVER_ERROR);
1391     }
1392     return;
1393 }
1394
1395 if (r->request_body
1396     && r->request_body->buf
1397     && r->request_body->temp_file
1398     && r == r->main)
1399 {
1400     /*
1401      * the r->request_body->buf can be reused for one request only,
1402      * the subrequests should allocate their own temporary bufs
1403      */
1404
1405     u->output.free = ngx_alloc_chain_link(r->pool);
1406     if (u->output.free == NULL) {
1407         ngx_http_upstream_finalize_request(r, u,
1408                                         NGX_HTTP_INTERNAL_SERVER_ERROR);
1409     }
1410     return;
1411 }
1412
1413 u->output.free->buf = r->request_body->buf;
1414 u->output.free->next = NULL;
1415 u->output.allocated = 1;
1416
1417 r->request_body->buf->pos = r->request_body->buf->start;
1418 r->request_body->buf->last = r->request_body->buf->start;
1419 r->request_body->buf->tag = u->output.tag;
1420 }
1421
1422 u->request_sent = 0;
1423
1424 if (rc == NGX_AGAIN) {
1425     ngx_add_timer(c->write, u->conf->connect_timeout);
1426     return;
1427 }
1428
1429 #if (NGX_HTTP_SSL)
1430
1431 if (u->ssl && c->ssl == NULL) {
1432     ngx_http_upstream_ssl_init_connection(r, u, c);
1433     return;
1434 }
1435 #endif
1436
1437 ngx_http_upstream_send_request(r, u);
1438 }
1439
1440 #if (NGX_HTTP_SSL)

```



```

1442 static void
1443 ngx_http_upstream_ssl_init_connection(ngx_http_request_t *r,
1444 ngx_http_upstream_t *u, ngx_connection_t *c)
1445 {
1446     ngx_int_t rc;
1447
1448     if (ngx_http_upstream_test_connect(c) != NGX_OK) {
1449         ngx_http_upstream_next(r, u, NGX_HTTP_UPSTREAM_FT_ERROR);
1450         return;
1451     }
1452
1453     if (ngx_ssl_create_connection(u->conf->ssl, c,
1454         NGX_SSL_BUFFER|NGX_SSL_CLIENT)
1455         != NGX_OK)
1456     {
1457         ngx_http_upstream_finalize_request(r, u,
1458             NGX_HTTP_INTERNAL_SERVER_ERROR);
1459         return;
1460     }
1461
1462     c->sendfile = 0;
1463     u->output.sendfile = 0;
1464
1465     if (u->conf->ssl_server_name || u->conf->ssl_verify) {
1466         if (ngx_http_upstream_ssl_name(r, u, c) != NGX_OK) {
1467             ngx_http_upstream_finalize_request(r, u,
1468                 NGX_HTTP_INTERNAL_SERVER_ERROR);
1469             return;
1470         }
1471     }
1472
1473     if (u->conf->ssl_session_reuse) {
1474         if (u->peer.set_session(&u->peer, u->peer.data) != NGX_OK) {
1475             ngx_http_upstream_finalize_request(r, u,
1476                 NGX_HTTP_INTERNAL_SERVER_ERROR);
1477             return;
1478         }
1479     }
1480
1481     r->connection->log->action = "SSL handshaking to upstream";
1482
1483     rc = ngx_ssl_handshake(c);
1484
1485     if (rc == NGX_AGAIN) {
1486         if (!c->write->timer_set) {
1487             ngx_add_timer(c->write, u->conf->connect_timeout);
1488         }
1489
1490         c->ssl->handler = ngx_http_upstream_ssl_handshake;
1491         return;
1492     }
1493
1494     ngx_http_upstream_ssl_handshake(c);
1495 }
1496
1497 static void
1498 ngx_http_upstream_ssl_handshake(ngx_connection_t *c)
1499 {
1500     long rc;
1501     ngx_http_request_t *r;
1502     ngx_http_upstream_t *u;
1503
1504     r = c->data;
1505     u = r->upstream;
1506
1507     ngx_http_set_log_request(c->log, r);
1508
1509     if (c->ssl->handshaked) {
1510         if (u->conf->ssl_verify) {
1511             rc = SSL_get_verify_result(c->ssl->connection);
1512
1513             if (rc != X509_V_OK) {

```

```

1518         ngx_log_error(NGX_LOG_ERR, c->log, 0,
1519             "upstream SSL certificate verify error: (%l:%s)",
1520             rc, X509_verify_cert_error_string(rc));
1521     goto failed;
1522 }
1523
1524     if (ngx_ssl_check_host(c, &u->ssl_name) != NGX_OK) {
1525         ngx_log_error(NGX_LOG_ERR, c->log, 0,
1526             "upstream SSL certificate does not match \"%V\"",
1527             &u->ssl_name);
1528         goto failed;
1529     }
1530 }
1531
1532     if (u->conf->ssl_session_reuse) {
1533         u->peer.save_session(&u->peer, u->peer.data);
1534     }
1535
1536     c->write->handler = ngx_http_upstream_handler;
1537     c->read->handler = ngx_http_upstream_handler;
1538
1539     c = r->connection;
1540
1541     ngx_http_upstream_send_request(r, u);
1542
1543     ngx_http_run_posted_requests(c);
1544     return;
1545 }
1546
1547 failed:
1548
1549     c = r->connection;
1550
1551     ngx_http_upstream_next(r, u, NGX_HTTP_UPSTREAM_FT_ERROR);
1552
1553     ngx_http_run_posted_requests(c);
1554 }
1555
1556
1557 static ngx_int_t
1558 ngx_http_upstream_ssl_name(ngx_http_request_t *r, ngx_http_upstream_t *u,
1559     ngx_connection_t *c)
1560 {
1561     u_char      *p, *last;
1562     ngx_str_t    name;
1563
1564     if (u->conf->ssl_name) {
1565         if (ngx_http_complex_value(r, u->conf->ssl_name, &name) != NGX_OK) {
1566             return NGX_ERROR;
1567         }
1568     } else {
1569         name = u->ssl_name;
1570     }
1571
1572     if (name.len == 0) {
1573         goto done;
1574     }
1575
1576     /*
1577     * ssl name here may contain port, notably if derived from $proxy_host
1578     * or $http_host; we have to strip it
1579     */
1580
1581     p = name.data;
1582     last = name.data + name.len;
1583
1584     if (*p == '[') {
1585         p = ngx_strlchr(p, last, ']');
1586
1587         if (p == NULL) {
1588             p = name.data;
1589         }
1590     }
1591
1592     p = ngx_strlchr(p, last, ':');

```

```

1594     if (p != NULL) {
1595         name.len = p - name.data;
1596     }
1597
1598
1599     if (!u->conf->ssl_server_name) {
1600         goto done;
1601     }
1602
1603 #ifdef SSL_CTRL_SET_TLSEXT_HOSTNAME
1604
1605     /* as per RFC 6066, literal IPv4 and IPv6 addresses are not permitted */
1606
1607     if (name.len == 0 || *name.data == '[') {
1608         goto done;
1609     }
1610
1611     if (ngx_inet_addr(name.data, name.len) != INADDR_NONE) {
1612         goto done;
1613     }
1614
1615     /*
1616     * SSL_set_tlsext_host_name() needs a null-terminated string,
1617     * hence we explicitly null-terminate name here
1618     */
1619
1620     p = ngx_pnalloc(r->pool, name.len + 1);
1621     if (p == NULL) {
1622         return NGX_ERROR;
1623     }
1624
1625     (void) ngx_cpystn(p, name.data, name.len + 1);
1626
1627     name.data = p;
1628
1629     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1630                  "upstream SSL server name: \"%s\"", name.data);
1631
1632     if (SSL_set_tlsext_host_name(c->ssl->connection, name.data) == 0) {
1633         ngx_ssl_error(NGX_LOG_ERR, r->connection->log, 0,
1634                     "SSL_set_tlsext_host_name(\"%s\") failed", name.data);
1635         return NGX_ERROR;
1636     }
1637
1638 #endif
1639
1640 done:
1641
1642     u->ssl_name = name;
1643
1644     return NGX_OK;
1645 }
1646
1647 #endif
1648
1649
1650 static ngx_int_t
1651 ngx_http_upstream_reinit(ngx_http_request_t *r, ngx_http_upstream_t *u)
1652 {
1653     off_t      file_pos;
1654     ngx_chain_t *cl;
1655
1656     if (u->reinit_request(r) != NGX_OK) {
1657         return NGX_ERROR;
1658     }
1659
1660     u->keepalive = 0;
1661     u->upgrade = 0;
1662
1663     ngx_memzero(&u->headers_in, sizeof(ngx_http_upstream_headers_in_t));
1664     u->headers_in.content_length_n = -1;
1665     u->headers_in.last_modified_time = -1;
1666
1667     if (ngx_list_init(&u->headers_in.headers, r->pool, 8,
1668                     sizeof(ngx_table_elt_t))
1669         != NGX_OK)

```

```

1670     {
1671         return NGX\_ERROR;
1672     }
1673
1674     /* reinit the request chain */
1675
1676     file_pos = 0;
1677
1678     for (cl = u->request_bufs; cl; cl = cl->next) {
1679         cl->buf->pos = cl->buf->start;
1680
1681         /* there is at most one file */
1682
1683         if (cl->buf->in_file) {
1684             cl->buf->file_pos = file_pos;
1685             file_pos = cl->buf->file_last;
1686         }
1687     }
1688
1689     /* reinit the subrequest's ngx\_output\_chain\(\) context */
1690
1691     if (r->request_body && r->request_body->temp_file
1692         && r != r->main && u->output.buf)
1693     {
1694         u->output.free = ngx\_alloc\_chain\_link(r->pool);
1695         if (u->output.free == NULL) {
1696             return NGX\_ERROR;
1697         }
1698
1699         u->output.free->buf = u->output.buf;
1700         u->output.free->next = NULL;
1701
1702         u->output.buf->pos = u->output.buf->start;
1703         u->output.buf->last = u->output.buf->start;
1704     }
1705
1706     u->output.buf = NULL;
1707     u->output.in = NULL;
1708     u->output.busy = NULL;
1709
1710     /* reinit u->buffer */
1711
1712     u->buffer.pos = u->buffer.start;
1713
1714     #if (NGX_HTTP_CACHE)
1715
1716     if (r->cache) {
1717         u->buffer.pos += r->cache->header_start;
1718     }
1719
1720     #endif
1721
1722     u->buffer.last = u->buffer.pos;
1723
1724     return NGX\_OK;
1725 }
1726
1727
1728 static void
1729 ngx\_http\_upstream\_send\_request(ngx\_http\_request\_t *r, ngx\_http\_upstream\_t *u)
1730 {
1731     ngx\_int\_t rc;
1732     ngx\_connection\_t *c;
1733
1734     c = u->peer.connection;
1735
1736     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, c->log, 0,
1737         "http upstream send request");
1738
1739     if (!u->request_sent && ngx\_http\_upstream\_test\_connect(c) != NGX\_OK) {
1740         ngx\_http\_upstream\_next(r, u, NGX\_HTTP\_UPSTREAM\_FT\_ERROR);
1741         return;
1742     }
1743
1744     c->log->action = "sending request to upstream";
1745

```

```

1746 rc = ngx_output_chain(&u->output, u->request_sent ? NULL : u->request_bufs);
1747
1748 u->request_sent = 1;
1749
1750 if (rc == NGX_ERROR) {
1751     ngx_http_upstream_next(r, u, NGX_HTTP_UPSTREAM_FT_ERROR);
1752     return;
1753 }
1754
1755 if (c->write->timer_set) {
1756     ngx_del_timer(c->write);
1757 }
1758
1759 if (rc == NGX_AGAIN) {
1760     ngx_add_timer(c->write, u->conf->send_timeout);
1761
1762     if (ngx_handle_write_event(c->write, u->conf->send_lowat) != NGX_OK) {
1763         ngx_http_upstream_finalize_request(r, u,
1764             NGX_HTTP_INTERNAL_SERVER_ERROR);
1765         return;
1766     }
1767     return;
1768 }
1769
1770 /* rc == NGX_OK */
1771
1772 if (c->tcp_nopush == NGX_TCP_NOPUSH_SET) {
1773     if (ngx_tcp_push(c->fd) == NGX_ERROR) {
1774         ngx_log_error(NGX_LOG_CRIT, c->log, ngx_socket_errno,
1775             ngx_tcp_push_n " failed");
1776         ngx_http_upstream_finalize_request(r, u,
1777             NGX_HTTP_INTERNAL_SERVER_ERROR);
1778         return;
1779     }
1780 }
1781
1782 c->tcp_nopush = NGX_TCP_NOPUSH_UNSET;
1783 }
1784
1785 u->write_event_handler = ngx_http_upstream_dummy_handler;
1786
1787 if (ngx_handle_write_event(c->write, 0) != NGX_OK) {
1788     ngx_http_upstream_finalize_request(r, u,
1789         NGX_HTTP_INTERNAL_SERVER_ERROR);
1790     return;
1791 }
1792
1793 ngx_add_timer(c->read, u->conf->read_timeout);
1794
1795 if (c->read->ready) {
1796     ngx_http_upstream_process_header(r, u);
1797     return;
1798 }
1799 }
1800
1801
1802 static void
1803 ngx_http_upstream_send_request_handler(ngx_http_request_t *r,
1804     ngx_http_upstream_t *u)
1805 {
1806     ngx_connection_t *c;
1807
1808     c = u->peer.connection;
1809
1810     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1811         "http upstream send request handler");
1812
1813     if (c->write->timedout) {
1814         ngx_http_upstream_next(r, u, NGX_HTTP_UPSTREAM_FT_TIMEOUT);
1815         return;
1816     }
1817
1818 #if (NGX_HTTP_SSL)
1819
1820     if (u->ssl && c->ssl == NULL) {
1821         ngx_http_upstream_ssl_init_connection(r, u, c);

```

```

1822     return;
1823 }
1824
1825 #endif
1826
1827 if (u->header_sent) {
1828     u->write_event_handler = ngx\_http\_upstream\_dummy\_handler;
1829
1830     (void) ngx\_handle\_write\_event(c->write, 0);
1831
1832     return;
1833 }
1834
1835 ngx\_http\_upstream\_send\_request(r, u);
1836 }
1837
1838
1839 static void
1840 ngx\_http\_upstream\_process\_header(ngx\_http\_request\_t *r, ngx\_http\_upstream\_t *u)
1841 {
1842     ssize_t          n;
1843     ngx\_int\_t        rc;
1844     ngx\_time\_t      *tp;
1845     ngx\_connection\_t *c;
1846
1847     c = u->peer.connection;
1848
1849     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, c->log, 0,
1850                 "http upstream process header");
1851
1852     c->log->action = "reading response header from upstream";
1853
1854     if (c->read->timedout) {
1855         ngx\_http\_upstream\_next(r, u, NGX\_HTTP\_UPSTREAM\_FT\_TIMEOUT);
1856         return;
1857     }
1858
1859     if (!u->request_sent && ngx\_http\_upstream\_test\_connect(c) != NGX\_OK) {
1860         ngx\_http\_upstream\_next(r, u, NGX\_HTTP\_UPSTREAM\_FT\_ERROR);
1861         return;
1862     }
1863
1864     if (u->buffer.start == NULL) {
1865         u->buffer.start = ngx\_palloc(r->pool, u->conf->buffer_size);
1866         if (u->buffer.start == NULL) {
1867             ngx\_http\_upstream\_finalize\_request(r, u,
1868                                             NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);
1869             return;
1870         }
1871
1872         u->buffer.pos = u->buffer.start;
1873         u->buffer.last = u->buffer.start;
1874         u->buffer.end = u->buffer.start + u->conf->buffer_size;
1875         u->buffer.temporary = 1;
1876
1877         u->buffer.tag = u->output.tag;
1878
1879         if (ngx\_list\_init(&u->headers_in.headers, r->pool, 8,
1880                        sizeof(ngx\_table\_elt\_t))
1881             != NGX\_OK)
1882         {
1883             ngx\_http\_upstream\_finalize\_request(r, u,
1884                                             NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);
1885             return;
1886         }
1887
1888     #if (NGX\_HTTP\_CACHE)
1889
1890         if (r->cache) {
1891             u->buffer.pos += r->cache->header_start;
1892             u->buffer.last = u->buffer.pos;
1893         }
1894     #endif
1895 }
1896
1897 for ( ;; ) {

```

```

1898     n = c->recv(c, u->buffer.last, u->buffer.end - u->buffer.last);
1899
1900     if (n == NGX_AGAIN) {
1901     #if 0
1902         ngx_add_timer(rev, u->read_timeout);
1903     #endif
1904
1905         if (ngx_handle_read_event(c->read, 0) != NGX_OK) {
1906             ngx_http_upstream_finalize_request(r, u,
1907                 NGX_HTTP_INTERNAL_SERVER_ERROR);
1908             return;
1909         }
1910
1911         return;
1912     }
1913
1914     if (n == 0) {
1915         ngx_log_error(NGX_LOG_ERR, c->log, 0,
1916             "upstream prematurely closed connection");
1917     }
1918
1919     if (n == NGX_ERROR || n == 0) {
1920         ngx_http_upstream_next(r, u, NGX_HTTP_UPSTREAM_FT_ERROR);
1921         return;
1922     }
1923
1924     u->buffer.last += n;
1925
1926     #if 0
1927     u->valid_header_in = 0;
1928
1929     u->peer.cached = 0;
1930     #endif
1931
1932     rc = u->process_header(r);
1933
1934     if (rc == NGX_AGAIN) {
1935
1936         if (u->buffer.last == u->buffer.end) {
1937             ngx_log_error(NGX_LOG_ERR, c->log, 0,
1938                 "upstream sent too big header");
1939
1940             ngx_http_upstream_next(r, u,
1941                 NGX_HTTP_UPSTREAM_FT_INVALID_HEADER);
1942             return;
1943         }
1944
1945         continue;
1946     }
1947
1948     break;
1949 }
1950
1951 if (rc == NGX_HTTP_UPSTREAM_INVALID_HEADER) {
1952     ngx_http_upstream_next(r, u, NGX_HTTP_UPSTREAM_FT_INVALID_HEADER);
1953     return;
1954 }
1955
1956 if (rc == NGX_ERROR) {
1957     ngx_http_upstream_finalize_request(r, u,
1958         NGX_HTTP_INTERNAL_SERVER_ERROR);
1959     return;
1960 }
1961
1962 /* rc == NGX_OK */
1963
1964 tp = ngx_timeofday();
1965 u->state->header_sec = tp->sec - u->state->response_sec;
1966 u->state->header_msec = tp->msec - u->state->response_msec;
1967
1968 if (u->headers_in.status_n >= NGX_HTTP_SPECIAL_RESPONSE) {
1969
1970     if (ngx_http_upstream_test_next(r, u) == NGX_OK) {
1971         return;
1972     }
1973

```

```

1974     if (ngx\_http\_upstream\_intercept\_errors(r, u) == NGX\_OK) {
1975         return;
1976     }
1977 }
1978
1979
1980 if (ngx\_http\_upstream\_process\_headers(r, u) != NGX\_OK) {
1981     return;
1982 }
1983
1984 if (!r->subrequest_in_memory) {
1985     ngx\_http\_upstream\_send\_response(r, u);
1986     return;
1987 }
1988
1989 /* subrequest content in memory */
1990
1991 if (u->input_filter == NULL) {
1992     u->input_filter_init = ngx\_http\_upstream\_non\_buffered\_filter\_init;
1993     u->input_filter = ngx\_http\_upstream\_non\_buffered\_filter;
1994     u->input_filter_ctx = r;
1995 }
1996
1997 if (u->input_filter_init(u->input_filter_ctx) == NGX\_ERROR) {
1998     ngx\_http\_upstream\_finalize\_request(r, u, NGX\_ERROR);
1999     return;
2000 }
2001
2002 n = u->buffer.last - u->buffer.pos;
2003
2004 if (n) {
2005     u->buffer.last = u->buffer.pos;
2006
2007     u->state->response_length += n;
2008
2009     if (u->input_filter(u->input_filter_ctx, n) == NGX\_ERROR) {
2010         ngx\_http\_upstream\_finalize\_request(r, u, NGX\_ERROR);
2011         return;
2012     }
2013 }
2014
2015 if (u->length == 0) {
2016     ngx\_http\_upstream\_finalize\_request(r, u, 0);
2017     return;
2018 }
2019
2020 u->read_event_handler = ngx\_http\_upstream\_process\_body\_in\_memory;
2021
2022 ngx\_http\_upstream\_process\_body\_in\_memory(r, u);
2023 }
2024
2025
2026 static ngx\_int\_t
2027 ngx\_http\_upstream\_test\_next(ngx\_http\_request\_t *r, ngx\_http\_upstream\_t *u)
2028 {
2029     ngx\_uint\_t          status;
2030     ngx\_http\_upstream\_next\_t *un;
2031
2032     status = u->headers_in.status_n;
2033
2034     for (un = ngx\_http\_upstream\_next\_errors; un->status; un++) {
2035
2036         if (status != un->status) {
2037             continue;
2038         }
2039
2040         if (u->peer.tries > 1 && (u->conf->next_upstream & un->mask)) {
2041             ngx\_http\_upstream\_next(r, u, un->mask);
2042             return NGX\_OK;
2043         }
2044
2045 #if (NGX_HTTP_CACHE)
2046
2047         if (u->cache_status == NGX\_HTTP\_CACHE\_EXPIRED
2048             && (u->conf->cache_use_stale & un->mask))
2049             {

```



```

2050     ngx_int_t rc;
2051
2052     rc = u->reinit_request(r);
2053
2054     if (rc == NGX_OK) {
2055         u->cache_status = NGX_HTTP_CACHE_STALE;
2056         rc = ngx_http_upstream_cache_send(r, u);
2057     }
2058
2059     ngx_http_upstream_finalize_request(r, u, rc);
2060     return NGX_OK;
2061 }
2062
2063 #endif
2064 }
2065
2066 #if (NGX_HTTP_CACHE)
2067
2068     if (status == NGX_HTTP_NOT_MODIFIED
2069         && u->cache_status == NGX_HTTP_CACHE_EXPIRED
2070         && u->conf->cache_revalidate)
2071     {
2072         time_t now, valid;
2073         ngx_int_t rc;
2074
2075         ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
2076                      "http upstream not modified");
2077
2078         now = ngx_time();
2079         valid = r->cache->valid_sec;
2080
2081         rc = u->reinit_request(r);
2082
2083         if (rc != NGX_OK) {
2084             ngx_http_upstream_finalize_request(r, u, rc);
2085             return NGX_OK;
2086         }
2087
2088         u->cache_status = NGX_HTTP_CACHE_REVALIDATED;
2089         rc = ngx_http_upstream_cache_send(r, u);
2090
2091         if (valid == 0) {
2092             valid = r->cache->valid_sec;
2093         }
2094
2095         if (valid == 0) {
2096             valid = ngx_http_file_cache_valid(u->conf->cache_valid,
2097                                             u->headers_in.status_n);
2098
2099             if (valid) {
2100                 valid = now + valid;
2101             }
2102         }
2103
2104         if (valid) {
2105             r->cache->valid_sec = valid;
2106             r->cache->date = now;
2107
2108             ngx_http_file_cache_update_header(r);
2109         }
2110
2111         ngx_http_upstream_finalize_request(r, u, rc);
2112         return NGX_OK;
2113     }
2114 #endif
2115
2116     return NGX_DECLINED;
2117 }
2118
2119
2120 static ngx_int_t
2121 ngx_http_upstream_intercept_errors(ngx_http_request_t *r,
2122                                   ngx_http_upstream_t *u)
2123 {
2124     ngx_int_t status;
2125     ngx_uint_t i;

```

```

2126     ngx_table_elt_t      *h;
2127     ngx_http_err_page_t  *err_page;
2128     ngx_http_core_loc_conf_t *clcf;
2129
2130     status = u->headers_in.status_n;
2131
2132     if (status == NGX_HTTP_NOT_FOUND && u->conf->intercept_404) {
2133         ngx_http_upstream_finalize_request(r, u, NGX_HTTP_NOT_FOUND);
2134         return NGX_OK;
2135     }
2136
2137     if (!u->conf->intercept_errors) {
2138         return NGX_DECLINED;
2139     }
2140
2141     clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
2142
2143     if (clcf->error_pages == NULL) {
2144         return NGX_DECLINED;
2145     }
2146
2147     err_page = clcf->error_pages->elts;
2148     for (i = 0; i < clcf->error_pages->nelts; i++) {
2149
2150         if (err_page[i].status == status) {
2151
2152             if (status == NGX_HTTP_UNAUTHORIZED
2153                 && u->headers_in.www_authenticate)
2154             {
2155                 h = ngx_list_push(&r->headers_out.headers);
2156
2157                 if (h == NULL) {
2158                     ngx_http_upstream_finalize_request(r, u,
2159                                                         NGX_HTTP_INTERNAL_SERVER_ERROR);
2160                     return NGX_OK;
2161                 }
2162
2163                 *h = *u->headers_in.www_authenticate;
2164
2165                 r->headers_out.www_authenticate = h;
2166             }
2167
2168             #if (NGX_HTTP_CACHE)
2169
2170             if (r->cache) {
2171                 time_t valid;
2172
2173                 valid = ngx_http_file_cache_valid(u->conf->cache_valid, status);
2174
2175                 if (valid) {
2176                     r->cache->valid_sec = ngx_time() + valid;
2177                     r->cache->error = status;
2178                 }
2179
2180                 ngx_http_file_cache_free(r->cache, u->pipe->temp_file);
2181             }
2182             #endif
2183             ngx_http_upstream_finalize_request(r, u, status);
2184
2185             return NGX_OK;
2186         }
2187     }
2188
2189     return NGX_DECLINED;
2190 }
2191
2192
2193 static ngx_int_t
2194 ngx_http_upstream_test_connect(ngx_connection_t *c)
2195 {
2196     int      err;
2197     socklen_t len;
2198
2199     #if (NGX_HAVE_KQUEUE)
2200
2201     if (ngx_event_flags & NGX_USE_KQUEUE_EVENT) {

```

```

2202     if (c->write->pending_eof || c->read->pending_eof) {
2203         if (c->write->pending_eof) {
2204             err = c->write->kq_errno;
2205
2206         } else {
2207             err = c->read->kq_errno;
2208         }
2209
2210         c->log->action = "connecting to upstream";
2211         (void) ngx_connection_error(c, err,
2212             "kevent() reported that connect() failed");
2213         return NGX_ERROR;
2214     }
2215
2216 } else
2217 #endif
2218 {
2219     err = 0;
2220     len = sizeof(int);
2221
2222     /*
2223      * BSDs and Linux return 0 and set a pending error in err
2224      * Solaris returns -1 and sets errno
2225      */
2226
2227     if (getsockopt(c->fd, SOL_SOCKET, SO_ERROR, (void *) &err, &len)
2228         == -1)
2229     {
2230         err = ngx_socket_errno;
2231     }
2232
2233     if (err) {
2234         c->log->action = "connecting to upstream";
2235         (void) ngx_connection_error(c, err, "connect() failed");
2236         return NGX_ERROR;
2237     }
2238 }
2239
2240 return NGX_OK;
2241 }
2242
2243
2244 static ngx_int_t
2245 ngx_http_upstream_process_headers(ngx_http_request_t *r, ngx_http_upstream_t *u)
2246 {
2247     ngx_str_t          uri, args;
2248     ngx_uint_t         i, flags;
2249     ngx_list_part_t   *part;
2250     ngx_table_elt_t   *h;
2251     ngx_http_upstream_header_t *hh;
2252     ngx_http_upstream_main_conf_t *umcf;
2253
2254     umcf = ngx_http_get_module_main_conf(r, ngx_http_upstream_module);
2255
2256     if (u->headers_in.x_accel_redirect
2257         && !(u->conf->ignore_headers & NGX_HTTP_UPSTREAM_IGN_XA_REDIRECT))
2258     {
2259         ngx_http_upstream_finalize_request(r, u, NGX_DECLINED);
2260
2261         part = &u->headers_in.headers.part;
2262         h = part->elts;
2263
2264         for (i = 0; /* void */; i++) {
2265
2266             if (i >= part->nelts) {
2267                 if (part->next == NULL) {
2268                     break;
2269                 }
2270
2271                 part = part->next;
2272                 h = part->elts;
2273                 i = 0;
2274             }
2275
2276             hh = ngx_hash_find(&umcf->headers_in_hash, h[i].hash,
2277                 h[i].lowercase_key, h[i].key.len);

```

```

2278     if (hh && hh->redirect) {
2279         if (hh->copy_handler(r, &h[i], hh->conf) != NGX_OK) {
2280             ngx_http_finalize_request(r,
2281                                     NGX_HTTP_INTERNAL_SERVER_ERROR);
2282             return NGX_DONE;
2283         }
2284     }
2285 }
2286
2287 uri = u->headers_in.x_accel_redirect->value;
2288
2289 if (uri.data[0] == '@') {
2290     ngx_http_named_location(r, &uri);
2291
2292 } else {
2293     ngx_str_null(&args);
2294     flags = NGX_HTTP_LOG_UNSAFE;
2295
2296     if (ngx_http_parse_unsafe_uri(r, &uri, &args, &flags) != NGX_OK) {
2297         ngx_http_finalize_request(r, NGX_HTTP_NOT_FOUND);
2298         return NGX_DONE;
2299     }
2300
2301     if (r->method != NGX_HTTP_HEAD) {
2302         r->method = NGX_HTTP_GET;
2303     }
2304
2305     ngx_http_internal_redirect(r, &uri, &args);
2306 }
2307
2308 ngx_http_finalize_request(r, NGX_DONE);
2309 return NGX_DONE;
2310 }
2311
2312 part = &u->headers_in.headers.part;
2313 h = part->elts;
2314
2315 for (i = 0; /* void */; i++) {
2316     if (i >= part->nelts) {
2317         if (part->next == NULL) {
2318             break;
2319         }
2320     }
2321
2322     part = part->next;
2323     h = part->elts;
2324     i = 0;
2325 }
2326
2327 if (ngx_hash_find(&u->conf->hide_headers_hash, h[i].hash,
2328                 h[i].lowercase_key, h[i].key.len))
2329 {
2330     continue;
2331 }
2332
2333 hh = ngx_hash_find(&umcf->headers_in_hash, h[i].hash,
2334                   h[i].lowercase_key, h[i].key.len);
2335
2336 if (hh) {
2337     if (hh->copy_handler(r, &h[i], hh->conf) != NGX_OK) {
2338         ngx_http_upstream_finalize_request(r, u,
2339                                           NGX_HTTP_INTERNAL_SERVER_ERROR);
2340         return NGX_DONE;
2341     }
2342 }
2343
2344 continue;
2345 }
2346
2347 if (ngx_http_upstream_copy_header_line(r, &h[i], 0) != NGX_OK) {
2348     ngx_http_upstream_finalize_request(r, u,
2349                                       NGX_HTTP_INTERNAL_SERVER_ERROR);
2350     return NGX_DONE;
2351 }
2352 }
2353

```

```

2354     if (r->headers_out.server && r->headers_out.server->value.data == NULL) {
2355         r->headers_out.server->hash = 0;
2356     }
2357
2358     if (r->headers_out.date && r->headers_out.date->value.data == NULL) {
2359         r->headers_out.date->hash = 0;
2360     }
2361
2362     r->headers_out.status = u->headers_in.status_n;
2363     r->headers_out.status_line = u->headers_in.status_line;
2364
2365     r->headers_out.content_length_n = u->headers_in.content_length_n;
2366
2367     r->disable_not_modified = !u->cacheable;
2368
2369     if (u->conf->force_ranges) {
2370         r->allow_ranges = 1;
2371         r->single_range = 1;
2372
2373     #if (NGX_HTTP_CACHE)
2374         if (r->cached) {
2375             r->single_range = 0;
2376         }
2377     #endif
2378     }
2379
2380     u->length = -1;
2381
2382     return NGX_OK;
2383 }
2384
2385
2386 static void
2387 ngx_http_upstream_process_body_in_memory(ngx_http_request_t *r,
2388     ngx_http_upstream_t *u)
2389 {
2390     size_t      size;
2391     ssize_t     n;
2392     ngx_buf_t   *b;
2393     ngx_event_t *rev;
2394     ngx_connection_t *c;
2395
2396     c = u->peer.connection;
2397     rev = c->read;
2398
2399     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, c->log, 0,
2400         "http upstream process body on memory");
2401
2402     if (rev->timedout) {
2403         ngx_connection_error(c, NGX_ETIMEDOUT, "upstream timed out");
2404         ngx_http_upstream_finalize_request(r, u, NGX_HTTP_GATEWAY_TIME_OUT);
2405         return;
2406     }
2407
2408     b = &u->buffer;
2409
2410     for ( ;; ) {
2411
2412         size = b->end - b->last;
2413
2414         if (size == 0) {
2415             ngx_log_error(NGX_LOG_ALERT, c->log, 0,
2416                 "upstream buffer is too small to read response");
2417             ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
2418             return;
2419         }
2420
2421         n = c->recv(c, b->last, size);
2422
2423         if (n == NGX_AGAIN) {
2424             break;
2425         }
2426
2427         if (n == 0 || n == NGX_ERROR) {
2428             ngx_http_upstream_finalize_request(r, u, n);
2429             return;

```

```

2430     }
2431
2432     u->state->response_length += n;
2433
2434     if (u->input_filter(u->input_filter_ctx, n) == NGX_ERROR) {
2435         ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
2436         return;
2437     }
2438
2439     if (!rev->ready) {
2440         break;
2441     }
2442 }
2443
2444 if (u->length == 0) {
2445     ngx_http_upstream_finalize_request(r, u, 0);
2446     return;
2447 }
2448
2449 if (ngx_handle_read_event(rev, 0) != NGX_OK) {
2450     ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
2451     return;
2452 }
2453
2454 if (rev->active) {
2455     ngx_add_timer(rev, u->conf->read_timeout);
2456 }
2457 else if (rev->timer_set) {
2458     ngx_del_timer(rev);
2459 }
2460 }
2461
2462
2463 static void
2464 ngx_http_upstream_send_response(ngx_http_request_t *r, ngx_http_upstream_t *u)
2465 {
2466     int                tcp_nodelay;
2467     ssize_t           n;
2468     ngx_int_t         rc;
2469     ngx_event_pipe_t *p;
2470     ngx_connection_t *c;
2471     ngx_http_core_loc_conf_t *clcf;
2472
2473     rc = ngx_http_send_header(r);
2474
2475     if (rc == NGX_ERROR || rc > NGX_OK || r->post_action) {
2476         ngx_http_upstream_finalize_request(r, u, rc);
2477         return;
2478     }
2479
2480     u->header_sent = 1;
2481
2482     if (u->upgrade) {
2483         ngx_http_upstream_upgrade(r, u);
2484         return;
2485     }
2486
2487     c = r->connection;
2488
2489     if (r->header_only) {
2490         if (!u->buffering) {
2491             ngx_http_upstream_finalize_request(r, u, rc);
2492             return;
2493         }
2494     }
2495
2496     if (!u->cacheable && !u->store) {
2497         ngx_http_upstream_finalize_request(r, u, rc);
2498         return;
2499     }
2500
2501     u->pipe->downstream_error = 1;
2502 }
2503
2504 if (r->request_body && r->request_body->temp_file) {
2505     ngx_pool_run_cleanup_file(r->pool, r->request_body->temp_file->file.fd);

```

```

2506     r->request_body->temp_file->file.fd = NGX\_INVALID\_FILE;
2507 }
2508
2509 clcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_core\_module);
2510
2511 if (!u->buffering) {
2512
2513     if (u->input_filter == NULL) {
2514         u->input_filter_init = ngx\_http\_upstream\_non\_buffered\_filter\_init;
2515         u->input_filter = ngx\_http\_upstream\_non\_buffered\_filter;
2516         u->input_filter_ctx = r;
2517     }
2518
2519     u->read_event_handler = ngx\_http\_upstream\_process\_non\_buffered\_upstream;
2520     r->write_event_handler =
2521         ngx\_http\_upstream\_process\_non\_buffered\_downstream;
2522
2523     r->limit_rate = 0;
2524
2525     if (u->input_filter_init(u->input_filter_ctx) == NGX\_ERROR) {
2526         ngx\_http\_upstream\_finalize\_request(r, u, NGX\_ERROR);
2527         return;
2528     }
2529
2530     if (clcf->tcp_nodelay && c->tcp_nodelay == NGX\_TCP\_NODELAY\_UNSET) {
2531         ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, c->log, 0, "tcp_nodelay");
2532
2533         tcp_nodelay = 1;
2534
2535         if (setsockopt(c->fd, IPPROTO_TCP, TCP_NODELAY,
2536             (const void *) &tcp_nodelay, sizeof(int)) == -1)
2537         {
2538             ngx\_connection\_error(c, ngx\_socket\_errno,
2539                 "setsockopt(TCP_NODELAY) failed");
2540             ngx\_http\_upstream\_finalize\_request(r, u, NGX\_ERROR);
2541             return;
2542         }
2543
2544         c->tcp_nodelay = NGX\_TCP\_NODELAY\_SET;
2545     }
2546
2547     n = u->buffer.last - u->buffer.pos;
2548
2549     if (n) {
2550         u->buffer.last = u->buffer.pos;
2551
2552         u->state->response_length += n;
2553
2554         if (u->input_filter(u->input_filter_ctx, n) == NGX\_ERROR) {
2555             ngx\_http\_upstream\_finalize\_request(r, u, NGX\_ERROR);
2556             return;
2557         }
2558
2559         ngx\_http\_upstream\_process\_non\_buffered\_downstream(r);
2560
2561     } else {
2562         u->buffer.pos = u->buffer.start;
2563         u->buffer.last = u->buffer.start;
2564
2565         if (ngx\_http\_send\_special(r, NGX\_HTTP\_FLUSH) == NGX\_ERROR) {
2566             ngx\_http\_upstream\_finalize\_request(r, u, NGX\_ERROR);
2567             return;
2568         }
2569
2570         if (u->peer.connection->read->ready || u->length == 0) {
2571             ngx\_http\_upstream\_process\_non\_buffered\_upstream(r, u);
2572         }
2573     }
2574
2575     return;
2576 }
2577
2578 /* TODO: preallocate event_pipe bufs, look "Content-Length" */
2579
2580 #if (NGX\_HTTP\_CACHE)
2581

```

```

2582 if (r->cache && r->cache->file.fd != NGX\_INVALID\_FILE) {
2583     ngx\_pool\_run\_cleanup\_file(r->pool, r->cache->file.fd);
2584     r->cache->file.fd = NGX\_INVALID\_FILE;
2585 }
2586
2587 switch (ngx\_http\_test\_predicates(r, u->conf->no_cache)) {
2588
2589 case NGX\_ERROR:
2590     ngx\_http\_upstream\_finalize\_request(r, u, NGX\_ERROR);
2591     return;
2592
2593 case NGX\_DECLINED:
2594     u->cacheable = 0;
2595     break;
2596
2597 default: /* NGX\_OK */
2598
2599     if (u->cache_status == NGX\_HTTP\_CACHE\_BYPASS) {
2600
2601         /* create cache if previously bypassed */
2602
2603         if (ngx\_http\_file\_cache\_create(r) != NGX\_OK) {
2604             ngx\_http\_upstream\_finalize\_request(r, u, NGX\_ERROR);
2605             return;
2606         }
2607     }
2608
2609     break;
2610 }
2611
2612 if (u->cacheable) {
2613     time_t now, valid;
2614
2615     now = ngx\_time();
2616
2617     valid = r->cache->valid_sec;
2618
2619     if (valid == 0) {
2620         valid = ngx\_http\_file\_cache\_valid(u->conf->cache_valid,
2621                                           u->headers_in.status_n);
2622         if (valid) {
2623             r->cache->valid_sec = now + valid;
2624         }
2625     }
2626
2627     if (valid) {
2628         r->cache->date = now;
2629         r->cache->body_start = (u_short) (u->buffer.pos - u->buffer.start);
2630
2631         if (u->headers_in.status_n == NGX\_HTTP\_OK
2632             || u->headers_in.status_n == NGX\_HTTP\_PARTIAL\_CONTENT)
2633         {
2634             r->cache->last_modified = u->headers_in.last_modified_time;
2635
2636             if (u->headers_in.etag) {
2637                 r->cache->etag = u->headers_in.etag->value;
2638             }
2639         }
2640
2641         if (ngx\_http\_file\_cache\_set\_header(r, u->buffer.start) != NGX\_OK) {
2642             ngx\_http\_upstream\_finalize\_request(r, u, NGX\_ERROR);
2643             return;
2644         }
2645     }
2646     } else {
2647         u->cacheable = 0;
2648     }
2649 }
2650
2651 ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, c->log, 0,
2652               "http cacheable: %d", u->cacheable);
2653
2654 if (u->cacheable == 0 && r->cache) {
2655     ngx\_http\_file\_cache\_free(r->cache, u->pipe->temp_file);
2656 }
2657

```



```

2658 #endif
2659
2660     p = u->pipe;
2661
2662     p->output_filter = (ngx\_event\_pipe\_output\_filter\_pt) ngx\_http\_output\_filter;
2663     p->output_ctx = r;
2664     p->tag = u->output.tag;
2665     p->bufs = u->conf->bufs;
2666     p->busy_size = u->conf->busy_buffers_size;
2667     p->upstream = u->peer.connection;
2668     p->downstream = c;
2669     p->pool = r->pool;
2670     p->log = c->log;
2671     p->limit_rate = u->conf->limit_rate;
2672     p->start_sec = ngx\_time();
2673
2674     p->cacheable = u->cacheable || u->store;
2675
2676     p->temp_file = ngx\_palloc(r->pool, sizeof\(ngx\_temp\_file\_t\));
2677     if (p->temp_file == NULL) {
2678         ngx\_http\_upstream\_finalize\_request(r, u, NGX\_ERROR);
2679         return;
2680     }
2681
2682     p->temp_file->file.fd = NGX\_INVALID\_FILE;
2683     p->temp_file->file.log = c->log;
2684     p->temp_file->path = u->conf->temp_path;
2685     p->temp_file->pool = r->pool;
2686
2687     if (p->cacheable) {
2688         p->temp_file->persistent = 1;
2689
2690 #if (NGX_HTTP_CACHE)
2691         if (r->cache && r->cache->file_cache->temp_path) {
2692             p->temp_file->path = r->cache->file_cache->temp_path;
2693         }
2694 #endif
2695     } else {
2696         p->temp_file->log_level = NGX\_LOG\_WARN;
2697         p->temp_file->warn = "an upstream response is buffered "
2698             "to a temporary file";
2699     }
2700
2701     p->max_temp_file_size = u->conf->max_temp_file_size;
2702     p->temp_file_write_size = u->conf->temp_file_write_size;
2703
2704     p->preread_bufs = ngx\_alloc\_chain\_link(r->pool);
2705     if (p->preread_bufs == NULL) {
2706         ngx\_http\_upstream\_finalize\_request(r, u, NGX\_ERROR);
2707         return;
2708     }
2709
2710     p->preread_bufs->buf = &u->buffer;
2711     p->preread_bufs->next = NULL;
2712     u->buffer.recycled = 1;
2713
2714     p->preread_size = u->buffer.last - u->buffer.pos;
2715
2716     if (u->cacheable) {
2717         p->buf_to_file = ngx\_calloc\_buf(r->pool);
2718         if (p->buf_to_file == NULL) {
2719             ngx\_http\_upstream\_finalize\_request(r, u, NGX\_ERROR);
2720             return;
2721         }
2722
2723         p->buf_to_file->start = u->buffer.start;
2724         p->buf_to_file->pos = u->buffer.start;
2725         p->buf_to_file->last = u->buffer.pos;
2726         p->buf_to_file->temporary = 1;
2727     }
2728
2729     if (ngx\_event\_flags & NGX\_USE\_AIO\_EVENT) {
2730         /* the posted aio operation may corrupt a shadow buffer */
2731         p->single_buf = 1;
2732     }
2733

```

```

2734 }
2735
2736 /* TODO: p->free_bufs = 0 if use ngx_create_chain_of_bufs() */
2737 p->free_bufs = 1;
2738
2739 /*
2740  * event_pipe would do u->buffer.last += p->preread_size
2741  * as though these bytes were read
2742  */
2743 u->buffer.last = u->buffer.pos;
2744
2745 if (u->conf->cyclic_temp_file) {
2746
2747     /*
2748      * we need to disable the use of sendfile() if we use cyclic temp file
2749      * because the writing a new data may interfere with sendfile()
2750      * that uses the same kernel file pages (at least on FreeBSD)
2751      */
2752
2753     p->cyclic_temp_file = 1;
2754     c->sendfile = 0;
2755
2756 } else {
2757     p->cyclic_temp_file = 0;
2758 }
2759
2760 p->read_timeout = u->conf->read_timeout;
2761 p->send_timeout = clcf->send_timeout;
2762 p->send_lowat = clcf->send_lowat;
2763
2764 p->length = -1;
2765
2766 if (u->input_filter_init
2767     && u->input_filter_init(p->input_ctx) != NGX_OK)
2768 {
2769     ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
2770     return;
2771 }
2772
2773 u->read_event_handler = ngx_http_upstream_process_upstream;
2774 r->write_event_handler = ngx_http_upstream_process_downstream;
2775
2776 ngx_http_upstream_process_upstream(r, u);
2777 }
2778
2779
2780 static void
2781 ngx_http_upstream_upgrade(ngx_http_request_t *r, ngx_http_upstream_t *u)
2782 {
2783     int                tcp_nodelay;
2784     ngx_connection_t  *c;
2785     ngx_http_core_loc_conf_t *clcf;
2786
2787     c = r->connection;
2788     clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
2789
2790     /* TODO: prevent upgrade if not requested or not possible */
2791
2792     r->keepalive = 0;
2793     c->log->action = "proxying upgraded connection";
2794
2795     u->read_event_handler = ngx_http_upstream_upgraded_read_upstream;
2796     u->write_event_handler = ngx_http_upstream_upgraded_write_upstream;
2797     r->read_event_handler = ngx_http_upstream_upgraded_read_downstream;
2798     r->write_event_handler = ngx_http_upstream_upgraded_write_downstream;
2799
2800     if (clcf->tcp_nodelay) {
2801         tcp_nodelay = 1;
2802
2803         if (c->tcp_nodelay == NGX_TCP_NODELAY_UNSET) {
2804             ngx_log_debug0(NGX_LOG_DEBUG_HTTP, c->log, 0, "tcp_nodelay");
2805
2806             if (setsockopt(c->fd, IPPROTO_TCP, TCP_NODELAY,
2807                 (const void *) &tcp_nodelay, sizeof(int)) == -1)
2808             {
2809                 ngx_connection_error(c, ngx_socket_errno,

```

```

2810         "setsockopt(TCP_NODELAY) failed");
2811         ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
2812         return;
2813     }
2814
2815     c->tcp_nodelay = NGX_TCP_NODELAY_SET;
2816 }
2817
2818 if (u->peer.connection->tcp_nodelay == NGX_TCP_NODELAY_UNSET) {
2819     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, u->peer.connection->log, 0,
2820         "tcp_nodelay");
2821
2822     if (setsockopt(u->peer.connection->fd, IPPROTO_TCP, TCP_NODELAY,
2823         (const void *) &tcp_nodelay, sizeof(int)) == -1)
2824     {
2825         ngx_connection_error(u->peer.connection, ngx_socket_errno,
2826             "setsockopt(TCP_NODELAY) failed");
2827         ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
2828         return;
2829     }
2830
2831     u->peer.connection->tcp_nodelay = NGX_TCP_NODELAY_SET;
2832 }
2833 }
2834
2835 if (ngx_http_send_special(r, NGX_HTTP_FLUSH) == NGX_ERROR) {
2836     ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
2837     return;
2838 }
2839
2840 if (u->peer.connection->read->ready
2841     || u->buffer.pos != u->buffer.last)
2842 {
2843     ngx_post_event(c->read, &ngx_posted_events);
2844     ngx_http_upstream_process_upgraded(r, 1, 1);
2845     return;
2846 }
2847
2848 ngx_http_upstream_process_upgraded(r, 0, 1);
2849 }
2850
2851
2852 static void
2853 ngx_http_upstream_upgraded_read_downstream(ngx_http_request_t *r)
2854 {
2855     ngx_http_upstream_process_upgraded(r, 0, 0);
2856 }
2857
2858
2859 static void
2860 ngx_http_upstream_upgraded_write_downstream(ngx_http_request_t *r)
2861 {
2862     ngx_http_upstream_process_upgraded(r, 1, 1);
2863 }
2864
2865
2866 static void
2867 ngx_http_upstream_upgraded_read_upstream(ngx_http_request_t *r,
2868     ngx_http_upstream_t *u)
2869 {
2870     ngx_http_upstream_process_upgraded(r, 1, 0);
2871 }
2872
2873
2874 static void
2875 ngx_http_upstream_upgraded_write_upstream(ngx_http_request_t *r,
2876     ngx_http_upstream_t *u)
2877 {
2878     ngx_http_upstream_process_upgraded(r, 0, 1);
2879 }
2880
2881
2882 static void
2883 ngx_http_upstream_process_upgraded(ngx_http_request_t *r,
2884     ngx_uint_t from_upstream, ngx_uint_t do_write)
2885 {

```

```

2886     size_t                size;
2887     ssize_t               n;
2888     ngx_buf_t             *b;
2889     ngx_connection_t      *c, *downstream, *upstream, *dst, *src;
2890     ngx_http_upstream_t   *u;
2891     ngx_http_core_loc_conf_t *clcf;
2892
2893     c = r->connection;
2894     u = r->upstream;
2895
2896     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, c->log, 0,
2897                  "http upstream process upgraded, fu:%ui", from_upstream);
2898
2899     downstream = c;
2900     upstream = u->peer.connection;
2901
2902     if (downstream->write->timedout) {
2903         c->timedout = 1;
2904         ngx_connection_error(c, NGX_ETIMEDOUT, "client timed out");
2905         ngx_http_upstream_finalize_request(r, u, NGX_HTTP_REQUEST_TIME_OUT);
2906         return;
2907     }
2908
2909     if (upstream->read->timedout || upstream->write->timedout) {
2910         ngx_connection_error(c, NGX_ETIMEDOUT, "upstream timed out");
2911         ngx_http_upstream_finalize_request(r, u, NGX_HTTP_GATEWAY_TIME_OUT);
2912         return;
2913     }
2914
2915     if (from_upstream) {
2916         src = upstream;
2917         dst = downstream;
2918         b = &u->buffer;
2919
2920     } else {
2921         src = downstream;
2922         dst = upstream;
2923         b = &u->from_client;
2924
2925         if (r->header_in->last > r->header_in->pos) {
2926             b = r->header_in;
2927             b->end = b->last;
2928             do_write = 1;
2929         }
2930
2931         if (b->start == NULL) {
2932             b->start = ngx_palloc(r->pool, u->conf->buffer_size);
2933             if (b->start == NULL) {
2934                 ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
2935                 return;
2936             }
2937
2938             b->pos = b->start;
2939             b->last = b->start;
2940             b->end = b->start + u->conf->buffer_size;
2941             b->temporary = 1;
2942             b->tag = u->output.tag;
2943         }
2944     }
2945
2946     for ( ;; ) {
2947
2948         if (do_write) {
2949
2950             size = b->last - b->pos;
2951
2952             if (size && dst->write->ready) {
2953
2954                 n = dst->send(dst, b->pos, size);
2955
2956                 if (n == NGX_ERROR) {
2957                     ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
2958                     return;
2959                 }
2960
2961                 if (n > 0) {

```

```

2962         b->pos += n;
2963
2964         if (b->pos == b->last) {
2965             b->pos = b->start;
2966             b->last = b->start;
2967         }
2968     }
2969 }
2970 }
2971
2972 size = b->end - b->last;
2973
2974 if (size && src->read->ready) {
2975
2976     n = src->recv(src, b->last, size);
2977
2978     if (n == NGX\_AGAIN || n == 0) {
2979         break;
2980     }
2981
2982     if (n > 0) {
2983         do_write = 1;
2984         b->last += n;
2985
2986         continue;
2987     }
2988
2989     if (n == NGX\_ERROR) {
2990         src->read->eof = 1;
2991     }
2992 }
2993
2994 break;
2995 }
2996
2997 if ((upstream->read->eof && u->buffer.pos == u->buffer.last)
2998     || (downstream->read->eof && u->from_client.pos == u->from_client.last)
2999     || (downstream->read->eof && upstream->read->eof))
3000 {
3001     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, c->log, 0,
3002                 "http upstream upgraded done");
3003     ngx\_http\_upstream\_finalize\_request(r, u, 0);
3004     return;
3005 }
3006
3007 clcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_core\_module);
3008
3009 if (ngx\_handle\_write\_event(upstream->write, u->conf->send_lowat)
3010     != NGX\_OK)
3011 {
3012     ngx\_http\_upstream\_finalize\_request(r, u, NGX\_ERROR);
3013     return;
3014 }
3015
3016 if (upstream->write->active && !upstream->write->ready) {
3017     ngx\_add\_timer(upstream->write, u->conf->send_timeout);
3018 }
3019 else if (upstream->write->timer_set) {
3020     ngx\_del\_timer(upstream->write);
3021 }
3022
3023 if (ngx\_handle\_read\_event(upstream->read, 0) != NGX\_OK) {
3024     ngx\_http\_upstream\_finalize\_request(r, u, NGX\_ERROR);
3025     return;
3026 }
3027
3028 if (upstream->read->active && !upstream->read->ready) {
3029     ngx\_add\_timer(upstream->read, u->conf->read_timeout);
3030 }
3031 else if (upstream->read->timer_set) {
3032     ngx\_del\_timer(upstream->read);
3033 }
3034
3035 if (ngx\_handle\_write\_event(downstream->write, clcf->send_lowat)
3036     != NGX\_OK)
3037 {

```

```

3038     ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
3039     return;
3040 }
3041
3042 if (ngx_handle_read_event(downstream->read, 0) != NGX_OK) {
3043     ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
3044     return;
3045 }
3046
3047 if (downstream->write->active && !downstream->write->ready) {
3048     ngx_add_timer(downstream->write, clcf->send_timeout);
3049
3050 } else if (downstream->write->timer_set) {
3051     ngx_del_timer(downstream->write);
3052 }
3053 }
3054
3055
3056 static void
3057 ngx_http_upstream_process_non_buffered_downstream(ngx_http_request_t *r)
3058 {
3059     ngx_event_t      *wev;
3060     ngx_connection_t *c;
3061     ngx_http_upstream_t *u;
3062
3063     c = r->connection;
3064     u = r->upstream;
3065     wev = c->write;
3066
3067     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, c->log, 0,
3068                  "http upstream process non buffered downstream");
3069
3070     c->log->action = "sending to client";
3071
3072     if (wev->timedout) {
3073         c->timedout = 1;
3074         ngx_connection_error(c, NGX_ETIMEDOUT, "client timed out");
3075         ngx_http_upstream_finalize_request(r, u, NGX_HTTP_REQUEST_TIME_OUT);
3076         return;
3077     }
3078
3079     ngx_http_upstream_process_non_buffered_request(r, 1);
3080 }
3081
3082
3083 static void
3084 ngx_http_upstream_process_non_buffered_upstream(ngx_http_request_t *r,
3085         ngx_http_upstream_t *u)
3086 {
3087     ngx_connection_t *c;
3088
3089     c = u->peer.connection;
3090
3091     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, c->log, 0,
3092                  "http upstream process non buffered upstream");
3093
3094     c->log->action = "reading upstream";
3095
3096     if (c->read->timedout) {
3097         ngx_connection_error(c, NGX_ETIMEDOUT, "upstream timed out");
3098         ngx_http_upstream_finalize_request(r, u, NGX_HTTP_GATEWAY_TIME_OUT);
3099         return;
3100     }
3101
3102     ngx_http_upstream_process_non_buffered_request(r, 0);
3103 }
3104
3105
3106 static void
3107 ngx_http_upstream_process_non_buffered_request(ngx_http_request_t *r,
3108         ngx_uint_t do_write)
3109 {
3110     size_t      size;
3111     ssize_t     n;
3112     ngx_buf_t   *b;
3113     ngx_int_t   rc;

```

```

3114 ngx\_connection\_t *downstream, *upstream;
3115 ngx\_http\_upstream\_t *u;
3116 ngx\_http\_core\_loc\_conf\_t *clcf;
3117
3118 u = r->upstream;
3119 downstream = r->connection;
3120 upstream = u->peer.connection;
3121
3122 b = &u->buffer;
3123
3124 do_write = do_write || u->length == 0;
3125
3126 for ( ;; ) {
3127
3128     if (do_write) {
3129
3130         if (u->out_bufs || u->busy_bufs) {
3131             rc = ngx\_http\_output\_filter(r, u->out_bufs);
3132
3133             if (rc == NGX\_ERROR) {
3134                 ngx\_http\_upstream\_finalize\_request(r, u, NGX\_ERROR);
3135                 return;
3136             }
3137
3138             ngx\_chain\_update\_chains(r->pool, &u->free_bufs, &u->busy_bufs,
3139                                 &u->out_bufs, u->output.tag);
3140         }
3141
3142         if (u->busy_bufs == NULL) {
3143
3144             if (u->length == 0
3145                 || (upstream->read->eof && u->length == -1))
3146             {
3147                 ngx\_http\_upstream\_finalize\_request(r, u, 0);
3148                 return;
3149             }
3150
3151             if (upstream->read->eof) {
3152                 ngx\_log\_error(NGX\_LOG\_ERR, upstream->log, 0,
3153                             "upstream prematurely closed connection");
3154
3155                 ngx\_http\_upstream\_finalize\_request(r, u,
3156                                                 NGX\_HTTP\_BAD\_GATEWAY);
3157                 return;
3158             }
3159
3160             if (upstream->read->error) {
3161                 ngx\_http\_upstream\_finalize\_request(r, u,
3162                                                 NGX\_HTTP\_BAD\_GATEWAY);
3163                 return;
3164             }
3165
3166             b->pos = b->start;
3167             b->last = b->start;
3168         }
3169     }
3170
3171     size = b->end - b->last;
3172
3173     if (size && upstream->read->ready) {
3174
3175         n = upstream->recv(upstream, b->last, size);
3176
3177         if (n == NGX\_AGAIN) {
3178             break;
3179         }
3180
3181         if (n > 0) {
3182             u->state->response_length += n;
3183
3184             if (u->input_filter(u->input_filter_ctx, n) == NGX\_ERROR) {
3185                 ngx\_http\_upstream\_finalize\_request(r, u, NGX\_ERROR);
3186                 return;
3187             }
3188         }
3189     }

```

```

3190         do_write = 1;
3191
3192         continue;
3193     }
3194
3195     break;
3196 }
3197
3198 clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
3199
3200 if (downstream->data == r) {
3201     if (ngx_handle_write_event(downstream->write, clcf->send_lowat)
3202         != NGX_OK)
3203     {
3204         ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
3205         return;
3206     }
3207 }
3208
3209 if (downstream->write->active && !downstream->write->ready) {
3210     ngx_add_timer(downstream->write, clcf->send_timeout);
3211 }
3212 else if (downstream->write->timer_set) {
3213     ngx_del_timer(downstream->write);
3214 }
3215
3216 if (ngx_handle_read_event(upstream->read, 0) != NGX_OK) {
3217     ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
3218     return;
3219 }
3220
3221 if (upstream->read->active && !upstream->read->ready) {
3222     ngx_add_timer(upstream->read, u->conf->read_timeout);
3223 }
3224 else if (upstream->read->timer_set) {
3225     ngx_del_timer(upstream->read);
3226 }
3227 }
3228
3229
3230 static ngx_int_t
3231 ngx_http_upstream_non_buffered_filter_init(void *data)
3232 {
3233     return NGX_OK;
3234 }
3235
3236
3237 static ngx_int_t
3238 ngx_http_upstream_non_buffered_filter(void *data, ssize_t bytes)
3239 {
3240     ngx_http_request_t *r = data;
3241
3242     ngx_buf_t *b;
3243     ngx_chain_t *cl, **ll;
3244     ngx_http_upstream_t *u;
3245
3246     u = r->upstream;
3247
3248     for (cl = u->out_bufs, ll = &u->out_bufs; cl; cl = cl->next) {
3249         ll = &cl->next;
3250     }
3251
3252     cl = ngx_chain_get_free_buf(r->pool, &u->free_bufs);
3253     if (cl == NULL) {
3254         return NGX_ERROR;
3255     }
3256
3257     *ll = cl;
3258
3259     cl->buf->flush = 1;
3260     cl->buf->memory = 1;
3261
3262     b = &u->buffer;
3263
3264     cl->buf->pos = b->last;
3265     b->last += bytes;

```



```

3266     cl->buf->last = b->last;
3267     cl->buf->tag = u->output.tag;
3268
3269     if (u->length == -1) {
3270         return NGX\_OK;
3271     }
3272
3273     u->length -= bytes;
3274
3275     return NGX\_OK;
3276 }
3277
3278
3279 static void
3280 ngx_http_upstream_process_downstream(ngx\_http\_request\_t *r)
3281 {
3282     ngx\_event\_t          *wev;
3283     ngx\_connection\_t    *c;
3284     ngx\_event\_pipe\_t    *p;
3285     ngx\_http\_upstream\_t *u;
3286
3287     c = r->connection;
3288     u = r->upstream;
3289     p = u->pipe;
3290     wev = c->write;
3291
3292     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, c->log, 0,
3293                 "http upstream process downstream");
3294
3295     c->log->action = "sending to client";
3296
3297     if (wev->timedout) {
3298
3299         if (wev->delayed) {
3300
3301             wev->timedout = 0;
3302             wev->delayed = 0;
3303
3304             if (!wev->ready) {
3305                 ngx\_add\_timer(wev, p->send_timeout);
3306
3307                 if (ngx\_handle\_write\_event(wev, p->send_lowat) != NGX\_OK) {
3308                     ngx\_http\_upstream\_finalize\_request(r, u, NGX\_ERROR);
3309                 }
3310
3311                 return;
3312             }
3313
3314             if (ngx\_event\_pipe(p, wev->write) == NGX\_ABORT) {
3315                 ngx\_http\_upstream\_finalize\_request(r, u, NGX\_ERROR);
3316                 return;
3317             }
3318
3319         } else {
3320             p->downstream_error = 1;
3321             c->timedout = 1;
3322             ngx\_connection\_error(c, NGX\_ETIMEDOUT, "client timed out");
3323         }
3324     } else {
3325
3326         if (wev->delayed) {
3327
3328             ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, c->log, 0,
3329                 "http downstream delayed");
3330
3331             if (ngx\_handle\_write\_event(wev, p->send_lowat) != NGX\_OK) {
3332                 ngx\_http\_upstream\_finalize\_request(r, u, NGX\_ERROR);
3333             }
3334
3335             return;
3336         }
3337     }
3338
3339     if (ngx\_event\_pipe(p, 1) == NGX\_ABORT) {
3340         ngx\_http\_upstream\_finalize\_request(r, u, NGX\_ERROR);
3341         return;

```

```

3342     }
3343 }
3344
3345 ngx\_http\_upstream\_process\_request(r);
3346 }
3347
3348
3349 static void
3350 ngx\_http\_upstream\_process\_upstream(ngx\_http\_request\_t *r,
3351 ngx\_http\_upstream\_t *u)
3352 {
3353     ngx\_event\_t *rev;
3354     ngx\_event\_pipe\_t *p;
3355     ngx\_connection\_t *c;
3356
3357     c = u->peer.connection;
3358     p = u->pipe;
3359     rev = c->read;
3360
3361     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, c->log, 0,
3362         "http upstream process upstream");
3363
3364     c->log->action = "reading upstream";
3365
3366     if (rev->timedout) {
3367         if (rev->delayed) {
3368             rev->timedout = 0;
3369             rev->delayed = 0;
3370
3371             if (!rev->ready) {
3372                 ngx\_add\_timer(rev, p->read_timeout);
3373
3374                 if (ngx\_handle\_read\_event(rev, 0) != NGX\_OK) {
3375                     ngx\_http\_upstream\_finalize\_request(r, u, NGX\_ERROR);
3376                 }
3377
3378                 return;
3379             }
3380
3381             if (ngx\_event\_pipe(p, 0) == NGX\_ABORT) {
3382                 ngx\_http\_upstream\_finalize\_request(r, u, NGX\_ERROR);
3383                 return;
3384             }
3385
3386         } else {
3387             p->upstream_error = 1;
3388             ngx\_connection\_error(c, NGX\_ETIMEDOUT, "upstream timed out");
3389         }
3390     } else {
3391         if (rev->delayed) {
3392             ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, c->log, 0,
3393                 "http upstream delayed");
3394
3395             if (ngx\_handle\_read\_event(rev, 0) != NGX\_OK) {
3396                 ngx\_http\_upstream\_finalize\_request(r, u, NGX\_ERROR);
3397             }
3398
3399             return;
3400         }
3401
3402         if (ngx\_event\_pipe(p, 0) == NGX\_ABORT) {
3403             ngx\_http\_upstream\_finalize\_request(r, u, NGX\_ERROR);
3404             return;
3405         }
3406     }
3407
3408     ngx\_http\_upstream\_process\_request(r);
3409 }
3410
3411
3412
3413 ngx\_http\_upstream\_process\_request(r);
3414 }
3415
3416
3417 static void

```

```

3418 ngx_http_upstream_process_request(ngx_http_request_t *r)
3419 {
3420     ngx_temp_file_t    *tf;
3421     ngx_event_pipe_t   *p;
3422     ngx_http_upstream_t *u;
3423
3424     u = r->upstream;
3425     p = u->pipe;
3426
3427     if (u->peer.connection) {
3428
3429         if (u->store) {
3430
3431             if (p->upstream_eof || p->upstream_done) {
3432
3433                 tf = p->temp_file;
3434
3435                 if (u->headers_in.status_n == NGX_HTTP_OK
3436                     && (p->upstream_done || p->length == -1)
3437                     && (u->headers_in.content_length_n == -1
3438                         || u->headers_in.content_length_n == tf->offset))
3439                 {
3440                     ngx_http_upstream_store(r, u);
3441                 }
3442             }
3443         }
3444
3445         #if (NGX_HTTP_CACHE)
3446
3447         if (u->cacheable) {
3448
3449             if (p->upstream_done) {
3450                 ngx_http_file_cache_update(r, p->temp_file);
3451
3452             } else if (p->upstream_eof) {
3453
3454                 tf = p->temp_file;
3455
3456                 if (p->length == -1
3457                     && (u->headers_in.content_length_n == -1
3458                         || u->headers_in.content_length_n
3459                             == tf->offset - (off_t) r->cache->body_start))
3460                 {
3461                     ngx_http_file_cache_update(r, tf);
3462
3463                 } else {
3464                     ngx_http_file_cache_free(r->cache, tf);
3465                 }
3466
3467             } else if (p->upstream_error) {
3468                 ngx_http_file_cache_free(r->cache, p->temp_file);
3469             }
3470         }
3471
3472         #endif
3473
3474         if (p->upstream_done || p->upstream_eof || p->upstream_error) {
3475             ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
3476                 "http upstream exit: %p", p->out);
3477
3478             if (p->upstream_done
3479                 || (p->upstream_eof && p->length == -1))
3480             {
3481                 ngx_http_upstream_finalize_request(r, u, 0);
3482                 return;
3483             }
3484
3485             if (p->upstream_eof) {
3486                 ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
3487                     "upstream prematurely closed connection");
3488             }
3489
3490             ngx_http_upstream_finalize_request(r, u, NGX_HTTP_BAD_GATEWAY);
3491             return;
3492         }
3493     }

```

```

3494     if (p->downstream_error) {
3495         ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
3496             "http upstream downstream error");
3497     }
3498     if (!u->cacheable && !u->store && u->peer.connection) {
3499         ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
3500     }
3501 }
3502 }
3503 }
3504 }
3505 }
3506 static void
3507 ngx_http_upstream_store(ngx_http_request_t *r, ngx_http_upstream_t *u)
3508 {
3509     size_t          root;
3510     time_t          lm;
3511     ngx_str_t       path;
3512     ngx_temp_file_t *tf;
3513     ngx_ext_rename_file_t  ext;
3514
3515     tf = u->pipe->temp_file;
3516
3517     if (tf->file.fd == NGX_INVALID_FILE) {
3518         /* create file for empty 200 response */
3519
3520         tf = ngx_palloc(r->pool, sizeof(ngx_temp_file_t));
3521         if (tf == NULL) {
3522             return;
3523         }
3524
3525         tf->file.fd = NGX_INVALID_FILE;
3526         tf->file.log = r->connection->log;
3527         tf->path = u->conf->temp_path;
3528         tf->pool = r->pool;
3529         tf->persistent = 1;
3530
3531         if (ngx_create_temp_file(&tf->file, tf->path, tf->pool,
3532             tf->persistent, tf->clean, tf->access)
3533             != NGX_OK)
3534         {
3535             return;
3536         }
3537     }
3538
3539     u->pipe->temp_file = tf;
3540 }
3541
3542 ext.access = u->conf->store_access;
3543 ext.path_access = u->conf->store_access;
3544 ext.time = -1;
3545 ext.create_path = 1;
3546 ext.delete_file = 1;
3547 ext.log = r->connection->log;
3548
3549 if (u->headers_in.last_modified) {
3550
3551     lm = ngx_http_parse_time(u->headers_in.last_modified->value.data,
3552         u->headers_in.last_modified->value.len);
3553
3554     if (lm != NGX_ERROR) {
3555         ext.time = lm;
3556         ext.fd = tf->file.fd;
3557     }
3558 }
3559
3560 if (u->conf->store_lengths == NULL) {
3561
3562     if (ngx_http_map_uri_to_path(r, &path, &root, 0) == NULL) {
3563         return;
3564     }
3565
3566 } else {
3567     if (ngx_http_script_run(r, &path, u->conf->store_lengths->elts, 0,
3568         u->conf->store_values->elts)
3569         == NULL)

```

```

3570     {
3571         return;
3572     }
3573 }
3574
3575 path.len--;
3576
3577 ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
3578     "upstream stores \"%s\" to \"%s\"",
3579     tf->file.name.data, path.data);
3580
3581 (void) ngx_ext_rename_file(&tf->file.name, &path, &ext);
3582
3583 u->store = 0;
3584 }
3585
3586
3587 static void
3588 ngx_http_upstream_dummy_handler(ngx_http_request_t *r, ngx_http_upstream_t *u)
3589 {
3590     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
3591         "http upstream dummy handler");
3592 }
3593
3594
3595 static void
3596 ngx_http_upstream_next(ngx_http_request_t *r, ngx_http_upstream_t *u,
3597     ngx_uint_t ft_type)
3598 {
3599     ngx_msec_t timeout;
3600     ngx_uint_t status, state;
3601
3602     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
3603         "http next upstream, %xi", ft_type);
3604
3605     if (u->peer.sockaddr) {
3606
3607         if (ft_type == NGX_HTTP_UPSTREAM_FT_HTTP_403
3608             || ft_type == NGX_HTTP_UPSTREAM_FT_HTTP_404)
3609         {
3610             state = NGX_PEER_NEXT;
3611
3612         } else {
3613             state = NGX_PEER_FAILED;
3614         }
3615
3616         u->peer.free(&u->peer, u->peer.data, state);
3617         u->peer.sockaddr = NULL;
3618     }
3619
3620     if (ft_type == NGX_HTTP_UPSTREAM_FT_TIMEOUT) {
3621         ngx_log_error(NGX_LOG_ERR, r->connection->log, NGX_ETIMEDOUT,
3622             "upstream timed out");
3623     }
3624
3625     if (u->peer.cached && ft_type == NGX_HTTP_UPSTREAM_FT_ERROR) {
3626         status = 0;
3627
3628         /* TODO: inform balancer instead */
3629
3630         u->peer.tries++;
3631     } else {
3632         switch (ft_type) {
3633
3634             case NGX_HTTP_UPSTREAM_FT_TIMEOUT:
3635                 status = NGX_HTTP_GATEWAY_TIME_OUT;
3636                 break;
3637
3638             case NGX_HTTP_UPSTREAM_FT_HTTP_500:
3639                 status = NGX_HTTP_INTERNAL_SERVER_ERROR;
3640                 break;
3641
3642             case NGX_HTTP_UPSTREAM_FT_HTTP_403:
3643                 status = NGX_HTTP_FORBIDDEN;
3644                 break;
3645

```

```

3646 case NGX_HTTP_UPSTREAM_FT_HTTP_404:
3647     status = NGX_HTTP_NOT_FOUND;
3648     break;
3649
3650
3651 /*
3652  * NGX_HTTP_UPSTREAM_FT_BUSY_LOCK and NGX_HTTP_UPSTREAM_FT_MAX_WAITING
3653  * never reach here
3654  */
3655
3656 default:
3657     status = NGX_HTTP_BAD_GATEWAY;
3658 }
3659 }
3660
3661 if (r->connection->error) {
3662     ngx_http_upstream_finalize_request(r, u,
3663                                         NGX_HTTP_CLIENT_CLOSED_REQUEST);
3664     return;
3665 }
3666
3667 if (status) {
3668     u->state->status = status;
3669     timeout = u->conf->next_upstream_timeout;
3670
3671     if (u->peer.tries == 0
3672         || !(u->conf->next_upstream & ft_type)
3673         || (timeout && ngx_current_msec - u->peer.start_time >= timeout))
3674     {
3675 #if (NGX_HTTP_CACHE)
3676
3677         if (u->cache_status == NGX_HTTP_CACHE_EXPIRED
3678             && (u->conf->cache_use_stale & ft_type))
3679         {
3680             ngx_int_t rc;
3681
3682             rc = u->reinit_request(r);
3683
3684             if (rc == NGX_OK) {
3685                 u->cache_status = NGX_HTTP_CACHE_STALE;
3686                 rc = ngx_http_upstream_cache_send(r, u);
3687             }
3688
3689             ngx_http_upstream_finalize_request(r, u, rc);
3690             return;
3691         }
3692 #endif
3693
3694         ngx_http_upstream_finalize_request(r, u, status);
3695         return;
3696     }
3697 }
3698
3699 if (u->peer.connection) {
3700     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
3701                   "close http upstream connection: %d",
3702                   u->peer.connection->fd);
3703 #if (NGX_HTTP_SSL)
3704
3705     if (u->peer.connection->ssl) {
3706         u->peer.connection->ssl->no_wait_shutdown = 1;
3707         u->peer.connection->ssl->no_send_shutdown = 1;
3708
3709         (void) ngx_ssl_shutdown(u->peer.connection);
3710     }
3711 #endif
3712
3713     if (u->peer.connection->pool) {
3714         ngx_destroy_pool(u->peer.connection->pool);
3715     }
3716
3717     ngx_close_connection(u->peer.connection);
3718     u->peer.connection = NULL;
3719 }
3720
3721 ngx_http_upstream_connect(r, u);

```

```

3722 }
3723
3724
3725 static void
3726 ngx_http_upstream_cleanup(void *data)
3727 {
3728     ngx_http_request_t *r = data;
3729
3730     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
3731                 "cleanup http upstream request: \"%V\"", &r->uri);
3732
3733     ngx_http_upstream_finalize_request(r, r->upstream, NGX_DONE);
3734 }
3735
3736
3737 static void
3738 ngx_http_upstream_finalize_request(ngx_http_request_t *r,
3739     ngx_http_upstream_t *u, ngx_int_t rc)
3740 {
3741     ngx_uint_t flush;
3742     ngx_time_t *tp;
3743
3744     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
3745                 "finalize http upstream request: %i", rc);
3746
3747     if (u->cleanup) {
3748         *u->cleanup = NULL;
3749         u->cleanup = NULL;
3750     }
3751
3752     if (u->resolved && u->resolved->ctx) {
3753         ngx_resolve_name_done(u->resolved->ctx);
3754         u->resolved->ctx = NULL;
3755     }
3756
3757     if (u->state && u->state->response_sec) {
3758         tp = ngx_timeofday();
3759         u->state->response_sec = tp->sec - u->state->response_sec;
3760         u->state->response_msec = tp->msec - u->state->response_msec;
3761
3762         if (u->pipe && u->pipe->read_length) {
3763             u->state->response_length = u->pipe->read_length;
3764         }
3765     }
3766
3767     u->finalize_request(r, rc);
3768
3769     if (u->peer.free && u->peer.sockaddr) {
3770         u->peer.free(&u->peer, u->peer.data, 0);
3771         u->peer.sockaddr = NULL;
3772     }
3773
3774     if (u->peer.connection) {
3775
3776 #if (NGX_HTTP_SSL)
3777
3778         /* TODO: do not shutdown persistent connection */
3779
3780         if (u->peer.connection->ssl) {
3781
3782             /*
3783              * We send the "close notify" shutdown alert to the upstream only
3784              * and do not wait its "close notify" shutdown alert.
3785              * It is acceptable according to the TLS standard.
3786              */
3787
3788             u->peer.connection->ssl->no_wait_shutdown = 1;
3789
3790             (void) ngx_ssl_shutdown(u->peer.connection);
3791         }
3792 #endif
3793
3794         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
3795                 "close http upstream connection: %d",
3796                 u->peer.connection->fd);
3797

```

```

3798     if (u->peer.connection->pool) {
3799         ngx\_destroy\_pool(u->peer.connection->pool);
3800     }
3801
3802     ngx\_close\_connection(u->peer.connection);
3803 }
3804
3805 u->peer.connection = NULL;
3806
3807 if (u->pipe && u->pipe->temp_file) {
3808     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
3809         "http upstream temp fd: %d",
3810         u->pipe->temp_file->file.fd);
3811 }
3812
3813 if (u->store && u->pipe && u->pipe->temp_file
3814     && u->pipe->temp_file->file.fd != NGX\_INVALID\_FILE)
3815 {
3816     if (ngx\_delete\_file(u->pipe->temp_file->file.name.data)
3817         == NGX\_FILE\_ERROR)
3818     {
3819         ngx\_log\_error(NGX\_LOG\_CRIT, r->connection->log, ngx\_errno,
3820             ngx\_delete\_file\_n " \"%s\" failed",
3821             u->pipe->temp_file->file.name.data);
3822     }
3823 }
3824
3825 #if (NGX\_HTTP\_CACHE)
3826
3827 if (r->cache) {
3828
3829     if (u->cacheable) {
3830
3831         if (rc == NGX\_HTTP\_BAD\_GATEWAY || rc == NGX\_HTTP\_GATEWAY\_TIME\_OUT) {
3832             time_t valid;
3833
3834             valid = ngx\_http\_file\_cache\_valid(u->conf->cache_valid, rc);
3835
3836             if (valid) {
3837                 r->cache->valid_sec = ngx\_time() + valid;
3838                 r->cache->error = rc;
3839             }
3840         }
3841     }
3842
3843     ngx\_http\_file\_cache\_free(r->cache, u->pipe->temp_file);
3844 }
3845
3846 #endif
3847
3848 if (r->subrequest_in_memory
3849     && u->headers_in.status_n >= NGX\_HTTP\_SPECIAL\_RESPONSE)
3850 {
3851     u->buffer.last = u->buffer.pos;
3852 }
3853
3854 if (rc == NGX\_DECLINED) {
3855     return;
3856 }
3857
3858 r->connection->log->action = "sending to client";
3859
3860 if (!u->header_sent
3861     || rc == NGX\_HTTP\_REQUEST\_TIME\_OUT
3862     || rc == NGX\_HTTP\_CLIENT\_CLOSED\_REQUEST)
3863 {
3864     ngx\_http\_finalize\_request(r, rc);
3865     return;
3866 }
3867
3868 flush = 0;
3869
3870 if (rc >= NGX\_HTTP\_SPECIAL\_RESPONSE) {
3871     rc = NGX\_ERROR;
3872     flush = 1;
3873 }

```



```

3874     if (r->header_only) {
3875         ngx_http_finalize_request(r, rc);
3876         return;
3877     }
3878
3879     if (rc == 0) {
3880         rc = ngx_http_send_special(r, NGX_HTTP_LAST);
3881     }
3882     else if (flush) {
3883         r->keepalive = 0;
3884         rc = ngx_http_send_special(r, NGX_HTTP_FLUSH);
3885     }
3886
3887     ngx_http_finalize_request(r, rc);
3888 }
3889
3890
3891
3892 static ngx_int_t
3893 ngx_http_upstream_process_header_line(ngx_http_request_t *r, ngx_table_elt_t *h,
3894     ngx_uint_t offset)
3895 {
3896     ngx_table_elt_t **ph;
3897
3898     ph = (ngx_table_elt_t **) ((char *) &r->upstream->headers_in + offset);
3899
3900     if (*ph == NULL) {
3901         *ph = h;
3902     }
3903
3904     return NGX_OK;
3905 }
3906
3907
3908 static ngx_int_t
3909 ngx_http_upstream_ignore_header_line(ngx_http_request_t *r, ngx_table_elt_t *h,
3910     ngx_uint_t offset)
3911 {
3912     return NGX_OK;
3913 }
3914
3915
3916 static ngx_int_t
3917 ngx_http_upstream_process_content_length(ngx_http_request_t *r,
3918     ngx_table_elt_t *h, ngx_uint_t offset)
3919 {
3920     ngx_http_upstream_t *u;
3921
3922     u = r->upstream;
3923
3924     u->headers_in.content_length = h;
3925     u->headers_in.content_length_n = ngx_atoof(h->value.data, h->value.len);
3926
3927     return NGX_OK;
3928 }
3929
3930
3931 static ngx_int_t
3932 ngx_http_upstream_process_last_modified(ngx_http_request_t *r,
3933     ngx_table_elt_t *h, ngx_uint_t offset)
3934 {
3935     ngx_http_upstream_t *u;
3936
3937     u = r->upstream;
3938
3939     u->headers_in.last_modified = h;
3940
3941     #if (NGX_HTTP_CACHE)
3942
3943     if (u->cacheable) {
3944         u->headers_in.last_modified_time = ngx_http_parse_time(h->value.data,
3945             h->value.len);
3946     }
3947
3948     #endif
3949

```

```

3950     return NGX_OK;
3951 }
3952
3953
3954 static ngx_int_t
3955 ngx_http_upstream_process_set_cookie(ngx_http_request_t *r, ngx_table_elt_t *h,
3956     ngx_uint_t offset)
3957 {
3958     ngx_array_t      *pa;
3959     ngx_table_elt_t  **ph;
3960     ngx_http_upstream_t *u;
3961
3962     u = r->upstream;
3963     pa = &u->headers_in.cookies;
3964
3965     if (pa->elts == NULL) {
3966         if (ngx_array_init(pa, r->pool, 1, sizeof(ngx_table_elt_t *)) != NGX_OK)
3967             {
3968                 return NGX_ERROR;
3969             }
3970     }
3971
3972     ph = ngx_array_push(pa);
3973     if (ph == NULL) {
3974         return NGX_ERROR;
3975     }
3976
3977     *ph = h;
3978
3979     #if (NGX_HTTP_CACHE)
3980     if (!(u->conf->ignore_headers & NGX_HTTP_UPSTREAM_IGN_SET_COOKIE)) {
3981         u->cacheable = 0;
3982     }
3983     #endif
3984
3985     return NGX_OK;
3986 }
3987
3988
3989 static ngx_int_t
3990 ngx_http_upstream_process_cache_control(ngx_http_request_t *r,
3991     ngx_table_elt_t *h, ngx_uint_t offset)
3992 {
3993     ngx_array_t      *pa;
3994     ngx_table_elt_t  **ph;
3995     ngx_http_upstream_t *u;
3996
3997     u = r->upstream;
3998     pa = &u->headers_in.cache_control;
3999
4000     if (pa->elts == NULL) {
4001         if (ngx_array_init(pa, r->pool, 2, sizeof(ngx_table_elt_t *)) != NGX_OK)
4002             {
4003                 return NGX_ERROR;
4004             }
4005     }
4006
4007     ph = ngx_array_push(pa);
4008     if (ph == NULL) {
4009         return NGX_ERROR;
4010     }
4011
4012     *ph = h;
4013
4014     #if (NGX_HTTP_CACHE)
4015     {
4016         u_char      *p, *start, *last;
4017         ngx_int_t  n;
4018
4019         if (u->conf->ignore_headers & NGX_HTTP_UPSTREAM_IGN_CACHE_CONTROL) {
4020             return NGX_OK;
4021         }
4022
4023         if (r->cache == NULL) {
4024             return NGX_OK;
4025         }

```

```

4026
4027     if (r->cache->valid_sec != 0 && u->headers_in.x_accel_expires != NULL) {
4028         return NGX\_OK;
4029     }
4030
4031     start = h->value.data;
4032     last = start + h->value.len;
4033
4034     if (ngx\_strlcasestrn(start, last, (u_char *) "no-cache", 8 - 1) != NULL
4035         || ngx\_strlcasestrn(start, last, (u_char *) "no-store", 8 - 1) != NULL
4036         || ngx\_strlcasestrn(start, last, (u_char *) "private", 7 - 1) != NULL)
4037     {
4038         u->cacheable = 0;
4039         return NGX\_OK;
4040     }
4041
4042     p = ngx\_strlcasestrn(start, last, (u_char *) "s-maxage=", 9 - 1);
4043     offset = 9;
4044
4045     if (p == NULL) {
4046         p = ngx\_strlcasestrn(start, last, (u_char *) "max-age=", 8 - 1);
4047         offset = 8;
4048     }
4049
4050     if (p == NULL) {
4051         return NGX\_OK;
4052     }
4053
4054     n = 0;
4055
4056     for (p += offset; p < last; p++) {
4057         if (*p == ',' || *p == ';' || *p == ' ') {
4058             break;
4059         }
4060
4061         if (*p >= '0' && *p <= '9') {
4062             n = n * 10 + *p - '0';
4063             continue;
4064         }
4065
4066         u->cacheable = 0;
4067         return NGX\_OK;
4068     }
4069
4070     if (n == 0) {
4071         u->cacheable = 0;
4072         return NGX\_OK;
4073     }
4074
4075     r->cache->valid_sec = ngx\_time() + n;
4076 }
4077 #endif
4078
4079     return NGX\_OK;
4080 }
4081
4082
4083 static ngx\_int\_t
4084 ngx\_http\_upstream\_process\_expires(ngx\_http\_request\_t *r, ngx\_table\_elt\_t *h,
4085     ngx\_uint\_t offset)
4086 {
4087     ngx\_http\_upstream\_t *u;
4088
4089     u = r->upstream;
4090     u->headers_in.expires = h;
4091
4092     #if (NGX_HTTP_CACHE)
4093     {
4094         time_t expires;
4095
4096         if (u->conf->ignore_headers & NGX\_HTTP\_UPSTREAM\_IGN\_EXPIRES) {
4097             return NGX\_OK;
4098         }
4099
4100         if (r->cache == NULL) {
4101             return NGX\_OK;

```

```

4102     }
4103
4104     if (r->cache->valid_sec != 0) {
4105         return NGX\_OK;
4106     }
4107
4108     expires = ngx\_http\_parse\_time(h->value.data, h->value.len);
4109
4110     if (expires == NGX\_ERROR || expires < ngx\_time()) {
4111         u->cacheable = 0;
4112         return NGX\_OK;
4113     }
4114
4115     r->cache->valid_sec = expires;
4116 }
4117 #endif
4118
4119     return NGX\_OK;
4120 }
4121
4122
4123 static ngx\_int\_t
4124 ngx\_http\_upstream\_process\_accel\_expires(ngx\_http\_request\_t *r,
4125     ngx\_table\_elt\_t *h, ngx\_uint\_t offset)
4126 {
4127     ngx\_http\_upstream\_t *u;
4128
4129     u = r->upstream;
4130     u->headers_in.x_accel_expires = h;
4131
4132     #if (NGX\_HTTP\_CACHE)
4133     {
4134         u_char    *p;
4135         size_t    len;
4136         ngx\_int\_t  n;
4137
4138         if (u->conf->ignore_headers & NGX\_HTTP\_UPSTREAM\_IGN\_XA\_EXPIRES) {
4139             return NGX\_OK;
4140         }
4141
4142         if (r->cache == NULL) {
4143             return NGX\_OK;
4144         }
4145
4146         len = h->value.len;
4147         p = h->value.data;
4148
4149         if (p[0] != '@') {
4150             n = ngx\_atoi(p, len);
4151
4152             switch (n) {
4153             case 0:
4154                 u->cacheable = 0;
4155                 /* fall through */
4156
4157             case NGX\_ERROR:
4158                 return NGX\_OK;
4159
4160             default:
4161                 r->cache->valid_sec = ngx\_time() + n;
4162                 return NGX\_OK;
4163             }
4164         }
4165
4166         p++;
4167         len--;
4168
4169         n = ngx\_atoi(p, len);
4170
4171         if (n != NGX\_ERROR) {
4172             r->cache->valid_sec = n;
4173         }
4174     }
4175 #endif
4176
4177     return NGX\_OK;

```

```

4178 }
4179
4180
4181 static ngx_int_t
4182 ngx_http_upstream_process_limit_rate(ngx_http_request_t *r, ngx_table_elt_t *h,
4183 ngx_uint_t offset)
4184 {
4185     ngx_int_t n;
4186     ngx_http_upstream_t *u;
4187
4188     u = r->upstream;
4189     u->headers_in.x_accel_limit_rate = h;
4190
4191     if (u->conf->ignore_headers & NGX_HTTP_UPSTREAM_IGN_XA_LIMIT_RATE) {
4192         return NGX_OK;
4193     }
4194
4195     n = ngx_atoi(h->value.data, h->value.len);
4196
4197     if (n != NGX_ERROR) {
4198         r->limit_rate = (size_t) n;
4199     }
4200
4201     return NGX_OK;
4202 }
4203
4204
4205 static ngx_int_t
4206 ngx_http_upstream_process_buffering(ngx_http_request_t *r, ngx_table_elt_t *h,
4207 ngx_uint_t offset)
4208 {
4209     u_char c0, c1, c2;
4210     ngx_http_upstream_t *u;
4211
4212     u = r->upstream;
4213
4214     if (u->conf->ignore_headers & NGX_HTTP_UPSTREAM_IGN_XA_BUFFERING) {
4215         return NGX_OK;
4216     }
4217
4218     if (u->conf->change_buffering) {
4219
4220         if (h->value.len == 2) {
4221             c0 = ngx_tolower(h->value.data[0]);
4222             c1 = ngx_tolower(h->value.data[1]);
4223
4224             if (c0 == 'n' && c1 == 'o') {
4225                 u->buffering = 0;
4226             }
4227
4228         } else if (h->value.len == 3) {
4229             c0 = ngx_tolower(h->value.data[0]);
4230             c1 = ngx_tolower(h->value.data[1]);
4231             c2 = ngx_tolower(h->value.data[2]);
4232
4233             if (c0 == 'y' && c1 == 'e' && c2 == 's') {
4234                 u->buffering = 1;
4235             }
4236         }
4237     }
4238
4239     return NGX_OK;
4240 }
4241
4242
4243 static ngx_int_t
4244 ngx_http_upstream_process_charset(ngx_http_request_t *r, ngx_table_elt_t *h,
4245 ngx_uint_t offset)
4246 {
4247     if (r->upstream->conf->ignore_headers & NGX_HTTP_UPSTREAM_IGN_XA_CHARSET) {
4248         return NGX_OK;
4249     }
4250
4251     r->headers_out.override_charset = &h->value;
4252
4253     return NGX_OK;

```

```

4254 }
4255
4256
4257 static ngx_int_t
4258 ngx_http_upstream_process_connection(ngx_http_request_t *r, ngx_table_elt_t *h,
4259 ngx_uint_t offset)
4260 {
4261     r->upstream->headers_in.connection = h;
4262
4263     if (ngx_strlcasestrn(h->value.data, h->value.data + h->value.len,
4264         (u_char *) "close", 5 - 1)
4265         != NULL)
4266     {
4267         r->upstream->headers_in.connection_close = 1;
4268     }
4269
4270     return NGX_OK;
4271 }
4272
4273
4274 static ngx_int_t
4275 ngx_http_upstream_process_transfer_encoding(ngx_http_request_t *r,
4276 ngx_table_elt_t *h, ngx_uint_t offset)
4277 {
4278     r->upstream->headers_in.transfer_encoding = h;
4279
4280     if (ngx_strlcasestrn(h->value.data, h->value.data + h->value.len,
4281         (u_char *) "chunked", 7 - 1)
4282         != NULL)
4283     {
4284         r->upstream->headers_in.chunked = 1;
4285     }
4286
4287     return NGX_OK;
4288 }
4289
4290
4291 static ngx_int_t
4292 ngx_http_upstream_process_vary(ngx_http_request_t *r,
4293 ngx_table_elt_t *h, ngx_uint_t offset)
4294 {
4295     ngx_http_upstream_t *u;
4296
4297     u = r->upstream;
4298     u->headers_in.vary = h;
4299
4300     #if (NGX_HTTP_CACHE)
4301
4302     if (u->conf->ignore_headers & NGX_HTTP_UPSTREAM_IGN_VARY) {
4303         return NGX_OK;
4304     }
4305
4306     if (r->cache == NULL) {
4307         return NGX_OK;
4308     }
4309
4310     if (h->value.len > NGX_HTTP_CACHE_VARY_LEN
4311         || (h->value.len == 1 && h->value.data[0] == '*'))
4312     {
4313         u->cacheable = 0;
4314     }
4315
4316     r->cache->vary = h->value;
4317
4318     #endif
4319
4320     return NGX_OK;
4321 }
4322
4323
4324 static ngx_int_t
4325 ngx_http_upstream_copy_header_line(ngx_http_request_t *r, ngx_table_elt_t *h,
4326 ngx_uint_t offset)
4327 {
4328     ngx_table_elt_t *ho, **ph;
4329

```

```

4330     ho = ngx_list_push(&r->headers_out.headers);
4331     if (ho == NULL) {
4332         return NGX_ERROR;
4333     }
4334
4335     *ho = *h;
4336
4337     if (offset) {
4338         ph = (ngx_table_elt_t **) ((char *) &r->headers_out + offset);
4339         *ph = ho;
4340     }
4341
4342     return NGX_OK;
4343 }
4344
4345
4346 static ngx_int_t
4347 ngx_http_upstream_copy_multi_header_lines(ngx_http_request_t *r,
4348     ngx_table_elt_t *h, ngx_uint_t offset)
4349 {
4350     ngx_array_t      *pa;
4351     ngx_table_elt_t  *ho, **ph;
4352
4353     pa = (ngx_array_t *) ((char *) &r->headers_out + offset);
4354
4355     if (pa->elts == NULL) {
4356         if (ngx_array_init(pa, r->pool, 2, sizeof(ngx_table_elt_t *)) != NGX_OK)
4357             {
4358                 return NGX_ERROR;
4359             }
4360     }
4361
4362     ph = ngx_array_push(pa);
4363     if (ph == NULL) {
4364         return NGX_ERROR;
4365     }
4366
4367     ho = ngx_list_push(&r->headers_out.headers);
4368     if (ho == NULL) {
4369         return NGX_ERROR;
4370     }
4371
4372     *ho = *h;
4373     *ph = ho;
4374
4375     return NGX_OK;
4376 }
4377
4378
4379 static ngx_int_t
4380 ngx_http_upstream_copy_content_type(ngx_http_request_t *r, ngx_table_elt_t *h,
4381     ngx_uint_t offset)
4382 {
4383     u_char  *p, *last;
4384
4385     r->headers_out.content_type_len = h->value.len;
4386     r->headers_out.content_type = h->value;
4387     r->headers_out.content_type_lowercase = NULL;
4388
4389     for (p = h->value.data; *p; p++) {
4390
4391         if (*p != ';') {
4392             continue;
4393         }
4394
4395         last = p;
4396
4397         while (++p == ' ') { /* void */ }
4398
4399         if (*p == '\0') {
4400             return NGX_OK;
4401         }
4402
4403         if (ngx_strncasemp(p, (u_char *) "charset=", 8) != 0) {
4404             continue;
4405         }

```

```

4406     p += 8;
4407
4408     r->headers_out.content_type_len = last - h->value.data;
4409
4410     if (*p == '\0') {
4411         p++;
4412     }
4413
4414     last = h->value.data + h->value.len;
4415
4416     if (*(last - 1) == '\0') {
4417         last--;
4418     }
4419
4420     r->headers_out.charset.len = last - p;
4421     r->headers_out.charset.data = p;
4422
4423     return NGX_OK;
4424 }
4425
4426 return NGX_OK;
4427 }
4428
4429
4430
4431 static ngx_int_t
4432 ngx_http_upstream_copy_last_modified(ngx_http_request_t *r, ngx_table_elt_t *h,
4433     ngx_uint_t offset)
4434 {
4435     ngx_table_elt_t *ho;
4436
4437     ho = ngx_list_push(&r->headers_out.headers);
4438     if (ho == NULL) {
4439         return NGX_ERROR;
4440     }
4441
4442     *ho = *h;
4443
4444     r->headers_out.last_modified = ho;
4445
4446 #if (NGX_HTTP_CACHE)
4447
4448     if (r->upstream->cacheable) {
4449         r->headers_out.last_modified_time =
4450             r->upstream->headers_in.last_modified_time;
4451     }
4452
4453 #endif
4454
4455     return NGX_OK;
4456 }
4457
4458
4459 static ngx_int_t
4460 ngx_http_upstream_rewrite_location(ngx_http_request_t *r, ngx_table_elt_t *h,
4461     ngx_uint_t offset)
4462 {
4463     ngx_int_t rc;
4464     ngx_table_elt_t *ho;
4465
4466     ho = ngx_list_push(&r->headers_out.headers);
4467     if (ho == NULL) {
4468         return NGX_ERROR;
4469     }
4470
4471     *ho = *h;
4472
4473     if (r->upstream->rewrite_redirect) {
4474         rc = r->upstream->rewrite_redirect(r, ho, 0);
4475
4476         if (rc == NGX_DECLINED) {
4477             return NGX_OK;
4478         }
4479
4480         if (rc == NGX_OK) {
4481             r->headers_out.location = ho;

```



```

4482         ngx\_log\_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
4483             "rewritten location: \"%V\"", &ho->value);
4484     }
4485     return rc;
4486 }
4487
4488 if (ho->value.data[0] != '/') {
4489     r->headers_out.location = ho;
4490 }
4491
4492 /*
4493  * we do not set r->headers_out.location here to avoid the handling
4494  * the local redirects without a host name by ngx\_http\_header\_filter\(\)
4495  */
4496
4497 return NGX_OK;
4498 }
4499
4500 static ngx\_int\_t
4501 ngx\_http\_upstream\_rewrite\_refresh(ngx\_http\_request\_t *r, ngx\_table\_elt\_t *h,
4502     ngx\_uint\_t offset)
4503 {
4504     u_char          *p;
4505     ngx\_int\_t      rc;
4506     ngx\_table\_elt\_t *ho;
4507
4508     ho = ngx\_list\_push(&r->headers_out.headers);
4509     if (ho == NULL) {
4510         return NGX_ERROR;
4511     }
4512
4513     *ho = *h;
4514
4515     if (r->upstream->rewrite_redirect) {
4516
4517         p = ngx\_strcasestrn(ho->value.data, "url=", 4 - 1);
4518
4519         if (p) {
4520             rc = r->upstream->rewrite_redirect(r, ho, p + 4 - ho->value.data);
4521         } else {
4522             return NGX_OK;
4523         }
4524
4525         if (rc == NGX_DECLINED) {
4526             return NGX_OK;
4527         }
4528
4529         if (rc == NGX_OK) {
4530             r->headers_out.refresh = ho;
4531
4532             ngx\_log\_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
4533                 "rewritten refresh: \"%V\"", &ho->value);
4534         }
4535
4536         return rc;
4537     }
4538
4539     r->headers_out.refresh = ho;
4540
4541     return NGX_OK;
4542 }
4543
4544 static ngx\_int\_t
4545 ngx\_http\_upstream\_rewrite\_set\_cookie(ngx\_http\_request\_t *r, ngx\_table\_elt\_t *h,
4546     ngx\_uint\_t offset)
4547 {
4548     ngx\_int\_t      rc;
4549     ngx\_table\_elt\_t *ho;
4550
4551     ho = ngx\_list\_push(&r->headers_out.headers);
4552     if (ho == NULL) {

```

```

4558     return NGX\_ERROR;
4559 }
4560
4561 *ho = *h;
4562
4563 if (r->upstream->rewrite_cookie) {
4564     rc = r->upstream->rewrite_cookie(r, ho);
4565
4566     if (rc == NGX\_DECLINED) {
4567         return NGX\_OK;
4568     }
4569
4570 #if (NGX\_DEBUG)
4571     if (rc == NGX\_OK) {
4572         ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
4573             "rewritten cookie: \"%V\"", &ho->value);
4574     }
4575 #endif
4576
4577     return rc;
4578 }
4579
4580 return NGX\_OK;
4581 }
4582
4583
4584 static ngx\_int\_t
4585 ngx\_http\_upstream\_copy\_allow\_ranges(ngx\_http\_request\_t *r,
4586     ngx\_table\_elt\_t *h, ngx\_uint\_t offset)
4587 {
4588     ngx\_table\_elt\_t *ho;
4589
4590     if (r->upstream->conf->force_ranges) {
4591         return NGX\_OK;
4592     }
4593
4594 #if (NGX\_HTTP\_CACHE)
4595
4596     if (r->cached) {
4597         r->allow_ranges = 1;
4598         return NGX\_OK;
4599     }
4600
4601     if (r->upstream->cacheable) {
4602         r->allow_ranges = 1;
4603         r->single_range = 1;
4604         return NGX\_OK;
4605     }
4606
4607 #endif
4608
4609     ho = ngx\_list\_push(&r->headers_out.headers);
4610     if (ho == NULL) {
4611         return NGX\_ERROR;
4612     }
4613
4614     *ho = *h;
4615
4616     r->headers_out.accept_ranges = ho;
4617
4618     return NGX\_OK;
4619 }
4620
4621
4622 #if (NGX\_HTTP\_GZIP)
4623
4624 static ngx\_int\_t
4625 ngx\_http\_upstream\_copy\_content\_encoding(ngx\_http\_request\_t *r,
4626     ngx\_table\_elt\_t *h, ngx\_uint\_t offset)
4627 {
4628     ngx\_table\_elt\_t *ho;
4629
4630     ho = ngx\_list\_push(&r->headers_out.headers);
4631     if (ho == NULL) {
4632         return NGX\_ERROR;
4633     }

```

```

4634     *ho = *h;
4635
4636
4637     r->headers_out.content_encoding = ho;
4638
4639     return NGX_OK;
4640 }
4641
4642 #endif
4643
4644
4645 static ngx_int_t
4646 ngx_http_upstream_add_variables(ngx_conf_t *cf)
4647 {
4648     ngx_http_variable_t *var, *v;
4649
4650     for (v = ngx_http_upstream_vars; v->name.len; v++) {
4651         var = ngx_http_add_variable(cf, &v->name, v->flags);
4652         if (var == NULL) {
4653             return NGX_ERROR;
4654         }
4655
4656         var->get_handler = v->get_handler;
4657         var->data = v->data;
4658     }
4659
4660     return NGX_OK;
4661 }
4662
4663
4664 static ngx_int_t
4665 ngx_http_upstream_addr_variable(ngx_http_request_t *r,
4666     ngx_http_variable_value_t *v, uintptr_t data)
4667 {
4668     u_char                *p;
4669     size_t                len;
4670     ngx_uint_t          i;
4671     ngx_http_upstream_state_t *state;
4672
4673     v->valid = 1;
4674     v->no_cacheable = 0;
4675     v->not_found = 0;
4676
4677     if (r->upstream_states == NULL || r->upstream_states->nelts == 0) {
4678         v->not_found = 1;
4679         return NGX_OK;
4680     }
4681
4682     len = 0;
4683     state = r->upstream_states->elts;
4684
4685     for (i = 0; i < r->upstream_states->nelts; i++) {
4686         if (state[i].peer) {
4687             len += state[i].peer->len + 2;
4688         } else {
4689             len += 3;
4690         }
4691     }
4692
4693     p = ngx_pnalloc(r->pool, len);
4694     if (p == NULL) {
4695         return NGX_ERROR;
4696     }
4697
4698     v->data = p;
4699
4700
4701     i = 0;
4702
4703     for ( ;; ) {
4704         if (state[i].peer) {
4705             p = ngx_cpymem(p, state[i].peer->data, state[i].peer->len);
4706         }
4707
4708         if (++i == r->upstream_states->nelts) {
4709             break;

```

```

4710     }
4711
4712     if (state[i].peer) {
4713         *p++ = ',';
4714         *p++ = ' ';
4715
4716     } else {
4717         *p++ = ' ';
4718         *p++ = ':';
4719         *p++ = ' ';
4720
4721         if (++i == r->upstream_states->nelts) {
4722             break;
4723         }
4724
4725         continue;
4726     }
4727 }
4728
4729 v->len = p - v->data;
4730
4731 return NGX_OK;
4732 }
4733
4734
4735 static ngx_int_t
4736 ngx_http_upstream_status_variable(ngx_http_request_t *r,
4737     ngx_http_variable_value_t *v, uintptr_t data)
4738 {
4739     u_char          *p;
4740     size_t          len;
4741     ngx_uint_t      i;
4742     ngx_http_upstream_state_t *state;
4743
4744     v->valid = 1;
4745     v->no_cacheable = 0;
4746     v->not_found = 0;
4747
4748     if (r->upstream_states == NULL || r->upstream_states->nelts == 0) {
4749         v->not_found = 1;
4750         return NGX_OK;
4751     }
4752
4753     len = r->upstream_states->nelts * (3 + 2);
4754
4755     p = ngx_pnalloc(r->pool, len);
4756     if (p == NULL) {
4757         return NGX_ERROR;
4758     }
4759
4760     v->data = p;
4761
4762     i = 0;
4763     state = r->upstream_states->elts;
4764
4765     for ( ;; ) {
4766         if (state[i].status) {
4767             p = ngx_sprintf(p, "%ui", state[i].status);
4768
4769         } else {
4770             *p++ = '-';
4771         }
4772
4773         if (++i == r->upstream_states->nelts) {
4774             break;
4775         }
4776
4777         if (state[i].peer) {
4778             *p++ = ',';
4779             *p++ = ' ';
4780
4781         } else {
4782             *p++ = ' ';
4783             *p++ = ':';
4784             *p++ = ' ';
4785

```

```

4786         if (++i == r->upstream_states->nelts) {
4787             break;
4788         }
4789
4790         continue;
4791     }
4792 }
4793
4794 v->len = p - v->data;
4795
4796 return NGX_OK;
4797 }
4798
4799
4800 static ngx_int_t
4801 ngx_http_upstream_response_time_variable(ngx_http_request_t *r,
4802 ngx_http_variable_value_t *v, uintptr_t data)
4803 {
4804     u_char *p;
4805     size_t len;
4806     ngx_uint_t i;
4807     ngx_msec_int_t ms;
4808     ngx_http_upstream_state_t *state;
4809
4810     v->valid = 1;
4811     v->no_cacheable = 0;
4812     v->not_found = 0;
4813
4814     if (r->upstream_states == NULL || r->upstream_states->nelts == 0) {
4815         v->not_found = 1;
4816         return NGX_OK;
4817     }
4818
4819     len = r->upstream_states->nelts * (NGX_TIME_T_LEN + 4 + 2);
4820
4821     p = ngx_pnalloc(r->pool, len);
4822     if (p == NULL) {
4823         return NGX_ERROR;
4824     }
4825
4826     v->data = p;
4827
4828     i = 0;
4829     state = r->upstream_states->elts;
4830
4831     for ( ;; ) {
4832         if (state[i].status) {
4833
4834             if (data
4835                 && state[i].header_sec != (time_t) NGX_ERROR)
4836             {
4837                 ms = (ngx_msec_int_t)
4838                     (state[i].header_sec * 1000 + state[i].header_msec);
4839
4840             } else {
4841                 ms = (ngx_msec_int_t)
4842                     (state[i].response_sec * 1000 + state[i].response_msec);
4843             }
4844
4845             ms = ngx_max(ms, 0);
4846             p = ngx_sprintf(p, "%T.%03M", (time_t) ms / 1000, ms % 1000);
4847
4848         } else {
4849             *p++ = '-';
4850         }
4851
4852         if (++i == r->upstream_states->nelts) {
4853             break;
4854         }
4855
4856         if (state[i].peer) {
4857             *p++ = ',';
4858             *p++ = ' ';
4859
4860         } else {
4861             *p++ = ' ';

```

```

4862         *p++ = ':';
4863         *p++ = ' ';
4864
4865         if (++i == r->upstream_states->nelts) {
4866             break;
4867         }
4868
4869         continue;
4870     }
4871 }
4872
4873 v->len = p - v->data;
4874
4875 return NGX_OK;
4876 }
4877
4878
4879 static ngx_int_t
4880 ngx_http_upstream_response_length_variable(ngx_http_request_t *r,
4881 ngx_http_variable_value_t *v, uintptr_t data)
4882 {
4883     u_char                *p;
4884     size_t                len;
4885     ngx_uint_t            i;
4886     ngx_http_upstream_state_t *state;
4887
4888     v->valid = 1;
4889     v->no_cacheable = 0;
4890     v->not_found = 0;
4891
4892     if (r->upstream_states == NULL || r->upstream_states->nelts == 0) {
4893         v->not_found = 1;
4894         return NGX_OK;
4895     }
4896
4897     len = r->upstream_states->nelts * (NGX_OFF_T_LEN + 2);
4898
4899     p = ngx_pnalloc(r->pool, len);
4900     if (p == NULL) {
4901         return NGX_ERROR;
4902     }
4903
4904     v->data = p;
4905
4906     i = 0;
4907     state = r->upstream_states->elts;
4908
4909     for ( ;; ) {
4910         p = ngx_sprintf(p, "%0", state[i].response_length);
4911
4912         if (++i == r->upstream_states->nelts) {
4913             break;
4914         }
4915
4916         if (state[i].peer) {
4917             *p++ = ',';
4918             *p++ = ' ';
4919         } else {
4920             *p++ = ' ';
4921             *p++ = ':';
4922             *p++ = ' ';
4923         }
4924
4925         if (++i == r->upstream_states->nelts) {
4926             break;
4927         }
4928
4929         continue;
4930     }
4931 }
4932
4933 v->len = p - v->data;
4934
4935 return NGX_OK;
4936 }
4937

```

```

4938
4939 ngx_int_t
4940 ngx_http_upstream_header_variable(ngx_http_request_t *r,
4941 ngx_http_variable_value_t *v, uintptr_t data)
4942 {
4943     if (r->upstream == NULL) {
4944         v->not_found = 1;
4945         return NGX_OK;
4946     }
4947
4948     return ngx_http_variable_unknown_header(v, (ngx_str_t *) data,
4949                                             &r->upstream->headers_in.headers.part,
4950                                             sizeof("upstream_http_") - 1);
4951 }
4952
4953
4954 ngx_int_t
4955 ngx_http_upstream_cookie_variable(ngx_http_request_t *r,
4956 ngx_http_variable_value_t *v, uintptr_t data)
4957 {
4958     ngx_str_t *name = (ngx_str_t *) data;
4959
4960     ngx_str_t cookie, s;
4961
4962     if (r->upstream == NULL) {
4963         v->not_found = 1;
4964         return NGX_OK;
4965     }
4966
4967     s.len = name->len - (sizeof("upstream_cookie_") - 1);
4968     s.data = name->data + sizeof("upstream_cookie_") - 1;
4969
4970     if (ngx_http_parse_set_cookie_lines(&r->upstream->headers_in.cookies,
4971                                       &s, &cookie)
4972         == NGX_DECLINED)
4973     {
4974         v->not_found = 1;
4975         return NGX_OK;
4976     }
4977
4978     v->len = cookie.len;
4979     v->valid = 1;
4980     v->no_cacheable = 0;
4981     v->not_found = 0;
4982     v->data = cookie.data;
4983
4984     return NGX_OK;
4985 }
4986
4987
4988 #if (NGX_HTTP_CACHE)
4989
4990 ngx_int_t
4991 ngx_http_upstream_cache_status(ngx_http_request_t *r,
4992 ngx_http_variable_value_t *v, uintptr_t data)
4993 {
4994     ngx_uint_t n;
4995
4996     if (r->upstream == NULL || r->upstream->cache_status == 0) {
4997         v->not_found = 1;
4998         return NGX_OK;
4999     }
5000
5001     n = r->upstream->cache_status - 1;
5002
5003     v->valid = 1;
5004     v->no_cacheable = 0;
5005     v->not_found = 0;
5006     v->len = ngx_http_cache_status[n].len;
5007     v->data = ngx_http_cache_status[n].data;
5008
5009     return NGX_OK;
5010 }
5011
5012
5013 static ngx_int_t

```

```

5014 ngx_http_upstream_cache_last_modified(ngx_http_request_t *r,
5015 ngx_http_variable_value_t *v, uintptr_t data)
5016 {
5017     u_char *p;
5018
5019     if (r->upstream == NULL
5020         || !r->upstream->conf->cache_revalidate
5021         || r->upstream->cache_status != NGX_HTTP_CACHE_EXPIRED
5022         || r->cache->last_modified == -1)
5023     {
5024         v->not_found = 1;
5025         return NGX_OK;
5026     }
5027
5028     p = ngx_pnalloc(r->pool, sizeof("Mon, 28 Sep 1970 06:00:00 GMT") - 1);
5029     if (p == NULL) {
5030         return NGX_ERROR;
5031     }
5032
5033     v->len = ngx_http_time(p, r->cache->last_modified) - p;
5034     v->valid = 1;
5035     v->no_cacheable = 0;
5036     v->not_found = 0;
5037     v->data = p;
5038
5039     return NGX_OK;
5040 }
5041
5042
5043 static ngx_int_t
5044 ngx_http_upstream_cache_etag(ngx_http_request_t *r,
5045 ngx_http_variable_value_t *v, uintptr_t data)
5046 {
5047     if (r->upstream == NULL
5048         || !r->upstream->conf->cache_revalidate
5049         || r->upstream->cache_status != NGX_HTTP_CACHE_EXPIRED
5050         || r->cache->etag.len == 0)
5051     {
5052         v->not_found = 1;
5053         return NGX_OK;
5054     }
5055
5056     v->valid = 1;
5057     v->no_cacheable = 0;
5058     v->not_found = 0;
5059     v->len = r->cache->etag.len;
5060     v->data = r->cache->etag.data;
5061
5062     return NGX_OK;
5063 }
5064
5065 #endif
5066
5067
5068 static char *
5069 ngx_http_upstream(ngx_conf_t *cf, ngx_command_t *cmd, void *dummy)
5070 {
5071     char *rv;
5072     void *mconf;
5073     ngx_str_t *value;
5074     ngx_url_t u;
5075     ngx_uint_t m;
5076     ngx_conf_t pcf;
5077     ngx_http_module_t *module;
5078     ngx_http_conf_ctx_t *ctx, *http_ctx;
5079     ngx_http_upstream_srv_conf_t *uscf;
5080
5081     ngx_memzero(&u, sizeof(ngx_url_t));
5082
5083     value = cf->args->elts;
5084     u.host = value[1];
5085     u.no_resolve = 1;
5086     u.no_port = 1;
5087
5088     uscf = ngx_http_upstream_add(cf, &u, NGX_HTTP_UPSTREAM_CREATE
5089 |NGX_HTTP_UPSTREAM_WEIGHT

```



```

5090 |NGX_HTTP_UPSTREAM_MAX_FAILS
5091 |NGX_HTTP_UPSTREAM_FAIL_TIMEOUT
5092 |NGX_HTTP_UPSTREAM_DOWN
5093 |NGX_HTTP_UPSTREAM_BACKUP);
5094 if (uscf == NULL) {
5095     return NGX_CONF_ERROR;
5096 }
5097
5098
5099 ctx = ngx_palloc(cf->pool, sizeof(ngx_http_conf_ctx_t));
5100 if (ctx == NULL) {
5101     return NGX_CONF_ERROR;
5102 }
5103
5104 http_ctx = cf->ctx;
5105 ctx->main_conf = http_ctx->main_conf;
5106
5107 /* the upstream{}'s srv_conf */
5108
5109 ctx->srv_conf = ngx_palloc(cf->pool, sizeof(void *) * ngx_http_max_module);
5110 if (ctx->srv_conf == NULL) {
5111     return NGX_CONF_ERROR;
5112 }
5113
5114 ctx->srv_conf[ngx_http_upstream_module.ctx_index] = uscf;
5115
5116 uscf->srv_conf = ctx->srv_conf;
5117
5118 /* the upstream{}'s loc_conf */
5119
5120 ctx->loc_conf = ngx_palloc(cf->pool, sizeof(void *) * ngx_http_max_module);
5121 if (ctx->loc_conf == NULL) {
5122     return NGX_CONF_ERROR;
5123 }
5124
5125
5126 for (m = 0; ngx_modules[m]; m++) {
5127     if (ngx_modules[m]->type != NGX_HTTP_MODULE) {
5128         continue;
5129     }
5130
5131     module = ngx_modules[m]->ctx;
5132
5133     if (module->create_srv_conf) {
5134         mconf = module->create_srv_conf(cf);
5135         if (mconf == NULL) {
5136             return NGX_CONF_ERROR;
5137         }
5138
5139         ctx->srv_conf[ngx_modules[m]->ctx_index] = mconf;
5140     }
5141
5142     if (module->create_loc_conf) {
5143         mconf = module->create_loc_conf(cf);
5144         if (mconf == NULL) {
5145             return NGX_CONF_ERROR;
5146         }
5147
5148         ctx->loc_conf[ngx_modules[m]->ctx_index] = mconf;
5149     }
5150 }
5151
5152 uscf->servers = ngx_array_create(cf->pool, 4,
5153                                 sizeof(ngx_http_upstream_server_t));
5154 if (uscf->servers == NULL) {
5155     return NGX_CONF_ERROR;
5156 }
5157
5158
5159 /* parse inside upstream{} */
5160
5161 pcf = *cf;
5162 cf->ctx = ctx;
5163 cf->cmd_type = NGX_HTTP_UPS_CONF;
5164
5165 rv = ngx_conf_parse(cf, NULL);

```

```

5166     *cf = pcf;
5167
5168
5169     if (rv != NGX\_CONF\_OK) {
5170         return rv;
5171     }
5172
5173     if (uscf->servers->nelts == 0) {
5174         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
5175             "no servers are inside upstream");
5176         return NGX\_CONF\_ERROR;
5177     }
5178
5179     return rv;
5180 }
5181
5182
5183 static char *
5184 ngx\_http\_upstream\_server(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
5185 {
5186     ngx\_http\_upstream\_srv\_conf\_t *uscf = conf;
5187
5188     time_t                fail_timeout;
5189     ngx\_str\_t             *value, s;
5190     ngx\_url\_t             u;
5191     ngx\_int\_t             weight, max_fails;
5192     ngx\_uint\_t           i;
5193     ngx\_http\_upstream\_server\_t *us;
5194
5195     us = ngx\_array\_push(uscf->servers);
5196     if (us == NULL) {
5197         return NGX\_CONF\_ERROR;
5198     }
5199
5200     ngx\_memzero(us, sizeof(ngx\_http\_upstream\_server\_t));
5201
5202     value = cf->args->elts;
5203
5204     weight = 1;
5205     max_fails = 1;
5206     fail_timeout = 10;
5207
5208     for (i = 2; i < cf->args->nelts; i++) {
5209
5210         if (ngx\_strncmp(value[i].data, "weight=", 7) == 0) {
5211
5212             if (!(uscf->flags & NGX\_HTTP\_UPSTREAM\_WEIGHT)) {
5213                 goto not_supported;
5214             }
5215
5216             weight = ngx\_atoi(&value[i].data[7], value[i].len - 7);
5217
5218             if (weight == NGX\_ERROR || weight == 0) {
5219                 goto invalid;
5220             }
5221
5222             continue;
5223         }
5224
5225         if (ngx\_strncmp(value[i].data, "max_fails=", 10) == 0) {
5226
5227             if (!(uscf->flags & NGX\_HTTP\_UPSTREAM\_MAX\_FAILS)) {
5228                 goto not_supported;
5229             }
5230
5231             max_fails = ngx\_atoi(&value[i].data[10], value[i].len - 10);
5232
5233             if (max_fails == NGX\_ERROR) {
5234                 goto invalid;
5235             }
5236
5237             continue;
5238         }
5239
5240         if (ngx\_strncmp(value[i].data, "fail_timeout=", 13) == 0) {

```

```

5242     if (!(uscf->flags & NGX\_HTTP\_UPSTREAM\_FAIL\_TIMEOUT)) {
5243         goto not_supported;
5244     }
5245
5246     s.len = value[i].len - 13;
5247     s.data = &value[i].data[13];
5248
5249     fail_timeout = ngx\_parse\_time(&s, 1);
5250
5251     if (fail_timeout == (time_t) NGX\_ERROR) {
5252         goto invalid;
5253     }
5254
5255     continue;
5256 }
5257
5258 if (ngx\_strcmp(value[i].data, "backup") == 0) {
5259
5260     if (!(uscf->flags & NGX\_HTTP\_UPSTREAM\_BACKUP)) {
5261         goto not_supported;
5262     }
5263
5264     us->backup = 1;
5265
5266     continue;
5267 }
5268
5269 if (ngx\_strcmp(value[i].data, "down") == 0) {
5270
5271     if (!(uscf->flags & NGX\_HTTP\_UPSTREAM\_DOWN)) {
5272         goto not_supported;
5273     }
5274
5275     us->down = 1;
5276
5277     continue;
5278 }
5279
5280 goto invalid;
5281 }
5282
5283 ngx\_memzero(&u, sizeof(ngx\_url\_t));
5284
5285 u.url = value[1];
5286 u.default_port = 80;
5287
5288 if (ngx\_parse\_url(cf->pool, &u) != NGX\_OK) {
5289     if (u.err) {
5290         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
5291             "%s in upstream \"%V\"", u.err, &u.url);
5292     }
5293
5294     return NGX\_CONF\_ERROR;
5295 }
5296
5297 us->name = u.url;
5298 us->addrs = u.addrs;
5299 us->naddrs = u.naddrs;
5300 us->weight = weight;
5301 us->max_fails = max_fails;
5302 us->fail_timeout = fail_timeout;
5303
5304 return NGX\_CONF\_OK;
5305
5306 invalid:
5307
5308 ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
5309     "invalid parameter \"%V\"", &value[i]);
5310
5311 return NGX\_CONF\_ERROR;
5312
5313 not_supported:
5314
5315 ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
5316     "balancing method does not support parameter \"%V\"",
5317     &value[i]);

```

```

5318     return NGX\_CONF\_ERROR;
5319 }
5320 }
5321
5322
5323 ngx\_http\_upstream\_srv\_conf\_t *
5324 ngx\_http\_upstream\_add(ngx\_conf\_t *cf, ngx\_url\_t *u, ngx\_uint\_t flags)
5325 {
5326     ngx\_uint\_t i;
5327     ngx\_http\_upstream\_server\_t *us;
5328     ngx\_http\_upstream\_srv\_conf\_t *uscf, **uscfp;
5329     ngx\_http\_upstream\_main\_conf\_t *umcf;
5330
5331     if (!(flags & NGX\_HTTP\_UPSTREAM\_CREATE)) {
5332
5333         if (ngx\_parse\_url(cf->pool, u) != NGX\_OK) {
5334             if (u->err) {
5335                 ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
5336                     "%s in upstream \"%V\"", u->err, &u->url);
5337             }
5338
5339             return NULL;
5340         }
5341     }
5342
5343     umcf = ngx\_http\_conf\_get\_module\_main\_conf(cf, ngx\_http\_upstream\_module);
5344
5345     uscfp = umcf->upstreams.elts;
5346
5347     for (i = 0; i < umcf->upstreams.nelts; i++) {
5348
5349         if (uscfp[i]->host.len != u->host.len
5350             || ngx\_strncasecmp(uscfp[i]->host.data, u->host.data, u->host.len)
5351                 != 0)
5352         {
5353             continue;
5354         }
5355
5356         if ((flags & NGX\_HTTP\_UPSTREAM\_CREATE)
5357             && (uscfp[i]->flags & NGX\_HTTP\_UPSTREAM\_CREATE))
5358         {
5359             ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
5360                 "duplicate upstream \"%V\"", &u->host);
5361             return NULL;
5362         }
5363
5364         if ((uscfp[i]->flags & NGX\_HTTP\_UPSTREAM\_CREATE) && !u->no_port) {
5365             ngx\_conf\_log\_error(NGX\_LOG\_WARN, cf, 0,
5366                 "upstream \"%V\" may not have port %d",
5367                 &u->host, u->port);
5368             return NULL;
5369         }
5370
5371         if ((flags & NGX\_HTTP\_UPSTREAM\_CREATE) && !uscfp[i]->no_port) {
5372             ngx\_log\_error(NGX\_LOG\_WARN, cf->log, 0,
5373                 "upstream \"%V\" may not have port %d in %s:%ui",
5374                 &u->host, uscfp[i]->port,
5375                 uscfp[i]->file_name, uscfp[i]->line);
5376             return NULL;
5377         }
5378
5379         if (uscfp[i]->port && u->port
5380             && uscfp[i]->port != u->port)
5381         {
5382             continue;
5383         }
5384
5385         if (uscfp[i]->default_port && u->default_port
5386             && uscfp[i]->default_port != u->default_port)
5387         {
5388             continue;
5389         }
5390
5391         if (flags & NGX\_HTTP\_UPSTREAM\_CREATE) {
5392             uscfp[i]->flags = flags;
5393         }

```

```

5394     return uscfp[i];
5395 }
5396
5397
5398 uscf = ngx_palloc(cf->pool, sizeof(ngx_http_upstream_srv_conf_t));
5399 if (uscf == NULL) {
5400     return NULL;
5401 }
5402
5403 uscf->flags = flags;
5404 uscf->host = u->host;
5405 uscf->file_name = cf->conf_file->file.name.data;
5406 uscf->line = cf->conf_file->line;
5407 uscf->port = u->port;
5408 uscf->default_port = u->default_port;
5409 uscf->no_port = u->no_port;
5410
5411 if (u->naddrs == 1) {
5412     uscf->servers = ngx_array_create(cf->pool, 1,
5413                                     sizeof(ngx_http_upstream_server_t));
5414     if (uscf->servers == NULL) {
5415         return NULL;
5416     }
5417
5418     us = ngx_array_push(uscf->servers);
5419     if (us == NULL) {
5420         return NULL;
5421     }
5422
5423     ngx_memzero(us, sizeof(ngx_http_upstream_server_t));
5424
5425     us->addrs = u->addrs;
5426     us->naddrs = 1;
5427 }
5428
5429 uscfp = ngx_array_push(&umcf->upstreams);
5430 if (uscfp == NULL) {
5431     return NULL;
5432 }
5433
5434 *uscfp = uscf;
5435
5436 return uscf;
5437 }
5438
5439
5440 char *
5441 ngx_http_upstream_bind_set_slot(ngx_conf_t *cf, ngx_command_t *cmd,
5442 void *conf)
5443 {
5444     char *p = conf;
5445
5446     ngx_int_t rc;
5447     ngx_str_t *value;
5448     ngx_http_complex_value_t cv;
5449     ngx_http_upstream_local_t **plocal, *local;
5450     ngx_http_compile_complex_value_t ccv;
5451
5452     plocal = (ngx_http_upstream_local_t **) (p + cmd->offset);
5453
5454     if (*plocal != NGX_CONF_UNSET_PTR) {
5455         return "is duplicate";
5456     }
5457
5458     value = cf->args->elts;
5459
5460     if (ngx_strcmp(value[1].data, "off") == 0) {
5461         *plocal = NULL;
5462         return NGX_CONF_OK;
5463     }
5464
5465     ngx_memzero(&ccv, sizeof(ngx_http_compile_complex_value_t));
5466
5467     ccv.cf = cf;
5468     ccv.value = &value[1];
5469     ccv.complex_value = &cv;

```

```

5470     if (ngx_http_compile_complex_value(&ccv) != NGX_OK) {
5471         return NGX_CONF_ERROR;
5472     }
5473
5474     local = ngx_palloc(cf->pool, sizeof(ngx_http_upstream_local_t));
5475     if (local == NULL) {
5476         return NGX_CONF_ERROR;
5477     }
5478
5479     *plocal = local;
5480
5481     if (cv.lengths) {
5482         local->value = ngx_palloc(cf->pool, sizeof(ngx_http_complex_value_t));
5483         if (local->value == NULL) {
5484             return NGX_CONF_ERROR;
5485         }
5486     }
5487
5488     *local->value = cv;
5489
5490     return NGX_CONF_OK;
5491 }
5492
5493 local->addr = ngx_palloc(cf->pool, sizeof(ngx_addr_t));
5494 if (local->addr == NULL) {
5495     return NGX_CONF_ERROR;
5496 }
5497
5498 rc = ngx_parse_addr(cf->pool, local->addr, value[1].data, value[1].len);
5499
5500 switch (rc) {
5501 case NGX_OK:
5502     local->addr->name = value[1];
5503     return NGX_CONF_OK;
5504
5505 case NGX_DECLINED:
5506     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
5507                        "invalid address \"%V\"", &value[1]);
5508     /* fall through */
5509
5510 default:
5511     return NGX_CONF_ERROR;
5512 }
5513 }
5514
5515
5516 static ngx_addr_t *
5517 ngx_http_upstream_get_local(ngx_http_request_t *r,
5518 ngx_http_upstream_local_t *local)
5519 {
5520     ngx_int_t rc;
5521     ngx_str_t val;
5522     ngx_addr_t *addr;
5523
5524     if (local == NULL) {
5525         return NULL;
5526     }
5527
5528     if (local->value == NULL) {
5529         return local->addr;
5530     }
5531
5532     if (ngx_http_complex_value(r, local->value, &val) != NGX_OK) {
5533         return NULL;
5534     }
5535
5536     if (val.len == 0) {
5537         return NULL;
5538     }
5539
5540     addr = ngx_palloc(r->pool, sizeof(ngx_addr_t));
5541     if (addr == NULL) {
5542         return NULL;
5543     }
5544
5545     rc = ngx_parse_addr(r->pool, addr, val.data, val.len);

```

```

5546
5547 switch (rc) {
5548 case NGX_OK:
5549     addr->name = val;
5550     return addr;
5551
5552 case NGX_DECLINED:
5553     ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
5554                 "invalid local address \"%V\"", &val);
5555     /* fall through */
5556
5557 default:
5558     return NULL;
5559 }
5560 }
5561
5562
5563 char *
5564 ngx_http_upstream_param_set_slot(ngx_conf_t *cf, ngx_command_t *cmd,
5565 void *conf)
5566 {
5567     char *p = conf;
5568
5569     ngx_str_t          *value;
5570     ngx_array_t        **a;
5571     ngx_http_upstream_param_t *param;
5572
5573     a = (ngx_array_t **) (p + cmd->offset);
5574
5575     if (*a == NULL) {
5576         *a = ngx_array_create(cf->pool, 4, sizeof(ngx_http_upstream_param_t));
5577         if (*a == NULL) {
5578             return NGX_CONF_ERROR;
5579         }
5580     }
5581
5582     param = ngx_array_push(*a);
5583     if (param == NULL) {
5584         return NGX_CONF_ERROR;
5585     }
5586
5587     value = cf->args->elts;
5588
5589     param->key = value[1];
5590     param->value = value[2];
5591     param->skip_empty = 0;
5592
5593     if (cf->args->nelts == 4) {
5594         if (ngx_strcmp(value[3].data, "if_not_empty") != 0) {
5595             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
5596                             "invalid parameter \"%V\"", &value[3]);
5597             return NGX_CONF_ERROR;
5598         }
5599
5600         param->skip_empty = 1;
5601     }
5602
5603     return NGX_CONF_OK;
5604 }
5605
5606
5607 ngx_int_t
5608 ngx_http_upstream_hide_headers_hash(ngx_conf_t *cf,
5609 ngx_http_upstream_conf_t *conf, ngx_http_upstream_conf_t *prev,
5610 ngx_str_t *default_hide_headers, ngx_hash_init_t *hash)
5611 {
5612     ngx_str_t          *h;
5613     ngx_uint_t         i, j;
5614     ngx_array_t        hide_headers;
5615     ngx_hash_key_t     *hk;
5616
5617     if (conf->hide_headers == NGX_CONF_UNSET_PTR
5618         && conf->pass_headers == NGX_CONF_UNSET_PTR)
5619     {
5620         conf->hide_headers = prev->hide_headers;
5621         conf->pass_headers = prev->pass_headers;

```

```

5622     conf->hide_headers_hash = prev->hide_headers_hash;
5623
5624     if (conf->hide_headers_hash.buckets
5625 #if (NGX_HTTP_CACHE)
5626         && ((conf->cache == 0) == (prev->cache == 0))
5627 #endif
5628         )
5629     {
5630         return NGX\_OK;
5631     }
5632
5633 } else {
5634     if (conf->hide_headers == NGX\_CONF\_UNSET\_PTR) {
5635         conf->hide_headers = prev->hide_headers;
5636     }
5637
5638     if (conf->pass_headers == NGX\_CONF\_UNSET\_PTR) {
5639         conf->pass_headers = prev->pass_headers;
5640     }
5641 }
5642
5643 if (ngx\_array\_init(&hide_headers, cf->temp_pool, 4, sizeof\(ngx\_hash\_key\_t\))
5644     != NGX\_OK)
5645 {
5646     return NGX\_ERROR;
5647 }
5648
5649 for (h = default_hide_headers; h->len; h++) {
5650     hk = ngx\_array\_push(&hide_headers);
5651     if (hk == NULL) {
5652         return NGX\_ERROR;
5653     }
5654
5655     hk->key = *h;
5656     hk->key_hash = ngx\_hash\_key\_lc(h->data, h->len);
5657     hk->value = (void *) 1;
5658 }
5659
5660 if (conf->hide_headers != NGX\_CONF\_UNSET\_PTR) {
5661
5662     h = conf->hide_headers->elts;
5663
5664     for (i = 0; i < conf->hide_headers->nelts; i++) {
5665
5666         hk = hide_headers.elts;
5667
5668         for (j = 0; j < hide_headers.nelts; j++) {
5669             if (ngx\_strcasecmp(h[i].data, hk[j].key.data) == 0) {
5670                 goto exist;
5671             }
5672         }
5673
5674         hk = ngx\_array\_push(&hide_headers);
5675         if (hk == NULL) {
5676             return NGX\_ERROR;
5677         }
5678
5679         hk->key = h[i];
5680         hk->key_hash = ngx\_hash\_key\_lc(h[i].data, h[i].len);
5681         hk->value = (void *) 1;
5682
5683     exist:
5684
5685         continue;
5686     }
5687 }
5688
5689 if (conf->pass_headers != NGX\_CONF\_UNSET\_PTR) {
5690
5691     h = conf->pass_headers->elts;
5692     hk = hide_headers.elts;
5693
5694     for (i = 0; i < conf->pass_headers->nelts; i++) {
5695         for (j = 0; j < hide_headers.nelts; j++) {

```



```

5698         if (hk[j].key.data == NULL) {
5699             continue;
5700         }
5701
5702         if (ngx_strcasecmp(h[i].data, hk[j].key.data) == 0) {
5703             hk[j].key.data = NULL;
5704             break;
5705         }
5706     }
5707 }
5708 }
5709
5710 hash->hash = &conf->hide_headers_hash;
5711 hash->key = ngx_hash_key_lc;
5712 hash->pool = cf->pool;
5713 hash->temp_pool = NULL;
5714
5715 return ngx_hash_init(hash, hide_headers.elts, hide_headers.nelts);
5716 }
5717
5718 static void *
5719 ngx_http_upstream_create_main_conf(ngx_conf_t *cf)
5720 {
5721     ngx_http_upstream_main_conf_t *umcf;
5722
5723     umcf = ngx_palloc(cf->pool, sizeof(ngx_http_upstream_main_conf_t));
5724     if (umcf == NULL) {
5725         return NULL;
5726     }
5727
5728     if (ngx_array_init(&umcf->upstreams, cf->pool, 4,
5729         sizeof(ngx_http_upstream_srv_conf_t *)))
5730     {
5731         != NGX_OK)
5732     {
5733         return NULL;
5734     }
5735
5736     return umcf;
5737 }
5738
5739 static char *
5740 ngx_http_upstream_init_main_conf(ngx_conf_t *cf, void *conf)
5741 {
5742     ngx_http_upstream_main_conf_t *umcf = conf;
5743
5744     ngx_uint_t i;
5745     ngx_array_t headers_in;
5746     ngx_hash_key_t *hk;
5747     ngx_hash_init_t hash;
5748     ngx_http_upstream_init_pt init;
5749     ngx_http_upstream_header_t *header;
5750     ngx_http_upstream_srv_conf_t **uscfp;
5751
5752     uscfp = umcf->upstreams.elts;
5753
5754     for (i = 0; i < umcf->upstreams.nelts; i++) {
5755         init = uscfp[i]->peer.init_upstream ? uscfp[i]->peer.init_upstream :
5756             ngx_http_upstream_init_round_robin;
5757
5758         if (init(cf, uscfp[i]) != NGX_OK) {
5759             return NGX_CONF_ERROR;
5760         }
5761     }
5762
5763     /* upstream_headers_in_hash */
5764
5765     if (ngx_array_init(&headers_in, cf->temp_pool, 32, sizeof(ngx_hash_key_t))
5766         != NGX_OK)
5767     {
5768         return NGX_CONF_ERROR;
5769     }
5770
5771     return NGX_CONF_ERROR;
5772 }
5773

```

```
5774 for (header = ngx_http_upstream_headers_in; header->name.len; header++) {
5775     hk = ngx_array_push(&headers_in);
5776     if (hk == NULL) {
5777         return NGX_CONF_ERROR;
5778     }
5779
5780     hk->key = header->name;
5781     hk->key_hash = ngx_hash_key_lc(header->name.data, header->name.len);
5782     hk->value = header;
5783 }
5784
5785 hash.hash = &umcf->headers_in_hash;
5786 hash.key = ngx_hash_key_lc;
5787 hash.max_size = 512;
5788 hash.bucket_size = ngx_align(64, ngx_cacheline_size);
5789 hash.name = "upstream_headers_in_hash";
5790 hash.pool = cf->pool;
5791 hash.temp_pool = NULL;
5792
5793 if (ngx_hash_init(&hash, headers_in.elts, headers_in.nelts) != NGX_OK) {
5794     return NGX_CONF_ERROR;
5795 }
5796
5797 return NGX_CONF_OK;
5798 }
```

[One Level Up](#)

[Top Level](#)

## src/http/nginx\_http\_upstream.h - nginx-1.7.10

### Data types defined

- [ngx\\_http\\_upstream\\_conf\\_t](#)
- [ngx\\_http\\_upstream\\_handler\\_pt](#)
- [ngx\\_http\\_upstream\\_header\\_t](#)
- [ngx\\_http\\_upstream\\_headers\\_in\\_t](#)
- [ngx\\_http\\_upstream\\_init\\_peer\\_pt](#)
- [ngx\\_http\\_upstream\\_init\\_pt](#)
- [ngx\\_http\\_upstream\\_local\\_t](#)
- [ngx\\_http\\_upstream\\_main\\_conf\\_t](#)
- [ngx\\_http\\_upstream\\_next\\_t](#)
- [ngx\\_http\\_upstream\\_param\\_t](#)
- [ngx\\_http\\_upstream\\_peer\\_t](#)
- [ngx\\_http\\_upstream\\_resolved\\_t](#)
- [ngx\\_http\\_upstream\\_s](#)
- [ngx\\_http\\_upstream\\_server\\_t](#)
- [ngx\\_http\\_upstream\\_srv\\_conf\\_s](#)
- [ngx\\_http\\_upstream\\_srv\\_conf\\_t](#)
- [ngx\\_http\\_upstream\\_state\\_t](#)

### Macros defined

- [NGX\\_HTTP\\_UPSTREAM\\_BACKUP](#)
- [NGX\\_HTTP\\_UPSTREAM\\_CREATE](#)
- [NGX\\_HTTP\\_UPSTREAM\\_DOWN](#)
- [NGX\\_HTTP\\_UPSTREAM\\_FAIL\\_TIMEOUT](#)
- [NGX\\_HTTP\\_UPSTREAM\\_FT\\_BUSY\\_LOCK](#)
- [NGX\\_HTTP\\_UPSTREAM\\_FT\\_ERROR](#)
- [NGX\\_HTTP\\_UPSTREAM\\_FT\\_HTTP\\_403](#)
- [NGX\\_HTTP\\_UPSTREAM\\_FT\\_HTTP\\_404](#)
- [NGX\\_HTTP\\_UPSTREAM\\_FT\\_HTTP\\_500](#)
- [NGX\\_HTTP\\_UPSTREAM\\_FT\\_HTTP\\_502](#)
- [NGX\\_HTTP\\_UPSTREAM\\_FT\\_HTTP\\_503](#)

- [NGX\\_HTTP\\_UPSTREAM\\_FT\\_HTTP\\_504](#)
- [NGX\\_HTTP\\_UPSTREAM\\_FT\\_INVALID\\_HEADER](#)
- [NGX\\_HTTP\\_UPSTREAM\\_FT\\_MAX\\_WAITING](#)
- [NGX\\_HTTP\\_UPSTREAM\\_FT\\_NOLIVE](#)
- [NGX\\_HTTP\\_UPSTREAM\\_FT\\_OFF](#)
- [NGX\\_HTTP\\_UPSTREAM\\_FT\\_STATUS](#)
- [NGX\\_HTTP\\_UPSTREAM\\_FT\\_TIMEOUT](#)
- [NGX\\_HTTP\\_UPSTREAM\\_FT\\_UPDATING](#)
- [NGX\\_HTTP\\_UPSTREAM\\_IGN\\_CACHE\\_CONTROL](#)
- [NGX\\_HTTP\\_UPSTREAM\\_IGN\\_EXPIRES](#)
- [NGX\\_HTTP\\_UPSTREAM\\_IGN\\_SET\\_COOKIE](#)
- [NGX\\_HTTP\\_UPSTREAM\\_IGN\\_VARY](#)
- [NGX\\_HTTP\\_UPSTREAM\\_IGN\\_XA\\_BUFFERING](#)
- [NGX\\_HTTP\\_UPSTREAM\\_IGN\\_XA\\_CHARSET](#)
- [NGX\\_HTTP\\_UPSTREAM\\_IGN\\_XA\\_EXPIRES](#)
- [NGX\\_HTTP\\_UPSTREAM\\_IGN\\_XA\\_LIMIT\\_RATE](#)
- [NGX\\_HTTP\\_UPSTREAM\\_IGN\\_XA\\_REDIRECT](#)
- [NGX\\_HTTP\\_UPSTREAM\\_INVALID\\_HEADER](#)
- [NGX\\_HTTP\\_UPSTREAM\\_MAX\\_FAILS](#)
- [NGX\\_HTTP\\_UPSTREAM\\_WEIGHT](#)
- [\\_NGX\\_HTTP\\_UPSTREAM\\_H\\_INCLUDED](#)
- [ngx\\_http\\_conf\\_upstream\\_srv\\_conf](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_HTTP_UPSTREAM_H_INCLUDED_
9 #define _NGX_HTTP_UPSTREAM_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14 #include <ngx_event.h>
15 #include <ngx_event_connect.h>
16 #include <ngx_event_pipe.h>
17 #include <ngx_http.h>
18
19
20 #define NGX_HTTP_UPSTREAM_FT_ERROR          0x00000002
21 #define NGX_HTTP_UPSTREAM_FT_TIMEOUT      0x00000004
22 #define NGX_HTTP_UPSTREAM_FT_INVALID_HEADER 0x00000008

```

```

23 #define NGX_HTTP_UPSTREAM_FT_HTTP_500      0x00000010
24 #define NGX_HTTP_UPSTREAM_FT_HTTP_502      0x00000020
25 #define NGX_HTTP_UPSTREAM_FT_HTTP_503      0x00000040
26 #define NGX_HTTP_UPSTREAM_FT_HTTP_504      0x00000080
27 #define NGX_HTTP_UPSTREAM_FT_HTTP_403      0x00000100
28 #define NGX_HTTP_UPSTREAM_FT_HTTP_404      0x00000200
29 #define NGX_HTTP_UPSTREAM_FT_UPDATING      0x00000400
30 #define NGX_HTTP_UPSTREAM_FT_BUSY_LOCK     0x00000800
31 #define NGX_HTTP_UPSTREAM_FT_MAX_WAITING   0x00001000
32 #define NGX_HTTP_UPSTREAM_FT_NOLIVE        0x40000000
33 #define NGX_HTTP_UPSTREAM_FT_OFF           0x80000000
34
35 #define NGX_HTTP_UPSTREAM_FT_STATUS         (NGX_HTTP_UPSTREAM_FT_HTTP_500 \
36 |NGX_HTTP_UPSTREAM_FT_HTTP_502 \
37 |NGX_HTTP_UPSTREAM_FT_HTTP_503 \
38 |NGX_HTTP_UPSTREAM_FT_HTTP_504 \
39 |NGX_HTTP_UPSTREAM_FT_HTTP_403 \
40 |NGX_HTTP_UPSTREAM_FT_HTTP_404)
41
42 #define NGX_HTTP_UPSTREAM_INVALID_HEADER    40
43
44
45 #define NGX_HTTP_UPSTREAM_IGN_XA_REDIRECT   0x00000002
46 #define NGX_HTTP_UPSTREAM_IGN_XA_EXPIRES   0x00000004
47 #define NGX_HTTP_UPSTREAM_IGN_EXPIRES      0x00000008
48 #define NGX_HTTP_UPSTREAM_IGN_CACHE_CONTROL 0x00000010
49 #define NGX_HTTP_UPSTREAM_IGN_SET_COOKIE   0x00000020
50 #define NGX_HTTP_UPSTREAM_IGN_XA_LIMIT_RATE 0x00000040
51 #define NGX_HTTP_UPSTREAM_IGN_XA_BUFFERING 0x00000080
52 #define NGX_HTTP_UPSTREAM_IGN_XA_CHARSET   0x00000100
53 #define NGX_HTTP_UPSTREAM_IGN_VARY         0x00000200
54
55
56 typedef struct {
57     ngx_msec_t                bl_time;
58     ngx_uint_t               bl_state;
59
60     ngx_uint_t               status;
61     time_t                    response_sec;
62     ngx_uint_t               response_msec;
63     time_t                    header_sec;
64     ngx_uint_t               header_msec;
65     off_t                     response_length;
66
67     ngx_str_t                 *peer;
68 } ngx_http_upstream_state_t;
69
70
71 typedef struct {
72     ngx_hash_t                headers_in_hash;
73     ngx_array_t              upstreams;
74                                 /* ngx_http_upstream_srv_conf_t */
75 } ngx_http_upstream_main_conf_t;
76
77 typedef struct ngx_http_upstream_srv_conf_s ngx_http_upstream_srv_conf_t;
78
79 typedef ngx_int_t (*ngx_http_upstream_init_pt)(ngx_conf_t *cf,
80     ngx_http_upstream_srv_conf_t *us);
81 typedef ngx_int_t (*ngx_http_upstream_init_peer_pt)(ngx_http_request_t *r,
82     ngx_http_upstream_srv_conf_t *us);
83
84
85 typedef struct {
86     ngx_http_upstream_init_pt    init_upstream;
87     ngx_http_upstream_init_peer_pt init;
88     void                          *data;
89 } ngx_http_upstream_peer_t;
90
91
92 typedef struct {
93     ngx_str_t                name;
94     ngx_addr_t               *addrs;
95     ngx_uint_t               naddrs;
96     ngx_uint_t               weight;
97     ngx_uint_t               max_fails;
98     time_t                      fail_timeout;

```

```

99
100     unsigned                                down:1;
101     unsigned                                backup:1;
102 } ngx_http_upstream_server_t;
103
104
105 #define NGX_HTTP_UPSTREAM_CREATE            0x0001
106 #define NGX_HTTP_UPSTREAM_WEIGHT          0x0002
107 #define NGX_HTTP_UPSTREAM_MAX_FAILS       0x0004
108 #define NGX_HTTP_UPSTREAM_FAIL_TIMEOUT    0x0008
109 #define NGX_HTTP_UPSTREAM_DOWN            0x0010
110 #define NGX_HTTP_UPSTREAM_BACKUP         0x0020
111
112
113 struct ngx_http_upstream_srv_conf_s {
114     ngx_http_upstream_peer_t    peer;
115     void                        **srv_conf;
116
117     ngx_array_t                 *servers; /* ngx_http_upstream_server_t */
118
119     ngx_uint_t                  flags;
120     ngx_str_t                   host;
121     u_char                      *file_name;
122     ngx_uint_t                  line;
123     in_port_t                   port;
124     in_port_t                   default_port;
125     ngx_uint_t                  no_port; /* unsigned no_port:1 */
126 };
127
128
129 typedef struct {
130     ngx_addr_t                  *addr;
131     ngx_http_complex_value_t    *value;
132 } ngx_http_upstream_local_t;
133
134
135 typedef struct {
136     ngx_http_upstream_srv_conf_t *upstream;
137
138     ngx_msec_t                  connect_timeout;
139     ngx_msec_t                  send_timeout;
140     ngx_msec_t                  read_timeout;
141     ngx_msec_t                  timeout;
142     ngx_msec_t                  next_upstream_timeout;
143
144     size_t                      send_lowat;
145     size_t                      buffer_size;
146     size_t                      limit_rate;
147
148     size_t                      busy_buffers_size;
149     size_t                      max_temp_file_size;
150     size_t                      temp_file_write_size;
151
152     size_t                      busy_buffers_size_conf;
153     size_t                      max_temp_file_size_conf;
154     size_t                      temp_file_write_size_conf;
155
156     ngx_bufs_t                  bufs;
157
158     ngx_uint_t                  ignore_headers;
159     ngx_uint_t                  next_upstream;
160     ngx_uint_t                  store_access;
161     ngx_uint_t                  next_upstream_tries;
162     ngx_flag_t                  buffering;
163     ngx_flag_t                  pass_request_headers;
164     ngx_flag_t                  pass_request_body;
165
166     ngx_flag_t                  ignore_client_abort;
167     ngx_flag_t                  intercept_errors;
168     ngx_flag_t                  cyclic_temp_file;
169     ngx_flag_t                  force_ranges;
170
171     ngx_path_t                  *temp_path;
172
173     ngx_hash_t                  hide_headers_hash;
174     ngx_array_t                  *hide_headers;

```

```

175     ngx\_array\_t                                *pass_headers;
176
177     ngx\_http\_upstream\_local\_t                   *local;
178
179     #if (NGX_HTTP_CACHE)
180     ngx\_shm\_zone\_t                                *cache_zone;
181     ngx\_http\_complex\_value\_t                     *cache_value;
182
183     ngx\_uint\_t                                    cache_min_uses;
184     ngx\_uint\_t                                    cache_use_stale;
185     ngx\_uint\_t                                    cache_methods;
186
187     ngx\_flag\_t                                    cache_lock;
188     ngx\_msec\_t                                    cache_lock_timeout;
189     ngx\_msec\_t                                    cache_lock_age;
190
191     ngx\_flag\_t                                    cache_revalidate;
192
193     ngx\_array\_t                                    *cache_valid;
194     ngx\_array\_t                                    *cache_bypass;
195     ngx\_array\_t                                    *no_cache;
196 #endif
197
198     ngx\_array\_t                                    *store_lengths;
199     ngx\_array\_t                                    *store_values;
200
201     #if (NGX_HTTP_CACHE)
202     signed                                        cache:2;
203 #endif
204     signed                                        store:2;
205     unsigned intercept_404:1;
206     unsigned change_buffering:1;
207
208     #if (NGX_HTTP_SSL)
209     ngx\_ssl\_t                                        *ssl;
210     ngx\_flag\_t                                    ssl_session_reuse;
211
212     ngx\_http\_complex\_value\_t                     *ssl_name;
213     ngx\_flag\_t                                    ssl_server_name;
214     ngx\_flag\_t                                    ssl_verify;
215 #endif
216
217     ngx\_str\_t                                    module;
218 } ngx\_http\_upstream\_conf\_t;
219
220
221 typedef struct {
222     ngx\_str\_t                                    name;
223     ngx\_http\_header\_handler\_pt                   handler;
224     ngx\_uint\_t                                    offset;
225     ngx\_http\_header\_handler\_pt                   copy_handler;
226     ngx\_uint\_t                                    conf;
227     ngx\_uint\_t                                    redirect; /* unsigned redirect:1; */
228 } ngx\_http\_upstream\_header\_t;
229
230
231 typedef struct {
232     ngx\_list\_t                                    headers;
233
234     ngx\_uint\_t                                    status_n;
235     ngx\_str\_t                                    status_line;
236
237     ngx\_table\_elt\_t                               *status;
238     ngx\_table\_elt\_t                               *date;
239     ngx\_table\_elt\_t                               *server;
240     ngx\_table\_elt\_t                               *connection;
241
242     ngx\_table\_elt\_t                               *expires;
243     ngx\_table\_elt\_t                               *etag;
244     ngx\_table\_elt\_t                               *x_accel_expires;
245     ngx\_table\_elt\_t                               *x_accel_redirect;
246     ngx\_table\_elt\_t                               *x_accel_limit_rate;
247
248     ngx\_table\_elt\_t                               *content_type;
249     ngx\_table\_elt\_t                               *content_length;
250

```

```

251     ngx_table_elt_t      *last_modified;
252     ngx_table_elt_t      *location;
253     ngx_table_elt_t      *accept_ranges;
254     ngx_table_elt_t      *www_authenticate;
255     ngx_table_elt_t      *transfer_encoding;
256     ngx_table_elt_t      *vary;
257
258 #if (NGX_HTTP_GZIP)
259     ngx_table_elt_t      *content_encoding;
260 #endif
261
262     ngx_array_t           cache_control;
263     ngx_array_t           cookies;
264
265     off_t                 content_length_n;
266     time_t                last_modified_time;
267
268     unsigned              connection_close:1;
269     unsigned              chunked:1;
270 } ngx_http_upstream_headers_in_t;
271
272
273 typedef struct {
274     ngx_str_t             host;
275     in_port_t            port;
276     ngx_uint_t           no_port; /* unsigned no_port:1 */
277
278     ngx_uint_t           naddrs;
279     ngx_addr_t           *addrs;
280
281     struct sockaddr      *sockaddr;
282     socklen_t            socklen;
283
284     ngx_resolver_ctx_t   *ctx;
285 } ngx_http_upstream_resolved_t;
286
287
288 typedef void (*ngx_http_upstream_handler_pt)(ngx_http_request_t *r,
289     ngx_http_upstream_t *u);
290
291
292 struct ngx_http_upstream_s {
293     ngx_http_upstream_handler_pt  read_event_handler;
294     ngx_http_upstream_handler_pt  write_event_handler;
295
296     ngx_peer_connection_t         peer;
297
298     ngx_event_pipe_t             *pipe;
299
300     ngx_chain_t                  *request_bufs;
301
302     ngx_output_chain_ctx_t       output;
303     ngx_chain_writer_ctx_t       writer;
304
305     ngx_http_upstream_conf_t     *conf;
306 #if (NGX_HTTP_CACHE)
307     ngx_array_t                  *caches;
308 #endif
309
310     ngx_http_upstream_headers_in_t  headers_in;
311
312     ngx_http_upstream_resolved_t   *resolved;
313
314     ngx_buf_t                       from_client;
315
316     ngx_buf_t                       buffer;
317     off_t                            length;
318
319     ngx_chain_t                     *out_bufs;
320     ngx_chain_t                     *busy_bufs;
321     ngx_chain_t                     *free_bufs;
322
323     ngx_int_t                       (*input_filter_init)(void *data);
324     ngx_int_t                       (*input_filter)(void *data, ssize_t bytes);
325     void                            *input_filter_ctx;
326

```



```

327 #if (NGX_HTTP_CACHE)
328     ngx_int_t                (*create_key)(ngx_http_request_t *r);
329 #endif
330     ngx_int_t                (*create_request)(ngx_http_request_t *r);
331     ngx_int_t                (*reinit_request)(ngx_http_request_t *r);
332     ngx_int_t                (*process_header)(ngx_http_request_t *r);
333     void                    (*abort_request)(ngx_http_request_t *r);
334     void                    (*finalize_request)(ngx_http_request_t *r,
335         ngx_int_t rc);
336     ngx_int_t                (*rewrite_redirect)(ngx_http_request_t *r,
337         ngx_table_elt_t *h, size_t prefix);
338     ngx_int_t                (*rewrite_cookie)(ngx_http_request_t *r,
339         ngx_table_elt_t *h);
340
341     ngx_msec_t                timeout;
342
343     ngx_http_upstream_state_t *state;
344
345     ngx_str_t                method;
346     ngx_str_t                schema;
347     ngx_str_t                uri;
348
349 #if (NGX_HTTP_SSL)
350     ngx_str_t                ssl_name;
351 #endif
352
353     ngx_http_cleanup_pt      *cleanup;
354
355     unsigned                 store:1;
356     unsigned                 cacheable:1;
357     unsigned                 accel:1;
358     unsigned                 ssl:1;
359 #if (NGX_HTTP_CACHE)
360     unsigned                 cache_status:3;
361 #endif
362
363     unsigned                 buffering:1;
364     unsigned                 keepalive:1;
365     unsigned                 upgrade:1;
366
367     unsigned                 request_sent:1;
368     unsigned                 header_sent:1;
369 };
370
371
372 typedef struct {
373     ngx_uint_t                status;
374     ngx_uint_t                mask;
375 } ngx_http_upstream_next_t;
376
377
378 typedef struct {
379     ngx_str_t                key;
380     ngx_str_t                value;
381     ngx_uint_t                skip_empty;
382 } ngx_http_upstream_param_t;
383
384
385 ngx_int_t ngx_http_upstream_cookie_variable(ngx_http_request_t *r,
386     ngx_http_variable_value_t *v, uintptr_t data);
387 ngx_int_t ngx_http_upstream_header_variable(ngx_http_request_t *r,
388     ngx_http_variable_value_t *v, uintptr_t data);
389
390 ngx_int_t ngx_http_upstream_create(ngx_http_request_t *r);
391 void ngx_http_upstream_init(ngx_http_request_t *r);
392 ngx_http_upstream_srv_conf_t *ngx_http_upstream_add(ngx_conf_t *cf,
393     ngx_url_t *u, ngx_uint_t flags);
394 char *ngx_http_upstream_bind_set_slot(ngx_conf_t *cf, ngx_command_t *cmd,
395     void *conf);
396 char *ngx_http_upstream_param_set_slot(ngx_conf_t *cf, ngx_command_t *cmd,
397     void *conf);
398 ngx_int_t ngx_http_upstream_hide_headers_hash(ngx_conf_t *cf,
399     ngx_http_upstream_conf_t *conf, ngx_http_upstream_conf_t *prev,
400     ngx_str_t *default_hide_headers, ngx_hash_init_t *hash);
401
402

```

```
403 #define ngx_http_conf_upstream_srv_conf(uscf, module) \
404     uscf->srv_conf[module.ctx_index]
405
406
407 extern ngx\_module\_t ngx_http_upstream_module;
408 extern ngx\_conf\_bitmask\_t ngx_http_upstream_cache_method_mask[];
409 extern ngx\_conf\_bitmask\_t ngx_http_upstream_ignore_headers_masks[];
410
411
412 #endif /* \_NGX\_HTTP\_UPSTREAM\_H\_INCLUDED */
```

[One Level Up](#)

[Top Level](#)

## src/event/nginx\_event\_pipe.c - nginx-1.7.10

### Functions defined

- [ngx\\_event\\_pipe](#)
- [ngx\\_event\\_pipe\\_add\\_free\\_buf](#)
- [ngx\\_event\\_pipe\\_copy\\_input\\_filter](#)
- [ngx\\_event\\_pipe\\_drain\\_chains](#)
- [ngx\\_event\\_pipe\\_read\\_upstream](#)
- [ngx\\_event\\_pipe\\_remove\\_shadow\\_links](#)
- [ngx\\_event\\_pipe\\_write\\_chain\\_to\\_temp\\_file](#)
- [ngx\\_event\\_pipe\\_write\\_to\\_downstream](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11 #include <ngx_event_pipe.h>
12
13
14 static ngx_int_t ngx_event_pipe_read_upstream(ngx_event_pipe_t *p);
15 static ngx_int_t ngx_event_pipe_write_to_downstream(ngx_event_pipe_t *p);
16
17 static ngx_int_t ngx_event_pipe_write_chain_to_temp_file(ngx_event_pipe_t *p);
18 static ngx_inline void ngx_event_pipe_remove_shadow_links(ngx_buf_t *buf);
19 static ngx_int_t ngx_event_pipe_drain_chains(ngx_event_pipe_t *p);
20
21
22 ngx_int_t
23 ngx_event_pipe(ngx_event_pipe_t *p, ngx_int_t do_write)
24 {
25     u_int      flags;
26     ngx_int_t  rc;
27     ngx_event_t *rev, *wev;
28
29     for ( ;; ) {
30         if (do_write) {
31             p->log->action = "sending to client";
32
33             rc = ngx_event_pipe_write_to_downstream(p);
34
35             if (rc == NGX_ABORT) {
36                 return NGX_ABORT;
37             }
38
39             if (rc == NGX_BUSY) {
40                 return NGX_OK;
41             }
42         }
43
44         p->read = 0;
45         p->upstream_blocked = 0;
46
47         p->log->action = "reading upstream";
```

```

48     if (ngx\_event\_pipe\_read\_upstream(p) == NGX\_ABORT) {
49         return NGX\_ABORT;
50     }
51
52     if (!p->read && !p->upstream_blocked) {
53         break;
54     }
55
56     do_write = 1;
57 }
58
59 if (p->upstream->fd != (ngx\_socket\_t) -1) {
60     rev = p->upstream->read;
61
62     flags = (rev->eof || rev->error) ? NGX\_CLOSE\_EVENT : 0;
63
64     if (ngx\_handle\_read\_event(rev, flags) != NGX\_OK) {
65         return NGX\_ABORT;
66     }
67
68     if (!rev->delayed) {
69         if (rev->active && !rev->ready) {
70             ngx\_add\_timer(rev, p->read_timeout);
71
72         } else if (rev->timer_set) {
73             ngx\_del\_timer(rev);
74         }
75     }
76 }
77
78 if (p->downstream->fd != (ngx\_socket\_t) -1
79     && p->downstream->data == p->output_ctx)
80 {
81     wev = p->downstream->write;
82     if (ngx\_handle\_write\_event(wev, p->send_lowat) != NGX\_OK) {
83         return NGX\_ABORT;
84     }
85
86     if (!wev->delayed) {
87         if (wev->active && !wev->ready) {
88             ngx\_add\_timer(wev, p->send_timeout);
89
90         } else if (wev->timer_set) {
91             ngx\_del\_timer(wev);
92         }
93     }
94 }
95
96 return NGX\_OK;
97 }
98
99
100
101 static ngx\_int\_t
102 ngx\_event\_pipe\_read\_upstream(ngx\_event\_pipe\_t *p)
103 {
104     off\_t         limit;
105     ssize\_t      n, size;
106     ngx\_int\_t    rc;
107     ngx\_buf\_t   *b;
108     ngx\_msec\_t  delay;
109     ngx\_chain\_t *chain, *cl, *ln;
110
111     if (p->upstream_eof || p->upstream_error || p->upstream_done) {
112         return NGX\_OK;
113     }
114
115     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_EVENT, p->log, 0,
116                 "pipe read upstream: %d", p->upstream->read->ready);
117
118     for ( ;; ) {
119
120         if (p->upstream_eof || p->upstream_error || p->upstream_done) {
121             break;
122         }
123

```

```

124     if (p->preread_bufs == NULL && !p->upstream->read->ready) {
125         break;
126     }
127
128     if (p->preread_bufs) {
129
130         /* use the pre-read bufs if they exist */
131
132         chain = p->preread_bufs;
133         p->preread_bufs = NULL;
134         n = p->preread_size;
135
136         ngx_log_debug1(NGX_LOG_DEBUG_EVENT, p->log, 0,
137             "pipe preread: %z", n);
138
139         if (n) {
140             p->read = 1;
141         }
142
143     } else {
144
145     #if (NGX_HAVE_KQUEUE)
146
147         /*
148         * kqueue notifies about the end of file or a pending error.
149         * This test allows not to allocate a buf on these conditions
150         * and not to call c->recv_chain().
151         */
152
153         if (p->upstream->read->available == 0
154             && p->upstream->read->pending_eof)
155         {
156             p->upstream->read->ready = 0;
157             p->upstream->read->eof = 1;
158             p->upstream_eof = 1;
159             p->read = 1;
160
161             if (p->upstream->read->kq_errno) {
162                 p->upstream->read->error = 1;
163                 p->upstream_error = 1;
164                 p->upstream_eof = 0;
165
166                 ngx_log_error(NGX_LOG_ERR, p->log,
167                     p->upstream->read->kq_errno,
168                     "kevent() reported that upstream "
169                     "closed connection");
170             }
171
172             break;
173         }
174     #endif
175
176         if (p->limit_rate) {
177             if (p->upstream->read->delayed) {
178                 break;
179             }
180
181             limit = (off_t) p->limit_rate * (ngx_time() - p->start_sec + 1)
182                 - p->read_length;
183
184             if (limit <= 0) {
185                 p->upstream->read->delayed = 1;
186                 delay = (ngx_msec_t) (- limit * 1000 / p->limit_rate + 1);
187                 ngx_add_timer(p->upstream->read, delay);
188                 break;
189             }
190
191         } else {
192             limit = 0;
193         }
194
195         if (p->free_raw_bufs) {
196
197             /* use the free bufs if they exist */
198
199             chain = p->free_raw_bufs;

```

```

200     if (p->single_buf) {
201         p->free_raw_bufs = p->free_raw_bufs->next;
202         chain->next = NULL;
203     } else {
204         p->free_raw_bufs = NULL;
205     }
206
207 } else if (p->allocated < p->bufs.num) {
208
209     /* allocate a new buf if it's still allowed */
210
211     b = ngx_create_temp_buf(p->pool, p->bufs.size);
212     if (b == NULL) {
213         return NGX_ABORT;
214     }
215
216     p->allocated++;
217
218     chain = ngx_alloc_chain_link(p->pool);
219     if (chain == NULL) {
220         return NGX_ABORT;
221     }
222
223     chain->buf = b;
224     chain->next = NULL;
225
226 } else if (!p->cacheable
227           && p->downstream->data == p->output_ctx
228           && p->downstream->write->ready
229           && !p->downstream->write->delayed)
230 {
231     /*
232      * if the bufs are not needed to be saved in a cache and
233      * a downstream is ready then write the bufs to a downstream
234      */
235
236     p->upstream_blocked = 1;
237
238     ngx_log_debug0(NGX_LOG_DEBUG_EVENT, p->log, 0,
239                  "pipe downstream ready");
240
241     break;
242
243 } else if (p->cacheable
244           || p->temp_file->offset < p->max_temp_file_size)
245 {
246     /*
247      * if it is allowed, then save some bufs from p->in
248      * to a temporary file, and add them to a p->out chain
249      */
250
251     rc = ngx_event_pipe_write_chain_to_temp_file(p);
252
253     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, p->log, 0,
254                  "pipe temp offset: %0", p->temp_file->offset);
255
256     if (rc == NGX_BUSY) {
257         break;
258     }
259
260     if (rc == NGX_AGAIN) {
261         if (ngx_event_flags & NGX_USE_LEVEL_EVENT
262             && p->upstream->read->active
263             && p->upstream->read->ready)
264         {
265             {
266                 if (ngx_del_event(p->upstream->read, NGX_READ_EVENT, 0)
267                     == NGX_ERROR)
268                 {
269                     return NGX_ABORT;
270                 }
271             }
272         }
273     }
274
275     if (rc != NGX_OK) {
276         return rc;

```

```

276     }
277
278     chain = p->free_raw_bufs;
279     if (p->single_buf) {
280         p->free_raw_bufs = p->free_raw_bufs->next;
281         chain->next = NULL;
282     } else {
283         p->free_raw_bufs = NULL;
284     }
285
286 } else {
287
288     /* there are no bufs to read in */
289
290     ngx_log_debug0(NGX_LOG_DEBUG_EVENT, p->log, 0,
291                  "no pipe bufs to read in");
292
293     break;
294 }
295
296 n = p->upstream->recv_chain(p->upstream, chain, limit);
297
298 ngx_log_debug1(NGX_LOG_DEBUG_EVENT, p->log, 0,
299               "pipe recv chain: %z", n);
300
301 if (p->free_raw_bufs) {
302     chain->next = p->free_raw_bufs;
303 }
304 p->free_raw_bufs = chain;
305
306 if (n == NGX_ERROR) {
307     p->upstream_error = 1;
308     return NGX_ERROR;
309 }
310
311 if (n == NGX_AGAIN) {
312     if (p->single_buf) {
313         ngx_event_pipe_remove_shadow_links(chain->buf);
314     }
315
316     break;
317 }
318
319 p->read = 1;
320
321 if (n == 0) {
322     p->upstream_eof = 1;
323     break;
324 }
325 }
326
327 delay = p->limit_rate ? (ngx_msec_t) n * 1000 / p->limit_rate : 0;
328
329 p->read_length += n;
330 cl = chain;
331 p->free_raw_bufs = NULL;
332
333 while (cl && n > 0) {
334
335     ngx_event_pipe_remove_shadow_links(cl->buf);
336
337     size = cl->buf->end - cl->buf->last;
338
339     if (n >= size) {
340         cl->buf->last = cl->buf->end;
341
342         /* STUB */ cl->buf->num = p->num++;
343
344         if (p->input_filter(p, cl->buf) == NGX_ERROR) {
345             return NGX_ABORT;
346         }
347
348         n -= size;
349         ln = cl;
350         cl = cl->next;
351         ngx_free_chain(p->pool, ln);

```

```

352     } else {
353         cl->buf->last += n;
354         n = 0;
355     }
356 }
357
358
359 if (cl) {
360     for (ln = cl; ln->next; ln = ln->next) { /* void */
361
362         ln->next = p->free_raw_bufs;
363         p->free_raw_bufs = cl;
364     }
365
366     if (delay > 0) {
367         p->upstream->read->delayed = 1;
368         ngx_add_timer(p->upstream->read, delay);
369         break;
370     }
371 }
372
373 #if (NGX_DEBUG)
374
375 for (cl = p->busy; cl; cl = cl->next) {
376     ngx_log_debug8(NGX_LOG_DEBUG_EVENT, p->log, 0,
377         "pipe buf busy s:%d t:%d f:%d "
378         "%p, pos %p, size: %z "
379         "file: %O, size: %z",
380         (cl->buf->shadow ? 1 : 0),
381         cl->buf->temporary, cl->buf->in_file,
382         cl->buf->start, cl->buf->pos,
383         cl->buf->last - cl->buf->pos,
384         cl->buf->file_pos,
385         cl->buf->file_last - cl->buf->file_pos);
386 }
387
388 for (cl = p->out; cl; cl = cl->next) {
389     ngx_log_debug8(NGX_LOG_DEBUG_EVENT, p->log, 0,
390         "pipe buf out s:%d t:%d f:%d "
391         "%p, pos %p, size: %z "
392         "file: %O, size: %z",
393         (cl->buf->shadow ? 1 : 0),
394         cl->buf->temporary, cl->buf->in_file,
395         cl->buf->start, cl->buf->pos,
396         cl->buf->last - cl->buf->pos,
397         cl->buf->file_pos,
398         cl->buf->file_last - cl->buf->file_pos);
399 }
400
401 for (cl = p->in; cl; cl = cl->next) {
402     ngx_log_debug8(NGX_LOG_DEBUG_EVENT, p->log, 0,
403         "pipe buf in s:%d t:%d f:%d "
404         "%p, pos %p, size: %z "
405         "file: %O, size: %z",
406         (cl->buf->shadow ? 1 : 0),
407         cl->buf->temporary, cl->buf->in_file,
408         cl->buf->start, cl->buf->pos,
409         cl->buf->last - cl->buf->pos,
410         cl->buf->file_pos,
411         cl->buf->file_last - cl->buf->file_pos);
412 }
413
414 for (cl = p->free_raw_bufs; cl; cl = cl->next) {
415     ngx_log_debug8(NGX_LOG_DEBUG_EVENT, p->log, 0,
416         "pipe buf free s:%d t:%d f:%d "
417         "%p, pos %p, size: %z "
418         "file: %O, size: %z",
419         (cl->buf->shadow ? 1 : 0),
420         cl->buf->temporary, cl->buf->in_file,
421         cl->buf->start, cl->buf->pos,
422         cl->buf->last - cl->buf->pos,
423         cl->buf->file_pos,
424         cl->buf->file_last - cl->buf->file_pos);
425 }
426
427 ngx_log_debug1(NGX_LOG_DEBUG_EVENT, p->log, 0,

```



```

428         "pipe length: %0", p->length);
429
430 #endif
431
432 if (p->free_raw_bufs && p->length != -1) {
433     cl = p->free_raw_bufs;
434
435     if (cl->buf->last - cl->buf->pos >= p->length) {
436
437         p->free_raw_bufs = cl->next;
438
439         /* STUB */ cl->buf->num = p->num++;
440
441         if (p->input_filter(p, cl->buf) == NGX_ERROR) {
442             return NGX_ABORT;
443         }
444
445         ngx_free_chain(p->pool, cl);
446     }
447 }
448
449 if (p->length == 0) {
450     p->upstream_done = 1;
451     p->read = 1;
452 }
453
454 if ((p->upstream_eof || p->upstream_error) && p->free_raw_bufs) {
455
456     /* STUB */ p->free_raw_bufs->buf->num = p->num++;
457
458     if (p->input_filter(p, p->free_raw_bufs->buf) == NGX_ERROR) {
459         return NGX_ABORT;
460     }
461
462     p->free_raw_bufs = p->free_raw_bufs->next;
463
464     if (p->free_bufs && p->buf_to_file == NULL) {
465         for (cl = p->free_raw_bufs; cl; cl = cl->next) {
466             if (cl->buf->shadow == NULL) {
467                 ngx_pfree(p->pool, cl->buf->start);
468             }
469         }
470     }
471 }
472
473 if (p->cacheable && (p->in || p->buf_to_file)) {
474
475     ngx_log_debug0(NGX_LOG_DEBUG_EVENT, p->log, 0,
476                 "pipe write chain");
477
478     if (ngx_event_pipe_write_chain_to_temp_file(p) == NGX_ABORT) {
479         return NGX_ABORT;
480     }
481 }
482
483 return NGX_OK;
484 }
485
486
487 static ngx_int_t
488 ngx_event_pipe_write_to_downstream(ngx_event_pipe_t *p)
489 {
490     u_char          *prev;
491     size_t          bsize;
492     ngx_int_t       rc;
493     ngx_uint_t      flush, flushed, prev_last_shadow;
494     ngx_chain_t     *out, **ll, *cl;
495     ngx_connection_t *downstream;
496
497     downstream = p->downstream;
498
499     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, p->log, 0,
500                 "pipe write downstream: %d", downstream->write->ready);
501
502     flushed = 0;
503

```

```

504 for ( ;; ) {
505     if (p->downstream_error) {
506         return ngx_event_pipe_drain_chains(p);
507     }
508
509     if (p->upstream_eof || p->upstream_error || p->upstream_done) {
510
511         /* pass the p->out and p->in chains to the output filter */
512
513         for (cl = p->busy; cl; cl = cl->next) {
514             cl->buf->recycled = 0;
515         }
516
517         if (p->out) {
518             ngx_log_debug0(NGX_LOG_DEBUG_EVENT, p->log, 0,
519                 "pipe write downstream flush out");
520
521             for (cl = p->out; cl; cl = cl->next) {
522                 cl->buf->recycled = 0;
523             }
524
525             rc = p->output_filter(p->output_ctx, p->out);
526
527             if (rc == NGX_ERROR) {
528                 p->downstream_error = 1;
529                 return ngx_event_pipe_drain_chains(p);
530             }
531
532             p->out = NULL;
533         }
534
535         if (p->in) {
536             ngx_log_debug0(NGX_LOG_DEBUG_EVENT, p->log, 0,
537                 "pipe write downstream flush in");
538
539             for (cl = p->in; cl; cl = cl->next) {
540                 cl->buf->recycled = 0;
541             }
542
543             rc = p->output_filter(p->output_ctx, p->in);
544
545             if (rc == NGX_ERROR) {
546                 p->downstream_error = 1;
547                 return ngx_event_pipe_drain_chains(p);
548             }
549
550             p->in = NULL;
551         }
552
553         ngx_log_debug0(NGX_LOG_DEBUG_EVENT, p->log, 0,
554             "pipe write downstream done");
555
556         /* TODO: free unused bufs */
557
558         p->downstream_done = 1;
559         break;
560     }
561
562     if (downstream->data != p->output_ctx
563         || !downstream->write->ready
564         || downstream->write->delayed)
565     {
566         break;
567     }
568
569     /* bsize is the size of the busy recycled bufs */
570
571     prev = NULL;
572     bsize = 0;
573
574     for (cl = p->busy; cl; cl = cl->next) {
575
576         if (cl->buf->recycled) {
577             if (prev == cl->buf->start) {
578                 continue;
579             }

```

```

580         bsize += cl->buf->end - cl->buf->start;
581         prev = cl->buf->start;
582     }
583 }
584
585
586 ngx_log_debug1(NGX_LOG_DEBUG_EVENT, p->log, 0,
587     "pipe write busy: %uz", bsize);
588
589 out = NULL;
590
591 if (bsize >= (size_t) p->busy_size) {
592     flush = 1;
593     goto flush;
594 }
595
596 flush = 0;
597 ll = NULL;
598 prev_last_shadow = 1;
599
600 for ( ;; ) {
601     if (p->out) {
602         cl = p->out;
603
604         if (cl->buf->recycled) {
605             ngx_log_error(NGX_LOG_ALERT, p->log, 0,
606                 "recycled buffer in pipe out chain");
607         }
608
609         p->out = p->out->next;
610
611     } else if (!p->cacheable && p->in) {
612         cl = p->in;
613
614         ngx_log_debug3(NGX_LOG_DEBUG_EVENT, p->log, 0,
615             "pipe write buf ls:%d %p %z",
616             cl->buf->last_shadow,
617             cl->buf->pos,
618             cl->buf->last - cl->buf->pos);
619
620         if (cl->buf->recycled && prev_last_shadow) {
621             if (bsize + cl->buf->end - cl->buf->start > p->busy_size) {
622                 flush = 1;
623                 break;
624             }
625
626             bsize += cl->buf->end - cl->buf->start;
627         }
628
629         prev_last_shadow = cl->buf->last_shadow;
630
631         p->in = p->in->next;
632
633     } else {
634         break;
635     }
636
637     cl->next = NULL;
638
639     if (out) {
640         *ll = cl;
641     } else {
642         out = cl;
643     }
644     ll = &cl->next;
645 }
646
647 flush:
648
649 ngx_log_debug2(NGX_LOG_DEBUG_EVENT, p->log, 0,
650     "pipe write: out:%p, f:%d", out, flush);
651
652 if (out == NULL) {
653
654     if (!flush) {
655         break;

```

```

656     }
657
658     /* a workaround for AIO */
659     if (flushed++ > 10) {
660         return NGX\_BUSY;
661     }
662 }
663
664 rc = p->output_filter(p->output_ctx, out);
665
666 ngx\_chain\_update\_chains(p->pool, &p->free, &p->busy, &out, p->tag);
667
668 if (rc == NGX\_ERROR) {
669     p->downstream_error = 1;
670     return ngx\_event\_pipe\_drain\_chains(p);
671 }
672
673 for (cl = p->free; cl; cl = cl->next) {
674
675     if (cl->buf->temp_file) {
676         if (p->cacheable || !p->cyclic_temp_file) {
677             continue;
678         }
679
680         /* reset p->temp_offset if all bufs had been sent */
681
682         if (cl->buf->file_last == p->temp_file->offset) {
683             p->temp_file->offset = 0;
684         }
685     }
686
687     /* TODO: free buf if p->free_bufs && upstream done */
688
689     /* add the free shadow raw buf to p->free_raw_bufs */
690
691     if (cl->buf->last_shadow) {
692         if (ngx\_event\_pipe\_add\_free\_buf(p, cl->buf->shadow) != NGX\_OK) {
693             return NGX\_ABORT;
694         }
695
696         cl->buf->last_shadow = 0;
697     }
698
699     cl->buf->shadow = NULL;
700 }
701 }
702
703 return NGX\_OK;
704 }
705
706
707 static ngx\_int\_t
708 ngx\_event\_pipe\_write\_chain\_to\_temp\_file(ngx\_event\_pipe\_t *p)
709 {
710     ssize\_t      size, bsize, n;
711     ngx\_buf\_t   *b;
712     ngx\_uint\_t  prev_last_shadow;
713     ngx\_chain\_t *cl, *tl, *next, *out, **ll, **last_out, **last_free, fl;
714
715     if (p->buf_to_file) {
716         fl.buf = p->buf_to_file;
717         fl.next = p->in;
718         out = &fl;
719
720     } else {
721         out = p->in;
722     }
723
724     if (!p->cacheable) {
725
726         size = 0;
727         cl = out;
728         ll = NULL;
729         prev_last_shadow = 1;
730
731         ngx\_log\_debug1(NGX\_LOG\_DEBUG\_EVENT, p->log, 0,

```

```

732         "pipe offset: %0", p->temp_file->offset);
733
734     do {
735         bsize = cl->buf->last - cl->buf->pos;
736
737         ngx_log_debug4(NGX_LOG_DEBUG_EVENT, p->log, 0,
738             "pipe buf ls:%d %p, pos %p, size: %z",
739             cl->buf->last_shadow, cl->buf->start,
740             cl->buf->pos, bsize);
741
742         if (prev_last_shadow
743             && ((size + bsize > p->temp_file_write_size)
744                 || (p->temp_file->offset + size + bsize
745                     > p->max_temp_file_size)))
746         {
747             break;
748         }
749
750         prev_last_shadow = cl->buf->last_shadow;
751
752         size += bsize;
753         ll = &cl->next;
754         cl = cl->next;
755
756     } while (cl);
757
758     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, p->log, 0, "size: %z", size);
759
760     if (ll == NULL) {
761         return NGX_BUSY;
762     }
763
764     if (cl) {
765         p->in = cl;
766         *ll = NULL;
767
768     } else {
769         p->in = NULL;
770         p->last_in = &p->in;
771     }
772
773 } else {
774     p->in = NULL;
775     p->last_in = &p->in;
776 }
777
778 n = ngx_write_chain_to_temp_file(p->temp_file, out);
779
780 if (n == NGX_ERROR) {
781     return NGX_ABORT;
782 }
783
784 if (p->buf_to_file) {
785     p->temp_file->offset = p->buf_to_file->last - p->buf_to_file->pos;
786     n -= p->buf_to_file->last - p->buf_to_file->pos;
787     p->buf_to_file = NULL;
788     out = out->next;
789 }
790
791 if (n > 0) {
792     /* update previous buffer or add new buffer */
793
794     if (p->out) {
795         for (cl = p->out; cl->next; cl = cl->next) { /* void */ }
796
797         b = cl->buf;
798
799         if (b->file_last == p->temp_file->offset) {
800             p->temp_file->offset += n;
801             b->file_last = p->temp_file->offset;
802             goto free;
803         }
804
805         last_out = &cl->next;
806
807     } else {

```

```

808     last_out = &p->out;
809 }
810
811 cl = ngx_chain_get_free_buf(p->pool, &p->free);
812 if (cl == NULL) {
813     return NGX_ABORT;
814 }
815
816 b = cl->buf;
817
818 ngx_memzero(b, sizeof(ngx_buf_t));
819
820 b->tag = p->tag;
821
822 b->file = &p->temp_file->file;
823 b->file_pos = p->temp_file->offset;
824 p->temp_file->offset += n;
825 b->file_last = p->temp_file->offset;
826
827 b->in_file = 1;
828 b->temp_file = 1;
829
830 *last_out = cl;
831 }
832
833 free:
834
835 for (last_free = &p->free_raw_bufs;
836     *last_free != NULL;
837     last_free = &(*last_free)->next)
838 {
839     /* void */
840 }
841
842 for (cl = out; cl; cl = next) {
843     next = cl->next;
844
845     cl->next = p->free;
846     p->free = cl;
847
848     b = cl->buf;
849
850     if (b->last_shadow) {
851
852         tl = ngx_alloc_chain_link(p->pool);
853         if (tl == NULL) {
854             return NGX_ABORT;
855         }
856
857         tl->buf = b->shadow;
858         tl->next = NULL;
859
860         *last_free = tl;
861         last_free = &tl->next;
862
863         b->shadow->pos = b->shadow->start;
864         b->shadow->last = b->shadow->start;
865
866         ngx_event_pipe_remove_shadow_links(b->shadow);
867     }
868 }
869
870 return NGX_OK;
871 }
872
873
874 /* the copy input filter */
875
876 ngx_int_t
877 ngx_event_pipe_copy_input_filter(ngx_event_pipe_t *p, ngx_buf_t *buf)
878 {
879     ngx_buf_t     *b;
880     ngx_chain_t   *cl;
881
882     if (buf->pos == buf->last) {
883         return NGX_OK;

```

```

884     }
885
886     cl = ngx_chain_get_free_buf(p->pool, &p->free);
887     if (cl == NULL) {
888         return NGX_ERROR;
889     }
890
891     b = cl->buf;
892
893     ngx_memcpy(b, buf, sizeof(ngx_buf_t));
894     b->shadow = buf;
895     b->tag = p->tag;
896     b->last_shadow = 1;
897     b->recycled = 1;
898     buf->shadow = b;
899
900     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, p->log, 0, "input buf #%d", b->num);
901
902     if (p->in) {
903         *p->last_in = cl;
904     } else {
905         p->in = cl;
906     }
907     p->last_in = &cl->next;
908
909     if (p->length == -1) {
910         return NGX_OK;
911     }
912
913     p->length -= b->last - b->pos;
914
915     return NGX_OK;
916 }
917
918
919 static ngx_inline void
920 ngx_event_pipe_remove_shadow_links(ngx_buf_t *buf)
921 {
922     ngx_buf_t *b, *next;
923
924     b = buf->shadow;
925
926     if (b == NULL) {
927         return;
928     }
929
930     while (!b->last_shadow) {
931         next = b->shadow;
932
933         b->temporary = 0;
934         b->recycled = 0;
935
936         b->shadow = NULL;
937         b = next;
938     }
939
940     b->temporary = 0;
941     b->recycled = 0;
942     b->last_shadow = 0;
943
944     b->shadow = NULL;
945
946     buf->shadow = NULL;
947 }
948
949
950 ngx_int_t
951 ngx_event_pipe_add_free_buf(ngx_event_pipe_t *p, ngx_buf_t *b)
952 {
953     ngx_chain_t *cl;
954
955     cl = ngx_alloc_chain_link(p->pool);
956     if (cl == NULL) {
957         return NGX_ERROR;
958     }
959

```

```

960     if (p->buf_to_file && b->start == p->buf_to_file->start) {
961         b->pos = p->buf_to_file->last;
962         b->last = p->buf_to_file->last;
963
964     } else {
965         b->pos = b->start;
966         b->last = b->start;
967     }
968
969     b->shadow = NULL;
970
971     cl->buf = b;
972
973     if (p->free_raw_bufs == NULL) {
974         p->free_raw_bufs = cl;
975         cl->next = NULL;
976
977         return NGX_OK;
978     }
979
980     if (p->free_raw_bufs->buf->pos == p->free_raw_bufs->buf->last) {
981
982         /* add the free buf to the list start */
983
984         cl->next = p->free_raw_bufs;
985         p->free_raw_bufs = cl;
986
987         return NGX_OK;
988     }
989
990     /* the first free buf is partially filled, thus add the free buf after it */
991
992     cl->next = p->free_raw_bufs->next;
993     p->free_raw_bufs->next = cl;
994
995     return NGX_OK;
996 }
997
998
999 static ngx_int_t
1000 ngx_event_pipe_drain_chains(ngx_event_pipe_t *p)
1001 {
1002     ngx_chain_t  *cl, *tl;
1003
1004     for ( ;; ) {
1005         if (p->busy) {
1006             cl = p->busy;
1007             p->busy = NULL;
1008
1009         } else if (p->out) {
1010             cl = p->out;
1011             p->out = NULL;
1012
1013         } else if (p->in) {
1014             cl = p->in;
1015             p->in = NULL;
1016
1017         } else {
1018             return NGX_OK;
1019         }
1020
1021         while (cl) {
1022             if (cl->buf->last_shadow) {
1023                 if (ngx_event_pipe_add_free_buf(p, cl->buf->shadow) != NGX_OK) {
1024                     return NGX_ABORT;
1025                 }
1026
1027                 cl->buf->last_shadow = 0;
1028             }
1029
1030             cl->buf->shadow = NULL;
1031             tl = cl->next;
1032             cl->next = p->free;
1033             p->free = cl;
1034             cl = tl;
1035         }

```



```
1036 }  
1037 }
```

[One Level Up](#)

[Top Level](#)

# src/event/nginx\_event\_pipe.h - nginx-1.7.10

## Data types defined

- [ngx\\_event\\_pipe\\_input\\_filter\\_pt](#)
- [ngx\\_event\\_pipe\\_output\\_filter\\_pt](#)
- [ngx\\_event\\_pipe\\_s](#)
- [ngx\\_event\\_pipe\\_t](#)

## Macros defined

- [\\_NGX\\_EVENT\\_PIPE\\_H\\_INCLUDED](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_EVENT\_PIPE\_H\_INCLUDED
9 #define \_NGX\_EVENT\_PIPE\_H\_INCLUDED
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14 #include <ngx_event.h>
15
16
17 typedef struct ngx\_event\_pipe\_s ngx_event_pipe_t;
18
19 typedef ngx\_int\_t (*ngx\_event\_pipe\_input\_filter\_pt)(ngx\_event\_pipe\_t *p,
20 ngx\_buf\_t *buf);
21
22 typedef ngx\_int\_t (*ngx\_event\_pipe\_output\_filter\_pt)(void *data,
23 ngx\_chain\_t *chain);
24
25 struct ngx\_event\_pipe\_s {
26     ngx\_connection\_t *upstream;
27     ngx\_connection\_t *downstream;
28
29     ngx\_chain\_t *free_raw_bufs;
30     ngx\_chain\_t *in;
31     ngx\_chain\_t **last_in;
32
33     ngx\_chain\_t *out;
34     ngx\_chain\_t *free;
35     ngx\_chain\_t *busy;
36
37     /*
38      * the input filter i.e. that moves HTTP/1.1 chunks
39      * from the raw bufs to an incoming chain
40      */
41
42     ngx\_event\_pipe\_input\_filter\_pt input_filter;
43     void *input_ctx;
44
45     ngx\_event\_pipe\_output\_filter\_pt output_filter;
46     void *output_ctx;
47
48     unsigned read:1;
49     unsigned cacheable:1;
50     unsigned single_buf:1;
```

```

51     unsigned        free_bufs:1;
52     unsigned        upstream_done:1;
53     unsigned        upstream_error:1;
54     unsigned        upstream_eof:1;
55     unsigned        upstream_blocked:1;
56     unsigned        downstream_done:1;
57     unsigned        downstream_error:1;
58     unsigned        cyclic_temp_file:1;
59
60     ngx\_int\_t         allocated;
61     ngx\_bufs\_t       bufs;
62     ngx\_buf\_tag\_t    tag;
63
64     ssize_t         busy_size;
65
66     off_t           read_length;
67     off_t           length;
68
69     off_t           max_temp_file_size;
70     ssize_t        temp_file_write_size;
71
72     ngx\_msec\_t       read_timeout;
73     ngx\_msec\_t       send_timeout;
74     ssize_t        send_lowat;
75
76     ngx\_pool\_t       *pool;
77     ngx\_log\_t        *log;
78
79     ngx\_chain\_t     *preread_bufs;
80     size_t         preread_size;
81     ngx\_buf\_t       *buf_to_file;
82
83     size_t         limit_rate;
84     time_t         start_sec;
85
86     ngx\_temp\_file\_t *temp_file;
87
88     /* STUB */ int  num;
89 };
90
91
92 ngx\_int\_t ngx\_event\_pipe(ngx\_event\_pipe\_t *p, ngx\_int\_t do_write);
93 ngx\_int\_t ngx\_event\_pipe\_copy\_input\_filter(ngx\_event\_pipe\_t *p, ngx\_buf\_t *buf);
94 ngx\_int\_t ngx\_event\_pipe\_add\_free\_buf(ngx\_event\_pipe\_t *p, ngx\_buf\_t *b);
95
96
97 #endif /* \_NGX\_EVENT\_PIPE\_H\_INCLUDED\_ */

```

[One Level Up](#)

[Top Level](#)

# src/http/nginx\_http\_upstream\_round\_robin.c - nginx-1.7.10

## Functions defined

- [ngx\\_http\\_upstream\\_create\\_round\\_robin\\_peer](#)
- [ngx\\_http\\_upstream\\_empty\\_save\\_session](#)
- [ngx\\_http\\_upstream\\_empty\\_set\\_session](#)
- [ngx\\_http\\_upstream\\_free\\_round\\_robin\\_peer](#)
- [ngx\\_http\\_upstream\\_get\\_peer](#)
- [ngx\\_http\\_upstream\\_get\\_round\\_robin\\_peer](#)
- [ngx\\_http\\_upstream\\_init\\_round\\_robin](#)
- [ngx\\_http\\_upstream\\_init\\_round\\_robin\\_peer](#)
- [ngx\\_http\\_upstream\\_save\\_round\\_robin\\_peer\\_session](#)
- [ngx\\_http\\_upstream\\_set\\_round\\_robin\\_peer\\_session](#)

## Macros defined

- [ngx\\_http\\_upstream\\_tries](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 #define ngx_http_upstream_tries(p) ((p)->number \
14     + ((p)->next ? (p)->next->number : 0))
15
16
17 static ngx_http_upstream_rr_peer_t *ngx_http_upstream_get_peer(
18     ngx_http_upstream_rr_peer_data_t *rrp);
19
20 #if (NGX_HTTP_SSL)
21
22 static ngx_int_t ngx_http_upstream_empty_set_session(ngx_peer_connection_t *pc,
23     void *data);
24 static void ngx_http_upstream_empty_save_session(ngx_peer_connection_t *pc,
25     void *data);
26
27 #endif
28
29
30 ngx_int_t
31 ngx_http_upstream_init_round_robin(ngx_conf_t *cf,
32     ngx_http_upstream_srv_conf_t *us)
33 {
34     ngx_url_t          u;
35     ngx_uint_t         i, j, n, w;
36     ngx_http_upstream_server_t *server;
```

```

37 ngx_http_upstream_rr_peer_t *peer;
38 ngx_http_upstream_rr_peers_t *peers, *backup;
39
40 us->peer.init = ngx_http_upstream_init_round_robin_peer;
41
42 if (us->servers) {
43     server = us->servers->elts;
44
45     n = 0;
46     w = 0;
47
48     for (i = 0; i < us->servers->nelts; i++) {
49         if (server[i].backup) {
50             continue;
51         }
52
53         n += server[i].naddrs;
54         w += server[i].naddrs * server[i].weight;
55     }
56
57     if (n == 0) {
58         ngx_log_error(NGX_LOG_EMERG, cf->log, 0,
59             "no servers in upstream \"%V\" in %s:%ui",
60             &us->host, us->file_name, us->line);
61         return NGX_ERROR;
62     }
63
64     peers = ngx_palloc(cf->pool, sizeof(ngx_http_upstream_rr_peers_t)
65         + sizeof(ngx_http_upstream_rr_peer_t) * (n - 1));
66     if (peers == NULL) {
67         return NGX_ERROR;
68     }
69
70     peers->single = (n == 1);
71     peers->number = n;
72     peers->weighted = (w != n);
73     peers->total_weight = w;
74     peers->name = &us->host;
75
76     n = 0;
77     peer = peers->peer;
78
79     for (i = 0; i < us->servers->nelts; i++) {
80         if (server[i].backup) {
81             continue;
82         }
83
84         for (j = 0; j < server[i].naddrs; j++) {
85             peer[n].sockaddr = server[i].addrs[j].sockaddr;
86             peer[n].socklen = server[i].addrs[j].socklen;
87             peer[n].name = server[i].addrs[j].name;
88             peer[n].weight = server[i].weight;
89             peer[n].effective_weight = server[i].weight;
90             peer[n].current_weight = 0;
91             peer[n].max_fails = server[i].max_fails;
92             peer[n].fail_timeout = server[i].fail_timeout;
93             peer[n].down = server[i].down;
94             peer[n].server = server[i].name;
95             n++;
96         }
97     }
98
99     us->peer.data = peers;
100
101     /* backup servers */
102
103     n = 0;
104     w = 0;
105
106     for (i = 0; i < us->servers->nelts; i++) {
107         if (!server[i].backup) {
108             continue;
109         }
110
111         n += server[i].naddrs;
112         w += server[i].naddrs * server[i].weight;

```

```

113     }
114
115     if (n == 0) {
116         return NGX_OK;
117     }
118
119     backup = ngx_palloc(cf->pool, sizeof(ngx_http_upstream_rr_peers_t)
120         + sizeof(ngx_http_upstream_rr_peer_t) * (n - 1));
121     if (backup == NULL) {
122         return NGX_ERROR;
123     }
124
125     peers->single = 0;
126     backup->single = 0;
127     backup->number = n;
128     backup->weighted = (w != n);
129     backup->total_weight = w;
130     backup->name = &us->host;
131
132     n = 0;
133     peer = backup->peer;
134
135     for (i = 0; i < us->servers->nelts; i++) {
136         if (!server[i].backup) {
137             continue;
138         }
139
140         for (j = 0; j < server[i].naddrs; j++) {
141             peer[n].sockaddr = server[i].addrs[j].sockaddr;
142             peer[n].socklen = server[i].addrs[j].socklen;
143             peer[n].name = server[i].addrs[j].name;
144             peer[n].weight = server[i].weight;
145             peer[n].effective_weight = server[i].weight;
146             peer[n].current_weight = 0;
147             peer[n].max_fails = server[i].max_fails;
148             peer[n].fail_timeout = server[i].fail_timeout;
149             peer[n].down = server[i].down;
150             peer[n].server = server[i].name;
151             n++;
152         }
153     }
154
155     peers->next = backup;
156
157     return NGX_OK;
158 }
159
160
161 /* an upstream implicitly defined by proxy_pass, etc. */
162
163 if (us->port == 0) {
164     ngx_log_error(NGX_LOG_EMERG, cf->log, 0,
165         "no port in upstream \"%V\" in %s:%ui",
166         &us->host, us->file_name, us->line);
167     return NGX_ERROR;
168 }
169
170 ngx_memzero(&u, sizeof(ngx_url_t));
171
172 u.host = us->host;
173 u.port = us->port;
174
175 if (ngx_inet_resolve_host(cf->pool, &u) != NGX_OK) {
176     if (u.err) {
177         ngx_log_error(NGX_LOG_EMERG, cf->log, 0,
178             "%s in upstream \"%V\" in %s:%ui",
179             u.err, &us->host, us->file_name, us->line);
180     }
181
182     return NGX_ERROR;
183 }
184
185 n = u.naddrs;
186
187 peers = ngx_palloc(cf->pool, sizeof(ngx_http_upstream_rr_peers_t)
188     + sizeof(ngx_http_upstream_rr_peer_t) * (n - 1));

```

```

189     if (peers == NULL) {
190         return NGX\_ERROR;
191     }
192
193     peers->single = (n == 1);
194     peers->number = n;
195     peers->weighted = 0;
196     peers->total_weight = n;
197     peers->name = &us->host;
198
199     peer = peers->peer;
200
201     for (i = 0; i < u.naddrs; i++) {
202         peer[i].sockaddr = u.addrs[i].sockaddr;
203         peer[i].socklen = u.addrs[i].socklen;
204         peer[i].name = u.addrs[i].name;
205         peer[i].weight = 1;
206         peer[i].effective_weight = 1;
207         peer[i].current_weight = 0;
208         peer[i].max_fails = 1;
209         peer[i].fail_timeout = 10;
210     }
211
212     us->peer.data = peers;
213
214     /* implicitly defined upstream has no backup servers */
215
216     return NGX\_OK;
217 }
218
219
220 ngx\_int\_t
221 ngx\_http\_upstream\_init\_round\_robin\_peer(ngx\_http\_request\_t *r,
222     ngx\_http\_upstream\_srv\_conf\_t *us)
223 {
224     ngx\_uint\_t n;
225     ngx\_http\_upstream\_rr\_peer\_data\_t *rrp;
226
227     rrp = r->upstream->peer.data;
228
229     if (rrp == NULL) {
230         rrp = ngx\_palloc(r->pool, sizeof(ngx\_http\_upstream\_rr\_peer\_data\_t));
231         if (rrp == NULL) {
232             return NGX\_ERROR;
233         }
234
235         r->upstream->peer.data = rrp;
236     }
237
238     rrp->peers = us->peer.data;
239     rrp->current = 0;
240
241     n = rrp->peers->number;
242
243     if (rrp->peers->next && rrp->peers->next->number > n) {
244         n = rrp->peers->next->number;
245     }
246
247     if (n <= 8 * sizeof(uintptr\_t)) {
248         rrp->tried = &rrp->data;
249         rrp->data = 0;
250
251     } else {
252         n = (n + (8 * sizeof(uintptr\_t) - 1)) / (8 * sizeof(uintptr\_t));
253
254         rrp->tried = ngx\_pcalloc(r->pool, n * sizeof(uintptr\_t));
255         if (rrp->tried == NULL) {
256             return NGX\_ERROR;
257         }
258     }
259
260     r->upstream->peer.get = ngx\_http\_upstream\_get\_round\_robin\_peer;
261     r->upstream->peer.free = ngx\_http\_upstream\_free\_round\_robin\_peer;
262     r->upstream->peer.tries = ngx\_http\_upstream\_tries(rrp->peers);
263     #if (NGX\_HTTP\_SSL)
264     r->upstream->peer.set_session =

```

```

265         ngx_http_upstream_set_round_robin_peer_session;
266     r->upstream->peer.save_session =
267         ngx_http_upstream_save_round_robin_peer_session;
268 #endif
269
270     return NGX_OK;
271 }
272
273
274 ngx_int_t
275 ngx_http_upstream_create_round_robin_peer(ngx_http_request_t *r,
276     ngx_http_upstream_resolved_t *ur)
277 {
278     u_char          *p;
279     size_t          len;
280     socklen_t       socklen;
281     ngx_uint_t      i, n;
282     struct sockaddr *sockaddr;
283     ngx_http_upstream_rr_peer_t *peer;
284     ngx_http_upstream_rr_peers_t *peers;
285     ngx_http_upstream_rr_peer_data_t *rrp;
286
287     rrp = r->upstream->peer.data;
288
289     if (rrp == NULL) {
290         rrp = ngx_palloc(r->pool, sizeof(ngx_http_upstream_rr_peer_data_t));
291         if (rrp == NULL) {
292             return NGX_ERROR;
293         }
294
295         r->upstream->peer.data = rrp;
296     }
297
298     peers = ngx_palloc(r->pool, sizeof(ngx_http_upstream_rr_peers_t)
299         + sizeof(ngx_http_upstream_rr_peer_t) * (ur->naddrs - 1));
300     if (peers == NULL) {
301         return NGX_ERROR;
302     }
303
304     peers->single = (ur->naddrs == 1);
305     peers->number = ur->naddrs;
306     peers->name = &ur->host;
307
308     peer = peers->peer;
309
310     if (ur->sockaddr) {
311         peer[0].sockaddr = ur->sockaddr;
312         peer[0].socklen = ur->socklen;
313         peer[0].name = ur->host;
314         peer[0].weight = 1;
315         peer[0].effective_weight = 1;
316         peer[0].current_weight = 0;
317         peer[0].max_fails = 1;
318         peer[0].fail_timeout = 10;
319
320     } else {
321
322         for (i = 0; i < ur->naddrs; i++) {
323
324             socklen = ur->addrs[i].socklen;
325
326             sockaddr = ngx_palloc(r->pool, socklen);
327             if (sockaddr == NULL) {
328                 return NGX_ERROR;
329             }
330
331             ngx_memcpy(sockaddr, ur->addrs[i].sockaddr, socklen);
332
333             switch (sockaddr->sa_family) {
334 #if (NGX_HAVE_INET6)
335                 case AF_INET6:
336                     ((struct sockaddr_in6 *) sockaddr)->sin6_port = htons(ur->port);
337                     break;
338 #endif
339                 default: /* AF_INET */
340                     ((struct sockaddr_in *) sockaddr)->sin_port = htons(ur->port);

```



```

341     }
342
343     p = ngx_pnalloc(r->pool, NGX_SOCKADDR_STRLEN);
344     if (p == NULL) {
345         return NGX_ERROR;
346     }
347
348     len = ngx_sock_ntop(sockaddr, socklen, p, NGX_SOCKADDR_STRLEN, 1);
349
350     peer[i].sockaddr = sockaddr;
351     peer[i].socklen = socklen;
352     peer[i].name.len = len;
353     peer[i].name.data = p;
354     peer[i].weight = 1;
355     peer[i].effective_weight = 1;
356     peer[i].current_weight = 0;
357     peer[i].max_fails = 1;
358     peer[i].fail_timeout = 10;
359 }
360 }
361
362 rrp->peers = peers;
363 rrp->current = 0;
364
365 if (rrp->peers->number <= 8 * sizeof(uintptr_t)) {
366     rrp->tried = &rrp->data;
367     rrp->data = 0;
368 } else {
369     n = (rrp->peers->number + (8 * sizeof(uintptr_t) - 1))
370         / (8 * sizeof(uintptr_t));
371
372     rrp->tried = ngx_pcalloc(r->pool, n * sizeof(uintptr_t));
373     if (rrp->tried == NULL) {
374         return NGX_ERROR;
375     }
376 }
377 }
378
379 r->upstream->peer.get = ngx_http_upstream_get_round_robin_peer;
380 r->upstream->peer.free = ngx_http_upstream_free_round_robin_peer;
381 r->upstream->peer.tries = ngx_http_upstream_tries(rrp->peers);
382 #if (NGX_HTTP_SSL)
383 r->upstream->peer.set_session = ngx_http_upstream_empty_set_session;
384 r->upstream->peer.save_session = ngx_http_upstream_empty_save_session;
385 #endif
386
387 return NGX_OK;
388 }
389
390
391 ngx_int_t
392 ngx_http_upstream_get_round_robin_peer(ngx_peer_connection_t *pc, void *data)
393 {
394     ngx_http_upstream_rr_peer_data_t *rrp = data;
395
396     ngx_int_t rc;
397     ngx_uint_t i, n;
398     ngx_http_upstream_rr_peer_t *peer;
399     ngx_http_upstream_rr_peers_t *peers;
400
401     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, pc->log, 0,
402         "get rr peer, try: %ui", pc->tries);
403
404     pc->cached = 0;
405     pc->connection = NULL;
406
407     peers = rrp->peers;
408
409     /* ngx_lock_mutex(peers->mutex); */
410
411     if (peers->single) {
412         peer = &peers->peer[0];
413
414         if (peer->down) {
415             goto failed;
416         }

```

```

417 } else {
418
419     /* there are several peers */
420
421     peer = ngx_http_upstream_get_peer(rrp);
422
423     if (peer == NULL) {
424         goto failed;
425     }
426
427     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, pc->log, 0,
428                  "get rr peer, current: %ui %i",
429                  rrp->current, peer->current_weight);
430
431 }
432
433 pc->sockaddr = peer->sockaddr;
434 pc->socklen = peer->socklen;
435 pc->name = &peer->name;
436
437 /* ngx_unlock_mutex(peers->mutex); */
438
439 return NGX_OK;
440
441 failed:
442
443 if (peers->next) {
444
445     /* ngx_unlock_mutex(peers->mutex); */
446
447     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, pc->log, 0, "backup servers");
448
449     rrp->peers = peers->next;
450
451     n = (rrp->peers->number + (8 * sizeof(uintptr_t) - 1))
452         / (8 * sizeof(uintptr_t));
453
454     for (i = 0; i < n; i++) {
455         rrp->tried[i] = 0;
456     }
457
458     rc = ngx_http_upstream_get_round_robin_peer(pc, rrp);
459
460     if (rc != NGX_BUSY) {
461         return rc;
462     }
463
464     /* ngx_lock_mutex(peers->mutex); */
465 }
466
467 /* all peers failed, mark them as live for quick recovery */
468
469 for (i = 0; i < peers->number; i++) {
470     peers->peer[i].fails = 0;
471 }
472
473 /* ngx_unlock_mutex(peers->mutex); */
474
475 pc->name = peers->name;
476
477 return NGX_BUSY;
478 }
479
480
481 static ngx_http_upstream_rr_peer_t *
482 ngx_http_upstream_get_peer(ngx_http_upstream_rr_peer_data_t *rrp)
483 {
484     time_t          now;
485     uintptr_t       m;
486     ngx_int_t      total;
487     ngx_uint_t     i, n;
488     ngx_http_upstream_rr_peer_t *peer, *best;
489
490     now = ngx_time();
491
492     best = NULL;

```

```

493     total = 0;
494
495     for (i = 0; i < rrp->peers->number; i++) {
496
497         n = i / (8 * sizeof(uintptr_t));
498         m = (uintptr_t) 1 << i % (8 * sizeof(uintptr_t));
499
500         if (rrp->tried[n] & m) {
501             continue;
502         }
503
504         peer = &rrp->peers->peer[i];
505
506         if (peer->down) {
507             continue;
508         }
509
510         if (peer->max_fails
511             && peer->fails >= peer->max_fails
512             && now - peer->checked <= peer->fail_timeout)
513         {
514             continue;
515         }
516
517         peer->current_weight += peer->effective_weight;
518         total += peer->effective_weight;
519
520         if (peer->effective_weight < peer->weight) {
521             peer->effective_weight++;
522         }
523
524         if (best == NULL || peer->current_weight > best->current_weight) {
525             best = peer;
526         }
527     }
528
529     if (best == NULL) {
530         return NULL;
531     }
532
533     i = best - &rrp->peers->peer[0];
534
535     rrp->current = i;
536
537     n = i / (8 * sizeof(uintptr_t));
538     m = (uintptr_t) 1 << i % (8 * sizeof(uintptr_t));
539
540     rrp->tried[n] |= m;
541
542     best->current_weight -= total;
543
544     if (now - best->checked > best->fail_timeout) {
545         best->checked = now;
546     }
547
548     return best;
549 }
550
551
552 void
553 ngx_http_upstream_free_round_robin_peer(ngx_peer_connection_t *pc, void *data,
554     ngx_uint_t state)
555 {
556     ngx_http_upstream_rr_peer_data_t *rrp = data;
557
558     time_t now;
559     ngx_http_upstream_rr_peer_t *peer;
560
561     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, pc->log, 0,
562         "free rr peer %ui %ui", pc->tries, state);
563
564     /* TODO: NGX_PEER_KEEPALIVE */
565
566     if (rrp->peers->single) {
567         pc->tries = 0;
568         return;

```

```

569     }
570
571     peer = &rrp->peers->peer[rrp->current];
572
573     if (state & NGX\_PEER\_FAILED) {
574         now = ngx\_time();
575
576         /* ngx_lock_mutex(rrp->peers->mutex); */
577
578         peer->fails++;
579         peer->accessed = now;
580         peer->checked = now;
581
582         if (peer->max_fails) {
583             peer->effective_weight -= peer->weight / peer->max_fails;
584         }
585
586         ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, pc->log, 0,
587                     "free rr peer failed: %ui %i",
588                     rrp->current, peer->effective_weight);
589
590         if (peer->effective_weight < 0) {
591             peer->effective_weight = 0;
592         }
593
594         /* ngx_unlock_mutex(rrp->peers->mutex); */
595     } else {
596
597         /* mark peer live if check passed */
598
599         if (peer->accessed < peer->checked) {
600             peer->fails = 0;
601         }
602     }
603
604     if (pc->tries) {
605         pc->tries--;
606     }
607
608     /* ngx_unlock_mutex(rrp->peers->mutex); */
609 }
610
611
612
613 #if (NGX\_HTTP\_SSL)
614
615 ngx\_int\_t
616 ngx\_http\_upstream\_set\_round\_robin\_peer\_session(ngx\_peer\_connection\_t *pc,
617 void *data)
618 {
619     ngx\_http\_upstream\_rr\_peer\_data\_t *rrp = data;
620
621     ngx\_int\_t rc;
622     ngx\_ssl\_session\_t *ssl_session;
623     ngx\_http\_upstream\_rr\_peer\_t *peer;
624
625     peer = &rrp->peers->peer[rrp->current];
626
627     /* TODO: threads only mutex */
628     /* ngx_lock_mutex(rrp->peers->mutex); */
629
630     ssl_session = peer->ssl_session;
631
632     rc = ngx\_ssl\_set\_session(pc->connection, ssl_session);
633
634     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, pc->log, 0,
635                 "set session: %p", ssl_session);
636
637     /* ngx_unlock_mutex(rrp->peers->mutex); */
638
639     return rc;
640 }
641
642
643 void
644 ngx\_http\_upstream\_save\_round\_robin\_peer\_session(ngx\_peer\_connection\_t *pc,

```

```

645 void *data)
646 {
647     ngx\_http\_upstream\_rr\_peer\_data\_t *rrp = data;
648
649     ngx\_ssl\_session\_t *old_ssl_session, *ssl_session;
650     ngx\_http\_upstream\_rr\_peer\_t *peer;
651
652     ssl_session = ngx\_ssl\_get\_session(pc->connection);
653
654     if (ssl_session == NULL) {
655         return;
656     }
657
658     ngx\_log\_debug1(NGX_LOG_DEBUG_HTTP, pc->log, 0,
659         "save session: %p", ssl_session);
660
661     peer = &rrp->peers->peer[rrp->current];
662
663     /* TODO: threads only mutex */
664     /* ngx\_lock\_mutex(rrp->peers->mutex); */
665
666     old_ssl_session = peer->ssl_session;
667     peer->ssl_session = ssl_session;
668
669     /* ngx\_unlock\_mutex(rrp->peers->mutex); */
670
671     if (old_ssl_session) {
672
673         ngx\_log\_debug1(NGX_LOG_DEBUG_HTTP, pc->log, 0,
674             "old session: %p", old_ssl_session);
675
676         /* TODO: may block */
677
678         ngx\_ssl\_free\_session(old_ssl_session);
679     }
680 }
681
682
683 static ngx\_int\_t
684 ngx\_http\_upstream\_empty\_set\_session(ngx\_peer\_connection\_t *pc, void *data)
685 {
686     return NGX_OK;
687 }
688
689
690 static void
691 ngx\_http\_upstream\_empty\_save\_session(ngx\_peer\_connection\_t *pc, void *data)
692 {
693     return;
694 }
695
696 #endif

```

[One Level Up](#)

[Top Level](#)

# src/http/nginx\_http\_upstream\_round\_robin.h - nginx-1.7.10

## Data types defined

- [ngx\\_http\\_upstream\\_rr\\_peer\\_data\\_t](#)
- [ngx\\_http\\_upstream\\_rr\\_peer\\_t](#)
- [ngx\\_http\\_upstream\\_rr\\_peers\\_s](#)
- [ngx\\_http\\_upstream\\_rr\\_peers\\_t](#)

## Macros defined

- [\\_NGX\\_HTTP\\_UPSTREAM\\_ROUND\\_ROBIN\\_H\\_INCLUDED\\_](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_HTTP\_UPSTREAM\_ROUND\_ROBIN\_H\_INCLUDED\_
9 #define \_NGX\_HTTP\_UPSTREAM\_ROUND\_ROBIN\_H\_INCLUDED\_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14 #include <ngx_http.h>
15
16
17 typedef struct {
18     struct sockaddr      *sockaddr;
19     socklen_t            socklen;
20     ngx\_str\_t              name;
21     ngx\_str\_t              server;
22
23     ngx\_int\_t              current_weight;
24     ngx\_int\_t              effective_weight;
25     ngx\_int\_t              weight;
26
27     ngx\_uint\_t            fails;
28     time_t               accessed;
29     time_t               checked;
30
31     ngx\_uint\_t            max_fails;
32     time_t               fail_timeout;
33
34     ngx\_uint\_t            down;          /* unsigned down:1; */
35
36 #if (NGX_HTTP_SSL)
37     ngx\_ssl\_session\_t    *ssl_session; /* local to a process */
38 #endif
39 } ngx_http_upstream_rr_peer_t;
40
41
42 typedef struct ngx\_http\_upstream\_rr\_peers\_s ngx_http_upstream_rr_peers_t;
43
44 struct ngx_http_upstream_rr_peers_s {
45     ngx\_uint\_t            number;
46
47     /* ngx\_mutex\_t          *mutex; */
48
49     ngx\_uint\_t            total_weight;
50
```

```

51     unsigned                single:1;
52     unsigned                weighted:1;
53
54     ngx\_str\_t                 *name;
55
56     ngx\_http\_upstream\_rr\_peers\_t *next;
57
58     ngx\_http\_upstream\_rr\_peer\_t peer[1];
59 };
60
61
62 typedef struct {
63     ngx\_http\_upstream\_rr\_peers\_t *peers;
64     ngx\_uint\_t                   current;
65     uintptr\_t                   *tried;
66     uintptr\_t                   data;
67 } ngx\_http\_upstream\_rr\_peer\_data\_t;
68
69
70 ngx\_int\_t ngx\_http\_upstream\_init\_round\_robin(ngx\_conf\_t *cf,
71     ngx\_http\_upstream\_srv\_conf\_t *us);
72 ngx\_int\_t ngx\_http\_upstream\_init\_round\_robin\_peer(ngx\_http\_request\_t *r,
73     ngx\_http\_upstream\_srv\_conf\_t *us);
74 ngx\_int\_t ngx\_http\_upstream\_create\_round\_robin\_peer(ngx\_http\_request\_t *r,
75     ngx\_http\_upstream\_resolved\_t *ur);
76 ngx\_int\_t ngx\_http\_upstream\_get\_round\_robin\_peer(ngx\_peer\_connection\_t *pc,
77     void *data);
78 void ngx\_http\_upstream\_free\_round\_robin\_peer(ngx\_peer\_connection\_t *pc,
79     void *data, ngx\_uint\_t state);
80
81 #if (NGX_HTTP_SSL)
82 ngx\_int\_t
83     ngx\_http\_upstream\_set\_round\_robin\_peer\_session(ngx\_peer\_connection\_t *pc,
84     void *data);
85 void ngx\_http\_upstream\_save\_round\_robin\_peer\_session(ngx\_peer\_connection\_t *pc,
86     void *data);
87 #endif
88
89
90 #endif /* \_NGX\_HTTP\_UPSTREAM\_ROUND\_ROBIN\_H\_INCLUDED */

```

[One Level Up](#)

[Top Level](#)

## src/http/nginx\_http\_file\_cache.c - nginx-1.7.10

### Global variables defined

- [ngx\\_http\\_cache\\_status](#)
- [ngx\\_http\\_file\\_cache\\_key](#)

### Functions defined

- [ngx\\_http\\_cache aio event handler](#)
- [ngx\\_http\\_cache\\_send](#)
- [ngx\\_http\\_file\\_cache\\_add](#)
- [ngx\\_http\\_file\\_cache\\_add\\_file](#)
- [ngx\\_http\\_file\\_cache aio read](#)
- [ngx\\_http\\_file\\_cache\\_cleanup](#)
- [ngx\\_http\\_file\\_cache\\_create](#)
- [ngx\\_http\\_file\\_cache\\_create\\_key](#)
- [ngx\\_http\\_file\\_cache\\_delete](#)
- [ngx\\_http\\_file\\_cache\\_delete\\_file](#)
- [ngx\\_http\\_file\\_cache\\_exists](#)
- [ngx\\_http\\_file\\_cache\\_expire](#)
- [ngx\\_http\\_file\\_cache\\_forced\\_expire](#)
- [ngx\\_http\\_file\\_cache\\_free](#)
- [ngx\\_http\\_file\\_cache\\_init](#)
- [ngx\\_http\\_file\\_cache\\_loader](#)
- [ngx\\_http\\_file\\_cache\\_loader\\_sleep](#)
- [ngx\\_http\\_file\\_cache\\_lock](#)
- [ngx\\_http\\_file\\_cache\\_lock\\_wait](#)
- [ngx\\_http\\_file\\_cache\\_lock\\_wait\\_handler](#)
- [ngx\\_http\\_file\\_cache\\_lookup](#)
- [ngx\\_http\\_file\\_cache\\_manage\\_directory](#)
- [ngx\\_http\\_file\\_cache\\_manage\\_file](#)
- [ngx\\_http\\_file\\_cache\\_manager](#)
- [ngx\\_http\\_file\\_cache\\_name](#)
- [ngx\\_http\\_file\\_cache\\_new](#)



- [ngx http file cache noop](#)
- [ngx http file cache open](#)
- [ngx http file cache rbtree insert value](#)
- [ngx http file cache read](#)
- [ngx http file cache reopen](#)
- [ngx http file cache set header](#)
- [ngx http file cache set slot](#)
- [ngx http file cache update](#)
- [ngx http file cache update header](#)
- [ngx http file cache update variant](#)
- [ngx http file cache valid](#)
- [ngx http file cache valid set slot](#)
- [ngx http file cache vary](#)
- [ngx http file cache vary header](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11 #include <ngx_md5.h>
12
13
14 static ngx_int_t ngx_http_file_cache_lock(ngx_http_request_t *r,
15     ngx_http_cache_t *c);
16 static void ngx_http_file_cache_lock_wait_handler(ngx_event_t *ev);
17 static void ngx_http_file_cache_lock_wait(ngx_http_request_t *r,
18     ngx_http_cache_t *c);
19 static ngx_int_t ngx_http_file_cache_read(ngx_http_request_t *r,
20     ngx_http_cache_t *c);
21 static ssize_t ngx_http_file_cache_aio_read(ngx_http_request_t *r,
22     ngx_http_cache_t *c);
23 #if (NGX_HAVE_FILE_AIO)
24 static void ngx_http_cache_aio_event_handler(ngx_event_t *ev);
25 #endif
26 static ngx_int_t ngx_http_file_cache_exists(ngx_http_file_cache_t *cache,
27     ngx_http_cache_t *c);
28 static ngx_int_t ngx_http_file_cache_name(ngx_http_request_t *r,
29     ngx_path_t *path);
30 static ngx_http_file_cache_node_t *
31     ngx_http_file_cache_lookup(ngx_http_file_cache_t *cache, u_char *key);
32 static void ngx_http_file_cache_rbtree_insert_value(ngx_rbtree_node_t *temp,
33     ngx_rbtree_node_t *node, ngx_rbtree_node_t *sentinel);
34 static void ngx_http_file_cache_vary(ngx_http_request_t *r, u_char *vary,
35     size_t len, u_char *hash);
36 static void ngx_http_file_cache_vary_header(ngx_http_request_t *r,
37     ngx_md5_t *md5, ngx_str_t *name);
38 static ngx_int_t ngx_http_file_cache_reopen(ngx_http_request_t *r,
39     ngx_http_cache_t *c);
40 static ngx_int_t ngx_http_file_cache_update_variant(ngx_http_request_t *r,
41     ngx_http_cache_t *c);

```

```

42 static void ngx_http_file_cache_cleanup(void *data);
43 static time_t ngx_http_file_cache_forced_expire(ngx_http_file_cache_t *cache);
44 static time_t ngx_http_file_cache_expire(ngx_http_file_cache_t *cache);
45 static void ngx_http_file_cache_delete(ngx_http_file_cache_t *cache,
46     ngx_queue_t *q, u_char *name);
47 static void ngx_http_file_cache_loader_sleep(ngx_http_file_cache_t *cache);
48 static ngx_int_t ngx_http_file_cache_noop(ngx_tree_ctx_t *ctx,
49     ngx_str_t *path);
50 static ngx_int_t ngx_http_file_cache_manage_file(ngx_tree_ctx_t *ctx,
51     ngx_str_t *path);
52 static ngx_int_t ngx_http_file_cache_manage_directory(ngx_tree_ctx_t *ctx,
53     ngx_str_t *path);
54 static ngx_int_t ngx_http_file_cache_add_file(ngx_tree_ctx_t *ctx,
55     ngx_str_t *path);
56 static ngx_int_t ngx_http_file_cache_add(ngx_http_file_cache_t *cache,
57     ngx_http_cache_t *c);
58 static ngx_int_t ngx_http_file_cache_delete_file(ngx_tree_ctx_t *ctx,
59     ngx_str_t *path);
60
61
62 ngx_str_t ngx_http_cache_status[] = {
63     ngx_string("MISS"),
64     ngx_string("BYPASS"),
65     ngx_string("EXPIRED"),
66     ngx_string("STALE"),
67     ngx_string("UPDATING"),
68     ngx_string("REVALIDATED"),
69     ngx_string("HIT")
70 };
71
72
73 static u_char ngx_http_file_cache_key[] = { LF, 'K', 'E', 'Y', ':', ' ' };
74
75
76 static ngx_int_t
77 ngx_http_file_cache_init(ngx_shm_zone_t *shm_zone, void *data)
78 {
79     ngx_http_file_cache_t *ocache = data;
80
81     size_t len;
82     ngx_uint_t n;
83     ngx_http_file_cache_t *cache;
84
85     cache = shm_zone->data;
86
87     if (ocache) {
88         if (ngx_strcmp(cache->path->name.data, ocache->path->name.data) != 0) {
89             ngx_log_error(NGX_LOG_EMERG, shm_zone->shm.log, 0,
90                 "cache \"%V\" uses the \"%V\" cache path "
91                 "while previously it used the \"%V\" cache path",
92                 &shm_zone->shm.name, &cache->path->name,
93                 &ocache->path->name);
94
95             return NGX_ERROR;
96         }
97
98         for (n = 0; n < 3; n++) {
99             if (cache->path->level[n] != ocache->path->level[n]) {
100                 ngx_log_error(NGX_LOG_EMERG, shm_zone->shm.log, 0,
101                     "cache \"%V\" had previously different levels",
102                     &shm_zone->shm.name);
103                 return NGX_ERROR;
104             }
105         }
106
107         cache->sh = ocache->sh;
108
109         cache->shpool = ocache->shpool;
110         cache->bsize = ocache->bsize;
111
112         cache->max_size /= cache->bsize;
113
114         if (!cache->sh->cold || cache->sh->loading) {
115             cache->path->loader = NULL;
116         }
117

```

```

118     return NGX\_OK;
119 }
120
121 cache->shpool = (ngx\_slab\_pool\_t *) shm_zone->shm.addr;
122
123 if (shm_zone->shm.exists) {
124     cache->sh = cache->shpool->data;
125     cache->bsize = ngx\_fs\_bsize(cache->path->name.data);
126
127     return NGX\_OK;
128 }
129
130 cache->sh = ngx\_slab\_alloc(cache->shpool, sizeof\(ngx\_http\_file\_cache\_sh\_t\));
131 if (cache->sh == NULL) {
132     return NGX\_ERROR;
133 }
134
135 cache->shpool->data = cache->sh;
136
137 ngx\_rbtree\_init(&cache->sh->rbtree, &cache->sh->sentinel,
138                ngx\_http\_file\_cache\_rbtree\_insert\_value);
139
140 ngx\_queue\_init(&cache->sh->queue);
141
142 cache->sh->cold = 1;
143 cache->sh->loading = 0;
144 cache->sh->size = 0;
145
146 cache->bsize = ngx\_fs\_bsize(cache->path->name.data);
147
148 cache->max_size /= cache->bsize;
149
150 len = sizeof(" in cache keys zone \\\"") + shm_zone->shm.name.len;
151
152 cache->shpool->log_ctx = ngx\_slab\_alloc(cache->shpool, len);
153 if (cache->shpool->log_ctx == NULL) {
154     return NGX\_ERROR;
155 }
156
157 ngx\_sprintf(cache->shpool->log_ctx, " in cache keys zone \"%V\\\"%Z",
158            &shm_zone->shm.name);
159
160 cache->shpool->log_nomem = 0;
161
162 return NGX\_OK;
163 }
164
165
166 ngx\_int\_t
167 ngx\_http\_file\_cache\_new(ngx\_http\_request\_t *r)
168 {
169     ngx\_http\_cache\_t *c;
170
171     c = ngx\_palloc(r->pool, sizeof\(ngx\_http\_cache\_t\));
172     if (c == NULL) {
173         return NGX\_ERROR;
174     }
175
176     if (ngx\_array\_init(&c->keys, r->pool, 4, sizeof\(ngx\_str\_t\)) != NGX\_OK) {
177         return NGX\_ERROR;
178     }
179
180     r->cache = c;
181     c->file.log = r->connection->log;
182     c->file.fd = NGX\_INVALID\_FILE;
183
184     c->last_modified = -1;
185
186     return NGX\_OK;
187 }
188
189
190 ngx\_int\_t
191 ngx\_http\_file\_cache\_create(ngx\_http\_request\_t *r)
192 {
193     ngx\_http\_cache\_t *c;

```

```

194     ngx_pool_cleanup_t    *cIn;
195     ngx_http_file_cache_t *cache;
196
197     c = r->cache;
198     cache = c->file_cache;
199
200     cIn = ngx_pool_cleanup_add(r->pool, 0);
201     if (cIn == NULL) {
202         return NGX_ERROR;
203     }
204
205     cIn->handler = ngx_http_file_cache_cleanup;
206     cIn->data = c;
207
208     if (ngx_http_file_cache_exists(cache, c) == NGX_ERROR) {
209         return NGX_ERROR;
210     }
211
212     if (ngx_http_file_cache_name(r, cache->path) != NGX_OK) {
213         return NGX_ERROR;
214     }
215
216     return NGX_OK;
217 }
218
219
220 void
221 ngx_http_file_cache_create_key(ngx_http_request_t *r)
222 {
223     size_t          len;
224     ngx_str_t       *key;
225     ngx_uint_t      i;
226     ngx_md5_t       md5;
227     ngx_http_cache_t *c;
228
229     c = r->cache;
230
231     len = 0;
232
233     ngx_crc32_init(c->crc32);
234     ngx_md5_init(&md5);
235
236     key = c->keys.elts;
237     for (i = 0; i < c->keys.nelts; i++) {
238         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
239             "http cache key: \"%V\"", &key[i]);
240
241         len += key[i].len;
242
243         ngx_crc32_update(&c->crc32, key[i].data, key[i].len);
244         ngx_md5_update(&md5, key[i].data, key[i].len);
245     }
246
247     c->header_start = sizeof(ngx_http_file_cache_header_t)
248         + sizeof(ngx_http_file_cache_key) + len + 1;
249
250     ngx_crc32_final(c->crc32);
251     ngx_md5_final(c->key, &md5);
252
253     ngx_memcpy(c->main, c->key, NGX_HTTP_CACHE_KEY_LEN);
254 }
255
256
257 ngx_int_t
258 ngx_http_file_cache_open(ngx_http_request_t *r)
259 {
260     ngx_int_t          rc, rv;
261     ngx_uint_t         cold, test;
262     ngx_http_cache_t   *c;
263     ngx_pool_cleanup_t *cIn;
264     ngx_open_file_info_t of;
265     ngx_http_file_cache_t *cache;
266     ngx_http_core_loc_conf_t *clcf;
267
268     c = r->cache;
269

```

```

270     if (c->waiting) {
271         return NGX\_AGAIN;
272     }
273
274     if (c->reading) {
275         return ngx\_http\_file\_cache\_read(r, c);
276     }
277
278     cache = c->file_cache;
279
280     if (c->node == NULL) {
281         cln = ngx\_pool\_cleanup\_add(r->pool, 0);
282         if (cln == NULL) {
283             return NGX\_ERROR;
284         }
285
286         cln->handler = ngx\_http\_file\_cache\_cleanup;
287         cln->data = c;
288     }
289
290     rc = ngx\_http\_file\_cache\_exists(cache, c);
291
292     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
293                 "http file cache exists: %i e:%d", rc, c->exists);
294
295     if (rc == NGX\_ERROR) {
296         return rc;
297     }
298
299     if (rc == NGX\_AGAIN) {
300         return NGX\_HTTP\_CACHE\_SCARCE;
301     }
302
303     cold = cache->sh->cold;
304
305     if (rc == NGX\_OK) {
306
307         if (c->error) {
308             return c->error;
309         }
310
311         c->temp_file = 1;
312         test = c->exists ? 1 : 0;
313         rv = NGX\_DECLINED;
314
315     } else { /* rc == NGX\_DECLINED */
316
317         if (c->min_uses > 1) {
318
319             if (!cold) {
320                 return NGX\_HTTP\_CACHE\_SCARCE;
321             }
322
323             test = 1;
324             rv = NGX\_HTTP\_CACHE\_SCARCE;
325
326         } else {
327             c->temp_file = 1;
328             test = cold ? 1 : 0;
329             rv = NGX\_DECLINED;
330         }
331     }
332
333     if (ngx\_http\_file\_cache\_name(r, cache->path) != NGX\_OK) {
334         return NGX\_ERROR;
335     }
336
337     if (!test) {
338         goto done;
339     }
340
341     clcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_core\_module);
342
343     ngx\_memzero(&of, sizeof(ngx\_open\_file\_info\_t));
344
345     of.uniq = c->uniq;

```

```

346 of.valid = clcf->open_file_cache_valid;
347 of.min_uses = clcf->open_file_cache_min_uses;
348 of.events = clcf->open_file_cache_events;
349 of.directio = NGX\_OPEN\_FILE\_DIRECTIO\_OFF;
350 of.read_ahead = clcf->read_ahead;
351
352 if (ngx\_open\_cached\_file(clcf->open_file_cache, &c->file.name, &of, r->pool)
353     != NGX\_OK)
354 {
355     switch (of.err) {
356
357     case 0:
358         return NGX\_ERROR;
359
360     case NGX\_ENOENT:
361     case NGX\_ENOTDIR:
362         goto done;
363
364     default:
365         ngx\_log\_error(NGX\_LOG\_CRIT, r->connection->log, of.err,
366             ngx\_open\_file\_n " \"%s\" failed", c->file.name.data);
367         return NGX\_ERROR;
368     }
369 }
370
371 ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
372     "http file cache fd: %d", of.fd);
373
374 c->file.fd = of.fd;
375 c->file.log = r->connection->log;
376 c->uniq = of.uniq;
377 c->length = of.size;
378 c->fs_size = (of.fs_size + cache->bsize - 1) / cache->bsize;
379
380 c->buf = ngx\_create\_temp\_buf(r->pool, c->body_start);
381 if (c->buf == NULL) {
382     return NGX\_ERROR;
383 }
384
385 return ngx\_http\_file\_cache\_read(r, c);
386
387 done:
388
389 if (rv == NGX\_DECLINED) {
390     return ngx\_http\_file\_cache\_lock(r, c);
391 }
392
393 return rv;
394 }
395
396
397 static ngx\_int\_t
398 ngx\_http\_file\_cache\_lock(ngx\_http\_request\_t *r, ngx\_http\_cache\_t *c)
399 {
400     ngx\_msec\_t          now, timer;
401     ngx\_http\_file\_cache\_t *cache;
402
403     if (!c->lock) {
404         return NGX\_DECLINED;
405     }
406
407     now = ngx\_current\_msec;
408
409     cache = c->file_cache;
410
411     ngx\_shmtx\_lock(&cache->shpool->mutex);
412
413     timer = c->node->lock_time - now;
414
415     if (!c->node->updating || (ngx\_msec\_int\_t) timer <= 0) {
416         c->node->updating = 1;
417         c->node->lock_time = now + c->lock_age;
418         c->updating = 1;
419         c->lock_time = c->node->lock_time;
420     }
421

```

```

422     ngx_shmtx_unlock(&cache->shpool->mutex);
423
424     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
425                 "http file cache lock u:%d wt:%M",
426                 c->updating, c->wait_time);
427
428     if (c->updating) {
429         return NGX_DECLINED;
430     }
431
432     if (c->lock_timeout == 0) {
433         return NGX_HTTP_CACHE_SCARCE;
434     }
435
436     c->waiting = 1;
437
438     if (c->wait_time == 0) {
439         c->wait_time = now + c->lock_timeout;
440
441         c->wait_event.handler = ngx_http_file_cache_lock_wait_handler;
442         c->wait_event.data = r;
443         c->wait_event.log = r->connection->log;
444     }
445
446     timer = c->wait_time - now;
447
448     ngx_add_timer(&c->wait_event, (timer > 500) ? 500 : timer);
449
450     r->main->blocked++;
451
452     return NGX_AGAIN;
453 }
454
455
456 static void
457 ngx_http_file_cache_lock_wait_handler(ngx_event_t *ev)
458 {
459     ngx_connection_t *c;
460     ngx_http_request_t *r;
461
462     r = ev->data;
463     c = r->connection;
464
465     ngx_http_set_log_request(c->log, r);
466
467     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
468                 "http file cache wait: \"%V?%V\"", &r->uri, &r->args);
469
470     ngx_http_file_cache_lock_wait(r, r->cache);
471
472     ngx_http_run_posted_requests(c);
473 }
474
475
476 static void
477 ngx_http_file_cache_lock_wait(ngx_http_request_t *r, ngx_http_cache_t *c)
478 {
479     ngx_uint_t wait;
480     ngx_msec_t now, timer;
481     ngx_http_file_cache_t *cache;
482
483     now = ngx_current_msec;
484
485     timer = c->wait_time - now;
486
487     if ((ngx_msec_int_t) timer <= 0) {
488         ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
489                     "cache lock timeout");
490         c->lock_timeout = 0;
491         goto wakeup;
492     }
493
494     cache = c->file_cache;
495     wait = 0;
496
497     ngx_shmtx_lock(&cache->shpool->mutex);

```

```

498 timer = c->node->lock_time - now;
499
500
501 if (c->node->updating && (ngx_msec_int_t) timer > 0) {
502     wait = 1;
503 }
504
505 ngx_shmtx_unlock(&cache->shpool->mutex);
506
507 if (wait) {
508     ngx_add_timer(&c->wait_event, (timer > 500) ? 500 : timer);
509     return;
510 }
511
512 wakeup:
513
514 c->waiting = 0;
515 r->main->blocked--;
516 r->write_event_handler(r);
517 }
518
519
520 static ngx_int_t
521 ngx_http_file_cache_read(ngx_http_request_t *r, ngx_http_cache_t *c)
522 {
523     time_t          now;
524     ssize_t         n;
525     ngx_int_t       rc;
526     ngx_http_file_cache_t *cache;
527     ngx_http_file_cache_header_t *h;
528
529     n = ngx_http_file_cache_aio_read(r, c);
530
531     if (n < 0) {
532         return n;
533     }
534
535     if ((size_t) n < c->header_start) {
536         ngx_log_error(NGX_LOG_CRIT, r->connection->log, 0,
537             "cache file \"%s\" is too small", c->file.name.data);
538         return NGX_DECLINED;
539     }
540
541     h = (ngx_http_file_cache_header_t *) c->buf->pos;
542
543     if (h->version != NGX_HTTP_CACHE_VERSION) {
544         ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
545             "cache file \"%s\" version mismatch", c->file.name.data);
546         return NGX_DECLINED;
547     }
548
549     if (h->crc32 != c->crc32) {
550         ngx_log_error(NGX_LOG_CRIT, r->connection->log, 0,
551             "cache file \"%s\" has md5 collision", c->file.name.data);
552         return NGX_DECLINED;
553     }
554
555     if ((size_t) h->body_start > c->body_start) {
556         ngx_log_error(NGX_LOG_CRIT, r->connection->log, 0,
557             "cache file \"%s\" has too long header",
558             c->file.name.data);
559         return NGX_DECLINED;
560     }
561
562     if (h->vary_len > NGX_HTTP_CACHE_VARY_LEN) {
563         ngx_log_error(NGX_LOG_CRIT, r->connection->log, 0,
564             "cache file \"%s\" has incorrect vary length",
565             c->file.name.data);
566         return NGX_DECLINED;
567     }
568
569     if (h->vary_len) {
570         ngx_http_file_cache_vary(r, h->vary, h->vary_len, c->variant);
571
572         if (ngx_memcmp(c->variant, h->variant, NGX_HTTP_CACHE_KEY_LEN) != 0) {
573             ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,

```



```

574         "http file cache vary mismatch");
575     return ngx_http_file_cache_reopen(r, c);
576 }
577 }
578
579 c->buf->last += n;
580
581 c->valid_sec = h->valid_sec;
582 c->last_modified = h->last_modified;
583 c->date = h->date;
584 c->valid_msec = h->valid_msec;
585 c->header_start = h->header_start;
586 c->body_start = h->body_start;
587 c->etag.len = h->etag.len;
588 c->etag.data = h->etag;
589
590 r->cached = 1;
591
592 cache = c->file_cache;
593
594 if (cache->sh->cold) {
595     ngx_shmtx_lock(&cache->shpool->mutex);
596
597     if (!c->node->exists) {
598         c->node->uses = 1;
599         c->node->body_start = c->body_start;
600         c->node->exists = 1;
601         c->node->uniq = c->uniq;
602         c->node->fs_size = c->fs_size;
603
604         cache->sh->size += c->fs_size;
605     }
606
607     ngx_shmtx_unlock(&cache->shpool->mutex);
608 }
609
610 now = ngx_time();
611
612 if (c->valid_sec < now) {
613     ngx_shmtx_lock(&cache->shpool->mutex);
614
615     if (c->node->updating) {
616         rc = NGX_HTTP_CACHE_UPDATING;
617     } else {
618         c->node->updating = 1;
619         c->updating = 1;
620         c->lock_time = c->node->lock_time;
621         rc = NGX_HTTP_CACHE_STALE;
622     }
623
624     ngx_shmtx_unlock(&cache->shpool->mutex);
625
626     ngx_log_debug3(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
627         "http file cache expired: %i %T %T",
628         rc, c->valid_sec, now);
629
630     return rc;
631 }
632
633 return NGX_OK;
634 }
635
636 }
637
638
639
640 static ssize_t
641 ngx_http_file_cache_aio_read(ngx_http_request_t *r, ngx_http_cache_t *c)
642 {
643     #if (NGX_HAVE_FILE_AIO)
644         ssize_t          n;
645         ngx_http_core_loc_conf_t *clcf;
646
647         if (!ngx_file_aio) {
648             goto noaio;
649         }

```

```

650     clcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_core\_module);
651
652     if (!clcf->aio) {
653         goto noaio;
654     }
655
656     n = ngx\_file\_aio\_read(&c->file, c->buf->pos, c->body_start, 0, r->pool);
657
658     if (n != NGX\_AGAIN) {
659         c->reading = 0;
660         return n;
661     }
662
663     c->reading = 1;
664
665     c->file.aio->data = r;
666     c->file.aio->handler = ngx\_http\_cache\_aio\_event\_handler;
667
668     r->main->blocked++;
669     r->aio = 1;
670
671     return NGX\_AGAIN;
672
673 noaio:
674
675 #endif
676
677     return ngx\_read\_file(&c->file, c->buf->pos, c->body_start, 0);
678 }
679
680
681 #if (NGX\_HAVE\_FILE\_AIO)
682
683 static void
684 ngx\_http\_cache\_aio\_event\_handler(ngx\_event\_t *ev)
685 {
686     ngx\_event\_aio\_t    *aio;
687     ngx\_connection\_t   *c;
688     ngx\_http\_request\_t *r;
689
690     aio = ev->data;
691     r = aio->data;
692     c = r->connection;
693
694     ngx\_http\_set\_log\_request(c->log, r);
695
696     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, c->log, 0,
697                 "http file cache aio: \"%V?%V\"", &r->uri, &r->args);
698
699     r->main->blocked--;
700     r->aio = 0;
701
702     r->write_event_handler(r);
703
704     ngx\_http\_run\_posted\_requests(c);
705 }
706
707 #endif
708
709
710 static ngx\_int\_t
711 ngx\_http\_file\_cache\_exists(ngx\_http\_file\_cache\_t *cache, ngx\_http\_cache\_t *c)
712 {
713     ngx\_int\_t          rc;
714     ngx\_http\_file\_cache\_node\_t *fcn;
715
716     ngx\_shmtx\_lock(&cache->shpool->mutex);
717
718     fcn = c->node;
719
720     if (fcn == NULL) {
721         fcn = ngx\_http\_file\_cache\_lookup(cache, c->key);
722     }
723
724     if (fcn) {

```

```

726     ngx_queue_remove(&fcn->queue);
727
728     if (c->node == NULL) {
729         fcn->uses++;
730         fcn->count++;
731     }
732
733     if (fcn->error) {
734
735         if (fcn->valid_sec < ngx_time()) {
736             goto renew;
737         }
738
739         rc = NGX_OK;
740
741         goto done;
742     }
743
744     if (fcn->exists || fcn->uses >= c->min_uses) {
745
746         c->exists = fcn->exists;
747         if (fcn->body_start) {
748             c->body_start = fcn->body_start;
749         }
750
751         rc = NGX_OK;
752
753         goto done;
754     }
755
756     rc = NGX_AGAIN;
757
758     goto done;
759 }
760
761 fcn = ngx_slab_alloc_locked(cache->shpool,
762                             sizeof(ngx_http_file_cache_node_t));
763 if (fcn == NULL) {
764     ngx_shmtx_unlock(&cache->shpool->mutex);
765
766     (void) ngx_http_file_cache_forced_expire(cache);
767
768     ngx_shmtx_lock(&cache->shpool->mutex);
769
770     fcn = ngx_slab_alloc_locked(cache->shpool,
771                                 sizeof(ngx_http_file_cache_node_t));
772     if (fcn == NULL) {
773         ngx_log_error(NGX_LOG_ALERT, ngx_cycle->log, 0,
774                     "could not allocate node%s", cache->shpool->log_ctx);
775         rc = NGX_ERROR;
776         goto failed;
777     }
778 }
779
780 ngx_memcpy((u_char *) &fcn->node.key, c->key, sizeof(ngx_rbtree_key_t));
781
782 ngx_memcpy(fcn->key, &c->key[sizeof(ngx_rbtree_key_t)],
783           NGX_HTTP_CACHE_KEY_LEN - sizeof(ngx_rbtree_key_t));
784
785 ngx_rbtree_insert(&cache->sh->rbtree, &fcn->node);
786
787 fcn->uses = 1;
788 fcn->count = 1;
789
790 renew:
791
792 rc = NGX_DECLINED;
793
794 fcn->valid_msec = 0;
795 fcn->error = 0;
796 fcn->exists = 0;
797 fcn->valid_sec = 0;
798 fcn->uniq = 0;
799 fcn->body_start = 0;
800 fcn->fs_size = 0;
801

```

```

802 done:
803
804     fcn->expire = ngx_time() + cache->inactive;
805
806     ngx_queue_insert_head(&cache->sh->queue, &fcn->queue);
807
808     c->uniq = fcn->uniq;
809     c->error = fcn->error;
810     c->node = fcn;
811
812 failed:
813
814     ngx_shmtx_unlock(&cache->shpool->mutex);
815
816     return rc;
817 }
818
819
820 static ngx_int_t
821 ngx_http_file_cache_name(ngx_http_request_t *r, ngx_path_t *path)
822 {
823     u_char          *p;
824     ngx_http_cache_t *c;
825
826     c = r->cache;
827
828     if (c->file.name.len) {
829         return NGX_OK;
830     }
831
832     c->file.name.len = path->name.len + 1 + path->len
833                     + 2 * NGX_HTTP_CACHE_KEY_LEN;
834
835     c->file.name.data = ngx_pnalloc(r->pool, c->file.name.len + 1);
836     if (c->file.name.data == NULL) {
837         return NGX_ERROR;
838     }
839
840     ngx_memcpy(c->file.name.data, path->name.data, path->name.len);
841
842     p = c->file.name.data + path->name.len + 1 + path->len;
843     p = ngx_hex_dump(p, c->key, NGX_HTTP_CACHE_KEY_LEN);
844     *p = '\0';
845
846     ngx_create_hashed_filename(path, c->file.name.data, c->file.name.len);
847
848     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
849                 "cache file: \"%s\"", c->file.name.data);
850
851     return NGX_OK;
852 }
853
854
855 static ngx_http_file_cache_node_t *
856 ngx_http_file_cache_lookup(ngx_http_file_cache_t *cache, u_char *key)
857 {
858     ngx_int_t          rc;
859     ngx_rbtrees_key_t  node_key;
860     ngx_rbtrees_node_t *node, *sentinel;
861     ngx_http_file_cache_node_t *fcn;
862
863     ngx_memcpy((u_char *) &node_key, key, sizeof(ngx_rbtrees_key_t));
864
865     node = cache->sh->rbtree.root;
866     sentinel = cache->sh->rbtree.sentinel;
867
868     while (node != sentinel) {
869
870         if (node_key < node->key) {
871             node = node->left;
872             continue;
873         }
874
875         if (node_key > node->key) {
876             node = node->right;
877             continue;

```

```

878     }
879
880     /* node_key == node->key */
881
882     fcn = (ngx\_http\_file\_cache\_node\_t *) node;
883
884     rc = ngx\_memcmp(&key[sizeof\(ngx\_rbtree\_key\_t\)], fcn->key,
885                   NGX\_HTTP\_CACHE\_KEY\_LEN - sizeof\(ngx\_rbtree\_key\_t\));
886
887     if (rc == 0) {
888         return fcn;
889     }
890
891     node = (rc < 0) ? node->left : node->right;
892 }
893
894 /* not found */
895
896 return NULL;
897 }
898
899
900 static void
901 ngx\_http\_file\_cache\_rbtree\_insert\_value(ngx\_rbtree\_node\_t *temp,
902 ngx\_rbtree\_node\_t *node, ngx\_rbtree\_node\_t *sentinel)
903 {
904     ngx\_rbtree\_node\_t      **p;
905     ngx\_http\_file\_cache\_node\_t *cn, *cnt;
906
907     for ( ;; ) {
908         if (node->key < temp->key) {
909             p = &temp->left;
910
911         } else if (node->key > temp->key) {
912             p = &temp->right;
913
914         } else { /* node->key == temp->key */
915
916             cn = (ngx\_http\_file\_cache\_node\_t *) node;
917             cnt = (ngx\_http\_file\_cache\_node\_t *) temp;
918
919             p = (ngx\_memcmp(cn->key, cnt->key,
920                           NGX\_HTTP\_CACHE\_KEY\_LEN - sizeof\(ngx\_rbtree\_key\_t\))
921                 < 0)
922                 ? &temp->left : &temp->right;
923
924         }
925
926         if (*p == sentinel) {
927             break;
928         }
929
930         temp = *p;
931     }
932
933     *p = node;
934     node->parent = temp;
935     node->left = sentinel;
936     node->right = sentinel;
937     ngx\_rbt\_red(node);
938 }
939
940
941
942
943 static void
944 ngx\_http\_file\_cache\_vary(ngx\_http\_request\_t *r, u_char *vary, size_t len,
945 u_char *hash)
946 {
947     u_char      *p, *last;
948     ngx\_str\_t   name;
949     ngx\_md5\_t   md5;
950     u_char      buf[NGX\_HTTP\_CACHE\_VARY\_LEN];
951
952     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
953                  "http file cache vary: \"%s\"", len, vary);

```

```

954     ngx_md5_init(&md5);
955     ngx_md5_update(&md5, r->cache->main, NGX_HTTP_CACHE_KEY_LEN);
956
957
958     ngx_strlow(buf, vary, len);
959
960     p = buf;
961     last = buf + len;
962
963     while (p < last) {
964         while (p < last && (*p == ' ' || *p == ',')) { p++; }
965
966         name.data = p;
967
968         while (p < last && *p != ',' && *p != ' ') { p++; }
969
970         name.len = p - name.data;
971
972         if (name.len == 0) {
973             break;
974         }
975
976         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
977             "http file cache vary: %V", &name);
978
979         ngx_md5_update(&md5, name.data, name.len);
980         ngx_md5_update(&md5, (u_char *) ":", sizeof(":") - 1);
981
982         ngx_http_file_cache_vary_header(r, &md5, &name);
983
984         ngx_md5_update(&md5, (u_char *) CRLE, sizeof(CRLE) - 1);
985     }
986
987     ngx_md5_final(hash, &md5);
988 }
989
990
991
992 static void
993 ngx_http_file_cache_vary_header(ngx_http_request_t *r, ngx_md5_t *md5,
994     ngx_str_t *name)
995 {
996     size_t      len;
997     u_char      *p, *start, *last;
998     ngx_uint_t   i, multiple, normalize;
999     ngx_list_part_t *part;
1000     ngx_table_elt_t *header;
1001
1002     multiple = 0;
1003     normalize = 0;
1004
1005     if (name->len == sizeof("Accept-Charset") - 1
1006         && ngx_strncasecmp(name->data, (u_char *) "Accept-Charset",
1007             sizeof("Accept-Charset") - 1) == 0)
1008     {
1009         normalize = 1;
1010     }
1011     else if (name->len == sizeof("Accept-Encoding") - 1
1012         && ngx_strncasecmp(name->data, (u_char *) "Accept-Encoding",
1013             sizeof("Accept-Encoding") - 1) == 0)
1014     {
1015         normalize = 1;
1016     }
1017     else if (name->len == sizeof("Accept-Language") - 1
1018         && ngx_strncasecmp(name->data, (u_char *) "Accept-Language",
1019             sizeof("Accept-Language") - 1) == 0)
1020     {
1021         normalize = 1;
1022     }
1023
1024     part = &r->headers_in.headers.part;
1025     header = part->elts;
1026
1027     for (i = 0; /* void */ ; i++) {
1028         if (i >= part->nelts) {

```

```

1030         if (part->next == NULL) {
1031             break;
1032         }
1033
1034         part = part->next;
1035         header = part->elts;
1036         i = 0;
1037     }
1038
1039     if (header[i].hash == 0) {
1040         continue;
1041     }
1042
1043     if (header[i].key.len != name->len) {
1044         continue;
1045     }
1046
1047     if (ngx_strncasecmp(header[i].key.data, name->data, name->len) != 0) {
1048         continue;
1049     }
1050
1051     if (!normalize) {
1052
1053         if (multiple) {
1054             ngx_md5_update(md5, (u_char *) ",", sizeof(",") - 1);
1055         }
1056
1057         ngx_md5_update(md5, header[i].value.data, header[i].value.len);
1058
1059         multiple = 1;
1060
1061         continue;
1062     }
1063
1064     /* normalize spaces */
1065
1066     p = header[i].value.data;
1067     last = p + header[i].value.len;
1068
1069     while (p < last) {
1070
1071         while (p < last && (*p == ' ' || *p == ',')) { p++; }
1072
1073         start = p;
1074
1075         while (p < last && *p != ',' && *p != ' ') { p++; }
1076
1077         len = p - start;
1078
1079         if (len == 0) {
1080             break;
1081         }
1082
1083         if (multiple) {
1084             ngx_md5_update(md5, (u_char *) ",", sizeof(",") - 1);
1085         }
1086
1087         ngx_md5_update(md5, start, len);
1088
1089         multiple = 1;
1090     }
1091 }
1092 }
1093
1094
1095 static ngx_int_t
1096 ngx_http_file_cache_reopen(ngx_http_request_t *r, ngx_http_cache_t *c)
1097 {
1098     ngx_http_file_cache_t *cache;
1099
1100     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, c->file.log, 0,
1101                  "http file cache reopen");
1102
1103     if (c->secondary) {
1104         ngx_log_error(NGX_LOG_CRIT, r->connection->log, 0,
1105                      "cache file \"%s\" has incorrect vary hash",

```

```

1106         c->file.name.data);
1107     return NGX\_DECLINED;
1108 }
1109
1110 cache = c->file_cache;
1111
1112 ngx\_shmtx\_lock(&cache->shpool->mutex);
1113
1114 c->node->count--;
1115 c->node = NULL;
1116
1117 ngx\_shmtx\_unlock(&cache->shpool->mutex);
1118
1119 c->secondary = 1;
1120 c->file.name.len = 0;
1121 c->body_start = c->buf->end - c->buf->start;
1122
1123 ngx\_memcpy(c->key, c->variant, NGX\_HTTP\_CACHE\_KEY\_LEN);
1124
1125 return ngx\_http\_file\_cache\_open(r);
1126 }
1127
1128
1129 ngx\_int\_t
1130 ngx\_http\_file\_cache\_set\_header(ngx\_http\_request\_t *r, u_char *buf)
1131 {
1132     ngx\_http\_file\_cache\_header\_t *h = (ngx\_http\_file\_cache\_header\_t *) buf;
1133
1134     u_char          *p;
1135     ngx\_str\_t       *key;
1136     ngx\_uint\_t      i;
1137     ngx\_http\_cache\_t *c;
1138
1139     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
1140                 "http file cache set header");
1141
1142     c = r->cache;
1143
1144     ngx\_memzero(h, sizeof(ngx\_http\_file\_cache\_header\_t));
1145
1146     h->version = NGX\_HTTP\_CACHE\_VERSION;
1147     h->valid_sec = c->valid_sec;
1148     h->last_modified = c->last_modified;
1149     h->date = c->date;
1150     h->crc32 = c->crc32;
1151     h->valid_msec = (u_short) c->valid_msec;
1152     h->header_start = (u_short) c->header_start;
1153     h->body_start = (u_short) c->body_start;
1154
1155     if (c->etag.len <= NGX\_HTTP\_CACHE\_ETAG\_LEN) {
1156         h->etag_len = (u_char) c->etag.len;
1157         ngx\_memcpy(h->etag, c->etag.data, c->etag.len);
1158     }
1159
1160     if (c->vary.len) {
1161         if (c->vary.len > NGX\_HTTP\_CACHE\_VARY\_LEN) {
1162             /* should not happen */
1163             c->vary.len = NGX\_HTTP\_CACHE\_VARY\_LEN;
1164         }
1165
1166         h->vary_len = (u_char) c->vary.len;
1167         ngx\_memcpy(h->vary, c->vary.data, c->vary.len);
1168
1169         ngx\_http\_file\_cache\_vary(r, c->vary.data, c->vary.len, c->variant);
1170         ngx\_memcpy(h->variant, c->variant, NGX\_HTTP\_CACHE\_KEY\_LEN);
1171     }
1172
1173     if (ngx\_http\_file\_cache\_update\_variant(r, c) != NGX\_OK) {
1174         return NGX\_ERROR;
1175     }
1176
1177     p = buf + sizeof(ngx\_http\_file\_cache\_header\_t);
1178
1179     p = ngx\_cpymem(p, ngx\_http\_file\_cache\_key, sizeof(ngx\_http\_file\_cache\_key));
1180
1181     key = c->keys.elts;

```



```

1182     for (i = 0; i < c->keys.nelts; i++) {
1183         p = ngx_copy(p, key[i].data, key[i].len);
1184     }
1185
1186     *p = LF;
1187
1188     return NGX_OK;
1189 }
1190
1191
1192 static ngx_int_t
1193 ngx_http_file_cache_update_variant(ngx_http_request_t *r, ngx_http_cache_t *c)
1194 {
1195     ngx_http_file_cache_t *cache;
1196
1197     if (!c->secondary) {
1198         return NGX_OK;
1199     }
1200
1201     if (c->vary.len
1202         && ngx_memcmp(c->variant, c->key, NGX_HTTP_CACHE_KEY_LEN) == 0)
1203     {
1204         return NGX_OK;
1205     }
1206
1207     /*
1208      * if the variant hash doesn't match one we used as a secondary
1209      * cache key, switch back to the original key
1210      */
1211
1212     cache = c->file_cache;
1213
1214     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1215                  "http file cache main key");
1216
1217     ngx_shmtx_lock(&cache->shpool->mutex);
1218
1219     c->node->count--;
1220     c->node->updating = 0;
1221     c->node = NULL;
1222
1223     ngx_shmtx_unlock(&cache->shpool->mutex);
1224
1225     c->file.name.len = 0;
1226
1227     ngx_memcpy(c->key, c->main, NGX_HTTP_CACHE_KEY_LEN);
1228
1229     if (ngx_http_file_cache_exists(cache, c) == NGX_ERROR) {
1230         return NGX_ERROR;
1231     }
1232
1233     if (ngx_http_file_cache_name(r, cache->path) != NGX_OK) {
1234         return NGX_ERROR;
1235     }
1236
1237     return NGX_OK;
1238 }
1239
1240
1241 void
1242 ngx_http_file_cache_update(ngx_http_request_t *r, ngx_temp_file_t *tf)
1243 {
1244     off_t             fs_size;
1245     ngx_int_t         rc;
1246     ngx_file_uniq_t   uniq;
1247     ngx_file_info_t   fi;
1248     ngx_http_cache_t *c;
1249     ngx_ext_rename_file_t  ext;
1250     ngx_http_file_cache_t *cache;
1251
1252     c = r->cache;
1253
1254     if (c->updated) {
1255         return;
1256     }
1257

```

```

1258 ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
1259     "http file cache update");
1260
1261 cache = c->file_cache;
1262
1263 c->updated = 1;
1264 c->updating = 0;
1265
1266 uniq = 0;
1267 fs_size = 0;
1268
1269 ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
1270     "http file cache rename: \"%s\" to \"%s\"",
1271     tf->file.name.data, c->file.name.data);
1272
1273 ext.access = NGX\_FILE\_OWNER\_ACCESS;
1274 ext.path_access = NGX\_FILE\_OWNER\_ACCESS;
1275 ext.time = -1;
1276 ext.create_path = 1;
1277 ext.delete_file = 1;
1278 ext.log = r->connection->log;
1279
1280 rc = ngx\_ext\_rename\_file(&tf->file.name, &c->file.name, &ext);
1281
1282 if (rc == NGX\_OK) {
1283     if (ngx\_fd\_info(tf->file.fd, &fi) == NGX\_FILE\_ERROR) {
1284         ngx\_log\_error(NGX\_LOG\_CRIT, r->connection->log, ngx\_errno,
1285             ngx\_fd\_info\_n " \"%s\" failed", tf->file.name.data);
1286
1287         rc = NGX\_ERROR;
1288     } else {
1289         uniq = ngx\_file\_uniq(&fi);
1290         fs_size = (ngx\_file\_fs\_size(&fi) + cache->bsize - 1) / cache->bsize;
1291     }
1292 }
1293
1294 ngx\_shmtx\_lock(&cache->shpool->mutex);
1295
1296 c->node->count--;
1297 c->node->uniq = uniq;
1298 c->node->body_start = c->body_start;
1299
1300 cache->sh->size += fs_size - c->node->fs_size;
1301 c->node->fs_size = fs_size;
1302
1303 if (rc == NGX\_OK) {
1304     c->node->exists = 1;
1305 }
1306
1307 c->node->updating = 0;
1308
1309 ngx\_shmtx\_unlock(&cache->shpool->mutex);
1310 }
1311
1312 void
1313 ngx\_http\_file\_cache\_update\_header(ngx\_http\_request\_t *r)
1314 {
1315     ssize_t          n;
1316     ngx\_err\_t      err;
1317     ngx\_file\_t      file;
1318     ngx\_file\_info\_t fi;
1319     ngx\_http\_cache\_t *c;
1320     ngx\_http\_file\_cache\_header\_t h;
1321
1322     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
1323         "http file cache update header");
1324
1325     c = r->cache;
1326
1327     ngx\_memzero(&file, sizeof(ngx\_file\_t));
1328
1329     file.name = c->file.name;
1330     file.log = r->connection->log;

```

```

1334 file.fd = ngx_open_file(file.name.data, NGX_FILE_RDWR, NGX_FILE_OPEN, 0);
1335
1336 if (file.fd == NGX_INVALID_FILE) {
1337     err = ngx_errno;
1338
1339     /* cache file may have been deleted */
1340
1341     if (err == NGX_ENOENT) {
1342         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1343             "http file cache \"%s\" not found",
1344             file.name.data);
1345         return;
1346     }
1347
1348     ngx_log_error(NGX_LOG_CRIT, r->connection->log, err,
1349         ngx_open_file_n " \"%s\" failed", file.name.data);
1350     return;
1351 }
1352
1353 /*
1354  * make sure cache file wasn't replaced;
1355  * if it was, do nothing
1356  */
1357
1358 if (ngx_fd_info(file.fd, &fi) == NGX_FILE_ERROR) {
1359     ngx_log_error(NGX_LOG_CRIT, r->connection->log, ngx_errno,
1360         ngx_fd_info_n " \"%s\" failed", file.name.data);
1361     goto done;
1362 }
1363
1364 if (c->uniq != ngx_file_uniq(&fi)
1365     || c->length != ngx_file_size(&fi))
1366 {
1367     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1368         "http file cache \"%s\" changed",
1369         file.name.data);
1370     goto done;
1371 }
1372
1373 n = ngx_read_file(&file, (u_char *) &h,
1374     sizeof(ngx_http_file_cache_header_t), 0);
1375
1376 if (n == NGX_ERROR) {
1377     goto done;
1378 }
1379
1380 if ((size_t) n != sizeof(ngx_http_file_cache_header_t)) {
1381     ngx_log_error(NGX_LOG_CRIT, r->connection->log, 0,
1382         ngx_read_file_n " read only %z of %z from \"%s\"",
1383         n, sizeof(ngx_http_file_cache_header_t), file.name.data);
1384     goto done;
1385 }
1386
1387 if (h.version != NGX_HTTP_CACHE_VERSION
1388     || h.last_modified != c->last_modified
1389     || h.crc32 != c->crc32
1390     || h.header_start != c->header_start
1391     || h.body_start != c->body_start)
1392 {
1393     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1394         "http file cache \"%s\" content changed",
1395         file.name.data);
1396     goto done;
1397 }
1398
1399 /*
1400  * update cache file header with new data,
1401  * notably h.valid_sec and h.date
1402  */
1403
1404 ngx_memzero(&h, sizeof(ngx_http_file_cache_header_t));
1405
1406 h.version = NGX_HTTP_CACHE_VERSION;
1407 h.valid_sec = c->valid_sec;
1408 h.last_modified = c->last_modified;
1409 h.date = c->date;

```

```

1410     h.crc32 = c->crc32;
1411     h.valid_msec = (u_short) c->valid_msec;
1412     h.header_start = (u_short) c->header_start;
1413     h.body_start = (u_short) c->body_start;
1414
1415     if (c->etag.len <= NGX\_HTTP\_CACHE\_ETAG\_LEN) {
1416         h.etag_len = (u_char) c->etag.len;
1417         ngx\_memcpy(h.etag, c->etag.data, c->etag.len);
1418     }
1419
1420     if (c->vary.len) {
1421         if (c->vary.len > NGX\_HTTP\_CACHE\_VARY\_LEN) {
1422             /* should not happen */
1423             c->vary.len = NGX\_HTTP\_CACHE\_VARY\_LEN;
1424         }
1425
1426         h.vary_len = (u_char) c->vary.len;
1427         ngx\_memcpy(h.vary, c->vary.data, c->vary.len);
1428
1429         ngx\_http\_file\_cache\_vary(r, c->vary.data, c->vary.len, c->variant);
1430         ngx\_memcpy(h.variant, c->variant, NGX\_HTTP\_CACHE\_KEY\_LEN);
1431     }
1432
1433     (void) ngx\_write\_file(&file, (u_char *) &h,
1434                          sizeof(ngx\_http\_file\_cache\_header\_t), 0);
1435
1436 done:
1437
1438     if (ngx\_close\_file(file.fd) == NGX\_FILE\_ERROR) {
1439         ngx\_log\_error(NGX\_LOG\_ALERT, r->connection->log, ngx\_errno,
1440                    ngx\_close\_file\_n " \"%s\" failed", file.name.data);
1441     }
1442 }
1443
1444
1445 ngx\_int\_t
1446 ngx\_http\_cache\_send(ngx\_http\_request\_t *r)
1447 {
1448     ngx\_int\_t         rc;
1449     ngx\_buf\_t         *b;
1450     ngx\_chain\_t       out;
1451     ngx\_http\_cache\_t *c;
1452
1453     c = r->cache;
1454
1455     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
1456                  "http file cache send: %s", c->file.name.data);
1457
1458     if (r != r->main && c->length - c->body_start == 0) {
1459         return ngx\_http\_send\_header(r);
1460     }
1461
1462     /* we need to allocate all before the header would be sent */
1463
1464     b = ngx\_palloc(r->pool, sizeof(ngx\_buf\_t));
1465     if (b == NULL) {
1466         return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
1467     }
1468
1469     b->file = ngx\_palloc(r->pool, sizeof(ngx\_file\_t));
1470     if (b->file == NULL) {
1471         return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
1472     }
1473
1474     rc = ngx\_http\_send\_header(r);
1475
1476     if (rc == NGX\_ERROR || rc > NGX\_OK || r->header_only) {
1477         return rc;
1478     }
1479
1480     b->file_pos = c->body_start;
1481     b->file_last = c->length;
1482
1483     b->in_file = (c->length - c->body_start) ? 1: 0;
1484     b->last_buf = (r == r->main) ? 1: 0;
1485     b->last_in_chain = 1;

```

```

1486     b->file->fd = c->file.fd;
1487     b->file->name = c->file.name;
1488     b->file->log = r->connection->log;
1489
1490
1491     out.buf = b;
1492     out.next = NULL;
1493
1494     return ngx_http_output_filter(r, &out);
1495 }
1496
1497 void
1498 ngx_http_file_cache_free(ngx_http_cache_t *c, ngx_temp_file_t *tf)
1499 {
1500     ngx_http_file_cache_t      *cache;
1501     ngx_http_file_cache_node_t *fcn;
1502
1503     if (c->updated || c->node == NULL) {
1504         return;
1505     }
1506
1507     cache = c->file_cache;
1508
1509     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, c->file.log, 0,
1510         "http file cache free, fd: %d", c->file.fd);
1511
1512     ngx_shmtx_lock(&cache->shpool->mutex);
1513
1514     fcn = c->node;
1515     fcn->count--;
1516
1517     if (c->updating && fcn->lock_time == c->lock_time) {
1518         fcn->updating = 0;
1519     }
1520
1521     if (c->error) {
1522         fcn->error = c->error;
1523
1524         if (c->valid_sec) {
1525             fcn->valid_sec = c->valid_sec;
1526             fcn->valid_msec = c->valid_msec;
1527         }
1528     }
1529
1530     } else if (!fcn->exists && fcn->count == 0 && c->min_uses == 1) {
1531         ngx_queue_remove(&fcn->queue);
1532         ngx_rbtrees_delete(&cache->sh->rbtree, &fcn->node);
1533         ngx_slab_free_locked(cache->shpool, fcn);
1534         c->node = NULL;
1535     }
1536
1537     ngx_shmtx_unlock(&cache->shpool->mutex);
1538
1539     c->updated = 1;
1540     c->updating = 0;
1541
1542     if (c->temp_file) {
1543         if (tf && tf->file.fd != NGX_INVALID_FILE) {
1544             ngx_log_debug1(NGX_LOG_DEBUG_HTTP, c->file.log, 0,
1545                 "http file cache incomplete: \"%s\"",
1546                 tf->file.name.data);
1547
1548             if (ngx_delete_file(tf->file.name.data) == NGX_FILE_ERROR) {
1549                 ngx_log_error(NGX_LOG_CRIT, c->file.log, ngx_errno,
1550                     ngx_delete_file_n " \"%s\" failed",
1551                     tf->file.name.data);
1552             }
1553         }
1554     }
1555
1556     if (c->wait_event.timer_set) {
1557         ngx_del_timer(&c->wait_event);
1558     }
1559 }
1560
1561

```

```

1562 static void
1563 ngx_http_file_cache_cleanup(void *data)
1564 {
1565     ngx_http_cache_t *c = data;
1566
1567     if (c->updated) {
1568         return;
1569     }
1570
1571     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, c->file.log, 0,
1572                  "http file cache cleanup");
1573
1574     if (c->updating) {
1575         ngx_log_error(NGX_LOG_ALERT, c->file.log, 0,
1576                      "stalled cache updating, error:%ui", c->error);
1577     }
1578
1579     ngx_http_file_cache_free(c, NULL);
1580 }
1581
1582
1583 static time_t
1584 ngx_http_file_cache_forced_expire(ngx_http_file_cache_t *cache)
1585 {
1586     u_char          *name;
1587     size_t          len;
1588     time_t          wait;
1589     ngx_uint_t      tries;
1590     ngx_path_t      *path;
1591     ngx_queue_t     *q;
1592     ngx_http_file_cache_node_t *fcn;
1593
1594     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, ngx_cycle->log, 0,
1595                  "http file cache forced expire");
1596
1597     path = cache->path;
1598     len = path->name.len + 1 + path->len + 2 * NGX_HTTP_CACHE_KEY_LEN;
1599
1600     name = ngx_alloc(len + 1, ngx_cycle->log);
1601     if (name == NULL) {
1602         return 10;
1603     }
1604
1605     ngx_memcpy(name, path->name.data, path->name.len);
1606
1607     wait = 10;
1608     tries = 20;
1609
1610     ngx_shmtx_lock(&cache->shpool->mutex);
1611
1612     for (q = ngx_queue_last(&cache->sh->queue);
1613          q != ngx_queue_sentinel(&cache->sh->queue);
1614          q = ngx_queue_prev(q))
1615     {
1616         fcn = ngx_queue_data(q, ngx_http_file_cache_node_t, queue);
1617
1618         ngx_log_debug6(NGX_LOG_DEBUG_HTTP, ngx_cycle->log, 0,
1619                      "http file cache forced expire: #%d %d %02xd%02xd%02xd%02xd",
1620                      fcn->count, fcn->exists,
1621                      fcn->key[0], fcn->key[1], fcn->key[2], fcn->key[3]);
1622
1623         if (fcn->count == 0) {
1624             ngx_http_file_cache_delete(cache, q, name);
1625             wait = 0;
1626         } else {
1627             if (--tries) {
1628                 continue;
1629             }
1630
1631             wait = 1;
1632         }
1633     }
1634
1635     break;
1636 }
1637

```

```

1638     ngx_shmtx_unlock(&cache->shpool->mutex);
1639
1640     ngx_free(name);
1641
1642     return wait;
1643 }
1644
1645
1646 static time_t
1647 ngx_http_file_cache_expire(ngx_http_file_cache_t *cache)
1648 {
1649     u_char                *name, *p;
1650     size_t                len;
1651     time_t                now, wait;
1652     ngx_path_t            *path;
1653     ngx_queue_t           *q;
1654     ngx_http_file_cache_node_t *fcn;
1655     u_char                key[2 * NGX_HTTP_CACHE_KEY_LEN];
1656
1657     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, ngx_cycle->log, 0,
1658                  "http file cache expire");
1659
1660     path = cache->path;
1661     len = path->name.len + 1 + path->len + 2 * NGX_HTTP_CACHE_KEY_LEN;
1662
1663     name = ngx_alloc(len + 1, ngx_cycle->log);
1664     if (name == NULL) {
1665         return 10;
1666     }
1667
1668     ngx_memcpy(name, path->name.data, path->name.len);
1669
1670     now = ngx_time();
1671
1672     ngx_shmtx_lock(&cache->shpool->mutex);
1673
1674     for ( ;; ) {
1675
1676         if (ngx_quit || ngx_terminate) {
1677             wait = 1;
1678             break;
1679         }
1680
1681         if (ngx_queue_empty(&cache->sh->queue)) {
1682             wait = 10;
1683             break;
1684         }
1685
1686         q = ngx_queue_last(&cache->sh->queue);
1687
1688         fcn = ngx_queue_data(q, ngx_http_file_cache_node_t, queue);
1689
1690         wait = fcn->expire - now;
1691
1692         if (wait > 0) {
1693             wait = wait > 10 ? 10 : wait;
1694             break;
1695         }
1696
1697         ngx_log_debug6(NGX_LOG_DEBUG_HTTP, ngx_cycle->log, 0,
1698                      "http file cache expire: #%d %d %02xd%02xd%02xd%02xd",
1699                      fcn->count, fcn->exists,
1700                      fcn->key[0], fcn->key[1], fcn->key[2], fcn->key[3]);
1701
1702         if (fcn->count == 0) {
1703             ngx_http_file_cache_delete(cache, q, name);
1704             continue;
1705         }
1706
1707         if (fcn->deleting) {
1708             wait = 1;
1709             break;
1710         }
1711
1712         p = ngx_hex_dump(key, (u_char *) &fcn->node.key,
1713                        sizeof(ngx_rbtree_key_t));

```

```

1714     len = NGX\_HTTP\_CACHE\_KEY\_LEN - sizeof(ngx\_rbtree\_key\_t);
1715     (void) ngx\_hex\_dump(p, fcn->key, len);
1716
1717     /*
1718     * abnormally exited workers may leave locked cache entries,
1719     * and although it may be safe to remove them completely,
1720     * we prefer to just move them to the top of the inactive queue
1721     */
1722
1723     ngx\_queue\_remove(q);
1724     fcn->expire = ngx\_time() + cache->inactive;
1725     ngx\_queue\_insert\_head(&cache->sh->queue, &fcn->queue);
1726
1727     ngx\_log\_error(NGX\_LOG\_ALERT, ngx\_cycle->log, 0,
1728                 "ignore long locked inactive cache entry %*s, count:%d",
1729                 2 * NGX\_HTTP\_CACHE\_KEY\_LEN, key, fcn->count);
1730 }
1731
1732 ngx\_shmtx\_unlock(&cache->shpool->mutex);
1733
1734 ngx\_free(name);
1735
1736 return wait;
1737 }
1738
1739
1740 static void
1741 ngx\_http\_file\_cache\_delete(ngx\_http\_file\_cache\_t *cache, ngx\_queue\_t *q,
1742     u_char *name)
1743 {
1744     u_char                *p;
1745     size_t                len;
1746     ngx\_path\_t            *path;
1747     ngx\_http\_file\_cache\_node\_t *fcn;
1748
1749     fcn = ngx\_queue\_data(q, ngx\_http\_file\_cache\_node\_t, queue);
1750
1751     if (fcn->exists) {
1752         cache->sh->size -= fcn->fs_size;
1753
1754         path = cache->path;
1755         p = name + path->name.len + 1 + path->len;
1756         p = ngx\_hex\_dump(p, (u_char *) &fcn->node.key,
1757                         sizeof(ngx\_rbtree\_key\_t));
1758         len = NGX\_HTTP\_CACHE\_KEY\_LEN - sizeof(ngx\_rbtree\_key\_t);
1759         p = ngx\_hex\_dump(p, fcn->key, len);
1760         *p = '\0';
1761
1762         fcn->count++;
1763         fcn->deleting = 1;
1764         ngx\_shmtx\_unlock(&cache->shpool->mutex);
1765
1766         len = path->name.len + 1 + path->len + 2 * NGX\_HTTP\_CACHE\_KEY\_LEN;
1767         ngx\_create\_hashed\_filename(path, name, len);
1768
1769         ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, ngx\_cycle->log, 0,
1770                     "http file cache expire: \"%s\"", name);
1771
1772         if (ngx\_delete\_file(name) == NGX\_FILE\_ERROR) {
1773             ngx\_log\_error(NGX\_LOG\_CRIT, ngx\_cycle->log, ngx\_errno,
1774                         ngx\_delete\_file\_n " \"%s\" failed", name);
1775         }
1776
1777         ngx\_shmtx\_lock(&cache->shpool->mutex);
1778         fcn->count--;
1779         fcn->deleting = 0;
1780     }
1781
1782     if (fcn->count == 0) {
1783         ngx\_queue\_remove(q);
1784         ngx\_rbtree\_delete(&cache->sh->rbtree, &fcn->node);
1785         ngx\_slab\_free\_locked(cache->shpool, fcn);
1786     }
1787 }
1788
1789

```



```

1790 static time_t
1791 ngx_http_file_cache_manager(void *data)
1792 {
1793     ngx_http_file_cache_t *cache = data;
1794
1795     off_t size;
1796     time_t next, wait;
1797
1798     next = ngx_http_file_cache_expire(cache);
1799
1800     cache->last = ngx_current_msec;
1801     cache->files = 0;
1802
1803     for ( ;; ) {
1804         ngx_shmtx_lock(&cache->shpool->mutex);
1805
1806         size = cache->sh->size;
1807
1808         ngx_shmtx_unlock(&cache->shpool->mutex);
1809
1810         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, ngx_cycle->log, 0,
1811             "http file cache size: %0", size);
1812
1813         if (size < cache->max_size) {
1814             return next;
1815         }
1816
1817         wait = ngx_http_file_cache_forced_expire(cache);
1818
1819         if (wait > 0) {
1820             return wait;
1821         }
1822
1823         if (ngx_quit || ngx_terminate) {
1824             return next;
1825         }
1826     }
1827 }
1828
1829
1830 static void
1831 ngx_http_file_cache_loader(void *data)
1832 {
1833     ngx_http_file_cache_t *cache = data;
1834
1835     ngx_tree_ctx_t tree;
1836
1837     if (!cache->sh->cold || cache->sh->loading) {
1838         return;
1839     }
1840
1841     if (!ngx_atomic_cmp_set(&cache->sh->loading, 0, ngx_pid)) {
1842         return;
1843     }
1844
1845     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, ngx_cycle->log, 0,
1846         "http file cache loader");
1847
1848     tree.init_handler = NULL;
1849     tree.file_handler = ngx_http_file_cache_manage_file;
1850     tree.pre_tree_handler = ngx_http_file_cache_manage_directory;
1851     tree.post_tree_handler = ngx_http_file_cache_noop;
1852     tree.spec_handler = ngx_http_file_cache_delete_file;
1853     tree.data = cache;
1854     tree.alloc = 0;
1855     tree.log = ngx_cycle->log;
1856
1857     cache->last = ngx_current_msec;
1858     cache->files = 0;
1859
1860     if (ngx_walk_tree(&tree, &cache->path->name) == NGX_ABORT) {
1861         cache->sh->loading = 0;
1862         return;
1863     }
1864
1865     cache->sh->cold = 0;

```

```

1866     cache->sh->loading = 0;
1867
1868     ngx_log_error(NGX_LOG_NOTICE, ngx_cycle->log, 0,
1869         "http file cache: %V %.3fM, bsize: %uz",
1870         &cache->path->name,
1871         ((double) cache->sh->size * cache->bsize) / (1024 * 1024),
1872         cache->bsize);
1873 }
1874
1875
1876 static ngx_int_t
1877 ngx_http_file_cache_noop(ngx_tree_ctx_t *ctx, ngx_str_t *path)
1878 {
1879     return NGX_OK;
1880 }
1881
1882
1883 static ngx_int_t
1884 ngx_http_file_cache_manage_file(ngx_tree_ctx_t *ctx, ngx_str_t *path)
1885 {
1886     ngx_msec_t      elapsed;
1887     ngx_http_file_cache_t *cache;
1888
1889     cache = ctx->data;
1890
1891     if (ngx_http_file_cache_add_file(ctx, path) != NGX_OK) {
1892         (void) ngx_http_file_cache_delete_file(ctx, path);
1893     }
1894
1895     if (++cache->files >= cache->loader_files) {
1896         ngx_http_file_cache_loader_sleep(cache);
1897     } else {
1898         ngx_time_update();
1899
1900         elapsed = ngx_abs((ngx_msec_int_t) (ngx_current_msec - cache->last));
1901
1902         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, ngx_cycle->log, 0,
1903             "http file cache loader time elapsed: %M", elapsed);
1904
1905         if (elapsed >= cache->loader_threshold) {
1906             ngx_http_file_cache_loader_sleep(cache);
1907         }
1908     }
1909
1910     return (ngx_quit || ngx_terminate) ? NGX_ABORT : NGX_OK;
1911 }
1912
1913
1914
1915 static ngx_int_t
1916 ngx_http_file_cache_manage_directory(ngx_tree_ctx_t *ctx, ngx_str_t *path)
1917 {
1918     if (path->len >= 5
1919         && ngx_strncmp(path->data + path->len - 5, "/temp", 5) == 0)
1920     {
1921         return NGX_DECLINED;
1922     }
1923
1924     return NGX_OK;
1925 }
1926
1927
1928 static void
1929 ngx_http_file_cache_loader_sleep(ngx_http_file_cache_t *cache)
1930 {
1931     ngx_msleep(cache->loader_sleep);
1932
1933     ngx_time_update();
1934
1935     cache->last = ngx_current_msec;
1936     cache->files = 0;
1937 }
1938
1939
1940 static ngx_int_t
1941 ngx_http_file_cache_add_file(ngx_tree_ctx_t *ctx, ngx_str_t *name)

```

```

1942 {
1943     u_char                *p;
1944     ngx_int_t            n;
1945     ngx_uint_t           i;
1946     ngx_http_cache_t     c;
1947     ngx_http_file_cache_t *cache;
1948
1949     if (name->len < 2 * NGX_HTTP_CACHE_KEY_LEN) {
1950         return NGX_ERROR;
1951     }
1952
1953     if (ctx->size < (off_t) sizeof(ngx_http_file_cache_header_t)) {
1954         ngx_log_error(NGX_LOG_CRIT, ctx->log, 0,
1955             "cache file \"%s\" is too small", name->data);
1956         return NGX_ERROR;
1957     }
1958
1959     ngx_memzero(&c, sizeof(ngx_http_cache_t));
1960     cache = ctx->data;
1961
1962     c.length = ctx->size;
1963     c.fs_size = (ctx->fs_size + cache->bsize - 1) / cache->bsize;
1964
1965     p = &name->data[name->len - 2 * NGX_HTTP_CACHE_KEY_LEN];
1966
1967     for (i = 0; i < NGX_HTTP_CACHE_KEY_LEN; i++) {
1968         n = ngx_hextoi(p, 2);
1969
1970         if (n == NGX_ERROR) {
1971             return NGX_ERROR;
1972         }
1973
1974         p += 2;
1975
1976         c.key[i] = (u_char) n;
1977     }
1978
1979     return ngx_http_file_cache_add(cache, &c);
1980 }
1981
1982
1983 static ngx_int_t
1984 ngx_http_file_cache_add(ngx_http_file_cache_t *cache, ngx_http_cache_t *c)
1985 {
1986     ngx_http_file_cache_node_t *fcn;
1987
1988     ngx_shmtx_lock(&cache->shpool->mutex);
1989
1990     fcn = ngx_http_file_cache_lookup(cache, c->key);
1991
1992     if (fcn == NULL) {
1993
1994         fcn = ngx_slab_alloc_locked(cache->shpool,
1995             sizeof(ngx_http_file_cache_node_t));
1996         if (fcn == NULL) {
1997             ngx_shmtx_unlock(&cache->shpool->mutex);
1998             return NGX_ERROR;
1999         }
2000
2001         ngx_memcpy((u_char *) &fcn->node.key, c->key, sizeof(ngx_rbtree_key_t));
2002
2003         ngx_memcpy(fcn->key, &c->key[sizeof(ngx_rbtree_key_t)],
2004             NGX_HTTP_CACHE_KEY_LEN - sizeof(ngx_rbtree_key_t));
2005
2006         ngx_rbtree_insert(&cache->sh->rbtree, &fcn->node);
2007
2008         fcn->uses = 1;
2009         fcn->exists = 1;
2010         fcn->fs_size = c->fs_size;
2011
2012         cache->sh->size += c->fs_size;
2013
2014     } else {
2015         ngx_queue_remove(&fcn->queue);
2016     }
2017

```

```

2018     fcn->expire = ngx_time() + cache->inactive;
2019
2020     ngx_queue_insert_head(&cache->sh->queue, &fcn->queue);
2021
2022     ngx_shmtx_unlock(&cache->shpool->mutex);
2023
2024     return NGX_OK;
2025 }
2026
2027
2028 static ngx_int_t
2029 ngx_http_file_cache_delete_file(ngx_tree_ctx_t *ctx, ngx_str_t *path)
2030 {
2031     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, ctx->log, 0,
2032                  "http file cache delete: \"%s\"", path->data);
2033
2034     if (ngx_delete_file(path->data) == NGX_FILE_ERROR) {
2035         ngx_log_error(NGX_LOG_CRIT, ctx->log, ngx_errno,
2036                      ngx_delete_file_n " \"%s\" failed", path->data);
2037     }
2038
2039     return NGX_OK;
2040 }
2041
2042
2043 time_t
2044 ngx_http_file_cache_valid(ngx_array_t *cache_valid, ngx_uint_t status)
2045 {
2046     ngx_uint_t      i;
2047     ngx_http_cache_valid_t *valid;
2048
2049     if (cache_valid == NULL) {
2050         return 0;
2051     }
2052
2053     valid = cache_valid->elts;
2054     for (i = 0; i < cache_valid->nelts; i++) {
2055
2056         if (valid[i].status == 0) {
2057             return valid[i].valid;
2058         }
2059
2060         if (valid[i].status == status) {
2061             return valid[i].valid;
2062         }
2063     }
2064
2065     return 0;
2066 }
2067
2068
2069 char *
2070 ngx_http_file_cache_set_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
2071 {
2072     char *confp = conf;
2073
2074     off_t      max_size;
2075     u_char     *last, *p;
2076     time_t     inactive;
2077     size_t     len;
2078     ssize_t    size;
2079     ngx_str_t  s, name, *value;
2080     ngx_int_t  loader_files;
2081     ngx_msec_t loader_sleep, loader_threshold;
2082     ngx_uint_t i, n, use_temp_path;
2083     ngx_array_t *caches;
2084     ngx_http_file_cache_t *cache, **ce;
2085
2086     cache = ngx_palloc(cf->pool, sizeof(ngx_http_file_cache_t));
2087     if (cache == NULL) {
2088         return NGX_CONF_ERROR;
2089     }
2090
2091     cache->path = ngx_palloc(cf->pool, sizeof(ngx_path_t));
2092     if (cache->path == NULL) {
2093         return NGX_CONF_ERROR;

```

```

2094     }
2095
2096     use_temp_path = 1;
2097
2098     inactive = 600;
2099     loader_files = 100;
2100     loader_sleep = 50;
2101     loader_threshold = 200;
2102
2103     name.len = 0;
2104     size = 0;
2105     max_size = NGX_MAX_OFF_T_VALUE;
2106
2107     value = cf->args->elts;
2108
2109     cache->path->name = value[1];
2110
2111     if (cache->path->name.data[cache->path->name.len - 1] == '/') {
2112         cache->path->name.len--;
2113     }
2114
2115     if (ngx_conf_full_name(cf->cycle, &cache->path->name, 0) != NGX_OK) {
2116         return NGX_CONF_ERROR;
2117     }
2118
2119     for (i = 2; i < cf->args->nelts; i++) {
2120
2121         if (ngx_strncmp(value[i].data, "levels=", 7) == 0) {
2122
2123             p = value[i].data + 7;
2124             last = value[i].data + value[i].len;
2125
2126             for (n = 0; n < 3 && p < last; n++) {
2127
2128                 if (*p > '0' && *p < '3') {
2129
2130                     cache->path->level[n] = *p++ - '0';
2131                     cache->path->len += cache->path->level[n] + 1;
2132
2133                     if (p == last) {
2134                         break;
2135                     }
2136
2137                     if (*p++ == ':' && n < 2 && p != last) {
2138                         continue;
2139                     }
2140
2141                     goto invalid_levels;
2142                 }
2143
2144                 goto invalid_levels;
2145             }
2146
2147             if (cache->path->len < 10 + 3) {
2148                 continue;
2149             }
2150
2151             invalid_levels:
2152
2153             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
2154                 "invalid \"levels\" \"%V\"", &value[i]);
2155             return NGX_CONF_ERROR;
2156         }
2157
2158         if (ngx_strncmp(value[i].data, "use_temp_path=", 14) == 0) {
2159
2160             if (ngx_strcmp(&value[i].data[14], "on") == 0) {
2161                 use_temp_path = 1;
2162
2163             } else if (ngx_strcmp(&value[i].data[14], "off") == 0) {
2164                 use_temp_path = 0;
2165
2166             } else {
2167                 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
2168                     "invalid use_temp_path value \"%V\", "
2169                     "it must be \"on\" or \"off\"",

```

```

2170         &value[i]);
2171     return NGX\_CONF\_ERROR;
2172 }
2173
2174     continue;
2175 }
2176
2177 if (ngx\_strncmp(value[i].data, "keys_zone=", 10) == 0) {
2178     name.data = value[i].data + 10;
2179
2180     p = (u_char *) ngx\_strchr(name.data, ':');
2181
2182     if (p) {
2183         name.len = p - name.data;
2184
2185         p++;
2186
2187         s.len = value[i].data + value[i].len - p;
2188         s.data = p;
2189
2190         size = ngx\_parse\_size(&s);
2191         if (size > 8191) {
2192             continue;
2193         }
2194     }
2195 }
2196
2197     ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
2198         "invalid keys zone size \"%V\"", &value[i]);
2199     return NGX\_CONF\_ERROR;
2200 }
2201
2202 if (ngx\_strncmp(value[i].data, "inactive=", 9) == 0) {
2203     s.len = value[i].len - 9;
2204     s.data = value[i].data + 9;
2205
2206     inactive = ngx\_parse\_time(&s, 1);
2207     if (inactive == (time_t) NGX\_ERROR) {
2208         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
2209             "invalid inactive value \"%V\"", &value[i]);
2210         return NGX\_CONF\_ERROR;
2211     }
2212 }
2213
2214     continue;
2215 }
2216
2217 if (ngx\_strncmp(value[i].data, "max_size=", 9) == 0) {
2218     s.len = value[i].len - 9;
2219     s.data = value[i].data + 9;
2220
2221     max_size = ngx\_parse\_offset(&s);
2222     if (max_size < 0) {
2223         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
2224             "invalid max_size value \"%V\"", &value[i]);
2225         return NGX\_CONF\_ERROR;
2226     }
2227 }
2228
2229     continue;
2230 }
2231
2232 if (ngx\_strncmp(value[i].data, "loader_files=", 13) == 0) {
2233     loader_files = ngx\_atoi(value[i].data + 13, value[i].len - 13);
2234     if (loader_files == NGX\_ERROR) {
2235         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
2236             "invalid loader_files value \"%V\"", &value[i]);
2237         return NGX\_CONF\_ERROR;
2238     }
2239 }
2240
2241     continue;
2242 }
2243
2244 if (ngx\_strncmp(value[i].data, "loader_sleep=", 13) == 0) {
2245

```

```

2246     s.len = value[i].len - 13;
2247     s.data = value[i].data + 13;
2248
2249     loader_sleep = ngx_parse_time(&s, 0);
2250     if (loader_sleep == (ngx_msec_t) NGX_ERROR) {
2251         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
2252             "invalid loader_sleep value \"%V\"", &value[i]);
2253         return NGX_CONF_ERROR;
2254     }
2255
2256     continue;
2257 }
2258
2259 if (ngx_strncmp(value[i].data, "loader_threshold=", 17) == 0) {
2260
2261     s.len = value[i].len - 17;
2262     s.data = value[i].data + 17;
2263
2264     loader_threshold = ngx_parse_time(&s, 0);
2265     if (loader_threshold == (ngx_msec_t) NGX_ERROR) {
2266         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
2267             "invalid loader_threshold value \"%V\"", &value[i]);
2268         return NGX_CONF_ERROR;
2269     }
2270
2271     continue;
2272 }
2273
2274 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
2275     "invalid parameter \"%V\"", &value[i]);
2276 return NGX_CONF_ERROR;
2277 }
2278
2279 if (name.len == 0 || size == 0) {
2280     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
2281         "\"%V\" must have \"keys_zone\" parameter",
2282         &cmd->name);
2283     return NGX_CONF_ERROR;
2284 }
2285
2286 cache->path->manager = ngx_http_file_cache_manager;
2287 cache->path->loader = ngx_http_file_cache_loader;
2288 cache->path->data = cache;
2289 cache->path->conf_file = cf->conf_file->file.name.data;
2290 cache->path->line = cf->conf_file->line;
2291 cache->loader_files = loader_files;
2292 cache->loader_sleep = loader_sleep;
2293 cache->loader_threshold = loader_threshold;
2294
2295 if (ngx_add_path(cf, &cache->path) != NGX_OK) {
2296     return NGX_CONF_ERROR;
2297 }
2298
2299 if (!use_temp_path) {
2300     cache->temp_path = ngx_palloc(cf->pool, sizeof(ngx_path_t));
2301     if (cache->temp_path == NULL) {
2302         return NGX_CONF_ERROR;
2303     }
2304
2305     len = cache->path->name.len + sizeof("/temp") - 1;
2306
2307     p = ngx_pnalloc(cf->pool, len + 1);
2308     if (p == NULL) {
2309         return NGX_CONF_ERROR;
2310     }
2311
2312     cache->temp_path->name.len = len;
2313     cache->temp_path->name.data = p;
2314
2315     p = ngx_cpymem(p, cache->path->name.data, cache->path->name.len);
2316     ngx_memcpy(p, "/temp", sizeof("/temp"));
2317
2318     ngx_memcpy(&cache->temp_path->level, &cache->path->level,
2319         3 * sizeof(size_t));
2320
2321     cache->temp_path->len = cache->path->len;

```

```

2322     cache->temp_path->conf_file = cf->conf_file->file.name.data;
2323     cache->temp_path->line = cf->conf_file->line;
2324
2325     if (ngx_add_path(cf, &cache->temp_path) != NGX_OK) {
2326         return NGX_CONF_ERROR;
2327     }
2328 }
2329
2330 cache->shm_zone = ngx_shared_memory_add(cf, &name, size, cmd->post);
2331 if (cache->shm_zone == NULL) {
2332     return NGX_CONF_ERROR;
2333 }
2334
2335 if (cache->shm_zone->data) {
2336     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
2337         "duplicate zone \"%V\"", &name);
2338     return NGX_CONF_ERROR;
2339 }
2340
2341
2342 cache->shm_zone->init = ngx_http_file_cache_init;
2343 cache->shm_zone->data = cache;
2344
2345 cache->inactive = inactive;
2346 cache->max_size = max_size;
2347
2348 caches = (ngx_array_t *) (confp + cmd->offset);
2349
2350 ce = ngx_array_push(caches);
2351 if (ce == NULL) {
2352     return NGX_CONF_ERROR;
2353 }
2354
2355 *ce = cache;
2356
2357 return NGX_CONF_OK;
2358 }
2359
2360
2361 char *
2362 ngx_http_file_cache_valid_set_slot(ngx_conf_t *cf, ngx_command_t *cmd,
2363 void *conf)
2364 {
2365     char *p = conf;
2366
2367     time_t          valid;
2368     ngx_str_t       *value;
2369     ngx_uint_t      i, n, status;
2370     ngx_array_t     **a;
2371     ngx_http_cache_valid_t *v;
2372     static ngx_uint_t statuses[] = { 200, 301, 302 };
2373
2374     a = (ngx_array_t **) (p + cmd->offset);
2375
2376     if (*a == NGX_CONF_UNSET_PTR) {
2377         *a = ngx_array_create(cf->pool, 1, sizeof(ngx_http_cache_valid_t));
2378         if (*a == NULL) {
2379             return NGX_CONF_ERROR;
2380         }
2381     }
2382
2383     value = cf->args->elts;
2384     n = cf->args->nelts - 1;
2385
2386     valid = ngx_parse_time(&value[n], 1);
2387     if (valid == (time_t) NGX_ERROR) {
2388         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
2389             "invalid time value \"%V\"", &value[n]);
2390         return NGX_CONF_ERROR;
2391     }
2392
2393     if (n == 1) {
2394
2395         for (i = 0; i < 3; i++) {
2396             v = ngx_array_push(*a);
2397             if (v == NULL) {

```



```
2398         return NGX\_CONF\_ERROR;
2399     }
2400
2401     v->status = statuses[i];
2402     v->valid = valid;
2403 }
2404
2405 return NGX\_CONF\_OK;
2406 }
2407
2408 for (i = 1; i < n; i++) {
2409     if (ngx\_strcmp(value[i].data, "any") == 0) {
2410         status = 0;
2411     } else {
2412         status = ngx\_atoi(value[i].data, value[i].len);
2413         if (status < 100) {
2414             ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
2415                 "invalid status \"%V\"", &value[i]);
2416             return NGX\_CONF\_ERROR;
2417         }
2418     }
2419
2420     v = ngx\_array\_push(*a);
2421     if (v == NULL) {
2422         return NGX\_CONF\_ERROR;
2423     }
2424
2425     v->status = status;
2426     v->valid = valid;
2427 }
2428
2429 return NGX\_CONF\_OK;
2430 }
2431
2432 return NGX\_CONF\_OK;
2433 }
2434 }
```

[One Level Up](#)

[Top Level](#)

# src/http/nginx\_http\_cache.h - nginx-1.7.10

## Data types defined

- [ngx\\_http\\_cache\\_s](#)
- [ngx\\_http\\_cache\\_valid\\_t](#)
- [ngx\\_http\\_file\\_cache\\_header\\_t](#)
- [ngx\\_http\\_file\\_cache\\_node\\_t](#)
- [ngx\\_http\\_file\\_cache\\_s](#)
- [ngx\\_http\\_file\\_cache\\_sh\\_t](#)

## Macros defined

- [NGX\\_HTTP\\_CACHE\\_BYPASS](#)
- [NGX\\_HTTP\\_CACHE\\_ETAG\\_LEN](#)
- [NGX\\_HTTP\\_CACHE\\_EXPIRED](#)
- [NGX\\_HTTP\\_CACHE\\_HIT](#)
- [NGX\\_HTTP\\_CACHE\\_KEY\\_LEN](#)
- [NGX\\_HTTP\\_CACHE\\_MISS](#)
- [NGX\\_HTTP\\_CACHE\\_REVALIDATED](#)
- [NGX\\_HTTP\\_CACHE\\_SCARCE](#)
- [NGX\\_HTTP\\_CACHE\\_STALE](#)
- [NGX\\_HTTP\\_CACHE\\_UPDATING](#)
- [NGX\\_HTTP\\_CACHE\\_VARY\\_LEN](#)
- [NGX\\_HTTP\\_CACHE\\_VERSION](#)
- [\\_NGX\\_HTTP\\_CACHE\\_H\\_INCLUDED](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_HTTP\_CACHE\_H\_INCLUDED
9 #define \_NGX\_HTTP\_CACHE\_H\_INCLUDED
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14 #include <ngx_http.h>
15
16
17 #define NGX\_HTTP\_CACHE\_MISS 1
18 #define NGX\_HTTP\_CACHE\_BYPASS 2
```

```

19 #define NGX_HTTP_CACHE_EXPIRED 3
20 #define NGX_HTTP_CACHE_STALE 4
21 #define NGX_HTTP_CACHE_UPDATING 5
22 #define NGX_HTTP_CACHE_REVALIDATED 6
23 #define NGX_HTTP_CACHE_HIT 7
24 #define NGX_HTTP_CACHE_SCARCE 8
25
26 #define NGX_HTTP_CACHE_KEY_LEN 16
27 #define NGX_HTTP_CACHE_ETAG_LEN 42
28 #define NGX_HTTP_CACHE_VARY_LEN 42
29
30 #define NGX_HTTP_CACHE_VERSION 3
31
32
33 typedef struct {
34     ngx_uint_t status;
35     time_t valid;
36 } ngx_http_cache_valid_t;
37
38
39 typedef struct {
40     ngx_rbtree_node_t node;
41     ngx_queue_t queue;
42
43     u_char key[NGX_HTTP_CACHE_KEY_LEN
44             - sizeof(ngx_rbtree_key_t)];
45
46     unsigned count:20;
47     unsigned uses:10;
48     unsigned valid_msec:10;
49     unsigned error:10;
50     unsigned exists:1;
51     unsigned updating:1;
52     unsigned deleting:1;
53     /* 11 unused bits */
54
55     ngx_file_uniq_t uniq;
56     time_t expire;
57     time_t valid_sec;
58     size_t body_start;
59     off_t fs_size;
60     ngx_msec_t lock_time;
61 } ngx_http_file_cache_node_t;
62
63
64 struct ngx_http_cache_s {
65     ngx_file_t file;
66     ngx_array_t keys;
67     uint32_t crc32;
68     u_char key[NGX_HTTP_CACHE_KEY_LEN];
69     u_char main[NGX_HTTP_CACHE_KEY_LEN];
70
71     ngx_file_uniq_t uniq;
72     time_t valid_sec;
73     time_t last_modified;
74     time_t date;
75
76     ngx_str_t etag;
77     ngx_str_t vary;
78     u_char variant[NGX_HTTP_CACHE_KEY_LEN];
79
80     size_t header_start;
81     size_t body_start;
82     off_t length;
83     off_t fs_size;
84
85     ngx_uint_t min_uses;
86     ngx_uint_t error;
87     ngx_uint_t valid_msec;
88
89     ngx_buf_t *buf;
90
91     ngx_http_file_cache_t *file_cache;
92     ngx_http_file_cache_node_t *node;
93
94     ngx_msec_t lock_timeout;

```

```

95     ngx_msec_t          lock_age;
96     ngx_msec_t          lock_time;
97     ngx_msec_t          wait_time;
98
99     ngx_event_t          wait_event;
100
101     unsigned             lock:1;
102     unsigned             waiting:1;
103
104     unsigned             updated:1;
105     unsigned             updating:1;
106     unsigned             exists:1;
107     unsigned             temp_file:1;
108     unsigned             reading:1;
109     unsigned             secondary:1;
110 };
111
112
113 typedef struct {
114     ngx_uint_t           version;
115     time_t               valid_sec;
116     time_t               last_modified;
117     time_t               date;
118     uint32_t             crc32;
119     u_short              valid_msec;
120     u_short              header_start;
121     u_short              body_start;
122     u_char               etag_len;
123     u_char               etag[NGX_HTTP_CACHE_ETAG_LEN];
124     u_char               vary_len;
125     u_char               vary[NGX_HTTP_CACHE_VARY_LEN];
126     u_char               variant[NGX_HTTP_CACHE_KEY_LEN];
127 } ngx_http_file_cache_header_t;
128
129
130 typedef struct {
131     ngx_rbtree_t         rbtree;
132     ngx_rbtree_node_t   sentinel;
133     ngx_queue_t         queue;
134     ngx_atomic_t        cold;
135     ngx_atomic_t        loading;
136     off_t               size;
137 } ngx_http_file_cache_sh_t;
138
139
140 struct ngx_http_file_cache_s {
141     ngx_http_file_cache_sh_t *sh;
142     ngx_slab_pool_t *shpool;
143
144     ngx_path_t *path;
145     ngx_path_t *temp_path;
146
147     off_t max_size;
148     size_t bsize;
149
150     time_t inactive;
151
152     ngx_uint_t files;
153     ngx_uint_t loader_files;
154     ngx_msec_t last;
155     ngx_msec_t loader_sleep;
156     ngx_msec_t loader_threshold;
157
158     ngx_shm_zone_t *shm_zone;
159 };
160
161
162 ngx_int_t ngx_http_file_cache_new(ngx_http_request_t *r);
163 ngx_int_t ngx_http_file_cache_create(ngx_http_request_t *r);
164 void ngx_http_file_cache_create_key(ngx_http_request_t *r);
165 ngx_int_t ngx_http_file_cache_open(ngx_http_request_t *r);
166 ngx_int_t ngx_http_file_cache_set_header(ngx_http_request_t *r, u_char *buf);
167 void ngx_http_file_cache_update(ngx_http_request_t *r, ngx_temp_file_t *tf);
168 void ngx_http_file_cache_update_header(ngx_http_request_t *r);
169 ngx_int_t ngx_http_cache_send(ngx_http_request_t *r);
170 void ngx_http_file_cache_free(ngx_http_cache_t *c, ngx_temp_file_t *tf);

```

```
171 time_t ngx_http_file_cache_valid(ngx_array_t *cache_valid, ngx_uint_t status);
172
173 char *ngx_http_file_cache_set_slot(ngx_conf_t *cf, ngx_command_t *cmd,
174     void *conf);
175 char *ngx_http_file_cache_valid_set_slot(ngx_conf_t *cf, ngx_command_t *cmd,
176     void *conf);
177
178
179 extern ngx_str_t ngx_http_cache_status[];
180
181
182 #endif /* NGX_HTTP_CACHE_H_INCLUDED */
```

[One Level Up](#)

[Top Level](#)

## src/event/modules/nginx\_epoll\_module.c - nginx-1.7.10

### Global variables defined

- [ep](#)
- [epoll\\_name](#)
- [event\\_list](#)
- [nevents](#)
- [ngx\\_aio\\_ctx](#)
- [ngx\\_epoll\\_commands](#)
- [ngx\\_epoll\\_module](#)
- [ngx\\_epoll\\_module\\_ctx](#)
- [ngx\\_eventfd](#)
- [ngx\\_eventfd\\_conn](#)
- [ngx\\_eventfd\\_event](#)

### Data types defined

- [aio\\_context\\_t](#)
- [epoll\\_data](#)
- [epoll\\_data\\_t](#)
- [epoll\\_event](#)
- [io\\_event](#)
- [ngx\\_epoll\\_conf\\_t](#)

### Functions defined

- [epoll\\_create](#)
- [epoll\\_ctl](#)
- [epoll\\_wait](#)
- [io\\_destroy](#)
- [io\\_getevents](#)
- [io\\_setup](#)
- [ngx\\_epoll\\_add\\_connection](#)
- [ngx\\_epoll\\_add\\_event](#)
- [ngx\\_epoll\\_aio\\_init](#)
- [ngx\\_epoll\\_create\\_conf](#)

- [ngx\\_epoll\\_del\\_connection](#)
- [ngx\\_epoll\\_del\\_event](#)
- [ngx\\_epoll\\_done](#)
- [ngx\\_epoll\\_eventfd\\_handler](#)
- [ngx\\_epoll\\_init](#)
- [ngx\\_epoll\\_init\\_conf](#)
- [ngx\\_epoll\\_process\\_events](#)

## Macros defined

- [EPOLLERR](#)
- [EPOLLET](#)
- [EPOLLHUP](#)
- [EPOLLIN](#)
- [EPOLLMSG](#)
- [EPOLLONESHOT](#)
- [EPOLLOUT](#)
- [EPOLLPRI](#)
- [EPOLLRDBAND](#)
- [EPOLLRDHUP](#)
- [EPOLLRDNORM](#)
- [EPOLLWRBAND](#)
- [EPOLLWRNORM](#)
- [EPOLL\\_CTL\\_ADD](#)
- [EPOLL\\_CTL\\_DEL](#)
- [EPOLL\\_CTL\\_MOD](#)
- [SYS\\_eventfd](#)
- [SYS\\_io\\_destroy](#)
- [SYS\\_io\\_getevents](#)
- [SYS\\_io\\_setup](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
```

```

8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 #if (NGX_TEST_BUILD_EPOLL)
14
15 /* epoll declarations */
16
17 #define EPOLLIN      0x001
18 #define EPOLLPRI    0x002
19 #define EPOLLOUT    0x004
20 #define EPOLLRDNORM 0x040
21 #define EPOLLRDBAND 0x080
22 #define EPOLLWRNORM 0x100
23 #define EPOLLWRBAND 0x200
24 #define EPOLLMSG    0x400
25 #define EPOLLERR    0x008
26 #define EPOLLHUP    0x010
27
28 #define EPOLLRDHUP  0x2000
29
30 #define EPOLLET     0x80000000
31 #define EPOLLONESHOT 0x40000000
32
33 #define EPOLL_CTL_ADD 1
34 #define EPOLL_CTL_DEL 2
35 #define EPOLL_CTL_MOD 3
36
37 typedef union epoll_data {
38     void      *ptr;
39     int       fd;
40     uint32_t   u32;
41     uint64_t   u64;
42 } epoll_data_t;
43
44 struct epoll_event {
45     uint32_t   events;
46     epoll_data_t data;
47 };
48
49
50 int epoll_create(int size);
51
52 int epoll_create(int size)
53 {
54     return -1;
55 }
56
57
58 int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
59
60 int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)
61 {
62     return -1;
63 }
64
65
66 int epoll_wait(int epfd, struct epoll_event *events, int nevents, int timeout);
67
68 int epoll_wait(int epfd, struct epoll_event *events, int nevents, int timeout)
69 {
70     return -1;
71 }
72
73 #if (NGX_HAVE_FILE_AIO)
74
75 #define SYS_io_setup      245
76 #define SYS_io_destroy   246
77 #define SYS_io_getevents 247
78 #define SYS_eventfd      323
79
80 typedef u_int   aio_context_t;
81
82 struct io_event {
83     uint64_t data; /* the data field from the iocb */

```



```

84     uint64_t  obj;    /* what iocb this event came from */
85     int64_t   res;    /* result code for this event */
86     int64_t   res2;   /* secondary result */
87 };
88
89
90 #endif
91 #endif
92
93
94 typedef struct {
95     ngx_uint_t  events;
96     ngx_uint_t  aio_requests;
97 } ngx_epoll_conf_t;
98
99
100 static ngx_int_t ngx_epoll_init(ngx_cycle_t *cycle, ngx_msec_t timer);
101 static void ngx_epoll_done(ngx_cycle_t *cycle);
102 static ngx_int_t ngx_epoll_add_event(ngx_event_t *ev, ngx_int_t event,
103     ngx_uint_t flags);
104 static ngx_int_t ngx_epoll_del_event(ngx_event_t *ev, ngx_int_t event,
105     ngx_uint_t flags);
106 static ngx_int_t ngx_epoll_add_connection(ngx_connection_t *c);
107 static ngx_int_t ngx_epoll_del_connection(ngx_connection_t *c,
108     ngx_uint_t flags);
109 static ngx_int_t ngx_epoll_process_events(ngx_cycle_t *cycle, ngx_msec_t timer,
110     ngx_uint_t flags);
111
112 #if (NGX_HAVE_FILE_AIO)
113 static void ngx_epoll_eventfd_handler(ngx_event_t *ev);
114 #endif
115
116 static void *ngx_epoll_create_conf(ngx_cycle_t *cycle);
117 static char *ngx_epoll_init_conf(ngx_cycle_t *cycle, void *conf);
118
119 static int          ep = -1;
120 static struct epoll_event *event_list;
121 static ngx_uint_t  nevents;
122
123 #if (NGX_HAVE_FILE_AIO)
124
125 int          ngx_eventfd = -1;
126 aio_context_t ngx_aio_ctx = 0;
127
128 static ngx_event_t ngx_eventfd_event;
129 static ngx_connection_t ngx_eventfd_conn;
130
131 #endif
132
133 static ngx_str_t    epoll_name = ngx_string("epoll");
134
135 static ngx_command_t ngx_epoll_commands[] = {
136
137     { ngx_string("epoll_events"),
138       NGX_EVENT_CONF|NGX_CONF_TAKE1,
139       ngx_conf_set_num_slot,
140       0,
141       offsetof(ngx_epoll_conf_t, events),
142       NULL },
143
144     { ngx_string("worker_aio_requests"),
145       NGX_EVENT_CONF|NGX_CONF_TAKE1,
146       ngx_conf_set_num_slot,
147       0,
148       offsetof(ngx_epoll_conf_t, aio_requests),
149       NULL },
150
151     ngx_null_command
152 };
153
154
155 ngx_event_module_t ngx_epoll_module_ctx = {
156     &epoll_name,
157     ngx_epoll_create_conf,          /* create configuration */
158     ngx_epoll_init_conf,          /* init configuration */
159

```

```

160     {
161         ngx\_epoll\_add\_event,          /* add an event */
162         ngx\_epoll\_del\_event,        /* delete an event */
163         ngx\_epoll\_add\_event,        /* enable an event */
164         ngx\_epoll\_del\_event,        /* disable an event */
165         ngx\_epoll\_add\_connection,  /* add an connection */
166         ngx\_epoll\_del\_connection,  /* delete an connection */
167         NULL,                    /* process the changes */
168         ngx\_epoll\_process\_events,  /* process the events */
169         ngx\_epoll\_init,           /* init the events */
170         ngx\_epoll\_done,          /* done the events */
171     }
172 };
173
174 ngx\_module\_t ngx\_epoll\_module = {
175     NGX\_MODULE\_V1,
176     &ngx\_epoll\_module\_ctx,      /* module context */
177     ngx\_epoll\_commands,      /* module directives */
178     NGX\_EVENT\_MODULE,        /* module type */
179     NULL,                    /* init master */
180     NULL,                    /* init module */
181     NULL,                    /* init process */
182     NULL,                    /* init thread */
183     NULL,                    /* exit thread */
184     NULL,                    /* exit process */
185     NULL,                    /* exit master */
186     NGX\_MODULE\_V1\_PADDING
187 };
188
189
190 #if (NGX\_HAVE\_FILE\_AIO)
191
192 /*
193 * We call io\_setup\(\), io\_destroy\(\) io\_submit\(\), and io\_getevents\(\) directly
194 * as syscalls instead of libaio usage, because the library header file
195 * supports eventfd\(\) since 0.3.107 version only.
196 */
197
198 static int
199 io\_setup(u\_int nr_reqs, aio\_context\_t *ctx)
200 {
201     return syscall(SYS\_io\_setup, nr_reqs, ctx);
202 }
203
204
205 static int
206 io\_destroy(aio\_context\_t ctx)
207 {
208     return syscall(SYS\_io\_destroy, ctx);
209 }
210
211
212 static int
213 io\_getevents(aio\_context\_t ctx, long min_nr, long nr, struct io\_event *events,
214             struct timespec *tmo)
215 {
216     return syscall(SYS\_io\_getevents, ctx, min_nr, nr, events, tmo);
217 }
218
219
220 static void
221 ngx\_epoll\_aio\_init(ngx\_cycle\_t *cycle, ngx\_epoll\_conf\_t *epcf)
222 {
223     int n;
224     struct epoll\_event ee;
225
226     #if (NGX\_HAVE\_SYS\_EVENTFD\_H)
227         ngx\_eventfd = eventfd(0, 0);
228     #else
229         ngx\_eventfd = syscall(SYS\_eventfd, 0);
230     #endif
231
232     if (ngx\_eventfd == -1) {
233         ngx\_log\_error(NGX\_LOG\_EMERG, cycle->log, ngx\_errno,
234                     "eventfd() failed");
235         ngx\_file\_aio = 0;

```

```

236     return;
237 }
238
239 ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
240     "eventfd: %d", ngx_eventfd);
241
242 n = 1;
243
244 if (ioctl(ngx_eventfd, FIONBIO, &n) == -1) {
245     ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
246         "ioctl(eventfd, FIONBIO) failed");
247     goto failed;
248 }
249
250 if (io_setup(epcf->aio_requests, &ngx_aio_ctx) == -1) {
251     ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
252         "io_setup() failed");
253     goto failed;
254 }
255
256 ngx_eventfd_event.data = &ngx_eventfd_conn;
257 ngx_eventfd_event.handler = ngx_epoll_eventfd_handler;
258 ngx_eventfd_event.log = cycle->log;
259 ngx_eventfd_event.active = 1;
260 ngx_eventfd_conn.fd = ngx_eventfd;
261 ngx_eventfd_conn.read = &ngx_eventfd_event;
262 ngx_eventfd_conn.log = cycle->log;
263
264 ee.events = EPOLLIN|EPOLLET;
265 ee.data.ptr = &ngx_eventfd_conn;
266
267 if (epoll_ctl(ep, EPOLL_CTL_ADD, ngx_eventfd, &ee) != -1) {
268     return;
269 }
270
271 ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
272     "epoll_ctl(EPOLL_CTL_ADD, eventfd) failed");
273
274 if (io_destroy(ngx_aio_ctx) == -1) {
275     ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
276         "io_destroy() failed");
277 }
278
279 failed:
280
281 if (close(ngx_eventfd) == -1) {
282     ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
283         "eventfd close() failed");
284 }
285
286 ngx_eventfd = -1;
287 ngx_aio_ctx = 0;
288 ngx_file_aio = 0;
289 }
290
291 #endif
292
293
294 static ngx_int_t
295 ngx_epoll_init(ngx_cycle_t *cycle, ngx_msec_t timer)
296 {
297     ngx_epoll_conf_t *epcf;
298
299     epcf = ngx_event_get_conf(cycle->conf_ctx, ngx_epoll_module);
300
301     if (ep == -1) {
302         ep = epoll_create(cycle->connection_n / 2);
303
304         if (ep == -1) {
305             ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
306                 "epoll_create() failed");
307             return NGX_ERROR;
308         }
309     }
310
311 #if (NGX_HAVE_FILE_AIO)

```

```

312     ngx_epoll_aio_init(cycle, epcf);
313
314 #endif
315     }
316
317     if (nevents < epcf->events) {
318         if (event_list) {
319             ngx_free(event_list);
320         }
321
322         event_list = ngx_alloc(sizeof(struct epoll_event) * epcf->events,
323                               cycle->log);
324         if (event_list == NULL) {
325             return NGX_ERROR;
326         }
327     }
328
329     nevents = epcf->events;
330
331     ngx_io = ngx_os_io;
332
333     ngx_event_actions = ngx_epoll_module_ctx.actions;
334
335 #if (NGX_HAVE_CLEAR_EVENT)
336     ngx_event_flags = NGX_USE_CLEAR_EVENT
337 #else
338     ngx_event_flags = NGX_USE_LEVEL_EVENT
339 #endif
340                     |NGX_USE_GREEDY_EVENT
341                     |NGX_USE_EPOLL_EVENT;
342
343     return NGX_OK;
344 }
345
346
347 static void
348 ngx_epoll_done(ngx_cycle_t *cycle)
349 {
350     if (close(ep) == -1) {
351         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
352                     "epoll close() failed");
353     }
354
355     ep = -1;
356
357 #if (NGX_HAVE_FILE_AIO)
358
359     if (ngx_eventfd != -1) {
360
361         if (io_destroy(ngx_aio_ctx) == -1) {
362             ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
363                         "io_destroy() failed");
364         }
365
366         if (close(ngx_eventfd) == -1) {
367             ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
368                         "eventfd close() failed");
369         }
370
371         ngx_eventfd = -1;
372     }
373
374     ngx_aio_ctx = 0;
375
376 #endif
377
378     ngx_free(event_list);
379
380     event_list = NULL;
381     nevents = 0;
382 }
383
384
385 static ngx_int_t
386 ngx_epoll_add_event(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags)
387 {

```

```

388     int                op;
389     uint32_t          events, prev;
390     ngx_event_t      *e;
391     ngx_connection_t *c;
392     struct epoll_event ee;
393
394     c = ev->data;
395
396     events = (uint32_t) event;
397
398     if (event == NGX_READ_EVENT) {
399         e = c->write;
400         prev = EPOLLOUT;
401 #if (NGX_READ_EVENT != EPOLLIN|EPOLLRDHUP)
402         events = EPOLLIN|EPOLLRDHUP;
403 #endif
404
405     } else {
406         e = c->read;
407         prev = EPOLLIN|EPOLLRDHUP;
408 #if (NGX_WRITE_EVENT != EPOLLOUT)
409         events = EPOLLOUT;
410 #endif
411     }
412
413     if (e->active) {
414         op = EPOLL_CTL_MOD;
415         events |= prev;
416
417     } else {
418         op = EPOLL_CTL_ADD;
419     }
420
421     ee.events = events | (uint32_t) flags;
422     ee.data.ptr = (void *) ((uintptr_t) c | ev->instance);
423
424     ngx_log_debug3(NGX_LOG_DEBUG_EVENT, ev->log, 0,
425                  "epoll add event: fd:%d op:%d ev:%08XD",
426                  c->fd, op, ee.events);
427
428     if (epoll_ctl(ep, op, c->fd, &ee) == -1) {
429         ngx_log_error(NGX_LOG_ALERT, ev->log, ngx_errno,
430                      "epoll_ctl(%d, %d) failed", op, c->fd);
431         return NGX_ERROR;
432     }
433
434     ev->active = 1;
435 #if 0
436     ev->oneshot = (flags & NGX_ONESHOT_EVENT) ? 1 : 0;
437 #endif
438
439     return NGX_OK;
440 }
441
442
443 static ngx_int_t
444 ngx_epoll_del_event(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags)
445 {
446     int                op;
447     uint32_t          prev;
448     ngx_event_t      *e;
449     ngx_connection_t *c;
450     struct epoll_event ee;
451
452     /*
453      * when the file descriptor is closed, the epoll automatically deletes
454      * it from its queue, so we do not need to delete explicitly the event
455      * before the closing the file descriptor
456      */
457
458     if (flags & NGX_CLOSE_EVENT) {
459         ev->active = 0;
460         return NGX_OK;
461     }
462
463     c = ev->data;

```

```

464     if (event == NGX\_READ\_EVENT) {
465         e = c->write;
466         prev = EPOLLOUT;
467     } else {
468         e = c->read;
469         prev = EPOLLIN|EPOLLRDHUP;
470     }
471 }
472
473 if (e->active) {
474     op = EPOLL\_CTL\_MOD;
475     ee.events = prev | (uint32_t) flags;
476     ee.data.ptr = (void *) ((uintptr_t) c | ev->instance);
477 } else {
478     op = EPOLL\_CTL\_DEL;
479     ee.events = 0;
480     ee.data.ptr = NULL;
481 }
482
483 ngx\_log\_debug3(NGX\_LOG\_DEBUG\_EVENT, ev->log, 0,
484     "epoll del event: fd:%d op:%d ev:%08XD",
485     c->fd, op, ee.events);
486
487 if (epoll\_ctl(ep, op, c->fd, &ee) == -1) {
488     ngx\_log\_error(NGX\_LOG\_ALERT, ev->log, ngx\_errno,
489         "epoll\_ctl(%d, %d) failed", op, c->fd);
490     return NGX\_ERROR;
491 }
492
493 ev->active = 0;
494
495 return NGX\_OK;
496 }
497
498
499
500
501 static ngx\_int\_t
502 ngx\_epoll\_add\_connection(ngx\_connection\_t *c)
503 {
504     struct epoll\_event ee;
505
506     ee.events = EPOLLIN|EPOLLOUT|EPOLLET|EPOLLRDHUP;
507     ee.data.ptr = (void *) ((uintptr_t) c | c->read->instance);
508
509     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_EVENT, c->log, 0,
510         "epoll add connection: fd:%d ev:%08XD", c->fd, ee.events);
511
512     if (epoll\_ctl(ep, EPOLL\_CTL\_ADD, c->fd, &ee) == -1) {
513         ngx\_log\_error(NGX\_LOG\_ALERT, c->log, ngx\_errno,
514             "epoll\_ctl(EPOLL\_CTL\_ADD, %d) failed", c->fd);
515         return NGX\_ERROR;
516     }
517
518     c->read->active = 1;
519     c->write->active = 1;
520
521     return NGX\_OK;
522 }
523
524
525 static ngx\_int\_t
526 ngx\_epoll\_del\_connection(ngx\_connection\_t *c, ngx\_uint\_t flags)
527 {
528     int op;
529     struct epoll\_event ee;
530
531     /*
532      * when the file descriptor is closed the epoll automatically deletes
533      * it from its queue so we do not need to delete explicitly the event
534      * before the closing the file descriptor
535      */
536
537     if (flags & NGX\_CLOSE\_EVENT) {
538         c->read->active = 0;
539         c->write->active = 0;

```

```

540     return NGX_OK;
541 }
542
543 ngx_log_debug1(NGX_LOG_DEBUG_EVENT, c->log, 0,
544     "epoll del connection: fd:%d", c->fd);
545
546 op = EPOLL_CTL_DEL;
547 ee.events = 0;
548 ee.data.ptr = NULL;
549
550 if (epoll_ctl(ep, op, c->fd, &ee) == -1) {
551     ngx_log_error(NGX_LOG_ALERT, c->log, ngx_errno,
552         "epoll_ctl(%d, %d) failed", op, c->fd);
553     return NGX_ERROR;
554 }
555
556 c->read->active = 0;
557 c->write->active = 0;
558
559 return NGX_OK;
560 }
561
562
563 static ngx_int_t
564 ngx_epoll_process_events(ngx_cycle_t *cycle, ngx_msec_t timer, ngx_uint_t flags)
565 {
566     int                events;
567     uint32_t           revents;
568     ngx_int_t         instance, i;
569     ngx_uint_t        level;
570     ngx_err_t         err;
571     ngx_event_t      *rev, *wev;
572     ngx_queue_t      *queue;
573     ngx_connection_t *c;
574
575     /* NGX_TIMER_INFINITE == INFTIM */
576
577     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
578         "epoll timer: %M", timer);
579
580     events = epoll_wait(ep, event_list, (int) nevents, timer);
581
582     err = (events == -1) ? ngx_errno : 0;
583
584     if (flags & NGX_UPDATE_TIME || ngx_event_timer_alarm) {
585         ngx_time_update();
586     }
587
588     if (err) {
589         if (err == NGX_EINTR) {
590
591             if (ngx_event_timer_alarm) {
592                 ngx_event_timer_alarm = 0;
593                 return NGX_OK;
594             }
595
596             level = NGX_LOG_INFO;
597
598         } else {
599             level = NGX_LOG_ALERT;
600         }
601
602         ngx_log_error(level, cycle->log, err, "epoll_wait() failed");
603         return NGX_ERROR;
604     }
605
606     if (events == 0) {
607         if (timer != NGX_TIMER_INFINITE) {
608             return NGX_OK;
609         }
610
611         ngx_log_error(NGX_LOG_ALERT, cycle->log, 0,
612             "epoll_wait() returned no events without timeout");
613         return NGX_ERROR;
614     }
615

```

```

616 for (i = 0; i < events; i++) {
617     c = event\_list[i].data.ptr;
618
619     instance = (uintptr_t) c & 1;
620     c = (ngx\_connection\_t *) ((uintptr_t) c & (uintptr_t) ~1);
621
622     rev = c->read;
623
624     if (c->fd == -1 || rev->instance != instance) {
625
626         /*
627          * the stale event from a file descriptor
628          * that was just closed in this iteration
629          */
630
631         ngx\_log\_debug1(NGX\_LOG\_DEBUG\_EVENT, cycle->log, 0,
632             "epoll: stale event %p", c);
633         continue;
634     }
635
636     revents = event\_list[i].events;
637
638     ngx\_log\_debug3(NGX\_LOG\_DEBUG\_EVENT, cycle->log, 0,
639         "epoll: fd:%d ev:%04XD d:%p",
640         c->fd, revents, event\_list[i].data.ptr);
641
642     if (revents & (EPOLLERR|EPOLLHUP)) {
643         ngx\_log\_debug2(NGX\_LOG\_DEBUG\_EVENT, cycle->log, 0,
644             "epoll wait() error on fd:%d ev:%04XD",
645             c->fd, revents);
646     }
647
648     #if 0
649     if (revents & ~(EPOLLIN|EPOLLOUT|EPOLLERR|EPOLLHUP)) {
650         ngx\_log\_error(NGX\_LOG\_ALERT, cycle->log, 0,
651             "strange epoll wait() events fd:%d ev:%04XD",
652             c->fd, revents);
653     }
654     #endif
655
656     if ((revents & (EPOLLERR|EPOLLHUP))
657         && (revents & (EPOLLIN|EPOLLOUT)) == 0)
658     {
659         /*
660          * if the error events were returned without EPOLLIN or EPOLLOUT,
661          * then add these flags to handle the events at least in one
662          * active handler
663          */
664
665         revents |= EPOLLIN|EPOLLOUT;
666     }
667
668     if ((revents & EPOLLIN) && rev->active) {
669
670         #if (NGX\_HAVE\_EPOLLRDHUP)
671         if (revents & EPOLLRDHUP) {
672             rev->pending_eof = 1;
673         }
674         #endif
675
676         rev->ready = 1;
677
678         if (flags & NGX\_POST\_EVENTS) {
679             queue = rev->accept ? ngx\_posted\_accept\_events
680                 : ngx\_posted\_events;
681
682             ngx\_post\_event(rev, queue);
683
684         } else {
685             rev->handler(rev);
686         }
687     }
688
689     wev = c->write;
690
691     if ((revents & EPOLLOUT) && wev->active) {

```



```

692     if (c->fd == -1 || wev->instance != instance) {
693
694         /*
695          * the stale event from a file descriptor
696          * that was just closed in this iteration
697          */
698
699
700         ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
701                       "epoll: stale event %p", c);
702         continue;
703     }
704
705     wev->ready = 1;
706
707     if (flags & NGX_POST_EVENTS) {
708         ngx_post_event(wev, &ngx_posted_events);
709     } else {
710         wev->handler(wev);
711     }
712 }
713 }
714 }
715
716 return NGX_OK;
717 }
718
719
720 #if (NGX_HAVE_FILE_AIO)
721
722 static void
723 ngx_epoll_eventfd_handler(ngx_event_t *ev)
724 {
725     int             n, events;
726     long            i;
727     uint64_t        ready;
728     ngx_err_t       err;
729     ngx_event_t     *e;
730     ngx_event_aio_t *aio;
731     struct io_event event[64];
732     struct timespec  ts;
733
734     ngx_log_debug0(NGX_LOG_DEBUG_EVENT, ev->log, 0, "eventfd handler");
735
736     n = read(ngx_eventfd, &ready, 8);
737
738     err = ngx_errno;
739
740     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, ev->log, 0, "eventfd: %d", n);
741
742     if (n != 8) {
743         if (n == -1) {
744             if (err == NGX_EAGAIN) {
745                 return;
746             }
747
748             ngx_log_error(NGX_LOG_ALERT, ev->log, err, "read(eventfd) failed");
749             return;
750         }
751
752         ngx_log_error(NGX_LOG_ALERT, ev->log, 0,
753                       "read(eventfd) returned only %d bytes", n);
754         return;
755     }
756
757     ts.tv_sec = 0;
758     ts.tv_nsec = 0;
759
760     while (ready) {
761
762         events = io_getevents(ngx_aio_ctx, 1, 64, event, &ts);
763
764         ngx_log_debug1(NGX_LOG_DEBUG_EVENT, ev->log, 0,
765                       "io_getevents: %l", events);
766
767         if (events > 0) {

```

```

768         ready -= events;
769
770         for (i = 0; i < events; i++) {
771             ngx\_log\_debug4(NGX\_LOG\_DEBUG\_EVENT, ev->log, 0,
772                 "io\_event: %uXL %uXL %L %L",
773                 event[i].data, event[i].obj,
774                 event[i].res, event[i].res2);
775
776             e = (ngx\_event\_t *) (uintptr_t) event[i].data;
777
778             e->complete = 1;
779             e->active = 0;
780             e->ready = 1;
781
782             aio = e->data;
783             aio->res = event[i].res;
784
785             ngx\_post\_event(e, &ngx\_posted\_events);
786         }
787
788         continue;
789     }
790
791     if (events == 0) {
792         return;
793     }
794
795     /* events == -1 */
796     ngx\_log\_error(NGX\_LOG\_ALERT, ev->log, ngx\_errno,
797                 "io\_getevents() failed");
798     return;
799 }
800 }
801
802 #endif
803
804
805
806 static void *
807 ngx\_epoll\_create\_conf(ngx\_cycle\_t *cycle)
808 {
809     ngx\_epoll\_conf\_t *epcf;
810
811     epcf = ngx\_palloc(cycle->pool, sizeof(ngx\_epoll\_conf\_t));
812     if (epcf == NULL) {
813         return NULL;
814     }
815
816     epcf->events = NGX\_CONF\_UNSET;
817     epcf->aio_requests = NGX\_CONF\_UNSET;
818
819     return epcf;
820 }
821
822
823 static char *
824 ngx\_epoll\_init\_conf(ngx\_cycle\_t *cycle, void *conf)
825 {
826     ngx\_epoll\_conf\_t *epcf = conf;
827
828     ngx\_conf\_init\_uint\_value(epcf->events, 512);
829     ngx\_conf\_init\_uint\_value(epcf->aio_requests, 32);
830
831     return NGX\_CONF\_OK;
832 }

```

[One Level Up](#)

[Top Level](#)

# src/os/unix/nginx\_linux\_aio\_read.c - nginx-1.7.10

## Functions defined

- [io\\_submit](#)
- [ngx\\_file\\_aio\\_event\\_handler](#)
- [ngx\\_file\\_aio\\_read](#)

## Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 extern int          ngx_eventfd;
14 extern aio_context_t ngx_aio_ctx;
15
16
17 static void ngx_file_aio_event_handler(ngx_event_t *ev);
18
19
20 static int
21 io_submit(aio_context_t ctx, long n, struct iocb **paiocb)
22 {
23     return syscall(SYS_io_submit, ctx, n, paiocb);
24 }
25
26
27 ssize_t
28 ngx_file_aio_read(ngx_file_t *file, u_char *buf, size_t size, off_t offset,
29                 ngx_pool_t *pool)
30 {
31     ngx_err_t      err;
32     struct iocb     *piocb[1];
33     ngx_event_t     *ev;
34     ngx_event_aio_t *aio;
35
36     if (!ngx_file_aio) {
37         return ngx_read_file(file, buf, size, offset);
38     }
39
40     aio = file->aio;
41
42     if (aio == NULL) {
43         aio = ngx_palloc(pool, sizeof(ngx_event_aio_t));
44         if (aio == NULL) {
45             return NGX_ERROR;
46         }
47
48         aio->file = file;
49         aio->fd = file->fd;
50         aio->event.data = aio;
51         aio->event.ready = 1;
52         aio->event.log = file->log;
53         file->aio = aio;
54     }
55
56     ev = &aio->event;
57
58     if (!ev->ready) {
```

```

59     ngx_log_error(NGX_LOG_ALERT, file->log, 0,
60                 "second aio post for \"%V\"", &file->name);
61     return NGX_AGAIN;
62 }
63
64 ngx_log_debug4(NGX_LOG_DEBUG_CORE, file->log, 0,
65               "aio complete:%d @%O:%z %V",
66               ev->complete, offset, size, &file->name);
67
68 if (ev->complete) {
69     ev->active = 0;
70     ev->complete = 0;
71
72     if (aio->res >= 0) {
73         ngx_set_errno(0);
74         return aio->res;
75     }
76
77     ngx_set_errno(-aio->res);
78
79     ngx_log_error(NGX_LOG_CRIT, file->log, ngx_errno,
80                 "aio read \"%s\" failed", file->name.data);
81
82     return NGX_ERROR;
83 }
84
85 ngx_memzero(&aio->aiocb, sizeof(struct iocb));
86
87 aio->aiocb.aio_data = (uint64_t) (uintptr_t) ev;
88 aio->aiocb.aio_lio_opcode = IOCB_CMD_PREAD;
89 aio->aiocb.aio_fildes = file->fd;
90 aio->aiocb.aio_buf = (uint64_t) (uintptr_t) buf;
91 aio->aiocb.aio_nbytes = size;
92 aio->aiocb.aio_offset = offset;
93 aio->aiocb.aio_flags = IOCB_FLAG_RESFD;
94 aio->aiocb.aio_resfd = ngx_eventfd;
95
96 ev->handler = ngx_file_aio_event_handler;
97
98 piocb[0] = &aio->aiocb;
99
100 if (io_submit(ngx_aio_ctx, 1, piocb) == 1) {
101     ev->active = 1;
102     ev->ready = 0;
103     ev->complete = 0;
104
105     return NGX_AGAIN;
106 }
107
108 err = ngx_errno;
109
110 if (err == NGX_EAGAIN) {
111     return ngx_read_file(file, buf, size, offset);
112 }
113
114 ngx_log_error(NGX_LOG_CRIT, file->log, err,
115             "io_submit(\"%V\") failed", &file->name);
116
117 if (err == NGX_ENOSYS) {
118     ngx_file_aio = 0;
119     return ngx_read_file(file, buf, size, offset);
120 }
121
122 return NGX_ERROR;
123 }
124
125
126 static void
127 ngx_file_aio_event_handler(ngx_event_t *ev)
128 {
129     ngx_event_aio_t *aio;
130
131     aio = ev->data;
132
133     ngx_log_debug2(NGX_LOG_DEBUG_CORE, ev->log, 0,
134                 "aio event handler fd:%d %V", aio->fd, &aio->file->name);

```

```
135
136 aio->handler(ev);
137 }
```

[One Level Up](#)

[Top Level](#)

# src/os/unix/nginx\_freebsd\_config.h - nginx-1.7.10

## Data types defined

- [ngx\\_aiocb\\_t](#)

## Macros defined

- [CMMSG\\_DATA](#)
- [CMMSG\\_DATA](#)
- [CMMSG\\_LEN](#)
- [CMMSG\\_LEN](#)
- [CMMSG\\_SPACE](#)
- [CMMSG\\_SPACE](#)
- [IOV\\_MAX](#)
- [NGX\\_HAVE\\_DEBUG\\_MALLOC](#)
- [NGX\\_HAVE\\_INHERITED\\_NONBLOCK](#)
- [NGX\\_HAVE\\_OS\\_SPECIFIC\\_INIT](#)
- [NGX\\_KEEPALIVE\\_FACTOR](#)
- [NGX\\_LISTEN\\_BACKLOG](#)
- [\\_NGX\\_FREEBSD\\_CONFIG\\_H\\_INCLUDED](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_FREEBSD\_CONFIG\_H\_INCLUDED
9 #define \_NGX\_FREEBSD\_CONFIG\_H\_INCLUDED
10
11
12 #include <sys/types.h>
13 #include <sys/time.h>
14 #include <unistd.h>
15 #include <stdarg.h>
16 #include <stddef.h>          /* offsetof() */
17 #include <stdio.h>
18 #include <stdlib.h>
19 #include <ctype.h>
20 #include <errno.h>
21 #include <string.h>
22 #include <signal.h>
23 #include <pwd.h>
24 #include <grp.h>
25 #include <dirent.h>
26 #include <glob.h>
27 #include <time.h>
28 #include <sys/param.h>      /* ALIGN() */
29 #include <sys/mount.h>     /* statfs() */
```

```

30
31 #include <sys/filio.h>          /* FIONBIO */
32 #include <sys/uio.h>
33 #include <sys/stat.h>
34 #include <fcntl.h>
35
36 #include <sys/wait.h>
37 #include <sys/mman.h>
38 #include <sys/resource.h>
39 #include <sched.h>
40
41 #include <sys/socket.h>
42 #include <netinet/in.h>
43 #include <netinet/tcp.h>      /* TCP_NODELAY, TCP_NOPUSH */
44 #include <arpa/inet.h>
45 #include <netdb.h>
46 #include <sys/un.h>
47
48 #include <libutil.h>          /* setproctitle() before 4.1 */
49 #include <osreldate.h>
50 #include <sys/sysctl.h>
51
52
53 #if __FreeBSD_version < 400017
54
55 /*
56 * FreeBSD 3.x has no CMSG_SPACE() and CMSG_LEN() and has the broken CMSG_DATA()
57 */
58
59 #undef CMSG_SPACE
60 #define CMSG_SPACE(l)        (ALIGN(sizeof(struct cmsghdr)) + ALIGN(l))
61
62 #undef CMSG_LEN
63 #define CMSG_LEN(l)         (ALIGN(sizeof(struct cmsghdr)) + (l))
64
65 #undef CMSG_DATA
66 #define CMSG_DATA(cmsg)     ((u_char *)(cmsg) + ALIGN(sizeof(struct cmsghdr)))
67
68 #endif
69
70
71 #include <ngx_auto_config.h>
72
73
74 #if (NGX_HAVE_POSIX_SEM)
75 #include <semaphore.h>
76 #endif
77
78
79 #if (NGX_HAVE_POLL)
80 #include <poll.h>
81 #endif
82
83
84 #if (NGX_HAVE_KQUEUE)
85 #include <sys/event.h>
86 #endif
87
88
89 #if (NGX_HAVE_FILE_AIO || NGX_HAVE_AIO)
90 #include <aio.h>
91 typedef struct aiocb ngx_aiocb_t;
92 #endif
93
94
95 #define NGX_LISTEN_BACKLOG    -1
96
97
98 #ifdef __DragonFly__
99 #define NGX_KEEPALIVE_FACTOR  1000
100 #endif
101
102
103 #if (__FreeBSD_version < 430000 || __FreeBSD_version < 500012)
104
105 pid_t rfork_thread(int flags, void *stack, int (*func)(void *arg), void *arg);

```

```
106
107 #endif
108
109 #ifndef IOV_MAX
110 #define IOV_MAX 1024
111 #endif
112
113
114 #ifndef NGX_HAVE_INHERITED_NONBLOCK
115 #define NGX_HAVE_INHERITED_NONBLOCK 1
116 #endif
117
118
119 #define NGX_HAVE_OS_SPECIFIC_INIT 1
120 #define NGX_HAVE_DEBUG_MALLOC 1
121
122
123 extern char **environ;
124 extern char *malloc_options;
125
126
127 #endif /* NGX_FREEBSD_CONFIG_H_INCLUDED */
```

[One Level Up](#)

[Top Level](#)



# src/os/unix/nginx\_readv\_chain.c - nginx-1.7.10

## Functions defined

- [ngx\\_readv\\_chain](#)

## Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 ssize_t
14 ngx_readv_chain(ngx_connection_t *c, ngx_chain_t *chain, off_t limit)
15 {
16     u_char      *prev;
17     ssize_t      n, size;
18     ngx_err_t    err;
19     ngx_array_t  vec;
20     ngx_event_t  *rev;
21     struct iovec *iov, iovs[NGX_IOVS_PREALLOCATE];
22
23     rev = c->read;
24
25     #if (NGX_HAVE_KQUEUE)
26
27     if (ngx_event_flags & NGX_USE_KQUEUE_EVENT) {
28         ngx_log_debug3(NGX_LOG_DEBUG_EVENT, c->log, 0,
29             "readv: eof:%d, avail:%d, err:%d",
30             rev->pending_eof, rev->available, rev->kq_errno);
31
32         if (rev->available == 0) {
33             if (rev->pending_eof) {
34                 rev->ready = 0;
35                 rev->eof = 1;
36
37                 ngx_log_error(NGX_LOG_INFO, c->log, rev->kq_errno,
38                     "kevent() reported about a closed connection");
39
40                 if (rev->kq_errno) {
41                     rev->error = 1;
42                     ngx_set_socket_errno(rev->kq_errno);
43                     return NGX_ERROR;
44                 }
45
46                 return 0;
47             } else {
48                 return NGX_AGAIN;
49             }
50         }
51     }
52 }
53
54 #endif
55
56 prev = NULL;
57 iov = NULL;
58 size = 0;
59
60 vec.elts = iovs;
61 vec.nelts = 0;
62 vec.size = sizeof(struct iovec);
```

```

63     vec.nalloc = NGX\_IOVS\_PREALLOCATE;
64     vec.pool = c->pool;
65
66     /* coalesce the neighbouring bufs */
67
68     while (chain) {
69         n = chain->buf->end - chain->buf->last;
70
71         if (limit) {
72             if (size >= limit) {
73                 break;
74             }
75
76             if (size + n > limit) {
77                 n = (ssize_t) (limit - size);
78             }
79         }
80
81         if (prev == chain->buf->last) {
82             iov->iov_len += n;
83
84         } else {
85             if (vec.nelts >= IOV\_MAX) {
86                 break;
87             }
88
89             iov = ngx\_array\_push(&vec);
90             if (iov == NULL) {
91                 return NGX\_ERROR;
92             }
93
94             iov->iov_base = (void *) chain->buf->last;
95             iov->iov_len = n;
96         }
97
98         size += n;
99         prev = chain->buf->end;
100        chain = chain->next;
101    }
102
103    ngx\_log\_debug2(NGX\_LOG\_DEBUG\_EVENT, c->log, 0,
104        "readv: %d, last:%d", vec.nelts, iov->iov_len);
105
106    do {
107        n = readv(c->fd, (struct iovec *) vec.elts, vec.nelts);
108
109        if (n >= 0) {
110
111            #if (NGX_HAVE_KQUEUE)
112
113                if (ngx\_event\_flags & NGX\_USE\_KQUEUE\_EVENT) {
114                    rev->available -= n;
115
116                    /*
117                    * rev->available may be negative here because some additional
118                    * bytes may be received between kevent() and recv()
119                    */
120
121                    if (rev->available <= 0) {
122                        if (!rev->pending_eof) {
123                            rev->ready = 0;
124                        }
125
126                        if (rev->available < 0) {
127                            rev->available = 0;
128                        }
129                    }
130
131                    if (n == 0) {
132
133                        /*
134                        * on FreeBSD recv() may return 0 on closed socket
135                        * even if queue reported about available data
136                        */
137
138                    #if 0

```

```

139         ngx_log_error(NGX_LOG_ALERT, c->log, 0,
140             "readv() returned 0 while kevent() reported "
141             "%d available bytes", rev->available);
142     #endif
143
144         rev->ready = 0;
145         rev->eof = 1;
146         rev->available = 0;
147     }
148
149     return n;
150 }
151
152 #endif /* NGX_HAVE_KQUEUE */
153
154     if (n < size && !(ngx_event_flags & NGX_USE_GREEDY_EVENT)) {
155         rev->ready = 0;
156     }
157
158     if (n == 0) {
159         rev->eof = 1;
160     }
161
162     return n;
163 }
164
165 err = ngx_socket_errno;
166
167 if (err == NGX_EAGAIN || err == NGX_EINTR) {
168     ngx_log_debug0(NGX_LOG_DEBUG_EVENT, c->log, err,
169         "readv() not ready");
170     n = NGX_AGAIN;
171 } else {
172     n = ngx_connection_error(c, err, "readv() failed");
173     break;
174 }
175
176 } while (err == NGX_EINTR);
177
178 rev->ready = 0;
179
180 if (n == NGX_ERROR) {
181     c->read->error = 1;
182 }
183
184 return n;
185 }
186 }

```

[One Level Up](#)

[Top Level](#)

# src/os/unix/nginx\_udp\_recv.c - nginx-1.7.10

## Functions defined

- [ngx\\_udp\\_unix\\_recv](#)
- [ngx\\_udp\\_unix\\_recv](#)

## Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 #if (NGX_HAVE_KQUEUE)
14
15 ssize_t
16 ngx_udp_unix_recv(ngx_connection_t *c, u_char *buf, size_t size)
17 {
18     ssize_t    n;
19     ngx_err_t  err;
20     ngx_event_t *rev;
21
22     rev = c->read;
23
24     do {
25         n = recv(c->fd, buf, size, 0);
26
27         ngx_log_debug3(NGX_LOG_DEBUG_EVENT, c->log, 0,
28             "recv: fd:%d %d of %d", c->fd, n, size);
29
30         if (n >= 0) {
31             if (ngx_event_flags & NGX_USE_KQUEUE_EVENT) {
32                 rev->available -= n;
33
34                 /*
35                  * rev->available may be negative here because some additional
36                  * bytes may be received between kevent() and recv()
37                  */
38
39                 if (rev->available <= 0) {
40                     rev->ready = 0;
41                     rev->available = 0;
42                 }
43             }
44
45             return n;
46         }
47
48         err = ngx_socket_errno;
49
50         if (err == NGX_EAGAIN || err == NGX_EINTR) {
51             ngx_log_debug0(NGX_LOG_DEBUG_EVENT, c->log, err,
52                 "recv() not ready");
53             n = NGX_AGAIN;
54
55         } else {
56             n = ngx_connection_error(c, err, "recv() failed");
57             break;
58         }
59     } while (err == NGX_EINTR);
60 }
```

```

61     rev->ready = 0;
62
63     if (n == NGX_ERROR) {
64         rev->error = 1;
65     }
66
67     return n;
68 }
69
70
71 #else /* ! NGX_HAVE_KQUEUE */
72
73 ssize_t
74 ngx_udp_unix_recv(ngx_connection_t *c, u_char *buf, size_t size)
75 {
76     ssize_t     n;
77     ngx_err_t   err;
78     ngx_event_t *rev;
79
80     rev = c->read;
81
82     do {
83         n = recv(c->fd, buf, size, 0);
84
85         ngx_log_debug3(NGX_LOG_DEBUG_EVENT, c->log, 0,
86             "recv: fd:%d %d of %d", c->fd, n, size);
87
88         if (n >= 0) {
89             return n;
90         }
91
92         err = ngx_socket_errno;
93
94         if (err == NGX_EAGAIN || err == NGX_EINTR) {
95             ngx_log_debug0(NGX_LOG_DEBUG_EVENT, c->log, err,
96                 "recv() not ready");
97             n = NGX_AGAIN;
98         }
99         else {
100             n = ngx_connection_error(c, err, "recv() failed");
101             break;
102         }
103     } while (err == NGX_EINTR);
104
105     rev->ready = 0;
106
107     if (n == NGX_ERROR) {
108         rev->error = 1;
109     }
110
111     return n;
112 }
113
114
115 #endif /* NGX_HAVE_KQUEUE */

```

[One Level Up](#)

[Top Level](#)

# src/os/unix/nginx\_send.c - nginx-1.7.10

## Functions defined

- [ngx\\_unix\\_send](#)

## Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 ssize_t
14 ngx_unix_send(ngx_connection_t *c, u_char *buf, size_t size)
15 {
16     ssize_t    n;
17     ngx_err_t  err;
18     ngx_event_t *wev;
19
20     wev = c->write;
21
22     #if (NGX_HAVE_KQUEUE)
23
24     if ((ngx_event_flags & NGX_USE_KQUEUE_EVENT) && wev->pending_eof) {
25         (void) ngx_connection_error(c, wev->kq_errno,
26                                     "kevent() reported about an closed connection");
27         wev->error = 1;
28         return NGX_ERROR;
29     }
30
31     #endif
32
33     for ( ;; ) {
34         n = send(c->fd, buf, size, 0);
35
36         ngx_log_debug3(NGX_LOG_DEBUG_EVENT, c->log, 0,
37                       "send: fd:%d %d of %d", c->fd, n, size);
38
39         if (n > 0) {
40             if (n < (ssize_t) size) {
41                 wev->ready = 0;
42             }
43
44             c->sent += n;
45
46             return n;
47         }
48
49         err = ngx_socket_errno;
50
51         if (n == 0) {
52             ngx_log_error(NGX_LOG_ALERT, c->log, err, "send() returned zero");
53             wev->ready = 0;
54             return n;
55         }
56
57         if (err == NGX_EAGAIN || err == NGX_EINTR) {
58             wev->ready = 0;
59
60             ngx_log_debug0(NGX_LOG_DEBUG_EVENT, c->log, err,
61                           "send() not ready");
62         }
63     }
```

```
63         if (err == NGX\_EAGAIN) {
64             return NGX\_AGAIN;
65         }
66
67     } else {
68         wev->error = 1;
69         (void) ngx\_connection\_error(c, err, "send() failed");
70         return NGX\_ERROR;
71     }
72 }
73 }
```

[One Level Up](#)

[Top Level](#)

## src/os/unix/nginx\_freebsd\_sendfile\_chain.c - nginx-1.7.10

### Functions defined

- [ngx\\_freebsd\\_sendfile\\_chain](#)

### Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 /*
14 * Although FreeBSD sendfile() allows to pass a header and a trailer,
15 * it cannot send a header with a part of the file in one packet until
16 * FreeBSD 5.3. Besides, over the fast ethernet connection sendfile()
17 * may send the partially filled packets, i.e. the 8 file pages may be sent
18 * as the 11 full 1460-bytes packets, then one incomplete 324-bytes packet,
19 * and then again the 11 full 1460-bytes packets.
20 *
21 * Therefore we use the TCP_NOPUSH option (similar to Linux's TCP_CORK)
22 * to postpone the sending - it not only sends a header and the first part of
23 * the file in one packet, but also sends the file pages in the full packets.
24 *
25 * But until FreeBSD 4.5 turning TCP_NOPUSH off does not flush a pending
26 * data that less than MSS, so that data may be sent with 5 second delay.
27 * So we do not use TCP_NOPUSH on FreeBSD prior to 4.5, although it can be used
28 * for non-keepalive HTTP connections.
29 */
30
31
32 ngx\_chain\_t *
33 ngx\_freebsd\_sendfile\_chain(ngx\_connection\_t *c, ngx\_chain\_t *in, off_t limit)
34 {
35     int                rc, flags;
36     off_t              send, prev_send, sent;
37     size_t             file_size;
38     ssize_t            n;
39     ngx\_uint\_t         eintr, eagain;
40     ngx\_err\_t         err;
41     ngx\_buf\_t         *file;
42     ngx\_event\_t       *wev;
43     ngx\_chain\_t       *cl;
44     ngx\_iovec\_t        header, trailer;
45     struct sf_hdrtr    hdtr;
46     struct iovec        headers[NGX\_IOVS\_PREALLOCATE];
47     struct iovec        trailers[NGX\_IOVS\_PREALLOCATE];
48
49     wev = c->write;
50
51     if (!wev->ready) {
52         return in;
53     }
54
55     #if (NGX_HAVE_KQUEUE)
56
57     if ((ngx\_event\_flags & NGX\_USE\_KQUEUE\_EVENT) && wev->pending_eof) {
58         (void) ngx\_connection\_error(c, wev->kq_errno,
59             "kevent() reported about an closed connection");
60         wev->error = 1;
61         return NGX\_CHAIN\_ERROR;
62     }
```



```

63
64 #endif
65
66 /* the maximum limit size is the maximum size_t value - the page size */
67
68 if (limit == 0 || limit > (off_t) (NGX_MAX_SIZE_T_VALUE - ngx_pagesize)) {
69     limit = NGX_MAX_SIZE_T_VALUE - ngx_pagesize;
70 }
71
72 send = 0;
73 eagain = 0;
74 flags = 0;
75
76 header.iovs = headers;
77 header.nalloc = NGX\_IOVS\_PREALLOCATE;
78
79 trailer.iovs = trailers;
80 trailer.nalloc = NGX\_IOVS\_PREALLOCATE;
81
82 for ( ;; ) {
83     eintr = 0;
84     prev_send = send;
85
86     /* create the header iovec and coalesce the neighbouring bufs */
87
88     cl = ngx\_output\_chain\_to\_iovec(&header, in, limit - send, c->log);
89
90     if (cl == NGX\_CHAIN\_ERROR) {
91         return NGX\_CHAIN\_ERROR;
92     }
93
94     send += header.size;
95
96     if (cl && cl->buf->in_file && send < limit) {
97         file = cl->buf;
98
99         /* coalesce the neighbouring file bufs */
100
101         file_size = (size_t) ngx\_chain\_coalesce\_file(&cl, limit - send);
102
103         send += file_size;
104
105         /* create the trailer iovec and coalesce the neighbouring bufs */
106
107         cl = ngx\_output\_chain\_to\_iovec(&trailer, cl, limit - send, c->log);
108
109         if (cl == NGX\_CHAIN\_ERROR) {
110             return NGX\_CHAIN\_ERROR;
111         }
112
113         send += trailer.size;
114
115         if (ngx\_freebsd\_use\_tcp\_nopush
116             && c->tcp_nopush == NGX\_TCP\_NOPUSH\_UNSET)
117         {
118             if (ngx\_tcp\_nopush(c->fd) == NGX\_ERROR) {
119                 err = ngx\_socket\_errno;
120
121                 /*
122                 * there is a tiny chance to be interrupted, however,
123                 * we continue a processing without the TCP_NOPUSH
124                 */
125
126                 if (err != NGX\_EINTR) {
127                     wev->error = 1;
128                     (void) ngx\_connection\_error(c, err,
129                                             ngx\_tcp\_nopush\_n " failed");
130                     return NGX\_CHAIN\_ERROR;
131                 }
132
133             } else {
134                 c->tcp_nopush = NGX\_TCP\_NOPUSH\_SET;
135
136                 ngx\_log\_debug0(NGX\_LOG\_DEBUG\_EVENT, c->log, 0,
137                             "tcp_nopush");
138             }
139         }
140     }
141 }

```

```

139     }
140
141     /*
142     * sendfile() does unneeded work if sf_hdr's count is 0,
143     * but corresponding pointer is not NULL
144     */
145
146     hdr.headers = header.count ? header.iovs : NULL;
147     hdr.hdr_cnt = header.count;
148     hdr.trailers = trailer.count ? trailer.iovs : NULL;
149     hdr.trl_cnt = trailer.count;
150
151     /*
152     * the "nbytes bug" of the old sendfile() syscall:
153     * http://bugs.freebsd.org/33771
154     */
155
156     if (!ngx_freebsd_sendfile_nbytes_bug) {
157         header.size = 0;
158     }
159
160     sent = 0;
161
162     #if (NGX_HAVE_AIO_SENDFILE)
163     flags = c->aio_sendfile ? SF_NODISKIO : 0;
164 #endif
165
166     rc = sendfile(file->file->fd, c->fd, file->file_pos,
167                 file_size + header.size, &hdr, &sent, flags);
168
169     if (rc == -1) {
170         err = ngx_errno;
171
172         switch (err) {
173             case NGX_EAGAIN:
174                 eagain = 1;
175                 break;
176
177             case NGX_EINTR:
178                 eintr = 1;
179                 break;
180
181             #if (NGX_HAVE_AIO_SENDFILE)
182             case NGX_EBUSY:
183                 c->busy_sendfile = file;
184                 break;
185 #endif
186
187             default:
188                 wev->error = 1;
189                 (void) ngx_connection_error(c, err, "sendfile() failed");
190                 return NGX_CHAIN_ERROR;
191         }
192
193         ngx_log_debug1(NGX_LOG_DEBUG_EVENT, c->log, err,
194                     "sendfile() sent only %0 bytes", sent);
195
196     /*
197     * sendfile() in FreeBSD 3.x-4.x may return value >= 0
198     * on success, although only 0 is documented
199     */
200
201     } else if (rc >= 0 && sent == 0) {
202
203         /*
204         * if rc is OK and sent equal to zero, then someone
205         * has truncated the file, so the offset became beyond
206         * the end of the file
207         */
208
209         ngx_log_error(NGX_LOG_ALERT, c->log, 0,
210                     "sendfile() reported that \"%s\" was truncated at %0",
211                     file->file->name.data, file->file_pos);
212
213         return NGX_CHAIN_ERROR;
214     }

```

```

215         ngx_log_debug4(NGX_LOG_DEBUG_EVENT, c->log, 0,
216             "sendfile: %d, @%O %O:%uz",
217             rc, file->file_pos, sent, file_size + header.size);
218
219     } else {
220         n = ngx_writev(c, &header);
221
222         if (n == NGX_ERROR) {
223             return NGX_CHAIN_ERROR;
224         }
225
226         sent = (n == NGX_AGAIN) ? 0 : n;
227     }
228
229     c->sent += sent;
230
231     in = ngx_chain_update_sent(in, sent);
232
233     #if (NGX_HAVE_AIO_SENDFILE)
234     if (c->busy_sendfile) {
235         return in;
236     }
237     #endif
238
239     if (eagain) {
240         /*
241          * sendfile() may return EAGAIN, even if it has sent a whole file
242          * part, it indicates that the successive sendfile() call would
243          * return EAGAIN right away and would not send anything.
244          * We use it as a hint.
245          */
246
247         wev->ready = 0;
248         return in;
249     }
250
251     if (eintr) {
252         send = prev_send + sent;
253         continue;
254     }
255
256     if (send - prev_send != sent) {
257         wev->ready = 0;
258         return in;
259     }
260
261     if (send >= limit || in == NULL) {
262         return in;
263     }
264 }
265 }
266 }
267 }

```

[One Level Up](#)

[Top Level](#)

## src/os/unix/nginx\_daemon.c - nginx-1.7.10

### Functions defined

- [ngx\\_daemon](#)

### Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10
11
12  ngx_int_t
13  ngx_daemon(ngx_log_t *log)
14  {
15      int  fd;
16
17      switch (fork()) {
18      case -1:
19          ngx_log_error(NGX_LOG_EMERG, log, ngx_errno, "fork() failed");
20          return NGX_ERROR;
21
22      case 0:
23          break;
24
25      default:
26          exit(0);
27      }
28
29      ngx_pid = ngx_getpid();
30
31      if (setsid() == -1) {
32          ngx_log_error(NGX_LOG_EMERG, log, ngx_errno, "setsid() failed");
33          return NGX_ERROR;
34      }
35
36      umask(0);
37
38      fd = open("/dev/null", O_RDWR);
39      if (fd == -1) {
40          ngx_log_error(NGX_LOG_EMERG, log, ngx_errno,
41                      "open(\"/dev/null\") failed");
42          return NGX_ERROR;
43      }
44
45      if (dup2(fd, STDIN_FILENO) == -1) {
46          ngx_log_error(NGX_LOG_EMERG, log, ngx_errno, "dup2(STDIN) failed");
47          return NGX_ERROR;
48      }
49
50      if (dup2(fd, STDOUT_FILENO) == -1) {
51          ngx_log_error(NGX_LOG_EMERG, log, ngx_errno, "dup2(STDOUT) failed");
52          return NGX_ERROR;
53      }
54
55      #if 0
56      if (dup2(fd, STDERR_FILENO) == -1) {
57          ngx_log_error(NGX_LOG_EMERG, log, ngx_errno, "dup2(STDERR) failed");
58          return NGX_ERROR;
59      }
60      #endif
61
62      if (fd > STDERR_FILENO) {
```

```
63     if (close(fd) == -1) {
64         ngx_log_error(NGX_LOG_EMERG, log, ngx_errno, "close() failed");
65         return NGX_ERROR;
66     }
67 }
68
69 return NGX_OK;
70 }
```

[One Level Up](#)

[Top Level](#)

# src/os/unix/nginx\_aio\_read\_chain.c - nginx-1.7.10

## Functions defined

- [ngx\\_aio\\_read\\_chain](#)

## Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 ssize_t
14 ngx_aio_read_chain(ngx_connection_t *c, ngx_chain_t *cl, off_t limit)
15 {
16     int          n;
17     u_char       *buf, *prev;
18     size_t       size;
19     ssize_t      total;
20
21     if (c->read->pending_eof) {
22         c->read->ready = 0;
23         return 0;
24     }
25
26     total = 0;
27
28     while (cl) {
29
30         /* we can post the single aio operation only */
31
32         if (!c->read->ready) {
33             return total ? total : NGX_AGAIN;
34         }
35
36         buf = cl->buf->last;
37         prev = cl->buf->last;
38         size = 0;
39
40         /* coalesce the neighbouring bufs */
41
42         while (cl && prev == cl->buf->last) {
43             size += cl->buf->end - cl->buf->last;
44             prev = cl->buf->end;
45             cl = cl->next;
46         }
47
48         n = ngx_aio_read(c, buf, size);
49
50         ngx_log_debug1(NGX_LOG_DEBUG_EVENT, c->log, 0, "aio_read: %d", n);
51
52         if (n == NGX_AGAIN) {
53             return total ? total : NGX_AGAIN;
54         }
55
56         if (n == NGX_ERROR) {
57             return NGX_ERROR;
58         }
59
60         if (n == 0) {
61             c->read->pending_eof = 1;
62             if (total) {
```

```
63         c->read->eof = 0;
64         c->read->ready = 1;
65     }
66     return total;
67 }
68
69 if (n > 0) {
70     total += n;
71 }
72
73     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, c->log, 0,
74         "aio_read total: %d", total);
75 }
76
77 return total ? total : NGX\_AGAIN;
78 }
```

[One Level Up](#)

[Top Level](#)

# src/os/unix/nginx\_aio\_write.c - nginx-1.7.10

## Functions defined

- [ngx\\_aio\\_write](#)

## Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 extern int  ngx_kqueue;
14
15
16 ssize_t
17 ngx_aio_write(ngx_connection_t *c, u_char *buf, size_t size)
18 {
19     int          n;
20     ngx_event_t *wev;
21
22     wev = c->write;
23
24     if (!wev->ready) {
25         return NGX_AGAIN;
26     }
27
28     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, wev->log, 0,
29                  "aio: wev->complete: %d", wev->complete);
30
31     if (!wev->complete) {
32         ngx_memzero(&wev->aiocb, sizeof(struct aiocb));
33
34         wev->aiocb.aio_fildes = c->fd;
35         wev->aiocb.aio_buf = buf;
36         wev->aiocb.aio_nbytes = size;
37
38         #if (NGX_HAVE_KQUEUE)
39             wev->aiocb.aio_sigevent.sigev_notify_kqueue = ngx_kqueue;
40             wev->aiocb.aio_sigevent.sigev_notify = SIGEV_KEVENT;
41             wev->aiocb.aio_sigevent.sigev_value.sigval_ptr = wev;
42         #endif
43
44         if (aio_write(&wev->aiocb) == -1) {
45             ngx_log_error(NGX_LOG_CRIT, wev->log, ngx_errno,
46                          "aio_write() failed");
47             return NGX_ERROR;
48         }
49
50         ngx_log_debug0(NGX_LOG_DEBUG_EVENT, wev->log, 0, "aio_write: OK");
51
52         wev->active = 1;
53         wev->ready = 0;
54     }
55
56     wev->complete = 0;
57
58     n = aio_error(&wev->aiocb);
59     if (n == -1) {
60         ngx_log_error(NGX_LOG_CRIT, wev->log, ngx_errno, "aio_error() failed");
61         wev->error = 1;
62         return NGX_ERROR;
63     }
64 }
```



```

63     }
64
65     if (n != 0) {
66         if (n == NGX_EINPROGRESS) {
67             if (wev->ready) {
68                 ngx_log_error(NGX_LOG_ALERT, wev->log, n,
69                     "aio_write() still in progress");
70                 wev->ready = 0;
71             }
72             return NGX_AGAIN;
73         }
74
75         ngx_log_error(NGX_LOG_CRIT, wev->log, n, "aio_write() failed");
76         wev->error = 1;
77         wev->ready = 0;
78
79         #if 1
80             n = aio_return(&wev->aioCb);
81             if (n == -1) {
82                 ngx_log_error(NGX_LOG_ALERT, wev->log, ngx_errno,
83                     "aio_return() failed");
84             }
85
86             ngx_log_error(NGX_LOG_CRIT, wev->log, n, "aio_return() %d", n);
87         #endif
88
89         return NGX_ERROR;
90     }
91
92     n = aio_return(&wev->aioCb);
93     if (n == -1) {
94         ngx_log_error(NGX_LOG_ALERT, wev->log, ngx_errno,
95             "aio_return() failed");
96
97         wev->error = 1;
98         wev->ready = 0;
99         return NGX_ERROR;
100    }
101
102
103    ngx_log_debug1(NGX_LOG_DEBUG_EVENT, wev->log, 0, "aio_write: %d", n);
104
105    wev->active = 0;
106    wev->ready = 1;
107
108    return n;
109 }

```

[One Level Up](#)

[Top Level](#)

# src/os/unix/nginx\_aio\_write\_chain.c - nginx-1.7.10

## Functions defined

- [ngx\\_aio\\_write\\_chain](#)

## Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 ngx_chain_t *
14 ngx_aio_write_chain(ngx_connection_t *c, ngx_chain_t *in, off_t limit)
15 {
16     u_char      *buf, *prev;
17     off_t        send, sent;
18     size_t       len;
19     ssize_t      n, size;
20     ngx_chain_t *cl;
21
22     /* the maximum limit size is the maximum size_t value - the page size */
23
24     if (limit == 0 || limit > (off_t) (NGX_MAX_SIZE_T_VALUE - ngx_pagesize)) {
25         limit = NGX_MAX_SIZE_T_VALUE - ngx_pagesize;
26     }
27
28     send = 0;
29     sent = 0;
30     cl = in;
31
32     while (cl) {
33
34         if (cl->buf->pos == cl->buf->last) {
35             cl = cl->next;
36             continue;
37         }
38
39         /* we can post the single aio operation only */
40
41         if (!c->write->ready) {
42             return cl;
43         }
44
45         buf = cl->buf->pos;
46         prev = buf;
47         len = 0;
48
49         /* coalesce the neighbouring bufs */
50
51         while (cl && prev == cl->buf->pos && send < limit) {
52             if (ngx_buf_special(cl->buf)) {
53                 continue;
54             }
55
56             size = cl->buf->last - cl->buf->pos;
57
58             if (send + size > limit) {
59                 size = limit - send;
60             }
61
62             len += size;
```

```

63     prev = cl->buf->pos + size;
64     send += size;
65     cl = cl->next;
66 }
67
68 n = ngx_aio_write(c, buf, len);
69
70 ngx_log_debug1(NGX_LOG_DEBUG_EVENT, c->log, 0, "aio_write: %z", n);
71
72 if (n == NGX_ERROR) {
73     return NGX_CHAIN_ERROR;
74 }
75
76 if (n > 0) {
77     sent += n;
78     c->sent += n;
79 }
80
81 ngx_log_debug1(NGX_LOG_DEBUG_EVENT, c->log, 0,
82     "aio_write sent: %0", c->sent);
83
84 for (cl = in; cl; cl = cl->next) {
85
86     if (sent >= cl->buf->last - cl->buf->pos) {
87         sent -= cl->buf->last - cl->buf->pos;
88         cl->buf->pos = cl->buf->last;
89
90         continue;
91     }
92
93     cl->buf->pos += sent;
94
95     break;
96 }
97 }
98
99 return cl;
100 }

```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_cpuid.c - nginx-1.7.10

### Functions defined

- [ngx\\_cpuid](#)
- [ngx\\_cpuid](#)
- [ngx\\_cpuid](#)
- [ngx\\_cpuid](#)

### Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10
11
12  #if ( ( __i386__ || __amd64__ ) && ( __GNUC__ || __INTEL_COMPILER ) )
13
14
15  static ngx_inline void ngx_cpuid(uint32_t i, uint32_t *buf);
16
17
18  #if ( __i386__ )
19
20  static ngx_inline void
21  ngx_cpuid(uint32_t i, uint32_t *buf)
22  {
23
24      /*
25       * we could not use %ebx as output parameter if gcc builds PIC,
26       * and we could not save %ebx on stack, because %esp is used,
27       * when the -fomit-frame-pointer optimization is specified.
28       */
29
30      __asm__ (
31
32          "    mov    %%ebx, %%esi;  "
33
34          "    cpuid;                "
35          "    mov    %%eax, (%1);  "
36          "    mov    %%ebx, 4(%1); "
37          "    mov    %%edx, 8(%1); "
38          "    mov    %%ecx, 12(%1); "
39
40          "    mov    %%esi, %%ebx; "
41
42          : : "a" (i), "D" (buf) : "ecx", "edx", "esi", "memory" );
43  }
44
45
46  #else /* __amd64__ */
47
48
49  static ngx_inline void
50  ngx_cpuid(uint32_t i, uint32_t *buf)
51  {
52      uint32_t  eax, ebx, ecx, edx;
53
54      __asm__ (
55
56          "cpuid"
```

```

57 : "=a" (eax), "=b" (ebx), "=c" (ecx), "=d" (edx) : "a" (i) );
58
59
60 buf[0] = eax;
61 buf[1] = ebx;
62 buf[2] = edx;
63 buf[3] = ecx;
64 }
65
66
67 #endif
68
69
70 /* auto detect the L2 cache line size of modern and widespread CPUs */
71
72 void
73 ngx_cpuinfo(void)
74 {
75     u_char    *vendor;
76     uint32_t  vbuf[5], cpu[4], model;
77
78     vbuf[0] = 0;
79     vbuf[1] = 0;
80     vbuf[2] = 0;
81     vbuf[3] = 0;
82     vbuf[4] = 0;
83
84     ngx_cpuid(0, vbuf);
85
86     vendor = (u_char *) &vbuf[1];
87
88     if (vbuf[0] == 0) {
89         return;
90     }
91
92     ngx_cpuid(1, cpu);
93
94     if (ngx_strcmp(vendor, "GenuineIntel") == 0) {
95
96         switch ((cpu[0] & 0xf00) >> 8) {
97
98             /* Pentium */
99             case 5:
100                 ngx_cacheline_size = 32;
101                 break;
102
103             /* Pentium Pro, II, III */
104             case 6:
105                 ngx_cacheline_size = 32;
106
107                 model = ((cpu[0] & 0xf0000) >> 8) | (cpu[0] & 0xf0);
108
109                 if (model >= 0xd0) {
110                     /* Intel Core, Core 2, Atom */
111                     ngx_cacheline_size = 64;
112                 }
113
114                 break;
115
116             /*
117             * Pentium 4, although its cache line size is 64 bytes,
118             * it prefetches up to two cache lines during memory read
119             */
120             case 15:
121                 ngx_cacheline_size = 128;
122                 break;
123         }
124
125     } else if (ngx_strcmp(vendor, "AuthenticAMD") == 0) {
126         ngx_cacheline_size = 64;
127     }
128 }
129
130 #else
131
132

```

```
133 void
134 ngx_cpuserinfo(void)
135 {
136 }
137
138
139 #endif
```

[One Level Up](#)

[Top Level](#)

## src/os/unix/nginx\_time.c - nginx-1.7.10

### Functions defined

- [ngx\\_libc\\_gmtime](#)
- [ngx\\_libc\\_localtime](#)
- [ngx\\_localtime](#)
- [ngx\\_timezone\\_update](#)

### Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10
11
12  /*
13  * FreeBSD does not test /etc/localtime change, however, we can workaround it
14  * by calling tzset() with TZ and then without TZ to update timezone.
15  * The trick should work since FreeBSD 2.1.0.
16  *
17  * Linux does not test /etc/localtime change in localtime(),
18  * but may stat("/etc/localtime") several times in every strftime(),
19  * therefore we use it to update timezone.
20  *
21  * Solaris does not test /etc/TIMEZONE change too and no workaround available.
22  */
23
24 void
25 ngx_timezone_update(void)
26 {
27     #if (NGX_FREEBSD)
28
29         if (getenv("TZ")) {
30             return;
31         }
32
33         putenv("TZ=UTC");
34
35         tzset();
36
37         unsetenv("TZ");
38
39         tzset();
40
41     #elif (NGX_LINUX)
42         time_t    s;
43         struct tm *t;
44         char      buf[4];
45
46         s = time(0);
47
48         t = localtime(&s);
49
50         strftime(buf, 4, "%H", t);
51
52     #endif
53 }
54
55 void
```

```
57 ngx_localtime(time_t s, ngx_tm_t *tm)
58 {
59     #if (NGX_HAVE_LOCALTIME_R)
60         (void) localtime_r(&s, tm);
61
62     #else
63         ngx_tm_t *t;
64
65         t = localtime(&s);
66         *tm = *t;
67
68     #endif
69
70     tm->ngx_tm_mon++;
71     tm->ngx_tm_year += 1900;
72 }
73
74
75 void
76 ngx_libc_localtime(time_t s, struct tm *tm)
77 {
78     #if (NGX_HAVE_LOCALTIME_R)
79         (void) localtime_r(&s, tm);
80
81     #else
82         struct tm *t;
83
84         t = localtime(&s);
85         *tm = *t;
86
87     #endif
88 }
89
90
91 void
92 ngx_libc_gmtime(time_t s, struct tm *tm)
93 {
94     #if (NGX_HAVE_LOCALTIME_R)
95         (void) gmtime_r(&s, tm);
96
97     #else
98         struct tm *t;
99
100        t = gmtime(&s);
101        *tm = *t;
102
103    #endif
104 }
```

[One Level Up](#)

[Top Level](#)



## src/http/modules/nginx\_http\_limit\_conn\_module.c - nginx-1.7.10

### Global variables defined

- [ngx\\_http\\_limit\\_conn\\_commands](#)
- [ngx\\_http\\_limit\\_conn\\_log\\_levels](#)
- [ngx\\_http\\_limit\\_conn\\_module](#)
- [ngx\\_http\\_limit\\_conn\\_module\\_ctx](#)
- [ngx\\_http\\_limit\\_conn\\_status\\_bounds](#)

### Data types defined

- [ngx\\_http\\_limit\\_conn\\_cleanup\\_t](#)
- [ngx\\_http\\_limit\\_conn\\_conf\\_t](#)
- [ngx\\_http\\_limit\\_conn\\_ctx\\_t](#)
- [ngx\\_http\\_limit\\_conn\\_limit\\_t](#)
- [ngx\\_http\\_limit\\_conn\\_node\\_t](#)

### Functions defined

- [ngx\\_http\\_limit\\_conn](#)
- [ngx\\_http\\_limit\\_conn\\_cleanup](#)
- [ngx\\_http\\_limit\\_conn\\_cleanup\\_all](#)
- [ngx\\_http\\_limit\\_conn\\_create\\_conf](#)
- [ngx\\_http\\_limit\\_conn\\_handler](#)
- [ngx\\_http\\_limit\\_conn\\_init](#)
- [ngx\\_http\\_limit\\_conn\\_init\\_zone](#)
- [ngx\\_http\\_limit\\_conn\\_lookup](#)
- [ngx\\_http\\_limit\\_conn\\_merge\\_conf](#)
- [ngx\\_http\\_limit\\_conn\\_rbtrees\\_insert\\_value](#)
- [ngx\\_http\\_limit\\_conn\\_zone](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
```

```

11
12
13 typedef struct {
14     u_char                color;
15     u_char                len;
16     u_short              conn;
17     u_char                data[1];
18 } ngx_http_limit_conn_node_t;
19
20
21 typedef struct {
22     ngx_shm_zone_t        *shm_zone;
23     ngx_rbtree_node_t     *node;
24 } ngx_http_limit_conn_cleanup_t;
25
26
27 typedef struct {
28     ngx_rbtree_t          *rbtree;
29     ngx_http_complex_value_t key;
30 } ngx_http_limit_conn_ctx_t;
31
32
33 typedef struct {
34     ngx_shm_zone_t        *shm_zone;
35     ngx_uint_t            conn;
36 } ngx_http_limit_conn_limit_t;
37
38
39 typedef struct {
40     ngx_array_t           limits;
41     ngx_uint_t            log_level;
42     ngx_uint_t            status_code;
43 } ngx_http_limit_conn_conf_t;
44
45
46 static ngx_rbtree_node_t *ngx_http_limit_conn_lookup(ngx_rbtree_t *rbtree,
47     ngx_str_t *key, uint32_t hash);
48 static void ngx_http_limit_conn_cleanup(void *data);
49 static ngx_inline void ngx_http_limit_conn_cleanup_all(ngx_pool_t *pool);
50
51 static void *ngx_http_limit_conn_create_conf(ngx_conf_t *cf);
52 static char *ngx_http_limit_conn_merge_conf(ngx_conf_t *cf, void *parent,
53     void *child);
54 static char *ngx_http_limit_conn_zone(ngx_conf_t *cf, ngx_command_t *cmd,
55     void *conf);
56 static char *ngx_http_limit_conn(ngx_conf_t *cf, ngx_command_t *cmd,
57     void *conf);
58 static ngx_int_t ngx_http_limit_conn_init(ngx_conf_t *cf);
59
60
61 static ngx_conf_enum_t ngx_http_limit_conn_log_levels[] = {
62     { ngx_string("info"), NGX_LOG_INFO },
63     { ngx_string("notice"), NGX_LOG_NOTICE },
64     { ngx_string("warn"), NGX_LOG_WARN },
65     { ngx_string("error"), NGX_LOG_ERR },
66     { ngx_null_string, 0 }
67 };
68
69
70 static ngx_conf_num_bounds_t ngx_http_limit_conn_status_bounds = {
71     ngx_conf_check_num_bounds, 400, 599
72 };
73
74
75 static ngx_command_t ngx_http_limit_conn_commands[] = {
76
77     { ngx_string("limit_conn_zone"),
78     NGX_HTTP_MAIN_CONF|NGX_CONF_TAKE2,
79     ngx_http_limit_conn_zone,
80     0,
81     0,
82     NULL },
83
84     { ngx_string("limit_conn"),
85     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE2,
86     ngx_http_limit_conn,

```

```

87     NGX\_HTTP\_LOC\_CONF\_OFFSET,
88     0,
89     NULL },
90
91     { ngx\_string\("limit\_conn\_log\_level"\),
92       NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
93       ngx\_conf\_set\_enum\_slot,
94       NGX\_HTTP\_LOC\_CONF\_OFFSET,
95       offsetof\(ngx\_http\_limit\_conn\_conf\_t, log\_level\),
96       &ngx\_http\_limit\_conn\_log\_levels },
97
98     { ngx\_string\("limit\_conn\_status"\),
99       NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
100      ngx\_conf\_set\_num\_slot,
101      NGX\_HTTP\_LOC\_CONF\_OFFSET,
102      offsetof\(ngx\_http\_limit\_conn\_conf\_t, status\_code\),
103      &ngx\_http\_limit\_conn\_status\_bounds },
104
105      ngx\_null\_command
106 };
107
108
109 static ngx\_http\_module\_t ngx\_http\_limit\_conn\_module\_ctx = {
110     NULL,                                /* preconfiguration */
111     ngx\_http\_limit\_conn\_init,           /* postconfiguration */
112
113     NULL,                                /* create main configuration */
114     NULL,                                /* init main configuration */
115
116     NULL,                                /* create server configuration */
117     NULL,                                /* merge server configuration */
118
119     ngx\_http\_limit\_conn\_create\_conf,      /* create location configuration */
120     ngx\_http\_limit\_conn\_merge\_conf        /* merge location configuration */
121 };
122
123
124 ngx\_module\_t ngx\_http\_limit\_conn\_module = {
125     NGX\_MODULE\_V1,
126     &ngx\_http\_limit\_conn\_module\_ctx,      /* module context */
127     ngx\_http\_limit\_conn\_commands,        /* module directives */
128     NGX\_HTTP\_MODULE,                    /* module type */
129     NULL,                                /* init master */
130     NULL,                                /* init module */
131     NULL,                                /* init process */
132     NULL,                                /* init thread */
133     NULL,                                /* exit thread */
134     NULL,                                /* exit process */
135     NULL,                                /* exit master */
136     NGX\_MODULE\_V1\_PADDING
137 };
138
139
140 static ngx\_int\_t
141 ngx\_http\_limit\_conn\_handler(ngx\_http\_request\_t *r)
142 {
143     size\_t                n;
144     uint32\_t             hash;
145     ngx\_str\_t            key;
146     ngx\_uint\_t           i;
147     ngx\_slab\_pool\_t      *shpool;
148     ngx\_rbtrees\_node\_t   *node;
149     ngx\_pool\_cleanup\_t   *cln;
150     ngx\_http\_limit\_conn\_ctx\_t *ctx;
151     ngx\_http\_limit\_conn\_node\_t *lc;
152     ngx\_http\_limit\_conn\_conf\_t *lccf;
153     ngx\_http\_limit\_conn\_limit\_t *limits;
154     ngx\_http\_limit\_conn\_cleanup\_t *lccln;
155
156     if (r->main->limit_conn_set) {
157         return NGX\_DECLINED;
158     }
159
160     lccf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_limit\_conn\_module);
161     limits = lccf->limits.elts;
162

```

```

163 for (i = 0; i < lccf->limits.nelts; i++) {
164     ctx = limits[i].shm_zone->data;
165
166     if (ngx_http_complex_value(r, &ctx->key, &key) != NGX_OK) {
167         return NGX_HTTP_INTERNAL_SERVER_ERROR;
168     }
169
170     if (key.len == 0) {
171         continue;
172     }
173
174     if (key.len > 255) {
175         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
176             "the value of the \"%V\" key "
177             "is more than 255 bytes: \"%V\"",
178             &ctx->key.value, &key);
179         continue;
180     }
181
182     r->main->limit_conn_set = 1;
183
184     hash = ngx_crc32_short(key.data, key.len);
185
186     shpool = (ngx_slab_pool_t *) limits[i].shm_zone->shm.addr;
187
188     ngx_shmtx_lock(&shpool->mutex);
189
190     node = ngx_http_limit_conn_lookup(ctx->rbtree, &key, hash);
191
192     if (node == NULL) {
193
194         n = offsetof(ngx_rbtree_node_t, color)
195             + offsetof(ngx_http_limit_conn_node_t, data)
196             + key.len;
197
198         node = ngx_slab_alloc_locked(shpool, n);
199
200         if (node == NULL) {
201             ngx_shmtx_unlock(&shpool->mutex);
202             ngx_http_limit_conn_cleanup_all(r->pool);
203             return lccf->status_code;
204         }
205
206         lc = (ngx_http_limit_conn_node_t *) &node->color;
207
208         node->key = hash;
209         lc->len = (u_char) key.len;
210         lc->conn = 1;
211         ngx_memcpy(lc->data, key.data, key.len);
212
213         ngx_rbtree_insert(ctx->rbtree, node);
214     } else {
215
216         lc = (ngx_http_limit_conn_node_t *) &node->color;
217
218         if ((ngx_uint_t) lc->conn >= limits[i].conn) {
219
220             ngx_shmtx_unlock(&shpool->mutex);
221
222             ngx_log_error(lccf->log_level, r->connection->log, 0,
223                 "limiting connections by zone \"%V\"",
224                 &limits[i].shm_zone->shm.name);
225
226             ngx_http_limit_conn_cleanup_all(r->pool);
227             return lccf->status_code;
228         }
229     }
230
231     lc->conn++;
232 }
233
234 ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
235     "limit conn: %08XD %d", node->key, lc->conn);
236
237 ngx_shmtx_unlock(&shpool->mutex);
238

```

```

239     cln = ngx_pool_cleanup_add(r->pool,
240                               sizeof(ngx_http_limit_conn_cleanup_t));
241     if (cln == NULL) {
242         return NGX_HTTP_INTERNAL_SERVER_ERROR;
243     }
244
245     cln->handler = ngx_http_limit_conn_cleanup;
246     lccln = cln->data;
247
248     lccln->shm_zone = limits[i].shm_zone;
249     lccln->node = node;
250 }
251
252 return NGX_DECLINED;
253 }
254
255
256 static void
257 ngx_http_limit_conn_rbtree_insert_value(ngx_rbtree_node_t *temp,
258 ngx_rbtree_node_t *node, ngx_rbtree_node_t *sentinel)
259 {
260     ngx_rbtree_node_t **p;
261     ngx_http_limit_conn_node_t *lcn, *lcnt;
262
263     for ( ;; ) {
264
265         if (node->key < temp->key) {
266             p = &temp->left;
267
268         } else if (node->key > temp->key) {
269             p = &temp->right;
270
271         } else { /* node->key == temp->key */
272
273             lcn = (ngx_http_limit_conn_node_t *) &node->color;
274             lcnt = (ngx_http_limit_conn_node_t *) &temp->color;
275
276             p = (ngx_memn2cmp(lcn->data, lcnt->data, lcn->len, lcnt->len) < 0)
277                 ? &temp->left : &temp->right;
278         }
279
280         if (*p == sentinel) {
281             break;
282         }
283
284         temp = *p;
285     }
286
287     *p = node;
288     node->parent = temp;
289     node->left = sentinel;
290     node->right = sentinel;
291     ngx_rbt_red(node);
292 }
293
294
295
296
297 static ngx_rbtree_node_t *
298 ngx_http_limit_conn_lookup(ngx_rbtree_t *rbtree, ngx_str_t *key, uint32_t hash)
299 {
300     ngx_int_t rc;
301     ngx_rbtree_node_t *node, *sentinel;
302     ngx_http_limit_conn_node_t *lcn;
303
304     node = rbtree->root;
305     sentinel = rbtree->sentinel;
306
307     while (node != sentinel) {
308
309         if (hash < node->key) {
310             node = node->left;
311             continue;
312         }
313
314         if (hash > node->key) {

```

```

315         node = node->right;
316         continue;
317     }
318
319     /* hash == node->key */
320
321     lcn = (ngx_http_limit_conn_node_t *) &node->color;
322
323     rc = ngx_memn2cmp(key->data, lcn->data, key->len, (size_t) lcn->len);
324
325     if (rc == 0) {
326         return node;
327     }
328
329     node = (rc < 0) ? node->left : node->right;
330 }
331
332 return NULL;
333 }
334
335
336 static void
337 ngx_http_limit_conn_cleanup(void *data)
338 {
339     ngx_http_limit_conn_cleanup_t *lccln = data;
340
341     ngx_slab_pool_t *shpool;
342     ngx_rbtree_node_t *node;
343     ngx_http_limit_conn_ctx_t *ctx;
344     ngx_http_limit_conn_node_t *lc;
345
346     ctx = lccln->shm_zone->data;
347     shpool = (ngx_slab_pool_t *) lccln->shm_zone->shm.addr;
348     node = lccln->node;
349     lc = (ngx_http_limit_conn_node_t *) &node->color;
350
351     ngx_shmtx_lock(&shpool->mutex);
352
353     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, lccln->shm_zone->shm.log, 0,
354                  "limit conn cleanup: %08XD %d", node->key, lc->conn);
355
356     lc->conn--;
357
358     if (lc->conn == 0) {
359         ngx_rbtree_delete(ctx->rbtree, node);
360         ngx_slab_free_locked(shpool, node);
361     }
362
363     ngx_shmtx_unlock(&shpool->mutex);
364 }
365
366
367 static ngx_inline void
368 ngx_http_limit_conn_cleanup_all(ngx_pool_t *pool)
369 {
370     ngx_pool_cleanup_t *cln;
371
372     cln = pool->cleanup;
373
374     while (cln && cln->handler == ngx_http_limit_conn_cleanup) {
375         ngx_http_limit_conn_cleanup(cln->data);
376         cln = cln->next;
377     }
378
379     pool->cleanup = cln;
380 }
381
382
383 static ngx_int_t
384 ngx_http_limit_conn_init_zone(ngx_shm_zone_t *shm_zone, void *data)
385 {
386     ngx_http_limit_conn_ctx_t *octx = data;
387
388     size_t len;
389     ngx_slab_pool_t *shpool;
390     ngx_rbtree_node_t *sentinel;

```

```

391 ngx\_http\_limit\_conn\_ctx\_t *ctx;
392
393 ctx = shm_zone->data;
394
395 if (octx) {
396     if (ctx->key.value.len != octx->key.value.len
397         || ngx\_strncmp(ctx->key.value.data, octx->key.value.data,
398             ctx->key.value.len)
399             != 0)
400     {
401         ngx\_log\_error(NGX\_LOG\_EMERG, shm_zone->shm.log, 0,
402             "limit_conn_zone \"%V\" uses the \"%V\" key "
403             "while previously it used the \"%V\" key",
404             &shm_zone->shm.name, &ctx->key.value,
405             &octx->key.value);
406         return NGX\_ERROR;
407     }
408
409     ctx->rbtree = octx->rbtree;
410
411     return NGX\_OK;
412 }
413
414 shpool = (ngx\_slab\_pool\_t *) shm_zone->shm.addr;
415
416 if (shm_zone->shm.exists) {
417     ctx->rbtree = shpool->data;
418
419     return NGX\_OK;
420 }
421
422 ctx->rbtree = ngx\_slab\_alloc(shpool, sizeof(ngx\_rbtree\_t));
423 if (ctx->rbtree == NULL) {
424     return NGX\_ERROR;
425 }
426
427 shpool->data = ctx->rbtree;
428
429 sentinel = ngx\_slab\_alloc(shpool, sizeof(ngx\_rbtree\_node\_t));
430 if (sentinel == NULL) {
431     return NGX\_ERROR;
432 }
433
434 ngx\_rbtree\_init(ctx->rbtree, sentinel,
435     ngx\_http\_limit\_conn\_rbtree\_insert\_value);
436
437 len = sizeof(" in limit_conn_zone \"%\"") + shm_zone->shm.name.len;
438
439 shpool->log_ctx = ngx\_slab\_alloc(shpool, len);
440 if (shpool->log_ctx == NULL) {
441     return NGX\_ERROR;
442 }
443
444 ngx\_sprintf(shpool->log_ctx, " in limit_conn_zone \"%V\"%Z",
445     &shm_zone->shm.name);
446
447 return NGX\_OK;
448 }
449
450 static void *
451 ngx\_http\_limit\_conn\_create\_conf(ngx\_conf\_t *cf)
452 {
453     ngx\_http\_limit\_conn\_conf\_t *conf;
454
455     conf = ngx\_palloc(cf->pool, sizeof(ngx\_http\_limit\_conn\_conf\_t));
456     if (conf == NULL) {
457         return NULL;
458     }
459
460     /*
461     * set by ngx\_palloc():
462     *
463     *     conf->limits.elts = NULL;
464     */
465 }
466

```

```

467     conf->log_level = NGX\_CONF\_UNSET\_UINT;
468     conf->status_code = NGX\_CONF\_UNSET\_UINT;
469
470     return conf;
471 }
472
473
474 static char *
475 ngx_http_limit_conn_merge_conf(ngx\_conf\_t *cf, void *parent, void *child)
476 {
477     ngx\_http\_limit\_conn\_conf\_t *prev = parent;
478     ngx\_http\_limit\_conn\_conf\_t *conf = child;
479
480     if (conf->limits.elts == NULL) {
481         conf->limits = prev->limits;
482     }
483
484     ngx\_conf\_merge\_uint\_value(conf->log_level, prev->log_level, NGX\_LOG\_ERR);
485     ngx\_conf\_merge\_uint\_value(conf->status_code, prev->status_code,
486                             NGX\_HTTP\_SERVICE\_UNAVAILABLE);
487
488     return NGX\_CONF\_OK;
489 }
490
491
492 static char *
493 ngx_http_limit_conn_zone(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
494 {
495     u_char                *p;
496     ssize_t               size;
497     ngx\_str\_t              value, name, s;
498     ngx\_uint\_t             i;
499     ngx\_shm\_zone\_t        *shm_zone;
500     ngx\_http\_limit\_conn\_ctx\_t *ctx;
501     ngx\_http\_compile\_complex\_value\_t ccv;
502
503     value = cf->args->elts;
504
505     ctx = ngx\_palloc(cf->pool, sizeof(ngx\_http\_limit\_conn\_ctx\_t));
506     if (ctx == NULL) {
507         return NGX\_CONF\_ERROR;
508     }
509
510     ngx\_memzero(&ccv, sizeof(ngx\_http\_compile\_complex\_value\_t));
511
512     ccv.cf = cf;
513     ccv.value = &value[1];
514     ccv.complex_value = &ctx->key;
515
516     if (ngx\_http\_compile\_complex\_value(&ccv) != NGX\_OK) {
517         return NGX\_CONF\_ERROR;
518     }
519
520     size = 0;
521     name.len = 0;
522
523     for (i = 2; i < cf->args->nelts; i++) {
524
525         if (ngx\_strncmp(value[i].data, "zone=", 5) == 0) {
526
527             name.data = value[i].data + 5;
528
529             p = (u_char *) ngx\_strchr(name.data, ':');
530
531             if (p == NULL) {
532                 ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
533                                 "invalid zone size \"%V\"", &value[i]);
534                 return NGX\_CONF\_ERROR;
535             }
536
537             name.len = p - name.data;
538
539             s.data = p + 1;
540             s.len = value[i].data + value[i].len - s.data;
541
542             size = ngx\_parse\_size(&s);

```



```

543     if (size == NGX\_ERROR) {
544         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
545             "invalid zone size \"%V\"", &value[i]);
546         return NGX\_CONF\_ERROR;
547     }
548 }
549
550     if (size < (ssize_t) (8 * ngx\_pagesize)) {
551         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
552             "zone \"%V\" is too small", &value[i]);
553         return NGX\_CONF\_ERROR;
554     }
555
556     continue;
557 }
558
559     ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
560         "invalid parameter \"%V\"", &value[i]);
561     return NGX\_CONF\_ERROR;
562 }
563
564 if (name.len == 0) {
565     ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
566         "\"%V\" must have \"zone\" parameter",
567         &cmd->name);
568     return NGX\_CONF\_ERROR;
569 }
570
571 shm_zone = ngx\_shared\_memory\_add(cf, &name, size,
572     &ngx\_http\_limit\_conn\_module);
573 if (shm_zone == NULL) {
574     return NGX\_CONF\_ERROR;
575 }
576
577 if (shm_zone->data) {
578     ctx = shm_zone->data;
579
580     ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
581         "%V \"%V\" is already bound to key \"%V\"",
582         &cmd->name, &name, &ctx->key.value);
583     return NGX\_CONF\_ERROR;
584 }
585
586 shm_zone->init = ngx\_http\_limit\_conn\_init\_zone;
587 shm_zone->data = ctx;
588
589 return NGX\_CONF\_OK;
590 }
591
592
593 static char *
594 ngx\_http\_limit\_conn(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
595 {
596     ngx\_shm\_zone\_t *shm_zone;
597     ngx\_http\_limit\_conn\_conf\_t *lccf = conf;
598     ngx\_http\_limit\_conn\_limit\_t *limit, *limits;
599
600     ngx\_str\_t *value;
601     ngx\_int\_t n;
602     ngx\_uint\_t i;
603
604     value = cf->args->elts;
605
606     shm_zone = ngx\_shared\_memory\_add(cf, &value[1], 0,
607         &ngx\_http\_limit\_conn\_module);
608     if (shm_zone == NULL) {
609         return NGX\_CONF\_ERROR;
610     }
611
612     limits = lccf->limits.elts;
613
614     if (limits == NULL) {
615         if (ngx\_array\_init(&lccf->limits, cf->pool, 1,
616             sizeof(ngx\_http\_limit\_conn\_limit\_t))
617             != NGX\_OK)
618             {

```

```

619         return NGX\_CONF\_ERROR;
620     }
621 }
622
623 for (i = 0; i < lccf->limits.nelts; i++) {
624     if (shm_zone == limits[i].shm_zone) {
625         return "is duplicate";
626     }
627 }
628
629 n = ngx\_atoi(value[2].data, value[2].len);
630 if (n <= 0) {
631     ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
632         "invalid number of connections \"%V\"", &value[2]);
633     return NGX\_CONF\_ERROR;
634 }
635
636 if (n > 65535) {
637     ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
638         "connection limit must be less 65536");
639     return NGX\_CONF\_ERROR;
640 }
641
642 limit = ngx\_array\_push(&lccf->limits);
643 if (limit == NULL) {
644     return NGX\_CONF\_ERROR;
645 }
646
647 limit->conn = n;
648 limit->shm_zone = shm_zone;
649
650 return NGX\_CONF\_OK;
651 }
652
653
654 static ngx\_int\_t
655 ngx\_http\_limit\_conn\_init(ngx\_conf\_t *cf)
656 {
657     ngx\_http\_handler\_pt *h;
658     ngx\_http\_core\_main\_conf\_t *cmcf;
659
660     cmcf = ngx\_http\_conf\_get\_module\_main\_conf(cf, ngx\_http\_core\_module);
661
662     h = ngx\_array\_push(&cmcf->phases[NGX\_HTTP\_PREACCESS\_PHASE].handlers);
663     if (h == NULL) {
664         return NGX\_ERROR;
665     }
666
667     *h = ngx\_http\_limit\_conn\_handler;
668
669     return NGX\_OK;
670 }

```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_limit\_req\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_limit\\_req\\_commands](#)
- [ngx\\_http\\_limit\\_req\\_log\\_levels](#)
- [ngx\\_http\\_limit\\_req\\_module](#)
- [ngx\\_http\\_limit\\_req\\_module\\_ctx](#)
- [ngx\\_http\\_limit\\_req\\_status\\_bounds](#)

## Data types defined

- [ngx\\_http\\_limit\\_req\\_conf\\_t](#)
- [ngx\\_http\\_limit\\_req\\_ctx\\_t](#)
- [ngx\\_http\\_limit\\_req\\_limit\\_t](#)
- [ngx\\_http\\_limit\\_req\\_node\\_t](#)
- [ngx\\_http\\_limit\\_req\\_shctx\\_t](#)

## Functions defined

- [ngx\\_http\\_limit\\_req](#)
- [ngx\\_http\\_limit\\_req\\_account](#)
- [ngx\\_http\\_limit\\_req\\_create\\_conf](#)
- [ngx\\_http\\_limit\\_req\\_delay](#)
- [ngx\\_http\\_limit\\_req\\_expire](#)
- [ngx\\_http\\_limit\\_req\\_handler](#)
- [ngx\\_http\\_limit\\_req\\_init](#)
- [ngx\\_http\\_limit\\_req\\_init\\_zone](#)
- [ngx\\_http\\_limit\\_req\\_lookup](#)
- [ngx\\_http\\_limit\\_req\\_merge\\_conf](#)
- [ngx\\_http\\_limit\\_req\\_rbtree\\_insert\\_value](#)
- [ngx\\_http\\_limit\\_req\\_zone](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
```

```

9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 typedef struct {
14     u_char                color;
15     u_char                dummy;
16     u_short              len;
17     ngx_queue_t          queue;
18     ngx_msec_t           last;
19     /* integer value, 1 corresponds to 0.001 r/s */
20     ngx_uint_t           excess;
21     ngx_uint_t           count;
22     u_char                data[1];
23 } ngx_http_limit_req_node_t;
24
25
26 typedef struct {
27     ngx_rbtree_t          rbtree;
28     ngx_rbtree_node_t    sentinel;
29     ngx_queue_t          queue;
30 } ngx_http_limit_req_shctx_t;
31
32
33 typedef struct {
34     ngx_http_limit_req_shctx_t *sh;
35     ngx_slab_pool_t         *shpool;
36     /* integer value, 1 corresponds to 0.001 r/s */
37     ngx_uint_t              rate;
38     ngx_http_complex_value_t key;
39     ngx_http_limit_req_node_t *node;
40 } ngx_http_limit_req_ctx_t;
41
42
43 typedef struct {
44     ngx_shm_zone_t         *shm_zone;
45     /* integer value, 1 corresponds to 0.001 r/s */
46     ngx_uint_t             burst;
47     ngx_uint_t             nodelay; /* unsigned nodelay:1 */
48 } ngx_http_limit_req_limit_t;
49
50
51 typedef struct {
52     ngx_array_t            limits;
53     ngx_uint_t             limit_log_level;
54     ngx_uint_t             delay_log_level;
55     ngx_uint_t             status_code;
56 } ngx_http_limit_req_conf_t;
57
58
59 static void ngx_http_limit_req_delay(ngx_http_request_t *r);
60 static ngx_int_t ngx_http_limit_req_lookup(ngx_http_limit_req_limit_t *limit,
61     ngx_uint_t hash, ngx_str_t *key, ngx_uint_t *ep, ngx_uint_t account);
62 static ngx_msec_t ngx_http_limit_req_account(ngx_http_limit_req_limit_t *limits,
63     ngx_uint_t n, ngx_uint_t *ep, ngx_http_limit_req_limit_t **limit);
64 static void ngx_http_limit_req_expire(ngx_http_limit_req_ctx_t *ctx,
65     ngx_uint_t n);
66
67 static void *ngx_http_limit_req_create_conf(ngx_conf_t *cf);
68 static char *ngx_http_limit_req_merge_conf(ngx_conf_t *cf, void *parent,
69     void *child);
70 static char *ngx_http_limit_req_zone(ngx_conf_t *cf, ngx_command_t *cmd,
71     void *conf);
72 static char *ngx_http_limit_req(ngx_conf_t *cf, ngx_command_t *cmd,
73     void *conf);
74 static ngx_int_t ngx_http_limit_req_init(ngx_conf_t *cf);
75
76
77 static ngx_conf_enum_t ngx_http_limit_req_log_levels[] = {
78     { ngx_string("info"), NGX_LOG_INFO },
79     { ngx_string("notice"), NGX_LOG_NOTICE },
80     { ngx_string("warn"), NGX_LOG_WARN },
81     { ngx_string("error"), NGX_LOG_ERR },
82     { ngx_null_string, 0 }
83 };
84

```

```

85
86 static ngx_conf_num_bounds_t ngx_http_limit_req_status_bounds = {
87     ngx_conf_check_num_bounds, 400, 599
88 };
89
90
91 static ngx_command_t ngx_http_limit_req_commands[] = {
92
93     { ngx_string("limit_req_zone"),
94       NGX_HTTP_MAIN_CONF|NGX_CONF_TAKE3,
95       ngx_http_limit_req_zone,
96       0,
97       0,
98       NULL },
99
100    { ngx_string("limit_req"),
101      NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE123,
102      ngx_http_limit_req,
103      NGX_HTTP_LOC_CONF_OFFSET,
104      0,
105      NULL },
106
107    { ngx_string("limit_req_log_level"),
108      NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
109      ngx_conf_set_enum_slot,
110      NGX_HTTP_LOC_CONF_OFFSET,
111      offsetof(ngx_http_limit_req_conf_t, limit_log_level),
112      &ngx_http_limit_req_log_levels },
113
114    { ngx_string("limit_req_status"),
115      NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
116      ngx_conf_set_num_slot,
117      NGX_HTTP_LOC_CONF_OFFSET,
118      offsetof(ngx_http_limit_req_conf_t, status_code),
119      &ngx_http_limit_req_status_bounds },
120
121    ngx_null_command
122 };
123
124
125 static ngx_http_module_t ngx_http_limit_req_module_ctx = {
126     NULL, /* preconfiguration */
127     ngx_http_limit_req_init, /* postconfiguration */
128
129     NULL, /* create main configuration */
130     NULL, /* init main configuration */
131
132     NULL, /* create server configuration */
133     NULL, /* merge server configuration */
134
135     ngx_http_limit_req_create_conf, /* create location configuration */
136     ngx_http_limit_req_merge_conf /* merge location configuration */
137 };
138
139
140 ngx_module_t ngx_http_limit_req_module = {
141     NGX_MODULE_V1,
142     &ngx_http_limit_req_module_ctx, /* module context */
143     ngx_http_limit_req_commands, /* module directives */
144     NGX_HTTP_MODULE, /* module type */
145     NULL, /* init master */
146     NULL, /* init module */
147     NULL, /* init process */
148     NULL, /* init thread */
149     NULL, /* exit thread */
150     NULL, /* exit process */
151     NULL, /* exit master */
152     NGX_MODULE_V1_PADDING
153 };
154
155
156 static ngx_int_t
157 ngx_http_limit_req_handler(ngx_http_request_t *r)
158 {
159     uint32_t hash;
160     ngx_str_t key;

```

```

161     ngx_int_t          rc;
162     ngx_uint_t         n, excess;
163     ngx_msec_t         delay;
164     ngx_http_limit_req_ctx_t *ctx;
165     ngx_http_limit_req_conf_t *lrcf;
166     ngx_http_limit_req_limit_t *limit, *limits;
167
168     if (r->main->limit_req_set) {
169         return NGX_DECLINED;
170     }
171
172     lrcf = ngx_http_get_module_loc_conf(r, ngx_http_limit_req_module);
173     limits = lrcf->limits.elts;
174
175     excess = 0;
176
177     rc = NGX_DECLINED;
178
179     #if (NGX_SUPPRESS_WARN)
180         limit = NULL;
181     #endif
182
183     for (n = 0; n < lrcf->limits.nelts; n++) {
184
185         limit = &limits[n];
186
187         ctx = limit->shm_zone->data;
188
189         if (ngx_http_complex_value(r, &ctx->key, &key) != NGX_OK) {
190             return NGX_HTTP_INTERNAL_SERVER_ERROR;
191         }
192
193         if (key.len == 0) {
194             continue;
195         }
196
197         if (key.len > 65535) {
198             ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
199                 "the value of the \"%V\" key "
200                 "is more than 65535 bytes: \"%V\"",
201                 &ctx->key.value, &key);
202             continue;
203         }
204
205         hash = ngx_crc32_short(key.data, key.len);
206
207         ngx_shmtx_lock(&ctx->shpool->mutex);
208
209         rc = ngx_http_limit_req_lookup(limit, hash, &key, &excess,
210             (n == lrcf->limits.nelts - 1));
211
212         ngx_shmtx_unlock(&ctx->shpool->mutex);
213
214         ngx_log_debug4(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
215             "limit_req[%ui]: %i %ui.%03ui",
216             n, rc, excess / 1000, excess % 1000);
217
218         if (rc != NGX_AGAIN) {
219             break;
220         }
221     }
222
223     if (rc == NGX_DECLINED) {
224         return NGX_DECLINED;
225     }
226
227     r->main->limit_req_set = 1;
228
229     if (rc == NGX_BUSY || rc == NGX_ERROR) {
230
231         if (rc == NGX_BUSY) {
232             ngx_log_error(lrcf->limit_log_level, r->connection->log, 0,
233                 "limiting requests, excess: %ui.%03ui by zone \"%V\"",
234                 excess / 1000, excess % 1000,
235                 &limit->shm_zone->shm.name);
236         }

```

```

237     while (n--) {
238         ctx = limits[n].shm_zone->data;
239
240         if (ctx->node == NULL) {
241             continue;
242         }
243
244         ngx_shmtx_lock(&ctx->shpool->mutex);
245
246         ctx->node->count--;
247
248         ngx_shmtx_unlock(&ctx->shpool->mutex);
249
250         ctx->node = NULL;
251     }
252
253     return lrcf->status_code;
254 }
255
256 /* rc == NGX_AGAIN || rc == NGX_OK */
257
258 if (rc == NGX_AGAIN) {
259     excess = 0;
260 }
261
262 delay = ngx_http_limit_req_account(limits, n, &excess, &limit);
263
264 if (!delay) {
265     return NGX_DECLINED;
266 }
267
268 ngx_log_error(lrcf->delay_log_level, r->connection->log, 0,
269             "delaying request, excess: %ui.%03ui, by zone \"%V\"",
270             excess / 1000, excess % 1000, &limit->shm_zone->shm.name);
271
272 if (ngx_handle_read_event(r->connection->read, 0) != NGX_OK) {
273     return NGX_HTTP_INTERNAL_SERVER_ERROR;
274 }
275
276 r->read_event_handler = ngx_http_test_reading;
277 r->write_event_handler = ngx_http_limit_req_delay;
278 ngx_add_timer(r->connection->write, delay);
279
280 return NGX_AGAIN;
281 }
282
283
284
285 static void
286 ngx_http_limit_req_delay(ngx_http_request_t *r)
287 {
288     ngx_event_t *wev;
289
290     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
291                 "limit_req delay");
292
293     wev = r->connection->write;
294
295     if (!wev->timedout) {
296         if (ngx_handle_write_event(wev, 0) != NGX_OK) {
297             ngx_http_finalize_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
298         }
299
300         return;
301     }
302
303     wev->timedout = 0;
304
305     if (ngx_handle_read_event(r->connection->read, 0) != NGX_OK) {
306         ngx_http_finalize_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
307         return;
308     }
309
310     r->read_event_handler = ngx_http_block_reading;
311     r->write_event_handler = ngx_http_core_run_phases;

```

```

313     ngx\_http\_core\_run\_phases(r);
314 }
315
316
317
318 static void
319 ngx\_http\_limit\_req\_rbtree\_insert\_value(ngx\_rbtree\_node\_t *temp,
320 ngx\_rbtree\_node\_t *node, ngx\_rbtree\_node\_t *sentinel)
321 {
322     ngx\_rbtree\_node\_t      **p;
323     ngx\_http\_limit\_req\_node\_t  *lrn, *lrnt;
324
325     for ( ;; ) {
326
327         if (node->key < temp->key) {
328
329             p = &temp->left;
330
331         } else if (node->key > temp->key) {
332
333             p = &temp->right;
334
335         } else { /* node->key == temp->key */
336
337             lrn = (ngx\_http\_limit\_req\_node\_t *) &node->color;
338             lrnt = (ngx\_http\_limit\_req\_node\_t *) &temp->color;
339
340             p = (ngx\_memn2cmp(lrn->data, lrnt->data, lrn->len, lrnt->len) < 0)
341                 ? &temp->left : &temp->right;
342         }
343
344         if (*p == sentinel) {
345             break;
346         }
347
348         temp = *p;
349     }
350
351     *p = node;
352     node->parent = temp;
353     node->left = sentinel;
354     node->right = sentinel;
355     ngx\_rbt\_red(node);
356 }
357
358
359 static ngx\_int\_t
360 ngx\_http\_limit\_req\_lookup(ngx\_http\_limit\_req\_limit\_t *limit, ngx\_uint\_t hash,
361 ngx\_str\_t *key, ngx\_uint\_t *ep, ngx\_uint\_t account)
362 {
363     size\_t                size;
364     ngx\_int\_t            rc, excess;
365     ngx\_time\_t          *tp;
366     ngx\_msec\_t          now;
367     ngx\_msec\_int\_t      ms;
368     ngx\_rbtree\_node\_t   *node, *sentinel;
369     ngx\_http\_limit\_req\_ctx\_t *ctx;
370     ngx\_http\_limit\_req\_node\_t *lr;
371
372     tp = ngx\_timeofday();
373     now = (ngx\_msec\_t) (tp->sec * 1000 + tp->msec);
374
375     ctx = limit->shm_zone->data;
376
377     node = ctx->sh->rbtree.root;
378     sentinel = ctx->sh->rbtree.sentinel;
379
380     while (node != sentinel) {
381
382         if (hash < node->key) {
383             node = node->left;
384             continue;
385         }
386
387         if (hash > node->key) {
388             node = node->right;

```



```

389     continue;
390 }
391
392 /* hash == node->key */
393
394 lr = (ngx\_http\_limit\_req\_node\_t *) &node->color;
395
396 rc = ngx\_memn2cmp(key->data, lr->data, key->len, (size\_t) lr->len);
397
398 if (rc == 0) {
399     ngx\_queue\_remove(&lr->queue);
400     ngx\_queue\_insert\_head(&ctx->sh->queue, &lr->queue);
401
402     ms = (ngx\_msec\_int\_t) (now - lr->last);
403
404     excess = lr->excess - ctx->rate * ngx\_abs(ms) / 1000 + 1000;
405
406     if (excess < 0) {
407         excess = 0;
408     }
409
410     *ep = excess;
411
412     if ((ngx\_uint\_t) excess > limit->burst) {
413         return NGX\_BUSY;
414     }
415
416     if (account) {
417         lr->excess = excess;
418         lr->last = now;
419         return NGX\_OK;
420     }
421
422     lr->count++;
423
424     ctx->node = lr;
425
426     return NGX\_AGAIN;
427 }
428
429 node = (rc < 0) ? node->left : node->right;
430 }
431
432 *ep = 0;
433
434 size = offsetof(ngx\_rbtree\_node\_t, color)
435     + offsetof(ngx\_http\_limit\_req\_node\_t, data)
436     + key->len;
437
438 ngx\_http\_limit\_req\_expire(ctx, 1);
439
440 node = ngx\_slab\_alloc\_locked(ctx->shpool, size);
441
442 if (node == NULL) {
443     ngx\_http\_limit\_req\_expire(ctx, 0);
444
445     node = ngx\_slab\_alloc\_locked(ctx->shpool, size);
446     if (node == NULL) {
447         ngx\_log\_error(NGX\_LOG\_ALERT, ngx\_cycle->log, 0,
448             "could not allocate node%s", ctx->shpool->log_ctx);
449         return NGX\_ERROR;
450     }
451 }
452
453 node->key = hash;
454
455 lr = (ngx\_http\_limit\_req\_node\_t *) &node->color;
456
457 lr->len = (u\_short) key->len;
458 lr->excess = 0;
459
460 ngx\_memcpy(lr->data, key->data, key->len);
461
462 ngx\_rbtree\_insert(&ctx->sh->rbtree, node);
463
464 ngx\_queue\_insert\_head(&ctx->sh->queue, &lr->queue);

```

```

465     if (account) {
466         lr->last = now;
467         lr->count = 0;
468         return NGX_OK;
469     }
470
471     lr->last = 0;
472     lr->count = 1;
473
474     ctx->node = lr;
475
476     return NGX_AGAIN;
477 }
478
479
480
481 static ngx_msec_t
482 ngx_http_limit_req_account(ngx_http_limit_req_limit_t *limits, ngx_uint_t n,
483     ngx_uint_t *ep, ngx_http_limit_req_limit_t **limit)
484 {
485     ngx_int_t          excess;
486     ngx_time_t         *tp;
487     ngx_msec_t         now, delay, max_delay;
488     ngx_msec_int_t     ms;
489     ngx_http_limit_req_ctx_t *ctx;
490     ngx_http_limit_req_node_t *lr;
491
492     excess = *ep;
493
494     if (excess == 0 || (*limit)->nodelay) {
495         max_delay = 0;
496     }
497     else {
498         ctx = (*limit)->shm_zone->data;
499         max_delay = excess * 1000 / ctx->rate;
500     }
501
502     while (n--) {
503         ctx = limits[n].shm_zone->data;
504         lr = ctx->node;
505
506         if (lr == NULL) {
507             continue;
508         }
509
510         ngx_shmtx_lock(&ctx->shpool->mutex);
511
512         tp = ngx_timeofday();
513
514         now = (ngx_msec_t) (tp->sec * 1000 + tp->msec);
515         ms = (ngx_msec_int_t) (now - lr->last);
516
517         excess = lr->excess - ctx->rate * ngx_abs(ms) / 1000 + 1000;
518
519         if (excess < 0) {
520             excess = 0;
521         }
522
523         lr->last = now;
524         lr->excess = excess;
525         lr->count--;
526
527         ngx_shmtx_unlock(&ctx->shpool->mutex);
528
529         ctx->node = NULL;
530
531         if (limits[n].nodelay) {
532             continue;
533         }
534
535         delay = excess * 1000 / ctx->rate;
536
537         if (delay > max_delay) {
538             max_delay = delay;
539             *ep = excess;
540             *limit = &limits[n];

```

```

541     }
542 }
543
544     return max_delay;
545 }
546
547
548 static void
549 ngx_http_limit_req_expire(ngx_http_limit_req_ctx_t *ctx, ngx_uint_t n)
550 {
551     ngx_int_t             excess;
552     ngx_time_t           *tp;
553     ngx_msec_t           now;
554     ngx_queue_t          *q;
555     ngx_msec_int_t       ms;
556     ngx_rbtrees_node_t   *node;
557     ngx_http_limit_req_node_t *lr;
558
559     tp = ngx_timeofday();
560
561     now = (ngx_msec_t) (tp->sec * 1000 + tp->msec);
562
563     /*
564      * n == 1 deletes one or two zero rate entries
565      * n == 0 deletes oldest entry by force
566      *           and one or two zero rate entries
567      */
568
569     while (n < 3) {
570
571         if (ngx_queue_empty(&ctx->sh->queue)) {
572             return;
573         }
574
575         q = ngx_queue_last(&ctx->sh->queue);
576
577         lr = ngx_queue_data(q, ngx_http_limit_req_node_t, queue);
578
579         if (lr->count) {
580
581             /*
582              * There is not much sense in looking further,
583              * because we bump nodes on the lookup stage.
584              */
585
586             return;
587         }
588
589         if (n++ != 0) {
590
591             ms = (ngx_msec_int_t) (now - lr->last);
592             ms = ngx_abs(ms);
593
594             if (ms < 60000) {
595                 return;
596             }
597
598             excess = lr->excess - ctx->rate * ms / 1000;
599
600             if (excess > 0) {
601                 return;
602             }
603         }
604
605         ngx_queue_remove(q);
606
607         node = (ngx_rbtrees_node_t *)
608             ((u_char *) lr - offsetof(ngx_rbtrees_node_t, color));
609
610         ngx_rbtrees_delete(&ctx->sh->rbtree, node);
611
612         ngx_slab_free_locked(ctx->shpool, node);
613     }
614 }
615
616

```

```

617 static ngx_int_t
618 ngx_http_limit_req_init_zone(ngx_shm_zone_t *shm_zone, void *data)
619 {
620     ngx_http_limit_req_ctx_t *octx = data;
621
622     size_t len;
623     ngx_http_limit_req_ctx_t *ctx;
624
625     ctx = shm_zone->data;
626
627     if (octx) {
628         if (ctx->key.value.len != octx->key.value.len
629             || ngx_strcmp(ctx->key.value.data, octx->key.value.data,
630                ctx->key.value.len)
631                != 0)
632         {
633             ngx_log_error(NGX_LOG_EMERG, shm_zone->shm.log, 0,
634                "limit_req \"%V\" uses the \"%V\" key "
635                "while previously it used the \"%V\" key",
636                &shm_zone->shm.name, &ctx->key.value,
637                &octx->key.value);
638             return NGX_ERROR;
639         }
640
641         ctx->sh = octx->sh;
642         ctx->shpool = octx->shpool;
643
644         return NGX_OK;
645     }
646
647     ctx->shpool = (ngx_slab_pool_t *) shm_zone->shm.addr;
648
649     if (shm_zone->shm.exists) {
650         ctx->sh = ctx->shpool->data;
651
652         return NGX_OK;
653     }
654
655     ctx->sh = ngx_slab_alloc(ctx->shpool, sizeof(ngx_http_limit_req_shctx_t));
656     if (ctx->sh == NULL) {
657         return NGX_ERROR;
658     }
659
660     ctx->shpool->data = ctx->sh;
661
662     ngx_rbtrees_init(&ctx->sh->rbtree, &ctx->sh->sentinel,
663        ngx_http_limit_req_rbtrees_insert_value);
664
665     ngx_queues_init(&ctx->sh->queue);
666
667     len = sizeof(" in limit_req zone \\\"") + shm_zone->shm.name.len;
668
669     ctx->shpool->log_ctx = ngx_slab_alloc(ctx->shpool, len);
670     if (ctx->shpool->log_ctx == NULL) {
671         return NGX_ERROR;
672     }
673
674     ngx_sprintf(ctx->shpool->log_ctx, " in limit_req zone \"%V\"%Z",
675        &shm_zone->shm.name);
676
677     ctx->shpool->log_nomem = 0;
678
679     return NGX_OK;
680 }
681
682
683 static void *
684 ngx_http_limit_req_create_conf(ngx_conf_t *cf)
685 {
686     ngx_http_limit_req_conf_t *conf;
687
688     conf = ngx_palloc(cf->pool, sizeof(ngx_http_limit_req_conf_t));
689     if (conf == NULL) {
690         return NULL;
691     }
692

```

```

693  /*
694  * set by ngx\_palloc\(\):
695  *
696  *     conf->limits.elts = NULL;
697  */
698
699  conf->limit_log_level = NGX\_CONF\_UNSET\_UINT;
700  conf->status_code = NGX\_CONF\_UNSET\_UINT;
701
702  return conf;
703 }
704
705
706 static char *
707 ngx_http_limit_req_merge_conf(ngx\_conf\_t *cf, void *parent, void *child)
708 {
709     ngx\_http\_limit\_req\_conf\_t *prev = parent;
710     ngx\_http\_limit\_req\_conf\_t *conf = child;
711
712     if (conf->limits.elts == NULL) {
713         conf->limits = prev->limits;
714     }
715
716     ngx\_conf\_merge\_uint\_value(conf->limit_log_level, prev->limit_log_level,
717                             NGX\_LOG\_ERR);
718
719     conf->delay_log_level = (conf->limit_log_level == NGX\_LOG\_INFO) ?
720                          NGX\_LOG\_INFO : conf->limit_log_level + 1;
721
722     ngx\_conf\_merge\_uint\_value(conf->status_code, prev->status_code,
723                             NGX\_HTTP\_SERVICE\_UNAVAILABLE);
724
725     return NGX\_CONF\_OK;
726 }
727
728
729 static char *
730 ngx_http_limit_req_zone(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
731 {
732     u_char                *p;
733     size_t                len;
734     ssize_t               size;
735     ngx\_str\_t             *value, name, s;
736     ngx\_int\_t             rate, scale;
737     ngx\_uint\_t            i;
738     ngx\_shm\_zone\_t       *shm_zone;
739     ngx\_http\_limit\_req\_ctx\_t *ctx;
740     ngx\_http\_compile\_complex\_value\_t ccv;
741
742     value = cf->args->elts;
743
744     ctx = ngx\_palloc(cf->pool, sizeof(ngx\_http\_limit\_req\_ctx\_t));
745     if (ctx == NULL) {
746         return NGX\_CONF\_ERROR;
747     }
748
749     ngx\_memzero(&ccv, sizeof(ngx\_http\_compile\_complex\_value\_t));
750
751     ccv.cf = cf;
752     ccv.value = &value[1];
753     ccv.complex_value = &ctx->key;
754
755     if (ngx\_http\_compile\_complex\_value(&ccv) != NGX\_OK) {
756         return NGX\_CONF\_ERROR;
757     }
758
759     size = 0;
760     rate = 1;
761     scale = 1;
762     name.len = 0;
763
764     for (i = 2; i < cf->args->nelts; i++) {
765
766         if (ngx\_strncmp(value[i].data, "zone=", 5) == 0) {
767
768             name.data = value[i].data + 5;

```

```

769     p = (u_char *) ngx_strchr(name.data, ':');
770
771
772     if (p == NULL) {
773         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
774             "invalid zone size \"%V\"", &value[i]);
775         return NGX_CONF_ERROR;
776     }
777
778     name.len = p - name.data;
779
780     s.data = p + 1;
781     s.len = value[i].data + value[i].len - s.data;
782
783     size = ngx_parse_size(&s);
784
785     if (size == NGX_ERROR) {
786         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
787             "invalid zone size \"%V\"", &value[i]);
788         return NGX_CONF_ERROR;
789     }
790
791     if (size < (ssize_t) (8 * ngx_pagesize)) {
792         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
793             "zone \"%V\" is too small", &value[i]);
794         return NGX_CONF_ERROR;
795     }
796
797     continue;
798 }
799
800 if (ngx_strncmp(value[i].data, "rate=", 5) == 0) {
801
802     len = value[i].len;
803     p = value[i].data + len - 3;
804
805     if (ngx_strncmp(p, "r/s", 3) == 0) {
806         scale = 1;
807         len -= 3;
808
809     } else if (ngx_strncmp(p, "r/m", 3) == 0) {
810         scale = 60;
811         len -= 3;
812     }
813
814     rate = ngx_atoi(value[i].data + 5, len - 5);
815     if (rate <= 0) {
816         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
817             "invalid rate \"%V\"", &value[i]);
818         return NGX_CONF_ERROR;
819     }
820
821     continue;
822 }
823
824 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
825     "invalid parameter \"%V\"", &value[i]);
826 return NGX_CONF_ERROR;
827 }
828
829 if (name.len == 0) {
830     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
831         "\"%V\" must have \"zone\" parameter",
832         &cmd->name);
833     return NGX_CONF_ERROR;
834 }
835
836 ctx->rate = rate * 1000 / scale;
837
838 shm_zone = ngx_shared_memory_add(cf, &name, size,
839     &ngx_http_limit_req_module);
840 if (shm_zone == NULL) {
841     return NGX_CONF_ERROR;
842 }
843
844 if (shm_zone->data) {

```

```

845     ctx = shm_zone->data;
846
847     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
848         "%V \"%V\" is already bound to key \"%V\"",
849         &cmd->name, &name, &ctx->key.value);
850     return NGX_CONF_ERROR;
851 }
852
853 shm_zone->init = ngx_http_limit_req_init_zone;
854 shm_zone->data = ctx;
855
856 return NGX_CONF_OK;
857 }
858
859
860 static char *
861 ngx_http_limit_req(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
862 {
863     ngx_http_limit_req_conf_t *lrcf = conf;
864
865     ngx_int_t          burst;
866     ngx_str_t          *value, s;
867     ngx_uint_t         i, nodelay;
868     ngx_shm_zone_t     *shm_zone;
869     ngx_http_limit_req_limit_t *limit, *limits;
870
871     value = cf->args->elts;
872
873     shm_zone = NULL;
874     burst = 0;
875     nodelay = 0;
876
877     for (i = 1; i < cf->args->nelts; i++) {
878
879         if (ngx_strncmp(value[i].data, "zone=", 5) == 0) {
880
881             s.len = value[i].len - 5;
882             s.data = value[i].data + 5;
883
884             shm_zone = ngx_shared_memory_add(cf, &s, 0,
885                 &ngx_http_limit_req_module);
886             if (shm_zone == NULL) {
887                 return NGX_CONF_ERROR;
888             }
889
890             continue;
891         }
892
893         if (ngx_strncmp(value[i].data, "burst=", 6) == 0) {
894
895             burst = ngx_atoi(value[i].data + 6, value[i].len - 6);
896             if (burst <= 0) {
897                 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
898                     "invalid burst rate \"%V\"", &value[i]);
899                 return NGX_CONF_ERROR;
900             }
901
902             continue;
903         }
904
905         if (ngx_strcmp(value[i].data, "nodelay") == 0) {
906             nodelay = 1;
907             continue;
908         }
909
910         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
911             "invalid parameter \"%V\"", &value[i]);
912         return NGX_CONF_ERROR;
913     }
914
915     if (shm_zone == NULL) {
916         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
917             "\"%V\" must have \"zone\" parameter",
918             &cmd->name);
919         return NGX_CONF_ERROR;
920     }

```

```

921
922 if (shm_zone->data == NULL) {
923     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
924         "unknown limit_req_zone \"%V\"",
925         &shm_zone->shm.name);
926     return NGX_CONF_ERROR;
927 }
928
929 limits = lrcf->limits.elts;
930
931 if (limits == NULL) {
932     if (ngx_array_init(&lrcf->limits, cf->pool, 1,
933         sizeof(ngx_http_limit_req_limit_t))
934         != NGX_OK)
935     {
936         return NGX_CONF_ERROR;
937     }
938 }
939
940 for (i = 0; i < lrcf->limits.nelts; i++) {
941     if (shm_zone == limits[i].shm_zone) {
942         return "is duplicate";
943     }
944 }
945
946 limit = ngx_array_push(&lrcf->limits);
947 if (limit == NULL) {
948     return NGX_CONF_ERROR;
949 }
950
951 limit->shm_zone = shm_zone;
952 limit->burst = burst * 1000;
953 limit->nodelay = nodelay;
954
955 return NGX_CONF_OK;
956 }
957
958
959 static ngx_int_t
960 ngx_http_limit_req_init(ngx_conf_t *cf)
961 {
962     ngx_http_handler_pt *h;
963     ngx_http_core_main_conf_t *cmcf;
964
965     cmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_core_module);
966
967     h = ngx_array_push(&cmcf->phases[NGX_HTTP_PREACCESS_PHASE].handlers);
968     if (h == NULL) {
969         return NGX_ERROR;
970     }
971
972     *h = ngx_http_limit_req_handler;
973
974     return NGX_OK;
975 }

```

[One Level Up](#)

[Top Level](#)



# src/http/nginx\_http\_busy\_lock.c - nginx-1.7.10

## Functions defined

- [ngx\\_http\\_busy\\_lock](#)
- [ngx\\_http\\_busy\\_lock\\_cacheable](#)
- [ngx\\_http\\_busy\\_lock\\_look\\_cacheable](#)
- [ngx\\_http\\_busy\\_unlock](#)
- [ngx\\_http\\_set\\_busy\\_lock\\_slot](#)

## Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13
14 static int ngx_http_busy_lock_look_cacheable(ngx_http_busy_lock_t *bl,
15                                             ngx_http_busy_lock_ctx_t *bc,
16                                             int lock);
17
18
19 int ngx_http_busy_lock(ngx_http_busy_lock_t *bl, ngx_http_busy_lock_ctx_t *bc)
20 {
21     if (bl->busy < bl->max_busy) {
22         bl->busy++;
23
24         if (bc->time) {
25             bc->time = 0;
26             bl->waiting--;
27         }
28
29         return NGX_OK;
30     }
31
32     if (bc->time) {
33         if (bc->time < bl->timeout) {
34             ngx_add_timer(bc->event, 1000);
35             return NGX_AGAIN;
36         }
37
38         bl->waiting--;
39         return NGX_DONE;
40     }
41
42     if (bl->timeout == 0) {
43         return NGX_DONE;
44     }
45
46     if (bl->waiting < bl->max_waiting) {
47         bl->waiting++;
48
49         #if 0
50             ngx_add_timer(bc->event, 1000);
51             bc->event->event_handler = bc->event_handler;
52         #endif
53     }
54 }
```

```

54     /* TODO: ngx_handle_level_read_event() */
55
56     return NGX_AGAIN;
57 }
58
59 return NGX_ERROR;
60 }
61
62
63
64 int ngx_http_busy_lock_cacheable(ngx_http_busy_lock_t *bl,
65                                 ngx_http_busy_lock_ctx_t *bc, int lock)
66 {
67     int rc;
68
69     rc = ngx_http_busy_lock_look_cacheable(bl, bc, lock);
70
71     ngx_log_debug3(NGX_LOG_DEBUG_HTTP, bc->event->log, 0,
72                  "http busylock: %d w:%d mw:%d",
73                  rc, bl->waiting, bl->max_waiting);
74
75     if (rc == NGX_OK) { /* no the same request, there's free slot */
76         return NGX_OK;
77     }
78
79     if (rc == NGX_ERROR && !lock) { /* no the same request, no free slot */
80         return NGX_OK;
81     }
82
83     /* rc == NGX_AGAIN: the same request */
84
85     if (bc->time) {
86         if (bc->time < bl->timeout) {
87             ngx_add_timer(bc->event, 1000);
88             return NGX_AGAIN;
89         }
90
91         bl->waiting--;
92         return NGX_DONE;
93     }
94
95     if (bl->timeout == 0) {
96         return NGX_DONE;
97     }
98
99     if (bl->waiting < bl->max_waiting) {
100 #if 0
101     bl->waiting++;
102     ngx_add_timer(bc->event, 1000);
103     bc->event->event_handler = bc->event_handler;
104 #endif
105
106     /* TODO: ngx_handle_level_read_event() */
107
108     return NGX_AGAIN;
109 }
110
111 return NGX_ERROR;
112 }
113
114
115
116 void ngx_http_busy_unlock(ngx_http_busy_lock_t *bl,
117                          ngx_http_busy_lock_ctx_t *bc)
118 {
119     if (bl == NULL) {
120         return;
121     }
122
123     if (bl->md5) {
124         bl->md5_mask[bc->slot / 8] &= ~(1 << (bc->slot & 7));
125         bl->cacheable--;
126     }
127
128     bl->busy--;
129 }

```

```

130
131
132 static int ngx_http_busy_lock_look_cacheable(ngx_http_busy_lock_t *bl,
133                                             ngx_http_busy_lock_ctx_t *bc,
134                                             int lock)
135 {
136     int    i, b, cacheable, free;
137     u_int  mask;
138
139     b = 0;
140     cacheable = 0;
141     free = -1;
142
143     #if (NGX_SUPPRESS_WARN)
144     mask = 0;
145     #endif
146
147     for (i = 0; i < bl->max_busy; i++) {
148
149         if ((b & 7) == 0) {
150             mask = bl->md5_mask[i / 8];
151         }
152
153         if (mask & 1) {
154             if (ngx_memcmp(&bl->md5[i * 16], bc->md5, 16) == 0) {
155                 return NGX_AGAIN;
156             }
157             cacheable++;
158
159         } else if (free == -1) {
160             free = i;
161         }
162
163         #if 1
164         if (cacheable == bl->cacheable) {
165             if (free == -1 && cacheable < bl->max_busy) {
166                 free = i + 1;
167             }
168
169             break;
170         }
171         #endif
172
173         mask >>= 1;
174         b++;
175     }
176
177     if (free == -1) {
178         return NGX_ERROR;
179     }
180
181     if (lock) {
182         if (bl->busy == bl->max_busy) {
183             return NGX_ERROR;
184         }
185
186         ngx_memcpy(&bl->md5[free * 16], bc->md5, 16);
187         bl->md5_mask[free / 8] |= 1 << (free & 7);
188         bc->slot = free;
189
190         bl->cacheable++;
191         bl->busy++;
192     }
193
194     return NGX_OK;
195 }
196
197
198 char *ngx_http_set_busy_lock_slot(ngx_conf_t *cf, ngx_command_t *cmd,
199                                 void *conf)
200 {
201     char *p = conf;
202
203     ngx_uint_t    i, dup, invalid;
204     ngx_str_t     *value, line;
205     ngx_http_busy_lock_t *bl, **blp;

```

```

206
207 blp = (ngx\_http\_busy\_lock\_t **) (p + cmd->offset);
208 if (*blp) {
209     return "is duplicate";
210 }
211
212 /* ngx_calloc_shared() */
213 bl = ngx\_pcalloc(cf->pool, sizeof(ngx\_http\_busy\_lock\_t));
214 if (bl == NULL) {
215     return NGX\_CONF\_ERROR;
216 }
217 *blp = bl;
218
219 /* ngx_calloc_shared() */
220 bl->mutex = ngx\_pcalloc(cf->pool, sizeof(ngx\_event\_mutex\_t));
221 if (bl->mutex == NULL) {
222     return NGX\_CONF\_ERROR;
223 }
224
225 dup = 0;
226 invalid = 0;
227 value = cf->args->elts;
228
229 for (i = 1; i < cf->args->nelts; i++) {
230
231     if (value[i].data[1] != '=') {
232         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
233             "invalid value \"%s\"", value[i].data);
234         return NGX\_CONF\_ERROR;
235     }
236
237     switch (value[i].data[0]) {
238
239     case 'b':
240         if (bl->max_busy) {
241             dup = 1;
242             break;
243         }
244
245         bl->max_busy = ngx\_atoi(value[i].data + 2, value[i].len - 2);
246         if (bl->max_busy == NGX\_ERROR) {
247             invalid = 1;
248             break;
249         }
250
251         continue;
252
253     case 'w':
254         if (bl->max_waiting) {
255             dup = 1;
256             break;
257         }
258
259         bl->max_waiting = ngx\_atoi(value[i].data + 2, value[i].len - 2);
260         if (bl->max_waiting == NGX\_ERROR) {
261             invalid = 1;
262             break;
263         }
264
265         continue;
266
267     case 't':
268         if (bl->timeout) {
269             dup = 1;
270             break;
271         }
272
273         line.len = value[i].len - 2;
274         line.data = value[i].data + 2;
275
276         bl->timeout = ngx\_parse\_time(&line, 1);
277         if (bl->timeout == (time_t) NGX\_ERROR) {
278             invalid = 1;
279             break;
280         }
281

```

```
282         continue;
283
284     default:
285         invalid = 1;
286     }
287
288     if (dup) {
289         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
290             "duplicate value \"%s\"", value[i].data);
291         return NGX_CONF_ERROR;
292     }
293
294     if (invalid) {
295         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
296             "invalid value \"%s\"", value[i].data);
297         return NGX_CONF_ERROR;
298     }
299 }
300
301 if (bl->timeout == 0 && bl->max_waiting) {
302     ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
303         "busy lock waiting is useless with zero timeout, ignoring");
304 }
305
306 return NGX_CONF_OK;
307 }
```

[One Level Up](#)

[Top Level](#)

# src/http/nginx\_http\_busy\_lock.h - nginx-1.7.10

## Data types defined

- [ngx\\_http\\_busy\\_lock\\_ctx\\_t](#)
- [ngx\\_http\\_busy\\_lock\\_t](#)

## Macros defined

- [\\_NGX\\_HTTP\\_BUSY\\_LOCK\\_H\\_INCLUDED](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_HTTP_BUSY_LOCK_H_INCLUDED
9 #define _NGX_HTTP_BUSY_LOCK_H_INCLUDED
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14 #include <ngx_event.h>
15 #include <ngx_http.h>
16
17
18 typedef struct {
19     u_char      *md5_mask;
20     char        *md5;
21     int         cacheable;
22
23     int         busy;
24     int         max_busy;
25
26     int         waiting;
27     int         max_waiting;
28
29     time_t      timeout;
30
31     ngx\_event\_mutex\_t *mutex;
32 } ngx\_http\_busy\_lock\_t;
33
34
35 typedef struct {
36     time_t      time;
37     ngx\_event\_t *event;
38     void        (*event_handler)(ngx\_event\_t *ev);
39     u_char      *md5;
40     int         slot;
41 } ngx\_http\_busy\_lock\_ctx\_t;
42
43
44 int ngx\_http\_busy\_lock(ngx\_http\_busy\_lock\_t *bl, ngx\_http\_busy\_lock\_ctx\_t *bc);
45 int ngx\_http\_busy\_lock\_cacheable(ngx\_http\_busy\_lock\_t *bl,
46     ngx\_http\_busy\_lock\_ctx\_t *bc, int lock);
47 void ngx\_http\_busy\_unlock(ngx\_http\_busy\_lock\_t *bl,
48     ngx\_http\_busy\_lock\_ctx\_t *bc);
49
50 char *ngx\_http\_set\_busy\_lock\_slot(ngx\_conf\_t *cf, ngx\_command\_t *cmd,
51     void *conf);
52
53
54 #endif /* \_NGX\_HTTP\_BUSY\_LOCK\_H\_INCLUDED */
```

[One Level Up](#)

[Top Level](#)

## src/http/modules/nginx\_http\_ssi\_filter\_module.c - nginx-1.7.10

### Global variables defined

- [ngx\\_http\\_next\\_body\\_filter](#)
- [ngx\\_http\\_next\\_header\\_filter](#)
- [ngx\\_http\\_ssi\\_block\\_params](#)
- [ngx\\_http\\_ssi\\_commands](#)
- [ngx\\_http\\_ssi\\_config\\_params](#)
- [ngx\\_http\\_ssi\\_echo\\_params](#)
- [ngx\\_http\\_ssi\\_filter\\_commands](#)
- [ngx\\_http\\_ssi\\_filter\\_module](#)
- [ngx\\_http\\_ssi\\_filter\\_module\\_ctx](#)
- [ngx\\_http\\_ssi\\_if\\_params](#)
- [ngx\\_http\\_ssi\\_include\\_params](#)
- [ngx\\_http\\_ssi\\_no\\_params](#)
- [ngx\\_http\\_ssi\\_none](#)
- [ngx\\_http\\_ssi\\_null\\_string](#)
- [ngx\\_http\\_ssi\\_set\\_params](#)
- [ngx\\_http\\_ssi\\_string](#)
- [ngx\\_http\\_ssi\\_timefmt](#)
- [ngx\\_http\\_ssi\\_vars](#)

### Data types defined

- [ngx\\_http\\_ssi\\_block\\_t](#)
- [ngx\\_http\\_ssi\\_loc\\_conf\\_t](#)
- [ngx\\_http\\_ssi\\_state\\_e](#)
- [ngx\\_http\\_ssi\\_var\\_t](#)

### Functions defined

- [ngx\\_http\\_ssi\\_block](#)
- [ngx\\_http\\_ssi\\_body\\_filter](#)
- [ngx\\_http\\_ssi\\_buffered](#)
- [ngx\\_http\\_ssi\\_config](#)
- [ngx\\_http\\_ssi\\_create\\_loc\\_conf](#)



- [ngx http ssi create main conf](#)
- [ngx http ssi date gmt local variable](#)
- [ngx http ssi echo](#)
- [ngx http ssi else](#)
- [ngx http ssi endblock](#)
- [ngx http ssi endif](#)
- [ngx http ssi evaluate string](#)
- [ngx http ssi filter init](#)
- [ngx http ssi get variable](#)
- [ngx http ssi header filter](#)
- [ngx http ssi if](#)
- [ngx http ssi include](#)
- [ngx http ssi init main conf](#)
- [ngx http ssi merge loc conf](#)
- [ngx http ssi output](#)
- [ngx http ssi parse](#)
- [ngx http ssi preconfiguration](#)
- [ngx http ssi regex match](#)
- [ngx http ssi set](#)
- [ngx http ssi set variable](#)
- [ngx http ssi stub output](#)

## Macros defined

- [NGX\\_HTTP\\_SSI\\_ADD\\_PREFIX](#)
- [NGX\\_HTTP\\_SSI\\_ADD\\_ZERO](#)
- [NGX\\_HTTP\\_SSI\\_BLOCK\\_NAME](#)
- [NGX\\_HTTP\\_SSI\\_CONFIG\\_ERRMSG](#)
- [NGX\\_HTTP\\_SSI\\_CONFIG\\_TIMEFMT](#)
- [NGX\\_HTTP\\_SSI\\_DATE\\_LEN](#)
- [NGX\\_HTTP\\_SSI\\_ECHO\\_DEFAULT](#)
- [NGX\\_HTTP\\_SSI\\_ECHO\\_ENCODING](#)
- [NGX\\_HTTP\\_SSI\\_ECHO\\_VAR](#)
- [NGX\\_HTTP\\_SSI\\_ERROR](#)
- [NGX\\_HTTP\\_SSI\\_IF\\_EXPR](#)

- [NGX\\_HTTP\\_SSI\\_INCLUDE\\_FILE](#)
- [NGX\\_HTTP\\_SSI\\_INCLUDE\\_SET](#)
- [NGX\\_HTTP\\_SSI\\_INCLUDE\\_STUB](#)
- [NGX\\_HTTP\\_SSI\\_INCLUDE\\_VIRTUAL](#)
- [NGX\\_HTTP\\_SSI\\_INCLUDE\\_WAIT](#)
- [NGX\\_HTTP\\_SSI\\_SET\\_VALUE](#)
- [NGX\\_HTTP\\_SSI\\_SET\\_VAR](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12 #define NGX_HTTP_SSI_ERROR          1
13
14 #define NGX_HTTP_SSI_DATE_LEN      2048
15
16 #define NGX_HTTP_SSI_ADD_PREFIX    1
17 #define NGX_HTTP_SSI_ADD_ZERO     2
18
19
20 typedef struct {
21     ngx_flag_t    enable;
22     ngx_flag_t    silent_errors;
23     ngx_flag_t    ignore_recycled_buffers;
24     ngx_flag_t    last_modified;
25
26     ngx_hash_t    types;
27
28     size_t        min_file_chunk;
29     size_t        value_len;
30
31     ngx_array_t   *types_keys;
32 } ngx_http_ssi_loc_conf_t;
33
34
35 typedef struct {
36     ngx_str_t     name;
37     ngx_uint_t    key;
38     ngx_str_t     value;
39 } ngx_http_ssi_var_t;
40
41
42 typedef struct {
43     ngx_str_t     name;
44     ngx_chain_t   *bufs;
45     ngx_uint_t    count;
46 } ngx_http_ssi_block_t;
47
48
49 typedef enum {
50     ssi_start_state = 0,
51     ssi_tag_state,
52     ssi_comment0_state,
53     ssi_comment1_state,
54     ssi_sharp_state,
55     ssi_precommand_state,
56     ssi_command_state,

```

```

57     ssi_preparam_state,
58     ssi_param_state,
59     ssi_preequal_state,
60     ssi_prevalue_state,
61     ssi_double_quoted_value_state,
62     ssi_quoted_value_state,
63     ssi_quoted_symbol_state,
64     ssi_postparam_state,
65     ssi_comment_end0_state,
66     ssi_comment_end1_state,
67     ssi_error_state,
68     ssi_error_end0_state,
69     ssi_error_end1_state
70 } ngx_http_ssi_state_e;
71
72
73 static ngx_int_t ngx_http_ssi_output(ngx_http_request_t *r,
74     ngx_http_ssi_ctx_t *ctx);
75 static void ngx_http_ssi_buffered(ngx_http_request_t *r,
76     ngx_http_ssi_ctx_t *ctx);
77 static ngx_int_t ngx_http_ssi_parse(ngx_http_request_t *r,
78     ngx_http_ssi_ctx_t *ctx);
79 static ngx_str_t *ngx_http_ssi_get_variable(ngx_http_request_t *r,
80     ngx_str_t *name, ngx_uint_t key);
81 static ngx_int_t ngx_http_ssi_evaluate_string(ngx_http_request_t *r,
82     ngx_http_ssi_ctx_t *ctx, ngx_str_t *text, ngx_uint_t flags);
83 static ngx_int_t ngx_http_ssi_regex_match(ngx_http_request_t *r,
84     ngx_str_t *pattern, ngx_str_t *str);
85
86 static ngx_int_t ngx_http_ssi_include(ngx_http_request_t *r,
87     ngx_http_ssi_ctx_t *ctx, ngx_str_t **params);
88 static ngx_int_t ngx_http_ssi_stub_output(ngx_http_request_t *r, void *data,
89     ngx_int_t rc);
90 static ngx_int_t ngx_http_ssi_set_variable(ngx_http_request_t *r, void *data,
91     ngx_int_t rc);
92 static ngx_int_t ngx_http_ssi_echo(ngx_http_request_t *r,
93     ngx_http_ssi_ctx_t *ctx, ngx_str_t **params);
94 static ngx_int_t ngx_http_ssi_config(ngx_http_request_t *r,
95     ngx_http_ssi_ctx_t *ctx, ngx_str_t **params);
96 static ngx_int_t ngx_http_ssi_set(ngx_http_request_t *r,
97     ngx_http_ssi_ctx_t *ctx, ngx_str_t **params);
98 static ngx_int_t ngx_http_ssi_if(ngx_http_request_t *r,
99     ngx_http_ssi_ctx_t *ctx, ngx_str_t **params);
100 static ngx_int_t ngx_http_ssi_else(ngx_http_request_t *r,
101     ngx_http_ssi_ctx_t *ctx, ngx_str_t **params);
102 static ngx_int_t ngx_http_ssi_endif(ngx_http_request_t *r,
103     ngx_http_ssi_ctx_t *ctx, ngx_str_t **params);
104 static ngx_int_t ngx_http_ssi_block(ngx_http_request_t *r,
105     ngx_http_ssi_ctx_t *ctx, ngx_str_t **params);
106 static ngx_int_t ngx_http_ssi_endblock(ngx_http_request_t *r,
107     ngx_http_ssi_ctx_t *ctx, ngx_str_t **params);
108
109 static ngx_int_t ngx_http_ssi_date_gmt_local_variable(ngx_http_request_t *r,
110     ngx_http_variable_value_t *v, uintptr_t gmt);
111
112 static ngx_int_t ngx_http_ssi_preconfiguration(ngx_conf_t *cf);
113 static void *ngx_http_ssi_create_main_conf(ngx_conf_t *cf);
114 static char *ngx_http_ssi_init_main_conf(ngx_conf_t *cf, void *conf);
115 static void *ngx_http_ssi_create_loc_conf(ngx_conf_t *cf);
116 static char *ngx_http_ssi_merge_loc_conf(ngx_conf_t *cf,
117     void *parent, void *child);
118 static ngx_int_t ngx_http_ssi_filter_init(ngx_conf_t *cf);
119
120
121 static ngx_command_t  ngx_http_ssi_filter_commands[] = {
122
123     { ngx_string("ssi"),
124       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_HTTP_LIF_CONF
125         |NGX_CONF_FLAG,
126       ngx_conf_set_flag_slot,
127       NGX_HTTP_LOC_CONF_OFFSET,
128       offsetof(ngx_http_ssi_loc_conf_t, enable),
129       NULL },
130
131     { ngx_string("ssi_silent_errors"),
132       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,

```

```

133     ngx_conf_set_flag_slot,
134     NGX_HTTP_LOC_CONF_OFFSET,
135     offsetof(ngx_http_ssi_loc_conf_t, silent_errors),
136     NULL },
137
138 { ngx_string("ssi_ignore_recycled_buffers"),
139     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
140     ngx_conf_set_flag_slot,
141     NGX_HTTP_LOC_CONF_OFFSET,
142     offsetof(ngx_http_ssi_loc_conf_t, ignore_recycled_buffers),
143     NULL },
144
145 { ngx_string("ssi_min_file_chunk"),
146     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
147     ngx_conf_set_size_slot,
148     NGX_HTTP_LOC_CONF_OFFSET,
149     offsetof(ngx_http_ssi_loc_conf_t, min_file_chunk),
150     NULL },
151
152 { ngx_string("ssi_value_length"),
153     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
154     ngx_conf_set_size_slot,
155     NGX_HTTP_LOC_CONF_OFFSET,
156     offsetof(ngx_http_ssi_loc_conf_t, value_len),
157     NULL },
158
159 { ngx_string("ssi_types"),
160     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_1MORE,
161     ngx_http_types_slot,
162     NGX_HTTP_LOC_CONF_OFFSET,
163     offsetof(ngx_http_ssi_loc_conf_t, types_keys),
164     &ngx_http_html_default_types[0] },
165
166 { ngx_string("ssi_last_modified"),
167     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
168     ngx_conf_set_flag_slot,
169     NGX_HTTP_LOC_CONF_OFFSET,
170     offsetof(ngx_http_ssi_loc_conf_t, last_modified),
171     NULL },
172
173     ngx_null_command
174 };
175
176
177
178 static ngx_http_module_t ngx_http_ssi_filter_module_ctx = {
179     ngx_http_ssi_preconfiguration,      /* preconfiguration */
180     ngx_http_ssi_filter_init,          /* postconfiguration */
181
182     ngx_http_ssi_create_main_conf,     /* create main configuration */
183     ngx_http_ssi_init_main_conf,      /* init main configuration */
184
185     NULL,                               /* create server configuration */
186     NULL,                               /* merge server configuration */
187
188     ngx_http_ssi_create_loc_conf,     /* create location configuration */
189     ngx_http_ssi_merge_loc_conf      /* merge location configuration */
190 };
191
192
193 ngx_module_t ngx_http_ssi_filter_module = {
194     NGX_MODULE_V1,
195     &ngx_http_ssi_filter_module_ctx,  /* module context */
196     ngx_http_ssi_filter_commands,    /* module directives */
197     NGX_HTTP_MODULE,                 /* module type */
198     NULL,                             /* init master */
199     NULL,                             /* init module */
200     NULL,                             /* init process */
201     NULL,                             /* init thread */
202     NULL,                             /* exit thread */
203     NULL,                             /* exit process */
204     NULL,                             /* exit master */
205     NGX_MODULE_V1_PADDING
206 };
207
208

```

```

209 static ngx_http_output_header_filter_pt ngx_http_next_header_filter;
210 static ngx_http_output_body_filter_pt ngx_http_next_body_filter;
211
212
213 static u_char ngx_http_ssi_string[] = "<!--";
214
215 static ngx_str_t ngx_http_ssi_none = ngx_string("(none)");
216 static ngx_str_t ngx_http_ssi_timefmt = ngx_string("%A, %d-%b-%Y %H:%M:%S %Z");
217 static ngx_str_t ngx_http_ssi_null_string = ngx_null_string;
218
219
220 #define NGX_HTTP_SSI_INCLUDE_VIRTUAL 0
221 #define NGX_HTTP_SSI_INCLUDE_FILE 1
222 #define NGX_HTTP_SSI_INCLUDE_WAIT 2
223 #define NGX_HTTP_SSI_INCLUDE_SET 3
224 #define NGX_HTTP_SSI_INCLUDE_STUB 4
225
226 #define NGX_HTTP_SSI_ECHO_VAR 0
227 #define NGX_HTTP_SSI_ECHO_DEFAULT 1
228 #define NGX_HTTP_SSI_ECHO_ENCODING 2
229
230 #define NGX_HTTP_SSI_CONFIG_ERRMSG 0
231 #define NGX_HTTP_SSI_CONFIG_TIMEFMT 1
232
233 #define NGX_HTTP_SSI_SET_VAR 0
234 #define NGX_HTTP_SSI_SET_VALUE 1
235
236 #define NGX_HTTP_SSI_IF_EXPR 0
237
238 #define NGX_HTTP_SSI_BLOCK_NAME 0
239
240
241 static ngx_http_ssi_param_t ngx_http_ssi_include_params[] = {
242     { ngx_string("virtual"), NGX_HTTP_SSI_INCLUDE_VIRTUAL, 0, 0 },
243     { ngx_string("file"), NGX_HTTP_SSI_INCLUDE_FILE, 0, 0 },
244     { ngx_string("wait"), NGX_HTTP_SSI_INCLUDE_WAIT, 0, 0 },
245     { ngx_string("set"), NGX_HTTP_SSI_INCLUDE_SET, 0, 0 },
246     { ngx_string("stub"), NGX_HTTP_SSI_INCLUDE_STUB, 0, 0 },
247     { ngx_null_string, 0, 0, 0 }
248 };
249
250
251 static ngx_http_ssi_param_t ngx_http_ssi_echo_params[] = {
252     { ngx_string("var"), NGX_HTTP_SSI_ECHO_VAR, 1, 0 },
253     { ngx_string("default"), NGX_HTTP_SSI_ECHO_DEFAULT, 0, 0 },
254     { ngx_string("encoding"), NGX_HTTP_SSI_ECHO_ENCODING, 0, 0 },
255     { ngx_null_string, 0, 0, 0 }
256 };
257
258
259 static ngx_http_ssi_param_t ngx_http_ssi_config_params[] = {
260     { ngx_string("errmsg"), NGX_HTTP_SSI_CONFIG_ERRMSG, 0, 0 },
261     { ngx_string("timefmt"), NGX_HTTP_SSI_CONFIG_TIMEFMT, 0, 0 },
262     { ngx_null_string, 0, 0, 0 }
263 };
264
265
266 static ngx_http_ssi_param_t ngx_http_ssi_set_params[] = {
267     { ngx_string("var"), NGX_HTTP_SSI_SET_VAR, 1, 0 },
268     { ngx_string("value"), NGX_HTTP_SSI_SET_VALUE, 1, 0 },
269     { ngx_null_string, 0, 0, 0 }
270 };
271
272
273 static ngx_http_ssi_param_t ngx_http_ssi_if_params[] = {
274     { ngx_string("expr"), NGX_HTTP_SSI_IF_EXPR, 1, 0 },
275     { ngx_null_string, 0, 0, 0 }
276 };
277
278
279 static ngx_http_ssi_param_t ngx_http_ssi_block_params[] = {
280     { ngx_string("name"), NGX_HTTP_SSI_BLOCK_NAME, 1, 0 },
281     { ngx_null_string, 0, 0, 0 }
282 };
283
284

```

```

285 static ngx_http_ssi_param_t  ngx_http_ssi_no_params[] = {
286     { ngx_null_string, 0, 0, 0 }
287 };
288
289
290 static ngx_http_ssi_command_t  ngx_http_ssi_commands[] = {
291     { ngx_string("include"), ngx_http_ssi_include,
292       ngx_http_ssi_include_params, 0, 0, 1 },
293     { ngx_string("echo"), ngx_http_ssi_echo,
294       ngx_http_ssi_echo_params, 0, 0, 0 },
295     { ngx_string("config"), ngx_http_ssi_config,
296       ngx_http_ssi_config_params, 0, 0, 0 },
297     { ngx_string("set"), ngx_http_ssi_set, ngx_http_ssi_set_params, 0, 0, 0 },
298
299     { ngx_string("if"), ngx_http_ssi_if, ngx_http_ssi_if_params, 0, 0, 0 },
300     { ngx_string("elif"), ngx_http_ssi_if, ngx_http_ssi_if_params,
301       NGX_HTTP_SSI_COND_IF, 0, 0 },
302     { ngx_string("else"), ngx_http_ssi_else, ngx_http_ssi_no_params,
303       NGX_HTTP_SSI_COND_IF, 0, 0 },
304     { ngx_string("endif"), ngx_http_ssi_endif, ngx_http_ssi_no_params,
305       NGX_HTTP_SSI_COND_ELSE, 0, 0 },
306
307     { ngx_string("block"), ngx_http_ssi_block,
308       ngx_http_ssi_block_params, 0, 0, 0 },
309     { ngx_string("endblock"), ngx_http_ssi_endblock,
310       ngx_http_ssi_no_params, 0, 1, 0 },
311
312     { ngx_null_string, NULL, NULL, 0, 0, 0 }
313 };
314
315
316 static ngx_http_variable_t  ngx_http_ssi_vars[] = {
317
318     { ngx_string("date_local"), NULL, ngx_http_ssi_date_gmt_local_variable, 0,
319       NGX_HTTP_VAR_NOCACHEABLE, 0 },
320
321     { ngx_string("date_gmt"), NULL, ngx_http_ssi_date_gmt_local_variable, 1,
322       NGX_HTTP_VAR_NOCACHEABLE, 0 },
323
324     { ngx_null_string, NULL, NULL, 0, 0, 0 }
325 };
326
327
328
329 static ngx_int_t
330 ngx_http_ssi_header_filter(ngx_http_request_t *r)
331 {
332     ngx_http_ssi_ctx_t      *ctx;
333     ngx_http_ssi_loc_conf_t *slcf;
334
335     slcf = ngx_http_get_module_loc_conf(r, ngx_http_ssi_filter_module);
336
337     if (!slcf->enable
338         || r->headers_out.content_length_n == 0
339         || ngx_http_test_content_type(r, &slcf->types) == NULL)
340     {
341         return ngx_http_next_header_filter(r);
342     }
343
344     ctx = ngx_palloc(r->pool, sizeof(ngx_http_ssi_ctx_t));
345     if (ctx == NULL) {
346         return NGX_ERROR;
347     }
348
349     ngx_http_set_ctx(r, ctx, ngx_http_ssi_filter_module);
350
351
352     ctx->value_len = slcf->value_len;
353     ctx->last_out = &ctx->out;
354
355     ctx->encoding = NGX_HTTP_SSI_ENTITY_ENCODING;
356     ctx->output = 1;
357
358     ctx->params.elts = ctx->params_array;
359     ctx->params.size = sizeof(ngx_table_elt_t);
360     ctx->params.nalloc = NGX_HTTP_SSI_PARAMS_N;

```

```

361     ctx->params.pool = r->pool;
362
363     ctx->timefmt = ngx\_http\_ssi\_timefmt;
364     ngx\_str\_set(&ctx->errmsg,
365             "[an error occurred while processing the directive]");
366
367     r->filter_need_in_memory = 1;
368
369     if (r == r->main) {
370         ngx\_http\_clear\_content\_length(r);
371         ngx\_http\_clear\_accept\_ranges(r);
372
373         if (!slcf->last_modified) {
374             ngx\_http\_clear\_last\_modified(r);
375             ngx\_http\_clear\_etag(r);
376
377         } else {
378             ngx\_http\_weak\_etag(r);
379         }
380     }
381
382     return ngx\_http\_next\_header\_filter(r);
383 }
384
385
386 static ngx\_int\_t
387 ngx\_http\_ssi\_body\_filter(ngx\_http\_request\_t *r, ngx\_chain\_t *in)
388 {
389     size_t                len;
390     ngx\_int\_t             rc;
391     ngx\_buf\_t             *b;
392     ngx\_uint\_t           i, index;
393     ngx\_chain\_t          *cl, **ll;
394     ngx\_table\_elt\_t      *param;
395     ngx\_http\_ssi\_ctx\_t    *ctx, *mctx;
396     ngx\_http\_ssi\_block\_t *bl;
397     ngx\_http\_ssi\_param\_t *prm;
398     ngx\_http\_ssi\_command\_t *cmd;
399     ngx\_http\_ssi\_loc\_conf\_t *slcf;
400     ngx\_http\_ssi\_main\_conf\_t *smcf;
401     ngx\_str\_t             *params[NGX\_HTTP\_SSI\_MAX\_PARAMS + 1];
402
403     ctx = ngx\_http\_get\_module\_ctx(r, ngx\_http\_ssi\_filter\_module);
404
405     if (ctx == NULL
406         || (in == NULL
407             && ctx->buf == NULL
408             && ctx->in == NULL
409             && ctx->busy == NULL))
410     {
411         return ngx\_http\_next\_body\_filter(r, in);
412     }
413
414     /* add the incoming chain to the chain ctx->in */
415
416     if (in) {
417         if (ngx\_chain\_add\_copy(r->pool, &ctx->in, in) != NGX\_OK) {
418             return NGX\_ERROR;
419         }
420     }
421
422     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
423                 "http ssi filter \"%V?%V\"", &r->uri, &r->args);
424
425     if (ctx->wait) {
426
427         if (r != r->connection->data) {
428             ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
429                         "http ssi filter wait \"%V?%V\" non-active",
430                         &ctx->wait->uri, &ctx->wait->args);
431
432             return NGX\_AGAIN;
433         }
434
435         if (ctx->wait->done) {
436             ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,

```

```

437         "http ssi filter wait \"%V?%V\" done",
438         &ctx->wait->uri, &ctx->wait->args);
439
440     ctx->wait = NULL;
441
442     } else {
443         ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
444             "http ssi filter wait \"%V?%V\"",
445             &ctx->wait->uri, &ctx->wait->args);
446
447         return ngx_http_next_body_filter(r, NULL);
448     }
449 }
450
451 sllcf = ngx_http_get_module_loc_conf(r, ngx_http_ssi_filter_module);
452
453 while (ctx->in || ctx->buf) {
454
455     if (ctx->buf == NULL) {
456         ctx->buf = ctx->in->buf;
457         ctx->in = ctx->in->next;
458         ctx->pos = ctx->buf->pos;
459     }
460
461     if (ctx->state == ssi_start_state) {
462         ctx->copy_start = ctx->pos;
463         ctx->copy_end = ctx->pos;
464     }
465
466     b = NULL;
467
468     while (ctx->pos < ctx->buf->last) {
469
470         ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
471             "saved: %d state: %d", ctx->saved, ctx->state);
472
473         rc = ngx_http_ssi_parse(r, ctx);
474
475         ngx_log_debug4(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
476             "parse: %d, looked: %d %p-%p",
477             rc, ctx->looked, ctx->copy_start, ctx->copy_end);
478
479         if (rc == NGX_ERROR) {
480             return rc;
481         }
482
483         if (ctx->copy_start != ctx->copy_end) {
484
485             if (ctx->output) {
486
487                 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
488                     "saved: %d", ctx->saved);
489
490                 if (ctx->saved) {
491
492                     if (ctx->free) {
493                         cl = ctx->free;
494                         ctx->free = ctx->free->next;
495                         b = cl->buf;
496                         ngx_memzero(b, sizeof(ngx_buf_t));
497
498                     } else {
499                         b = ngx_calloc_buf(r->pool);
500                         if (b == NULL) {
501                             return NGX_ERROR;
502                         }
503
504                         cl = ngx_alloc_chain_link(r->pool);
505                         if (cl == NULL) {
506                             return NGX_ERROR;
507                         }
508
509                         cl->buf = b;
510                     }
511
512                     b->memory = 1;

```



```

513     b->pos = ngx_http_ssi_string;
514     b->last = ngx_http_ssi_string + ctx->saved;
515
516     *ctx->last_out = cl;
517     ctx->last_out = &cl->next;
518
519     ctx->saved = 0;
520 }
521
522 if (ctx->free) {
523     cl = ctx->free;
524     ctx->free = ctx->free->next;
525     b = cl->buf;
526
527 } else {
528     b = ngx_alloc_buf(r->pool);
529     if (b == NULL) {
530         return NGX_ERROR;
531     }
532
533     cl = ngx_alloc_chain_link(r->pool);
534     if (cl == NULL) {
535         return NGX_ERROR;
536     }
537
538     cl->buf = b;
539 }
540
541 ngx_memcpy(b, ctx->buf, sizeof(ngx_buf_t));
542
543 b->pos = ctx->copy_start;
544 b->last = ctx->copy_end;
545 b->shadow = NULL;
546 b->last_buf = 0;
547 b->recycled = 0;
548
549 if (b->in_file) {
550     if (slcf->min_file_chunk < (size_t) (b->last - b->pos))
551     {
552         b->file_last = b->file_pos
553                     + (b->last - ctx->buf->pos);
554         b->file_pos += b->pos - ctx->buf->pos;
555     }
556     else {
557         b->in_file = 0;
558     }
559 }
560
561 cl->next = NULL;
562 *ctx->last_out = cl;
563 ctx->last_out = &cl->next;
564
565 } else {
566     if (ctx->block
567         && ctx->saved + (ctx->copy_end - ctx->copy_start))
568     {
569         b = ngx_create_temp_buf(r->pool,
570                                ctx->saved + (ctx->copy_end - ctx->copy_start));
571
572         if (b == NULL) {
573             return NGX_ERROR;
574         }
575
576         if (ctx->saved) {
577             b->last = ngx_cpymem(b->pos, ngx_http_ssi_string,
578                                 ctx->saved);
579         }
580
581         b->last = ngx_cpymem(b->last, ctx->copy_start,
582                             ctx->copy_end - ctx->copy_start);
583
584         cl = ngx_alloc_chain_link(r->pool);
585         if (cl == NULL) {
586             return NGX_ERROR;
587         }
588

```

```

589         cl->buf = b;
590         cl->next = NULL;
591
592         b = NULL;
593
594         mctx = ngx_http_get_module_ctx(r->main,
595                                     ngx_http_ssi_filter_module);
596         bl = mctx->blocks->elts;
597         for (ll = &bl[mctx->blocks->nelts - 1].bufs;
598             *ll;
599             ll = &(*ll)->next)
600         {
601             /* void */
602         }
603
604         *ll = cl;
605     }
606
607     ctx->saved = 0;
608 }
609 }
610
611 if (ctx->state == ssi_start_state) {
612     ctx->copy_start = ctx->pos;
613     ctx->copy_end = ctx->pos;
614 } else {
615     ctx->copy_start = NULL;
616     ctx->copy_end = NULL;
617 }
618
619 if (rc == NGX_AGAIN) {
620     continue;
621 }
622
623
624 b = NULL;
625
626 if (rc == NGX_OK) {
627
628     smcf = ngx_http_get_module_main_conf(r,
629                                         ngx_http_ssi_filter_module);
630
631     cmd = ngx_hash_find(&smcf->hash, ctx->key, ctx->command.data,
632                       ctx->command.len);
633
634     if (cmd == NULL) {
635         if (ctx->output) {
636             ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
637                         "invalid SSI command: \"%V\"",
638                         &ctx->command);
639             goto ssi_error;
640         }
641     }
642
643     continue;
644 }
645
646 if (!ctx->output && !cmd->block) {
647
648     if (ctx->block) {
649
650         /* reconstruct the SSI command text */
651
652         len = 5 + ctx->command.len + 4;
653
654         param = ctx->params.elts;
655         for (i = 0; i < ctx->params.nelts; i++) {
656             len += 1 + param[i].key.len + 2
657                 + param[i].value.len + 1;
658         }
659
660         b = ngx_create_temp_buf(r->pool, len);
661
662         if (b == NULL) {
663             return NGX_ERROR;
664         }

```

```

665         cl = ngx_alloc_chain_link(r->pool);
666     if (cl == NULL) {
667         return NGX_ERROR;
668     }
669
670     cl->buf = b;
671     cl->next = NULL;
672
673     *b->last++ = '<';
674     *b->last++ = '!';
675     *b->last++ = '-';
676     *b->last++ = '-';
677     *b->last++ = '#';
678
679     b->last = ngx_cpymem(b->last, ctx->command.data,
680                        ctx->command.len);
681
682     for (i = 0; i < ctx->params.nelts; i++) {
683         *b->last++ = ' ';
684         b->last = ngx_cpymem(b->last, param[i].key.data,
685                            param[i].key.len);
686         *b->last++ = '=';
687         *b->last++ = '"';
688         b->last = ngx_cpymem(b->last, param[i].value.data,
689                            param[i].value.len);
690         *b->last++ = '"';
691     }
692
693     *b->last++ = ' ';
694     *b->last++ = '-';
695     *b->last++ = '-';
696     *b->last++ = '>';
697
698     mctx = ngx_http_get_module_ctx(r->main,
699                                   ngx_http_ssi_filter_module);
700     bl = mctx->blocks->elts;
701     for (ll = &bl[mctx->blocks->nelts - 1].bufs;
702         *ll;
703         ll = &(*ll)->next)
704     {
705         /* void */
706     }
707
708     *ll = cl;
709
710     b = NULL;
711
712     continue;
713 }
714
715 if (cmd->conditional == 0) {
716     continue;
717 }
718 }
719
720 if (cmd->conditional
721     && (ctx->conditional == 0
722         || ctx->conditional > cmd->conditional))
723 {
724     ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
725                 "invalid context of SSI command: \"%V\"",
726                 &ctx->command);
727     goto ssi_error;
728 }
729
730 if (ctx->params.nelts > NGX_HTTP_SSI_MAX_PARAMS) {
731     ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
732                 "too many SSI command parameters: \"%V\"",
733                 &ctx->command);
734     goto ssi_error;
735 }
736
737 ngx_memzero(params,
738             (NGX_HTTP_SSI_MAX_PARAMS + 1) * sizeof(ngx_str_t *));
739
740

```

```

741 param = ctx->params.elts;
742
743 for (i = 0; i < ctx->params.nelts; i++) {
744     for (prm = cmd->params; prm->name.len; prm++) {
745         if (param[i].key.len != prm->name.len
746             || ngx_strncmp(param[i].key.data, prm->name.data,
747                            prm->name.len) != 0)
748         {
749             continue;
750         }
751
752         if (!prm->multiple) {
753             if (params[prm->index]) {
754                 ngx_log_error(NGX_LOG_ERR,
755                               r->connection->log, 0,
756                               "duplicate \"%V\" parameter "
757                               "in \"%V\" SSI command",
758                               &param[i].key, &ctx->command);
759
760                 goto ssi_error;
761             }
762
763             params[prm->index] = &param[i].value;
764
765             break;
766         }
767
768         for (index = prm->index; params[index]; index++) {
769             /* void */
770         }
771
772         params[index] = &param[i].value;
773
774         break;
775     }
776
777     if (prm->name.len == 0) {
778         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
779                       "invalid parameter name: \"%V\" "
780                       "in \"%V\" SSI command",
781                       &param[i].key, &ctx->command);
782
783         goto ssi_error;
784     }
785 }
786
787 for (prm = cmd->params; prm->name.len; prm++) {
788     if (prm->mandatory && params[prm->index] == 0) {
789         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
790                       "mandatory \"%V\" parameter is absent "
791                       "in \"%V\" SSI command",
792                       &prm->name, &ctx->command);
793
794         goto ssi_error;
795     }
796 }
797
798 if (cmd->flush && ctx->out) {
799     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
800                  "ssi flush");
801
802     if (ngx_http_ssi_output(r, ctx) == NGX_ERROR) {
803         return NGX_ERROR;
804     }
805 }
806
807 rc = cmd->handler(r, ctx, params);
808
809 if (rc == NGX_OK) {
810     continue;
811 }
812
813 if (rc == NGX_DONE || rc == NGX_AGAIN || rc == NGX_ERROR) {

```

```

817         ngx_http_ssi_buffered(r, ctx);
818         return rc;
819     }
820 }
821
822
823 /* rc == NGX_HTTP_SSI_ERROR */
824
825 ssi_error:
826
827     if (slcf->silent_errors) {
828         continue;
829     }
830
831     if (ctx->free) {
832         cl = ctx->free;
833         ctx->free = ctx->free->next;
834         b = cl->buf;
835         ngx_memzero(b, sizeof(ngx_buf_t));
836
837     } else {
838         b = ngx_calloc_buf(r->pool);
839         if (b == NULL) {
840             return NGX_ERROR;
841         }
842
843         cl = ngx_alloc_chain_link(r->pool);
844         if (cl == NULL) {
845             return NGX_ERROR;
846         }
847
848         cl->buf = b;
849     }
850
851     b->memory = 1;
852     b->pos = ctx->errmsg.data;
853     b->last = ctx->errmsg.data + ctx->errmsg.len;
854
855     cl->next = NULL;
856     *ctx->last_out = cl;
857     ctx->last_out = &cl->next;
858
859     continue;
860 }
861
862 if (ctx->buf->last_buf || ngx_buf_in_memory(ctx->buf)) {
863     if (b == NULL) {
864         if (ctx->free) {
865             cl = ctx->free;
866             ctx->free = ctx->free->next;
867             b = cl->buf;
868             ngx_memzero(b, sizeof(ngx_buf_t));
869
870         } else {
871             b = ngx_calloc_buf(r->pool);
872             if (b == NULL) {
873                 return NGX_ERROR;
874             }
875
876             cl = ngx_alloc_chain_link(r->pool);
877             if (cl == NULL) {
878                 return NGX_ERROR;
879             }
880
881             cl->buf = b;
882         }
883
884         b->sync = 1;
885
886         cl->next = NULL;
887         *ctx->last_out = cl;
888         ctx->last_out = &cl->next;
889     }
890
891     b->last_buf = ctx->buf->last_buf;
892     b->shadow = ctx->buf;

```

```

893         if (slcf->ignore_recycled_buffers == 0) {
894             b->recycled = ctx->buf->recycled;
895         }
896     }
897 }
898
899 ctx->buf = NULL;
900
901 ctx->saved = ctx->looked;
902 }
903
904 if (ctx->out == NULL && ctx->busy == NULL) {
905     return NGX_OK;
906 }
907
908 return ngx_http_ssi_output(r, ctx);
909 }
910
911
912 static ngx_int_t
913 ngx_http_ssi_output(ngx_http_request_t *r, ngx_http_ssi_ctx_t *ctx)
914 {
915     ngx_int_t    rc;
916     ngx_buf_t    *b;
917     ngx_chain_t  *cl;
918
919     #if 1
920     b = NULL;
921     for (cl = ctx->out; cl; cl = cl->next) {
922         ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
923             "ssi out: %p %p", cl->buf, cl->buf->pos);
924         if (cl->buf == b) {
925             ngx_log_error(NGX_LOG_ALERT, r->connection->log, 0,
926                 "the same buf was used in ssi");
927             ngx_debug_point();
928             return NGX_ERROR;
929         }
930         b = cl->buf;
931     }
932     #endif
933
934     rc = ngx_http_next_body_filter(r, ctx->out);
935
936     if (ctx->busy == NULL) {
937         ctx->busy = ctx->out;
938     } else {
939         for (cl = ctx->busy; cl->next; cl = cl->next) { /* void */ }
940         cl->next = ctx->out;
941     }
942
943     ctx->out = NULL;
944     ctx->last_out = &ctx->out;
945
946     while (ctx->busy) {
947         cl = ctx->busy;
948         b = cl->buf;
949
950         if (ngx_buf_size(b) != 0) {
951             break;
952         }
953
954         if (b->shadow) {
955             b->shadow->pos = b->shadow->last;
956         }
957
958         ctx->busy = cl->next;
959
960         if (ngx_buf_in_memory(b) || b->in_file) {
961             /* add data bufs only to the free buf chain */
962
963             cl->next = ctx->free;
964             ctx->free = cl;
965         }
966     }
967 }
968

```

```

969     ngx\_http\_ssi\_buffered(r, ctx);
970
971     return rc;
972 }
973
974
975
976 static void
977 ngx\_http\_ssi\_buffered(ngx\_http\_request\_t *r, ngx\_http\_ssi\_ctx\_t *ctx)
978 {
979     if (ctx->in || ctx->buf) {
980         r->buffered |= NGX\_HTTP\_SSI\_BUFFERED;
981     } else {
982         r->buffered &= ~NGX\_HTTP\_SSI\_BUFFERED;
983     }
984 }
985
986
987
988 static ngx\_int\_t
989 ngx\_http\_ssi\_parse(ngx\_http\_request\_t *r, ngx\_http\_ssi\_ctx\_t *ctx)
990 {
991     u_char          *p, *value, *last, *copy_end, ch;
992     size_t          looked;
993     ngx\_http\_ssi\_state\_e state;
994
995     state = ctx->state;
996     looked = ctx->looked;
997     last = ctx->buf->last;
998     copy_end = ctx->copy_end;
999
1000     for (p = ctx->pos; p < last; p++) {
1001
1002         ch = *p;
1003
1004         if (state == ssi_start_state) {
1005
1006             /* the tight loop */
1007
1008             for ( ;; ) {
1009                 if (ch == '<') {
1010                     copy_end = p;
1011                     looked = 1;
1012                     state = ssi_tag_state;
1013
1014                     goto tag_started;
1015                 }
1016
1017                 if (++p == last) {
1018                     break;
1019                 }
1020
1021                 ch = *p;
1022             }
1023
1024             ctx->state = state;
1025             ctx->pos = p;
1026             ctx->looked = looked;
1027             ctx->copy_end = p;
1028
1029             if (ctx->copy_start == NULL) {
1030                 ctx->copy_start = ctx->buf->pos;
1031             }
1032
1033             return NGX\_AGAIN;
1034
1035             tag_started:
1036
1037             continue;
1038         }
1039
1040         switch (state) {
1041
1042             case ssi_start_state:
1043                 /* not reached */
1044                 break;

```

```

1045
1046 case ssi_tag_state:
1047     switch (ch) {
1048         case '!':
1049             looked = 2;
1050             state = ssi_comment0_state;
1051             break;
1052
1053         case '<':
1054             copy_end = p;
1055             break;
1056
1057         default:
1058             copy_end = p;
1059             looked = 0;
1060             state = ssi_start_state;
1061             break;
1062     }
1063
1064     break;
1065
1066 case ssi_comment0_state:
1067     switch (ch) {
1068         case '-':
1069             looked = 3;
1070             state = ssi_comment1_state;
1071             break;
1072
1073         case '<':
1074             copy_end = p;
1075             looked = 1;
1076             state = ssi_tag_state;
1077             break;
1078
1079         default:
1080             copy_end = p;
1081             looked = 0;
1082             state = ssi_start_state;
1083             break;
1084     }
1085
1086     break;
1087
1088 case ssi_comment1_state:
1089     switch (ch) {
1090         case '-':
1091             looked = 4;
1092             state = ssi_sharp_state;
1093             break;
1094
1095         case '<':
1096             copy_end = p;
1097             looked = 1;
1098             state = ssi_tag_state;
1099             break;
1100
1101         default:
1102             copy_end = p;
1103             looked = 0;
1104             state = ssi_start_state;
1105             break;
1106     }
1107
1108     break;
1109
1110 case ssi_sharp_state:
1111     switch (ch) {
1112         case '#':
1113             if (p - ctx->pos < 4) {
1114                 ctx->saved = 0;
1115             }
1116             looked = 0;
1117             state = ssi_precommand_state;
1118             break;
1119
1120         case '<':

```



```

1121         copy_end = p;
1122         looked = 1;
1123         state = ssi_tag_state;
1124         break;
1125
1126     default:
1127         copy_end = p;
1128         looked = 0;
1129         state = ssi_start_state;
1130         break;
1131     }
1132
1133     break;
1134
1135 case ssi_precommand_state:
1136     switch (ch) {
1137     case ' ':
1138     case CR:
1139     case LF:
1140     case '\t':
1141         break;
1142
1143     default:
1144         ctx->command.len = 1;
1145         ctx->command.data = ngx_pnalloc(r->pool,
1146                                         NGX_HTTP_SSI_COMMAND_LEN);
1147         if (ctx->command.data == NULL) {
1148             return NGX_ERROR;
1149         }
1150
1151         ctx->command.data[0] = ch;
1152
1153         ctx->key = 0;
1154         ctx->key = ngx_hash(ctx->key, ch);
1155
1156         ctx->params.nelts = 0;
1157
1158         state = ssi_command_state;
1159         break;
1160     }
1161
1162     break;
1163
1164 case ssi_command_state:
1165     switch (ch) {
1166     case ' ':
1167     case CR:
1168     case LF:
1169     case '\t':
1170         state = ssi_preparam_state;
1171         break;
1172
1173     case '-':
1174         state = ssi_comment_end0_state;
1175         break;
1176
1177     default:
1178         if (ctx->command.len == NGX_HTTP_SSI_COMMAND_LEN) {
1179             ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1180                          "the \"%V%c...\" SSI command is too long",
1181                          &ctx->command, ch);
1182
1183             state = ssi_error_state;
1184             break;
1185         }
1186
1187         ctx->command.data[ctx->command.len++] = ch;
1188         ctx->key = ngx_hash(ctx->key, ch);
1189     }
1190
1191     break;
1192
1193 case ssi_preparam_state:
1194     switch (ch) {
1195     case ' ':
1196     case CR:

```

```

1197     case LF:
1198     case '\t':
1199         break;
1200
1201     case '-':
1202         state = ssi_comment_end0_state;
1203         break;
1204
1205     default:
1206         ctx->param = ngx_array_push(&ctx->params);
1207         if (ctx->param == NULL) {
1208             return NGX_ERROR;
1209         }
1210
1211         ctx->param->key.len = 1;
1212         ctx->param->key.data = ngx_pnalloc(r->pool,
1213                                         NGX_HTTP_SSI_PARAM_LEN);
1214         if (ctx->param->key.data == NULL) {
1215             return NGX_ERROR;
1216         }
1217
1218         ctx->param->key.data[0] = ch;
1219
1220         ctx->param->value.len = 0;
1221
1222         if (ctx->value_buf == NULL) {
1223             ctx->param->value.data = ngx_pnalloc(r->pool,
1224                                                 ctx->value_len + 1);
1225             if (ctx->param->value.data == NULL) {
1226                 return NGX_ERROR;
1227             }
1228         } else {
1229             ctx->param->value.data = ctx->value_buf;
1230         }
1231
1232         state = ssi_param_state;
1233         break;
1234     }
1235 }
1236
1237 break;
1238
1239 case ssi_param_state:
1240     switch (ch) {
1241     case ' ':
1242     case CR:
1243     case LF:
1244     case '\t':
1245         state = ssi_preequal_state;
1246         break;
1247
1248     case '=':
1249         state = ssi_prevalue_state;
1250         break;
1251
1252     case '-':
1253         state = ssi_error_end0_state;
1254
1255         ctx->param->key.data[ctx->param->key.len++] = ch;
1256         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1257                     "invalid \"%V\" parameter in \"%V\" SSI command",
1258                     &ctx->param->key, &ctx->command);
1259         break;
1260
1261     default:
1262         if (ctx->param->key.len == NGX_HTTP_SSI_PARAM_LEN) {
1263             state = ssi_error_state;
1264             ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1265                         "too long \"%V%c...\" parameter in "
1266                         "\"%V\" SSI command",
1267                         &ctx->param->key, ch, &ctx->command);
1268             break;
1269         }
1270
1271         ctx->param->key.data[ctx->param->key.len++] = ch;
1272     }

```

```

1273         break;
1274
1275     case ssi_preequal_state:
1276         switch (ch) {
1277             case ' ':
1278             case CR:
1279             case LF:
1280             case '\t':
1281                 break;
1282
1283             case '=':
1284                 state = ssi_prevalue_state;
1285                 break;
1286
1287             default:
1288                 if (ch == '-') {
1289                     state = ssi_error_end0_state;
1290                 } else {
1291                     state = ssi_error_state;
1292                 }
1293
1294                 ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1295                     "unexpected \"%c\" symbol after \"%V\" "
1296                     "parameter in \"%V\" SSI command",
1297                     ch, &ctx->param->key, &ctx->command);
1298
1299                 break;
1300         }
1301
1302         break;
1303
1304     case ssi_prevalue_state:
1305         switch (ch) {
1306             case ' ':
1307             case CR:
1308             case LF:
1309             case '\t':
1310                 break;
1311
1312             case '"':
1313                 state = ssi_double_quoted_value_state;
1314                 break;
1315
1316             case '\\':
1317                 state = ssi_quoted_value_state;
1318                 break;
1319
1320             default:
1321                 if (ch == '-') {
1322                     state = ssi_error_end0_state;
1323                 } else {
1324                     state = ssi_error_state;
1325                 }
1326
1327                 ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1328                     "unexpected \"%c\" symbol before value of "
1329                     "\"%V\" parameter in \"%V\" SSI command",
1330                     ch, &ctx->param->key, &ctx->command);
1331
1332                 break;
1333         }
1334
1335         break;
1336
1337     case ssi_double_quoted_value_state:
1338         switch (ch) {
1339             case '"':
1340                 state = ssi_postparam_state;
1341                 break;
1342
1343             case '\\':
1344                 ctx->saved_state = ssi_double_quoted_value_state;
1345                 state = ssi_quoted_symbol_state;
1346
1347                 /* fall through */
1348             default:

```

```

1349     if (ctx->param->value.len == ctx->value_len) {
1350         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1351             "too long \"%V%c...\" value of \"%V\" "
1352             "parameter in \"%V\" SSI command",
1353             &ctx->param->value, ch, &ctx->param->key,
1354             &ctx->command);
1355         state = ssi_error_state;
1356         break;
1357     }
1358
1359     ctx->param->value.data[ctx->param->value.len++] = ch;
1360 }
1361
1362 break;
1363
1364 case ssi_quoted_value_state:
1365     switch (ch) {
1366     case '\\':
1367         state = ssi_postparam_state;
1368         break;
1369
1370     case '\\\\':
1371         ctx->saved_state = ssi_quoted_value_state;
1372         state = ssi_quoted_symbol_state;
1373
1374         /* fall through */
1375
1376     default:
1377         if (ctx->param->value.len == ctx->value_len) {
1378             ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1379                 "too long \"%V%c...\" value of \"%V\" "
1380                 "parameter in \"%V\" SSI command",
1381                 &ctx->param->value, ch, &ctx->param->key,
1382                 &ctx->command);
1383             state = ssi_error_state;
1384             break;
1385         }
1386
1387         ctx->param->value.data[ctx->param->value.len++] = ch;
1388     }
1389
1390     break;
1391
1392 case ssi_quoted_symbol_state:
1393     state = ctx->saved_state;
1394
1395     if (ctx->param->value.len == ctx->value_len) {
1396         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1397             "too long \"%V%c...\" value of \"%V\" "
1398             "parameter in \"%V\" SSI command",
1399             &ctx->param->value, ch, &ctx->param->key,
1400             &ctx->command);
1401         state = ssi_error_state;
1402         break;
1403     }
1404
1405     ctx->param->value.data[ctx->param->value.len++] = ch;
1406
1407     break;
1408
1409 case ssi_postparam_state:
1410
1411     if (ctx->param->value.len + 1 < ctx->value_len / 2) {
1412         value = ngx_pnalloc(r->pool, ctx->param->value.len + 1);
1413         if (value == NULL) {
1414             return NGX_ERROR;
1415         }
1416
1417         ngx_memcpy(value, ctx->param->value.data,
1418             ctx->param->value.len);
1419
1420         ctx->value_buf = ctx->param->value.data;
1421         ctx->param->value.data = value;
1422
1423     } else {
1424         ctx->value_buf = NULL;

```

```

1425     }
1426
1427     switch (ch) {
1428     case ' ':
1429     case CR:
1430     case LF:
1431     case '\t':
1432         state = ssi_preparam_state;
1433         break;
1434
1435     case '-':
1436         state = ssi_comment_end0_state;
1437         break;
1438
1439     default:
1440         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1441             "unexpected \"%c\" symbol after \"%V\" value "
1442             "of \"%V\" parameter in \"%V\" SSI command",
1443             ch, &ctx->param->value, &ctx->param->key,
1444             &ctx->command);
1445         state = ssi_error_state;
1446         break;
1447     }
1448
1449     break;
1450
1451 case ssi_comment_end0_state:
1452     switch (ch) {
1453     case '-':
1454         state = ssi_comment_end1_state;
1455         break;
1456
1457     default:
1458         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1459             "unexpected \"%c\" symbol in \"%V\" SSI command",
1460             ch, &ctx->command);
1461         state = ssi_error_state;
1462         break;
1463     }
1464
1465     break;
1466
1467 case ssi_comment_end1_state:
1468     switch (ch) {
1469     case '>':
1470         ctx->state = ssi_start_state;
1471         ctx->pos = p + 1;
1472         ctx->looked = looked;
1473         ctx->copy_end = copy_end;
1474
1475         if (ctx->copy_start == NULL && copy_end) {
1476             ctx->copy_start = ctx->buf->pos;
1477         }
1478
1479         return NGX_OK;
1480
1481     default:
1482         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1483             "unexpected \"%c\" symbol in \"%V\" SSI command",
1484             ch, &ctx->command);
1485         state = ssi_error_state;
1486         break;
1487     }
1488
1489     break;
1490
1491 case ssi_error_state:
1492     switch (ch) {
1493     case '-':
1494         state = ssi_error_end0_state;
1495         break;
1496
1497     default:
1498         break;
1499     }
1500

```

```

1501         break;
1502
1503     case ssi_error_end0_state:
1504         switch (ch) {
1505             case '-':
1506                 state = ssi_error_end1_state;
1507                 break;
1508
1509             default:
1510                 state = ssi_error_state;
1511                 break;
1512         }
1513
1514         break;
1515
1516     case ssi_error_end1_state:
1517         switch (ch) {
1518             case '>':
1519                 ctx->state = ssi_start_state;
1520                 ctx->pos = p + 1;
1521                 ctx->looked = looked;
1522                 ctx->copy_end = copy_end;
1523
1524                 if (ctx->copy_start == NULL && copy_end) {
1525                     ctx->copy_start = ctx->buf->pos;
1526                 }
1527
1528                 return NGX\_HTTP\_SSI\_ERROR;
1529
1530             default:
1531                 state = ssi_error_state;
1532                 break;
1533         }
1534
1535         break;
1536     }
1537 }
1538
1539 ctx->state = state;
1540 ctx->pos = p;
1541 ctx->looked = looked;
1542
1543 ctx->copy_end = (state == ssi_start_state) ? p : copy_end;
1544
1545 if (ctx->copy_start == NULL && ctx->copy_end) {
1546     ctx->copy_start = ctx->buf->pos;
1547 }
1548
1549 return NGX\_AGAIN;
1550 }
1551
1552
1553 static ngx\_str\_t *
1554 ngx\_http\_ssi\_get\_variable(ngx\_http\_request\_t *r, ngx\_str\_t *name,
1555 ngx\_uint\_t key)
1556 {
1557     ngx\_uint\_t i;
1558     ngx\_list\_part\_t *part;
1559     ngx\_http\_ssi\_var\_t *var;
1560     ngx\_http\_ssi\_ctx\_t *ctx;
1561
1562     ctx = ngx\_http\_get\_module\_ctx(r->main, ngx\_http\_ssi\_filter\_module);
1563
1564     #if (NGX_PCRE)
1565     {
1566         ngx\_str\_t *value;
1567
1568         if (key >= '0' && key <= '9') {
1569             i = key - '0';
1570
1571             if (i < ctx->ncaptures) {
1572                 value = ngx\_palloc(r->pool, sizeof(ngx\_str\_t));
1573                 if (value == NULL) {
1574                     return NULL;
1575                 }
1576

```

```

1577         i *= 2;
1578
1579         value->data = ctx->captures_data + ctx->captures[i];
1580         value->len = ctx->captures[i + 1] - ctx->captures[i];
1581
1582         return value;
1583     }
1584 }
1585 }
1586 #endif
1587
1588 if (ctx->variables == NULL) {
1589     return NULL;
1590 }
1591
1592 part = &ctx->variables->part;
1593 var = part->elts;
1594
1595 for (i = 0; /* void */ ; i++) {
1596
1597     if (i >= part->nelts) {
1598         if (part->next == NULL) {
1599             break;
1600         }
1601
1602         part = part->next;
1603         var = part->elts;
1604         i = 0;
1605     }
1606
1607     if (name->len != var[i].name.len) {
1608         continue;
1609     }
1610
1611     if (key != var[i].key) {
1612         continue;
1613     }
1614
1615     if (ngx_strncmp(name->data, var[i].name.data, name->len) == 0) {
1616         return &var[i].value;
1617     }
1618 }
1619
1620 return NULL;
1621 }
1622
1623
1624 static ngx_int_t
1625 ngx_http_ssi_evaluate_string(ngx_http_request_t *r, ngx_http_ssi_ctx_t *ctx,
1626     ngx_str_t *text, ngx_uint_t flags)
1627 {
1628     u_char          ch, *p, **value, *data, *part_data;
1629     size_t          size, len, prefix, part_len;
1630     ngx_str_t       var, *val;
1631     ngx_int_t       key;
1632     ngx_uint_t      i, n, bracket, quoted;
1633     ngx_array_t     lengths, values;
1634     ngx_http_variable_value_t *vv;
1635
1636     n = ngx_http_script_variables_count(text);
1637
1638     if (n == 0) {
1639
1640         data = text->data;
1641         p = data;
1642
1643         if ((flags & NGX_HTTP_SSI_ADD_PREFIX) && text->data[0] != '/') {
1644
1645             for (prefix = r->uri.len; prefix; prefix--) {
1646                 if (r->uri.data[prefix - 1] == '/') {
1647                     break;
1648                 }
1649             }
1650
1651             if (prefix) {
1652                 len = prefix + text->len;

```

```

1653     data = ngx_pnalloc(r->pool, len);
1654     if (data == NULL) {
1655         return NGX_ERROR;
1656     }
1657
1658     p = ngx_copy(data, r->uri.data, prefix);
1659 }
1660 }
1661
1662 quoted = 0;
1663
1664 for (i = 0; i < text->len; i++) {
1665     ch = text->data[i];
1666
1667     if (!quoted) {
1668         if (ch == '\\') {
1669             quoted = 1;
1670             continue;
1671         }
1672     } else {
1673         quoted = 0;
1674
1675         if (ch != '\\') && ch != '\'' && ch != '"' && ch != '$') {
1676             *p++ = '\\';
1677         }
1678     }
1679
1680     *p++ = ch;
1681 }
1682
1683 text->len = p - data;
1684 text->data = data;
1685
1686 return NGX_OK;
1687 }
1688
1689 if (ngx_array_init(&lengths, r->pool, 8, sizeof(size_t *)) != NGX_OK) {
1690     return NGX_ERROR;
1691 }
1692
1693 if (ngx_array_init(&values, r->pool, 8, sizeof(u_char *)) != NGX_OK) {
1694     return NGX_ERROR;
1695 }
1696
1697 len = 0;
1698 i = 0;
1699
1700 while (i < text->len) {
1701     if (text->data[i] == '$') {
1702         var.len = 0;
1703
1704         if (++i == text->len) {
1705             goto invalid_variable;
1706         }
1707
1708         if (text->data[i] == '{') {
1709             bracket = 1;
1710
1711             if (++i == text->len) {
1712                 goto invalid_variable;
1713             }
1714
1715             var.data = &text->data[i];
1716         } else {
1717             bracket = 0;
1718             var.data = &text->data[i];
1719         }
1720
1721         for ( /* void */ ; i < text->len; i++, var.len++) {
1722             ch = text->data[i];

```



```

1729     if (ch == '}' && bracket) {
1730         i++;
1731         bracket = 0;
1732         break;
1733     }
1734
1735
1736     if ((ch >= 'A' && ch <= 'Z')
1737         || (ch >= 'a' && ch <= 'z')
1738         || (ch >= '0' && ch <= '9')
1739         || ch == '_')
1740     {
1741         continue;
1742     }
1743
1744     break;
1745 }
1746
1747 if (bracket) {
1748     ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1749                 "the closing bracket in \"%V\" "
1750                 "variable is missing", &var);
1751     return NGX_HTTP_SSI_ERROR;
1752 }
1753
1754 if (var.len == 0) {
1755     goto invalid_variable;
1756 }
1757
1758 key = ngx_hash_strlow(var.data, var.data, var.len);
1759
1760 val = ngx_http_ssi_get_variable(r, &var, key);
1761
1762 if (val == NULL) {
1763     vv = ngx_http_get_variable(r, &var, key);
1764     if (vv == NULL) {
1765         return NGX_ERROR;
1766     }
1767
1768     if (vv->not_found) {
1769         continue;
1770     }
1771
1772     part_data = vv->data;
1773     part_len = vv->len;
1774
1775 } else {
1776     part_data = val->data;
1777     part_len = val->len;
1778 }
1779
1780 } else {
1781     part_data = &text->data[i];
1782     quoted = 0;
1783
1784     for (p = part_data; i < text->len; i++) {
1785         ch = text->data[i];
1786
1787         if (!quoted) {
1788
1789             if (ch == '\\') {
1790                 quoted = 1;
1791                 continue;
1792             }
1793
1794             if (ch == '$') {
1795                 break;
1796             }
1797
1798         } else {
1799             quoted = 0;
1800
1801             if (ch != '\\') && ch != '\'' && ch != '"' && ch != '$') {
1802                 *p++ = '\\';
1803             }
1804         }

```

```

1805         *p++ = ch;
1806     }
1807
1808     part_len = p - part_data;
1809 }
1810
1811     len += part_len;
1812
1813     size = ngx_array_push(&lengths);
1814     if (size == NULL) {
1815         return NGX_ERROR;
1816     }
1817
1818     *size = part_len;
1819
1820     value = ngx_array_push(&values);
1821     if (value == NULL) {
1822         return NGX_ERROR;
1823     }
1824
1825     *value = part_data;
1826 }
1827
1828     prefix = 0;
1829
1830     size = lengths.elts;
1831     value = values.elts;
1832
1833     if (flags & NGX_HTTP_SSI_ADD_PREFIX) {
1834         for (i = 0; i < values.nelts; i++) {
1835             if (size[i] != 0) {
1836                 if (*value[i] != '/') {
1837                     for (prefix = r->uri.len; prefix; prefix--) {
1838                         if (r->uri.data[prefix - 1] == '/') {
1839                             len += prefix;
1840                             break;
1841                         }
1842                     }
1843                 }
1844             }
1845             break;
1846         }
1847     }
1848 }
1849
1850     p = ngx_pnalloc(r->pool, len + ((flags & NGX_HTTP_SSI_ADD_ZERO) ? 1 : 0));
1851     if (p == NULL) {
1852         return NGX_ERROR;
1853     }
1854
1855     text->len = len;
1856     text->data = p;
1857
1858     p = ngx_copy(p, r->uri.data, prefix);
1859
1860     for (i = 0; i < values.nelts; i++) {
1861         p = ngx_copy(p, value[i], size[i]);
1862     }
1863
1864     return NGX_OK;
1865
1866 invalid_variable:
1867     ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1868         "invalid variable name in \"%V\"", text);
1869
1870     return NGX_HTTP_SSI_ERROR;
1871 }
1872
1873 static ngx_int_t
1874 ngx_http_ssi_regex_match(ngx_http_request_t *r, ngx_str_t *pattern,
1875     ngx_str_t *str)
1876 {
1877     #if (NGX_PCRE)

```

```

1881 int rc, *captures;
1882 u_char *p, errstr[NGX_MAX_CONF_ERRSTR];
1883 size_t size;
1884 ngx_int_t key;
1885 ngx_str_t *vv, name, value;
1886 ngx_uint_t i, n;
1887 ngx_http_ssi_ctx_t *ctx;
1888 ngx_http_ssi_var_t *var;
1889 ngx_regex_compile_t rgc;
1890
1891 ngx_memzero(&rgc, sizeof(ngx_regex_compile_t));
1892
1893 rgc.pattern = *pattern;
1894 rgc.pool = r->pool;
1895 rgc.err.len = NGX_MAX_CONF_ERRSTR;
1896 rgc.err.data = errstr;
1897
1898 if (ngx_regex_compile(&rgc) != NGX_OK) {
1899     ngx_log_error(NGX_LOG_ERR, r->connection->log, 0, "%V", &rgc.err);
1900     return NGX_HTTP_SSI_ERROR;
1901 }
1902
1903 n = (rgc.captures + 1) * 3;
1904
1905 captures = ngx_palloc(r->pool, n * sizeof(int));
1906 if (captures == NULL) {
1907     return NGX_ERROR;
1908 }
1909
1910 rc = ngx_regex_exec(rgc.regex, str, captures, n);
1911
1912 if (rc < NGX_REGEX_NO_MATCHED) {
1913     ngx_log_error(NGX_LOG_ALERT, r->connection->log, 0,
1914                 ngx_regex_exec_n " failed: %i on \"%V\" using \"%V\"",
1915                 rc, str, pattern);
1916     return NGX_HTTP_SSI_ERROR;
1917 }
1918
1919 if (rc == NGX_REGEX_NO_MATCHED) {
1920     return NGX_DECLINED;
1921 }
1922
1923 ctx = ngx_http_get_module_ctx(r->main, ngx_http_ssi_filter_module);
1924
1925 ctx->ncaptures = rc;
1926 ctx->captures = captures;
1927 ctx->captures_data = str->data;
1928
1929 if (rgc.named_captures > 0) {
1930
1931     if (ctx->variables == NULL) {
1932         ctx->variables = ngx_list_create(r->pool, 4,
1933                                         sizeof(ngx_http_ssi_var_t));
1934
1935         if (ctx->variables == NULL) {
1936             return NGX_ERROR;
1937         }
1938
1939         size = rgc.name_size;
1940         p = rgc.names;
1941
1942         for (i = 0; i < (ngx_uint_t) rgc.named_captures; i++, p += size) {
1943
1944             name.data = &p[2];
1945             name.len = ngx_strlen(name.data);
1946
1947             n = 2 * ((p[0] << 8) + p[1]);
1948
1949             value.data = &str->data[captures[n]];
1950             value.len = captures[n + 1] - captures[n];
1951
1952             key = ngx_hash_strlow(name.data, name.data, name.len);
1953
1954             vv = ngx_http_ssi_get_variable(r, &name, key);
1955
1956             if (vv) {

```

```

1957         *vv = value;
1958         continue;
1959     }
1960
1961     var = ngx_list_push(ctx->variables);
1962     if (var == NULL) {
1963         return NGX_ERROR;
1964     }
1965
1966     var->name = name;
1967     var->key = key;
1968     var->value = value;
1969 }
1970 }
1971
1972 return NGX_OK;
1973
1974 #else
1975
1976 ngx_log_error(NGX_LOG_ALERT, r->connection->log, 0,
1977     "the using of the regex \"%V\" in SSI requires PCRE library",
1978     pattern);
1979 return NGX_HTTP_SSI_ERROR;
1980
1981 #endif
1982 }
1983
1984
1985 static ngx_int_t
1986 ngx_http_ssi_include(ngx_http_request_t *r, ngx_http_ssi_ctx_t *ctx,
1987     ngx_str_t **params)
1988 {
1989     ngx_int_t rc, key;
1990     ngx_str_t *uri, *file, *wait, *set, *stub, args;
1991     ngx_buf_t *b;
1992     ngx_uint_t flags, i;
1993     ngx_chain_t *cl, *tl, **ll, *out;
1994     ngx_http_request_t *sr;
1995     ngx_http_ssi_var_t *var;
1996     ngx_http_ssi_ctx_t *mctx;
1997     ngx_http_ssi_block_t *bl;
1998     ngx_http_post_subrequest_t *psr;
1999
2000     uri = params[NGX_HTTP_SSI_INCLUDE_VIRTUAL];
2001     file = params[NGX_HTTP_SSI_INCLUDE_FILE];
2002     wait = params[NGX_HTTP_SSI_INCLUDE_WAIT];
2003     set = params[NGX_HTTP_SSI_INCLUDE_SET];
2004     stub = params[NGX_HTTP_SSI_INCLUDE_STUB];
2005
2006     if (uri && file) {
2007         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
2008             "inlcusion may be either virtual=\"%V\" or file=\"%V\"",
2009             uri, file);
2010         return NGX_HTTP_SSI_ERROR;
2011     }
2012
2013     if (uri == NULL && file == NULL) {
2014         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
2015             "no parameter in \"include\" SSI command");
2016         return NGX_HTTP_SSI_ERROR;
2017     }
2018
2019     if (set && stub) {
2020         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
2021             "\"set\" and \"stub\" cannot be used together "
2022             "in \"include\" SSI command");
2023         return NGX_HTTP_SSI_ERROR;
2024     }
2025
2026     if (wait) {
2027         if (uri == NULL) {
2028             ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
2029                 "\"wait\" cannot be used with file=\"%V\"", file);
2030             return NGX_HTTP_SSI_ERROR;
2031         }
2032     }

```

```

2033     if (wait->len == 2
2034         && ngx_strncasecmp(wait->data, (u_char *) "no", 2) == 0)
2035     {
2036         wait = NULL;
2037
2038     } else if (wait->len != 3
2039               || ngx_strncasecmp(wait->data, (u_char *) "yes", 3) != 0)
2040     {
2041         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
2042                       "invalid value \"%V\" in the \"wait\" parameter",
2043                       wait);
2044         return NGX_HTTP_SSI_ERROR;
2045     }
2046 }
2047
2048 if (uri == NULL) {
2049     uri = file;
2050     wait = (ngx_str_t *) -1;
2051 }
2052
2053 rc = ngx_http_ssi_evaluate_string(r, ctx, uri, NGX_HTTP_SSI_ADD_PREFIX);
2054
2055 if (rc != NGX_OK) {
2056     return rc;
2057 }
2058
2059 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
2060               "ssi include: \"%V\"", uri);
2061
2062 ngx_str_null(&args);
2063 flags = NGX_HTTP_LOG_UNSAFE;
2064
2065 if (ngx_http_parse_unsafe_uri(r, uri, &args, &flags) != NGX_OK) {
2066     return NGX_HTTP_SSI_ERROR;
2067 }
2068
2069 psr = NULL;
2070
2071 mctx = ngx_http_get_module_ctx(r->main, ngx_http_ssi_filter_module);
2072
2073 if (stub) {
2074     if (mctx->blocks) {
2075         bl = mctx->blocks->elts;
2076         for (i = 0; i < mctx->blocks->nelts; i++) {
2077             if (stub->len == bl[i].name.len
2078                 && ngx_strncmp(stub->data, bl[i].name.data, stub->len) == 0)
2079             {
2080                 goto found;
2081             }
2082         }
2083     }
2084
2085     ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
2086                 "\"stub\"=\"%V\" for \"include\" not found", stub);
2087     return NGX_HTTP_SSI_ERROR;
2088 }
2089 found:
2090
2091 psr = ngx_palloc(r->pool, sizeof(ngx_http_post_subrequest_t));
2092 if (psr == NULL) {
2093     return NGX_ERROR;
2094 }
2095
2096 psr->handler = ngx_http_ssi_stub_output;
2097
2098 if (bl[i].count++) {
2099
2100     out = NULL;
2101     ll = &out;
2102
2103     for (t1 = bl[i].bufs; t1; t1 = t1->next) {
2104
2105         if (ctx->free) {
2106             cl = ctx->free;
2107             ctx->free = ctx->free->next;
2108             b = cl->buf;

```

```

2109     } else {
2110         b = ngx_alloc_buf(r->pool);
2111         if (b == NULL) {
2112             return NGX_ERROR;
2113         }
2114
2115         cl = ngx_alloc_chain_link(r->pool);
2116         if (cl == NULL) {
2117             return NGX_ERROR;
2118         }
2119
2120         cl->buf = b;
2121     }
2122
2123     ngx_memcpy(b, tl->buf, sizeof(ngx_buf_t));
2124
2125     b->pos = b->start;
2126
2127     *ll = cl;
2128     cl->next = NULL;
2129     ll = &cl->next;
2130 }
2131
2132 psr->data = out;
2133
2134 } else {
2135     psr->data = bl[i].bufs;
2136 }
2137 }
2138
2139 if (wait) {
2140     flags |= NGX_HTTP_SUBREQUEST_WAITED;
2141 }
2142
2143 if (set) {
2144     key = ngx_hash_strlow(set->data, set->data, set->len);
2145
2146     psr = ngx_palloc(r->pool, sizeof(ngx_http_post_subrequest_t));
2147     if (psr == NULL) {
2148         return NGX_ERROR;
2149     }
2150
2151     psr->handler = ngx_http_ssi_set_variable;
2152     psr->data = ngx_http_ssi_get_variable(r, set, key);
2153
2154     if (psr->data == NULL) {
2155         if (mctx->variables == NULL) {
2156             mctx->variables = ngx_list_create(r->pool, 4,
2157                 sizeof(ngx_http_ssi_var_t));
2158             if (mctx->variables == NULL) {
2159                 return NGX_ERROR;
2160             }
2161         }
2162
2163         var = ngx_list_push(mctx->variables);
2164         if (var == NULL) {
2165             return NGX_ERROR;
2166         }
2167
2168         var->name = *set;
2169         var->key = key;
2170         var->value = ngx_http_ssi_null_string;
2171         psr->data = &var->value;
2172     }
2173
2174     flags |= NGX_HTTP_SUBREQUEST_IN_MEMORY|NGX_HTTP_SUBREQUEST_WAITED;
2175 }
2176
2177 if (ngx_http_subrequest(r, uri, &args, &sr, psr, flags) != NGX_OK) {
2178     return NGX_HTTP_SSI_ERROR;
2179 }
2180
2181 if (wait == NULL && set == NULL) {
2182     return NGX_OK;
2183 }

```

```

2185     }
2186
2187     if (ctx->wait == NULL) {
2188         ctx->wait = sr;
2189
2190         return NGX_AGAIN;
2191
2192     } else {
2193         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
2194             "can only wait for one subrequest at a time");
2195     }
2196
2197     return NGX_OK;
2198 }
2199
2200
2201 static ngx_int_t
2202 ngx_http_ssi_stub_output(ngx_http_request_t *r, void *data, ngx_int_t rc)
2203 {
2204     ngx_chain_t *out;
2205
2206     if (rc == NGX_ERROR || r->connection->error || r->request_output) {
2207         return rc;
2208     }
2209
2210     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
2211         "ssi stub output: \"%V?%V\"", &r->uri, &r->args);
2212
2213     out = data;
2214
2215     if (!r->header_sent) {
2216         r->headers_out.content_type_len =
2217             r->parent->headers_out.content_type_len;
2218         r->headers_out.content_type = r->parent->headers_out.content_type;
2219
2220         if (ngx_http_send_header(r) == NGX_ERROR) {
2221             return NGX_ERROR;
2222         }
2223     }
2224
2225     return ngx_http_output_filter(r, out);
2226 }
2227
2228
2229 static ngx_int_t
2230 ngx_http_ssi_set_variable(ngx_http_request_t *r, void *data, ngx_int_t rc)
2231 {
2232     ngx_str_t *value = data;
2233
2234     if (r->upstream) {
2235         value->len = r->upstream->buffer.last - r->upstream->buffer.pos;
2236         value->data = r->upstream->buffer.pos;
2237     }
2238
2239     return rc;
2240 }
2241
2242
2243 static ngx_int_t
2244 ngx_http_ssi_echo(ngx_http_request_t *r, ngx_http_ssi_ctx_t *ctx,
2245     ngx_str_t **params)
2246 {
2247     u_char                *p;
2248     uintptr_t             len;
2249     ngx_int_t            key;
2250     ngx_buf_t           *b;
2251     ngx_str_t           *var, *value, *enc, text;
2252     ngx_chain_t         *cl;
2253     ngx_http_variable_value_t *vv;
2254
2255     var = params[NGX_HTTP_SSI_ECHO_VAR];
2256
2257     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
2258         "ssi echo \"%V\"", var);
2259
2260     key = ngx_hash_strlow(var->data, var->data, var->len);

```

```

2261 value = ngx_http_ssi_get_variable(r, var, key);
2262
2263
2264 if (value == NULL) {
2265     vv = ngx_http_get_variable(r, var, key);
2266
2267     if (vv == NULL) {
2268         return NGX_HTTP_SSI_ERROR;
2269     }
2270
2271     if (!vv->not_found) {
2272         text.data = vv->data;
2273         text.len = vv->len;
2274         value = &text;
2275     }
2276 }
2277
2278 if (value == NULL) {
2279     value = params[NGX_HTTP_SSI_ECHO_DEFAULT];
2280
2281     if (value == NULL) {
2282         value = &ngx_http_ssi_none;
2283     }
2284     else if (value->len == 0) {
2285         return NGX_OK;
2286     }
2287 }
2288 else {
2289     if (value->len == 0) {
2290         return NGX_OK;
2291     }
2292 }
2293
2294 enc = params[NGX_HTTP_SSI_ECHO_ENCODING];
2295
2296 if (enc) {
2297     if (enc->len == 4 && ngx_strncmp(enc->data, "none", 4) == 0) {
2298
2299         ctx->encoding = NGX_HTTP_SSI_NO_ENCODING;
2300
2301     } else if (enc->len == 3 && ngx_strncmp(enc->data, "url", 3) == 0) {
2302
2303         ctx->encoding = NGX_HTTP_SSI_URL_ENCODING;
2304
2305     } else if (enc->len == 6 && ngx_strncmp(enc->data, "entity", 6) == 0) {
2306
2307         ctx->encoding = NGX_HTTP_SSI_ENTITY_ENCODING;
2308
2309     } else {
2310         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
2311             "unknown encoding \"%V\" in the \"echo\" command",
2312             enc);
2313     }
2314 }
2315
2316 p = value->data;
2317
2318 switch (ctx->encoding) {
2319
2320 case NGX_HTTP_SSI_URL_ENCODING:
2321     len = 2 * ngx_escape_uri(NULL, value->data, value->len,
2322         NGX_ESCAPE_HTML);
2323
2324     if (len) {
2325         p = ngx_pnalloc(r->pool, value->len + len);
2326         if (p == NULL) {
2327             return NGX_HTTP_SSI_ERROR;
2328         }
2329
2330         (void) ngx_escape_uri(p, value->data, value->len, NGX_ESCAPE_HTML);
2331     }
2332
2333     len += value->len;
2334     break;
2335
2336 case NGX_HTTP_SSI_ENTITY_ENCODING:

```



```

2337     len = ngx_escape_html(NULL, value->data, value->len);
2338
2339     if (len) {
2340         p = ngx_pnalloc(r->pool, value->len + len);
2341         if (p == NULL) {
2342             return NGX_HTTP_SSI_ERROR;
2343         }
2344
2345         (void) ngx_escape_html(p, value->data, value->len);
2346     }
2347
2348     len += value->len;
2349     break;
2350
2351     default: /* NGX_HTTP_SSI_NO_ENCODING */
2352         len = value->len;
2353         break;
2354 }
2355
2356 b = ngx_calloc_buf(r->pool);
2357 if (b == NULL) {
2358     return NGX_HTTP_SSI_ERROR;
2359 }
2360
2361 cl = ngx_alloc_chain_link(r->pool);
2362 if (cl == NULL) {
2363     return NGX_HTTP_SSI_ERROR;
2364 }
2365
2366 b->memory = 1;
2367 b->pos = p;
2368 b->last = p + len;
2369
2370 cl->buf = b;
2371 cl->next = NULL;
2372 *ctx->last_out = cl;
2373 ctx->last_out = &cl->next;
2374
2375 return NGX_OK;
2376 }
2377
2378
2379 static ngx_int_t
2380 ngx_http_ssi_config(ngx_http_request_t *r, ngx_http_ssi_ctx_t *ctx,
2381     ngx_str_t **params)
2382 {
2383     ngx_str_t *value;
2384
2385     value = params[NGX_HTTP_SSI_CONFIG_TIMEFMT];
2386
2387     if (value) {
2388         ctx->timefmt.len = value->len;
2389         ctx->timefmt.data = ngx_pnalloc(r->pool, value->len + 1);
2390         if (ctx->timefmt.data == NULL) {
2391             return NGX_HTTP_SSI_ERROR;
2392         }
2393
2394         ngx_cpymem(ctx->timefmt.data, value->data, value->len + 1);
2395     }
2396
2397     value = params[NGX_HTTP_SSI_CONFIG_ERRMSG];
2398
2399     if (value) {
2400         ctx->errmsg = *value;
2401     }
2402
2403     return NGX_OK;
2404 }
2405
2406
2407 static ngx_int_t
2408 ngx_http_ssi_set(ngx_http_request_t *r, ngx_http_ssi_ctx_t *ctx,
2409     ngx_str_t **params)
2410 {
2411     ngx_int_t key, rc;
2412     ngx_str_t *name, *value, *vv;

```

```

2413     ngx_http_ssi_var_t *var;
2414     ngx_http_ssi_ctx_t *mctx;
2415
2416     mctx = ngx_http_get_module_ctx(r->main, ngx_http_ssi_filter_module);
2417
2418     if (mctx->variables == NULL) {
2419         mctx->variables = ngx_list_create(r->pool, 4,
2420                                         sizeof(ngx_http_ssi_var_t));
2421         if (mctx->variables == NULL) {
2422             return NGX_ERROR;
2423         }
2424     }
2425
2426     name = params[NGX_HTTP_SSI_SET_VAR];
2427     value = params[NGX_HTTP_SSI_SET_VALUE];
2428
2429     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
2430                  "ssi set \"%V\" \"%V\"", name, value);
2431
2432     rc = ngx_http_ssi_evaluate_string(r, ctx, value, 0);
2433
2434     if (rc != NGX_OK) {
2435         return rc;
2436     }
2437
2438     key = ngx_hash_strlow(name->data, name->data, name->len);
2439
2440     vv = ngx_http_ssi_get_variable(r, name, key);
2441
2442     if (vv) {
2443         *vv = *value;
2444         return NGX_OK;
2445     }
2446
2447     var = ngx_list_push(mctx->variables);
2448     if (var == NULL) {
2449         return NGX_ERROR;
2450     }
2451
2452     var->name = *name;
2453     var->key = key;
2454     var->value = *value;
2455
2456     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
2457                  "set: \"%V\"=\"%V\"", name, value);
2458
2459     return NGX_OK;
2460 }
2461
2462
2463 static ngx_int_t
2464 ngx_http_ssi_if(ngx_http_request_t *r, ngx_http_ssi_ctx_t *ctx,
2465                ngx_str_t **params)
2466 {
2467     u_char      *p, *last;
2468     ngx_str_t   *expr, left, right;
2469     ngx_int_t   rc;
2470     ngx_uint_t  negative, noregex, flags;
2471
2472     if (ctx->command.len == 2) {
2473         if (ctx->conditional) {
2474             ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
2475                          "the \"if\" command inside the \"if\" command");
2476             return NGX_HTTP_SSI_ERROR;
2477         }
2478     }
2479
2480     if (ctx->output_chosen) {
2481         ctx->output = 0;
2482         return NGX_OK;
2483     }
2484
2485     expr = params[NGX_HTTP_SSI_IF_EXPR];
2486
2487     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
2488                  "ssi if expr=\"%V\"", expr);

```

```

2489 left.data = expr->data;
2490 last = expr->data + expr->len;
2492
2493 for (p = left.data; p < last; p++) {
2494     if (*p >= 'A' && *p <= 'Z') {
2495         *p |= 0x20;
2496         continue;
2497     }
2498
2499     if ((*p >= 'a' && *p <= 'z')
2500         || (*p >= '0' && *p <= '9')
2501         || *p == '$' || *p == '{' || *p == '}' || *p == '_'
2502         || *p == '"' || *p == '\\')
2503     {
2504         continue;
2505     }
2506
2507     break;
2508 }
2509
2510 left.len = p - left.data;
2511
2512 while (p < last && *p == ' ') {
2513     p++;
2514 }
2515
2516 flags = 0;
2517
2518 ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
2519     "left: \"%V\"", &left);
2520
2521 rc = ngx\_http\_ssi\_evaluate\_string(r, ctx, &left, flags);
2522
2523 if (rc != NGX\_OK) {
2524     return rc;
2525 }
2526
2527 ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
2528     "evaluated left: \"%V\"", &left);
2529
2530 if (p == last) {
2531     if (left.len) {
2532         ctx->output = 1;
2533         ctx->output_chosen = 1;
2534     } else {
2535         ctx->output = 0;
2536     }
2537 }
2538
2539 ctx->conditional = NGX\_HTTP\_SSI\_COND\_IF;
2540
2541 return NGX\_OK;
2542 }
2543
2544 if (p < last && *p == '=') {
2545     negative = 0;
2546     p++;
2547 } else if (p + 1 < last && *p == '!' && *(p + 1) == '=') {
2548     negative = 1;
2549     p += 2;
2550 } else {
2551     goto invalid_expression;
2552 }
2553
2554 while (p < last && *p == ' ') {
2555     p++;
2556 }
2557
2558 if (p < last - 1 && *p == '/') {
2559     if (*(last - 1) != '/') {
2560         goto invalid_expression;
2561     }
2562 }
2563
2564

```

```

2565     noregex = 0;
2566     flags = NGX\_HTTP\_SSI\_ADD\_ZERO;
2567     last--;
2568     p++;
2569
2570 } else {
2571     noregex = 1;
2572     flags = 0;
2573
2574     if (p < last - 1 && p[0] == '\\\' && p[1] == '/') {
2575         p++;
2576     }
2577 }
2578
2579 right.len = last - p;
2580 right.data = p;
2581
2582 ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
2583     "right: \"%V\"", &right);
2584
2585 rc = ngx\_http\_ssi\_evaluate\_string(r, ctx, &right, flags);
2586
2587 if (rc != NGX\_OK) {
2588     return rc;
2589 }
2590
2591 ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
2592     "evaluted right: \"%V\"", &right);
2593
2594 if (noregex) {
2595     if (left.len != right.len) {
2596         rc = -1;
2597
2598     } else {
2599         rc = ngx\_strncmp(left.data, right.data, right.len);
2600     }
2601
2602 } else {
2603     right.data[right.len] = '\\0';
2604
2605     rc = ngx\_http\_ssi\_regex\_match(r, &right, &left);
2606
2607     if (rc == NGX\_OK) {
2608         rc = 0;
2609     } else if (rc == NGX\_DECLINED) {
2610         rc = -1;
2611     } else {
2612         return rc;
2613     }
2614 }
2615
2616 if ((rc == 0 && !negative) || (rc != 0 && negative)) {
2617     ctx->output = 1;
2618     ctx->output_chosen = 1;
2619
2620 } else {
2621     ctx->output = 0;
2622 }
2623
2624 ctx->conditional = NGX\_HTTP\_SSI\_COND\_IF;
2625
2626 return NGX\_OK;
2627
2628 invalid_expression:
2629
2630 ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, 0,
2631     "invalid expression in \"%V\"", expr);
2632
2633 return NGX\_HTTP\_SSI\_ERROR;
2634 }
2635
2636
2637 static ngx\_int\_t
2638 ngx\_http\_ssi\_else(ngx\_http\_request\_t *r, ngx\_http\_ssi\_ctx\_t *ctx,
2639     ngx\_str\_t **params)
2640 {

```

```

2641     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
2642                 "ssi else");
2643
2644     if (ctx->output_chosen) {
2645         ctx->output = 0;
2646     } else {
2647         ctx->output = 1;
2648     }
2649
2650     ctx->conditional = NGX_HTTP_SSI_COND_ELSE;
2651
2652     return NGX_OK;
2653 }
2654
2655 static ngx_int_t
2656 ngx_http_ssi_endif(ngx_http_request_t *r, ngx_http_ssi_ctx_t *ctx,
2657                  ngx_str_t **params)
2658 {
2659     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
2660                 "ssi endif");
2661
2662     ctx->output = 1;
2663     ctx->output_chosen = 0;
2664     ctx->conditional = 0;
2665
2666     return NGX_OK;
2667 }
2668
2669 static ngx_int_t
2670 ngx_http_ssi_block(ngx_http_request_t *r, ngx_http_ssi_ctx_t *ctx,
2671                  ngx_str_t **params)
2672 {
2673     ngx_http_ssi_ctx_t *mctx;
2674     ngx_http_ssi_block_t *bl;
2675
2676     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
2677                 "ssi block");
2678
2679     mctx = ngx_http_get_module_ctx(r->main, ngx_http_ssi_filter_module);
2680
2681     if (mctx->blocks == NULL) {
2682         mctx->blocks = ngx_array_create(r->pool, 4,
2683                                       sizeof(ngx_http_ssi_block_t));
2684         if (mctx->blocks == NULL) {
2685             return NGX_HTTP_SSI_ERROR;
2686         }
2687     }
2688
2689     bl = ngx_array_push(mctx->blocks);
2690     if (bl == NULL) {
2691         return NGX_HTTP_SSI_ERROR;
2692     }
2693
2694     bl->name = *params[NGX_HTTP_SSI_BLOCK_NAME];
2695     bl->bufs = NULL;
2696     bl->count = 0;
2697
2698     ctx->output = 0;
2699     ctx->block = 1;
2700
2701     return NGX_OK;
2702 }
2703
2704 static ngx_int_t
2705 ngx_http_ssi_endblock(ngx_http_request_t *r, ngx_http_ssi_ctx_t *ctx,
2706                    ngx_str_t **params)
2707 {
2708     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
2709                 "ssi endblock");
2710
2711     ctx->output = 1;
2712     ctx->block = 0;
2713
2714     return NGX_OK;
2715 }

```

```

2717     return NGX_OK;
2718 }
2719
2720
2721 static ngx_int_t
2722 ngx_http_ssi_date_gmt_local_variable(ngx_http_request_t *r,
2723     ngx_http_variable_value_t *v, uintptr_t gmt)
2724 {
2725     ngx_http_ssi_ctx_t *ctx;
2726     ngx_time_t *tp;
2727     ngx_str_t *timefmt;
2728     struct tm tm;
2729     char buf[NGX_HTTP_SSI_DATE_LEN];
2730
2731     v->valid = 1;
2732     v->no_cacheable = 0;
2733     v->not_found = 0;
2734
2735     tp = ngx_timeofday();
2736
2737     ctx = ngx_http_get_module_ctx(r, ngx_http_ssi_filter_module);
2738
2739     timefmt = ctx ? &ctx->timefmt : &ngx_http_ssi_timefmt;
2740
2741     if (timefmt->len == sizeof("%s") - 1
2742         && timefmt->data[0] == '%' && timefmt->data[1] == 's')
2743     {
2744         v->data = ngx_pnalloc(r->pool, NGX_TIME_T_LEN);
2745         if (v->data == NULL) {
2746             return NGX_ERROR;
2747         }
2748
2749         v->len = ngx_sprintf(v->data, "%T", tp->sec) - v->data;
2750
2751         return NGX_OK;
2752     }
2753
2754     if (gmt) {
2755         ngx_libc_gmtime(tp->sec, &tm);
2756     } else {
2757         ngx_libc_localtime(tp->sec, &tm);
2758     }
2759
2760     v->len = strftime(buf, NGX_HTTP_SSI_DATE_LEN,
2761         (char *) timefmt->data, &tm);
2762     if (v->len == 0) {
2763         return NGX_ERROR;
2764     }
2765
2766     v->data = ngx_pnalloc(r->pool, v->len);
2767     if (v->data == NULL) {
2768         return NGX_ERROR;
2769     }
2770
2771     ngx_memcpy(v->data, buf, v->len);
2772
2773     return NGX_OK;
2774 }
2775
2776
2777 static ngx_int_t
2778 ngx_http_ssi_preconfiguration(ngx_conf_t *cf)
2779 {
2780     ngx_int_t rc;
2781     ngx_http_variable_t *var, *v;
2782     ngx_http_ssi_command_t *cmd;
2783     ngx_http_ssi_main_conf_t *smcf;
2784
2785     for (v = ngx_http_ssi_vars; v->name.len; v++) {
2786         var = ngx_http_add_variable(cf, &v->name, v->flags);
2787         if (var == NULL) {
2788             return NGX_ERROR;
2789         }
2790
2791         var->get_handler = v->get_handler;
2792         var->data = v->data;

```

```

2793     }
2794
2795     smcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_ssi_filter_module);
2796
2797     for (cmd = ngx_http_ssi_commands; cmd->name.len; cmd++) {
2798         rc = ngx_hash_add_key(&smcf->commands, &cmd->name, cmd,
2799                               NGX_HASH_READONLY_KEY);
2800
2801         if (rc == NGX_OK) {
2802             continue;
2803         }
2804
2805         if (rc == NGX_BUSY) {
2806             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
2807                               "conflicting SSI command \"%V\"", &cmd->name);
2808         }
2809
2810         return NGX_ERROR;
2811     }
2812
2813     return NGX_OK;
2814 }
2815
2816
2817 static void *
2818 ngx_http_ssi_create_main_conf(ngx_conf_t *cf)
2819 {
2820     ngx_http_ssi_main_conf_t *smcf;
2821
2822     smcf = ngx_palloc(cf->pool, sizeof(ngx_http_ssi_main_conf_t));
2823     if (smcf == NULL) {
2824         return NULL;
2825     }
2826
2827     smcf->commands.pool = cf->pool;
2828     smcf->commands.temp_pool = cf->temp_pool;
2829
2830     if (ngx_hash_keys_array_init(&smcf->commands, NGX_HASH_SMALL) != NGX_OK) {
2831         return NULL;
2832     }
2833
2834     return smcf;
2835 }
2836
2837
2838 static char *
2839 ngx_http_ssi_init_main_conf(ngx_conf_t *cf, void *conf)
2840 {
2841     ngx_http_ssi_main_conf_t *smcf = conf;
2842
2843     ngx_hash_init_t hash;
2844
2845     hash.hash = &smcf->hash;
2846     hash.key = ngx_hash_key;
2847     hash.max_size = 1024;
2848     hash.bucket_size = ngx_cacheline_size;
2849     hash.name = "ssi_command_hash";
2850     hash.pool = cf->pool;
2851     hash.temp_pool = NULL;
2852
2853     if (ngx_hash_init(&hash, smcf->commands.keys.elts,
2854                     smcf->commands.keys.nelts)
2855         != NGX_OK)
2856     {
2857         return NGX_CONF_ERROR;
2858     }
2859
2860     return NGX_CONF_OK;
2861 }
2862
2863
2864 static void *
2865 ngx_http_ssi_create_loc_conf(ngx_conf_t *cf)
2866 {
2867     ngx_http_ssi_loc_conf_t *slcf;
2868

```

```

2869     slcf = ngx_palloc(cf->pool, sizeof(ngx_http_ssi_loc_conf_t));
2870     if (slcf == NULL) {
2871         return NULL;
2872     }
2873
2874     /*
2875     * set by ngx_palloc():
2876     *
2877     *     conf->types = { NULL };
2878     *     conf->types_keys = NULL;
2879     */
2880
2881     slcf->enable = NGX_CONF_UNSET;
2882     slcf->silent_errors = NGX_CONF_UNSET;
2883     slcf->ignore_recycled_buffers = NGX_CONF_UNSET;
2884     slcf->last_modified = NGX_CONF_UNSET;
2885
2886     slcf->min_file_chunk = NGX_CONF_UNSET_SIZE;
2887     slcf->value_len = NGX_CONF_UNSET_SIZE;
2888
2889     return slcf;
2890 }
2891
2892
2893 static char *
2894 ngx_http_ssi_merge_loc_conf(ngx_conf_t *cf, void *parent, void *child)
2895 {
2896     ngx_http_ssi_loc_conf_t *prev = parent;
2897     ngx_http_ssi_loc_conf_t *conf = child;
2898
2899     ngx_conf_merge_value(conf->enable, prev->enable, 0);
2900     ngx_conf_merge_value(conf->silent_errors, prev->silent_errors, 0);
2901     ngx_conf_merge_value(conf->ignore_recycled_buffers,
2902         prev->ignore_recycled_buffers, 0);
2903     ngx_conf_merge_value(conf->last_modified, prev->last_modified, 0);
2904
2905     ngx_conf_merge_size_value(conf->min_file_chunk, prev->min_file_chunk, 1024);
2906     ngx_conf_merge_size_value(conf->value_len, prev->value_len, 255);
2907
2908     if (ngx_http_merge_types(cf, &conf->types_keys, &conf->types,
2909         &prev->types_keys, &prev->types,
2910         ngx_http_html_default_types)
2911         != NGX_OK)
2912     {
2913         return NGX_CONF_ERROR;
2914     }
2915
2916     return NGX_CONF_OK;
2917 }
2918
2919
2920 static ngx_int_t
2921 ngx_http_ssi_filter_init(ngx_conf_t *cf)
2922 {
2923     ngx_http_next_header_filter = ngx_http_top_header_filter;
2924     ngx_http_top_header_filter = ngx_http_ssi_header_filter;
2925
2926     ngx_http_next_body_filter = ngx_http_top_body_filter;
2927     ngx_http_top_body_filter = ngx_http_ssi_body_filter;
2928
2929     return NGX_OK;
2930 }

```

[One Level Up](#)

[Top Level](#)



# src/http/modules/nginx\_http\_ssi\_filter\_module.h - nginx-1.7.10

## Data types defined

- [ngx\\_http\\_ssi\\_command\\_pt](#)
- [ngx\\_http\\_ssi\\_command\\_t](#)
- [ngx\\_http\\_ssi\\_ctx\\_t](#)
- [ngx\\_http\\_ssi\\_main\\_conf\\_t](#)
- [ngx\\_http\\_ssi\\_param\\_t](#)

## Macros defined

- [NGX\\_HTTP\\_SSI\\_COMMAND\\_LEN](#)
- [NGX\\_HTTP\\_SSI\\_COND\\_ELSE](#)
- [NGX\\_HTTP\\_SSI\\_COND\\_IF](#)
- [NGX\\_HTTP\\_SSI\\_ENTITY\\_ENCODING](#)
- [NGX\\_HTTP\\_SSI\\_MAX\\_PARAMS](#)
- [NGX\\_HTTP\\_SSI\\_NO\\_ENCODING](#)
- [NGX\\_HTTP\\_SSI\\_PARAMS\\_N](#)
- [NGX\\_HTTP\\_SSI\\_PARAM\\_LEN](#)
- [NGX\\_HTTP\\_SSI\\_URL\\_ENCODING](#)
- [\\_NGX\\_HTTP\\_SSI\\_FILTER\\_H\\_INCLUDED\\_](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_HTTP_SSI_FILTER_H_INCLUDED_
9 #define _NGX_HTTP_SSI_FILTER_H_INCLUDED_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14 #include <ngx_http.h>
15
16
17 #define NGX_HTTP_SSI_MAX_PARAMS          16
18
19 #define NGX_HTTP_SSI_COMMAND_LEN        32
20 #define NGX_HTTP_SSI_PARAM_LEN          32
21 #define NGX_HTTP_SSI_PARAMS_N           4
22
23
24 #define NGX_HTTP_SSI_COND_IF             1
25 #define NGX_HTTP_SSI_COND_ELSE          2
26
27
```

```

28 #define NGX_HTTP_SSI_NO_ENCODING      0
29 #define NGX_HTTP_SSI_URL_ENCODING    1
30 #define NGX_HTTP_SSI_ENTITY_ENCODING  2
31
32
33 typedef struct {
34     ngx_hash_t          hash;
35     ngx_hash_keys_arrays_t  commands;
36 } ngx_http_ssi_main_conf_t;
37
38
39 typedef struct {
40     ngx_buf_t           *buf;
41
42     u_char              *pos;
43     u_char              *copy_start;
44     u_char              *copy_end;
45
46     ngx_uint_t          key;
47     ngx_str_t           command;
48     ngx_array_t         params;
49     ngx_table_elt_t     *param;
50     ngx_table_elt_t     params_array[NGX_HTTP_SSI_PARAMS_N];
51
52     ngx_chain_t         *in;
53     ngx_chain_t         *out;
54     ngx_chain_t         **last_out;
55     ngx_chain_t         *busy;
56     ngx_chain_t         *free;
57
58     ngx_uint_t          state;
59     ngx_uint_t          saved_state;
60     size_t              saved;
61     size_t              looked;
62
63     size_t              value_len;
64
65     ngx_list_t          *variables;
66     ngx_array_t         *blocks;
67
68     #if (NGX_PCRE)
69     ngx_uint_t          ncaptures;
70     int                 *captures;
71     u_char              *captures_data;
72     #endif
73
74     unsigned            conditional:2;
75     unsigned            encoding:2;
76     unsigned            block:1;
77     unsigned            output:1;
78     unsigned            output_chosen:1;
79
80     ngx_http_request_t  *wait;
81     void                *value_buf;
82     ngx_str_t           timefmt;
83     ngx_str_t           errmsg;
84 } ngx_http_ssi_ctx_t;
85
86
87 typedef ngx_int_t (*ngx_http_ssi_command_pt) (ngx_http_request_t *r,
88     ngx_http_ssi_ctx_t *ctx, ngx_str_t **);
89
90
91 typedef struct {
92     ngx_str_t           name;
93     ngx_uint_t          index;
94
95     unsigned            mandatory:1;
96     unsigned            multiple:1;
97 } ngx_http_ssi_param_t;
98
99
100 typedef struct {
101     ngx_str_t           name;
102     ngx_http_ssi_command_pt handler;
103     ngx_http_ssi_param_t *params;

```

```
104
105     unsigned           conditional:2;
106     unsigned           block:1;
107     unsigned           flush:1;
108 } ngx_http_ssi_command_t;
109
110
111 extern ngx_module_t ngx_http_ssi_filter_module;
112
113
114 #endif /* NGX_HTTP_SSI_FILTER_H_INCLUDED */
```

[One Level Up](#)

[Top Level](#)

# src/http/nginx\_http\_postpone\_filter\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_next\\_body\\_filter](#)
- [ngx\\_http\\_postpone\\_filter\\_module](#)
- [ngx\\_http\\_postpone\\_filter\\_module\\_ctx](#)

## Functions defined

- [ngx\\_http\\_postpone\\_filter](#)
- [ngx\\_http\\_postpone\\_filter\\_add](#)
- [ngx\\_http\\_postpone\\_filter\\_init](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 static ngx_int_t ngx_http_postpone_filter_add(ngx_http_request_t *r,
14     ngx_chain_t *in);
15 static ngx_int_t ngx_http_postpone_filter_init(ngx_conf_t *cf);
16
17
18 static ngx_http_module_t ngx_http_postpone_filter_module_ctx = {
19     NULL, /* preconfiguration */
20     ngx_http_postpone_filter_init, /* postconfiguration */
21
22     NULL, /* create main configuration */
23     NULL, /* init main configuration */
24
25     NULL, /* create server configuration */
26     NULL, /* merge server configuration */
27
28     NULL, /* create location configuration */
29     NULL, /* merge location configuration */
30 };
31
32
33 ngx_module_t ngx_http_postpone_filter_module = {
34     NGX_MODULE_V1,
35     &ngx_http_postpone_filter_module_ctx, /* module context */
36     NULL, /* module directives */
37     NGX_HTTP_MODULE, /* module type */
38     NULL, /* init master */
39     NULL, /* init module */
40     NULL, /* init process */
41     NULL, /* init thread */
42     NULL, /* exit thread */
43     NULL, /* exit process */
44     NULL, /* exit master */
45     NGX_MODULE_V1_PADDING
46 };
47
48
```

```

49 static ngx_http_output_body_filter_pt  ngx_http_next_body_filter;
50
51
52 static ngx_int_t
53 ngx_http_postpone_filter(ngx_http_request_t *r, ngx_chain_t *in)
54 {
55     ngx_connection_t      *c;
56     ngx_http_postponed_request_t *pr;
57
58     c = r->connection;
59
60     ngx_log_debug3(NGX_LOG_DEBUG_HTTP, c->log, 0,
61                  "http postpone filter \"%V?%V\" %p", &r->uri, &r->args, in);
62
63     if (r != c->data) {
64
65         if (in) {
66             ngx_http_postpone_filter_add(r, in);
67             return NGX_OK;
68         }
69
70 #if 0
71         /* TODO: SSI may pass NULL */
72         ngx_log_error(NGX_LOG_ALERT, c->log, 0,
73                     "http postpone filter NULL inactive request");
74 #endif
75
76         return NGX_OK;
77     }
78
79     if (r->postponed == NULL) {
80
81         if (in || c->buffered) {
82             return ngx_http_next_body_filter(r->main, in);
83         }
84
85         return NGX_OK;
86     }
87
88     if (in) {
89         ngx_http_postpone_filter_add(r, in);
90     }
91
92     do {
93         pr = r->postponed;
94
95         if (pr->request) {
96
97             ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
98                          "http postpone filter wake \"%V?%V\"",
99                          &pr->request->uri, &pr->request->args);
100
101             r->postponed = pr->next;
102
103             c->data = pr->request;
104
105             return ngx_http_post_request(pr->request, NULL);
106         }
107
108         if (pr->out == NULL) {
109             ngx_log_error(NGX_LOG_ALERT, c->log, 0,
110                          "http postpone filter NULL output");
111
112         } else {
113             ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
114                          "http postpone filter output \"%V?%V\"",
115                          &r->uri, &r->args);
116
117             if (ngx_http_next_body_filter(r->main, pr->out) == NGX_ERROR) {
118                 return NGX_ERROR;
119             }
120         }
121
122         r->postponed = pr->next;
123
124     } while (r->postponed);

```

```

125     return NGX\_OK;
126 }
127
128
129
130 static ngx\_int\_t
131 ngx\_http\_postpone\_filter\_add(ngx\_http\_request\_t *r, ngx\_chain\_t *in)
132 {
133     ngx\_http\_postponed\_request\_t *pr, **ppr;
134
135     if (r->postponed) {
136         for (pr = r->postponed; pr->next; pr = pr->next) { /* void */ }
137
138         if (pr->request == NULL) {
139             goto found;
140         }
141
142         ppr = &pr->next;
143
144     } else {
145         ppr = &r->postponed;
146     }
147
148     pr = ngx\_palloc(r->pool, sizeof(ngx\_http\_postponed\_request\_t));
149     if (pr == NULL) {
150         return NGX\_ERROR;
151     }
152
153     *ppr = pr;
154
155     pr->request = NULL;
156     pr->out = NULL;
157     pr->next = NULL;
158
159 found:
160
161     if (ngx\_chain\_add\_copy(r->pool, &pr->out, in) == NGX\_OK) {
162         return NGX\_OK;
163     }
164
165     return NGX\_ERROR;
166 }
167
168
169 static ngx\_int\_t
170 ngx\_http\_postpone\_filter\_init(ngx\_conf\_t *cf)
171 {
172     ngx\_http\_next\_body\_filter = ngx\_http\_top\_body\_filter;
173     ngx\_http\_top\_body\_filter = ngx\_http\_postpone\_filter;
174
175     return NGX\_OK;
176 }

```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_degradation\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_degradation\\_commands](#)
- [ngx\\_http\\_degradation\\_module](#)
- [ngx\\_http\\_degradation\\_module\\_ctx](#)
- [ngx\\_http\\_degrade](#)

## Data types defined

- [ngx\\_http\\_degradation\\_loc\\_conf\\_t](#)
- [ngx\\_http\\_degradation\\_main\\_conf\\_t](#)

## Functions defined

- [ngx\\_http\\_degradation](#)
- [ngx\\_http\\_degradation\\_create\\_loc\\_conf](#)
- [ngx\\_http\\_degradation\\_create\\_main\\_conf](#)
- [ngx\\_http\\_degradation\\_handler](#)
- [ngx\\_http\\_degradation\\_init](#)
- [ngx\\_http\\_degradation\\_merge\\_loc\\_conf](#)
- [ngx\\_http\\_degraded](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 typedef struct {
14     size_t      sbrk_size;
15 } ngx_http_degradation_main_conf_t;
16
17
18 typedef struct {
19     ngx_uint_t  degrade;
20 } ngx_http_degradation_loc_conf_t;
21
22
23 static ngx_conf_enum_t  ngx_http_degrade[] = {
24     { ngx_string("204"), 204 },
25     { ngx_string("444"), 444 },
26     { ngx_null_string, 0 }
27 };
28
```

```

29
30 static void *ngx_http_degradation_create_main_conf(ngx_conf_t *cf);
31 static void *ngx_http_degradation_create_loc_conf(ngx_conf_t *cf);
32 static char *ngx_http_degradation_merge_loc_conf(ngx_conf_t *cf, void *parent,
33     void *child);
34 static char *ngx_http_degradation(ngx_conf_t *cf, ngx_command_t *cmd,
35     void *conf);
36 static ngx_int_t ngx_http_degradation_init(ngx_conf_t *cf);
37
38
39 static ngx_command_t  ngx_http_degradation_commands[] = {
40
41     { ngx_string("degradation"),
42       NGX_HTTP_MAIN_CONF|NGX_CONF_TAKE1,
43       ngx_http_degradation,
44       NGX_HTTP_MAIN_CONF_OFFSET,
45       0,
46       NULL },
47
48     { ngx_string("degrade"),
49       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
50       ngx_conf_set_enum_slot,
51       NGX_HTTP_LOC_CONF_OFFSET,
52       offsetof(ngx_http_degradation_loc_conf_t, degrade),
53       &ngx_http_degrade },
54
55     ngx_null_command
56 };
57
58
59 static ngx_http_module_t  ngx_http_degradation_module_ctx = {
60     NULL, /* preconfiguration */
61     ngx_http_degradation_init, /* postconfiguration */
62
63     ngx_http_degradation_create_main_conf, /* create main configuration */
64     NULL, /* init main configuration */
65
66     NULL, /* create server configuration */
67     NULL, /* merge server configuration */
68
69     ngx_http_degradation_create_loc_conf, /* create location configuration */
70     ngx_http_degradation_merge_loc_conf /* merge location configuration */
71 };
72
73
74 ngx_module_t  ngx_http_degradation_module = {
75     NGX_MODULE_V1,
76     &ngx_http_degradation_module_ctx, /* module context */
77     ngx_http_degradation_commands, /* module directives */
78     NGX_HTTP_MODULE, /* module type */
79     NULL, /* init master */
80     NULL, /* init module */
81     NULL, /* init process */
82     NULL, /* init thread */
83     NULL, /* exit thread */
84     NULL, /* exit process */
85     NULL, /* exit master */
86     NGX_MODULE_V1_PADDING
87 };
88
89
90 static ngx_int_t
91 ngx_http_degradation_handler(ngx_http_request_t *r)
92 {
93     ngx_http_degradation_loc_conf_t  *dlcf;
94
95     dlcf = ngx_http_get_module_loc_conf(r, ngx_http_degradation_module);
96
97     if (dlcf->degrade && ngx_http_degraded(r)) {
98         return dlcf->degrade;
99     }
100
101     return NGX_DECLINED;
102 }
103
104

```



```

105 ngx_uint_t
106 ngx_http_degraded(ngx_http_request_t *r)
107 {
108     time_t                now;
109     ngx_uint_t            log;
110     static size_t         sbrk_size;
111     static time_t         sbrk_time;
112     ngx_http_degradation_main_conf_t *dmcf;
113
114     dmcf = ngx_http_get_module_main_conf(r, ngx_http_degradation_module);
115
116     if (dmcf->sbrk_size) {
117
118         log = 0;
119         now = ngx_time();
120
121         /* lock mutex */
122
123         if (now != sbrk_time) {
124
125             /*
126              * ELF/i386 is loaded at 0x08000000, 128M
127              * ELF/amd64 is loaded at 0x00400000, 4M
128              *
129              * use a function address to subtract the loading address
130              */
131
132             sbrk_size = (size_t) sbrk(0) - ((uintptr_t) ngx_palloc & ~0x3FFFFFF);
133             sbrk_time = now;
134             log = 1;
135         }
136
137         /* unlock mutex */
138
139         if (sbrk_size >= dmcf->sbrk_size) {
140             if (log) {
141                 ngx_log_error(NGX_LOG_NOTICE, r->connection->log, 0,
142                     "degradation sbrk:%uZM",
143                     sbrk_size / (1024 * 1024));
144             }
145
146             return 1;
147         }
148     }
149
150     return 0;
151 }
152
153
154 static void *
155 ngx_http_degradation_create_main_conf(ngx_conf_t *cf)
156 {
157     ngx_http_degradation_main_conf_t *dmcf;
158
159     dmcf = ngx_palloc(cf->pool, sizeof(ngx_http_degradation_main_conf_t));
160     if (dmcf == NULL) {
161         return NULL;
162     }
163
164     return dmcf;
165 }
166
167
168 static void *
169 ngx_http_degradation_create_loc_conf(ngx_conf_t *cf)
170 {
171     ngx_http_degradation_loc_conf_t *conf;
172
173     conf = ngx_palloc(cf->pool, sizeof(ngx_http_degradation_loc_conf_t));
174     if (conf == NULL) {
175         return NULL;
176     }
177
178     conf->degrade = NGX_CONF_UNSET_UINT;
179
180     return conf;

```

```

181 }
182
183
184 static char *
185 ngx_http_degradation_merge_loc_conf(ngx_conf_t *cf, void *parent, void *child)
186 {
187     ngx_http_degradation_loc_conf_t *prev = parent;
188     ngx_http_degradation_loc_conf_t *conf = child;
189
190     ngx_conf_merge_uint_value(conf->degrade, prev->degrade, 0);
191
192     return NGX_CONF_OK;
193 }
194
195
196 static char *
197 ngx_http_degradation(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
198 {
199     ngx_http_degradation_main_conf_t *dmcf = conf;
200
201     ngx_str_t *value, s;
202
203     value = cf->args->elts;
204
205     if (ngx_strncmp(value[1].data, "sbrk=", 5) == 0) {
206
207         s.len = value[1].len - 5;
208         s.data = value[1].data + 5;
209
210         dmcf->sbrk_size = ngx_parse_size(&s);
211         if (dmcf->sbrk_size == (size_t) NGX_ERROR) {
212             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
213                 "invalid sbrk size \"%V\"", &value[1]);
214             return NGX_CONF_ERROR;
215         }
216
217         return NGX_CONF_OK;
218     }
219
220     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
221         "invalid parameter \"%V\"", &value[1]);
222
223     return NGX_CONF_ERROR;
224 }
225
226
227 static ngx_int_t
228 ngx_http_degradation_init(ngx_conf_t *cf)
229 {
230     ngx_http_handler_pt *h;
231     ngx_http_core_main_conf_t *cmcf;
232
233     cmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_core_module);
234
235     h = ngx_array_push(&cmcf->phases[NGX_HTTP_PREACCESS_PHASE].handlers);
236     if (h == NULL) {
237         return NGX_ERROR;
238     }
239
240     *h = ngx_http_degradation_handler;
241
242     return NGX_OK;
243 }

```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_spinlock.c - nginx-1.7.10

### Functions defined

- [ngx\\_spinlock](#)

### Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10
11
12 void
13 ngx_spinlock(ngx_atomic_t *lock, ngx_atomic_int_t value, ngx_uint_t spin)
14 {
15
16  #if (NGX_HAVE_ATOMIC_OPS)
17
18  ngx_uint_t i, n;
19
20  for ( ;; ) {
21
22  if (*lock == 0 && ngx_atomic_cmp_set(lock, 0, value)) {
23  return;
24  }
25
26  if (ngx_ncpu > 1) {
27
28  for (n = 1; n < spin; n <= 1) {
29
30  for (i = 0; i < n; i++) {
31  ngx_cpu_pause();
32  }
33
34  if (*lock == 0 && ngx_atomic_cmp_set(lock, 0, value)) {
35  return;
36  }
37  }
38  }
39
40  ngx_sched_yield();
41  }
42
43 #else
44
45 #if (NGX_THREADS)
46
47 #error ngx_spinlock() or ngx_atomic_cmp_set() are not defined !
48
49 #endif
50
51 #endif
52
53 }
```

# src/core/nginx\_syslog.h - nginx-1.7.10

## Data types defined

- [ngx\\_syslog\\_peer\\_t](#)

## Macros defined

- [\\_NGX\\_SYSLOG\\_H\\_INCLUDED\\_](#)

## Source code

```
1
2 /*
3  * Copyright (C) Nginx, Inc.
4  */
5
6
7 #ifndef \_NGX\_SYSLOG\_H\_INCLUDED\_
8 #define \_NGX\_SYSLOG\_H\_INCLUDED\_
9
10
11 typedef struct {
12     ngx\_pool\_t      *pool;
13     ngx\_uint\_t     facility;
14     ngx\_uint\_t     severity;
15     ngx\_str\_t      tag;
16
17     ngx\_addr\_t     server;
18     ngx\_connection\_t conn;
19     ngx\_uint\_t     busy; /* unsigned busy:1; */
20 } ngx_syslog_peer_t;
21
22
23 char *ngx\_syslog\_process\_conf(ngx\_conf\_t *cf, ngx\_syslog\_peer\_t *peer);
24 u_char *ngx\_syslog\_add\_header(ngx\_syslog\_peer\_t *peer, u_char *buf);
25 void ngx\_syslog\_writer(ngx\_log\_t *log, ngx\_uint\_t level, u_char *buf,
26     size_t len);
27 ssize_t ngx\_syslog\_send(ngx\_syslog\_peer\_t *peer, u_char *buf, size_t len);
28
29
30 #endif /* \_NGX\_SYSLOG\_H\_INCLUDED\_ */
```

## src/core/nginx\_syslog.c - nginx-1.7.10

### Global variables defined

- [facilities](#)
- [ngx\\_syslog\\_dummy\\_event](#)
- [ngx\\_syslog\\_dummy\\_log](#)
- [severities](#)

### Functions defined

- [ngx\\_syslog\\_add\\_header](#)
- [ngx\\_syslog\\_cleanup](#)
- [ngx\\_syslog\\_init\\_peer](#)
- [ngx\\_syslog\\_parse\\_args](#)
- [ngx\\_syslog\\_process\\_conf](#)
- [ngx\\_syslog\\_send](#)
- [ngx\\_syslog\\_writer](#)

### Macros defined

- [NGX\\_SYSLOG\\_MAX\\_STR](#)

### Source code

```
1
2 /*
3  * Copyright (C) Nginx, Inc.
4  */
5
6
7 #include <ngx_config.h>
8 #include <ngx_core.h>
9 #include <ngx_event.h>
10
11
12 #define NGX_SYSLOG_MAX_STR                                \
13     NGX_MAX_ERROR_STR + sizeof("<255>Jan 01 00:00:00 ") - 1 \
14     + (NGX_MAXHOSTNAMELEN - 1) + 1 /* space */           \
15     + 32 /* tag */ + 2 /* colon, space */
16
17
18 static char *ngx_syslog_parse_args(ngx_conf_t *cf, ngx_syslog_peer_t *peer);
19 static ngx_int_t ngx_syslog_init_peer(ngx_syslog_peer_t *peer);
20 static void ngx_syslog_cleanup(void *data);
21
22
23 static char *facilities[] = {
24     "kern", "user", "mail", "daemon", "auth", "intern", "lpr", "news", "uucp",
25     "clock", "authpriv", "ftp", "ntp", "audit", "alert", "cron", "local0",
26     "local1", "local2", "local3", "local4", "local5", "local6", "local7",
27     NULL
28 };
29
30 /* note 'error/warn' like in nginx.conf, not 'err/warning' */
```

```

31 static char *severities[] = {
32     "emerg", "alert", "crit", "error", "warn", "notice", "info", "debug", NULL
33 };
34
35 static ngx_log_t    ngx_syslog_dummy_log;
36 static ngx_event_t  ngx_syslog_dummy_event;
37
38
39 char *
40 ngx_syslog_process_conf(ngx_conf_t *cf, ngx_syslog_peer_t *peer)
41 {
42     peer->pool = cf->pool;
43     peer->facility = NGX_CONF_UNSET_UINT;
44     peer->severity = NGX_CONF_UNSET_UINT;
45
46     if (ngx_syslog_parse_args(cf, peer) != NGX_CONF_OK) {
47         return NGX_CONF_ERROR;
48     }
49
50     if (peer->server.sockaddr == NULL) {
51         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
52             "no syslog server specified");
53         return NGX_CONF_ERROR;
54     }
55
56     if (peer->facility == NGX_CONF_UNSET_UINT) {
57         peer->facility = 23; /* local7 */
58     }
59
60     if (peer->severity == NGX_CONF_UNSET_UINT) {
61         peer->severity = 6; /* info */
62     }
63
64     if (peer->tag.data == NULL) {
65         ngx_str_set(&peer->tag, "nginx");
66     }
67
68     peer->conn.fd = (ngx_socket_t) -1;
69
70     return NGX_CONF_OK;
71 }
72
73
74 static char *
75 ngx_syslog_parse_args(ngx_conf_t *cf, ngx_syslog_peer_t *peer)
76 {
77     u_char    *p, *comma, c;
78     size_t    len;
79     ngx_str_t  *value;
80     ngx_url_t  u;
81     ngx_uint_t i;
82
83     value = cf->args->elts;
84
85     p = value[1].data + sizeof("syslog:") - 1;
86
87     for ( ;; ) {
88         comma = (u_char *) ngx_strchr(p, ',');
89
90         if (comma != NULL) {
91             len = comma - p;
92             *comma = '\0';
93
94         } else {
95             len = value[1].data + value[1].len - p;
96         }
97
98         if (ngx_strncmp(p, "server=", 7) == 0) {
99
100             if (peer->server.sockaddr != NULL) {
101                 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
102                     "duplicate syslog \"server\"");
103                 return NGX_CONF_ERROR;
104             }
105
106             ngx_memzero(&u, sizeof(ngx_url_t));

```

```

107     u.url.data = p + 7;
108     u.url.len = len - 7;
109     u.default_port = 514;
110
111     if (ngx_parse_url(cf->pool, &u) != NGX_OK) {
112         if (u.err) {
113             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
114                 "%s in syslog server \"%V\"",
115                 u.err, &u.url);
116         }
117
118         return NGX_CONF_ERROR;
119     }
120
121     peer->server = u.addrs[0];
122
123 } else if (ngx_strncmp(p, "facility=", 9) == 0) {
124
125     if (peer->facility != NGX_CONF_UNSET_UINT) {
126         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
127             "duplicate syslog \"%facility\"");
128         return NGX_CONF_ERROR;
129     }
130
131     for (i = 0; facilities[i] != NULL; i++) {
132
133         if (ngx_strcmp(p + 9, facilities[i]) == 0) {
134             peer->facility = i;
135             goto next;
136         }
137     }
138
139     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
140         "unknown syslog facility \"%s\"", p + 9);
141     return NGX_CONF_ERROR;
142
143 } else if (ngx_strncmp(p, "severity=", 9) == 0) {
144
145     if (peer->severity != NGX_CONF_UNSET_UINT) {
146         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
147             "duplicate syslog \"%severity\"");
148         return NGX_CONF_ERROR;
149     }
150
151     for (i = 0; severities[i] != NULL; i++) {
152
153         if (ngx_strcmp(p + 9, severities[i]) == 0) {
154             peer->severity = i;
155             goto next;
156         }
157     }
158
159     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
160         "unknown syslog severity \"%s\"", p + 9);
161     return NGX_CONF_ERROR;
162
163 } else if (ngx_strncmp(p, "tag=", 4) == 0) {
164
165     if (peer->tag.data != NULL) {
166         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
167             "duplicate syslog \"%tag\"");
168         return NGX_CONF_ERROR;
169     }
170
171     /*
172     * RFC 3164: the TAG is a string of ABNF alphanumeric characters
173     * that MUST NOT exceed 32 characters.
174     */
175     if (len - 4 > 32) {
176         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
177             "syslog tag length exceeds 32");
178         return NGX_CONF_ERROR;
179     }
180
181     for (i = 4; i < len; i++) {

```

```

183         c = ngx_tolower(p[i]);
184
185         if (c < '0' || (c > '9' && c < 'a' && c != '_') || c > 'z') {
186             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
187                 "syslog \"tag\" only allows "
188                 "alphanumeric characters "
189                 "and underscore");
190             return NGX_CONF_ERROR;
191         }
192     }
193
194     peer->tag.data = p + 4;
195     peer->tag.len = len - 4;
196
197     } else {
198         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
199             "unknown syslog parameter \"%s\"", p);
200         return NGX_CONF_ERROR;
201     }
202
203 next:
204
205     if (comma == NULL) {
206         break;
207     }
208
209     p = comma + 1;
210 }
211
212 return NGX_CONF_OK;
213 }
214
215
216 u_char *
217 ngx_syslog_add_header(ngx_syslog_peer_t *peer, u_char *buf)
218 {
219     ngx_uint_t pri;
220
221     pri = peer->facility * 8 + peer->severity;
222
223     return ngx_sprintf(buf, "<%ui>%V %V %V: ", pri, &ngx_cached_syslog_time,
224         &ngx_cycle->hostname, &peer->tag);
225 }
226
227
228 void
229 ngx_syslog_writer(ngx_log_t *log, ngx_uint_t level, u_char *buf,
230     size_t len)
231 {
232     u_char          *p, msg[NGX_SYSLOG_MAX_STR];
233     ngx_uint_t      head_len;
234     ngx_syslog_peer_t *peer;
235
236     peer = log->wdata;
237
238     if (peer->busy) {
239         return;
240     }
241
242     peer->busy = 1;
243     peer->severity = level - 1;
244
245     p = ngx_syslog_add_header(peer, msg);
246     head_len = p - msg;
247
248     len -= NGX_LINEFEED_SIZE;
249
250     if (len > NGX_SYSLOG_MAX_STR - head_len) {
251         len = NGX_SYSLOG_MAX_STR - head_len;
252     }
253
254     p = ngx_snprintf(p, len, "%s", buf);
255
256     (void) ngx_syslog_send(peer, msg, p - msg);
257
258     peer->busy = 0;

```



```

259 }
260
261
262 ssize_t
263 ngx_syslog_send(ngx_syslog_peer_t *peer, u_char *buf, size_t len)
264 {
265     ssize_t n;
266
267     if (peer->conn.fd == (ngx_socket_t) -1) {
268         if (ngx_syslog_init_peer(peer) != NGX_OK) {
269             return NGX_ERROR;
270         }
271     }
272
273     /* log syslog socket events with valid log */
274     peer->conn.log = ngx_cycle->log;
275
276     if (ngx_send) {
277         n = ngx_send(&peer->conn, buf, len);
278     } else {
279         /* event module has not yet set ngx_io */
280         n = ngx_os_io.send(&peer->conn, buf, len);
281     }
282 }
283
284 #if (NGX_HAVE_UNIX_DOMAIN)
285
286 if (n == NGX_ERROR && peer->server.sockaddr->sa_family == AF_UNIX) {
287
288     if (ngx_close_socket(peer->conn.fd) == -1) {
289         ngx_log_error(NGX_LOG_ALERT, ngx_cycle->log, ngx_socket_errno,
290                     ngx_close_socket_n " failed");
291     }
292
293     peer->conn.fd = (ngx_socket_t) -1;
294 }
295
296 #endif
297
298     return n;
299 }
300
301
302 static ngx_int_t
303 ngx_syslog_init_peer(ngx_syslog_peer_t *peer)
304 {
305     ngx_socket_t fd;
306     ngx_pool_cleanup_t *cfn;
307
308     peer->conn.read = &ngx_syslog_dummy_event;
309     peer->conn.write = &ngx_syslog_dummy_event;
310
311     ngx_syslog_dummy_event.log = &ngx_syslog_dummy_log;
312
313     fd = ngx_socket(peer->server.sockaddr->sa_family, SOCK_DGRAM, 0);
314     if (fd == (ngx_socket_t) -1) {
315         ngx_log_error(NGX_LOG_ALERT, ngx_cycle->log, ngx_socket_errno,
316                     ngx_socket_n " failed");
317         return NGX_ERROR;
318     }
319
320     if (ngx_nonblocking(fd) == -1) {
321         ngx_log_error(NGX_LOG_ALERT, ngx_cycle->log, ngx_socket_errno,
322                     ngx_nonblocking_n " failed");
323         goto failed;
324     }
325
326     if (connect(fd, peer->server.sockaddr, peer->server.socklen) == -1) {
327         ngx_log_error(NGX_LOG_ALERT, ngx_cycle->log, ngx_socket_errno,
328                     "connect() failed");
329         goto failed;
330     }
331
332     cfn = ngx_pool_cleanup_add(peer->pool, 0);
333     if (cfn == NULL) {
334         goto failed;

```

```

335     }
336
337     cln->data = peer;
338     cln->handler = ngx\_syslog\_cleanup;
339
340     peer->conn.fd = fd;
341
342     /* UDP sockets are always ready to write */
343     peer->conn.write->ready = 1;
344
345     return NGX\_OK;
346
347 failed:
348
349     if (ngx\_close\_socket(fd) == -1) {
350         ngx\_log\_error(NGX\_LOG\_ALERT, ngx\_cycle->log, ngx\_socket\_errno,
351                     ngx\_close\_socket\_n " failed");
352     }
353
354     return NGX\_ERROR;
355 }
356
357
358 static void
359 ngx\_syslog\_cleanup(void *data)
360 {
361     ngx\_syslog\_peer\_t *peer = data;
362
363     /* prevents further use of this peer */
364     peer->busy = 1;
365
366     if (peer->conn.fd == (ngx\_socket\_t) -1) {
367         return;
368     }
369
370     if (ngx\_close\_socket(peer->conn.fd) == -1) {
371         ngx\_log\_error(NGX\_LOG\_ALERT, ngx\_cycle->log, ngx\_socket\_errno,
372                     ngx\_close\_socket\_n " failed");
373     }
374 }

```

[One Level Up](#)

[Top Level](#)

## src/event/modules/nginx\_devpoll\_module.c - nginx-1.7.10

### Global variables defined

- [change\\_index](#)
- [change\\_list](#)
- [devpoll\\_name](#)
- [dp](#)
- [event\\_list](#)
- [max\\_changes](#)
- [nchanges](#)
- [nevents](#)
- [ngx\\_devpoll\\_commands](#)
- [ngx\\_devpoll\\_module](#)
- [ngx\\_devpoll\\_module\\_ctx](#)

### Data types defined

- [dvpoll](#)
- [ngx\\_devpoll\\_conf\\_t](#)

### Functions defined

- [ngx\\_devpoll\\_add\\_event](#)
- [ngx\\_devpoll\\_create\\_conf](#)
- [ngx\\_devpoll\\_del\\_event](#)
- [ngx\\_devpoll\\_done](#)
- [ngx\\_devpoll\\_init](#)
- [ngx\\_devpoll\\_init\\_conf](#)
- [ngx\\_devpoll\\_process\\_events](#)
- [ngx\\_devpoll\\_set\\_event](#)

### Macros defined

- [DP\\_ISPOLLED](#)
- [DP\\_POLL](#)
- [POLLREMOVE](#)

### Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 #if (NGX_TEST_BUILD_DEVPOLL)
14
15 /* Solaris declarations */
16
17 #define POLLREMOVE    0x0800
18 #define DP_POLL       0xD001
19 #define DP_ISPOLLED   0xD002
20
21 struct dvpoll {
22     struct pollfd *dp_fds;
23     int dp_nfds;
24     int dp_timeout;
25 };
26
27 #endif
28
29
30 typedef struct {
31     ngx_uint_t changes;
32     ngx_uint_t events;
33 } ngx_devpoll_conf_t;
34
35
36 static ngx_int_t ngx_devpoll_init(ngx_cycle_t *cycle, ngx_msec_t timer);
37 static void ngx_devpoll_done(ngx_cycle_t *cycle);
38 static ngx_int_t ngx_devpoll_add_event(ngx_event_t *ev, ngx_int_t event,
39     ngx_uint_t flags);
40 static ngx_int_t ngx_devpoll_del_event(ngx_event_t *ev, ngx_int_t event,
41     ngx_uint_t flags);
42 static ngx_int_t ngx_devpoll_set_event(ngx_event_t *ev, ngx_int_t event,
43     ngx_uint_t flags);
44 static ngx_int_t ngx_devpoll_process_events(ngx_cycle_t *cycle,
45     ngx_msec_t timer, ngx_uint_t flags);
46
47 static void *ngx_devpoll_create_conf(ngx_cycle_t *cycle);
48 static char *ngx_devpoll_init_conf(ngx_cycle_t *cycle, void *conf);
49
50 static int dp = -1;
51 static struct pollfd *change_list, *event_list;
52 static ngx_uint_t nchanges, max_changes, nevents;
53
54 static ngx_event_t **change_index;
55
56
57 static ngx_str_t devpoll_name = ngx_string("/dev/poll");
58
59 static ngx_command_t ngx_devpoll_commands[] = {
60
61     { ngx_string("devpoll_changes"),
62       NGX_EVENT_CONF|NGX_CONF_TAKE1,
63       ngx_conf_set_num_slot,
64       0,
65       offsetof(ngx_devpoll_conf_t, changes),
66       NULL },
67
68     { ngx_string("devpoll_events"),
69       NGX_EVENT_CONF|NGX_CONF_TAKE1,
70       ngx_conf_set_num_slot,
71       0,
72       offsetof(ngx_devpoll_conf_t, events),
73       NULL },
74
75     ngx_null_command
76 };

```

```

77
78
79 ngx_event_module_t ngx_devpoll_module_ctx = {
80     &devpoll_name,
81     ngx_devpoll_create_conf,          /* create configuration */
82     ngx_devpoll_init_conf,          /* init configuration */
83
84     {
85         ngx_devpoll_add_event,      /* add an event */
86         ngx_devpoll_del_event,      /* delete an event */
87         ngx_devpoll_add_event,      /* enable an event */
88         ngx_devpoll_del_event,      /* disable an event */
89         NULL,                        /* add an connection */
90         NULL,                        /* delete an connection */
91         NULL,                        /* process the changes */
92         ngx_devpoll_process_events, /* process the events */
93         ngx_devpoll_init,           /* init the events */
94         ngx_devpoll_done,           /* done the events */
95     }
96 };
97
98
99 ngx_module_t ngx_devpoll_module = {
100     NGX_MODULE_V1,
101     &ngx_devpoll_module_ctx,        /* module context */
102     ngx_devpoll_commands,          /* module directives */
103     NGX_EVENT_MODULE,              /* module type */
104     NULL,                           /* init master */
105     NULL,                           /* init module */
106     NULL,                           /* init process */
107     NULL,                           /* init thread */
108     NULL,                           /* exit thread */
109     NULL,                           /* exit process */
110     NULL,                           /* exit master */
111     NGX_MODULE_V1_PADDING
112 };
113
114
115 static ngx_int_t
116 ngx_devpoll_init(ngx_cycle_t *cycle, ngx_msec_t timer)
117 {
118     size_t      n;
119     ngx_devpoll_conf_t *dpcf;
120
121     dpcf = ngx_event_get_conf(cycle->conf_ctx, ngx_devpoll_module);
122
123     if (dp == -1) {
124         dp = open("/dev/poll", O_RDWR);
125
126         if (dp == -1) {
127             ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
128                 "open(/dev/poll) failed");
129             return NGX_ERROR;
130         }
131     }
132
133     if (max_changes < dpcf->changes) {
134         if (nchanges) {
135             n = nchanges * sizeof(struct pollfd);
136             if (write(dp, change_list, n) != (ssize_t) n) {
137                 ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
138                     "write(/dev/poll) failed");
139                 return NGX_ERROR;
140             }
141
142             nchanges = 0;
143         }
144
145         if (change_list) {
146             ngx_free(change_list);
147         }
148
149         change_list = ngx_alloc(sizeof(struct pollfd) * dpcf->changes,
150             cycle->log);
151         if (change_list == NULL) {
152             return NGX_ERROR;

```

```

153     }
154
155     if (change_index) {
156         ngx_free(change_index);
157     }
158
159     change_index = ngx_alloc(sizeof(ngx_event_t) * dpcf->changes,
160                             cycle->log);
161     if (change_index == NULL) {
162         return NGX_ERROR;
163     }
164 }
165
166 max_changes = dpcf->changes;
167
168 if (nevents < dpcf->events) {
169     if (event_list) {
170         ngx_free(event_list);
171     }
172
173     event_list = ngx_alloc(sizeof(struct pollfd) * dpcf->events,
174                             cycle->log);
175     if (event_list == NULL) {
176         return NGX_ERROR;
177     }
178 }
179
180 nevents = dpcf->events;
181
182 ngx_io = ngx_os_io;
183
184 ngx_event_actions = ngx_devpoll_module.ctx.actions;
185
186 ngx_event_flags = NGX_USE_LEVEL_EVENT|NGX_USE_FD_EVENT;
187
188 return NGX_OK;
189 }
190
191
192 static void
193 ngx_devpoll_done(ngx_cycle_t *cycle)
194 {
195     if (close(dp) == -1) {
196         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
197                     "close(/dev/poll) failed");
198     }
199
200     dp = -1;
201
202     ngx_free(change_list);
203     ngx_free(event_list);
204     ngx_free(change_index);
205
206     change_list = NULL;
207     event_list = NULL;
208     change_index = NULL;
209     max_changes = 0;
210     nchanges = 0;
211     nevents = 0;
212 }
213
214
215 static ngx_int_t
216 ngx_devpoll_add_event(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags)
217 {
218     #if (NGX_DEBUG)
219         ngx_connection_t *c;
220     #endif
221
222     #if (NGX_READ_EVENT != POLLIN)
223         event = (event == NGX_READ_EVENT) ? POLLIN : POLLOUT;
224     #endif
225
226     #if (NGX_DEBUG)
227         c = ev->data;
228         ngx_log_debug2(NGX_LOG_DEBUG_EVENT, ev->log, 0,

```

```

229         "devpoll add event: fd:%d ev:%04Xi", c->fd, event);
230 #endif
231
232     ev->active = 1;
233
234     return ngx_devpoll_set_event(ev, event, 0);
235 }
236
237
238 static ngx_int_t
239 ngx_devpoll_del_event(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags)
240 {
241     ngx_event_t     *e;
242     ngx_connection_t *c;
243
244     c = ev->data;
245
246     #if (NGX_READ_EVENT != POLLIN)
247     event = (event == NGX_READ_EVENT) ? POLLIN : POLLOUT;
248     #endif
249
250     ngx_log_debug2(NGX_LOG_DEBUG_EVENT, ev->log, 0,
251                  "devpoll del event: fd:%d ev:%04Xi", c->fd, event);
252
253     if (ngx_devpoll_set_event(ev, POLLREMOVE, flags) == NGX_ERROR) {
254         return NGX_ERROR;
255     }
256
257     ev->active = 0;
258
259     if (flags & NGX_CLOSE_EVENT) {
260         e = (event == POLLIN) ? c->write : c->read;
261
262         if (e) {
263             e->active = 0;
264         }
265
266         return NGX_OK;
267     }
268
269     /* restore the pair event if it exists */
270
271     if (event == POLLIN) {
272         e = c->write;
273         event = POLLOUT;
274     } else {
275         e = c->read;
276         event = POLLIN;
277     }
278
279     if (e && e->active) {
280         return ngx_devpoll_set_event(e, event, 0);
281     }
282
283     return NGX_OK;
284 }
285
286
287
288 static ngx_int_t
289 ngx_devpoll_set_event(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags)
290 {
291     size_t          n;
292     ngx_connection_t *c;
293
294     c = ev->data;
295
296     ngx_log_debug3(NGX_LOG_DEBUG_EVENT, ev->log, 0,
297                  "devpoll fd:%d ev:%04Xi fl:%04Xi", c->fd, event, flags);
298
299     if (nchanges >= max_changes) {
300         ngx_log_error(NGX_LOG_WARN, ev->log, 0,
301                      "/dev/pool change list is filled up");
302
303         n = nchanges * sizeof(struct pollfd);
304         if (write(dp, change_list, n) != (ssize_t) n) {

```

```

305     ngx_log_error(NGX_LOG_ALERT, ev->log, ngx_errno,
306                 "write(/dev/poll) failed");
307     return NGX_ERROR;
308 }
309
310     nchanges = 0;
311 }
312
313     change_list[nchanges].fd = c->fd;
314     change_list[nchanges].events = (short) event;
315     change_list[nchanges].revents = 0;
316
317     change_index[nchanges] = ev;
318     ev->index = nchanges;
319
320     nchanges++;
321
322     if (flags & NGX_CLOSE_EVENT) {
323         n = nchanges * sizeof(struct pollfd);
324         if (write(dp, change_list, n) != (ssize_t) n) {
325             ngx_log_error(NGX_LOG_ALERT, ev->log, ngx_errno,
326                         "write(/dev/poll) failed");
327             return NGX_ERROR;
328         }
329
330         nchanges = 0;
331     }
332
333     return NGX_OK;
334 }
335
336 ngx_int_t
337 ngx_devpoll_process_events(ngx_cycle_t *cycle, ngx_msec_t timer,
338 ngx_uint_t flags)
339 {
340     int             events, revents, rc;
341     size_t          n;
342     ngx_fd_t        fd;
343     ngx_err_t        err;
344     ngx_int_t        i;
345     ngx_uint_t        level, instance;
346     ngx_event_t      *rev, *wev;
347     ngx_queue_t      *queue;
348     ngx_connection_t *c;
349     struct pollfd    pfd;
350     struct dvpoll    dvp;
351
352     /* NGX_TIMER_INFINITE == INFTIM */
353
354     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
355                 "devpoll timer: %M", timer);
356
357     if (nchanges) {
358         n = nchanges * sizeof(struct pollfd);
359         if (write(dp, change_list, n) != (ssize_t) n) {
360             ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
361                         "write(/dev/poll) failed");
362             return NGX_ERROR;
363         }
364     }
365
366     nchanges = 0;
367 }
368
369     dvp.dp_fds = event_list;
370     dvp.dp_nfds = (int) nevents;
371     dvp.dp_timeout = timer;
372     events = ioctl(dp, DP_POLL, &dvp);
373
374     err = (events == -1) ? ngx_errno : 0;
375
376     if (flags & NGX_UPDATE_TIME || ngx_event_timer_alarm) {
377         ngx_time_update();
378     }
379
380     if (err) {

```





```

457     }
458
459     break;
460 }
461
462     continue;
463 }
464
465     ngx_log_debug3(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
466         "devpoll: fd:%d, ev:%04Xd, rev:%04Xd",
467         fd, event_list[i].events, revents);
468
469     if (revents & (POLLERR|POLLHUP|POLLNVAL)) {
470         ngx_log_debug3(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
471             "ioctl(DP_POLL) error fd:%d ev:%04Xd rev:%04Xd",
472             fd, event_list[i].events, revents);
473     }
474
475     if (revents & ~(POLLIN|POLLOUT|POLLERR|POLLHUP|POLLNVAL)) {
476         ngx_log_error(NGX_LOG_ALERT, cycle->log, 0,
477             "strange ioctl(DP_POLL) events "
478             "fd:%d ev:%04Xd rev:%04Xd",
479             fd, event_list[i].events, revents);
480     }
481
482     if ((revents & (POLLERR|POLLHUP|POLLNVAL))
483         && (revents & (POLLIN|POLLOUT)) == 0)
484     {
485         /*
486          * if the error events were returned without POLLIN or POLLOUT,
487          * then add these flags to handle the events at least in one
488          * active handler
489          */
490
491         revents |= POLLIN|POLLOUT;
492     }
493
494     rev = c->read;
495
496     if ((revents & POLLIN) && rev->active) {
497         rev->ready = 1;
498
499         if (flags & NGX_POST_EVENTS) {
500             queue = rev->accept ? &ngx_posted_accept_events
501                 : &ngx_posted_events;
502
503             ngx_post_event(rev, queue);
504
505         } else {
506             instance = rev->instance;
507
508             rev->handler(rev);
509
510             if (c->fd == -1 || rev->instance != instance) {
511                 continue;
512             }
513         }
514     }
515
516     wev = c->write;
517
518     if ((revents & POLLOUT) && wev->active) {
519         wev->ready = 1;
520
521         if (flags & NGX_POST_EVENTS) {
522             ngx_post_event(wev, &ngx_posted_events);
523
524         } else {
525             wev->handler(wev);
526         }
527     }
528 }
529
530 return NGX_OK;
531 }
532

```

```
533 static void *
534 ngx_devpoll_create_conf(ngx_cycle_t *cycle)
535 {
536     ngx_devpoll_conf_t *dpcf;
537
538     dpcf = ngx_palloc(cycle->pool, sizeof(ngx_devpoll_conf_t));
539     if (dpcf == NULL) {
540         return NULL;
541     }
542
543     dpcf->changes = NGX_CONF_UNSET;
544     dpcf->events = NGX_CONF_UNSET;
545
546     return dpcf;
547 }
548
549
550
551 static char *
552 ngx_devpoll_init_conf(ngx_cycle_t *cycle, void *conf)
553 {
554     ngx_devpoll_conf_t *dpcf = conf;
555
556     ngx_conf_init_uint_value(dpcf->changes, 32);
557     ngx_conf_init_uint_value(dpcf->events, 32);
558
559     return NGX_CONF_OK;
560 }
```

[One Level Up](#)

[Top Level](#)

## src/event/nginx\_event\_busy\_lock.c - nginx-1.7.10

### Functions defined

- [ngx\\_event\\_busy\\_lock](#)
- [ngx\\_event\\_busy\\_lock\\_cacheable](#)
- [ngx\\_event\\_busy\\_lock\\_cancel](#)
- [ngx\\_event\\_busy\\_lock\\_handler](#)
- [ngx\\_event\\_busy\\_lock\\_look\\_cacheable](#)
- [ngx\\_event\\_busy\\_lock\\_posted\\_handler](#)
- [ngx\\_event\\_busy\\_unlock](#)

### Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 static ngx_int_t ngx_event_busy_lock_look_cacheable(ngx_event_busy_lock_t *bl,
14     ngx_event_busy_lock_ctx_t *ctx);
15 static void ngx_event_busy_lock_handler(ngx_event_t *ev);
16 static void ngx_event_busy_lock_posted_handler(ngx_event_t *ev);
17
18
19 /*
20 * NGX_OK:    the busy lock is held
21 * NGX_AGAIN: the all busy locks are held but we will wait the specified time
22 * NGX_BUSY: ctx->timer == 0: there are many the busy locks
23 *             ctx->timer != 0: there are many the waiting locks
24 */
25
26 ngx_int_t
27 ngx_event_busy_lock(ngx_event_busy_lock_t *bl, ngx_event_busy_lock_ctx_t *ctx)
28 {
29     ngx_int_t rc;
30
31     ngx_mutex_lock(bl->mutex);
32
33     ngx_log_debug2(NGX_LOG_DEBUG_EVENT, ctx->event->log, 0,
34         "event busy lock: b:%d mb:%d",
35         bl->busy, bl->max_busy);
36
37     if (bl->busy < bl->max_busy) {
38         bl->busy++;
39
40         rc = NGX_OK;
41
42     } else if (ctx->timer && bl->waiting < bl->max_waiting) {
43         bl->waiting++;
44         ngx_add_timer(ctx->event, ctx->timer);
45         ctx->event->handler = ngx_event_busy_lock_handler;
46
47     } if (bl->events) {
48         bl->last->next = ctx;
49     }
```

```

50     } else {
51         bl->events = ctx;
52     }
53
54     bl->last = ctx;
55
56     rc = NGX\_AGAIN;
57
58 } else {
59     rc = NGX\_BUSY;
60 }
61
62 ngx\_mutex\_unlock(bl->mutex);
63
64 return rc;
65 }
66
67
68 ngx\_int\_t
69 ngx\_event\_busy\_lock\_cacheable(ngx\_event\_busy\_lock\_t *bl,
70 ngx\_event\_busy\_lock\_ctx\_t *ctx)
71 {
72     ngx\_int\_t rc;
73
74     ngx\_mutex\_lock(bl->mutex);
75
76     rc = ngx\_event\_busy\_lock\_look\_cacheable(bl, ctx);
77
78     ngx\_log\_debug3(NGX\_LOG\_DEBUG\_EVENT, ctx->event->log, 0,
79         "event busy lock: %d w:%d mw:%d",
80         rc, bl->waiting, bl->max_waiting);
81
82     /*
83     * NGX\_OK:    no the same request, there is free slot and we locked it
84     * NGX\_BUSY:  no the same request and there is no free slot
85     * NGX\_AGAIN: the same request is processing
86     */
87
88     if (rc == NGX\_AGAIN) {
89
90         if (ctx->timer && bl->waiting < bl->max_waiting) {
91             bl->waiting++;
92             ngx\_add\_timer(ctx->event, ctx->timer);
93             ctx->event->handler = ngx\_event\_busy\_lock\_handler;
94
95             if (bl->events == NULL) {
96                 bl->events = ctx;
97             } else {
98                 bl->last->next = ctx;
99             }
100             bl->last = ctx;
101
102         } else {
103             rc = NGX\_BUSY;
104         }
105     }
106
107     ngx\_mutex\_unlock(bl->mutex);
108
109     return rc;
110 }
111
112
113 void
114 ngx\_event\_busy\_unlock(ngx\_event\_busy\_lock\_t *bl,
115 ngx\_event\_busy\_lock\_ctx\_t *ctx)
116 {
117     ngx\_event\_t *ev;
118     ngx\_event\_busy\_lock\_ctx\_t *wakeup;
119
120     ngx\_mutex\_lock(bl->mutex);
121
122     if (bl->events) {
123         wakeup = bl->events;
124         bl->events = bl->events->next;
125

```

```

126 } else {
127     wakeup = NULL;
128     bl->busy--;
129 }
130
131 /*
132  * MP: all ctx's and their queue must be in shared memory,
133  *     each ctx has pid to wake up
134  */
135
136 if (wakeup == NULL) {
137     ngx_mutex_unlock(bl->mutex);
138     return;
139 }
140
141 if (ctx->md5) {
142     for (wakeup = bl->events; wakeup; wakeup = wakeup->next) {
143         if (wakeup->md5 == NULL || wakeup->slot != ctx->slot) {
144             continue;
145         }
146
147         wakeup->handler = ngx_event_busy_lock_posted_handler;
148         wakeup->cache_updated = 1;
149
150         ev = wakeup->event;
151
152         ngx_post_event(ev, &ngx_posted_events);
153     }
154
155     ngx_mutex_unlock(bl->mutex);
156 } else {
157     bl->waiting--;
158
159     ngx_mutex_unlock(bl->mutex);
160
161     wakeup->handler = ngx_event_busy_lock_posted_handler;
162     wakeup->locked = 1;
163
164     ev = wakeup->event;
165
166     if (ev->timer_set) {
167         ngx_del_timer(ev);
168     }
169
170     ngx_post_event(ev, &ngx_posted_events);
171 }
172 }
173 }
174
175 void
176 ngx_event_busy_lock_cancel(ngx_event_busy_lock_t *bl,
177 ngx_event_busy_lock_ctx_t *ctx)
178 {
179     ngx_event_busy_lock_ctx_t *c, *p;
180
181     ngx_mutex_lock(bl->mutex);
182
183     bl->waiting--;
184
185     if (ctx == bl->events) {
186         bl->events = ctx->next;
187     }
188
189     else {
190         p = bl->events;
191         for (c = bl->events->next; c; c = c->next) {
192             if (c == ctx) {
193                 p->next = ctx->next;
194                 break;
195             }
196             p = c;
197         }
198     }
199
200     ngx_mutex_unlock(bl->mutex);
201 }

```

```

202
203
204 static ngx_int_t
205 ngx_event_busy_lock_look_cacheable(ngx_event_busy_lock_t *bl,
206     ngx_event_busy_lock_ctx_t *ctx)
207 {
208     ngx_int_t    free;
209     ngx_uint_t   i, bit, cacheable, mask;
210
211     bit = 0;
212     cacheable = 0;
213     free = -1;
214
215     #if (NGX_SUPPRESS_WARN)
216     mask = 0;
217     #endif
218
219     for (i = 0; i < bl->max_busy; i++) {
220
221         if ((bit & 7) == 0) {
222             mask = bl->md5_mask[i / 8];
223         }
224
225         if (mask & 1) {
226             if (ngx_memcmp(&bl->md5[i * 16], ctx->md5, 16) == 0) {
227                 ctx->waiting = 1;
228                 ctx->slot = i;
229                 return NGX_AGAIN;
230             }
231             cacheable++;
232
233         } else if (free == -1) {
234             free = i;
235         }
236
237         if (cacheable == bl->cacheable) {
238             if (free == -1 && cacheable < bl->max_busy) {
239                 free = i + 1;
240             }
241
242             break;
243         }
244
245         mask >>= 1;
246         bit++;
247     }
248
249     if (free == -1) {
250         return NGX_BUSY;
251     }
252
253     #if 0
254     if (bl->busy == bl->max_busy) {
255         return NGX_BUSY;
256     }
257     #endif
258
259     ngx_memcpy(&bl->md5[free * 16], ctx->md5, 16);
260     bl->md5_mask[free / 8] |= 1 << (free & 7);
261     ctx->slot = free;
262
263     bl->cacheable++;
264     bl->busy++;
265
266     return NGX_OK;
267 }
268
269
270 static void
271 ngx_event_busy_lock_handler(ngx_event_t *ev)
272 {
273     ev->handler = ngx_event_busy_lock_posted_handler;
274
275     ngx_post_event(ev, &ngx_posted_events);
276 }
277

```

```
278
279 static void
280 ngx_event_busy_lock_posted_handler(ngx\_event\_t *ev)
281 {
282     ngx\_event\_busy\_lock\_ctx\_t *ctx;
283
284     ctx = ev->data;
285     ctx->handler(ev);
286 }
```

[One Level Up](#)

[Top Level](#)



# src/event/nginx\_event\_busy\_lock.h - nginx-1.7.10

## Data types defined

- [ngx\\_event\\_busy\\_lock\\_ctx\\_s](#)
- [ngx\\_event\\_busy\\_lock\\_ctx\\_t](#)
- [ngx\\_event\\_busy\\_lock\\_t](#)

## Macros defined

- [\\_NGX\\_EVENT\\_BUSY\\_LOCK\\_H\\_INCLUDED](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef _NGX_EVENT_BUSY_LOCK_H_INCLUDED
9 #define _NGX_EVENT_BUSY_LOCK_H_INCLUDED
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14 #include <ngx_event.h>
15
16 typedef struct ngx_event_busy_lock_ctx_s  ngx_event_busy_lock_ctx_t;
17
18 struct ngx_event_busy_lock_ctx_s {
19     ngx_event_t          *event;
20     ngx_event_handler_pt  handler;
21     void                 *data;
22     ngx_msec_t          timer;
23
24     unsigned             locked:1;
25     unsigned             waiting:1;
26     unsigned             cache_updated:1;
27
28     char                 *md5;
29     ngx_int_t            slot;
30
31     ngx_event_busy_lock_ctx_t *next;
32 };
33
34
35 typedef struct {
36     u_char                 *md5_mask;
37     char                   *md5;
38     ngx_uint_t             cacheable;
39
40     ngx_uint_t             busy;
41     ngx_uint_t             max_busy;
42
43     ngx_uint_t             waiting;
44     ngx_uint_t             max_waiting;
45
46     ngx_event_busy_lock_ctx_t *events;
47     ngx_event_busy_lock_ctx_t *last;
48
49 #if (NGX_THREADS)
50     ngx_mutex_t           *mutex;
51 #endif
52 } ngx_event_busy_lock_t;
```

```
53
54
55 ngx\_int\_t ngx\_event\_busy\_lock(ngx\_event\_busy\_lock\_t *bl,
56     ngx\_event\_busy\_lock\_ctx\_t *ctx);
57 ngx\_int\_t ngx\_event\_busy\_lock\_cacheable(ngx\_event\_busy\_lock\_t *bl,
58     ngx\_event\_busy\_lock\_ctx\_t *ctx);
59 void ngx\_event\_busy\_unlock(ngx\_event\_busy\_lock\_t *bl,
60     ngx\_event\_busy\_lock\_ctx\_t *ctx);
61 void ngx\_event\_busy\_lock\_cancel(ngx\_event\_busy\_lock\_t *bl,
62     ngx\_event\_busy\_lock\_ctx\_t *ctx);
63
64
65 #endif /* \_NGX\_EVENT\_BUSY\_LOCK\_H\_INCLUDED */
```

[One Level Up](#)

[Top Level](#)

## src/event/modules/nginx\_poll\_module.c - nginx-1.7.10

### Global variables defined

- [event\\_list](#)
- [nevents](#)
- [ngx\\_poll\\_module](#)
- [ngx\\_poll\\_module\\_ctx](#)
- [poll\\_name](#)

### Functions defined

- [ngx\\_poll\\_add\\_event](#)
- [ngx\\_poll\\_del\\_event](#)
- [ngx\\_poll\\_done](#)
- [ngx\\_poll\\_init](#)
- [ngx\\_poll\\_init\\_conf](#)
- [ngx\\_poll\\_process\\_events](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 static ngx_int_t ngx_poll_init(ngx_cycle_t *cycle, ngx_msec_t timer);
14 static void ngx_poll_done(ngx_cycle_t *cycle);
15 static ngx_int_t ngx_poll_add_event(ngx_event_t *ev, ngx_int_t event,
16     ngx_uint_t flags);
17 static ngx_int_t ngx_poll_del_event(ngx_event_t *ev, ngx_int_t event,
18     ngx_uint_t flags);
19 static ngx_int_t ngx_poll_process_events(ngx_cycle_t *cycle, ngx_msec_t timer,
20     ngx_uint_t flags);
21 static char *ngx_poll_init_conf(ngx_cycle_t *cycle, void *conf);
22
23
24 static struct pollfd *event_list;
25 static ngx_uint_t nevents;
26
27
28 static ngx_str_t poll_name = ngx_string("poll");
29
30 ngx_event_module_t ngx_poll_module_ctx = {
31     &poll_name,
32     NULL, /* create configuration */
33     ngx_poll_init_conf, /* init configuration */
34
35     {
36         ngx_poll_add_event, /* add an event */
```

```

37     ngx_poll_del_event,          /* delete an event */
38     ngx_poll_add_event,        /* enable an event */
39     ngx_poll_del_event,        /* disable an event */
40     NULL,                      /* add a connection */
41     NULL,                      /* delete a connection */
42     NULL,                      /* process the changes */
43     ngx_poll_process_events,    /* process the events */
44     ngx_poll_init,             /* init the events */
45     ngx_poll_done,             /* done the events */
46 }
47
48 };
49
50 ngx_module_t ngx_poll_module = {
51     NGX_MODULE_V1,
52     &ngx_poll_module_ctx,      /* module context */
53     NULL,                      /* module directives */
54     NGX_EVENT_MODULE,         /* module type */
55     NULL,                      /* init master */
56     NULL,                      /* init module */
57     NULL,                      /* init process */
58     NULL,                      /* init thread */
59     NULL,                      /* exit thread */
60     NULL,                      /* exit process */
61     NULL,                      /* exit master */
62     NGX_MODULE_V1_PADDING
63 };
64
65
66
67 static ngx_int_t
68 ngx_poll_init(ngx_cycle_t *cycle, ngx_msec_t timer)
69 {
70     struct pollfd *list;
71
72     if (event_list == NULL) {
73         nevents = 0;
74     }
75
76     if (ngx_process >= NGX_PROCESS_WORKER
77         || cycle->old_cycle == NULL
78         || cycle->old_cycle->connection_n < cycle->connection_n)
79     {
80         list = ngx_alloc(sizeof(struct pollfd) * cycle->connection_n,
81                         cycle->log);
82         if (list == NULL) {
83             return NGX_ERROR;
84         }
85
86         if (event_list) {
87             ngx_memcpy(list, event_list, sizeof(ngx_event_t *) * nevents);
88             ngx_free(event_list);
89         }
90
91         event_list = list;
92     }
93
94     ngx_io = ngx_os_io;
95
96     ngx_event_actions = ngx_poll_module_ctx.actions;
97
98     ngx_event_flags = NGX_USE_LEVEL_EVENT|NGX_USE_FD_EVENT;
99
100    return NGX_OK;
101 }
102
103
104 static void
105 ngx_poll_done(ngx_cycle_t *cycle)
106 {
107     ngx_free(event_list);
108
109     event_list = NULL;
110 }
111
112

```

```

113 static ngx_int_t
114 ngx_poll_add_event(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags)
115 {
116     ngx_event_t     *e;
117     ngx_connection_t *c;
118
119     c = ev->data;
120
121     ev->active = 1;
122
123     if (ev->index != NGX_INVALID_INDEX) {
124         ngx_log_error(NGX_LOG_ALERT, ev->log, 0,
125             "poll event fd:%d ev:%i is already set", c->fd, event);
126         return NGX_OK;
127     }
128
129     if (event == NGX_READ_EVENT) {
130         e = c->write;
131         #if (NGX_READ_EVENT != POLLIN)
132             event = POLLIN;
133         #endif
134
135     } else {
136         e = c->read;
137         #if (NGX_WRITE_EVENT != POLLOUT)
138             event = POLLOUT;
139         #endif
140     }
141
142     ngx_log_debug2(NGX_LOG_DEBUG_EVENT, ev->log, 0,
143         "poll add event: fd:%d ev:%i", c->fd, event);
144
145     if (e == NULL || e->index == NGX_INVALID_INDEX) {
146         event_list[nevents].fd = c->fd;
147         event_list[nevents].events = (short) event;
148         event_list[nevents].revents = 0;
149
150         ev->index = nevents;
151         nevents++;
152
153     } else {
154         ngx_log_debug1(NGX_LOG_DEBUG_EVENT, ev->log, 0,
155             "poll add index: %i", e->index);
156
157         event_list[e->index].events |= (short) event;
158         ev->index = e->index;
159     }
160
161     return NGX_OK;
162 }
163
164
165 static ngx_int_t
166 ngx_poll_del_event(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags)
167 {
168     ngx_event_t     *e;
169     ngx_connection_t *c;
170
171     c = ev->data;
172
173     ev->active = 0;
174
175     if (ev->index == NGX_INVALID_INDEX) {
176         ngx_log_error(NGX_LOG_ALERT, ev->log, 0,
177             "poll event fd:%d ev:%i is already deleted",
178             c->fd, event);
179         return NGX_OK;
180     }
181
182     if (event == NGX_READ_EVENT) {
183         e = c->write;
184         #if (NGX_READ_EVENT != POLLIN)
185             event = POLLIN;
186         #endif
187
188     } else {

```

```

189     e = c->read;
190     #if (NGX_WRITE_EVENT != POLLOUT)
191         event = POLLOUT;
192     #endif
193     }
194
195     ngx_log_debug2(NGX_LOG_DEBUG_EVENT, ev->log, 0,
196                 "poll del event: fd:%d ev:%i", c->fd, event);
197
198     if (e == NULL || e->index == NGX_INVALID_INDEX) {
199         nevents--;
200
201         if (ev->index < nevents) {
202
203             ngx_log_debug2(NGX_LOG_DEBUG_EVENT, ev->log, 0,
204                 "index: copy event %ui to %i", nevents, ev->index);
205
206             event_list[ev->index] = event_list[nevents];
207
208             c = ngx_cycle->files[event_list[nevents].fd];
209
210             if (c->fd == -1) {
211                 ngx_log_error(NGX_LOG_ALERT, ev->log, 0,
212                     "unexpected last event");
213
214             } else {
215                 if (c->read->index == nevents) {
216                     c->read->index = ev->index;
217                 }
218
219                 if (c->write->index == nevents) {
220                     c->write->index = ev->index;
221                 }
222             }
223         }
224     } else {
225         ngx_log_debug1(NGX_LOG_DEBUG_EVENT, ev->log, 0,
226             "poll del index: %i", e->index);
227
228         event_list[e->index].events &= (short) ~event;
229     }
230
231     ev->index = NGX_INVALID_INDEX;
232
233     return NGX_OK;
234 }
235
236
237
238 static ngx_int_t
239 ngx_poll_process_events(ngx_cycle_t *cycle, ngx_msec_t timer, ngx_uint_t flags)
240 {
241     int                ready, revents;
242     ngx_err_t         err;
243     ngx_uint_t        i, found, level;
244     ngx_event_t       *ev;
245     ngx_queue_t       *queue;
246     ngx_connection_t  *c;
247
248     /* NGX_TIMER_INFINITE == INFTIM */
249
250     #if (NGX_DEBUG0)
251     if (cycle->log->log_level & NGX_LOG_DEBUG_ALL) {
252         for (i = 0; i < nevents; i++) {
253             ngx_log_debug3(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
254                 "poll: %ui: fd:%d ev:%04Xd",
255                 i, event_list[i].fd, event_list[i].events);
256         }
257     }
258     #endif
259
260     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0, "poll timer: %M", timer);
261
262     ready = poll(event_list, (u_int) nevents, (int) timer);
263
264     err = (ready == -1) ? ngx_errno : 0;

```

```

265
266     if (flags & NGX_UPDATE_TIME || ngx_event_timer_alarm) {
267         ngx_time_update();
268     }
269
270     ngx_log_debug2(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
271                 "poll ready %d of %ui", ready, nevents);
272
273     if (err) {
274         if (err == NGX_EINTR) {
275
276             if (ngx_event_timer_alarm) {
277                 ngx_event_timer_alarm = 0;
278                 return NGX_OK;
279             }
280
281             level = NGX_LOG_INFO;
282
283         } else {
284             level = NGX_LOG_ALERT;
285         }
286
287         ngx_log_error(level, cycle->log, err, "poll() failed");
288         return NGX_ERROR;
289     }
290
291     if (ready == 0) {
292         if (timer != NGX_TIMER_INFINITE) {
293             return NGX_OK;
294         }
295
296         ngx_log_error(NGX_LOG_ALERT, cycle->log, 0,
297                 "poll() returned no events without timeout");
298         return NGX_ERROR;
299     }
300
301     for (i = 0; i < nevents && ready; i++) {
302
303         revents = event_list[i].revents;
304
305         #if 1
306         ngx_log_debug4(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
307                 "poll: %ui: fd:%d ev:%04Xd rev:%04Xd",
308                 i, event_list[i].fd, event_list[i].events, revents);
309         #else
310         if (revents) {
311             ngx_log_debug4(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
312                 "poll: %ui: fd:%d ev:%04Xd rev:%04Xd",
313                 i, event_list[i].fd, event_list[i].events, revents);
314         }
315         #endif
316
317         if (revents & POLLNVAL) {
318             ngx_log_error(NGX_LOG_ALERT, cycle->log, 0,
319                 "poll() error fd:%d ev:%04Xd rev:%04Xd",
320                 event_list[i].fd, event_list[i].events, revents);
321         }
322
323         if (revents & ~(POLLIN|POLLOUT|POLLERR|POLLHUP|POLLNVAL)) {
324             ngx_log_error(NGX_LOG_ALERT, cycle->log, 0,
325                 "strange poll() events fd:%d ev:%04Xd rev:%04Xd",
326                 event_list[i].fd, event_list[i].events, revents);
327         }
328
329         if (event_list[i].fd == -1) {
330             /*
331              * the disabled event, a workaround for our possible bug,
332              * see the comment below
333              */
334             continue;
335         }
336
337         c = ngx_cycle->files[event_list[i].fd];
338
339         if (c->fd == -1) {
340             ngx_log_error(NGX_LOG_ALERT, cycle->log, 0, "unexpected event");

```

```

341
342     /*
343     * it is certainly our fault and it should be investigated,
344     * in the meantime we disable this event to avoid a CPU spinning
345     */
346
347     if (i == nevents - 1) {
348         nevents--;
349     } else {
350         event\_list[i].fd = -1;
351     }
352
353     continue;
354 }
355
356 if ((revents & (POLLERR|POLLHUP|POLLNVAL))
357     && (revents & (POLLIN|POLLOUT)) == 0)
358 {
359     /*
360     * if the error events were returned without POLLIN or POLLOUT,
361     * then add these flags to handle the events at least in one
362     * active handler
363     */
364
365     revents |= POLLIN|POLLOUT;
366 }
367
368 found = 0;
369
370 if ((revents & POLLIN) && c->read->active) {
371     found = 1;
372
373     ev = c->read;
374     ev->ready = 1;
375
376     queue = ev->accept ? &ngx\_posted\_accept\_events
377                       : &ngx\_posted\_events;
378
379     ngx\_post\_event(ev, queue);
380 }
381
382 if ((revents & POLLOUT) && c->write->active) {
383     found = 1;
384
385     ev = c->write;
386     ev->ready = 1;
387
388     ngx\_post\_event(ev, &ngx\_posted\_events);
389 }
390
391 if (found) {
392     ready--;
393     continue;
394 }
395 }
396
397 if (ready != 0) {
398     ngx\_log\_error(NGX\_LOG\_ALERT, cycle->log, 0, "poll ready != events");
399 }
400
401 return NGX\_OK;
402 }
403
404
405 static char *
406 ngx\_poll\_init\_conf(ngx\_cycle\_t *cycle, void *conf)
407 {
408     ngx\_event\_conf\_t *ecf;
409
410     ecf = ngx\_event\_get\_conf(cycle->conf_ctx, ngx\_event\_core\_module);
411
412     if (ecf->use != ngx\_poll\_module.ctx_index) {
413         return NGX\_CONF\_OK;
414     }
415
416     #if (NGX_THREADS)

```



```
417     ngx_log_error(NGX_LOG_EMERG, cycle->log, 0,  
418                 "poll() is not supported in the threaded mode");  
419     return NGX_CONF_ERROR;  
421  
422 #else  
423  
424     return NGX_CONF_OK;  
425  
426 #endif  
427 }
```

[One Level Up](#)

[Top Level](#)

# src/os/unix/nginx\_linux\_init.c - nginx-1.7.10

## Global variables defined

- [ngx\\_linux\\_io](#)
- [ngx\\_linux\\_kern\\_osrelease](#)
- [ngx\\_linux\\_kern\\_ostype](#)
- [ngx\\_linux\\_rtsig\\_max](#)

## Functions defined

- [ngx\\_os\\_specific\\_init](#)
- [ngx\\_os\\_specific\\_status](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10
11
12 u_char  ngx_linux_kern_ostype[50];
13 u_char  ngx_linux_kern_osrelease[50];
14
15 int     ngx_linux_rtsig_max;
16
17
18 static ngx_os_io_t ngx_linux_io = {
19     ngx_unix_recv,
20     ngx_readv_chain,
21     ngx_udp_unix_recv,
22     ngx_unix_send,
23     #if (NGX_HAVE_SENDFILE)
24     ngx_linux_sendfile_chain,
25     NGX_IO_SENDFILE
26 #else
27     ngx_writev_chain,
28     0
29 #endif
30 };
31
32
33 ngx_int_t
34 ngx_os_specific_init(ngx_log_t *log)
35 {
36     struct utsname u;
37
38     if (uname(&u) == -1) {
39         ngx_log_error(NGX_LOG_ALERT, log, ngx_errno, "uname() failed");
40         return NGX_ERROR;
41     }
42
43     (void) ngx_cpystyrn(ngx_linux_kern_ostype, (u_char *) u.sysname,
44                       sizeof(ngx_linux_kern_ostype));
45
46     (void) ngx_cpystyrn(ngx_linux_kern_osrelease, (u_char *) u.release,
47                       sizeof(ngx_linux_kern_osrelease));
48 }
```

```

49 #if (NGX_HAVE_RTSG)
50 {
51     int         name[2];
52     size_t      len;
53     ngx_err_t   err;
54
55     name[0] = CTL_KERN;
56     name[1] = KERN_RTSGMAX;
57     len = sizeof(ngx_linux_rtsig_max);
58
59     if (sysctl(name, 2, &ngx_linux_rtsig_max, &len, NULL, 0) == -1) {
60         err = ngx_errno;
61
62         if (err != NGX_ENOTDIR && err != NGX_ENOSYS) {
63             ngx_log_error(NGX_LOG_ALERT, log, err,
64                 "sysctl(KERN_RTSGMAX) failed");
65
66             return NGX_ERROR;
67         }
68
69         ngx_linux_rtsig_max = 0;
70     }
71 }
72 }
73 #endif
74
75 ngx_os_io = ngx_linux_io;
76
77 return NGX_OK;
78 }
79
80
81 void
82 ngx_os_specific_status(ngx_log_t *log)
83 {
84     ngx_log_error(NGX_LOG_NOTICE, log, 0, "OS: %s %s",
85         ngx_linux_kern_ostype, ngx_linux_kern_osrelease);
86
87     #if (NGX_HAVE_RTSG)
88         ngx_log_error(NGX_LOG_NOTICE, log, 0, "sysctl(KERN_RTSGMAX): %d",
89             ngx_linux_rtsig_max);
90     #endif
91 }

```

[One Level Up](#)

[Top Level](#)

## src/os/unix/nginx\_linux\_sendfile\_chain.c - nginx-1.7.10

### Functions defined

- [ngx\\_linux\\_sendfile\\_chain](#)

### Macros defined

- [NGX\\_SENDFILE\\_MAXSIZE](#)

### Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13  /*
14  * On Linux up to 2.4.21 sendfile() (syscall #187) works with 32-bit
15  * offsets only, and the including <sys/sendfile.h> breaks the compiling,
16  * if off_t is 64 bit wide. So we use own sendfile() definition, where offset
17  * parameter is int32_t, and use sendfile() for the file parts below 2G only,
18  * see src/os/unix/nginx_linux_config.h
19  *
20  * Linux 2.4.21 has the new sendfile64() syscall #239.
21  *
22  * On Linux up to 2.6.16 sendfile() does not allow to pass the count parameter
23  * more than 2G-1 bytes even on 64-bit platforms: it returns EINVAL,
24  * so we limit it to 2G-1 bytes.
25  */
26
27 #define NGX_SENDFILE_MAXSIZE  2147483647L
28
29
30 ngx_chain_t *
31 ngx_linux_sendfile_chain(ngx_connection_t *c, ngx_chain_t *in, off_t limit)
32 {
33     int                tcp_nodelay;
34     off_t              send, prev_send, sent;
35     size_t             file_size;
36     ssize_t           n;
37     ngx_err_t         err;
38     ngx_buf_t         *file;
39     ngx_uint_t        eintr;
40     ngx_event_t       *wev;
41     ngx_chain_t       *cl;
42     ngx_iovec_t       header;
43     struct iovec      headers[NGX_IOVS_PREALLOCATE];
44 #if (NGX_HAVE_SENDFILE64)
45     off_t              offset;
46 #else
47     int32_t            offset;
48 #endif
49
50     wev = c->write;
51
52     if (!wev->ready) {
53         return in;
54     }
55
56
57     /* the maximum limit size is 2G-1 - the page size */
```

```

58
59 if (limit == 0 || limit > (off_t) (NGX_SENDFILE_MAXSIZE - ngx_pagesize)) {
60     limit = NGX_SENDFILE_MAXSIZE - ngx_pagesize;
61 }
62
63
64 send = 0;
65
66 header.iovs = headers;
67 header.nalloc = NGX_IOVS_PREALLOCATE;
68
69 for ( ;; ) {
70     eintr = 0;
71     prev_send = send;
72
73     /* create the iovec and coalesce the neighbouring bufs */
74
75     cl = ngx_output_chain_to_iovec(&header, in, limit - send, c->log);
76
77     if (cl == NGX_CHAIN_ERROR) {
78         return NGX_CHAIN_ERROR;
79     }
80
81     send += header.size;
82
83     /* set TCP_CORK if there is a header before a file */
84
85     if (c->tcp_nopush == NGX_TCP_NOPUSH_UNSET
86         && header.count != 0
87         && cl
88         && cl->buf->in_file)
89     {
90         /* the TCP_CORK and TCP_NODELAY are mutually exclusive */
91
92         if (c->tcp_nodelay == NGX_TCP_NODELAY_SET) {
93
94             tcp_nodelay = 0;
95
96             if (setsockopt(c->fd, IPPROTO_TCP, TCP_NODELAY,
97                 (const void *) &tcp_nodelay, sizeof(int)) == -1)
98             {
99                 err = ngx_socket_errno;
100
101                 /*
102                 * there is a tiny chance to be interrupted, however,
103                 * we continue a processing with the TCP_NODELAY
104                 * and without the TCP_CORK
105                 */
106
107                 if (err != NGX_EINTR) {
108                     wev->error = 1;
109                     ngx_connection_error(c, err,
110                         "setsockopt(TCP_NODELAY) failed");
111                     return NGX_CHAIN_ERROR;
112                 }
113
114             } else {
115                 c->tcp_nodelay = NGX_TCP_NODELAY_UNSET;
116
117                 ngx_log_debug0(NGX_LOG_DEBUG_EVENT, c->log, 0,
118                     "no tcp_nodelay");
119             }
120         }
121
122         if (c->tcp_nodelay == NGX_TCP_NODELAY_UNSET) {
123
124             if (ngx_tcp_nopush(c->fd) == NGX_ERROR) {
125                 err = ngx_socket_errno;
126
127                 /*
128                 * there is a tiny chance to be interrupted, however,
129                 * we continue a processing without the TCP_CORK
130                 */
131
132                 if (err != NGX_EINTR) {
133                     wev->error = 1;

```

```

134         ngx_connection_error(c, err,
135                             ngx_tcp_nopush_n " failed");
136         return NGX_CHAIN_ERROR;
137     }
138
139     } else {
140         c->tcp_nopush = NGX_TCP_NOPUSH_SET;
141
142         ngx_log_debug0(NGX_LOG_DEBUG_EVENT, c->log, 0,
143                       "tcp_nopush");
144     }
145 }
146 }
147
148 /* get the file buf */
149
150 if (header.count == 0 && cl && cl->buf->in_file && send < limit) {
151     file = cl->buf;
152
153     /* coalesce the neighbouring file bufs */
154
155     file_size = (size_t) ngx_chain_coalesce_file(&cl, limit - send);
156
157     send += file_size;
158
159     #if 1
160     if (file_size == 0) {
161         ngx_debug_point();
162         return NGX_CHAIN_ERROR;
163     }
164     #endif
165     #if (NGX_HAVE_SENDFILE64)
166     offset = file->file_pos;
167     #else
168     offset = (int32_t) file->file_pos;
169     #endif
170
171     ngx_log_debug2(NGX_LOG_DEBUG_EVENT, c->log, 0,
172                  "sendfile: @%O %uz", file->file_pos, file_size);
173
174     n = sendfile(c->fd, file->file->fd, &offset, file_size);
175
176     if (n == -1) {
177         err = ngx_errno;
178
179         switch (err) {
180             case NGX_EAGAIN:
181                 break;
182
183             case NGX_EINTR:
184                 eintr = 1;
185                 break;
186
187             default:
188                 wev->error = 1;
189                 ngx_connection_error(c, err, "sendfile() failed");
190                 return NGX_CHAIN_ERROR;
191         }
192
193         ngx_log_debug0(NGX_LOG_DEBUG_EVENT, c->log, err,
194                       "sendfile() is not ready");
195     }
196
197     sent = n > 0 ? n : 0;
198
199     ngx_log_debug4(NGX_LOG_DEBUG_EVENT, c->log, 0,
200                  "sendfile: %z, @%O %O:%uz",
201                  n, file->file_pos, sent, file_size);
202 } else {
203     n = ngx_writev(c, &header);
204
205     if (n == NGX_ERROR) {
206         return NGX_CHAIN_ERROR;
207     }
208
209     sent = (n == NGX_AGAIN) ? 0 : n;

```

```
210     }
211
212     c->sent += sent;
213
214     in = ngx\_chain\_update\_sent(in, sent);
215
216     if (eintr) {
217         send = prev_send;
218         continue;
219     }
220
221     if (send - prev_send != sent) {
222         wev->ready = 0;
223         return in;
224     }
225
226     if (send >= limit || in == NULL) {
227         return in;
228     }
229 }
230 }
```

[One Level Up](#)

[Top Level](#)

## src/event/modules/nginx\_select\_module.c - nginx-1.7.10

### Global variables defined

- [event\\_index](#)
- [master\\_read\\_fd\\_set](#)
- [master\\_write\\_fd\\_set](#)
- [max\\_fd](#)
- [nevents](#)
- [ngx\\_select\\_module](#)
- [ngx\\_select\\_module\\_ctx](#)
- [select\\_name](#)
- [work\\_read\\_fd\\_set](#)
- [work\\_write\\_fd\\_set](#)

### Functions defined

- [ngx\\_select\\_add\\_event](#)
- [ngx\\_select\\_del\\_event](#)
- [ngx\\_select\\_done](#)
- [ngx\\_select\\_init](#)
- [ngx\\_select\\_init\\_conf](#)
- [ngx\\_select\\_process\\_events](#)
- [ngx\\_select\\_repair\\_fd\\_sets](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 static ngx_int_t ngx_select_init(ngx_cycle_t *cycle, ngx_msec_t timer);
14 static void ngx_select_done(ngx_cycle_t *cycle);
15 static ngx_int_t ngx_select_add_event(ngx_event_t *ev, ngx_int_t event,
16     ngx_uint_t flags);
17 static ngx_int_t ngx_select_del_event(ngx_event_t *ev, ngx_int_t event,
18     ngx_uint_t flags);
19 static ngx_int_t ngx_select_process_events(ngx_cycle_t *cycle, ngx_msec_t timer,
20     ngx_uint_t flags);
21 static void ngx_select_repair_fd_sets(ngx_cycle_t *cycle);
22 static char *ngx_select_init_conf(ngx_cycle_t *cycle, void *conf);
23
```



```

24
25 static fd_set      master_read_fd_set;
26 static fd_set      master_write_fd_set;
27 static fd_set      work_read_fd_set;
28 static fd_set      work_write_fd_set;
29
30 static ngx_int_t    max_fd;
31 static ngx_uint_t   nevents;
32
33 static ngx_event_t  **event_index;
34
35
36 static ngx_str_t    select_name = ngx_string("select");
37
38 ngx_event_module_t ngx_select_module_ctx = {
39     &select_name,
40     NULL,                /* create configuration */
41     ngx_select_init_conf, /* init configuration */
42
43     {
44         ngx_select_add_event, /* add an event */
45         ngx_select_del_event, /* delete an event */
46         ngx_select_add_event, /* enable an event */
47         ngx_select_del_event, /* disable an event */
48         NULL,                /* add an connection */
49         NULL,                /* delete an connection */
50         NULL,                /* process the changes */
51         ngx_select_process_events, /* process the events */
52         ngx_select_init,     /* init the events */
53         ngx_select_done      /* done the events */
54     }
55 };
56
57
58 ngx_module_t ngx_select_module = {
59     NGX_MODULE_V1,
60     &ngx_select_module_ctx, /* module context */
61     NULL,                  /* module directives */
62     NGX_EVENT_MODULE,     /* module type */
63     NULL,                 /* init master */
64     NULL,                 /* init module */
65     NULL,                 /* init process */
66     NULL,                 /* init thread */
67     NULL,                 /* exit thread */
68     NULL,                 /* exit process */
69     NULL,                 /* exit master */
70     NGX_MODULE_V1_PADDING
71 };
72
73
74 static ngx_int_t
75 ngx_select_init(ngx_cycle_t *cycle, ngx_msec_t timer)
76 {
77     ngx_event_t  **index;
78
79     if (event_index == NULL) {
80         FD_ZERO(&master_read_fd_set);
81         FD_ZERO(&master_write_fd_set);
82         nevents = 0;
83     }
84
85     if (ngx_process >= NGX_PROCESS_WORKER
86         || cycle->old_cycle == NULL
87         || cycle->old_cycle->connection_n < cycle->connection_n)
88     {
89         index = ngx_alloc(sizeof(ngx_event_t *) * 2 * cycle->connection_n,
90                           cycle->log);
91         if (index == NULL) {
92             return NGX_ERROR;
93         }
94
95         if (event_index) {
96             ngx_memcpy(index, event_index, sizeof(ngx_event_t *) * nevents);
97             ngx_free(event_index);
98         }
99

```

```

100     event_index = index;
101 }
102
103 ngx_io = ngx_os_io;
104
105 ngx_event_actions = ngx_select_module_ctx.actions;
106
107 ngx_event_flags = NGX_USE_LEVEL_EVENT;
108
109 max_fd = -1;
110
111 return NGX_OK;
112 }
113
114
115 static void
116 ngx_select_done(ngx_cycle_t *cycle)
117 {
118     ngx_free(event_index);
119
120     event_index = NULL;
121 }
122
123
124 static ngx_int_t
125 ngx_select_add_event(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags)
126 {
127     ngx_connection_t *c;
128
129     c = ev->data;
130
131     ngx_log_debug2(NGX_LOG_DEBUG_EVENT, ev->log, 0,
132                    "select add event fd:%d ev:%i", c->fd, event);
133
134     if (ev->index != NGX_INVALID_INDEX) {
135         ngx_log_error(NGX_LOG_ALERT, ev->log, 0,
136                       "select event fd:%d ev:%i is already set", c->fd, event);
137         return NGX_OK;
138     }
139
140     if ((event == NGX_READ_EVENT && ev->write)
141         || (event == NGX_WRITE_EVENT && !ev->write))
142     {
143         ngx_log_error(NGX_LOG_ALERT, ev->log, 0,
144                       "invalid select %s event fd:%d ev:%i",
145                       ev->write ? "write" : "read", c->fd, event);
146         return NGX_ERROR;
147     }
148
149     if (event == NGX_READ_EVENT) {
150         FD_SET(c->fd, &master_read_fd_set);
151     }
152     else if (event == NGX_WRITE_EVENT) {
153         FD_SET(c->fd, &master_write_fd_set);
154     }
155
156     if (max_fd != -1 && max_fd < c->fd) {
157         max_fd = c->fd;
158     }
159
160     ev->active = 1;
161
162     event_index[nevents] = ev;
163     ev->index = nevents;
164     nevents++;
165
166     return NGX_OK;
167 }
168
169
170 static ngx_int_t
171 ngx_select_del_event(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags)
172 {
173     ngx_event_t *e;
174     ngx_connection_t *c;
175

```

```

176     c = ev->data;
177
178     ev->active = 0;
179
180     if (ev->index == NGX\_INVALID\_INDEX) {
181         return NGX\_OK;
182     }
183
184     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_EVENT, ev->log, 0,
185         "select del event fd:%d ev:%i", c->fd, event);
186
187     if (event == NGX\_READ\_EVENT) {
188         FD\_CLR(c->fd, &master\_read\_fd\_set);
189
190     } else if (event == NGX\_WRITE\_EVENT) {
191         FD\_CLR(c->fd, &master\_write\_fd\_set);
192     }
193
194     if (max\_fd == c->fd) {
195         max\_fd = -1;
196     }
197
198     if (ev->index < --nevents) {
199         e = event\_index[nevents];
200         event\_index[ev->index] = e;
201         e->index = ev->index;
202     }
203
204     ev->index = NGX\_INVALID\_INDEX;
205
206     return NGX\_OK;
207 }
208
209
210 static ngx\_int\_t
211 ngx\_select\_process\_events(ngx\_cycle\_t *cycle, ngx\_msec\_t timer,
212     ngx\_uint\_t flags)
213 {
214     int                ready, nready;
215     ngx\_err\_t         err;
216     ngx\_uint\_t        i, found;
217     ngx\_event\_t       *ev;
218     ngx\_queue\_t       *queue;
219     struct timeval     tv, *tp;
220     ngx\_connection\_t *c;
221
222     if (max\_fd == -1) {
223         for (i = 0; i < nevents; i++) {
224             c = event\_index[i]->data;
225             if (max\_fd < c->fd) {
226                 max\_fd = c->fd;
227             }
228         }
229
230         ngx\_log\_debug1(NGX\_LOG\_DEBUG\_EVENT, cycle->log, 0,
231             "change max\_fd: %i", max\_fd);
232     }
233
234     #if (NGX\_DEBUG)
235     if (cycle->log->log_level & NGX\_LOG\_DEBUG\_ALL) {
236         for (i = 0; i < nevents; i++) {
237             ev = event\_index[i];
238             c = ev->data;
239             ngx\_log\_debug2(NGX\_LOG\_DEBUG\_EVENT, cycle->log, 0,
240                 "select event: fd:%d wr:%d", c->fd, ev->write);
241         }
242
243         ngx\_log\_debug1(NGX\_LOG\_DEBUG\_EVENT, cycle->log, 0,
244             "max\_fd: %i", max\_fd);
245     }
246     #endif
247
248     if (timer == NGX\_TIMER\_INFINITE) {
249         tp = NULL;
250
251     } else {

```

```

252     tv.tv_sec = (long) (timer / 1000);
253     tv.tv_usec = (long) ((timer % 1000) * 1000);
254     tp = &tv;
255 }
256
257 ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
258     "select timer: %M", timer);
259
260 work_read_fd_set = master_read_fd_set;
261 work_write_fd_set = master_write_fd_set;
262
263 ready = select(max_fd + 1, &work_read_fd_set, &work_write_fd_set, NULL, tp);
264
265 err = (ready == -1) ? ngx_errno : 0;
266
267 if (flags & NGX_UPDATE_TIME || ngx_event_timer_alarm) {
268     ngx_time_update();
269 }
270
271 ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
272     "select ready %d", ready);
273
274 if (err) {
275     ngx_uint_t level;
276
277     if (err == NGX_EINTR) {
278
279         if (ngx_event_timer_alarm) {
280             ngx_event_timer_alarm = 0;
281             return NGX_OK;
282         }
283
284         level = NGX_LOG_INFO;
285
286     } else {
287         level = NGX_LOG_ALERT;
288     }
289
290     ngx_log_error(level, cycle->log, err, "select() failed");
291
292     if (err == NGX_EBADF) {
293         ngx_select_repair_fd_sets(cycle);
294     }
295
296     return NGX_ERROR;
297 }
298
299 if (ready == 0) {
300     if (timer != NGX_TIMER_INFINITE) {
301         return NGX_OK;
302     }
303
304     ngx_log_error(NGX_LOG_ALERT, cycle->log, 0,
305         "select() returned no events without timeout");
306     return NGX_ERROR;
307 }
308
309 nready = 0;
310
311 for (i = 0; i < nevents; i++) {
312     ev = event_index[i];
313     c = ev->data;
314     found = 0;
315
316     if (ev->write) {
317         if (FD_ISSET(c->fd, &work_write_fd_set)) {
318             found = 1;
319             ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
320                 "select write %d", c->fd);
321         }
322     } else {
323         if (FD_ISSET(c->fd, &work_read_fd_set)) {
324             found = 1;
325             ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
326                 "select read %d", c->fd);

```

```

328     }
329 }
330
331 if (found) {
332     ev->ready = 1;
333
334     queue = ev->accept ? &ngx_posted_accept_events
335                       : &ngx_posted_events;
336
337     ngx_post_event(ev, queue);
338
339     nready++;
340 }
341 }
342
343 if (ready != nready) {
344     ngx_log_error(NGX_LOG_ALERT, cycle->log, 0,
345                 "select ready != events: %d:%d", ready, nready);
346
347     ngx_select_repair_fd_sets(cycle);
348 }
349
350 return NGX_OK;
351 }
352
353
354 static void
355 ngx_select_repair_fd_sets(ngx_cycle_t *cycle)
356 {
357     int         n;
358     socklen_t   len;
359     ngx_err_t   err;
360     ngx_socket_t s;
361
362     for (s = 0; s <= max_fd; s++) {
363
364         if (FD_ISSET(s, &master_read_fd_set) == 0) {
365             continue;
366         }
367
368         len = sizeof(int);
369
370         if (getsockopt(s, SOL_SOCKET, SO_TYPE, &n, &len) == -1) {
371             err = ngx_socket_errno;
372
373             ngx_log_error(NGX_LOG_ALERT, cycle->log, err,
374                         "invalid descriptor #%d in read fd_set", s);
375
376             FD_CLR(s, &master_read_fd_set);
377         }
378     }
379
380     for (s = 0; s <= max_fd; s++) {
381
382         if (FD_ISSET(s, &master_write_fd_set) == 0) {
383             continue;
384         }
385
386         len = sizeof(int);
387
388         if (getsockopt(s, SOL_SOCKET, SO_TYPE, &n, &len) == -1) {
389             err = ngx_socket_errno;
390
391             ngx_log_error(NGX_LOG_ALERT, cycle->log, err,
392                         "invalid descriptor #%d in write fd_set", s);
393
394             FD_CLR(s, &master_write_fd_set);
395         }
396     }
397
398     max_fd = -1;
399 }
400
401
402 static char *
403 ngx_select_init_conf(ngx_cycle_t *cycle, void *conf)

```

```
404 {
405     ngx\_event\_conf\_t *ecf;
406
407     ecf = ngx\_event\_get\_conf(cycle->conf_ctx, ngx\_event\_core\_module);
408
409     if (ecf->use != ngx\_select\_module.ctx_index) {
410         return NGX\_CONF\_OK;
411     }
412
413     /* disable warning: the default FD_SETSIZE is 1024U in FreeBSD 5.x */
414
415     if (cycle->connection_n > FD_SETSIZE) {
416         ngx\_log\_error(NGX\_LOG\_EMERG, cycle->log, 0,
417             "the maximum number of files "
418             "supported by select() is %ud", FD_SETSIZE);
419         return NGX\_CONF\_ERROR;
420     }
421
422     #if (NGX\_THREADS)
423
424         ngx\_log\_error(NGX\_LOG\_EMERG, cycle->log, 0,
425             "select() is not supported in the threaded mode");
426         return NGX\_CONF\_ERROR;
427
428     #else
429
430         return NGX\_CONF\_OK;
431
432     #endif
433 }
```

[One Level Up](#)

[Top Level](#)

## src/event/nginx\_event\_mutex.c - nginx-1.7.10

### Functions defined

- [ngx\\_event\\_mutex\\_timedlock](#)
- [ngx\\_event\\_mutex\\_unlock](#)

### Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 ngx_int_t ngx_event_mutex_timedlock(ngx_event_mutex_t *m, ngx_msec_t timer,
14                                     ngx_event_t *ev)
15 {
16     ngx_log_debug2(NGX_LOG_DEBUG_EVENT, ev->log, 0,
17                  "lock event mutex %p lock:%XD", m, m->lock);
18
19     if (m->lock) {
20
21         if (m->events == NULL) {
22             m->events = ev;
23
24         } else {
25             m->last->next = ev;
26         }
27
28         m->last = ev;
29         ev->next = NULL;
30
31 #if (NGX_THREADS0)
32     ev->light = 1;
33 #endif
34
35     ngx_add_timer(ev, timer);
36
37     return NGX_AGAIN;
38 }
39
40 m->lock = 1;
41
42 return NGX_OK;
43 }
44
45
46 ngx_int_t ngx_event_mutex_unlock(ngx_event_mutex_t *m, ngx_log_t *log)
47 {
48     ngx_event_t *ev;
49
50     if (m->lock == 0) {
51         ngx_log_error(NGX_LOG_ALERT, log, 0,
52                     "tring to unlock the free event mutex %p", m);
53         return NGX_ERROR;
54     }
55
56     ngx_log_debug2(NGX_LOG_DEBUG_EVENT, log, 0,
57                  "unlock event mutex %p, next event: %p", m, m->events);
58
59     m->lock = 0;
60 }
```

```
61     if (m->events) {
62         ev = m->events;
63         m->events = ev->next;
64
65         ev->next = ngx\_posted\_events;
66         ngx\_posted\_events = ev;
67     }
68
69     return NGX\_OK;
70 }
```

[One Level Up](#)

[Top Level](#)



## src/event/modules/nginx\_aio\_module.c - nginx-1.7.10

### Global variables defined

- [aio\\_name](#)
- [ngx\\_aio\\_module](#)
- [ngx\\_aio\\_module\\_ctx](#)
- [ngx\\_os\\_aio](#)

### Functions defined

- [ngx\\_aio\\_add\\_event](#)
- [ngx\\_aio\\_del\\_connection](#)
- [ngx\\_aio\\_del\\_event](#)
- [ngx\\_aio\\_done](#)
- [ngx\\_aio\\_init](#)
- [ngx\\_aio\\_process\\_events](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 extern ngx_event_module_t ngx_kqueue_module_ctx;
14
15
16 static ngx_int_t ngx_aio_init(ngx_cycle_t *cycle, ngx_msec_t timer);
17 static void ngx_aio_done(ngx_cycle_t *cycle);
18 static ngx_int_t ngx_aio_add_event(ngx_event_t *ev, ngx_int_t event,
19     ngx_uint_t flags);
20 static ngx_int_t ngx_aio_del_event(ngx_event_t *ev, ngx_int_t event,
21     ngx_uint_t flags);
22 static ngx_int_t ngx_aio_del_connection(ngx_connection_t *c, ngx_uint_t flags);
23 static ngx_int_t ngx_aio_process_events(ngx_cycle_t *cycle, ngx_msec_t timer,
24     ngx_uint_t flags);
25
26
27 ngx_os_io_t ngx_os_aio = {
28     ngx_aio_read,
29     ngx_aio_read_chain,
30     NULL,
31     ngx_aio_write,
32     ngx_aio_write_chain,
33     0
34 };
35
36
37 static ngx_str_t aio_name = ngx_string("aio");
38
```

```

39 ngx_event_module_t ngx_aio_module_ctx = {
40     &uaio\_name,
41     NULL, /* create configuration */
42     NULL, /* init configuration */
43
44     {
45         ngx_aio_add_event, /* add an event */
46         ngx_aio_del_event, /* delete an event */
47         NULL, /* enable an event */
48         NULL, /* disable an event */
49         NULL, /* add an connection */
50         ngx_aio_del_connection, /* delete an connection */
51         NULL, /* process the changes */
52         ngx_aio_process_events, /* process the events */
53         ngx_aio_init, /* init the events */
54         ngx_aio_done /* done the events */
55     }
56 };
57
58
59 ngx_module_t ngx_aio_module = {
60     NGX_MODULE_V1,
61     &ngx_aio_module_ctx, /* module context */
62     NULL, /* module directives */
63     NGX_EVENT_MODULE, /* module type */
64     NULL, /* init master */
65     NULL, /* init module */
66     NULL, /* init process */
67     NULL, /* init thread */
68     NULL, /* exit thread */
69     NULL, /* exit process */
70     NULL, /* exit master */
71     NGX_MODULE_V1_PADDING
72 };
73
74
75 #if (NGX_HAVE_KQUEUE)
76
77 static ngx_int_t
78 ngx_aio_init(ngx_cycle_t *cycle, ngx_msec_t timer)
79 {
80     if (ngx_kqueue_module_ctx.actions.init(cycle, timer) == NGX_ERROR) {
81         return NGX_ERROR;
82     }
83
84     ngx_io = ngx_os_aio;
85
86     ngx_event_flags = NGX_USE_AIO_EVENT;
87     ngx_event_actions = ngx_aio_module_ctx.actions;
88
89     return NGX_OK;
90 }
91
92
93
94 static void
95 ngx_aio_done(ngx_cycle_t *cycle)
96 {
97     ngx_kqueue_module_ctx.actions.done(cycle);
98 }
99
100
101 /* the event adding and deleting are needed for the listening sockets */
102
103 static ngx_int_t
104 ngx_aio_add_event(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags)
105 {
106     return ngx_kqueue_module_ctx.actions.add(ev, event, flags);
107 }
108
109
110 static ngx_int_t
111 ngx_aio_del_event(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags)
112 {
113     return ngx_kqueue_module_ctx.actions.del(ev, event, flags);
114 }

```

```

115
116
117 static ngx_int_t
118 ngx_aio_del_connection(ngx_connection_t *c, ngx_uint_t flags)
119 {
120     int rc;
121
122     if (c->read->active == 0 && c->write->active == 0) {
123         return NGX_OK;
124     }
125
126     if (flags & NGX_CLOSE_EVENT) {
127         return NGX_OK;
128     }
129
130     rc = aio_cancel(c->fd, NULL);
131
132     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, c->log, 0, "aio_cancel: %d", rc);
133
134     if (rc == AIO_CANCELED) {
135         c->read->active = 0;
136         c->write->active = 0;
137         return NGX_OK;
138     }
139
140     if (rc == AIO_ALLDONE) {
141         c->read->active = 0;
142         c->write->active = 0;
143         ngx_log_error(NGX_LOG_ALERT, c->log, 0,
144             "aio_cancel() returned AIO_ALLDONE");
145         return NGX_OK;
146     }
147
148     if (rc == -1) {
149         ngx_log_error(NGX_LOG_ALERT, c->log, ngx_errno,
150             "aio_cancel() failed");
151         return NGX_ERROR;
152     }
153
154     if (rc == AIO_NOTCANCELED) {
155         ngx_log_error(NGX_LOG_ALERT, c->log, 0,
156             "aio_cancel() returned AIO_NOTCANCELED");
157
158         return NGX_ERROR;
159     }
160
161     return NGX_OK;
162 }
163
164
165 static ngx_int_t
166 ngx_aio_process_events(ngx_cycle_t *cycle, ngx_msec_t timer, ngx_uint_t flags)
167 {
168     return ngx_kqueue_module_ctx.actions.process_events(cycle, timer, flags);
169 }
170
171 #endif /* NGX_HAVE_KQUEUE */

```

[One Level Up](#)

[Top Level](#)

## src/event/modules/nginx\_win32\_select\_module.c - nginx-1.7.10

### Global variables defined

- [event\\_index](#)
- [master\\_read\\_fd\\_set](#)
- [master\\_write\\_fd\\_set](#)
- [max\\_read](#)
- [max\\_write](#)
- [nevents](#)
- [ngx\\_select\\_module](#)
- [ngx\\_select\\_module\\_ctx](#)
- [select\\_name](#)
- [work\\_read\\_fd\\_set](#)
- [work\\_write\\_fd\\_set](#)

### Functions defined

- [ngx\\_select\\_add\\_event](#)
- [ngx\\_select\\_del\\_event](#)
- [ngx\\_select\\_done](#)
- [ngx\\_select\\_init](#)
- [ngx\\_select\\_init\\_conf](#)
- [ngx\\_select\\_process\\_events](#)
- [ngx\\_select\\_repair\\_fd\\_sets](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 static ngx_int_t ngx_select_init(ngx_cycle_t *cycle, ngx_msec_t timer);
14 static void ngx_select_done(ngx_cycle_t *cycle);
15 static ngx_int_t ngx_select_add_event(ngx_event_t *ev, ngx_int_t event,
16     ngx_uint_t flags);
17 static ngx_int_t ngx_select_del_event(ngx_event_t *ev, ngx_int_t event,
18     ngx_uint_t flags);
19 static ngx_int_t ngx_select_process_events(ngx_cycle_t *cycle, ngx_msec_t timer,
20     ngx_uint_t flags);
```

```

21 static void ngx_select_repair_fd_sets(ngx_cycle_t *cycle);
22 static char *ngx_select_init_conf(ngx_cycle_t *cycle, void *conf);
23
24
25 static fd_set      master_read_fd_set;
26 static fd_set      master_write_fd_set;
27 static fd_set      work_read_fd_set;
28 static fd_set      work_write_fd_set;
29
30 static ngx_uint_t  max_read;
31 static ngx_uint_t  max_write;
32 static ngx_uint_t  nevents;
33
34 static ngx_event_t **event_index;
35
36
37 static ngx_str_t   select_name = ngx_string("select");
38
39 ngx_event_module_t ngx_select_module_ctx = {
40     &select_name,
41     NULL,                /* create configuration */
42     ngx_select_init_conf, /* init configuration */
43
44     {
45         ngx_select_add_event, /* add an event */
46         ngx_select_del_event, /* delete an event */
47         ngx_select_add_event, /* enable an event */
48         ngx_select_del_event, /* disable an event */
49         NULL,                /* add an connection */
50         NULL,                /* delete an connection */
51         NULL,                /* process the changes */
52         ngx_select_process_events, /* process the events */
53         ngx_select_init,     /* init the events */
54         ngx_select_done,     /* done the events */
55     }
56 };
57
58
59 ngx_module_t ngx_select_module = {
60     NGX_MODULE_V1,
61     &ngx_select_module_ctx, /* module context */
62     NULL,                  /* module directives */
63     NGX_EVENT_MODULE,     /* module type */
64     NULL,                  /* init master */
65     NULL,                  /* init module */
66     NULL,                  /* init process */
67     NULL,                  /* init thread */
68     NULL,                  /* exit thread */
69     NULL,                  /* exit process */
70     NULL,                  /* exit master */
71     NGX_MODULE_V1_PADDING
72 };
73
74
75 static ngx_int_t
76 ngx_select_init(ngx_cycle_t *cycle, ngx_msec_t timer)
77 {
78     ngx_event_t **index;
79
80     if (event_index == NULL) {
81         FD_ZERO(&master_read_fd_set);
82         FD_ZERO(&master_write_fd_set);
83         nevents = 0;
84     }
85
86     if (ngx_process >= NGX_PROCESS_WORKER
87         || cycle->old_cycle == NULL
88         || cycle->old_cycle->connection_n < cycle->connection_n)
89     {
90         index = ngx_alloc(sizeof(ngx_event_t *) * 2 * cycle->connection_n,
91                           cycle->log);
92         if (index == NULL) {
93             return NGX_ERROR;
94         }
95
96         if (event_index) {

```

```

97     ngx_memcpy(index, event_index, sizeof(ngx_event_t *) * nevents);
98     ngx_free(event_index);
99 }
100
101     event_index = index;
102 }
103
104 ngx_io = ngx_os_io;
105
106 ngx_event_actions = ngx_select_module_ctx.actions;
107
108 ngx_event_flags = NGX_USE_LEVEL_EVENT;
109
110 max_read = 0;
111 max_write = 0;
112
113 return NGX_OK;
114 }
115
116
117 static void
118 ngx_select_done(ngx_cycle_t *cycle)
119 {
120     ngx_free(event_index);
121
122     event_index = NULL;
123 }
124
125
126 static ngx_int_t
127 ngx_select_add_event(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags)
128 {
129     ngx_connection_t *c;
130
131     c = ev->data;
132
133     ngx_log_debug2(NGX_LOG_DEBUG_EVENT, ev->log, 0,
134                  "select add event fd:%d ev:%i", c->fd, event);
135
136     if (ev->index != NGX_INVALID_INDEX) {
137         ngx_log_error(NGX_LOG_ALERT, ev->log, 0,
138                      "select event fd:%d ev:%i is already set", c->fd, event);
139         return NGX_OK;
140     }
141
142     if ((event == NGX_READ_EVENT && ev->write)
143         || (event == NGX_WRITE_EVENT && !ev->write))
144     {
145         ngx_log_error(NGX_LOG_ALERT, ev->log, 0,
146                      "invalid select %s event fd:%d ev:%i",
147                      ev->write ? "write" : "read", c->fd, event);
148         return NGX_ERROR;
149     }
150
151     if ((event == NGX_READ_EVENT && max_read >= FD_SETSIZE)
152         || (event == NGX_WRITE_EVENT && max_write >= FD_SETSIZE))
153     {
154         ngx_log_error(NGX_LOG_ERR, ev->log, 0,
155                      "maximum number of descriptors "
156                      "supported by select() is %d", FD_SETSIZE);
157         return NGX_ERROR;
158     }
159
160     if (event == NGX_READ_EVENT) {
161         FD_SET(c->fd, &master_read_fd_set);
162         max_read++;
163     }
164     else if (event == NGX_WRITE_EVENT) {
165         FD_SET(c->fd, &master_write_fd_set);
166         max_write++;
167     }
168
169     ev->active = 1;
170
171     event_index[nevents] = ev;
172     ev->index = nevents;

```

```

173     nevents++;
174
175     return NGX\_OK;
176 }
177
178
179 static ngx\_int\_t
180 ngx\_select\_del\_event(ngx\_event\_t *ev, ngx\_int\_t event, ngx\_uint\_t flags)
181 {
182     ngx\_event\_t     *e;
183     ngx\_connection\_t *c;
184
185     c = ev->data;
186
187     ev->active = 0;
188
189     if (ev->index == NGX\_INVALID\_INDEX) {
190         return NGX\_OK;
191     }
192
193     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_EVENT, ev->log, 0,
194                 "select del event fd:%d ev:%i", c->fd, event);
195
196     if (event == NGX\_READ\_EVENT) {
197         FD_CLR(c->fd, &master\_read\_fd\_set);
198         max\_read--;
199     }
200     else if (event == NGX\_WRITE\_EVENT) {
201         FD_CLR(c->fd, &master\_write\_fd\_set);
202         max\_write--;
203     }
204
205     if (ev->index < --nevents) {
206         e = event\_index[nevents];
207         event\_index[ev->index] = e;
208         e->index = ev->index;
209     }
210
211     ev->index = NGX\_INVALID\_INDEX;
212
213     return NGX\_OK;
214 }
215
216
217 static ngx\_int\_t
218 ngx\_select\_process\_events(ngx\_cycle\_t *cycle, ngx\_msec\_t timer,
219 ngx\_uint\_t flags)
220 {
221     int             ready, nready;
222     ngx\_err\_t      err;
223     ngx\_uint\_t     i, found;
224     ngx\_event\_t   *ev;
225     ngx\_queue\_t   *queue;
226     struct timeval tv, *tp;
227     ngx\_connection\_t *c;
228
229     #if (NGX\_DEBUG)
230     if (cycle->log->log_level & NGX\_LOG\_DEBUG\_ALL) {
231         for (i = 0; i < nevents; i++) {
232             ev = event\_index[i];
233             c = ev->data;
234             ngx\_log\_debug2(NGX\_LOG\_DEBUG\_EVENT, cycle->log, 0,
235                 "select event: fd:%d wr:%d", c->fd, ev->write);
236         }
237     }
238     #endif
239
240     if (timer == NGX\_TIMER\_INFINITE) {
241         tp = NULL;
242     }
243     else {
244         tv.tv_sec = (long) (timer / 1000);
245         tv.tv_usec = (long) ((timer % 1000) * 1000);
246         tp = &tv;
247     }
248

```

```

249 ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
250     "select timer: %M", timer);
251
252 work_read_fd_set = master_read_fd_set;
253 work_write_fd_set = master_write_fd_set;
254
255 if (max_read || max_write) {
256     ready = select(0, &work_read_fd_set, &work_write_fd_set, NULL, tp);
257
258 } else {
259
260     /*
261     * Winsock select() requires that at least one descriptor set must be
262     * be non-null, and any non-null descriptor set must contain at least
263     * one handle to a socket. Otherwise select() returns WSAEINVAL.
264     */
265
266     ngx_msleep(timer);
267
268     ready = 0;
269 }
270
271 err = (ready == -1) ? ngx_socket_errno : 0;
272
273 if (flags & NGX_UPDATE_TIME) {
274     ngx_time_update();
275 }
276
277 ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
278     "select ready %d", ready);
279
280 if (err) {
281     ngx_log_error(NGX_LOG_ALERT, cycle->log, err, "select() failed");
282
283     if (err == WSAENOTSOCK) {
284         ngx_select_repair_fd_sets(cycle);
285     }
286
287     return NGX_ERROR;
288 }
289
290 if (ready == 0) {
291     if (timer != NGX_TIMER_INFINITE) {
292         return NGX_OK;
293     }
294
295     ngx_log_error(NGX_LOG_ALERT, cycle->log, 0,
296         "select() returned no events without timeout");
297     return NGX_ERROR;
298 }
299
300 nready = 0;
301
302 for (i = 0; i < nevents; i++) {
303     ev = event_index[i];
304     c = ev->data;
305     found = 0;
306
307     if (ev->write) {
308         if (FD_ISSET(c->fd, &work_write_fd_set)) {
309             found = 1;
310             ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
311                 "select write %d", c->fd);
312         }
313
314     } else {
315         if (FD_ISSET(c->fd, &work_read_fd_set)) {
316             found = 1;
317             ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
318                 "select read %d", c->fd);
319         }
320     }
321
322     if (found) {
323         ev->ready = 1;
324

```



```

325         queue = ev->accept ? &ngx_posted_accept_events
326                             : &ngx_posted_events;
327
328         ngx_post_event(ev, queue);
329
330         nready++;
331     }
332 }
333
334 if (ready != nready) {
335     ngx_log_error(NGX_LOG_ALERT, cycle->log, 0,
336                 "select ready != events: %d:%d", ready, nready);
337
338     ngx_select_repair_fd_sets(cycle);
339 }
340
341 return NGX_OK;
342 }
343
344
345 static void
346 ngx_select_repair_fd_sets(ngx_cycle_t *cycle)
347 {
348     int             n;
349     u_int           i;
350     socklen_t       len;
351     ngx_err_t       err;
352     ngx_socket_t    s;
353
354     for (i = 0; i < master_read_fd_set.fd_count; i++) {
355
356         s = master_read_fd_set.fd_array[i];
357         len = sizeof(int);
358
359         if (getsockopt(s, SOL_SOCKET, SO_TYPE, (char *) &n, &len) == -1) {
360             err = ngx_socket_errno;
361
362             ngx_log_error(NGX_LOG_ALERT, cycle->log, err,
363                         "invalid descriptor #%d in read fd_set", s);
364
365             FD_CLR(s, &master_read_fd_set);
366         }
367     }
368
369     for (i = 0; i < master_write_fd_set.fd_count; i++) {
370
371         s = master_write_fd_set.fd_array[i];
372         len = sizeof(int);
373
374         if (getsockopt(s, SOL_SOCKET, SO_TYPE, (char *) &n, &len) == -1) {
375             err = ngx_socket_errno;
376
377             ngx_log_error(NGX_LOG_ALERT, cycle->log, err,
378                         "invalid descriptor #%d in write fd_set", s);
379
380             FD_CLR(s, &master_write_fd_set);
381         }
382     }
383 }
384
385
386 static char *
387 ngx_select_init_conf(ngx_cycle_t *cycle, void *conf)
388 {
389     ngx_event_conf_t *ecf;
390
391     ecf = ngx_event_get_conf(cycle->conf_ctx, ngx_event_core_module);
392
393     if (ecf->use != ngx_select_module.ctx_index) {
394         return NGX_CONF_OK;
395     }
396
397     return NGX_CONF_OK;
398 }

```

## src/os/unix/nginx\_darwin.h - nginx-1.7.10

### Macros defined

- [\\_NGX\\_DARWIN\\_H\\_INCLUDED](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_DARWIN\_H\_INCLUDED
9 #define \_NGX\_DARWIN\_H\_INCLUDED
10
11
12 void ngx\_debug\_init(void);
13 ngx\_chain\_t *ngx\_darwin\_sendfile\_chain(ngx\_connection\_t *c, ngx\_chain\_t *in,
14     off_t limit);
15
16 extern int      ngx\_darwin\_kern\_osreldate;
17 extern int      ngx\_darwin\_hw\_ncpu;
18 extern u_long   ngx\_darwin\_net\_inet\_tcp\_sendspace;
19
20 extern ngx\_uint\_t ngx\_debug\_malloc;
21
22
23 #endif /* \_NGX\_DARWIN\_H\_INCLUDED */
```

# src/os/unix/nginx\_darwin\_sendfile\_chain.c - nginx-1.7.10

## Functions defined

- [ngx\\_darwin\\_sendfile\\_chain](#)

## Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_event.h>
11
12
13 /*
14 * It seems that Darwin 9.4 (Mac OS X 1.5) sendfile() has the same
15 * old bug as early FreeBSD sendfile() syscall:
16 * http://bugs.freebsd.org/33771
17 *
18 * Besides sendfile() has another bug: if one calls sendfile()
19 * with both a header and a trailer, then sendfile() ignores a file part
20 * at all and sends only the header and the trailer together.
21 * For this reason we send a trailer only if there is no a header.
22 *
23 * Although sendfile() allows to pass a header or a trailer,
24 * it may send the header or the trailer and a part of the file
25 * in different packets. And FreeBSD workaround (TCP_NOPUSH option)
26 * does not help.
27 */
28
29
30 ngx_chain_t *
31 ngx_darwin_sendfile_chain(ngx_connection_t *c, ngx_chain_t *in, off_t limit)
32 {
33     int rc;
34     off_t send, prev_send, sent;
35     off_t file_size;
36     ssize_t n;
37     ngx_uint_t eintr;
38     ngx_err_t err;
39     ngx_buf_t *file;
40     ngx_event_t *wev;
41     ngx_chain_t *cl;
42     ngx_iovec_t header, trailer;
43     struct sf_hdrt hdtr;
44     struct iovec headers[NGX_IOVS_PREALLOCATE];
45     struct iovec trailers[NGX_IOVS_PREALLOCATE];
46
47     wev = c->write;
48
49     if (!wev->ready) {
50         return in;
51     }
52
53     #if (NGX_HAVE_KQUEUE)
54
55     if ((ngx_event_flags & NGX_USE_KQUEUE_EVENT) && wev->pending_eof) {
56         (void) ngx_connection_error(c, wev->kq_errno,
57             "kevent() reported about an closed connection");
58         wev->error = 1;
59         return NGX_CHAIN_ERROR;
60     }
61
62     #endif
```

```

63
64  /* the maximum limit size is the maximum size_t value - the page size */
65
66  if (limit == 0 || limit > (off_t) (NGX_MAX_SIZE_T_VALUE - ngx_pagesize)) {
67      limit = NGX_MAX_SIZE_T_VALUE - ngx_pagesize;
68  }
69
70  send = 0;
71
72  header.iovs = headers;
73  header.nalloc = NGX\_IOVS\_PREALLOCATE;
74
75  trailer.iovs = trailers;
76  trailer.nalloc = NGX\_IOVS\_PREALLOCATE;
77
78  for ( ;; ) {
79      eintr = 0;
80      prev_send = send;
81
82      /* create the header iovec and coalesce the neighbouring bufs */
83
84      cl = ngx\_output\_chain\_to\_iovec(&header, in, limit - send, c->log);
85
86      if (cl == NGX\_CHAIN\_ERROR) {
87          return NGX\_CHAIN\_ERROR;
88      }
89
90      send += header.size;
91
92      if (cl && cl->buf->in_file && send < limit) {
93          file = cl->buf;
94
95          /* coalesce the neighbouring file bufs */
96
97          file_size = ngx\_chain\_coalesce\_file(&cl, limit - send);
98
99          send += file_size;
100
101          if (header.count == 0) {
102
103              /*
104               * create the trailer iovec and coalesce the neighbouring bufs
105               */
106
107              cl = ngx\_output\_chain\_to\_iovec(&trailer, cl, limit - send,
108                                          c->log);
109
110              if (cl == NGX\_CHAIN\_ERROR) {
111                  return NGX\_CHAIN\_ERROR;
112              }
113
114              send += trailer.size;
115          } else {
116              trailer.count = 0;
117          }
118
119          /*
120           * sendfile() returns EINVAL if sf_hdr's count is 0,
121           * but corresponding pointer is not NULL
122           */
123
124          hdr.headers = header.count ? header.iovs : NULL;
125          hdr.hdr_cnt = header.count;
126          hdr.trailers = trailer.count ? trailer.iovs : NULL;
127          hdr.trl_cnt = trailer.count;
128
129          sent = header.size + file_size;
130
131          ngx\_log\_debug3(NGX\_LOG\_DEBUG\_EVENT, c->log, 0,
132                     "sendfile: @%0 %0 h:%uz",
133                     file->file_pos, sent, header.size);
134
135          rc = sendfile(file->file->fd, c->fd, file->file_pos,
136                      &sent, &hdr, 0);
137
138          if (rc == -1) {

```

```

139     err = ngx_errno;
140
141     switch (err) {
142     case NGX_EAGAIN:
143         break;
144
145     case NGX_EINTR:
146         eintr = 1;
147         break;
148
149     default:
150         wev->error = 1;
151         (void) ngx_connection_error(c, err, "sendfile() failed");
152         return NGX_CHAIN_ERROR;
153     }
154
155     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, c->log, err,
156                  "sendfile() sent only %O bytes", sent);
157 }
158
159 if (rc == 0 && sent == 0) {
160
161     /*
162      * if rc and sent equal to zero, then someone
163      * has truncated the file, so the offset became beyond
164      * the end of the file
165      */
166
167     ngx_log_error(NGX_LOG_ALERT, c->log, 0,
168                  "sendfile() reported that \"%s\" was truncated",
169                  file->file->name.data);
170
171     return NGX_CHAIN_ERROR;
172 }
173
174 ngx_log_debug4(NGX_LOG_DEBUG_EVENT, c->log, 0,
175               "sendfile: %d, @%O %O:%O",
176               rc, file->file_pos, sent, file_size + header.size);
177
178 } else {
179     n = ngx_writev(c, &header);
180
181     if (n == NGX_ERROR) {
182         return NGX_CHAIN_ERROR;
183     }
184
185     sent = (n == NGX_AGAIN) ? 0 : n;
186 }
187
188 c->sent += sent;
189
190 in = ngx_chain_update_sent(in, sent);
191
192 if (eintr) {
193     send = prev_send + sent;
194     continue;
195 }
196
197 if (send - prev_send != sent) {
198     wev->ready = 0;
199     return in;
200 }
201
202 if (send >= limit || in == NULL) {
203     return in;
204 }
205 }
206 }

```

[One Level Up](#)

[Top Level](#)

# src/os/unix/nginx\_darwin\_init.c - nginx-1.7.10

## Global variables defined

- [ngx\\_darwin\\_hw\\_ncpu](#)
- [ngx\\_darwin\\_io](#)
- [ngx\\_darwin\\_kern\\_ipc\\_somaxconn](#)
- [ngx\\_darwin\\_kern\\_osrelease](#)
- [ngx\\_darwin\\_kern\\_ostype](#)
- [ngx\\_darwin\\_net\\_inet\\_tcp\\_sendspace](#)
- [ngx\\_debug\\_malloc](#)
- [sysctls](#)

## Data types defined

- [sysctl\\_t](#)

## Functions defined

- [ngx\\_debug\\_init](#)
- [ngx\\_os\\_specific\\_init](#)
- [ngx\\_os\\_specific\\_status](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10
11
12 char    ngx_darwin_kern_ostype[16];
13 char    ngx_darwin_kern_osrelease[128];
14 int     ngx_darwin_hw_ncpu;
15 int     ngx_darwin_kern_ipc_somaxconn;
16 u_long  ngx_darwin_net_inet_tcp_sendspace;
17
18 ngx_uint_t  ngx_debug_malloc;
19
20
21 static ngx_os_io_t ngx_darwin_io = {
22     ngx_unix_recv,
23     ngx_readv_chain,
24     ngx_udp_unix_recv,
25     ngx_unix_send,
26 #if (NGX_HAVE_SENDFILE)
27     ngx_darwin_sendfile_chain,
28     NGX_IO_SENDFILE
29 #else
30     ngx_writev_chain,
```

```

31     0
32 #endif
33 };
34
35
36 typedef struct {
37     char      *name;
38     void      *value;
39     size_t    size;
40     ngx\_uint\_t  exists;
41 } sysctl\_t;
42
43
44 sysctl\_t sysctls[] = {
45     { "hw.ncpu",
46       &ngx\_darwin\_hw\_ncpu,
47       sizeof(ngx\_darwin\_hw\_ncpu), 0 },
48
49     { "net.inet.tcp.sendspace",
50       &ngx\_darwin\_net\_inet\_tcp\_sendspace,
51       sizeof(ngx\_darwin\_net\_inet\_tcp\_sendspace), 0 },
52
53     { "kern.ipc.somaxconn",
54       &ngx\_darwin\_kern\_ipc\_somaxconn,
55       sizeof(ngx\_darwin\_kern\_ipc\_somaxconn), 0 },
56
57     { NULL, NULL, 0, 0 }
58 };
59
60
61 void
62 ngx\_debug\_init(void)
63 {
64     #if (NGX_DEBUG_MALLOC)
65
66     /*
67      * MacOSX 10.6, 10.7: MallocScribble fills freed memory with 0x55
68      *                    and fills allocated memory with 0xAA.
69      * MacOSX 10.4, 10.5: MallocScribble fills freed memory with 0x55,
70      *                    MallocPreScribble fills allocated memory with 0xAA.
71      * MacOSX 10.3:      MallocScribble fills freed memory with 0x55,
72      *                    and no way to fill allocated memory.
73      */
74
75     setenv("MallocScribble", "1", 0);
76
77     ngx\_debug\_malloc = 1;
78
79 #else
80
81     if (getenv("MallocScribble")) {
82         ngx\_debug\_malloc = 1;
83     }
84
85 #endif
86 }
87
88
89 ngx\_int\_t
90 ngx\_os\_specific\_init(ngx\_log\_t *log)
91 {
92     size_t    size;
93     ngx\_err\_t  err;
94     ngx\_uint\_t  i;
95
96     size = sizeof(ngx\_darwin\_kern\_ostype);
97     if (sysctlbyname("kern.ostype", ngx\_darwin\_kern\_ostype, &size, NULL, 0)
98         == -1)
99     {
100         err = ngx\_errno;
101
102         if (err != NGX\_ENOENT) {
103
104             ngx\_log\_error(NGX\_LOG\_ALERT, log, err,
105                          "sysctlbyname(kern.ostype) failed");
106

```

```

107         if (err != NGX_ENOMEM) {
108             return NGX_ERROR;
109         }
110
111         ngx_darwin kern_ostype[size - 1] = '\0';
112     }
113 }
114
115 size = sizeof(ngx_darwin kern_osrelease);
116 if (sysctlbyname("kern.osrelease", ngx_darwin kern_osrelease, &size,
117                 NULL, 0)
118     == -1)
119 {
120     err = ngx_errno;
121
122     if (err != NGX_ENOENT) {
123
124         ngx_log_error(NGX_LOG_ALERT, log, err,
125                     "sysctlbyname(kern.osrelease) failed");
126
127         if (err != NGX_ENOMEM) {
128             return NGX_ERROR;
129         }
130
131         ngx_darwin kern_osrelease[size - 1] = '\0';
132     }
133 }
134
135 for (i = 0; sysctls[i].name; i++) {
136     size = sysctls[i].size;
137
138     if (sysctlbyname(sysctls[i].name, sysctls[i].value, &size, NULL, 0)
139         == 0)
140     {
141         sysctls[i].exists = 1;
142         continue;
143     }
144
145     err = ngx_errno;
146
147     if (err == NGX_ENOENT) {
148         continue;
149     }
150
151     ngx_log_error(NGX_LOG_ALERT, log, err,
152                 "sysctlbyname(%s) failed", sysctls[i].name);
153     return NGX_ERROR;
154 }
155
156 ngx_ncpu = ngx_darwin_hw_ncpu;
157
158 if (ngx_darwin kern_ipc_somaxconn > 32767) {
159     ngx_log_error(NGX_LOG_ALERT, log, 0,
160                 "sysctl kern.ipc.somaxconn must be less than 32768");
161     return NGX_ERROR;
162 }
163
164 ngx_tcp_nodelay and tcp_nopush = 1;
165
166 ngx_os_io = ngx_darwin_io;
167
168 return NGX_OK;
169 }
170
171 void
172 ngx_os_specific_status(ngx_log_t *log)
173 {
174     u_long    value;
175     ngx_uint_t i;
176
177     if (ngx_darwin kern_ostype[0]) {
178         ngx_log_error(NGX_LOG_NOTICE, log, 0, "OS: %s %s",
179                     ngx_darwin kern_ostype, ngx_darwin kern_osrelease);
180     }
181 }
182

```



```
183 for (i = 0; sysctls[i].name; i++) {
184     if (sysctls[i].exists) {
185         if (sysctls[i].size == sizeof(long)) {
186             value = *(long *) sysctls[i].value;
187
188         } else {
189             value = *(int *) sysctls[i].value;
190         }
191
192         ngx_log_error(NGX_LOG_NOTICE, log, 0, "%s: %l",
193                     sysctls[i].name, value);
194     }
195 }
196 }
```

[One Level Up](#)

[Top Level](#)

## src/os/unix/nginx\_freebsd.h - nginx-1.7.10

### Macros defined

- [\\_NGX\\_FREEBSD\\_H\\_INCLUDED](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_FREEBSD\_H\_INCLUDED
9 #define \_NGX\_FREEBSD\_H\_INCLUDED
10
11
12 void ngx\_debug\_init(void);
13 ngx\_chain\_t *ngx\_freebsd\_sendfile\_chain(ngx\_connection\_t *c, ngx\_chain\_t *in,
14     off_t limit);
15
16 extern int      ngx\_freebsd\_kern\_osreldate;
17 extern int      ngx\_freebsd\_hw\_ncpu;
18 extern u_long   ngx\_freebsd\_net\_inet\_tcp\_sendspace;
19
20 extern ngx\_uint\_t ngx\_freebsd\_sendfile\_nbytes\_bug;
21 extern ngx\_uint\_t ngx\_freebsd\_use\_tcp\_nopush;
22 extern ngx\_uint\_t ngx\_debug\_malloc;
23
24
25 #endif /* \_NGX\_FREEBSD\_H\_INCLUDED */
```

# src/os/unix/nginx\_gcc\_atomic\_amd64.h - nginx-1.7.10

## Functions defined

- [ngx\\_atomic\\_cmp\\_set](#)
- [ngx\\_atomic\\_fetch\\_add](#)

## Macros defined

- [NGX\\_SMP\\_LOCK](#)
- [NGX\\_SMP\\_LOCK](#)
- [ngx\\_cpu\\_pause](#)
- [ngx\\_memory\\_barrier](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #if (NGX_SMP)
9 #define NGX_SMP_LOCK "lock;"
10 #else
11 #define NGX_SMP_LOCK
12 #endif
13
14
15 /*
16  * "cmpxchgq r, [m]":
17  *
18  *     if (rax == [m]) {
19  *         zf = 1;
20  *         [m] = r;
21  *     } else {
22  *         zf = 0;
23  *         rax = [m];
24  *     }
25  *
26  *
27  * The "r" is any register, %rax (%r0) - %r16.
28  * The "=a" and "a" are the %rax register.
29  * Although we can return result in any register, we use "a" because it is
30  * used in cmpxchgq anyway. The result is actually in %al but not in $rax,
31  * however as the code is inlined gcc can test %al as well as %rax.
32  *
33  * The "cc" means that flags were changed.
34  */
35
36 static ngx_inline ngx_atomic_uint_t
37 ngx_atomic_cmp_set(ngx_atomic_t *lock, ngx_atomic_uint_t old,
38 ngx_atomic_uint_t set)
39 {
40     u_char res;
41
42     __asm__ volatile (
43
44         NGX_SMP_LOCK
45         "    cmpxchgq %3, %1;    "
46         "    sete     %0;      "
47
48         : "=a" (res) : "m" (*lock), "a" (old), "r" (set) : "cc", "memory");
```

```

49     return res;
50 }
51 }
52
53
54 /*
55  * "xaddq r, [m]":
56  *
57  *     temp = [m];
58  *     [m] += r;
59  *     r = temp;
60  *
61  *
62  * The "r" is any register, %rax (%r0) - %r16.
63  * The "cc" means that flags were changed.
64  */
65
66 static ngx\_inline ngx\_atomic\_int\_t
67 ngx_atomic_fetch_add(ngx\_atomic\_t *value, ngx\_atomic\_int\_t add)
68 {
69     __asm__ volatile (
70
71         NGX\_SMP\_LOCK
72         "    xaddq %0, %1;    "
73
74         : "+r" (add) : "m" (*value) : "cc", "memory");
75
76     return add;
77 }
78
79
80 #define ngx_memory_barrier()    __asm__ volatile (" ::: "memory")
81
82 #define ngx_cpu_pause()        __asm__ ("pause")

```

[One Level Up](#)

[Top Level](#)

# src/os/unix/nginx\_gcc\_atomic\_x86.h - nginx-1.7.10

## Functions defined

- [ngx\\_atomic\\_cmp\\_set](#)
- [ngx\\_atomic\\_fetch\\_add](#)
- [ngx\\_atomic\\_fetch\\_add](#)

## Macros defined

- [NGX\\_SMP\\_LOCK](#)
- [NGX\\_SMP\\_LOCK](#)
- [ngx\\_cpu\\_pause](#)
- [ngx\\_memory\\_barrier](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #if (NGX_SMP)
9 #define NGX_SMP_LOCK "lock;"
10 #else
11 #define NGX_SMP_LOCK
12 #endif
13
14
15 /*
16  * "cmpxchgl r, [m]":
17  *
18  *     if (eax == [m]) {
19  *         zf = 1;
20  *         [m] = r;
21  *     } else {
22  *         zf = 0;
23  *         eax = [m];
24  *     }
25  *
26  *
27  * The "r" means the general register.
28  * The "=a" and "a" are the %eax register.
29  * Although we can return result in any register, we use "a" because it is
30  * used in cmpxchgl anyway. The result is actually in %al but not in %eax,
31  * however, as the code is inlined gcc can test %al as well as %eax,
32  * and icc adds "movzbl %al, %eax" by itself.
33  *
34  * The "cc" means that flags were changed.
35  */
36
37 static ngx\_inline ngx\_atomic\_uint\_t
38 ngx_atomic_cmp_set(ngx\_atomic\_t *lock, ngx\_atomic\_uint\_t old,
39 ngx\_atomic\_uint\_t set)
40 {
41     u_char res;
42
43     __asm__ volatile (
44         NGX\_SMP\_LOCK
```

```

46     "    cmpxchgl  %3, %1;    "
47     "    sete    %0;        "
48
49     : "=a" (res) : "m" (*lock), "a" (old), "r" (set) : "cc", "memory");
50
51     return res;
52 }
53
54
55 /*
56 * "xaddl r, [m]":
57 *
58 *     temp = [m];
59 *     [m] += r;
60 *     r = temp;
61 *
62 *
63 * The "+r" means the general register.
64 * The "cc" means that flags were changed.
65 */
66
67
68 #if !(( __GNUC__ == 2 && __GNUC_MINOR__ <= 7 ) || ( __INTEL_COMPILER >= 800 ))
69
70 /*
71 * icc 8.1 and 9.0 compile broken code with -march=pentium4 option:
72 * ngx_atomic_fetch_add() always return the input "add" value,
73 * so we use the gcc 2.7 version.
74 *
75 * icc 8.1 and 9.0 with -march=pentiumpro option or icc 7.1 compile
76 * correct code.
77 */
78
79 static ngx_inline ngx_atomic_int_t
80 ngx_atomic_fetch_add(ngx_atomic_t *value, ngx_atomic_int_t add)
81 {
82     __asm__ volatile (
83
84         NGX_SMP_LOCK
85         "    xaddl  %0, %1;    "
86
87         : "+r" (add) : "m" (*value) : "cc", "memory");
88
89     return add;
90 }
91
92
93 #else
94
95 /*
96 * gcc 2.7 does not support "+r", so we have to use the fixed
97 * %eax ("=a" and "a") and this adds two superfluous instructions in the end
98 * of code, something like this: "mov %eax, %edx / mov %edx, %eax".
99 */
100
101 static ngx_inline ngx_atomic_int_t
102 ngx_atomic_fetch_add(ngx_atomic_t *value, ngx_atomic_int_t add)
103 {
104     ngx_atomic_uint_t old;
105
106     __asm__ volatile (
107
108         NGX_SMP_LOCK
109         "    xaddl  %2, %1;    "
110
111         : "=a" (old) : "m" (*value), "a" (add) : "cc", "memory");
112
113     return old;
114 }
115
116 #endif
117
118
119 /*
120 * on x86 the write operations go in a program order, so we need only
121 * to disable the gcc reorder optimizations

```

```
122 */
123
124 #define ngx_memory_barrier()    __asm__ volatile ("" ::: "memory")
125
126 /* old "as" does not support "pause" opcode */
127 #define ngx_cpu_pause()         __asm__ (".byte 0xf3, 0x90")
```

[One Level Up](#)

[Top Level](#)

## src/os/unix/nginx\_gcc\_atomic\_ppc.h - nginx-1.7.10

### Functions defined

- [ngx\\_atomic\\_cmp\\_set](#)
- [ngx\\_atomic\\_cmp\\_set](#)
- [ngx\\_atomic\\_fetch\\_add](#)
- [ngx\\_atomic\\_fetch\\_add](#)

### Macros defined

- [ngx\\_cpu\\_pause](#)
- [ngx\\_memory\\_barrier](#)
- [ngx\\_memory\\_barrier](#)
- [ngx\\_memory\\_barrier](#)
- [ngx\\_memory\\_barrier](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 /*
9  * The ppc assembler treats ";" as comment, so we have to use "\n".
10 * The minus in "bne-" is a hint for the branch prediction unit that
11 * this branch is unlikely to be taken.
12 * The "1b" means the nearest backward label "1" and the "1f" means
13 * the nearest forward label "1".
14 *
15 * The "b" means that the base registers can be used only, i.e.
16 * any register except r0. The r0 register always has a zero value and
17 * could not be used in "addi r0, r0, 1".
18 * The "&b" means that no input registers can be used.
19 *
20 * "sync"    read and write barriers
21 * "isync"   read barrier, is faster than "sync"
22 * "eieio"   write barrier, is faster than "sync"
23 * "lwsync"  write barrier, is faster than "eieio" on ppc64
24 */
25
26 #if (NGX_PTR_SIZE == 8)
27
28 static ngx\_inline ngx\_atomic\_uint\_t
29 ngx\_atomic\_cmp\_set(ngx\_atomic\_t *lock, ngx\_atomic\_uint\_t old,
30 ngx\_atomic\_uint\_t set)
31 {
32     ngx\_atomic\_uint\_t res, temp;
33
34     __asm__ volatile (
35
36         "    li      %0, 0          \n" /* preset "0" to "res"          */
37         "    lwsync                    \n" /* write barrier                */
38         "1:
39         "    ldarx   %1, 0, %2      \n" /* load from [lock] into "temp" */
40         "                                /* and store reservation        */
41         "    cmpd    %1, %3          \n" /* compare "temp" and "old"     */
```



```

42     "    bne-    2f        \n" /* not equal */
43     "    stdcx.  %4, 0, %2 \n" /* store "set" into [lock] if reservation */
44                                     /* is not cleared */
45     "    bne-    1b        \n" /* the reservation was cleared */
46     "    isync   \n" /* read barrier */
47     "    li      %0, 1     \n" /* set "1" to "res" */
48     "2:        \n"
49
50     : "&b" (res), "&b" (temp)
51     : "b" (lock), "b" (old), "b" (set)
52     : "cc", "memory");
53
54     return res;
55 }
56
57
58 static ngx_inline ngx_atomic_int_t
59 ngx_atomic_fetch_add(ngx_atomic_t *value, ngx_atomic_int_t add)
60 {
61     ngx_atomic_uint_t res, temp;
62
63     __asm__ volatile (
64
65     "    lwsync          \n" /* write barrier */
66     "1: ldarx    %0, 0, %2 \n" /* load from [value] into "res" */
67                                     /* and store reservation */
68     "    add      %1, %0, %3 \n" /* "res" + "add" store in "temp" */
69     "    stdcx.  %1, 0, %2 \n" /* store "temp" into [value] if reservation */
70                                     /* is not cleared */
71     "    bne-    1b        \n" /* try again if reservation was cleared */
72     "    isync   \n" /* read barrier */
73
74     : "&b" (res), "&b" (temp)
75     : "b" (value), "b" (add)
76     : "cc", "memory");
77
78     return res;
79 }
80
81
82 #if (NGX_SMP)
83 #define ngx_memory_barrier() \
84     __asm__ volatile ("isync \n lwsync \n" ::: "memory")
85 #else
86 #define ngx_memory_barrier() __asm__ volatile (" " ::: "memory")
87 #endif
88
89 #else
90
91 static ngx_inline ngx_atomic_uint_t
92 ngx_atomic_cmp_set(ngx_atomic_t *lock, ngx_atomic_uint_t old,
93 ngx_atomic_uint_t set)
94 {
95     ngx_atomic_uint_t res, temp;
96
97     __asm__ volatile (
98
99     "    li      %0, 0     \n" /* preset "0" to "res" */
100    "    eieio          \n" /* write barrier */
101    "1:              \n"
102    "    lwarx   %1, 0, %2 \n" /* load from [lock] into "temp" */
103                                     /* and store reservation */
104    "    cmpw   %1, %3     \n" /* compare "temp" and "old" */
105    "    bne-   2f        \n" /* not equal */
106    "    stwcx. %4, 0, %2 \n" /* store "set" into [lock] if reservation */
107                                     /* is not cleared */
108    "    bne-   1b        \n" /* the reservation was cleared */
109    "    isync \n" /* read barrier */
110    "    li     %0, 1     \n" /* set "1" to "res" */
111    "2:              \n"
112
113    : "&b" (res), "&b" (temp)
114    : "b" (lock), "b" (old), "b" (set)
115    : "cc", "memory");
116
117     return res;

```

```

118 }
119
120
121 static ngx_inline ngx_atomic_int_t
122 ngx_atomic_fetch_add(ngx_atomic_t *value, ngx_atomic_int_t add)
123 {
124     ngx_atomic_uint_t res, temp;
125
126     __asm__ volatile (
127
128         "    eieio          \n" /* write barrier          */
129         "1:  lwarx    %0, 0, %2 \n" /* load from [value] into "res" */
130         "        and store reservation          */
131         "    add      %1, %0, %3 \n" /* "res" + "add" store in "temp" */
132         "    stwcx.  %1, 0, %2 \n" /* store "temp" into [value] if reservation */
133         "                is not cleared          */
134         "    bne-   1b      \n" /* try again if reservation was cleared */
135         "    isync          \n" /* read barrier          */
136
137         : "=&b" (res), "=&b" (temp)
138         : "b" (value), "b" (add)
139         : "cc", "memory");
140
141     return res;
142 }
143
144
145 #if (NGX_SMP)
146 #define ngx_memory_barrier() \
147     __asm__ volatile ("isync \n eieio \n" ::: "memory")
148 #else
149 #define ngx_memory_barrier() __asm__ volatile (" " ::: "memory")
150 #endif
151
152 #endif
153
154
155 #define ngx_cpu_pause()

```

[One Level Up](#)

[Top Level](#)

## src/os/unix/nginx\_linux.h - nginx-1.7.10

### Macros defined

- [\\_NGX\\_LINUX\\_H\\_INCLUDED](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_LINUX\_H\_INCLUDED
9 #define \_NGX\_LINUX\_H\_INCLUDED
10
11
12 ngx\_chain\_t *ngx\_linux\_sendfile\_chain(ngx\_connection\_t *c, ngx\_chain\_t *in,
13     off\_t limit);
14
15 extern int ngx\_linux\_rtsig\_max;
16
17
18 #endif /* \_NGX\_LINUX\_H\_INCLUDED */
```

## src/os/unix/nginx\_setaffinity.h - nginx-1.7.10

### Macros defined

- [NGX\\_HAVE\\_CPU\\_AFFINITY](#)
- [\\_NGX\\_SETAFFINITY\\_H\\_INCLUDED](#)
- [ngx\\_setaffinity](#)

### Source code

```
1
2 /*
3  * Copyright (C) Nginx, Inc.
4  */
5
6 #ifndef \_NGX\_SETAFFINITY\_H\_INCLUDED
7 #define \_NGX\_SETAFFINITY\_H\_INCLUDED
8
9
10 #if (NGX_HAVE_SCHED_SETAFFINITY || NGX_HAVE_CPUSET_SETAFFINITY)
11
12 #define NGX\_HAVE\_CPU\_AFFINITY 1
13
14 void ngx\_setaffinity(uint64_t cpu_affinity, ngx\_log\_t *log);
15
16 #else
17
18 #define ngx\_setaffinity(cpu_affinity, log)
19
20 #endif
21
22
23 #endif /* \_NGX\_SETAFFINITY\_H\_INCLUDED */
```

# src/os/unix/nginx\_setproctitle.c - nginx-1.7.10

## Global variables defined

- [ngx\\_os\\_argv\\_last](#)

## Functions defined

- [ngx\\_init\\_setproctitle](#)
- [ngx\\_setproctitle](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10
11
12 #if (NGX_SETPROCTITLE_USES_ENV)
13
14 /*
15  * To change the process title in Linux and Solaris we have to set argv[1]
16  * to NULL and to copy the title to the same place where the argv[0] points to.
17  * However, argv[0] may be too small to hold a new title. Fortunately, Linux
18  * and Solaris store argv[] and environ[] one after another. So we should
19  * ensure that is the continuous memory and then we allocate the new memory
20  * for environ[] and copy it. After this we could use the memory starting
21  * from argv[0] for our process title.
22  *
23  * The Solaris's standard /bin/ps does not show the changed process title.
24  * You have to use "/usr/ucb/ps -w" instead. Besides, the UCB ps does not
25  * show a new title if its length less than the origin command line length.
26  * To avoid it we append to a new title the origin command line in the
27  * parenthesis.
28  */
29
30 extern char **environ;
31
32 static char *ngx_os_argv_last;
33
34 ngx\_int\_t
35 ngx\_init\_setproctitle(ngx\_log\_t *log)
36 {
37     u\_char      *p;
38     size\_t      size;
39     ngx\_uint\_t  i;
40
41     size = 0;
42
43     for (i = 0; environ[i]; i++) {
44         size += ngx\_strlen(environ[i]) + 1;
45     }
46
47     p = ngx\_alloc(size, log);
48     if (p == NULL) {
49         return NGX\_ERROR;
50     }
51
52     ngx\_os\_argv\_last = ngx\_os\_argv[0];
53
54     for (i = 0; ngx\_os\_argv[i]; i++) {
```

```

55     if (ngx_os_argv_last == ngx_os_argv[i]) {
56         ngx_os_argv_last = ngx_os_argv[i] + ngx_strlen(ngx_os_argv[i]) + 1;
57     }
58 }
59
60 for (i = 0; environ[i]; i++) {
61     if (ngx_os_argv_last == environ[i]) {
62
63         size = ngx_strlen(environ[i]) + 1;
64         ngx_os_argv_last = environ[i] + size;
65
66         ngx_cpysrtn(p, (u_char *) environ[i], size);
67         environ[i] = (char *) p;
68         p += size;
69     }
70 }
71
72 ngx_os_argv_last--;
73
74 return NGX_OK;
75 }
76
77
78 void
79 ngx_setproctitle(char *title)
80 {
81     u_char    *p;
82
83     #if (NGX_SOLARIS)
84
85     ngx_int_t  i;
86     size_t     size;
87
88     #endif
89
90     ngx_os_argv[1] = NULL;
91
92     p = ngx_cpysrtn((u_char *) ngx_os_argv[0], (u_char *) "nginx: ",
93                   ngx_os_argv_last - ngx_os_argv[0]);
94
95     p = ngx_cpysrtn(p, (u_char *) title, ngx_os_argv_last - (char *) p);
96
97     #if (NGX_SOLARIS)
98
99     size = 0;
100
101     for (i = 0; i < ngx_argc; i++) {
102         size += ngx_strlen(ngx_argv[i]) + 1;
103     }
104
105     if (size > (size_t) ((char *) p - ngx_os_argv[0])) {
106
107         /*
108          * ngx_setproctitle() is too rare operation so we use
109          * the non-optimized copies
110          */
111
112         p = ngx_cpysrtn(p, (u_char *) " (" , ngx_os_argv_last - (char *) p);
113
114         for (i = 0; i < ngx_argc; i++) {
115             p = ngx_cpysrtn(p, (u_char *) ngx_argv[i],
116                           ngx_os_argv_last - (char *) p);
117             p = ngx_cpysrtn(p, (u_char *) " ", ngx_os_argv_last - (char *) p);
118         }
119
120         if (*(p - 1) == ' ') {
121             *(p - 1) = '\0';
122         }
123     }
124
125     #endif
126
127     if (ngx_os_argv_last - (char *) p) {
128         ngx_memset(p, NGX_SETPROCTITLE_PAD, ngx_os_argv_last - (char *) p);
129     }
130

```

```
131     ngx_log_debug1(NGX_LOG_DEBUG_CORE, ngx_cycle->log, 0,  
132                   "setproctitle: \"%s\"", ngx_os_argv[0]);  
133 }  
134  
135 #endif /* NGX_SETPROCTITLE_USES_ENV */
```

[One Level Up](#)

[Top Level](#)

## src/os/unix/nginx\_solaris.h - nginx-1.7.10

### Macros defined

- [\\_NGX\\_SOLARIS\\_H\\_INCLUDED\\_](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_SOLARIS\_H\_INCLUDED\_
9 #define \_NGX\_SOLARIS\_H\_INCLUDED\_
10
11
12 ngx\_chain\_t *ngx\_solaris\_sendfilev\_chain(ngx\_connection\_t *c, ngx\_chain\_t *in,
13     off\_t limit);
14
15
16 #endif /* \_NGX\_SOLARIS\_H\_INCLUDED\_ */
```



# src/os/unix/nginx\_solaris\_init.c - nginx-1.7.10

## Global variables defined

- [ngx\\_solaris\\_io](#)
- [ngx\\_solaris\\_release](#)
- [ngx\\_solaris\\_sysname](#)
- [ngx\\_solaris\\_version](#)

## Functions defined

- [ngx\\_os\\_specific\\_init](#)
- [ngx\\_os\\_specific\\_status](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10
11
12 char ngx_solaris_sysname[20];
13 char ngx_solaris_release[10];
14 char ngx_solaris_version[50];
15
16
17 static ngx_os_io_t ngx_solaris_io = {
18     ngx_unix_recv,
19     ngx_ready_chain,
20     ngx_udp_unix_recv,
21     ngx_unix_send,
22     #if (NGX_HAVE_SENDFILE)
23     ngx_solaris_sendfilev_chain,
24     NGX_IO_SENDFILE
25     #else
26     ngx_writev_chain,
27     0
28     #endif
29 };
30
31
32 ngx_int_t
33 ngx_os_specific_init(ngx_log_t *log)
34 {
35     if (sysinfo(SI_SYSNAME, ngx_solaris_sysname, sizeof(ngx_solaris_sysname))
36         == -1)
37     {
38         ngx_log_error(NGX_LOG_ALERT, log, ngx_errno,
39                     "sysinfo(SI_SYSNAME) failed");
40         return NGX_ERROR;
41     }
42
43     if (sysinfo(SI_RELEASE, ngx_solaris_release, sizeof(ngx_solaris_release))
44         == -1)
45     {
46         ngx_log_error(NGX_LOG_ALERT, log, ngx_errno,
47                     "sysinfo(SI_RELEASE) failed");
48         return NGX_ERROR;
49     }
50 }
```

```
49     }
50
51     if (sysinfo(SI_VERSION, ngx_solaris_version, sizeof(ngx_solaris_version))
52         == -1)
53     {
54         ngx_log_error(NGX_LOG_ALERT, log, ngx_errno,
55                     "sysinfo(SI_SYSNAME) failed");
56         return NGX_ERROR;
57     }
58
59     ngx_os_io = ngx_solaris_io;
60
61     return NGX_OK;
62 }
63
64
65
66 void
67 ngx_os_specific_status(ngx_log_t *log)
68 {
69
70     ngx_log_error(NGX_LOG_NOTICE, log, 0, "OS: %s %s",
71                 ngx_solaris_sysname, ngx_solaris_release);
72
73     ngx_log_error(NGX_LOG_NOTICE, log, 0, "version: %s",
74                 ngx_solaris_version);
75 }
```

[One Level Up](#)

[Top Level](#)

# src/os/unix/nginx\_sunpro\_atomic\_sparc64.h - nginx-1.7.10

## Functions defined

- [ngx\\_atomic\\_cmp\\_set](#)
- [ngx\\_atomic\\_fetch\\_add](#)

## Macros defined

- [NGX\\_CASA](#)
- [NGX\\_CASA](#)
- [ngx\\_cpu\\_pause](#)
- [ngx\\_memory\\_barrier](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #if (NGX_PTR_SIZE == 4)
9 #define NGX_CASA ngx_casa
10 #else
11 #define NGX_CASA ngx_casxa
12 #endif
13
14
15 ngx\_atomic\_uint\_t
16 ngx_casa(ngx\_atomic\_uint\_t set, ngx\_atomic\_uint\_t old, ngx\_atomic\_t \*lock);
17
18 ngx\_atomic\_uint\_t
19 ngx_casxa(ngx\_atomic\_uint\_t set, ngx\_atomic\_uint\_t old, ngx\_atomic\_t \*lock);
20
21 /* the code in src/os/unix/nginx_sunpro_sparc64.il */
22
23
24 static ngx\_inline ngx\_atomic\_uint\_t
25 ngx_atomic_cmp_set(ngx\_atomic\_t \*lock, ngx\_atomic\_uint\_t old,
26 ngx\_atomic\_uint\_t set)
27 {
28     set = NGX_CASA(set, old, lock);
29
30     return (set == old);
31 }
32
33
34 static ngx\_inline ngx\_atomic\_int\_t
35 ngx_atomic_fetch_add(ngx\_atomic\_t \*value, ngx\_atomic\_int\_t add)
36 {
37     ngx\_atomic\_uint\_t old, res;
38
39     old = *value;
40
41     for ( ;; ) {
42
43         res = old + add;
44
45         res = NGX_CASA(res, old, value);
46
47         if (res == old) {
48             return res;
49         }
50     }
```

```
49     }
50
51     old = res;
52 }
53 }
54
55
56 #define ngx_memory_barrier()           \
57     __asm (".volatile");              \
58     __asm ("membar #LoadLoad | #LoadStore | #StoreStore | #StoreLoad");    \
59     __asm (".nonvolatile")
60
61 #define ngx_cpu_pause()
```

[One Level Up](#)

[Top Level](#)

# src/os/unix/rfork\_thread.S - nginx-1.7.10

## Macros defined

- [KERNCALL](#)

## Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <sys/syscall.h>
9  #include <machine/asm.h>
10
11  /*
12  * rfork_thread(3) - rfork_thread(flags, stack, func, arg);
13  */
14
15  #define      KERNCALL      int $0x80
16
17  ENTRY(rfork_thread)
18      push    %ebp
19      mov     %esp, %ebp
20      push   %esi
21
22      mov     12(%ebp), %esi      # the thread stack address
23
24      sub     $4, %esi
25      mov     20(%ebp), %eax     # the thread argument
26      mov     %eax, (%esi)
27
28      sub     $4, %esi
29      mov     16(%ebp), %eax     # the thread start address
30      mov     %eax, (%esi)
31
32      push   8(%ebp)             # rfork(2) flags
33      push   $0
34      mov     $SYS_rfork, %eax
35      KERNCALL
36      jc     error
37
38      cmp     $0, %edx
39      jne    child
40
41  parent:
42      add     $8, %esp
43      pop    %esi
44      leave
45      ret
46
47  child:
48      mov     %esi, %esp
49      pop    %eax
50      call   *%eax              # call a thread start address ...
51      add     $4, %esp
52
53      push   %eax
54      push   $0
55      mov     $SYS_exit, %eax   # ... and exit(2) after a thread would return
56      KERNCALL
57
58  error:
59      add     $8, %esp
60      pop    %esi
61      leave
62  PIC_PROLOGUE
```

```
63      /* libc's cerror: jmp PIC_PLT(HIDENAME(cerror)) */
64
65
66      push    %eax
67      call   PIC_PLT(CNAME(__error))
68      pop    %ecx
69      PIC_EPILOGUE
70      mov    %ecx, (%eax)
71      mov    $-1, %eax
72      mov    $-1, %edx
73      ret
```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_crc.h - nginx-1.7.10

### Functions defined

- [ngx\\_crc](#)

### Macros defined

- [\\_NGX\\_CRC\\_H\\_INCLUDED](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_CRC\_H\_INCLUDED
9 #define \_NGX\_CRC\_H\_INCLUDED
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 /* 32-bit crc16 */
17
18 static ngx\_inline uint32_t
19 ngx_crc(u_char *data, size_t len)
20 {
21     uint32_t sum;
22
23     for (sum = 0; len; len--) {
24
25         /*
26          * gcc 2.95.2 x86 and icc 7.1.006 compile
27          * that operator into the single "rol" opcode,
28          * msvc 6.0sp2 compiles it into four opcodes.
29          */
30         sum = sum >> 1 | sum << 31;
31
32         sum += *data++;
33     }
34
35     return sum;
36 }
37
38
39 #endif /* \_NGX\_CRC\_H\_INCLUDED */
```

## src/http/nginx\_http\_copy\_filter\_module.c - nginx-1.7.10

### Global variables defined

- [ngx\\_http\\_copy\\_filter\\_commands](#)
- [ngx\\_http\\_copy\\_filter\\_module](#)
- [ngx\\_http\\_copy\\_filter\\_module\\_ctx](#)
- [ngx\\_http\\_next\\_body\\_filter](#)

### Data types defined

- [ngx\\_http\\_copy\\_filter\\_conf\\_t](#)

### Functions defined

- [ngx\\_http\\_copy\\_aio\\_event\\_handler](#)
- [ngx\\_http\\_copy\\_aio\\_handler](#)
- [ngx\\_http\\_copy\\_aio\\_sendfile\\_event\\_handler](#)
- [ngx\\_http\\_copy\\_filter](#)
- [ngx\\_http\\_copy\\_filter\\_create\\_conf](#)
- [ngx\\_http\\_copy\\_filter\\_init](#)
- [ngx\\_http\\_copy\\_filter\\_merge\\_conf](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 typedef struct {
14     ngx_bufs_t  bufs;
15 } ngx_http_copy_filter_conf_t;
16
17
18 #if (NGX_HAVE_FILE_AIO)
19 static void ngx_http_copy_aio_handler(ngx_output_chain_ctx_t *ctx,
20     ngx_file_t *file);
21 static void ngx_http_copy_aio_event_handler(ngx_event_t *ev);
22 #if (NGX_HAVE_AIO_SENDFILE)
23 static void ngx_http_copy_aio_sendfile_event_handler(ngx_event_t *ev);
24 #endif
25 #endif
26
27 static void *ngx_http_copy_filter_create_conf(ngx_conf_t *cf);
28 static char *ngx_http_copy_filter_merge_conf(ngx_conf_t *cf,
29     void *parent, void *child);
30 static ngx_int_t ngx_http_copy_filter_init(ngx_conf_t *cf);
```



```

31
32
33 static ngx_command_t  ngx_http_copy_filter_commands[] = {
34
35     { ngx_string("output_buffers"),
36       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE2,
37       ngx_conf_set_bufs_slot,
38       NGX_HTTP_LOC_CONF_OFFSET,
39       offsetof(ngx_http_copy_filter_conf_t, bufs),
40       NULL },
41
42     ngx_null_command
43 };
44
45
46 static ngx_http_module_t  ngx_http_copy_filter_module_ctx = {
47     NULL,                                     /* preconfiguration */
48     ngx_http_copy_filter_init,               /* postconfiguration */
49
50     NULL,                                     /* create main configuration */
51     NULL,                                     /* init main configuration */
52
53     NULL,                                     /* create server configuration */
54     NULL,                                     /* merge server configuration */
55
56     ngx_http_copy_filter_create_conf,        /* create location configuration */
57     ngx_http_copy_filter_merge_conf         /* merge location configuration */
58 };
59
60
61 ngx_module_t  ngx_http_copy_filter_module = {
62     NGX_MODULE_V1,
63     &ngx_http_copy_filter_module_ctx,        /* module context */
64     ngx_http_copy_filter_commands,          /* module directives */
65     NGX_HTTP_MODULE,                        /* module type */
66     NULL,                                    /* init master */
67     NULL,                                    /* init module */
68     NULL,                                    /* init process */
69     NULL,                                    /* init thread */
70     NULL,                                    /* exit thread */
71     NULL,                                    /* exit process */
72     NULL,                                    /* exit master */
73     NGX_MODULE_V1_PADDING
74 };
75
76
77 static ngx_http_output_body_filter_pt  ngx_http_next_body_filter;
78
79
80 static ngx_int_t
81 ngx_http_copy_filter(ngx_http_request_t *r, ngx_chain_t *in)
82 {
83     ngx_int_t          rc;
84     ngx_connection_t  *c;
85     ngx_output_chain_ctx_t  *ctx;
86     ngx_http_core_loc_conf_t  *clcf;
87     ngx_http_copy_filter_conf_t  *conf;
88
89     c = r->connection;
90
91     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
92                  "http copy filter: \"%V?%V\"", &r->uri, &r->args);
93
94     ctx = ngx_http_get_module_ctx(r, ngx_http_copy_filter_module);
95
96     if (ctx == NULL) {
97         ctx = ngx_palloc(r->pool, sizeof(ngx_output_chain_ctx_t));
98         if (ctx == NULL) {
99             return NGX_ERROR;
100        }
101
102        ngx_http_set_ctx(r, ctx, ngx_http_copy_filter_module);
103
104        conf = ngx_http_get_module_loc_conf(r, ngx_http_copy_filter_module);
105        clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
106

```

```

107     ctx->sendfile = c->sendfile;
108     ctx->need_in_memory = r->main_filter_need_in_memory
109         || r->filter_need_in_memory;
110     ctx->need_in_temp = r->filter_need_temporary;
111
112     ctx->alignment = clcf->directio_alignment;
113
114     ctx->pool = r->pool;
115     ctx->bufs = conf->bufs;
116     ctx->tag = (ngx_buf_tag_t) &ngx_http_copy_filter_module;
117
118     ctx->output_filter = (ngx_output_chain_filter_pt)
119         ngx_http_next_body_filter;
120     ctx->filter_ctx = r;
121
122     #if (NGX_HAVE_FILE_AIO)
123         if (ngx_file_aio) {
124             if (clcf->aio) {
125                 ctx->aio_handler = ngx_http_copy_aio_handler;
126             }
127         }
128         #if (NGX_HAVE_AIO_SENDFILE)
129             c->aio_sendfile = (clcf->aio == NGX_HTTP_AIO_SENDFILE);
130         #endif
131     #endif
132
133     if (in && in->buf && ngx_buf_size(in->buf)) {
134         r->request_output = 1;
135     }
136 }
137
138 #if (NGX_HAVE_FILE_AIO)
139     ctx->aio = r->aio;
140 #endif
141
142     for ( ;; ) {
143         rc = ngx_output_chain(ctx, in);
144
145         if (ctx->in == NULL) {
146             r->buffered &= ~NGX_HTTP_COPY_BUFFERED;
147         } else {
148             r->buffered |= NGX_HTTP_COPY_BUFFERED;
149         }
150     }
151
152     ngx_log_debug3(NGX_LOG_DEBUG_HTTP, c->log, 0,
153         "http copy filter: %i \"%V?%V\"", rc, &r->uri, &r->args);
154
155     #if (NGX_HAVE_FILE_AIO && NGX_HAVE_AIO_SENDFILE)
156
157         if (c->busy_sendfile) {
158             ssize_t          n;
159             off_t            offset;
160             ngx_file_t       *file;
161             ngx_http_ephemeral_t *e;
162
163             if (r->aio) {
164                 c->busy_sendfile = NULL;
165                 return rc;
166             }
167
168             file = c->busy_sendfile->file;
169             offset = c->busy_sendfile->file_pos;
170
171             if (file->aio) {
172                 c->busy_count = (offset == file->aio->last_offset) ?
173                     c->busy_count + 1 : 0;
174                 file->aio->last_offset = offset;
175
176                 if (c->busy_count > 2) {
177                     ngx_log_error(NGX_LOG_ALERT, c->log, 0,
178                         "sendfile(%V) returned busy again",
179                         &file->name);
180                     c->aio_sendfile = 0;
181                 }
182             }
183         }
184     #endif

```

```

183         c->busy_sendfile = NULL;
184         e = (ngx\_http\_ephemeral\_t *) &r->uri_start;
185
186         n = ngx\_file\_aio\_read(file, &e->aio_preload, 1, offset, r->pool);
187
188         if (n > 0) {
189             in = NULL;
190             continue;
191         }
192
193         rc = n;
194
195         if (rc == NGX\_AGAIN) {
196             file->aio->data = r;
197             file->aio->handler = ngx\_http\_copy\_aio\_sendfile\_event\_handler;
198
199             r->main->blocked++;
200             r->aio = 1;
201         }
202     }
203 }
204 #endif
205
206     return rc;
207 }
208 }
209
210
211 #if (NGX\_HAVE\_FILE\_AIO)
212
213 static void
214 ngx\_http\_copy\_aio\_handler(ngx\_output\_chain\_ctx\_t *ctx, ngx\_file\_t *file)
215 {
216     ngx\_http\_request\_t *r;
217
218     r = ctx->filter_ctx;
219
220     file->aio->data = r;
221     file->aio->handler = ngx\_http\_copy\_aio\_event\_handler;
222
223     r->main->blocked++;
224     r->aio = 1;
225     ctx->aio = 1;
226 }
227
228
229 static void
230 ngx\_http\_copy\_aio\_event\_handler(ngx\_event\_t *ev)
231 {
232     ngx\_event\_aio\_t *aio;
233     ngx\_http\_request\_t *r;
234
235     aio = ev->data;
236     r = aio->data;
237
238     r->main->blocked--;
239     r->aio = 0;
240
241     r->connection->write->handler(r->connection->write);
242 }
243
244
245 #if (NGX\_HAVE\_AIO\_SENDFILE)
246
247 static void
248 ngx\_http\_copy\_aio\_sendfile\_event\_handler(ngx\_event\_t *ev)
249 {
250     ngx\_event\_aio\_t *aio;
251     ngx\_http\_request\_t *r;
252
253     aio = ev->data;
254     r = aio->data;
255
256     r->main->blocked--;
257     r->aio = 0;
258     ev->complete = 0;

```

```

259     r->connection->write->handler(r->connection->write);
260 }
261 }
262
263 #endif
264 #endif
265
266
267 static void *
268 ngx_http_copy_filter_create_conf(ngx_conf_t *cf)
269 {
270     ngx_http_copy_filter_conf_t *conf;
271
272     conf = ngx_palloc(cf->pool, sizeof(ngx_http_copy_filter_conf_t));
273     if (conf == NULL) {
274         return NULL;
275     }
276
277     conf->bufs.num = 0;
278
279     return conf;
280 }
281
282
283 static char *
284 ngx_http_copy_filter_merge_conf(ngx_conf_t *cf, void *parent, void *child)
285 {
286     ngx_http_copy_filter_conf_t *prev = parent;
287     ngx_http_copy_filter_conf_t *conf = child;
288
289     ngx_conf_merge_bufs_value(conf->bufs, prev->bufs, 1, 32768);
290
291     return NULL;
292 }
293
294
295 static ngx_int_t
296 ngx_http_copy_filter_init(ngx_conf_t *cf)
297 {
298     ngx_http_next_body_filter = ngx_http_top_body_filter;
299     ngx_http_top_body_filter = ngx_http_copy_filter;
300
301     return NGX_OK;
302 }
303

```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_access\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_access\\_commands](#)
- [ngx\\_http\\_access\\_module](#)
- [ngx\\_http\\_access\\_module\\_ctx](#)

## Data types defined

- [ngx\\_http\\_access\\_loc\\_conf\\_t](#)
- [ngx\\_http\\_access\\_rule6\\_t](#)
- [ngx\\_http\\_access\\_rule\\_t](#)
- [ngx\\_http\\_access\\_rule\\_un\\_t](#)

## Functions defined

- [ngx\\_http\\_access\\_create\\_loc\\_conf](#)
- [ngx\\_http\\_access\\_found](#)
- [ngx\\_http\\_access\\_handler](#)
- [ngx\\_http\\_access\\_inet](#)
- [ngx\\_http\\_access\\_inet6](#)
- [ngx\\_http\\_access\\_init](#)
- [ngx\\_http\\_access\\_merge\\_loc\\_conf](#)
- [ngx\\_http\\_access\\_rule](#)
- [ngx\\_http\\_access\\_unix](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 typedef struct {
14     in_addr_t      mask;
15     in_addr_t      addr;
16     ngx_uint_t     deny;      /* unsigned deny:1; */
17 } ngx_http_access_rule_t;
18
19 #if (NGX_HAVE_INET6)
20
21 typedef struct {
```



```

98     NULL,                                /* create server configuration */
99     NULL,                                /* merge server configuration */
100
101     ngx\_http\_access\_create\_loc\_conf,      /* create location configuration */
102     ngx\_http\_access\_merge\_loc\_conf       /* merge location configuration */
103 };
104
105
106 ngx\_module\_t ngx\_http\_access\_module = {
107     NGX\_MODULE\_V1,
108     &ngx\_http\_access\_module\_ctx,        /* module context */
109     ngx\_http\_access\_commands,         /* module directives */
110     NGX\_HTTP\_MODULE,                 /* module type */
111     NULL,                             /* init master */
112     NULL,                             /* init module */
113     NULL,                             /* init process */
114     NULL,                             /* init thread */
115     NULL,                             /* exit thread */
116     NULL,                             /* exit process */
117     NULL,                             /* exit master */
118     NGX\_MODULE\_V1\_PADDING
119 };
120
121
122 static ngx\_int\_t
123 ngx\_http\_access\_handler(ngx\_http\_request\_t *r)
124 {
125     struct sockaddr\_in          *sin;
126     ngx\_http\_access\_loc\_conf\_t *alcf;
127     #if (NGX_HAVE_INET6)
128     u\_char                      *p;
129     in\_addr\_t                  addr;
130     struct sockaddr\_in6       *sin6;
131     #endif
132
133     alcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_access\_module);
134
135     switch (r->connection->sockaddr->sa_family) {
136
137     case AF_INET:
138         if (alcf->rules) {
139             sin = (struct sockaddr\_in *) r->connection->sockaddr;
140             return ngx\_http\_access\_inet(r, alcf, sin->sin_addr.s_addr);
141         }
142         break;
143
144     #if (NGX_HAVE_INET6)
145
146     case AF_INET6:
147         sin6 = (struct sockaddr\_in6 *) r->connection->sockaddr;
148         p = sin6->sin6_addr.s6_addr;
149
150         if (alcf->rules && IN6\_IS\_ADDR\_V4MAPPED(&sin6->sin6_addr)) {
151             addr = p[12] << 24;
152             addr += p[13] << 16;
153             addr += p[14] << 8;
154             addr += p[15];
155             return ngx\_http\_access\_inet(r, alcf, htonl(addr));
156         }
157
158         if (alcf->rules6) {
159             return ngx\_http\_access\_inet6(r, alcf, p);
160         }
161
162         break;
163
164     #endif
165
166     #if (NGX_HAVE_UNIX_DOMAIN)
167
168     case AF_UNIX:
169         if (alcf->rules_un) {
170             return ngx\_http\_access\_unix(r, alcf);
171         }
172
173         break;

```

```

174
175 #endif
176 }
177
178     return NGX_DECLINED;
179 }
180
181
182 static ngx_int_t
183 ngx_http_access_inet(ngx_http_request_t *r, ngx_http_access_loc_conf_t *alcf,
184     in_addr_t addr)
185 {
186     ngx_uint_t          i;
187     ngx_http_access_rule_t *rule;
188
189     rule = alcf->rules->elts;
190     for (i = 0; i < alcf->rules->nelts; i++) {
191
192         ngx_log_debug3(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
193             "access: %08XD %08XD %08XD",
194             addr, rule[i].mask, rule[i].addr);
195
196         if ((addr & rule[i].mask) == rule[i].addr) {
197             return ngx_http_access_found(r, rule[i].deny);
198         }
199     }
200
201     return NGX_DECLINED;
202 }
203
204
205 #if (NGX_HAVE_INET6)
206
207 static ngx_int_t
208 ngx_http_access_inet6(ngx_http_request_t *r, ngx_http_access_loc_conf_t *alcf,
209     u_char *p)
210 {
211     ngx_uint_t          n;
212     ngx_uint_t          i;
213     ngx_http_access_rule6_t *rule6;
214
215     rule6 = alcf->rules6->elts;
216     for (i = 0; i < alcf->rules6->nelts; i++) {
217
218         #if (NGX_DEBUG)
219         {
220             size_t  cl, ml, al;
221             u_char ct[NGX_INET6_ADDRSTRLEN];
222             u_char mt[NGX_INET6_ADDRSTRLEN];
223             u_char at[NGX_INET6_ADDRSTRLEN];
224
225             cl = ngx_inet6_ntop(p, ct, NGX_INET6_ADDRSTRLEN);
226             ml = ngx_inet6_ntop(rule6[i].mask.s6_addr, mt, NGX_INET6_ADDRSTRLEN);
227             al = ngx_inet6_ntop(rule6[i].addr.s6_addr, at, NGX_INET6_ADDRSTRLEN);
228
229             ngx_log_debug6(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
230                 "access: %*s %*s %*s", cl, ct, ml, mt, al, at);
231         }
232         #endif
233
234         for (n = 0; n < 16; n++) {
235             if ((p[n] & rule6[i].mask.s6_addr[n]) != rule6[i].addr.s6_addr[n]) {
236                 goto next;
237             }
238         }
239
240         return ngx_http_access_found(r, rule6[i].deny);
241
242     next:
243         continue;
244     }
245
246     return NGX_DECLINED;
247 }
248
249 #endif

```



```

250
251
252 #if (NGX_HAVE_UNIX_DOMAIN)
253
254 static ngx_int_t
255 ngx_http_access_unix(ngx_http_request_t *r, ngx_http_access_loc_conf_t *alcf)
256 {
257     ngx_uint_t          i;
258     ngx_http_access_rule_un_t  *rule_un;
259
260     rule_un = alcf->rules_un->elts;
261     for (i = 0; i < alcf->rules_un->nelts; i++) {
262
263         /* TODO: check path */
264         if (1) {
265             return ngx_http_access_found(r, rule_un[i].deny);
266         }
267     }
268
269     return NGX_DECLINED;
270 }
271
272 #endif
273
274
275 static ngx_int_t
276 ngx_http_access_found(ngx_http_request_t *r, ngx_uint_t deny)
277 {
278     ngx_http_core_loc_conf_t  *clcf;
279
280     if (deny) {
281         clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
282
283         if (clcf->satisfy == NGX_HTTP_SATISFY_ALL) {
284             ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
285                 "access forbidden by rule");
286         }
287
288         return NGX_HTTP_FORBIDDEN;
289     }
290
291     return NGX_OK;
292 }
293
294
295 static char *
296 ngx_http_access_rule(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
297 {
298     ngx_http_access_loc_conf_t  *alcf = conf;
299
300     ngx_int_t          rc;
301     ngx_uint_t         all;
302     ngx_str_t          *value;
303     ngx_cidr_t         cidr;
304     ngx_http_access_rule_t  *rule;
305 #if (NGX_HAVE_INET6)
306     ngx_http_access_rule6_t  *rule6;
307 #endif
308 #if (NGX_HAVE_UNIX_DOMAIN)
309     ngx_http_access_rule_un_t  *rule_un;
310 #endif
311
312     ngx_memzero(&cidr, sizeof(ngx_cidr_t));
313
314     value = cf->args->elts;
315
316     all = (value[1].len == 3 && ngx_strcmp(value[1].data, "all") == 0);
317
318     if (!all) {
319
320 #if (NGX_HAVE_UNIX_DOMAIN)
321
322         if (value[1].len == 5 && ngx_strcmp(value[1].data, "unix:") == 0) {
323             cidr.family = AF_UNIX;
324             rc = NGX_OK;
325

```

```

326     } else {
327         rc = ngx_ptocidr(&value[1], &cidr);
328     }
329
330 #else
331 rc = ngx_ptocidr(&value[1], &cidr);
332 #endif
333
334 if (rc == NGX_ERROR) {
335     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
336         "invalid parameter \"%V\"", &value[1]);
337     return NGX_CONF_ERROR;
338 }
339
340 if (rc == NGX_DONE) {
341     ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
342         "low address bits of %V are meaningless", &value[1]);
343 }
344 }
345
346 if (cidr.family == AF_INET || all) {
347
348     if (alcf->rules == NULL) {
349         alcf->rules = ngx_array_create(cf->pool, 4,
350             sizeof(ngx_http_access_rule_t));
351         if (alcf->rules == NULL) {
352             return NGX_CONF_ERROR;
353         }
354     }
355
356     rule = ngx_array_push(alcf->rules);
357     if (rule == NULL) {
358         return NGX_CONF_ERROR;
359     }
360
361     rule->mask = cidr.u.in.mask;
362     rule->addr = cidr.u.in.addr;
363     rule->deny = (value[0].data[0] == 'd') ? 1 : 0;
364 }
365
366 #if (NGX_HAVE_INET6)
367 if (cidr.family == AF_INET6 || all) {
368
369     if (alcf->rules6 == NULL) {
370         alcf->rules6 = ngx_array_create(cf->pool, 4,
371             sizeof(ngx_http_access_rule6_t));
372         if (alcf->rules6 == NULL) {
373             return NGX_CONF_ERROR;
374         }
375     }
376
377     rule6 = ngx_array_push(alcf->rules6);
378     if (rule6 == NULL) {
379         return NGX_CONF_ERROR;
380     }
381
382     rule6->mask = cidr.u.in6.mask;
383     rule6->addr = cidr.u.in6.addr;
384     rule6->deny = (value[0].data[0] == 'd') ? 1 : 0;
385 }
386 #endif
387
388 #if (NGX_HAVE_UNIX_DOMAIN)
389 if (cidr.family == AF_UNIX || all) {
390
391     if (alcf->rules_un == NULL) {
392         alcf->rules_un = ngx_array_create(cf->pool, 1,
393             sizeof(ngx_http_access_rule_un_t));
394         if (alcf->rules_un == NULL) {
395             return NGX_CONF_ERROR;
396         }
397     }
398
399     rule_un = ngx_array_push(alcf->rules_un);
400     if (rule_un == NULL) {
401         return NGX_CONF_ERROR;

```

```

402     }
403
404     rule_un->deny = (value[0].data[0] == 'd') ? 1 : 0;
405 }
406 #endif
407
408     return NGX_CONF_OK;
409 }
410
411
412 static void *
413 ngx_http_access_create_loc_conf(ngx_conf_t *cf)
414 {
415     ngx_http_access_loc_conf_t *conf;
416
417     conf = ngx_palloc(cf->pool, sizeof(ngx_http_access_loc_conf_t));
418     if (conf == NULL) {
419         return NULL;
420     }
421
422     return conf;
423 }
424
425
426 static char *
427 ngx_http_access_merge_loc_conf(ngx_conf_t *cf, void *parent, void *child)
428 {
429     ngx_http_access_loc_conf_t *prev = parent;
430     ngx_http_access_loc_conf_t *conf = child;
431
432     if (conf->rules == NULL
433 #if (NGX_HAVE_INET6)
434         && conf->rules6 == NULL
435 #endif
436 #if (NGX_HAVE_UNIX_DOMAIN)
437         && conf->rules_un == NULL
438 #endif
439     ) {
440         conf->rules = prev->rules;
441 #if (NGX_HAVE_INET6)
442         conf->rules6 = prev->rules6;
443 #endif
444 #if (NGX_HAVE_UNIX_DOMAIN)
445         conf->rules_un = prev->rules_un;
446 #endif
447     }
448
449     return NGX_CONF_OK;
450 }
451
452
453 static ngx_int_t
454 ngx_http_access_init(ngx_conf_t *cf)
455 {
456     ngx_http_handler_pt *h;
457     ngx_http_core_main_conf_t *cmcf;
458
459     cmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_core_module);
460
461     h = ngx_array_push(&cmcf->phases[NGX_HTTP_ACCESS_PHASE].handlers);
462     if (h == NULL) {
463         return NGX_ERROR;
464     }
465
466     *h = ngx_http_access_handler;
467
468     return NGX_OK;
469 }

```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_addition\_filter\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_addition\\_commands](#)
- [ngx\\_http\\_addition\\_filter\\_module](#)
- [ngx\\_http\\_addition\\_filter\\_module\\_ctx](#)
- [ngx\\_http\\_next\\_body\\_filter](#)
- [ngx\\_http\\_next\\_header\\_filter](#)

## Data types defined

- [ngx\\_http\\_addition\\_conf\\_t](#)
- [ngx\\_http\\_addition\\_ctx\\_t](#)

## Functions defined

- [ngx\\_http\\_addition\\_body\\_filter](#)
- [ngx\\_http\\_addition\\_create\\_conf](#)
- [ngx\\_http\\_addition\\_filter\\_init](#)
- [ngx\\_http\\_addition\\_header\\_filter](#)
- [ngx\\_http\\_addition\\_merge\\_conf](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 typedef struct {
14     ngx_str_t    before_body;
15     ngx_str_t    after_body;
16
17     ngx_hash_t   types;
18     ngx_array_t  *types_keys;
19 } ngx_http_addition_conf_t;
20
21
22 typedef struct {
23     ngx_uint_t   before_body_sent;
24 } ngx_http_addition_ctx_t;
25
26
27 static void *ngx_http_addition_create_conf(ngx_conf_t *cf);
28 static char *ngx_http_addition_merge_conf(ngx_conf_t *cf, void *parent,
29     void *child);
30 static ngx_int_t ngx_http_addition_filter_init(ngx_conf_t *cf);
```

```

31
32
33 static ngx_command_t  ngx_http_addition_commands[] = {
34
35     { ngx_string("add_before_body"),
36       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
37       ngx_conf_set_str_slot,
38       NGX_HTTP_LOC_CONF_OFFSET,
39       offsetof(ngx_http_addition_conf_t, before_body),
40       NULL },
41
42     { ngx_string("add_after_body"),
43       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
44       ngx_conf_set_str_slot,
45       NGX_HTTP_LOC_CONF_OFFSET,
46       offsetof(ngx_http_addition_conf_t, after_body),
47       NULL },
48
49     { ngx_string("addition_types"),
50       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_1MORE,
51       ngx_http_types_slot,
52       NGX_HTTP_LOC_CONF_OFFSET,
53       offsetof(ngx_http_addition_conf_t, types_keys),
54       &ngx_http_html_default_types[0] },
55
56     ngx_null_command
57 };
58
59
60 static ngx_http_module_t  ngx_http_addition_filter_module_ctx = {
61     NULL,                                     /* preconfiguration */
62     ngx_http_addition_filter_init,           /* postconfiguration */
63
64     NULL,                                     /* create main configuration */
65     NULL,                                     /* init main configuration */
66
67     NULL,                                     /* create server configuration */
68     NULL,                                     /* merge server configuration */
69
70     ngx_http_addition_create_conf,          /* create location configuration */
71     ngx_http_addition_merge_conf           /* merge location configuration */
72 };
73
74
75 ngx_module_t  ngx_http_addition_filter_module = {
76     NGX_MODULE_V1,
77     &ngx_http_addition_filter_module_ctx, /* module context */
78     ngx_http_addition_commands,          /* module directives */
79     NGX_HTTP_MODULE,                     /* module type */
80     NULL,                                  /* init master */
81     NULL,                                  /* init module */
82     NULL,                                  /* init process */
83     NULL,                                  /* init thread */
84     NULL,                                  /* exit thread */
85     NULL,                                  /* exit process */
86     NULL,                                  /* exit master */
87     NGX_MODULE_V1_PADDING
88 };
89
90
91 static ngx_http_output_header_filter_pt  ngx_http_next_header_filter;
92 static ngx_http_output_body_filter_pt   ngx_http_next_body_filter;
93
94
95 static ngx_int_t
96 ngx_http_addition_header_filter(ngx_http_request_t *r)
97 {
98     ngx_http_addition_ctx_t  *ctx;
99     ngx_http_addition_conf_t  *conf;
100
101     if (r->headers_out.status != NGX_HTTP_OK || r != r->main) {
102         return ngx_http_next_header_filter(r);
103     }
104
105     conf = ngx_http_get_module_loc_conf(r, ngx_http_addition_filter_module);
106

```

```

107     if (conf->before_body.len == 0 && conf->after_body.len == 0) {
108         return ngx_http_next_header_filter(r);
109     }
110
111     if (ngx_http_test_content_type(r, &conf->types) == NULL) {
112         return ngx_http_next_header_filter(r);
113     }
114
115     ctx = ngx_palloc(r->pool, sizeof(ngx_http_addition_ctx_t));
116     if (ctx == NULL) {
117         return NGX_ERROR;
118     }
119
120     ngx_http_set_ctx(r, ctx, ngx_http_addition_filter_module);
121
122     ngx_http_clear_content_length(r);
123     ngx_http_clear_accept_ranges(r);
124     ngx_http_weak_etag(r);
125
126     return ngx_http_next_header_filter(r);
127 }
128
129
130 static ngx_int_t
131 ngx_http_addition_body_filter(ngx_http_request_t *r, ngx_chain_t *in)
132 {
133     ngx_int_t          rc;
134     ngx_uint_t        last;
135     ngx_chain_t       *cl;
136     ngx_http_request_t *sr;
137     ngx_http_addition_ctx_t *ctx;
138     ngx_http_addition_conf_t *conf;
139
140     if (in == NULL || r->header_only) {
141         return ngx_http_next_body_filter(r, in);
142     }
143
144     ctx = ngx_http_get_module_ctx(r, ngx_http_addition_filter_module);
145
146     if (ctx == NULL) {
147         return ngx_http_next_body_filter(r, in);
148     }
149
150     conf = ngx_http_get_module_loc_conf(r, ngx_http_addition_filter_module);
151
152     if (!ctx->before_body_sent) {
153         ctx->before_body_sent = 1;
154
155         if (conf->before_body.len) {
156             if (ngx_http_subrequest(r, &conf->before_body, NULL, &sr, NULL, 0)
157                 != NGX_OK)
158             {
159                 return NGX_ERROR;
160             }
161         }
162     }
163
164     if (conf->after_body.len == 0) {
165         ngx_http_set_ctx(r, NULL, ngx_http_addition_filter_module);
166         return ngx_http_next_body_filter(r, in);
167     }
168
169     last = 0;
170
171     for (cl = in; cl; cl = cl->next) {
172         if (cl->buf->last_buf) {
173             cl->buf->last_buf = 0;
174             cl->buf->sync = 1;
175             last = 1;
176         }
177     }
178
179     rc = ngx_http_next_body_filter(r, in);
180
181     if (rc == NGX_ERROR || !last || conf->after_body.len == 0) {
182         return rc;

```

```

183     }
184
185     if (ngx_http_subrequest(r, &conf->after_body, NULL, &sr, NULL, 0)
186         != NGX_OK)
187     {
188         return NGX_ERROR;
189     }
190
191     ngx_http_set_ctx(r, NULL, ngx_http_addition_filter_module);
192
193     return ngx_http_send_special(r, NGX_HTTP_LAST);
194 }
195
196
197 static ngx_int_t
198 ngx_http_addition_filter_init(ngx_conf_t *cf)
199 {
200     ngx_http_next_header_filter = ngx_http_top_header_filter;
201     ngx_http_top_header_filter = ngx_http_addition_header_filter;
202
203     ngx_http_next_body_filter = ngx_http_top_body_filter;
204     ngx_http_top_body_filter = ngx_http_addition_body_filter;
205
206     return NGX_OK;
207 }
208
209
210 static void *
211 ngx_http_addition_create_conf(ngx_conf_t *cf)
212 {
213     ngx_http_addition_conf_t *conf;
214
215     conf = ngx_palloc(cf->pool, sizeof(ngx_http_addition_conf_t));
216     if (conf == NULL) {
217         return NULL;
218     }
219
220     /*
221      * set by ngx_palloc():
222      *
223      *     conf->before_body = { 0, NULL };
224      *     conf->after_body = { 0, NULL };
225      *     conf->types = { NULL };
226      *     conf->types_keys = NULL;
227      */
228
229     return conf;
230 }
231
232
233 static char *
234 ngx_http_addition_merge_conf(ngx_conf_t *cf, void *parent, void *child)
235 {
236     ngx_http_addition_conf_t *prev = parent;
237     ngx_http_addition_conf_t *conf = child;
238
239     ngx_conf_merge_str_value(conf->before_body, prev->before_body, "");
240     ngx_conf_merge_str_value(conf->after_body, prev->after_body, "");
241
242     if (ngx_http_merge_types(cf, &conf->types_keys, &conf->types,
243         &prev->types_keys, &prev->types,
244         ngx_http_html_default_types)
245         != NGX_OK)
246     {
247         return NGX_CONF_ERROR;
248     }
249
250     return NGX_CONF_OK;
251 }

```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_auth\_basic\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_auth\\_basic\\_commands](#)
- [ngx\\_http\\_auth\\_basic\\_module](#)
- [ngx\\_http\\_auth\\_basic\\_module\\_ctx](#)

## Data types defined

- [ngx\\_http\\_auth\\_basic\\_ctx\\_t](#)
- [ngx\\_http\\_auth\\_basic\\_loc\\_conf\\_t](#)

## Functions defined

- [ngx\\_http\\_auth\\_basic\\_close](#)
- [ngx\\_http\\_auth\\_basic\\_create\\_loc\\_conf](#)
- [ngx\\_http\\_auth\\_basic\\_crypt\\_handler](#)
- [ngx\\_http\\_auth\\_basic\\_handler](#)
- [ngx\\_http\\_auth\\_basic\\_init](#)
- [ngx\\_http\\_auth\\_basic\\_merge\\_loc\\_conf](#)
- [ngx\\_http\\_auth\\_basic\\_set\\_realm](#)
- [ngx\\_http\\_auth\\_basic\\_user\\_file](#)

## Macros defined

- [NGX\\_HTTP\\_AUTH\\_BUF\\_SIZE](#)

## Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_http.h>
11 #include <ngx_crypt.h>
12
13
14 #define NGX_HTTP_AUTH_BUF_SIZE 2048
15
16
17 typedef struct {
18     ngx_str_t          passwd;
19 } ngx_http_auth_basic_ctx_t;
20
21
22 typedef struct {
```



```

23     ngx_http_complex_value_t *realm;
24     ngx_http_complex_value_t user_file;
25 } ngx_http_auth_basic_loc_conf_t;
26
27
28 static ngx_int_t ngx_http_auth_basic_handler(ngx_http_request_t *r);
29 static ngx_int_t ngx_http_auth_basic_crypt_handler(ngx_http_request_t *r,
30     ngx_http_auth_basic_ctx_t *ctx, ngx_str_t *passwd, ngx_str_t *realm);
31 static ngx_int_t ngx_http_auth_basic_set_realm(ngx_http_request_t *r,
32     ngx_str_t *realm);
33 static void ngx_http_auth_basic_close(ngx_file_t *file);
34 static void *ngx_http_auth_basic_create_loc_conf(ngx_conf_t *cf);
35 static char *ngx_http_auth_basic_merge_loc_conf(ngx_conf_t *cf,
36     void *parent, void *child);
37 static ngx_int_t ngx_http_auth_basic_init(ngx_conf_t *cf);
38 static char *ngx_http_auth_basic_user_file(ngx_conf_t *cf, ngx_command_t *cmd,
39     void *conf);
40
41
42 static ngx_command_t ngx_http_auth_basic_commands[] = {
43
44     { ngx_string("auth_basic"),
45       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_HTTP_LMT_CONF
46         |NGX_CONF_TAKE1,
47       ngx_http_set_complex_value_slot,
48       NGX_HTTP_LOC_CONF_OFFSET,
49       offsetof(ngx_http_auth_basic_loc_conf_t, realm),
50       NULL },
51
52     { ngx_string("auth_basic_user_file"),
53       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_HTTP_LMT_CONF
54         |NGX_CONF_TAKE1,
55       ngx_http_auth_basic_user_file,
56       NGX_HTTP_LOC_CONF_OFFSET,
57       offsetof(ngx_http_auth_basic_loc_conf_t, user_file),
58       NULL },
59
60     ngx_null_command
61 };
62
63
64 static ngx_http_module_t ngx_http_auth_basic_module_ctx = {
65     NULL, /* preconfiguration */
66     ngx_http_auth_basic_init, /* postconfiguration */
67
68     NULL, /* create main configuration */
69     NULL, /* init main configuration */
70
71     NULL, /* create server configuration */
72     NULL, /* merge server configuration */
73
74     ngx_http_auth_basic_create_loc_conf, /* create location configuration */
75     ngx_http_auth_basic_merge_loc_conf /* merge location configuration */
76 };
77
78
79 ngx_module_t ngx_http_auth_basic_module = {
80     NGX_MODULE_V1,
81     &ngx_http_auth_basic_module_ctx, /* module context */
82     ngx_http_auth_basic_commands, /* module directives */
83     NGX_HTTP_MODULE, /* module type */
84     NULL, /* init master */
85     NULL, /* init module */
86     NULL, /* init process */
87     NULL, /* init thread */
88     NULL, /* exit thread */
89     NULL, /* exit process */
90     NULL, /* exit master */
91     NGX_MODULE_V1_PADDING
92 };
93
94
95 static ngx_int_t
96 ngx_http_auth_basic_handler(ngx_http_request_t *r)
97 {
98     off_t
99         offset;

```

```

99     ssize_t                n;
100     ngx_fd_t              fd;
101     ngx_int_t             rc;
102     ngx_err_t             err;
103     ngx_str_t             pwd, realm, user_file;
104     ngx_uint_t            i, level, login, left, passwd;
105     ngx_file_t            file;
106     ngx_http_auth_basic_ctx_t *ctx;
107     ngx_http_auth_basic_loc_conf_t *alcf;
108     u_char                buf[NGX_HTTP_AUTH_BUF_SIZE];
109     enum {
110         sw_login,
111         sw_passwd,
112         sw_skip
113     } state;
114
115     alcf = ngx_http_get_module_loc_conf(r, ngx_http_auth_basic_module);
116
117     if (alcf->realm == NULL || alcf->user_file.value.data == NULL) {
118         return NGX_DECLINED;
119     }
120
121     if (ngx_http_complex_value(r, alcf->realm, &realm) != NGX_OK) {
122         return NGX_ERROR;
123     }
124
125     if (realm.len == 3 && ngx_strncmp(realm.data, "off", 3) == 0) {
126         return NGX_DECLINED;
127     }
128
129     ctx = ngx_http_get_module_ctx(r, ngx_http_auth_basic_module);
130
131     if (ctx) {
132         return ngx_http_auth_basic_crypt_handler(r, ctx, &ctx->passwd,
133                                                 &realm);
134     }
135
136     rc = ngx_http_auth_basic_user(r);
137
138     if (rc == NGX_DECLINED) {
139         ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
140                     "no user/password was provided for basic authentication");
141
142         return ngx_http_auth_basic_set_realm(r, &realm);
143     }
144
145     if (rc == NGX_ERROR) {
146         return NGX_HTTP_INTERNAL_SERVER_ERROR;
147     }
148
149     if (ngx_http_complex_value(r, &alcf->user_file, &user_file) != NGX_OK) {
150         return NGX_ERROR;
151     }
152
153     fd = ngx_open_file(user_file.data, NGX_FILE_RDONLY, NGX_FILE_OPEN, 0);
154
155     if (fd == NGX_INVALID_FILE) {
156         err = ngx_errno;
157
158         if (err == NGX_ENOENT) {
159             level = NGX_LOG_ERR;
160             rc = NGX_HTTP_FORBIDDEN;
161
162         } else {
163             level = NGX_LOG_CRIT;
164             rc = NGX_HTTP_INTERNAL_SERVER_ERROR;
165         }
166     }
167
168     ngx_log_error(level, r->connection->log, err,
169                 ngx_open_file_n " \"%s\" failed", user_file.data);
170
171     return rc;
172 }
173
174 ngx_memzero(&file, sizeof(ngx_file_t));

```



```

251         break;
252     }
253 }
254
255
256 if (state == sw_passwd) {
257     left = left + n - passwd;
258     ngx_memmove(buf, &buf[passwd], left);
259     passwd = 0;
260
261 } else {
262     left = 0;
263 }
264
265 offset += n;
266 }
267
268 ngx_http_auth_basic_close(&file);
269
270 if (state == sw_passwd) {
271     pwd.len = i - passwd;
272     pwd.data = ngx_pnalloc(r->pool, pwd.len + 1);
273     if (pwd.data == NULL) {
274         return NGX_HTTP_INTERNAL_SERVER_ERROR;
275     }
276
277     ngx_cpymem(pwd.data, &buf[passwd], pwd.len + 1);
278
279     return ngx_http_auth_basic_crypt_handler(r, NULL, &pwd, &realm);
280 }
281
282 ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
283             "user \"%V\" was not found in \"%V\"",
284             &r->headers_in.user, &user_file);
285
286 return ngx_http_auth_basic_set_realm(r, &realm);
287 }
288
289
290 static ngx_int_t
291 ngx_http_auth_basic_crypt_handler(ngx_http_request_t *r,
292     ngx_http_auth_basic_ctx_t *ctx, ngx_str_t *passwd, ngx_str_t *realm)
293 {
294     ngx_int_t rc;
295     u_char *encrypted;
296
297     rc = ngx_crypt(r->pool, r->headers_in.passwd.data, passwd->data,
298                 &encrypted);
299
300     ngx_log_debug3(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
301                 "rc: %d user: \"%V\" salt: \"%s\"",
302                 rc, &r->headers_in.user, passwd->data);
303
304     if (rc == NGX_OK) {
305         if (ngx_strcmp(encrypted, passwd->data) == 0) {
306             return NGX_OK;
307         }
308
309         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
310                 "encrypted: \"%s\"", encrypted);
311
312         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
313                 "user \"%V\": password mismatch",
314                 &r->headers_in.user);
315
316         return ngx_http_auth_basic_set_realm(r, realm);
317     }
318
319     if (rc == NGX_ERROR) {
320         return NGX_HTTP_INTERNAL_SERVER_ERROR;
321     }
322
323     /* rc == NGX_AGAIN */
324
325     if (ctx == NULL) {
326         ctx = ngx_palloc(r->pool, sizeof(ngx_http_auth_basic_ctx_t));

```

```

327     if (ctx == NULL) {
328         return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
329     }
330
331     ngx\_http\_set\_ctx(r, ctx, ngx\_http\_auth\_basic\_module);
332
333     ctx->passwd.len = passwd->len;
334     passwd->len++;
335
336     ctx->passwd.data = ngx\_pstrdup(r->pool, passwd);
337     if (ctx->passwd.data == NULL) {
338         return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
339     }
340
341 }
342
343 /* TODO: add mutex event */
344
345 return rc;
346 }
347
348
349 static ngx\_int\_t
350 ngx\_http\_auth\_basic\_set\_realm(ngx\_http\_request\_t *r, ngx\_str\_t *realm)
351 {
352     size\_t len;
353     u\_char *basic, *p;
354
355     r->headers_out.www_authenticate = ngx\_list\_push(&r->headers_out.headers);
356     if (r->headers_out.www_authenticate == NULL) {
357         return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
358     }
359
360     len = sizeof("Basic realm=\\\"\\") - 1 + realm->len;
361
362     basic = ngx\_pnalloc(r->pool, len);
363     if (basic == NULL) {
364         return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
365     }
366
367     p = ngx\_cpymem(basic, "Basic realm=\\\"", sizeof("Basic realm=\\\"", - 1);
368     p = ngx\_cpymem(p, realm->data, realm->len);
369     *p = '\\';
370
371     r->headers_out.www_authenticate->hash = 1;
372     ngx\_str\_set(&r->headers_out.www_authenticate->key, "WWW-Authenticate");
373     r->headers_out.www_authenticate->value.data = basic;
374     r->headers_out.www_authenticate->value.len = len;
375
376     return NGX\_HTTP\_UNAUTHORIZED;
377 }
378
379 static void
380 ngx\_http\_auth\_basic\_close(ngx\_file\_t *file)
381 {
382     if (ngx\_close\_file(file->fd) == NGX\_FILE\_ERROR) {
383         ngx\_log\_error(NGX\_LOG\_ALERT, file->log, ngx\_errno,
384             ngx\_close\_file\_n " \\\"%s\\\" failed", file->name.data);
385     }
386 }
387
388
389 static void *
390 ngx\_http\_auth\_basic\_create\_loc\_conf(ngx\_conf\_t *cf)
391 {
392     ngx\_http\_auth\_basic\_loc\_conf\_t *conf;
393
394     conf = ngx\_pcalloc(cf->pool, sizeof(ngx\_http\_auth\_basic\_loc\_conf\_t));
395     if (conf == NULL) {
396         return NULL;
397     }
398
399     return conf;
400 }
401
402

```

```

403 static char *
404 ngx_http_auth_basic_merge_loc_conf(ngx_conf_t *cf, void *parent, void *child)
405 {
406     ngx_http_auth_basic_loc_conf_t *prev = parent;
407     ngx_http_auth_basic_loc_conf_t *conf = child;
408
409     if (conf->realm == NULL) {
410         conf->realm = prev->realm;
411     }
412
413     if (conf->user_file.value.data == NULL) {
414         conf->user_file = prev->user_file;
415     }
416
417     return NGX_CONF_OK;
418 }
419
420
421 static ngx_int_t
422 ngx_http_auth_basic_init(ngx_conf_t *cf)
423 {
424     ngx_http_handler_pt *h;
425     ngx_http_core_main_conf_t *cmcf;
426
427     cmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_core_module);
428
429     h = ngx_array_push(&cmcf->phases[NGX_HTTP_ACCESS_PHASE].handlers);
430     if (h == NULL) {
431         return NGX_ERROR;
432     }
433
434     *h = ngx_http_auth_basic_handler;
435
436     return NGX_OK;
437 }
438
439
440 static char *
441 ngx_http_auth_basic_user_file(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
442 {
443     ngx_http_auth_basic_loc_conf_t *alcf = conf;
444
445     ngx_str_t value;
446     ngx_http_compile_complex_value_t ccv;
447
448     if (alcf->user_file.value.data) {
449         return "is duplicate";
450     }
451
452     value = cf->args->elts;
453
454     ngx_memzero(&ccv, sizeof(ngx_http_compile_complex_value_t));
455
456     ccv.cf = cf;
457     ccv.value = &value[1];
458     ccv.complex_value = &alcf->user_file;
459     ccv.zero = 1;
460     ccv.conf_prefix = 1;
461
462     if (ngx_http_compile_complex_value(&ccv) != NGX_OK) {
463         return NGX_CONF_ERROR;
464     }
465
466     return NGX_CONF_OK;
467 }

```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_crypt.c - nginx-1.7.10

### Functions defined

- [ngx\\_crypt](#)
- [ngx\\_crypt\\_apr1](#)
- [ngx\\_crypt\\_plain](#)
- [ngx\\_crypt\\_sha](#)
- [ngx\\_crypt\\_ssha](#)
- [ngx\\_crypt\\_to64](#)

### Source code

```
1
2  /*
3  * Copyright (C) Maxim Dounin
4  */
5
6
7  #include <ngx_config.h>
8  #include <ngx_core.h>
9  #include <ngx_crypt.h>
10 #include <ngx_md5.h>
11 #if (NGX_HAVE_SHA1)
12 #include <ngx_sha1.h>
13 #endif
14
15
16 #if (NGX_CRYPT)
17
18 static ngx_int_t ngx_crypt_apr1(ngx_pool_t *pool, u_char *key, u_char *salt,
19     u_char **encrypted);
20 static ngx_int_t ngx_crypt_plain(ngx_pool_t *pool, u_char *key, u_char *salt,
21     u_char **encrypted);
22
23 #if (NGX_HAVE_SHA1)
24
25 static ngx_int_t ngx_crypt_ssha(ngx_pool_t *pool, u_char *key, u_char *salt,
26     u_char **encrypted);
27 static ngx_int_t ngx_crypt_sha(ngx_pool_t *pool, u_char *key, u_char *salt,
28     u_char **encrypted);
29
30 #endif
31
32
33 static u_char *ngx_crypt_to64(u_char *p, uint32_t v, size_t n);
34
35
36 ngx_int_t
37 ngx_crypt(ngx_pool_t *pool, u_char *key, u_char *salt, u_char **encrypted)
38 {
39     if (ngx_strncmp(salt, "$apr1$", sizeof("$apr1$") - 1) == 0) {
40         return ngx_crypt_apr1(pool, key, salt, encrypted);
41     }
42     else if (ngx_strncmp(salt, "{PLAIN}", sizeof("{PLAIN}") - 1) == 0) {
43         return ngx_crypt_plain(pool, key, salt, encrypted);
44     }
45     #if (NGX_HAVE_SHA1)
46     else if (ngx_strncmp(salt, "{SSHA}", sizeof("{SSHA}") - 1) == 0) {
47         return ngx_crypt_ssha(pool, key, salt, encrypted);
48     }
49     else if (ngx_strncmp(salt, "{SHA}", sizeof("{SHA}") - 1) == 0) {
50         return ngx_crypt_sha(pool, key, salt, encrypted);
51     }
52 #endif
53 }
```

```

52     }
53
54     /* fallback to libc crypt() */
55
56     return ngx_libc_crypt(pool, key, salt, encrypted);
57 }
58
59
60 static ngx_int_t
61 ngx_crypt_apr1(ngx_pool_t *pool, u_char *key, u_char *salt, u_char **encrypted)
62 {
63     ngx_int_t      n;
64     ngx_uint_t     i;
65     u_char         *p, *last, final[16];
66     size_t         saltlen, keylen;
67     ngx_md5_t      md5, ctx1;
68
69     /* Apache's apr1 crypt is Poul-Henning Kamp's md5 crypt with $apr1$ magic */
70
71     keylen = ngx_strlen(key);
72
73     /* true salt: no magic, max 8 chars, stop at first $ */
74
75     salt += sizeof("$apr1$") - 1;
76     last = salt + 8;
77     for (p = salt; *p && *p != '$' && p < last; p++) { /* void */ }
78     saltlen = p - salt;
79
80     /* hash key and salt */
81
82     ngx_md5_init(&md5);
83     ngx_md5_update(&md5, key, keylen);
84     ngx_md5_update(&md5, (u_char *) "$apr1$", sizeof("$apr1$") - 1);
85     ngx_md5_update(&md5, salt, saltlen);
86
87     ngx_md5_init(&ctx1);
88     ngx_md5_update(&ctx1, key, keylen);
89     ngx_md5_update(&ctx1, salt, saltlen);
90     ngx_md5_update(&ctx1, key, keylen);
91     ngx_md5_final(final, &ctx1);
92
93     for (n = keylen; n > 0; n -= 16) {
94         ngx_md5_update(&md5, final, n > 16 ? 16 : n);
95     }
96
97     ngx_memzero(final, sizeof(final));
98
99     for (i = keylen; i; i >>= 1) {
100         if (i & 1) {
101             ngx_md5_update(&md5, final, 1);
102
103         } else {
104             ngx_md5_update(&md5, key, 1);
105         }
106     }
107
108     ngx_md5_final(final, &md5);
109
110     for (i = 0; i < 1000; i++) {
111         ngx_md5_init(&ctx1);
112
113         if (i & 1) {
114             ngx_md5_update(&ctx1, key, keylen);
115
116         } else {
117             ngx_md5_update(&ctx1, final, 16);
118         }
119
120         if (i % 3) {
121             ngx_md5_update(&ctx1, salt, saltlen);
122         }
123
124         if (i % 7) {
125             ngx_md5_update(&ctx1, key, keylen);
126         }
127

```



```

128     if (i & 1) {
129         ngx_md5_update(&ctx1, final, 16);
130
131     } else {
132         ngx_md5_update(&ctx1, key, keylen);
133     }
134
135     ngx_md5_final(final, &ctx1);
136 }
137
138 /* output */
139
140 *encrypted = ngx_pnalloc(pool, sizeof("$apr1$") - 1 + saltlen + 1 + 22 + 1);
141 if (*encrypted == NULL) {
142     return NGX_ERROR;
143 }
144
145 p = ngx_cpymem(*encrypted, "$apr1$", sizeof("$apr1$") - 1);
146 p = ngx_copy(p, salt, saltlen);
147 *p++ = '$';
148
149 p = ngx_crypt_to64(p, (final[ 0]<<16) | (final[ 6]<<8) | final[12], 4);
150 p = ngx_crypt_to64(p, (final[ 1]<<16) | (final[ 7]<<8) | final[13], 4);
151 p = ngx_crypt_to64(p, (final[ 2]<<16) | (final[ 8]<<8) | final[14], 4);
152 p = ngx_crypt_to64(p, (final[ 3]<<16) | (final[ 9]<<8) | final[15], 4);
153 p = ngx_crypt_to64(p, (final[ 4]<<16) | (final[10]<<8) | final[ 5], 4);
154 p = ngx_crypt_to64(p, final[11], 2);
155 *p = '\0';
156
157 return NGX_OK;
158 }
159
160
161 static u_char *
162 ngx_crypt_to64(u_char *p, uint32_t v, size_t n)
163 {
164     static u_char itoa64[] =
165         ".0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
166
167     while (n--) {
168         *p++ = itoa64[v & 0x3f];
169         v >>= 6;
170     }
171
172     return p;
173 }
174
175
176 static ngx_int_t
177 ngx_crypt_plain(ngx_pool_t *pool, u_char *key, u_char *salt, u_char **encrypted)
178 {
179     size_t len;
180     u_char *p;
181
182     len = ngx_strlen(key);
183
184     *encrypted = ngx_pnalloc(pool, sizeof("{PLAIN}") - 1 + len + 1);
185     if (*encrypted == NULL) {
186         return NGX_ERROR;
187     }
188
189     p = ngx_cpymem(*encrypted, "{PLAIN}", sizeof("{PLAIN}") - 1);
190     ngx_memcpy(p, key, len + 1);
191
192     return NGX_OK;
193 }
194
195
196 #if (NGX_HAVE_SHA1)
197
198 static ngx_int_t
199 ngx_crypt_ssha(ngx_pool_t *pool, u_char *key, u_char *salt, u_char **encrypted)
200 {
201     size_t len;
202     ngx_int_t rc;
203     ngx_str_t encoded, decoded;

```

```

204     ngx_sha1_t    sha1;
205
206     /* "{SSHA}" base64(SHA1(key salt) salt) */
207
208     /* decode base64 salt to find out true salt */
209
210     encoded.data = salt + sizeof("{SSHA}") - 1;
211     encoded.len = ngx_strlen(encoded.data);
212
213     len = ngx_max(ngx_base64_decoded_length(encoded.len), 20);
214
215     decoded.data = ngx_pnalloc(pool, len);
216     if (decoded.data == NULL) {
217         return NGX_ERROR;
218     }
219
220     rc = ngx_decode_base64(&decoded, &encoded);
221
222     if (rc != NGX_OK || decoded.len < 20) {
223         decoded.len = 20;
224     }
225
226     /* update SHA1 from key and salt */
227
228     ngx_sha1_init(&sha1);
229     ngx_sha1_update(&sha1, key, ngx_strlen(key));
230     ngx_sha1_update(&sha1, decoded.data + 20, decoded.len - 20);
231     ngx_sha1_final(decoded.data, &sha1);
232
233     /* encode it back to base64 */
234
235     len = sizeof("{SSHA}") - 1 + ngx_base64_encoded_length(decoded.len) + 1;
236
237     *encrypted = ngx_pnalloc(pool, len);
238     if (*encrypted == NULL) {
239         return NGX_ERROR;
240     }
241
242     encoded.data = ngx_cpymem(*encrypted, "{SSHA}", sizeof("{SSHA}") - 1);
243     ngx_encode_base64(&encoded, &decoded);
244     encoded.data[encoded.len] = '\0';
245
246     return NGX_OK;
247 }
248
249
250 static ngx_int_t
251 ngx_crypt_sha(ngx_pool_t *pool, u_char *key, u_char *salt, u_char **encrypted)
252 {
253     size_t    len;
254     ngx_str_t  encoded, decoded;
255     ngx_sha1_t sha1;
256     u_char    digest[20];
257
258     /* "{SHA}" base64(SHA1(key)) */
259
260     decoded.len = sizeof(digest);
261     decoded.data = digest;
262
263     ngx_sha1_init(&sha1);
264     ngx_sha1_update(&sha1, key, ngx_strlen(key));
265     ngx_sha1_final(digest, &sha1);
266
267     len = sizeof("{SHA}") - 1 + ngx_base64_encoded_length(decoded.len) + 1;
268
269     *encrypted = ngx_pnalloc(pool, len);
270     if (*encrypted == NULL) {
271         return NGX_ERROR;
272     }
273
274     encoded.data = ngx_cpymem(*encrypted, "{SHA}", sizeof("{SHA}") - 1);
275     ngx_encode_base64(&encoded, &decoded);
276     encoded.data[encoded.len] = '\0';
277
278     return NGX_OK;
279 }

```

```
280
281 #endif /* NGX_HAVE_SHA1 */
282
283 #endif /* NGX_CRYPT */
```

[One Level Up](#)

[Top Level](#)

# src/core/nginx\_sha1.h - nginx-1.7.10

## Data types defined

- [ngx\\_sha1\\_t](#)

## Macros defined

- [\\_NGX\\_SHA1\\_H\\_INCLUDED](#)
- [ngx\\_sha1\\_final](#)
- [ngx\\_sha1\\_init](#)
- [ngx\\_sha1\\_update](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_SHA1\_H\_INCLUDED
9 #define \_NGX\_SHA1\_H\_INCLUDED
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 #if (NGX_HAVE_OPENSSL_SHA1_H)
17 #include <openssl/sha.h>
18 #else
19 #include <sha.h>
20 #endif
21
22
23 typedef SHA_CTX ngx\_sha1\_t;
24
25
26 #define ngx\_sha1\_init    SHA1_Init
27 #define ngx\_sha1\_update  SHA1_Update
28 #define ngx\_sha1\_final  SHA1_Final
29
30
31 #endif /* \_NGX\_SHA1\_H\_INCLUDED */
```

# src/http/modules/nginx\_http\_auth\_request\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_auth\\_request\\_commands](#)
- [ngx\\_http\\_auth\\_request\\_module](#)
- [ngx\\_http\\_auth\\_request\\_module\\_ctx](#)

## Data types defined

- [ngx\\_http\\_auth\\_request\\_conf\\_t](#)
- [ngx\\_http\\_auth\\_request\\_ctx\\_t](#)
- [ngx\\_http\\_auth\\_request\\_variable\\_t](#)

## Functions defined

- [ngx\\_http\\_auth\\_request](#)
- [ngx\\_http\\_auth\\_request\\_create\\_conf](#)
- [ngx\\_http\\_auth\\_request\\_done](#)
- [ngx\\_http\\_auth\\_request\\_handler](#)
- [ngx\\_http\\_auth\\_request\\_init](#)
- [ngx\\_http\\_auth\\_request\\_merge\\_conf](#)
- [ngx\\_http\\_auth\\_request\\_set](#)
- [ngx\\_http\\_auth\\_request\\_set\\_variables](#)
- [ngx\\_http\\_auth\\_request\\_variable](#)

## Source code

```
1
2  /*
3  * Copyright (C) Maxim Dounin
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 typedef struct {
14     ngx_str_t          uri;
15     ngx_array_t        *vars;
16 } ngx_http_auth_request_conf_t;
17
18
19 typedef struct {
20     ngx_uint_t         done;
21     ngx_uint_t         status;
22     ngx_http_request_t *subrequest;
23 } ngx_http_auth_request_ctx_t;
24
```

```

25
26 typedef struct {
27     ngx_int_t          index;
28     ngx_http_complex_value_t  value;
29     ngx_http_set_variable_pt  set_handler;
30 } ngx_http_auth_request_variable_t;
31
32
33 static ngx_int_t ngx_http_auth_request_handler(ngx_http_request_t *r);
34 static ngx_int_t ngx_http_auth_request_done(ngx_http_request_t *r,
35     void *data, ngx_int_t rc);
36 static ngx_int_t ngx_http_auth_request_set_variables(ngx_http_request_t *r,
37     ngx_http_auth_request_ctx_t *ctx, ngx_http_auth_request_ctx_t *ctx);
38 static ngx_int_t ngx_http_auth_request_variable(ngx_http_request_t *r,
39     ngx_http_variable_value_t *v, uintptr_t data);
40 static void *ngx_http_auth_request_create_conf(ngx_conf_t *cf);
41 static char *ngx_http_auth_request_merge_conf(ngx_conf_t *cf,
42     void *parent, void *child);
43 static ngx_int_t ngx_http_auth_request_init(ngx_conf_t *cf);
44 static char *ngx_http_auth_request(ngx_conf_t *cf, ngx_command_t *cmd,
45     void *conf);
46 static char *ngx_http_auth_request_set(ngx_conf_t *cf, ngx_command_t *cmd,
47     void *conf);
48
49
50 static ngx_command_t  ngx_http_auth_request_commands[] = {
51
52     { ngx_string("auth_request"),
53       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
54       ngx_http_auth_request,
55       NGX_HTTP_LOC_CONF_OFFSET,
56       0,
57       NULL },
58
59     { ngx_string("auth_request_set"),
60       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE2,
61       ngx_http_auth_request_set,
62       NGX_HTTP_LOC_CONF_OFFSET,
63       0,
64       NULL },
65
66     ngx_null_command
67 };
68
69
70 static ngx_http_module_t  ngx_http_auth_request_module_ctx = {
71     NULL,                          /* preconfiguration */
72     ngx_http_auth_request_init,     /* postconfiguration */
73
74     NULL,                            /* create main configuration */
75     NULL,                            /* init main configuration */
76
77     NULL,                            /* create server configuration */
78     NULL,                            /* merge server configuration */
79
80     ngx_http_auth_request_create_conf, /* create location configuration */
81     ngx_http_auth_request_merge_conf /* merge location configuration */
82 };
83
84
85 ngx_module_t  ngx_http_auth_request_module = {
86     NGX_MODULE_V1,
87     &ngx_http_auth_request_module_ctx, /* module context */
88     ngx_http_auth_request_commands,    /* module directives */
89     NGX_HTTP_MODULE,                  /* module type */
90     NULL,                              /* init master */
91     NULL,                              /* init module */
92     NULL,                              /* init process */
93     NULL,                              /* init thread */
94     NULL,                              /* exit thread */
95     NULL,                              /* exit process */
96     NULL,                              /* exit master */
97     NGX_MODULE_V1_PADDING
98 };
99
100

```

```

101 static ngx_int_t
102 ngx_http_auth_request_handler(ngx_http_request_t *r)
103 {
104     ngx_table_elt_t      *h, *ho;
105     ngx_http_request_t    *sr;
106     ngx_http_post_subrequest_t *ps;
107     ngx_http_auth_request_ctx_t *ctx;
108     ngx_http_auth_request_conf_t *arcf;
109
110     arcf = ngx_http_get_module_loc_conf(r, ngx_http_auth_request_module);
111
112     if (arcf->uri.len == 0) {
113         return NGX_DECLINED;
114     }
115
116     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
117                 "auth request handler");
118
119     ctx = ngx_http_get_module_ctx(r, ngx_http_auth_request_module);
120
121     if (ctx != NULL) {
122         if (!ctx->done) {
123             return NGX_AGAIN;
124         }
125
126         /*
127          * as soon as we are done - explicitly set variables to make
128          * sure they will be available after internal redirects
129          */
130
131         if (ngx_http_auth_request_set_variables(r, arcf, ctx) != NGX_OK) {
132             return NGX_ERROR;
133         }
134
135         /* return appropriate status */
136
137         if (ctx->status == NGX_HTTP_FORBIDDEN) {
138             return ctx->status;
139         }
140
141         if (ctx->status == NGX_HTTP_UNAUTHORIZED) {
142             sr = ctx->subrequest;
143
144             h = sr->headers_out.www_authenticate;
145
146             if (!h && sr->upstream) {
147                 h = sr->upstream->headers_in.www_authenticate;
148             }
149
150             if (h) {
151                 ho = ngx_list_push(&r->headers_out.headers);
152                 if (ho == NULL) {
153                     return NGX_ERROR;
154                 }
155
156                 *ho = *h;
157
158                 r->headers_out.www_authenticate = ho;
159             }
160
161             return ctx->status;
162         }
163
164         if (ctx->status >= NGX_HTTP_OK
165             && ctx->status < NGX_HTTP_SPECIAL_RESPONSE)
166         {
167             return NGX_OK;
168         }
169
170         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
171                     "auth request unexpected status: %d", ctx->status);
172
173         return NGX_HTTP_INTERNAL_SERVER_ERROR;
174     }
175
176     ctx = ngx_palloc(r->pool, sizeof(ngx_http_auth_request_ctx_t));

```

```

177     if (ctx == NULL) {
178         return NGX_ERROR;
179     }
180
181     ps = ngx_palloc(r->pool, sizeof(ngx_http_post_subrequest_t));
182     if (ps == NULL) {
183         return NGX_ERROR;
184     }
185
186     ps->handler = ngx_http_auth_request_done;
187     ps->data = ctx;
188
189     if (ngx_http_subrequest(r, &arcf->uri, NULL, &sr, ps,
190                             NGX_HTTP_SUBREQUEST_WAITED)
191         != NGX_OK)
192     {
193         return NGX_ERROR;
194     }
195
196     /*
197     * allocate fake request body to avoid attempts to read it and to make
198     * sure real body file (if already read) won't be closed by upstream
199     */
200
201     sr->request_body = ngx_pcalloc(r->pool, sizeof(ngx_http_request_body_t));
202     if (sr->request_body == NULL) {
203         return NGX_ERROR;
204     }
205
206     sr->header_only = 1;
207
208     ctx->subrequest = sr;
209
210     ngx_http_set_ctx(r, ctx, ngx_http_auth_request_module);
211
212     return NGX_AGAIN;
213 }
214
215
216 static ngx_int_t
217 ngx_http_auth_request_done(ngx_http_request_t *r, void *data, ngx_int_t rc)
218 {
219     ngx_http_auth_request_ctx_t *ctx = data;
220
221     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
222                  "auth request done s:%d", r->headers_out.status);
223
224     ctx->done = 1;
225     ctx->status = r->headers_out.status;
226
227     return rc;
228 }
229
230
231 static ngx_int_t
232 ngx_http_auth_request_set_variables(ngx_http_request_t *r,
233     ngx_http_auth_request_conf_t *arcf, ngx_http_auth_request_ctx_t *ctx)
234 {
235     ngx_str_t val;
236     ngx_http_variable_t *v;
237     ngx_http_variable_value_t *vv;
238     ngx_http_auth_request_variable_t *av, *last;
239     ngx_http_core_main_conf_t *cmcf;
240
241     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
242                  "auth request set variables");
243
244     if (arcf->vars == NULL) {
245         return NGX_OK;
246     }
247
248     cmcf = ngx_http_get_module_main_conf(r, ngx_http_core_module);
249     v = cmcf->variables.elts;
250
251     av = arcf->vars->elts;
252     last = av + arcf->vars->nelts;

```



```

253 while (av < last) {
254     /*
255      * explicitly set new value to make sure it will be available after
256      * internal redirects
257      */
258
259     vv = &r->variables[av->index];
260
261     if (ngx_http_complex_value(ctx->subrequest, &av->value, &val)
262         != NGX_OK)
263     {
264         return NGX_ERROR;
265     }
266
267     vv->valid = 1;
268     vv->not_found = 0;
269     vv->data = val.data;
270     vv->len = val.len;
271
272     if (av->set_handler) {
273         /*
274          * set_handler only available in cmcf->variables_keys, so we store
275          * it explicitly
276          */
277
278         av->set_handler(r, vv, v[av->index].data);
279     }
280
281     av++;
282 }
283
284 return NGX_OK;
285 }
286
287
288
289 static ngx_int_t
290 ngx_http_auth_request_variable(ngx_http_request_t *r,
291 ngx_http_variable_value_t *v, uintptr_t data)
292 {
293     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
294         "auth request variable");
295
296     v->not_found = 1;
297
298     return NGX_OK;
299 }
300
301
302 static void *
303 ngx_http_auth_request_create_conf(ngx_conf_t *cf)
304 {
305     ngx_http_auth_request_conf_t *conf;
306
307     conf = ngx_palloc(cf->pool, sizeof(ngx_http_auth_request_conf_t));
308     if (conf == NULL) {
309         return NULL;
310     }
311
312     /*
313      * set by ngx_palloc():
314      *
315      *     conf->uri = { 0, NULL };
316      */
317
318     conf->vars = NGX_CONF_UNSET_PTR;
319
320     return conf;
321 }
322
323
324 static char *
325 ngx_http_auth_request_merge_conf(ngx_conf_t *cf, void *parent, void *child)
326 {
327     ngx_http_auth_request_conf_t *prev = parent;
328     ngx_http_auth_request_conf_t *conf = child;

```

```

329     ngx_conf_merge_str_value(conf->uri, prev->uri, "");
330     ngx_conf_merge_ptr_value(conf->vars, prev->vars, NULL);
331
332
333     return NGX_CONF_OK;
334 }
335
336
337 static ngx_int_t
338 ngx_http_auth_request_init(ngx_conf_t *cf)
339 {
340     ngx_http_handler_pt *h;
341     ngx_http_core_main_conf_t *cmcf;
342
343     cmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_core_module);
344
345     h = ngx_array_push(&cmcf->phases[NGX_HTTP_ACCESS_PHASE].handlers);
346     if (h == NULL) {
347         return NGX_ERROR;
348     }
349
350     *h = ngx_http_auth_request_handler;
351
352     return NGX_OK;
353 }
354
355
356 static char *
357 ngx_http_auth_request(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
358 {
359     ngx_http_auth_request_conf_t *arcf = conf;
360
361     ngx_str_t *value;
362
363     if (arcf->uri.data != NULL) {
364         return "is duplicate";
365     }
366
367     value = cf->args->elts;
368
369     if (ngx_strcmp(value[1].data, "off") == 0) {
370         arcf->uri.len = 0;
371         arcf->uri.data = (u_char *) "";
372
373         return NGX_CONF_OK;
374     }
375
376     arcf->uri = value[1];
377
378     return NGX_CONF_OK;
379 }
380
381
382 static char *
383 ngx_http_auth_request_set(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
384 {
385     ngx_http_auth_request_conf_t *arcf = conf;
386
387     ngx_str_t *value;
388     ngx_http_variable_t *v;
389     ngx_http_auth_request_variable_t *av;
390     ngx_http_compile_complex_value_t ccv;
391
392     value = cf->args->elts;
393
394     if (value[1].data[0] != '$') {
395         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
396             "invalid variable name \"%V\"", &value[1]);
397         return NGX_CONF_ERROR;
398     }
399
400     value[1].len--;
401     value[1].data++;
402
403     if (arcf->vars == NGX_CONF_UNSET_PTR) {
404         arcf->vars = ngx_array_create(cf->pool, 1,

```

```

405                                     sizeof(ngx\_http\_auth\_request\_variable\_t));
406     if (arcf->vars == NULL) {
407         return NGX\_CONF\_ERROR;
408     }
409 }
410
411 av = ngx\_array\_push(arcf->vars);
412 if (av == NULL) {
413     return NGX\_CONF\_ERROR;
414 }
415
416 v = ngx\_http\_add\_variable(cf, &value[1], NGX\_HTTP\_VAR\_CHANGEABLE);
417 if (v == NULL) {
418     return NGX\_CONF\_ERROR;
419 }
420
421 av->index = ngx\_http\_get\_variable\_index(cf, &value[1]);
422 if (av->index == NGX\_ERROR) {
423     return NGX\_CONF\_ERROR;
424 }
425
426 if (v->get_handler == NULL) {
427     v->get_handler = ngx\_http\_auth\_request\_variable;
428     v->data = (uintptr_t) av;
429 }
430
431 av->set_handler = v->set_handler;
432
433 ngx\_memzero(&ccv, sizeof(ngx\_http\_compile\_complex\_value\_t));
434
435 ccv.cf = cf;
436 ccv.value = &value[2];
437 ccv.complex_value = &av->value;
438
439 if (ngx\_http\_compile\_complex\_value(&ccv) != NGX\_OK) {
440     return NGX\_CONF\_ERROR;
441 }
442
443 return NGX\_CONF\_OK;
444 }

```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_autoindex\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_autoindex\\_commands](#)
- [ngx\\_http\\_autoindex\\_format](#)
- [ngx\\_http\\_autoindex\\_module](#)
- [ngx\\_http\\_autoindex\\_module\\_ctx](#)

## Data types defined

- [ngx\\_http\\_autoindex\\_entry\\_t](#)
- [ngx\\_http\\_autoindex\\_loc\\_conf\\_t](#)

## Functions defined

- [ngx\\_http\\_autoindex\\_cmp\\_entries](#)
- [ngx\\_http\\_autoindex\\_create\\_loc\\_conf](#)
- [ngx\\_http\\_autoindex\\_error](#)
- [ngx\\_http\\_autoindex\\_handler](#)
- [ngx\\_http\\_autoindex\\_html](#)
- [ngx\\_http\\_autoindex\\_init](#)
- [ngx\\_http\\_autoindex\\_json](#)
- [ngx\\_http\\_autoindex\\_jsonp\\_callback](#)
- [ngx\\_http\\_autoindex\\_merge\\_loc\\_conf](#)
- [ngx\\_http\\_autoindex\\_xml](#)

## Macros defined

- [NGX\\_HTTP\\_AUTOINDEX\\_HTML](#)
- [NGX\\_HTTP\\_AUTOINDEX\\_JSON](#)
- [NGX\\_HTTP\\_AUTOINDEX\\_JSONP](#)
- [NGX\\_HTTP\\_AUTOINDEX\\_NAME\\_LEN](#)
- [NGX\\_HTTP\\_AUTOINDEX\\_PREALLOCATE](#)
- [NGX\\_HTTP\\_AUTOINDEX\\_XML](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
```

```

5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 #if 0
14
15 typedef struct {
16     ngx_buf_t      *buf;
17     size_t          size;
18     ngx_pool_t     *pool;
19     size_t          alloc_size;
20     ngx_chain_t    **last_out;
21 } ngx_http_autoindex_ctx_t;
22
23 #endif
24
25
26 typedef struct {
27     ngx_str_t      name;
28     size_t         utf_len;
29     size_t         escape;
30     size_t         escape_html;
31
32     unsigned       dir:1;
33     unsigned       file:1;
34
35     time_t         mtime;
36     off_t          size;
37 } ngx_http_autoindex_entry_t;
38
39
40 typedef struct {
41     ngx_flag_t     enable;
42     ngx_uint_t     format;
43     ngx_flag_t     localtime;
44     ngx_flag_t     exact_size;
45 } ngx_http_autoindex_loc_conf_t;
46
47
48 #define NGX_HTTP_AUTOINDEX_HTML          0
49 #define NGX_HTTP_AUTOINDEX_JSON         1
50 #define NGX_HTTP_AUTOINDEX_JSONP       2
51 #define NGX_HTTP_AUTOINDEX_XML         3
52
53 #define NGX_HTTP_AUTOINDEX_PREALLOCATE  50
54
55 #define NGX_HTTP_AUTOINDEX_NAME_LEN     50
56
57
58 static ngx_buf_t *ngx_http_autoindex_html(ngx_http_request_t *r,
59     ngx_array_t *entries);
60 static ngx_buf_t *ngx_http_autoindex_json(ngx_http_request_t *r,
61     ngx_array_t *entries, ngx_str_t *callback);
62 static ngx_int_t ngx_http_autoindex_jsonp_callback(ngx_http_request_t *r,
63     ngx_str_t *callback);
64 static ngx_buf_t *ngx_http_autoindex_xml(ngx_http_request_t *r,
65     ngx_array_t *entries);
66
67 static int ngx_libc_cdecl ngx_http_autoindex_cmp_entries(const void *one,
68     const void *two);
69 static ngx_int_t ngx_http_autoindex_error(ngx_http_request_t *r,
70     ngx_dir_t *dir, ngx_str_t *name);
71
72 static ngx_int_t ngx_http_autoindex_init(ngx_conf_t *cf);
73 static void *ngx_http_autoindex_create_loc_conf(ngx_conf_t *cf);
74 static char *ngx_http_autoindex_merge_loc_conf(ngx_conf_t *cf,
75     void *parent, void *child);
76
77
78 static ngx_conf_enum_t ngx_http_autoindex_format[] = {
79     { ngx_string("html"), NGX_HTTP_AUTOINDEX_HTML },
80     { ngx_string("json"), NGX_HTTP_AUTOINDEX_JSON },

```

```

81     { ngx_string("jsonp"), NGX_HTTP_AUTOINDEX_JSONP },
82     { ngx_string("xml"), NGX_HTTP_AUTOINDEX_XML },
83     { ngx_null_string, 0 }
84 };
85
86
87 static ngx_command_t  ngx_http_autoindex_commands[] = {
88
89     { ngx_string("autoindex"),
90       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
91       ngx_conf_set_flag_slot,
92       NGX_HTTP_LOC_CONF_OFFSET,
93       offsetof(ngx_http_autoindex_loc_conf_t, enable),
94       NULL },
95
96     { ngx_string("autoindex_format"),
97       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
98       ngx_conf_set_enum_slot,
99       NGX_HTTP_LOC_CONF_OFFSET,
100      offsetof(ngx_http_autoindex_loc_conf_t, format),
101      &ngx_http_autoindex_format },
102
103     { ngx_string("autoindex_localtime"),
104       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
105       ngx_conf_set_flag_slot,
106       NGX_HTTP_LOC_CONF_OFFSET,
107       offsetof(ngx_http_autoindex_loc_conf_t, localtime),
108       NULL },
109
110     { ngx_string("autoindex_exact_size"),
111       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
112       ngx_conf_set_flag_slot,
113       NGX_HTTP_LOC_CONF_OFFSET,
114       offsetof(ngx_http_autoindex_loc_conf_t, exact_size),
115       NULL },
116
117     ngx_null_command
118 };
119
120
121 static ngx_http_module_t  ngx_http_autoindex_module_ctx = {
122     NULL,                                     /* preconfiguration */
123     ngx_http_autoindex_init,                 /* postconfiguration */
124
125     NULL,                                     /* create main configuration */
126     NULL,                                     /* init main configuration */
127
128     NULL,                                     /* create server configuration */
129     NULL,                                     /* merge server configuration */
130
131     ngx_http_autoindex_create_loc_conf,     /* create location configuration */
132     ngx_http_autoindex_merge_loc_conf      /* merge location configuration */
133 };
134
135
136 ngx_module_t  ngx_http_autoindex_module = {
137     NGX_MODULE_V1,
138     &ngx_http_autoindex_module_ctx,        /* module context */
139     ngx_http_autoindex_commands,          /* module directives */
140     NGX_HTTP_MODULE,                      /* module type */
141     NULL,                                  /* init master */
142     NULL,                                  /* init module */
143     NULL,                                  /* init process */
144     NULL,                                  /* init thread */
145     NULL,                                  /* exit thread */
146     NULL,                                  /* exit process */
147     NULL,                                  /* exit master */
148     NGX_MODULE_V1_PADDING
149 };
150
151
152 static ngx_int_t
153 ngx_http_autoindex_handler(ngx_http_request_t *r)
154 {
155     u_char          *last, *filename;
156     size_t          len, allocated, root;

```

```

157     ngx_err_t           err;
158     ngx_buf_t          *b;
159     ngx_int_t          rc;
160     ngx_str_t          path, callback;
161     ngx_dir_t          dir;
162     ngx_uint_t         level, format;
163     ngx_pool_t         *pool;
164     ngx_chain_t        out;
165     ngx_array_t        entries;
166     ngx_http_autoindex_entry_t *entry;
167     ngx_http_autoindex_loc_conf_t *alcf;
168
169     if (r->uri.data[r->uri.len - 1] != '/') {
170         return NGX_DECLINED;
171     }
172
173     if (!(r->method & (NGX_HTTP_GET|NGX_HTTP_HEAD))) {
174         return NGX_DECLINED;
175     }
176
177     alcf = ngx_http_get_module_loc_conf(r, ngx_http_autoindex_module);
178
179     if (!alcf->enable) {
180         return NGX_DECLINED;
181     }
182
183     /* NGX_DIR_MASK_LEN is lesser than NGX_HTTP_AUTOINDEX_PREALLOCATE */
184
185     last = ngx_http_map_uri_to_path(r, &path, &root,
186                                     NGX_HTTP_AUTOINDEX_PREALLOCATE);
187     if (last == NULL) {
188         return NGX_HTTP_INTERNAL_SERVER_ERROR;
189     }
190
191     allocated = path.len;
192     path.len = last - path.data;
193     if (path.len > 1) {
194         path.len--;
195     }
196     path.data[path.len] = '\0';
197
198     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
199                  "http autoindex: \"%s\"", path.data);
200
201     format = alcf->format;
202
203     if (format == NGX_HTTP_AUTOINDEX_JSONP) {
204         if (ngx_http_autoindex_jsonp_callback(r, &callback) != NGX_OK) {
205             return NGX_HTTP_BAD_REQUEST;
206         }
207
208         if (callback.len == 0) {
209             format = NGX_HTTP_AUTOINDEX_JSON;
210         }
211     }
212
213     if (ngx_open_dir(&path, &dir) == NGX_ERROR) {
214         err = ngx_errno;
215
216         if (err == NGX_ENOENT
217             || err == NGX_ENOTDIR
218             || err == NGX_ENAMETOOLONG)
219         {
220             level = NGX_LOG_ERR;
221             rc = NGX_HTTP_NOT_FOUND;
222
223         } else if (err == NGX_EACCES) {
224             level = NGX_LOG_ERR;
225             rc = NGX_HTTP_FORBIDDEN;
226
227         } else {
228             level = NGX_LOG_CRIT;
229             rc = NGX_HTTP_INTERNAL_SERVER_ERROR;
230         }
231
232         ngx_log_error(level, r->connection->log, err,

```

```

233         ngx_open_dir_n " \"%s\" failed", path.data);
234
235     return rc;
236 }
237
238 #if (NGX_SUPPRESS_WARN)
239
240     /* MSVC thinks 'entries' may be used without having been initialized */
241     ngx_memzero(&entries, sizeof(ngx_array_t));
242
243 #endif
244
245     /* TODO: pool should be temporary pool */
246     pool = r->pool;
247
248     if (ngx_array_init(&entries, pool, 40, sizeof(ngx_http_autoindex_entry_t))
249         != NGX_OK)
250     {
251         return ngx_http_autoindex_error(r, &dir, &path);
252     }
253
254     r->headers_out.status = NGX_HTTP_OK;
255
256     switch (format) {
257
258     case NGX_HTTP_AUTOINDEX_JSON:
259         ngx_str_set(&r->headers_out.content_type, "application/json");
260         break;
261
262     case NGX_HTTP_AUTOINDEX_JSONP:
263         ngx_str_set(&r->headers_out.content_type, "application/javascript");
264         break;
265
266     case NGX_HTTP_AUTOINDEX_XML:
267         ngx_str_set(&r->headers_out.content_type, "text/xml");
268         ngx_str_set(&r->headers_out.charset, "utf-8");
269         break;
270
271     default: /* NGX_HTTP_AUTOINDEX_HTML */
272         ngx_str_set(&r->headers_out.content_type, "text/html");
273         break;
274     }
275
276     r->headers_out.content_type_len = r->headers_out.content_type.len;
277     r->headers_out.content_type_lowercase = NULL;
278
279     rc = ngx_http_send_header(r);
280
281     if (rc == NGX_ERROR || rc > NGX_OK || r->header_only) {
282         if (ngx_close_dir(&dir) == NGX_ERROR) {
283             ngx_log_error(NGX_LOG_ALERT, r->connection->log, ngx_errno,
284                 ngx_close_dir_n " \"%V\" failed", &path);
285         }
286
287         return rc;
288     }
289
290     filename = path.data;
291     filename[path.len] = '/';
292
293     for ( ;; ) {
294         ngx_set_errno(0);
295
296         if (ngx_read_dir(&dir) == NGX_ERROR) {
297             err = ngx_errno;
298
299             if (err != NGX_ENOMOREFILES) {
300                 ngx_log_error(NGX_LOG_CRIT, r->connection->log, err,
301                     ngx_read_dir_n " \"%V\" failed", &path);
302                 return ngx_http_autoindex_error(r, &dir, &path);
303             }
304
305             break;
306         }
307
308         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,

```



```

309         "http autoindex file: \"%s\"", ngx_de_name(&dir));
310
311     len = ngx_de_namelen(&dir);
312
313     if (ngx_de_name(&dir)[0] == '.') {
314         continue;
315     }
316
317     if (!dir.valid_info) {
318
319         /* 1 byte for '/' and 1 byte for terminating '\0' */
320
321         if (path.len + 1 + len + 1 > allocated) {
322             allocated = path.len + 1 + len + 1
323                 + NGX_HTTP_AUTOINDEX_PREALLOCATE;
324
325             filename = ngx_pnalloc(pool, allocated);
326             if (filename == NULL) {
327                 return ngx_http_autoindex_error(r, &dir, &path);
328             }
329
330             last = ngx_cpysrtn(filename, path.data, path.len + 1);
331             *last++ = '/';
332         }
333
334         ngx_cpysrtn(last, ngx_de_name(&dir), len + 1);
335
336         if (ngx_de_info(filename, &dir) == NGX_FILE_ERROR) {
337             err = ngx_errno;
338
339             if (err != NGX_ENOENT && err != NGX_ELOOP) {
340                 ngx_log_error(NGX_LOG_CRIT, r->connection->log, err,
341                     ngx_de_info_n " \"%s\" failed", filename);
342
343                 if (err == NGX_EACCES) {
344                     continue;
345                 }
346
347                 return ngx_http_autoindex_error(r, &dir, &path);
348             }
349
350             if (ngx_de_link_info(filename, &dir) == NGX_FILE_ERROR) {
351                 ngx_log_error(NGX_LOG_CRIT, r->connection->log, ngx_errno,
352                     ngx_de_link_info_n " \"%s\" failed",
353                     filename);
354                 return ngx_http_autoindex_error(r, &dir, &path);
355             }
356         }
357     }
358
359     entry = ngx_array_push(&entries);
360     if (entry == NULL) {
361         return ngx_http_autoindex_error(r, &dir, &path);
362     }
363
364     entry->name.len = len;
365
366     entry->name.data = ngx_pnalloc(pool, len + 1);
367     if (entry->name.data == NULL) {
368         return ngx_http_autoindex_error(r, &dir, &path);
369     }
370
371     ngx_cpysrtn(entry->name.data, ngx_de_name(&dir), len + 1);
372
373     entry->dir = ngx_de_is_dir(&dir);
374     entry->file = ngx_de_is_file(&dir);
375     entry->mtime = ngx_de_mtime(&dir);
376     entry->size = ngx_de_size(&dir);
377 }
378
379 if (ngx_close_dir(&dir) == NGX_ERROR) {
380     ngx_log_error(NGX_LOG_ALERT, r->connection->log, ngx_errno,
381         ngx_close_dir_n " \"%V\" failed", &path);
382 }
383
384 if (entries.nelts > 1) {

```

```

385     ngx_qsort(entries.elts, (size_t) entries.nelts,
386               sizeof(ngx_http_autoindex_entry_t),
387               ngx_http_autoindex_cmp_entries);
388 }
389
390 switch (format) {
391
392 case NGX_HTTP_AUTOINDEX_JSON:
393     b = ngx_http_autoindex_json(r, &entries, NULL);
394     break;
395
396 case NGX_HTTP_AUTOINDEX_JSONP:
397     b = ngx_http_autoindex_json(r, &entries, &callback);
398     break;
399
400 case NGX_HTTP_AUTOINDEX_XML:
401     b = ngx_http_autoindex_xml(r, &entries);
402     break;
403
404 default: /* NGX_HTTP_AUTOINDEX_HTML */
405     b = ngx_http_autoindex_html(r, &entries);
406     break;
407 }
408
409 if (b == NULL) {
410     return NGX_ERROR;
411 }
412
413 /* TODO: free temporary pool */
414
415 if (r == r->main) {
416     b->last_buf = 1;
417 }
418
419 b->last_in_chain = 1;
420
421 out.buf = b;
422 out.next = NULL;
423
424 return ngx_http_output_filter(r, &out);
425 }
426
427
428 static ngx_buf_t *
429 ngx_http_autoindex_html(ngx_http_request_t *r, ngx_array_t *entries)
430 {
431     u_char          *last, scale;
432     off_t           length;
433     size_t          len, char_len, escape_html;
434     ngx_tm_t        tm;
435     ngx_buf_t       *b;
436     ngx_int_t       size;
437     ngx_uint_t      i, utf8;
438     ngx_time_t      *tp;
439     ngx_http_autoindex_entry_t *entry;
440     ngx_http_autoindex_loc_conf_t *alcf;
441
442     static u_char title[] =
443         "<html>" CRLF
444         "<head><title>Index of "
445         ;
446
447     static u_char header[] =
448         "</title></head>" CRLF
449         "<body bgcolor=\"white\">" CRLF
450         "<h1>Index of "
451         ;
452
453     static u_char tail[] =
454         "</body>" CRLF
455         "</html>" CRLF
456         ;
457
458     static char *months[] = { "Jan", "Feb", "Mar", "Apr", "May", "Jun",
459                               "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };
460

```

```

461 if (r->headers_out.charset.len == 5
462     && ngx_strncasecmp(r->headers_out.charset.data, (u_char *) "utf-8", 5)
463     == 0)
464 {
465     utf8 = 1;
466
467 } else {
468     utf8 = 0;
469 }
470
471 escape_html = ngx_escape_html(NULL, r->uri.data, r->uri.len);
472
473 len = sizeof(title) - 1
474     + r->uri.len + escape_html
475     + sizeof(header) - 1
476     + r->uri.len + escape_html
477     + sizeof("</h1>") - 1
478     + sizeof("<hr><pre><a href=\"../\">../</a>" CRLF) - 1
479     + sizeof("</pre><hr>") - 1
480     + sizeof(tail) - 1;
481
482 entry = entries->elts;
483 for (i = 0; i < entries->nelts; i++) {
484     entry[i].escape = 2 * ngx_escape_uri(NULL, entry[i].name.data,
485                                         entry[i].name.len,
486                                         NGX_ESCAPE_URI_COMPONENT);
487
488     entry[i].escape_html = ngx_escape_html(NULL, entry[i].name.data,
489                                           entry[i].name.len);
490
491     if (utf8) {
492         entry[i].utf_len = ngx_utf8_length(entry[i].name.data,
493                                           entry[i].name.len);
494     } else {
495         entry[i].utf_len = entry[i].name.len;
496     }
497
498     len += sizeof("<a href=\"") - 1
499         + entry[i].name.len + entry[i].escape
500         + 1 /* 1 is for "/" */
501         + sizeof(">") - 1
502         + entry[i].name.len - entry[i].utf_len
503         + entry[i].escape_html
504         + NGX_HTTP_AUTOINDEX_NAME_LEN + sizeof("&gt;") - 2
505         + sizeof("</a>") - 1
506         + sizeof(" 28-Sep-1970 12:00 ") - 1
507         + 20 /* the file size */
508         + 2;
509 }
510
511 b = ngx_create_temp_buf(r->pool, len);
512 if (b == NULL) {
513     return NULL;
514 }
515
516 b->last = ngx_cpymem(b->last, title, sizeof(title) - 1);
517
518 if (escape_html) {
519     b->last = (u_char *) ngx_escape_html(b->last, r->uri.data, r->uri.len);
520     b->last = ngx_cpymem(b->last, header, sizeof(header) - 1);
521     b->last = (u_char *) ngx_escape_html(b->last, r->uri.data, r->uri.len);
522
523 } else {
524     b->last = ngx_cpymem(b->last, r->uri.data, r->uri.len);
525     b->last = ngx_cpymem(b->last, header, sizeof(header) - 1);
526     b->last = ngx_cpymem(b->last, r->uri.data, r->uri.len);
527 }
528
529 b->last = ngx_cpymem(b->last, "</h1>", sizeof("</h1>") - 1);
530
531 b->last = ngx_cpymem(b->last, "<hr><pre><a href=\"../\">../</a>" CRLF,
532                    sizeof("<hr><pre><a href=\"../\">../</a>" CRLF) - 1);
533
534 alcf = ngx_http_get_module_loc_conf(r, ngx_http_autoindex_module);
535 tp = ngx_timeofday();
536

```

```

537 for (i = 0; i < entries->nelts; i++) {
538     b->last = ngx_cpymem(b->last, "<a href=\"", sizeof("<a href=\"") - 1);
539
540     if (entry[i].escape) {
541         ngx_escape_uri(b->last, entry[i].name.data, entry[i].name.len,
542             NGX_ESCAPE_URI_COMPONENT);
543
544         b->last += entry[i].name.len + entry[i].escape;
545
546     } else {
547         b->last = ngx_cpymem(b->last, entry[i].name.data,
548             entry[i].name.len);
549     }
550
551     if (entry[i].dir) {
552         *b->last++ = '/';
553     }
554
555     *b->last++ = '"';
556     *b->last++ = '>';
557
558     len = entry[i].utf_len;
559
560     if (entry[i].name.len != len) {
561         if (len > NGX_HTTP_AUTOINDEX_NAME_LEN) {
562             char_len = NGX_HTTP_AUTOINDEX_NAME_LEN - 3 + 1;
563
564         } else {
565             char_len = NGX_HTTP_AUTOINDEX_NAME_LEN + 1;
566         }
567
568         last = b->last;
569         b->last = ngx_utf8_cpystri(b->last, entry[i].name.data,
570             char_len, entry[i].name.len + 1);
571
572         if (entry[i].escape_html) {
573             b->last = (u_char *) ngx_escape_html(last, entry[i].name.data,
574                 b->last - last);
575         }
576
577         last = b->last;
578
579     } else {
580         if (entry[i].escape_html) {
581             if (len > NGX_HTTP_AUTOINDEX_NAME_LEN) {
582                 char_len = NGX_HTTP_AUTOINDEX_NAME_LEN - 3;
583
584             } else {
585                 char_len = len;
586             }
587
588             b->last = (u_char *) ngx_escape_html(b->last,
589                 entry[i].name.data, char_len);
590
591             last = b->last;
592         } else {
593             b->last = ngx_cpystri(b->last, entry[i].name.data,
594                 NGX_HTTP_AUTOINDEX_NAME_LEN + 1);
595
596             last = b->last - 3;
597         }
598     }
599
600     if (len > NGX_HTTP_AUTOINDEX_NAME_LEN) {
601         b->last = ngx_cpymem(last, "..&gt;</a>", sizeof("..&gt;</a>") - 1);
602
603     } else {
604         if (entry[i].dir && NGX_HTTP_AUTOINDEX_NAME_LEN - len > 0) {
605             *b->last++ = '/';
606             len++;
607         }
608
609         b->last = ngx_cpymem(b->last, "</a>", sizeof("</a>") - 1);
610
611         if (NGX_HTTP_AUTOINDEX_NAME_LEN - len > 0) {
612             ngx_memset(b->last, ' ', NGX_HTTP_AUTOINDEX_NAME_LEN - len);
613             b->last += NGX_HTTP_AUTOINDEX_NAME_LEN - len;

```

```

613     }
614 }
615
616 *b->last++ = ' ';
617
618 ngx_gmtime(entry[i].mtime + tp->gmtoff * 60 * alcf->localtime, &tm);
619
620 b->last = ngx_sprintf(b->last, "%02d-%s-%d %02d:%02d ",
621                     tm.ngx_tm_mday,
622                     months[tm.ngx_tm_mon - 1],
623                     tm.ngx_tm_year,
624                     tm.ngx_tm_hour,
625                     tm.ngx_tm_min);
626
627 if (alcf->exact_size) {
628     if (entry[i].dir) {
629         b->last = ngx_cpymem(b->last, "                -",
630                             sizeof("                -") - 1);
631     } else {
632         b->last = ngx_sprintf(b->last, "%190", entry[i].size);
633     }
634 }
635 } else {
636     if (entry[i].dir) {
637         b->last = ngx_cpymem(b->last, "                -",
638                             sizeof("                -") - 1);
639     }
640     } else {
641         length = entry[i].size;
642
643         if (length > 1024 * 1024 * 1024 - 1) {
644             size = (ngx_int_t) (length / (1024 * 1024 * 1024));
645             if ((length % (1024 * 1024 * 1024))
646                 > (1024 * 1024 * 1024 / 2 - 1))
647             {
648                 size++;
649             }
650             scale = 'G';
651         }
652         } else if (length > 1024 * 1024 - 1) {
653             size = (ngx_int_t) (length / (1024 * 1024));
654             if ((length % (1024 * 1024)) > (1024 * 1024 / 2 - 1)) {
655                 size++;
656             }
657             scale = 'M';
658         }
659         } else if (length > 9999) {
660             size = (ngx_int_t) (length / 1024);
661             if (length % 1024 > 511) {
662                 size++;
663             }
664             scale = 'K';
665         }
666         } else {
667             size = (ngx_int_t) length;
668             scale = '\0';
669         }
670     }
671     if (scale) {
672         b->last = ngx_sprintf(b->last, "%6i%c", size, scale);
673     }
674     } else {
675         b->last = ngx_sprintf(b->last, " %6i", size);
676     }
677 }
678 }
679
680 *b->last++ = CR;
681 *b->last++ = LF;
682 }
683
684 b->last = ngx_cpymem(b->last, "</pre><hr>", sizeof("</pre><hr>") - 1);
685
686 b->last = ngx_cpymem(b->last, tail, sizeof(tail) - 1);
687
688 return b;

```

```

689 }
690
691
692 static ngx_buf_t *
693 ngx_http_autoindex_json(ngx_http_request_t *r, ngx_array_t *entries,
694 ngx_str_t *callback)
695 {
696     size_t          len;
697     ngx_buf_t      *b;
698     ngx_uint_t      i;
699     ngx_http_autoindex_entry_t *entry;
700
701     len = sizeof("[ " CRLF CRLF "]") - 1;
702
703     if (callback) {
704         len += sizeof("/* callback */" CRLF "();") - 1 + callback->len;
705     }
706
707     entry = entries->elts;
708
709     for (i = 0; i < entries->nelts; i++) {
710         entry[i].escape = ngx_escape_json(NULL, entry[i].name.data,
711         entry[i].name.len);
712
713         len += sizeof("{ }," CRLF) - 1
714             + sizeof("\"name\": \"") - 1
715             + entry[i].name.len + entry[i].escape
716             + sizeof(", \"type\": \"directory\"") - 1
717             + sizeof(", \"mtime\": \"Wed, 31 Dec 1986 10:00:00 GMT\"") - 1;
718
719         if (entry[i].file) {
720             len += sizeof(", \"size\":") - 1 + NGX_OFF_T_LEN;
721         }
722     }
723
724     b = ngx_create_temp_buf(r->pool, len);
725     if (b == NULL) {
726         return NULL;
727     }
728
729     if (callback) {
730         b->last = ngx_cpymem(b->last, "/* callback */" CRLF,
731         sizeof("/* callback */" CRLF) - 1);
732
733         b->last = ngx_cpymem(b->last, callback->data, callback->len);
734
735         *b->last++ = '(';
736     }
737
738     *b->last++ = '[';
739
740     for (i = 0; i < entries->nelts; i++) {
741         b->last = ngx_cpymem(b->last, CRLF "{ \"name\": \"",
742         sizeof(CRLF "{ \"name\": \"") - 1);
743
744         if (entry[i].escape) {
745             b->last = (u_char *) ngx_escape_json(b->last, entry[i].name.data,
746             entry[i].name.len);
747         } else {
748             b->last = ngx_cpymem(b->last, entry[i].name.data,
749             entry[i].name.len);
750         }
751
752         b->last = ngx_cpymem(b->last, "\", \"type\": \"",
753         sizeof "\", \"type\": \"") - 1);
754
755         if (entry[i].dir) {
756             b->last = ngx_cpymem(b->last, "directory", sizeof("directory") - 1);
757
758         } else if (entry[i].file) {
759             b->last = ngx_cpymem(b->last, "file", sizeof("file") - 1);
760
761         } else {
762             b->last = ngx_cpymem(b->last, "other", sizeof("other") - 1);
763         }
764     }

```

```

765     b->last = ngx_cpymem(b->last, "\", \"mtime\":"",
766                       sizeof "\", \"mtime\":" - 1);
767
768     b->last = ngx_http_time(b->last, entry[i].mtime);
769
770     if (entry[i].file) {
771         b->last = ngx_cpymem(b->last, "\", \"size\":"",
772                           sizeof "\", \"size\":" - 1);
773         b->last = ngx_sprintf(b->last, "%0", entry[i].size);
774
775     } else {
776         *b->last++ = ' ';
777     }
778
779     b->last = ngx_cpymem(b->last, " },", sizeof(" },") - 1);
780 }
781
782 if (i > 0) {
783     b->last--; /* strip last comma */
784 }
785
786 b->last = ngx_cpymem(b->last, CRLF "]", sizeof(CRLF "]") - 1);
787
788 if (callback) {
789     *b->last++ = ')'; *b->last++ = ';';
790 }
791
792 return b;
793 }
794
795
796 static ngx_int_t
797 ngx_http_autoindex_jsonp_callback(ngx_http_request_t *r, ngx_str_t *callback)
798 {
799     u_char      *p, c, ch;
800     ngx_uint_t  i;
801
802     if (ngx_http_arg(r, (u_char *) "callback", 8, callback) != NGX_OK) {
803         callback->len = 0;
804         return NGX_OK;
805     }
806
807     if (callback->len > 128) {
808         ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
809                     "client sent too long callback name: \"%V\"", callback);
810         return NGX_DECLINED;
811     }
812
813     p = callback->data;
814
815     for (i = 0; i < callback->len; i++) {
816         ch = p[i];
817
818         c = (u_char) (ch | 0x20);
819         if (c >= 'a' && c <= 'z') {
820             continue;
821         }
822
823         if ((ch >= '0' && ch <= '9') || ch == '_' || ch == '.') {
824             continue;
825         }
826
827         ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
828                     "client sent invalid callback name: \"%V\"", callback);
829
830         return NGX_DECLINED;
831     }
832
833     return NGX_OK;
834 }
835
836
837 static ngx_buf_t *
838 ngx_http_autoindex_xml(ngx_http_request_t *r, ngx_array_t *entries)
839 {
840     size_t      len;

```

```

841     ngx_tm_t          tm;
842     ngx_buf_t        *b;
843     ngx_str_t        type;
844     ngx_uint_t       i;
845     ngx_http_autoindex_entry_t *entry;
846
847     static u_char head[] = "<?xml version=\\"1.0\\"?>" CRLF "<list>" CRLF;
848     static u_char tail[] = "</list>" CRLF;
849
850     len = sizeof(head) - 1 + sizeof(tail) - 1;
851
852     entry = entries->elts;
853
854     for (i = 0; i < entries->nelts; i++) {
855         entry[i].escape = ngx_escape_html(NULL, entry[i].name.data,
856                                         entry[i].name.len);
857
858         len += sizeof("<directory></directory>" CRLF) - 1
859              + entry[i].name.len + entry[i].escape
860              + sizeof(" mtime=\\"1986-12-31T10:00:00Z\\"") - 1;
861
862         if (entry[i].file) {
863             len += sizeof(" size=\\"\\") - 1 + NGX_OFF_T_LEN;
864         }
865     }
866
867     b = ngx_create_temp_buf(r->pool, len);
868     if (b == NULL) {
869         return NULL;
870     }
871
872     b->last = ngx_cpymem(b->last, head, sizeof(head) - 1);
873
874     for (i = 0; i < entries->nelts; i++) {
875         *b->last++ = '<';
876
877         if (entry[i].dir) {
878             ngx_str_set(&type, "directory");
879
880         } else if (entry[i].file) {
881             ngx_str_set(&type, "file");
882
883         } else {
884             ngx_str_set(&type, "other");
885         }
886
887         b->last = ngx_cpymem(b->last, type.data, type.len);
888
889         b->last = ngx_cpymem(b->last, " mtime=\\"", sizeof(" mtime=\\"") - 1);
890
891         ngx_qmtime(entry[i].mtime, &tm);
892
893         b->last = ngx_sprintf(b->last, "%4d-%02d-%02dT%02d:%02d:%02dZ",
894                             tm.ngx_tm_year, tm.ngx_tm_mon,
895                             tm.ngx_tm_mday, tm.ngx_tm_hour,
896                             tm.ngx_tm_min, tm.ngx_tm_sec);
897
898         if (entry[i].file) {
899             b->last = ngx_cpymem(b->last, "\" size=\\"",
900                               sizeof("\" size=\\"") - 1);
901             b->last = ngx_sprintf(b->last, "%0", entry[i].size);
902         }
903
904         *b->last++ = '"'; *b->last++ = '>';
905
906         if (entry[i].escape) {
907             b->last = (u_char *) ngx_escape_html(b->last, entry[i].name.data,
908                                                entry[i].name.len);
909         } else {
910             b->last = ngx_cpymem(b->last, entry[i].name.data,
911                               entry[i].name.len);
912         }
913
914         *b->last++ = '<'; *b->last++ = '/';
915
916         b->last = ngx_cpymem(b->last, type.data, type.len);

```



```

917     *b->last++ = '>';
918
919     *b->last++ = CR; *b->last++ = LF;
920 }
921
922 b->last = ngx_cpymem(b->last, tail, sizeof(tail) - 1);
923
924 return b;
925 }
926
927
928
929 static int ngx_libc_cdecl
930 ngx_http_autoindex_cmp_entries(const void *one, const void *two)
931 {
932     ngx_http_autoindex_entry_t *first = (ngx_http_autoindex_entry_t *) one;
933     ngx_http_autoindex_entry_t *second = (ngx_http_autoindex_entry_t *) two;
934
935     if (first->dir && !second->dir) {
936         /* move the directories to the start */
937         return -1;
938     }
939
940     if (!first->dir && second->dir) {
941         /* move the directories to the start */
942         return 1;
943     }
944
945     return (int) ngx_strcmp(first->name.data, second->name.data);
946 }
947
948
949 #if 0
950
951 static ngx_buf_t *
952 ngx_http_autoindex_alloc(ngx_http_autoindex_ctx_t *ctx, size_t size)
953 {
954     ngx_chain_t *cl;
955
956     if (ctx->buf) {
957         if ((size_t) (ctx->buf->end - ctx->buf->last) >= size) {
958             return ctx->buf;
959         }
960
961         ctx->size += ctx->buf->last - ctx->buf->pos;
962     }
963
964     ctx->buf = ngx_create_temp_buf(ctx->pool, ctx->alloc_size);
965     if (ctx->buf == NULL) {
966         return NULL;
967     }
968
969     cl = ngx_alloc_chain_link(ctx->pool);
970     if (cl == NULL) {
971         return NULL;
972     }
973
974     cl->buf = ctx->buf;
975     cl->next = NULL;
976
977     *ctx->last_out = cl;
978     ctx->last_out = &cl->next;
979
980     return ctx->buf;
981 }
982
983 #endif
984
985
986 static ngx_int_t
987 ngx_http_autoindex_error(ngx_http_request_t *r, ngx_dir_t *dir, ngx_str_t *name)
988 {
989     if (ngx_close_dir(dir) == NGX_ERROR) {
990         ngx_log_error(NGX_LOG_ALERT, r->connection->log, ngx_errno,
991             ngx_close_dir_n " \"%V\" failed", name);

```

```

993     }
994
995     return r->header_sent ? NGX\_ERROR : NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
996 }
997
998
999 static void *
1000 ngx_http_autoindex_create_loc_conf(ngx\_conf\_t *cf)
1001 {
1002     ngx\_http\_autoindex\_loc\_conf\_t *conf;
1003
1004     conf = ngx\_palloc(cf->pool, sizeof(ngx\_http\_autoindex\_loc\_conf\_t));
1005     if (conf == NULL) {
1006         return NULL;
1007     }
1008
1009     conf->enable = NGX\_CONF\_UNSET;
1010     conf->format = NGX\_CONF\_UNSET\_UINT;
1011     conf->localtime = NGX\_CONF\_UNSET;
1012     conf->exact_size = NGX\_CONF\_UNSET;
1013
1014     return conf;
1015 }
1016
1017
1018 static char *
1019 ngx_http_autoindex_merge_loc_conf(ngx\_conf\_t *cf, void *parent, void *child)
1020 {
1021     ngx\_http\_autoindex\_loc\_conf\_t *prev = parent;
1022     ngx\_http\_autoindex\_loc\_conf\_t *conf = child;
1023
1024     ngx\_conf\_merge\_value(conf->enable, prev->enable, 0);
1025     ngx\_conf\_merge\_uint\_value(conf->format, prev->format,
1026                               NGX\_HTTP\_AUTOINDEX\_HTML);
1027     ngx\_conf\_merge\_value(conf->localtime, prev->localtime, 0);
1028     ngx\_conf\_merge\_value(conf->exact_size, prev->exact_size, 1);
1029
1030     return NGX\_CONF\_OK;
1031 }
1032
1033
1034 static ngx\_int\_t
1035 ngx_http_autoindex_init(ngx\_conf\_t *cf)
1036 {
1037     ngx\_http\_handler\_pt *h;
1038     ngx\_http\_core\_main\_conf\_t *cmcf;
1039
1040     cmcf = ngx\_http\_conf\_get\_module\_main\_conf(cf, ngx\_http\_core\_module);
1041
1042     h = ngx\_array\_push(&cmcf->phases[NGX\_HTTP\_CONTENT\_PHASE].handlers);
1043     if (h == NULL) {
1044         return NGX\_ERROR;
1045     }
1046
1047     *h = ngx\_http\_autoindex\_handler;
1048
1049     return NGX\_OK;
1050 }

```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_browser\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_browser\\_commands](#)
- [ngx\\_http\\_browser\\_module](#)
- [ngx\\_http\\_browser\\_module\\_ctx](#)
- [ngx\\_http\\_browsers](#)
- [ngx\\_http\\_modern\\_browser\\_masks](#)

## Data types defined

- [ngx\\_http\\_browser\\_conf\\_t](#)
- [ngx\\_http\\_browser\\_variable\\_t](#)
- [ngx\\_http\\_modern\\_browser\\_mask\\_t](#)
- [ngx\\_http\\_modern\\_browser\\_t](#)

## Functions defined

- [ngx\\_http\\_ancient\\_browser](#)
- [ngx\\_http\\_ancient\\_browser\\_value](#)
- [ngx\\_http\\_browser](#)
- [ngx\\_http\\_browser\\_add\\_variable](#)
- [ngx\\_http\\_browser\\_create\\_conf](#)
- [ngx\\_http\\_browser\\_merge\\_conf](#)
- [ngx\\_http\\_browser\\_variable](#)
- [ngx\\_http\\_modern\\_browser](#)
- [ngx\\_http\\_modern\\_browser\\_sort](#)
- [ngx\\_http\\_modern\\_browser\\_value](#)
- [ngx\\_http\\_msie\\_variable](#)

## Macros defined

- [NGX\\_HTTP\\_ANCIENT\\_BROWSER](#)
- [NGX\\_HTTP\\_MODERN\\_BROWSER](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
```

```

5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 /*
14  * The module can check browser versions conforming to the following formats:
15  * X, X.X, X.X.X, and X.X.X.X. The maximum values of each format may be
16  * 4000, 4000.99, 4000.99.99, and 4000.99.99.99.
17  */
18
19
20 #define  NGX_HTTP_MODERN_BROWSER  0
21 #define  NGX_HTTP_ANCIENT_BROWSER 1
22
23
24 typedef struct {
25     u_char          browser[12];
26     size_t          skip;
27     size_t          add;
28     u_char          name[12];
29 } ngx_http_modern_browser_mask_t;
30
31
32 typedef struct {
33     ngx_uint_t      version;
34     size_t          skip;
35     size_t          add;
36     u_char          name[12];
37 } ngx_http_modern_browser_t;
38
39
40 typedef struct {
41     ngx_str_t       name;
42     ngx_http_get_variable_pt handler;
43     uintptr_t      data;
44 } ngx_http_browser_variable_t;
45
46
47 typedef struct {
48     ngx_array_t     *modern_browsers;
49     ngx_array_t     *ancient_browsers;
50     ngx_http_variable_value_t *modern_browser_value;
51     ngx_http_variable_value_t *ancient_browser_value;
52
53     unsigned        modern_unlisted_browsers:1;
54     unsigned        netscape4:1;
55 } ngx_http_browser_conf_t;
56
57
58 static ngx_int_t ngx_http_msie_variable(ngx_http_request_t *r,
59     ngx_http_variable_value_t *v, uintptr_t data);
60 static ngx_int_t ngx_http_browser_variable(ngx_http_request_t *r,
61     ngx_http_variable_value_t *v, uintptr_t data);
62
63 static ngx_uint_t ngx_http_browser(ngx_http_request_t *r,
64     ngx_http_browser_conf_t *cf);
65
66 static ngx_int_t ngx_http_browser_add_variable(ngx_conf_t *cf);
67 static void *ngx_http_browser_create_conf(ngx_conf_t *cf);
68 static char *ngx_http_browser_merge_conf(ngx_conf_t *cf, void *parent,
69     void *child);
70 static int ngx_libc_cdecl ngx_http_modern_browser_sort(const void *one,
71     const void *two);
72 static char *ngx_http_modern_browser(ngx_conf_t *cf, ngx_command_t *cmd,
73     void *conf);
74 static char *ngx_http_ancient_browser(ngx_conf_t *cf, ngx_command_t *cmd,
75     void *conf);
76 static char *ngx_http_modern_browser_value(ngx_conf_t *cf, ngx_command_t *cmd,
77     void *conf);
78 static char *ngx_http_ancient_browser_value(ngx_conf_t *cf, ngx_command_t *cmd,
79     void *conf);
80

```

```

81
82 static ngx_command_t  ngx_http_browser_commands[] = {
83
84     { ngx_string("modern_browser"),
85       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE12,
86       ngx_http_modern_browser,
87       NGX_HTTP_LOC_CONF_OFFSET,
88       0,
89       NULL },
90
91     { ngx_string("ancient_browser"),
92       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_1MORE,
93       ngx_http_ancient_browser,
94       NGX_HTTP_LOC_CONF_OFFSET,
95       0,
96       NULL },
97
98     { ngx_string("modern_browser_value"),
99       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
100      ngx_http_modern_browser_value,
101      NGX_HTTP_LOC_CONF_OFFSET,
102      0,
103      NULL },
104
105     { ngx_string("ancient_browser_value"),
106       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
107       ngx_http_ancient_browser_value,
108       NGX_HTTP_LOC_CONF_OFFSET,
109       0,
110       NULL },
111
112     ngx_null_command
113 };
114
115
116 static ngx_http_module_t  ngx_http_browser_module_ctx = {
117     ngx_http_browser_add_variable,      /* preconfiguration */
118     NULL,                                /* postconfiguration */
119
120     NULL,                                /* create main configuration */
121     NULL,                                /* init main configuration */
122
123     NULL,                                /* create server configuration */
124     NULL,                                /* merge server configuration */
125
126     ngx_http_browser_create_conf,       /* create location configuration */
127     ngx_http_browser_merge_conf        /* merge location configuration */
128 };
129
130
131 ngx_module_t  ngx_http_browser_module = {
132     NGX_MODULE_V1,
133     &ngx_http_browser_module_ctx,      /* module context */
134     ngx_http_browser_commands,         /* module directives */
135     NGX_HTTP_MODULE,                   /* module type */
136     NULL,                               /* init master */
137     NULL,                               /* init module */
138     NULL,                               /* init process */
139     NULL,                               /* init thread */
140     NULL,                               /* exit thread */
141     NULL,                               /* exit process */
142     NULL,                               /* exit master */
143     NGX_MODULE_V1_PADDING
144 };
145
146
147 static ngx_http_modern_browser_mask_t  ngx_http_modern_browser_masks[] = {
148
149     /* Opera must be the first browser to check */
150
151     /*
152     * "Opera/7.50 (X11; FreeBSD i386; U) [en]"
153     * "Mozilla/5.0 (X11; FreeBSD i386; U) Opera 7.50 [en]"
154     * "Mozilla/4.0 (compatible; MSIE 6.0; X11; FreeBSD i386) Opera 7.50 [en]"
155     * "Opera/8.0 (Windows NT 5.1; U; ru)"
156     * "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; en) Opera 8.0"

```

```

157     * "Opera/9.01 (X11; FreeBSD 6 i386; U; en)"
158     */
159
160     { "opera",
161       0,
162       sizeof("Opera ") - 1,
163       "Opera"},
164
165     /* "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)" */
166
167     { "msie",
168       sizeof("Mozilla/4.0 (compatible; ") - 1,
169       sizeof("MSIE ") - 1,
170       "MSIE "},
171
172     /*
173     * "Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.0.0) Gecko/20020610"
174     * "Mozilla/5.0 (Windows; U; Windows NT 5.1; ru-RU; rv:1.5) Gecko/20031006"
175     * "Mozilla/5.0 (Windows; U; Windows NT 5.1; ru-RU; rv:1.6) Gecko/20040206
176     *   Firefox/0.8"
177     * "Mozilla/5.0 (Windows; U; Windows NT 5.1; ru-RU; rv:1.7.8)
178     *   Gecko/20050511 Firefox/1.0.4"
179     * "Mozilla/5.0 (X11; U; FreeBSD i386; en-US; rv:1.8.0.5) Gecko/20060729
180     *   Firefox/1.5.0.5"
181     */
182
183     { "gecko",
184       sizeof("Mozilla/5.0 (") - 1,
185       sizeof("rv:") - 1,
186       "rv:"},
187
188     /*
189     * "Mozilla/5.0 (Macintosh; U; PPC Mac OS X; ru-ru) AppleWebKit/125.2
190     *   (KHTML, like Gecko) Safari/125.7"
191     * "Mozilla/5.0 (SymbianOS/9.1; U; en-us) AppleWebKit/413
192     *   (KHTML, like Gecko) Safari/413"
193     * "Mozilla/5.0 (Macintosh; U; PPC Mac OS X; en) AppleWebKit/418
194     *   (KHTML, like Gecko) Safari/417.9.3"
195     * "Mozilla/5.0 (Macintosh; U; PPC Mac OS X; ru-ru) AppleWebKit/418.8
196     *   (KHTML, like Gecko) Safari/419.3"
197     */
198
199     { "safari",
200       sizeof("Mozilla/5.0 (") - 1,
201       sizeof("Safari/") - 1,
202       "Safari/"},
203
204     /*
205     * "Mozilla/5.0 (compatible; Konqueror/3.1; Linux)"
206     * "Mozilla/5.0 (compatible; Konqueror/3.4; Linux) KHTML/3.4.2 (like Gecko)"
207     * "Mozilla/5.0 (compatible; Konqueror/3.5; FreeBSD) KHTML/3.5.1
208     *   (like Gecko)"
209     */
210
211     { "konqueror",
212       sizeof("Mozilla/5.0 (compatible; ") - 1,
213       sizeof("Konqueror/") - 1,
214       "Konqueror/"},
215
216     { "", 0, 0, "" }
217 };
218
219
220
221 static ngx_http_browser_variable_t ngx_http_browsers[] = {
222     { ngx_string("msie"), ngx_http_msie_variable, 0 },
223     { ngx_string("modern_browser"), ngx_http_browser_variable,
224       NGX_HTTP_MODERN_BROWSER },
225     { ngx_string("ancient_browser"), ngx_http_browser_variable,
226       NGX_HTTP_ANCIENT_BROWSER },
227     { ngx_null_string, NULL, 0 }
228 };
229
230
231 static ngx_int_t
232 ngx_http_browser_variable(ngx_http_request_t *r, ngx_http_variable_value_t *v,

```

```

233     uintptr_t data)
234 {
235     ngx_uint_t          rc;
236     ngx_http_browser_conf_t *cf;
237
238     cf = ngx_http_get_module_loc_conf(r, ngx_http_browser_module);
239
240     rc = ngx_http_browser(r, cf);
241
242     if (data == NGX_HTTP_MODERN_BROWSER && rc == NGX_HTTP_MODERN_BROWSER) {
243         *v = *cf->modern_browser_value;
244         return NGX_OK;
245     }
246
247     if (data == NGX_HTTP_ANCIENT_BROWSER && rc == NGX_HTTP_ANCIENT_BROWSER) {
248         *v = *cf->ancient_browser_value;
249         return NGX_OK;
250     }
251
252     *v = ngx_http_variable_null_value;
253     return NGX_OK;
254 }
255
256
257 static ngx_uint_t
258 ngx_http_browser(ngx_http_request_t *r, ngx_http_browser_conf_t *cf)
259 {
260     size_t          len;
261     u_char          *name, *ua, *last, c;
262     ngx_str_t       *ancient;
263     ngx_uint_t      i, version, ver, scale;
264     ngx_http_modern_browser_t *modern;
265
266     if (r->headers_in.user_agent == NULL) {
267         if (cf->modern_unlisted_browsers) {
268             return NGX_HTTP_MODERN_BROWSER;
269         }
270
271         return NGX_HTTP_ANCIENT_BROWSER;
272     }
273
274     ua = r->headers_in.user_agent->value.data;
275     len = r->headers_in.user_agent->value.len;
276     last = ua + len;
277
278     if (cf->modern_browsers) {
279         modern = cf->modern_browsers->elts;
280
281         for (i = 0; i < cf->modern_browsers->nelts; i++) {
282             name = ua + modern[i].skip;
283
284             if (name >= last) {
285                 continue;
286             }
287
288             name = (u_char *) ngx_strstr(name, modern[i].name);
289
290             if (name == NULL) {
291                 continue;
292             }
293
294             ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
295                 "browser: \"%s\"", name);
296
297             name += modern[i].add;
298
299             if (name >= last) {
300                 continue;
301             }
302
303             ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
304                 "version: \"%ui\" \"%s\"", modern[i].version, name);
305
306             version = 0;
307             ver = 0;
308             scale = 1000000;

```

```

309
310 while (name < last) {
311
312     c = *name++;
313
314     if (c >= '0' && c <= '9') {
315         ver = ver * 10 + (c - '0');
316         continue;
317     }
318
319     if (c == '.') {
320         version += ver * scale;
321
322         ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
323             "version: \"%ui\" \"%ui\"",
324             modern[i].version, version);
325
326         if (version > modern[i].version) {
327             return NGX_HTTP_MODERN_BROWSER;
328         }
329
330         ver = 0;
331         scale /= 100;
332         continue;
333     }
334
335     break;
336 }
337
338 version += ver * scale;
339
340 ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
341     "version: \"%ui\" \"%ui\"",
342     modern[i].version, version);
343
344 if (version >= modern[i].version) {
345     return NGX_HTTP_MODERN_BROWSER;
346 }
347
348 return NGX_HTTP_ANCIENT_BROWSER;
349 }
350
351 if (!cf->modern_unlisted_browsers) {
352     return NGX_HTTP_ANCIENT_BROWSER;
353 }
354 }
355
356 if (cf->netscape4) {
357     if (len > sizeof("Mozilla/4.72 ") - 1
358         && ngx_strncmp(ua, "Mozilla/", sizeof("Mozilla/") - 1) == 0
359         && ua[8] > '0' && ua[8] < '5')
360     {
361         return NGX_HTTP_ANCIENT_BROWSER;
362     }
363 }
364
365 if (cf->ancient_browsers) {
366     ancient = cf->ancient_browsers->elts;
367
368     for (i = 0; i < cf->ancient_browsers->nelts; i++) {
369         if (len >= ancient[i].len
370             && ngx_strstr(ua, ancient[i].data) != NULL)
371         {
372             return NGX_HTTP_ANCIENT_BROWSER;
373         }
374     }
375 }
376
377 if (cf->modern_unlisted_browsers) {
378     return NGX_HTTP_MODERN_BROWSER;
379 }
380
381 return NGX_HTTP_ANCIENT_BROWSER;
382 }
383
384

```



```

385 static ngx_int_t
386 ngx_http_msie_variable(ngx_http_request_t *r, ngx_http_variable_value_t *v,
387     uintptr_t data)
388 {
389     if (r->headers_in.msie) {
390         *v = ngx_http_variable_true_value;
391         return NGX_OK;
392     }
393
394     *v = ngx_http_variable_null_value;
395     return NGX_OK;
396 }
397
398
399 static ngx_int_t
400 ngx_http_browser_add_variable(ngx_conf_t *cf)
401 {
402     ngx_http_browser_variable_t *var;
403     ngx_http_variable_t *v;
404
405     for (var = ngx_http_browsers; var->name.len; var++) {
406
407         v = ngx_http_add_variable(cf, &var->name, NGX_HTTP_VAR_CHANGEABLE);
408         if (v == NULL) {
409             return NGX_ERROR;
410         }
411
412         v->get_handler = var->handler;
413         v->data = var->data;
414     }
415
416     return NGX_OK;
417 }
418
419
420 static void *
421 ngx_http_browser_create_conf(ngx_conf_t *cf)
422 {
423     ngx_http_browser_conf_t *conf;
424
425     conf = ngx_palloc(cf->pool, sizeof(ngx_http_browser_conf_t));
426     if (conf == NULL) {
427         return NULL;
428     }
429
430     /*
431     * set by ngx_palloc():
432     *
433     *     conf->modern_browsers = NULL;
434     *     conf->ancient_browsers = NULL;
435     *     conf->modern_browser_value = NULL;
436     *     conf->ancient_browser_value = NULL;
437     *
438     *     conf->modern_unlisted_browsers = 0;
439     *     conf->netscape4 = 0;
440     */
441
442     return conf;
443 }
444
445
446 static char *
447 ngx_http_browser_merge_conf(ngx_conf_t *cf, void *parent, void *child)
448 {
449     ngx_http_browser_conf_t *prev = parent;
450     ngx_http_browser_conf_t *conf = child;
451
452     ngx_uint_t i, n;
453     ngx_http_modern_browser_t *browsers, *opera;
454
455     /*
456     * At the merge the skip field is used to store the browser slot,
457     * it will be used in sorting and then will overwritten
458     * with a real skip value. The zero value means Opera.
459     */
460

```

```

461 if (conf->modern_browsers == NULL && conf->modern_unlisted_browsers == 0) {
462     conf->modern_browsers = prev->modern_browsers;
463     conf->modern_unlisted_browsers = prev->modern_unlisted_browsers;
464
465 } else if (conf->modern_browsers != NULL) {
466     browsers = conf->modern_browsers->elts;
467
468     for (i = 0; i < conf->modern_browsers->nelts; i++) {
469         if (browsers[i].skip == 0) {
470             goto found;
471         }
472     }
473
474     /*
475     * Opera may contain MSIE string, so if Opera was not enumerated
476     * as modern browsers, then add it and set a unreachable version
477     */
478
479     opera = ngx_array_push(conf->modern_browsers);
480     if (opera == NULL) {
481         return NGX_CONF_ERROR;
482     }
483
484     opera->skip = 0;
485     opera->version = 4001000000U;
486
487     browsers = conf->modern_browsers->elts;
488
489 found:
490
491     ngx_qsort(browsers, (size_t) conf->modern_browsers->nelts,
492             sizeof(ngx_http_modern_browser_t),
493             ngx_http_modern_browser_sort);
494
495     for (i = 0; i < conf->modern_browsers->nelts; i++) {
496         n = browsers[i].skip;
497
498         browsers[i].skip = ngx_http_modern_browser_masks[n].skip;
499         browsers[i].add = ngx_http_modern_browser_masks[n].add;
500         (void) ngx_cpystn(browsers[i].name,
501                         ngx_http_modern_browser_masks[n].name, 12);
502     }
503 }
504
505 if (conf->ancient_browsers == NULL && conf->netscape4 == 0) {
506     conf->ancient_browsers = prev->ancient_browsers;
507     conf->netscape4 = prev->netscape4;
508 }
509
510 if (conf->modern_browser_value == NULL) {
511     conf->modern_browser_value = prev->modern_browser_value;
512 }
513
514 if (conf->modern_browser_value == NULL) {
515     conf->modern_browser_value = &ngx_http_variable_true_value;
516 }
517
518 if (conf->ancient_browser_value == NULL) {
519     conf->ancient_browser_value = prev->ancient_browser_value;
520 }
521
522 if (conf->ancient_browser_value == NULL) {
523     conf->ancient_browser_value = &ngx_http_variable_true_value;
524 }
525
526 return NGX_CONF_OK;
527 }
528
529
530 static int ngx_libc_cdecl
531 ngx_http_modern_browser_sort(const void *one, const void *two)
532 {
533     ngx_http_modern_browser_t *first = (ngx_http_modern_browser_t *) one;
534     ngx_http_modern_browser_t *second = (ngx_http_modern_browser_t *) two;
535
536     return (first->skip - second->skip);

```

```

537 }
538
539
540 static char *
541 ngx_http_modern_browser(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
542 {
543     ngx_http_browser_conf_t *bcf = conf;
544
545     u_char                c;
546     ngx_str_t             *value;
547     ngx_uint_t            i, n, version, ver, scale;
548     ngx_http_modern_browser_t *browser;
549     ngx_http_modern_browser_mask_t *mask;
550
551     value = cf->args->elts;
552
553     if (cf->args->nelts == 2) {
554         if (ngx_strcmp(value[1].data, "unlisted") == 0) {
555             bcf->modern_unlisted_browsers = 1;
556             return NGX_CONF_OK;
557         }
558
559         return NGX_CONF_ERROR;
560     }
561
562     if (bcf->modern_browsers == NULL) {
563         bcf->modern_browsers = ngx_array_create(cf->pool, 5,
564                                                 sizeof(ngx_http_modern_browser_t));
565         if (bcf->modern_browsers == NULL) {
566             return NGX_CONF_ERROR;
567         }
568     }
569
570     browser = ngx_array_push(bcf->modern_browsers);
571     if (browser == NULL) {
572         return NGX_CONF_ERROR;
573     }
574
575     mask = ngx_http_modern_browser_masks;
576
577     for (n = 0; mask[n].browser[0] != '\0'; n++) {
578         if (ngx_strcasecmp(mask[n].browser, value[1].data) == 0) {
579             goto found;
580         }
581     }
582
583     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
584                       "unknown browser name \"%V\"", &value[1]);
585
586     return NGX_CONF_ERROR;
587
588 found:
589
590     /*
591     * at this stage the skip field is used to store the browser slot,
592     * it will be used in sorting in merge stage and then will overwritten
593     * with a real value
594     */
595
596     browser->skip = n;
597
598     version = 0;
599     ver = 0;
600     scale = 1000000;
601
602     for (i = 0; i < value[2].len; i++) {
603
604         c = value[2].data[i];
605
606         if (c >= '0' && c <= '9') {
607             ver = ver * 10 + (c - '0');
608             continue;
609         }
610
611         if (c == '.') {
612             version += ver * scale;

```

```

613         ver = 0;
614         scale /= 100;
615         continue;
616     }
617
618     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
619         "invalid browser version \"%V\"", &value[2]);
620
621     return NGX_CONF_ERROR;
622 }
623
624 version += ver * scale;
625
626 browser->version = version;
627
628 return NGX_CONF_OK;
629 }
630
631
632 static char *
633 ngx_http_ancient_browser(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
634 {
635     ngx_http_browser_conf_t *bcf = conf;
636
637     ngx_str_t    *value, *browser;
638     ngx_uint_t    i;
639
640     value = cf->args->elts;
641
642     for (i = 1; i < cf->args->nelts; i++) {
643         if (ngx_strcmp(value[i].data, "netscape4") == 0) {
644             bcf->netscape4 = 1;
645             continue;
646         }
647
648         if (bcf->ancient_browsers == NULL) {
649             bcf->ancient_browsers = ngx_array_create(cf->pool, 4,
650                 sizeof(ngx_str_t));
651             if (bcf->ancient_browsers == NULL) {
652                 return NGX_CONF_ERROR;
653             }
654         }
655
656         browser = ngx_array_push(bcf->ancient_browsers);
657         if (browser == NULL) {
658             return NGX_CONF_ERROR;
659         }
660
661         *browser = value[i];
662     }
663
664     return NGX_CONF_OK;
665 }
666
667
668 static char *
669 ngx_http_modern_browser_value(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
670 {
671     ngx_http_browser_conf_t *bcf = conf;
672
673     ngx_str_t    *value;
674
675     bcf->modern_browser_value = ngx_palloc(cf->pool,
676         sizeof(ngx_http_variable_value_t));
677     if (bcf->modern_browser_value == NULL) {
678         return NGX_CONF_ERROR;
679     }
680
681     value = cf->args->elts;
682
683     bcf->modern_browser_value->len = value[1].len;
684     bcf->modern_browser_value->valid = 1;
685     bcf->modern_browser_value->no_cacheable = 0;
686     bcf->modern_browser_value->not_found = 0;
687     bcf->modern_browser_value->data = value[1].data;
688

```

```
689     return NGX\_CONF\_OK;  
690 }  
691  
692  
693 static char *  
694 ngx_http_ancient_browser_value(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)  
695 {  
696     ngx\_http\_browser\_conf\_t *bcf = conf;  
697  
698     ngx\_str\_t *value;  
699  
700     bcf->ancient_browser_value = ngx\_palloc(cf->pool,  
701                                             sizeof(ngx\_http\_variable\_value\_t));  
702     if (bcf->ancient_browser_value == NULL) {  
703         return NGX\_CONF\_ERROR;  
704     }  
705  
706     value = cf->args->elts;  
707  
708     bcf->ancient_browser_value->len = value[1].len;  
709     bcf->ancient_browser_value->valid = 1;  
710     bcf->ancient_browser_value->no_cacheable = 0;  
711     bcf->ancient_browser_value->not_found = 0;  
712     bcf->ancient_browser_value->data = value[1].data;  
713  
714     return NGX\_CONF\_OK;  
715 }
```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_charset\_filter\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_charset\\_default\\_types](#)
- [ngx\\_http\\_charset\\_filter\\_commands](#)
- [ngx\\_http\\_charset\\_filter\\_module](#)
- [ngx\\_http\\_charset\\_filter\\_module\\_ctx](#)
- [ngx\\_http\\_next\\_body\\_filter](#)
- [ngx\\_http\\_next\\_header\\_filter](#)

## Data types defined

- [ngx\\_http\\_charset\\_conf\\_ctx\\_t](#)
- [ngx\\_http\\_charset\\_ctx\\_t](#)
- [ngx\\_http\\_charset\\_loc\\_conf\\_t](#)
- [ngx\\_http\\_charset\\_main\\_conf\\_t](#)
- [ngx\\_http\\_charset\\_recode\\_t](#)
- [ngx\\_http\\_charset\\_t](#)
- [ngx\\_http\\_charset\\_tables\\_t](#)

## Functions defined

- [ngx\\_http\\_add\\_charset](#)
- [ngx\\_http\\_charset\\_body\\_filter](#)
- [ngx\\_http\\_charset\\_create\\_loc\\_conf](#)
- [ngx\\_http\\_charset\\_create\\_main\\_conf](#)
- [ngx\\_http\\_charset\\_ctx](#)
- [ngx\\_http\\_charset\\_get\\_buf](#)
- [ngx\\_http\\_charset\\_get\\_buffer](#)
- [ngx\\_http\\_charset\\_header\\_filter](#)
- [ngx\\_http\\_charset\\_map](#)
- [ngx\\_http\\_charset\\_map\\_block](#)
- [ngx\\_http\\_charset\\_merge\\_loc\\_conf](#)
- [ngx\\_http\\_charset\\_postconfiguration](#)
- [ngx\\_http\\_charset\\_recode](#)
- [ngx\\_http\\_charset\\_recode\\_from\\_utf8](#)

- [ngx\\_http\\_charset\\_recode\\_to\\_utf8](#)
- [ngx\\_http\\_destination\\_charset](#)
- [ngx\\_http\\_get\\_charset](#)
- [ngx\\_http\\_main\\_request\\_charset](#)
- [ngx\\_http\\_set\\_charset](#)
- [ngx\\_http\\_set\\_charset\\_slot](#)
- [ngx\\_http\\_source\\_charset](#)

## Macros defined

- [NGX\\_HTML\\_ENTITY\\_LEN](#)
- [NGX\\_HTTP\\_CHARSET\\_OFF](#)
- [NGX\\_HTTP\\_CHARSET\\_VAR](#)
- [NGX\\_HTTP\\_NO\\_CHARSET](#)
- [NGX\\_UTF\\_LEN](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 #define NGX_HTTP_CHARSET_OFF    -2
14 #define NGX_HTTP_NO_CHARSET    -3
15 #define NGX_HTTP_CHARSET_VAR   0x10000
16
17 /* 1 byte length and up to 3 bytes for the UTF-8 encoding of the UCS-2 */
18 #define NGX_UTF_LEN            4
19
20 #define NGX_HTML_ENTITY_LEN    (sizeof("&#1114111;") - 1)
21
22
23 typedef struct {
24     u_char                **tables;
25     ngx_str_t             name;
26
27     unsigned               length:16;
28     unsigned               utf8:1;
29 } ngx_http_charset_t;
30
31
32 typedef struct {
33     ngx_int_t             src;
34     ngx_int_t             dst;
35 } ngx_http_charset_recode_t;
36
37
38 typedef struct {
39     ngx_int_t             src;
40     ngx_int_t             dst;
41     u_char                *src2dst;

```

```

42     u_char                                *dst2src;
43 } ngx_http_charset_tables_t;
44
45
46 typedef struct {
47     ngx_array_t                            charsets;        /* ngx_http_charset_t */
48     ngx_array_t                            tables;          /* ngx_http_charset_tables_t */
49     ngx_array_t                            recodes;         /* ngx_http_charset_recode_t */
50 } ngx_http_charset_main_conf_t;
51
52
53 typedef struct {
54     ngx_int_t                               charset;
55     ngx_int_t                               source_charset;
56     ngx_flag_t                              override_charset;
57
58     ngx_hash_t                              types;
59     ngx_array_t                             *types_keys;
60 } ngx_http_charset_loc_conf_t;
61
62
63 typedef struct {
64     u_char                                *table;
65     ngx_int_t                             charset;
66     ngx_str_t                             charset_name;
67
68     ngx_chain_t                            *busy;
69     ngx_chain_t                            *free_bufs;
70     ngx_chain_t                            *free_buffers;
71
72     size_t                                 saved_len;
73     u_char                                 saved[NGX_UTF_LEN];
74
75     unsigned                               length:16;
76     unsigned                               from_utf8:1;
77     unsigned                               to_utf8:1;
78 } ngx_http_charset_ctx_t;
79
80
81 typedef struct {
82     ngx_http_charset_tables_t             *table;
83     ngx_http_charset_t                   *charset;
84     ngx_uint_t                            characters;
85 } ngx_http_charset_conf_ctx_t;
86
87
88 static ngx_int_t ngx_http_destination_charset(ngx_http_request_t *r,
89     ngx_str_t *name);
90 static ngx_int_t ngx_http_main_request_charset(ngx_http_request_t *r,
91     ngx_str_t *name);
92 static ngx_int_t ngx_http_source_charset(ngx_http_request_t *r,
93     ngx_str_t *name);
94 static ngx_int_t ngx_http_get_charset(ngx_http_request_t *r, ngx_str_t *name);
95 static ngx_inline void ngx_http_set_charset(ngx_http_request_t *r,
96     ngx_str_t *charset);
97 static ngx_int_t ngx_http_charset_ctx(ngx_http_request_t *r,
98     ngx_http_charset_t *charsets, ngx_int_t charset, ngx_int_t source_charset);
99 static ngx_uint_t ngx_http_charset_recode(ngx_buf_t *b, u_char *table);
100 static ngx_chain_t *ngx_http_charset_recode_from_utf8(ngx_pool_t *pool,
101     ngx_buf_t *buf, ngx_http_charset_ctx_t *ctx);
102 static ngx_chain_t *ngx_http_charset_recode_to_utf8(ngx_pool_t *pool,
103     ngx_buf_t *buf, ngx_http_charset_ctx_t *ctx);
104
105 static ngx_chain_t *ngx_http_charset_get_buf(ngx_pool_t *pool,
106     ngx_http_charset_ctx_t *ctx);
107 static ngx_chain_t *ngx_http_charset_get_buffer(ngx_pool_t *pool,
108     ngx_http_charset_ctx_t *ctx, size_t size);
109
110 static char *ngx_http_charset_map_block(ngx_conf_t *cf, ngx_command_t *cmd,
111     void *conf);
112 static char *ngx_http_charset_map(ngx_conf_t *cf, ngx_command_t *dummy,
113     void *conf);
114
115 static char *ngx_http_set_charset_slot(ngx_conf_t *cf, ngx_command_t *cmd,
116     void *conf);
117 static ngx_int_t ngx_http_add_charset(ngx_array_t *charsets, ngx_str_t *name);

```



```

118 static void *ngx_http_charset_create_main_conf(ngx_conf_t *cf);
119 static void *ngx_http_charset_create_loc_conf(ngx_conf_t *cf);
120 static char *ngx_http_charset_merge_loc_conf(ngx_conf_t *cf,
121 void *parent, void *child);
122 static ngx_int_t ngx_http_charset_postconfiguration(ngx_conf_t *cf);
123
124
125
126 ngx_str_t ngx_http_charset_default_types[] = {
127     ngx_string("text/html"),
128     ngx_string("text/xml"),
129     ngx_string("text/plain"),
130     ngx_string("text/vnd.wap.wml"),
131     ngx_string("application/javascript"),
132     ngx_string("application/rss+xml"),
133     ngx_null_string
134 };
135
136
137 static ngx_command_t ngx_http_charset_filter_commands[] = {
138
139     { ngx_string("charset"),
140       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF
141         |NGX_HTTP_LIF_CONF|NGX_CONF_TAKE1,
142       ngx_http_set_charset_slot,
143       NGX_HTTP_LOC_CONF_OFFSET,
144       offsetof(ngx_http_charset_loc_conf_t, charset),
145       NULL },
146
147     { ngx_string("source_charset"),
148       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF
149         |NGX_HTTP_LIF_CONF|NGX_CONF_TAKE1,
150       ngx_http_set_charset_slot,
151       NGX_HTTP_LOC_CONF_OFFSET,
152       offsetof(ngx_http_charset_loc_conf_t, source_charset),
153       NULL },
154
155     { ngx_string("override_charset"),
156       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF
157         |NGX_HTTP_LIF_CONF|NGX_CONF_FLAG,
158       ngx_conf_set_flag_slot,
159       NGX_HTTP_LOC_CONF_OFFSET,
160       offsetof(ngx_http_charset_loc_conf_t, override_charset),
161       NULL },
162
163     { ngx_string("charset_types"),
164       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_1MORE,
165       ngx_http_types_slot,
166       NGX_HTTP_LOC_CONF_OFFSET,
167       offsetof(ngx_http_charset_loc_conf_t, types_keys),
168       &ngx_http_charset_default_types[0] },
169
170     { ngx_string("charset_map"),
171       NGX_HTTP_MAIN_CONF|NGX_CONF_BLOCK|NGX_CONF_TAKE2,
172       ngx_http_charset_map_block,
173       NGX_HTTP_MAIN_CONF_OFFSET,
174       0,
175       NULL },
176
177     ngx_null_command
178 };
179
180
181 static ngx_http_module_t ngx_http_charset_filter_module_ctx = {
182     NULL, /* preconfiguration */
183     ngx_http_charset_postconfiguration, /* postconfiguration */
184
185     ngx_http_charset_create_main_conf, /* create main configuration */
186     NULL, /* init main configuration */
187
188     NULL, /* create server configuration */
189     NULL, /* merge server configuration */
190
191     ngx_http_charset_create_loc_conf, /* create location configuration */
192     ngx_http_charset_merge_loc_conf /* merge location configuration */
193 };

```

```

194
195
196 ngx_module_t ngx_http_charset_filter_module = {
197     NGX_MODULE_V1,
198     &ngx_http_charset_filter_module_ctx, /* module context */
199     ngx_http_charset_filter_commands, /* module directives */
200     NGX_HTTP_MODULE, /* module type */
201     NULL, /* init master */
202     NULL, /* init module */
203     NULL, /* init process */
204     NULL, /* init thread */
205     NULL, /* exit thread */
206     NULL, /* exit process */
207     NULL, /* exit master */
208     NGX_MODULE_V1_PADDING
209 };
210
211
212 static ngx_http_output_header_filter_pt ngx_http_next_header_filter;
213 static ngx_http_output_body_filter_pt ngx_http_next_body_filter;
214
215
216 static ngx_int_t
217 ngx_http_charset_header_filter(ngx_http_request_t *r)
218 {
219     ngx_int_t charset, source_charset;
220     ngx_str_t dst, src;
221     ngx_http_charset_t *charsets;
222     ngx_http_charset_main_conf_t *mcf;
223
224     if (r == r->main) {
225         charset = ngx_http_destination_charset(r, &dst);
226
227     } else {
228         charset = ngx_http_main_request_charset(r, &dst);
229     }
230
231     if (charset == NGX_ERROR) {
232         return NGX_ERROR;
233     }
234
235     if (charset == NGX_DECLINED) {
236         return ngx_http_next_header_filter(r);
237     }
238
239     /* charset: charset index or NGX_HTTP_NO_CHARSET */
240
241     source_charset = ngx_http_source_charset(r, &src);
242
243     if (source_charset == NGX_ERROR) {
244         return NGX_ERROR;
245     }
246
247     /*
248     * source_charset: charset index, NGX_HTTP_NO_CHARSET,
249     * or NGX_HTTP_CHARSET_OFF
250     */
251
252     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
253         "charset: \"%V\" > \"%V\"", &src, &dst);
254
255     if (source_charset == NGX_HTTP_CHARSET_OFF) {
256         ngx_http_set_charset(r, &dst);
257
258         return ngx_http_next_header_filter(r);
259     }
260
261     if (charset == NGX_HTTP_NO_CHARSET
262         || source_charset == NGX_HTTP_NO_CHARSET)
263     {
264         if (source_charset != charset
265             || ngx_strncasecmp(dst.data, src.data, dst.len) != 0)
266         {
267             goto no_charset_map;
268         }
269

```

```

270     ngx_http_set_charset(r, &dst);
271
272     return ngx_http_next_header_filter(r);
273 }
274
275 if (source_charset == charset) {
276     r->headers_out.content_type.len = r->headers_out.content_type_len;
277
278     ngx_http_set_charset(r, &dst);
279
280     return ngx_http_next_header_filter(r);
281 }
282
283 /* source_charset != charset */
284
285 if (r->headers_out.content_encoding
286     && r->headers_out.content_encoding->value.len)
287 {
288     return ngx_http_next_header_filter(r);
289 }
290
291 mcf = ngx_http_get_module_main_conf(r, ngx_http_charset_filter_module);
292 charsets = mcf->charsets.elts;
293
294 if (charsets[source_charset].tables == NULL
295     || charsets[source_charset].tables[charset] == NULL)
296 {
297     goto no_charset_map;
298 }
299
300 r->headers_out.content_type.len = r->headers_out.content_type_len;
301
302 ngx_http_set_charset(r, &dst);
303
304 return ngx_http_charset_ctx(r, charsets, charset, source_charset);
305
306 no_charset_map:
307
308 ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
309             "no \"charset_map\" between the charsets \"%V\" and \"%V\"",
310             &src, &dst);
311
312 return ngx_http_next_header_filter(r);
313 }
314
315
316 static ngx_int_t
317 ngx_http_destination_charset(ngx_http_request_t *r, ngx_str_t *name)
318 {
319     ngx_int_t          charset;
320     ngx_http_charset_t *charsets;
321     ngx_http_variable_value_t *vv;
322     ngx_http_charset_loc_conf_t *mlcf;
323     ngx_http_charset_main_conf_t *mcf;
324
325     if (r->headers_out.content_type.len == 0) {
326         return NGX_DECLINED;
327     }
328
329     if (r->headers_out.override_charset
330         && r->headers_out.override_charset->len)
331     {
332         *name = r->headers_out.override_charset;
333
334         charset = ngx_http_get_charset(r, name);
335
336         if (charset != NGX_HTTP_NO_CHARSET) {
337             return charset;
338         }
339
340         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
341             "unknown charset \"%V\" to override", name);
342
343         return NGX_DECLINED;
344     }
345

```

```

346     mpcf = ngx_http_get_module_loc_conf(r, ngx_http_charset_filter_module);
347     charset = mpcf->charset;
348
349     if (charset == NGX_HTTP_CHARSET_OFF) {
350         return NGX_DECLINED;
351     }
352
353     if (r->headers_out.charset.len) {
354         if (mpcf->override_charset == 0) {
355             return NGX_DECLINED;
356         }
357
358     } else {
359         if (ngx_http_test_content_type(r, &mpcf->types) == NULL) {
360             return NGX_DECLINED;
361         }
362     }
363
364     if (charset < NGX_HTTP_CHARSET_VAR) {
365         mcf = ngx_http_get_module_main_conf(r, ngx_http_charset_filter_module);
366         charsets = mcf->charsets.elts;
367         *name = charsets[charset].name;
368         return charset;
369     }
370
371     vv = ngx_http_get_indexed_variable(r, charset - NGX_HTTP_CHARSET_VAR);
372
373     if (vv == NULL || vv->not_found) {
374         return NGX_ERROR;
375     }
376
377     name->len = vv->len;
378     name->data = vv->data;
379
380     return ngx_http_get_charset(r, name);
381 }
382
383
384 static ngx_int_t
385 ngx_http_main_request_charset(ngx_http_request_t *r, ngx_str_t *src)
386 {
387     ngx_int_t         charset;
388     ngx_str_t         *main_charset;
389     ngx_http_charset_ctx_t *ctx;
390
391     ctx = ngx_http_get_module_ctx(r->main, ngx_http_charset_filter_module);
392
393     if (ctx) {
394         *src = ctx->charset_name;
395         return ctx->charset;
396     }
397
398     main_charset = &r->main->headers_out.charset;
399
400     if (main_charset->len == 0) {
401         return NGX_DECLINED;
402     }
403
404     ctx = ngx_palloc(r->pool, sizeof(ngx_http_charset_ctx_t));
405     if (ctx == NULL) {
406         return NGX_ERROR;
407     }
408
409     ngx_http_set_ctx(r->main, ctx, ngx_http_charset_filter_module);
410
411     charset = ngx_http_get_charset(r, main_charset);
412
413     ctx->charset = charset;
414     ctx->charset_name = *main_charset;
415     *src = *main_charset;
416
417     return charset;
418 }
419
420
421 static ngx_int_t

```

```

422 ngx_http_source_charset(ngx_http_request_t *r, ngx_str_t *name)
423 {
424     ngx_int_t charset;
425     ngx_http_charset_t *charsets;
426     ngx_http_variable_value_t *vv;
427     ngx_http_charset_loc_conf_t *lcf;
428     ngx_http_charset_main_conf_t *mcf;
429
430     if (r->headers_out.charset.len) {
431         *name = r->headers_out.charset;
432         return ngx_http_get_charset(r, name);
433     }
434
435     lcf = ngx_http_get_module_loc_conf(r, ngx_http_charset_filter_module);
436
437     charset = lcf->source_charset;
438
439     if (charset == NGX_HTTP_CHARSET_OFF) {
440         name->len = 0;
441         return charset;
442     }
443
444     if (charset < NGX_HTTP_CHARSET_VAR) {
445         mcf = ngx_http_get_module_main_conf(r, ngx_http_charset_filter_module);
446         charsets = mcf->charsets.elts;
447         *name = charsets[charset].name;
448         return charset;
449     }
450
451     vv = ngx_http_get_indexed_variable(r, charset - NGX_HTTP_CHARSET_VAR);
452
453     if (vv == NULL || vv->not_found) {
454         return NGX_ERROR;
455     }
456
457     name->len = vv->len;
458     name->data = vv->data;
459
460     return ngx_http_get_charset(r, name);
461 }
462
463
464 static ngx_int_t
465 ngx_http_get_charset(ngx_http_request_t *r, ngx_str_t *name)
466 {
467     ngx_uint_t i, n;
468     ngx_http_charset_t *charset;
469     ngx_http_charset_main_conf_t *mcf;
470
471     mcf = ngx_http_get_module_main_conf(r, ngx_http_charset_filter_module);
472
473     charset = mcf->charsets.elts;
474     n = mcf->charsets.nelts;
475
476     for (i = 0; i < n; i++) {
477         if (charset[i].name.len != name->len) {
478             continue;
479         }
480
481         if (ngx_strncasecmp(charset[i].name.data, name->data, name->len) == 0) {
482             return i;
483         }
484     }
485
486     return NGX_HTTP_NO_CHARSET;
487 }
488
489
490 static ngx_inline void
491 ngx_http_set_charset(ngx_http_request_t *r, ngx_str_t *charset)
492 {
493     if (r != r->main) {
494         return;
495     }
496
497     if (r->headers_out.status == NGX_HTTP_MOVED_PERMANENTLY

```

```

498     || r->headers_out.status == NGX\_HTTP\_MOVED\_TEMPORARILY)
499     {
500         /*
501          * do not set charset for the redirect because NN 4.x
502          * use this charset instead of the next page charset
503          */
504
505         r->headers_out.charset.len = 0;
506         return;
507     }
508
509     r->headers_out.charset = *charset;
510 }
511
512
513 static ngx\_int\_t
514 ngx\_http\_charset\_ctx(ngx\_http\_request\_t *r, ngx\_http\_charset\_t *charsets,
515 ngx\_int\_t charset, ngx\_int\_t source_charset)
516 {
517     ngx\_http\_charset\_ctx\_t *ctx;
518
519     ctx = ngx\_palloc(r->pool, sizeof(ngx\_http\_charset\_ctx\_t));
520     if (ctx == NULL) {
521         return NGX\_ERROR;
522     }
523
524     ngx\_http\_set\_ctx(r, ctx, ngx\_http\_charset\_filter\_module);
525
526     ctx->table = charsets[source_charset].tables[charset];
527     ctx->charset = charset;
528     ctx->charset_name = charsets[charset].name;
529     ctx->length = charsets[charset].length;
530     ctx->from_utf8 = charsets[source_charset].utf8;
531     ctx->to_utf8 = charsets[charset].utf8;
532
533     r->filter_need_in_memory = 1;
534
535     if ((ctx->to_utf8 || ctx->from_utf8) && r == r->main) {
536         ngx\_http\_clear\_content\_length(r);
537     } else {
538         r->filter_need_temporary = 1;
539     }
540
541     return ngx\_http\_next\_header\_filter(r);
542 }
543
544
545
546 static ngx\_int\_t
547 ngx\_http\_charset\_body\_filter(ngx\_http\_request\_t *r, ngx\_chain\_t *in)
548 {
549     ngx\_int\_t          rc;
550     ngx\_buf\_t         *b;
551     ngx\_chain\_t      *c1, *out, **ll;
552     ngx\_http\_charset\_ctx\_t *ctx;
553
554     ctx = ngx\_http\_get\_module\_ctx(r, ngx\_http\_charset\_filter\_module);
555
556     if (ctx == NULL || ctx->table == NULL) {
557         return ngx\_http\_next\_body\_filter(r, in);
558     }
559
560     if ((ctx->to_utf8 || ctx->from_utf8) || ctx->busy) {
561
562         out = NULL;
563         ll = &out;
564
565         for (c1 = in; c1; c1 = c1->next) {
566             b = c1->buf;
567
568             if (ngx\_buf\_size(b) == 0) {
569
570                 *ll = ngx\_alloc\_chain\_link(r->pool);
571                 if (*ll == NULL) {
572                     return NGX\_ERROR;
573                 }

```

```

574         (*ll)->buf = b;
575         (*ll)->next = NULL;
576
577         ll = &(*ll)->next;
578
579         continue;
580     }
581
582     if (ctx->to_utf8) {
583         *ll = ngx_http_charset_recode_to_utf8(r->pool, b, ctx);
584     } else {
585         *ll = ngx_http_charset_recode_from_utf8(r->pool, b, ctx);
586     }
587
588     if (*ll == NULL) {
589         return NGX_ERROR;
590     }
591
592     while (*ll) {
593         ll = &(*ll)->next;
594     }
595
596     rc = ngx_http_next_body_filter(r, out);
597
598     if (out) {
599         if (ctx->busy == NULL) {
600             ctx->busy = out;
601         } else {
602             for (cl = ctx->busy; cl->next; cl = cl->next) { /* void */ }
603             cl->next = out;
604         }
605     }
606
607     while (ctx->busy) {
608         cl = ctx->busy;
609         b = cl->buf;
610
611         if (ngx_buf_size(b) != 0) {
612             break;
613         }
614
615         ctx->busy = cl->next;
616
617         if (b->tag != (ngx_buf_tag_t) &ngx_http_charset_filter_module) {
618             continue;
619         }
620
621         if (b->shadow) {
622             b->shadow->pos = b->shadow->last;
623         }
624
625         if (b->pos) {
626             cl->next = ctx->free_buffers;
627             ctx->free_buffers = cl;
628             continue;
629         }
630
631         cl->next = ctx->free_bufs;
632         ctx->free_bufs = cl;
633     }
634
635     return rc;
636 }
637
638 for (cl = in; cl; cl = cl->next) {
639     (void) ngx_http_charset_recode(cl->buf, ctx->table);
640 }
641
642 return ngx_http_next_body_filter(r, in);
643 }
644
645 }
646
647 }
648
649

```

```

650 static ngx_uint_t
651 ngx_http_charset_recode(ngx_buf_t *b, u_char *table)
652 {
653     u_char *p, *last;
654
655     last = b->last;
656
657     for (p = b->pos; p < last; p++) {
658         if (*p != table[*p]) {
659             goto recode;
660         }
661     }
662
663     return 0;
664
665 recode:
666
667     do {
668         if (*p != table[*p]) {
669             *p = table[*p];
670         }
671
672         p++;
673
674     } while (p < last);
675
676     b->in_file = 0;
677
678     return 1;
679 }
680
681
682
683
684 static ngx_chain_t *
685 ngx_http_charset_recode_from_utf8(ngx_pool_t *pool, ngx_buf_t *buf,
686 ngx_http_charset_ctx_t *ctx)
687 {
688     size_t len, size;
689     u_char c, *p, *src, *dst, *saved, **table;
690     uint32_t n;
691     ngx_buf_t *b;
692     ngx_uint_t i;
693     ngx_chain_t *out, *cl, **ll;
694
695     src = buf->pos;
696
697     if (ctx->saved_len == 0) {
698
699         for ( /* void */ ; src < buf->last; src++) {
700
701             if (*src < 0x80) {
702                 continue;
703             }
704
705             len = src - buf->pos;
706
707             if (len > 512) {
708                 out = ngx_http_charset_get_buf(pool, ctx);
709                 if (out == NULL) {
710                     return NULL;
711                 }
712
713                 b = out->buf;
714
715                 b->temporary = buf->temporary;
716                 b->memory = buf->memory;
717                 b->mmap = buf->mmap;
718                 b->flush = buf->flush;
719
720                 b->pos = buf->pos;
721                 b->last = src;
722
723                 out->buf = b;
724                 out->next = NULL;
725

```



```

726         size = buf->last - src;
727
728         saved = src;
729         n = ngx_utf8_decode(&saved, size);
730
731         if (n == 0xffffffff) {
732             /* incomplete UTF-8 symbol */
733
734             ngx_memcpy(ctx->saved, src, size);
735             ctx->saved_len = size;
736
737             b->shadow = buf;
738
739             return out;
740         }
741
742     } else {
743         out = NULL;
744         size = len + buf->last - src;
745         src = buf->pos;
746     }
747
748     if (size < NGX_HTML_ENTITY_LEN) {
749         size += NGX_HTML_ENTITY_LEN;
750     }
751
752     cl = ngx_http_charset_get_buffer(pool, ctx, size);
753     if (cl == NULL) {
754         return NULL;
755     }
756
757     if (out) {
758         out->next = cl;
759
760     } else {
761         out = cl;
762     }
763
764     b = cl->buf;
765     dst = b->pos;
766
767     goto recode;
768 }
769
770 out = ngx_alloc_chain_link(pool);
771 if (out == NULL) {
772     return NULL;
773 }
774
775 out->buf = buf;
776 out->next = NULL;
777
778 return out;
779 }
780
781 /* process incomplete UTF sequence from previous buffer */
782
783 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, pool->log, 0,
784               "http charset utf saved: %z", ctx->saved_len);
785
786 p = src;
787
788 for (i = ctx->saved_len; i < NGX_UTF_LEN; i++) {
789     ctx->saved[i] = *p++;
790
791     if (p == buf->last) {
792         break;
793     }
794 }
795
796 saved = ctx->saved;
797 n = ngx_utf8_decode(&saved, i);
798
799 c = '\0';
800
801 if (n < 0x10000) {

```

```

802     table = (u_char **) ctx->table;
803     p = table[n >> 8];
804
805     if (p) {
806         c = p[n & 0xff];
807     }
808
809 } else if (n == 0xfffffffffe) {
810
811     /* incomplete UTF-8 symbol */
812
813     if (i < NGX_UTF_LEN) {
814         out = ngx_http_charset_get_buf(pool, ctx);
815         if (out == NULL) {
816             return NULL;
817         }
818
819         b = out->buf;
820
821         b->pos = buf->pos;
822         b->last = buf->last;
823         b->sync = 1;
824         b->shadow = buf;
825
826         ngx_memcpy(&ctx->saved[ctx->saved_len], src, i);
827         ctx->saved_len += i;
828
829         return out;
830     }
831 }
832
833 size = buf->last - buf->pos;
834
835 if (size < NGX_HTML_ENTITY_LEN) {
836     size += NGX_HTML_ENTITY_LEN;
837 }
838
839 cl = ngx_http_charset_get_buffer(pool, ctx, size);
840 if (cl == NULL) {
841     return NULL;
842 }
843
844 out = cl;
845
846 b = cl->buf;
847 dst = b->pos;
848
849 if (c) {
850     *dst++ = c;
851 } else if (n == 0xfffffffffe) {
852     *dst++ = '?';
853
854     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, pool->log, 0,
855                  "http charset invalid utf 0");
856
857     saved = &ctx->saved[NGX_UTF_LEN];
858
859 } else if (n > 0x10ffff) {
860     *dst++ = '?';
861
862     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, pool->log, 0,
863                  "http charset invalid utf 1");
864
865 } else {
866     dst = ngx_sprintf(dst, "&#%uD;", n);
867 }
868
869 src += (saved - ctx->saved) - ctx->saved_len;
870 ctx->saved_len = 0;
871
872 recode:
873
874 ll = &cl->next;
875
876 table = (u_char **) ctx->table;

```

```

878
879 while (src < buf->last) {
880
881     if ((size_t) (b->end - dst) < NGX\_HTML\_ENTITY\_LEN) {
882         b->last = dst;
883
884         size = buf->last - src + NGX\_HTML\_ENTITY\_LEN;
885
886         cl = ngx\_http\_charset\_get\_buffer(pool, ctx, size);
887         if (cl == NULL) {
888             return NULL;
889         }
890
891         *ll = cl;
892         ll = &cl->next;
893
894         b = cl->buf;
895         dst = b->pos;
896     }
897
898     if (*src < 0x80) {
899         *dst++ = *src++;
900         continue;
901     }
902
903     len = buf->last - src;
904
905     n = ngx\_utf8\_decode(&src, len);
906
907     if (n < 0x10000) {
908         p = table[n >> 8];
909
910         if (p) {
911             c = p[n & 0xff];
912
913             if (c) {
914                 *dst++ = c;
915                 continue;
916             }
917         }
918     }
919
920     dst = ngx\_sprintf(dst, "&#%uD;", n);
921
922     continue;
923 }
924
925 if (n == 0xffffffff) {
926     /* incomplete UTF-8 symbol */
927
928     ngx\_memcpy(ctx->saved, src, len);
929     ctx->saved_len = len;
930
931     if (b->pos == dst) {
932         b->sync = 1;
933         b->temporary = 0;
934     }
935
936     break;
937 }
938
939 if (n > 0x10ffff) {
940     *dst++ = '?';
941
942     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, pool->log, 0,
943         "http charset invalid utf 2");
944
945     continue;
946 }
947
948 /* n > 0xffff */
949
950 dst = ngx\_sprintf(dst, "&#%uD;", n);
951 }
952
953 b->last = dst;

```

```

954     b->last_buf = buf->last_buf;
955     b->last_in_chain = buf->last_in_chain;
956     b->flush = buf->flush;
957
958     b->shadow = buf;
959
960     return out;
961 }
962 }
963
964 static ngx_chain_t *
965 ngx_http_charset_recode_to_utf8(ngx_pool_t *pool, ngx_buf_t *buf,
966     ngx_http_charset_ctx_t *ctx)
967 {
968     size_t      len, size;
969     u_char      *p, *src, *dst, *table;
970     ngx_buf_t   *b;
971     ngx_chain_t *out, *cl, **ll;
972
973     table = ctx->table;
974
975     for (src = buf->pos; src < buf->last; src++) {
976         if (table[*src * NGX_UTF_LEN] == '\1') {
977             continue;
978         }
979     }
980
981     goto recode;
982 }
983
984 out = ngx_alloc_chain_link(pool);
985 if (out == NULL) {
986     return NULL;
987 }
988
989 out->buf = buf;
990 out->next = NULL;
991
992     return out;
993
994 recode:
995
996     /*
997     * we assume that there are about half of characters to be recoded,
998     * so we preallocate "size / 2 + size / 2 * ctx->length"
999     */
1000
1001     len = src - buf->pos;
1002
1003     if (len > 512) {
1004         out = ngx_http_charset_get_buf(pool, ctx);
1005         if (out == NULL) {
1006             return NULL;
1007         }
1008
1009         b = out->buf;
1010
1011         b->temporary = buf->temporary;
1012         b->memory = buf->memory;
1013         b->mmap = buf->mmap;
1014         b->flush = buf->flush;
1015
1016         b->pos = buf->pos;
1017         b->last = src;
1018
1019         out->buf = b;
1020         out->next = NULL;
1021
1022         size = buf->last - src;
1023         size = size / 2 + size / 2 * ctx->length;
1024
1025     } else {
1026         out = NULL;
1027
1028         size = buf->last - src;
1029         size = len + size / 2 + size / 2 * ctx->length;

```

```

1030     src = buf->pos;
1031 }
1032
1033
1034 cl = ngx_http_charset_get_buffer(pool, ctx, size);
1035 if (cl == NULL) {
1036     return NULL;
1037 }
1038
1039 if (out) {
1040     out->next = cl;
1041
1042 } else {
1043     out = cl;
1044 }
1045
1046 ll = &cl->next;
1047
1048 b = cl->buf;
1049 dst = b->pos;
1050
1051 while (src < buf->last) {
1052
1053     p = &table[*src++ * NGX_UTF_LEN];
1054     len = *p++;
1055
1056     if ((size_t) (b->end - dst) < len) {
1057         b->last = dst;
1058
1059         size = buf->last - src;
1060         size = len + size / 2 + size / 2 * ctx->length;
1061
1062         cl = ngx_http_charset_get_buffer(pool, ctx, size);
1063         if (cl == NULL) {
1064             return NULL;
1065         }
1066
1067         *ll = cl;
1068         ll = &cl->next;
1069
1070         b = cl->buf;
1071         dst = b->pos;
1072     }
1073
1074     while (len) {
1075         *dst++ = *p++;
1076         len--;
1077     }
1078 }
1079
1080 b->last = dst;
1081
1082 b->last_buf = buf->last_buf;
1083 b->last_in_chain = buf->last_in_chain;
1084 b->flush = buf->flush;
1085
1086 b->shadow = buf;
1087
1088 return out;
1089 }
1090
1091
1092 static ngx_chain_t *
1093 ngx_http_charset_get_buf(ngx_pool_t *pool, ngx_http_charset_ctx_t *ctx)
1094 {
1095     ngx_chain_t *cl;
1096
1097     cl = ctx->free_bufs;
1098
1099     if (cl) {
1100         ctx->free_bufs = cl->next;
1101
1102         cl->buf->shadow = NULL;
1103         cl->next = NULL;
1104
1105         return cl;

```

```

1106     }
1107
1108     cl = ngx_alloc_chain_link(pool);
1109     if (cl == NULL) {
1110         return NULL;
1111     }
1112
1113     cl->buf = ngx_calloc_buf(pool);
1114     if (cl->buf == NULL) {
1115         return NULL;
1116     }
1117
1118     cl->next = NULL;
1119
1120     cl->buf->tag = (ngx_buf_tag_t) &ngx_http_charset_filter_module;
1121
1122     return cl;
1123 }
1124
1125
1126 static ngx_chain_t *
1127 ngx_http_charset_get_buffer(ngx_pool_t *pool, ngx_http_charset_ctx_t *ctx,
1128                             size_t size)
1129 {
1130     ngx_buf_t *b;
1131     ngx_chain_t *cl, **ll;
1132
1133     for (ll = &ctx->free_buffers, cl = ctx->free_buffers;
1134          cl;
1135          ll = &cl->next, cl = cl->next)
1136     {
1137         b = cl->buf;
1138
1139         if ((size_t) (b->end - b->start) >= size) {
1140             *ll = cl->next;
1141             cl->next = NULL;
1142
1143             b->pos = b->start;
1144             b->temporary = 1;
1145             b->shadow = NULL;
1146
1147             return cl;
1148         }
1149     }
1150
1151     cl = ngx_alloc_chain_link(pool);
1152     if (cl == NULL) {
1153         return NULL;
1154     }
1155
1156     cl->buf = ngx_create_temp_buf(pool, size);
1157     if (cl->buf == NULL) {
1158         return NULL;
1159     }
1160
1161     cl->next = NULL;
1162
1163     cl->buf->temporary = 1;
1164     cl->buf->tag = (ngx_buf_tag_t) &ngx_http_charset_filter_module;
1165
1166     return cl;
1167 }
1168
1169
1170 static char *
1171 ngx_http_charset_map_block(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
1172 {
1173     ngx_http_charset_main_conf_t *mcf = conf;
1174
1175     char *rv;
1176     u_char *p, *dst2src, **pp;
1177     ngx_int_t src, dst;
1178     ngx_uint_t i, n;
1179     ngx_str_t *value;
1180     ngx_conf_t pvcf;
1181     ngx_http_charset_t *charset;

```

```

1182     ngx_http_charset_tables_t    *table;
1183     ngx_http_charset_conf_ctx_t   ctx;
1184
1185     value = cf->args->elts;
1186
1187     src = ngx_http_add_charset(&mcf->charsets, &value[1]);
1188     if (src == NGX_ERROR) {
1189         return NGX_CONF_ERROR;
1190     }
1191
1192     dst = ngx_http_add_charset(&mcf->charsets, &value[2]);
1193     if (dst == NGX_ERROR) {
1194         return NGX_CONF_ERROR;
1195     }
1196
1197     if (src == dst) {
1198         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1199             "\\\"charset_map\\\" between the same charsets \"
1200             \"%V\\\" and \"%V\\\"\", &value[1], &value[2]);
1201         return NGX_CONF_ERROR;
1202     }
1203
1204     table = mcf->tables.elts;
1205     for (i = 0; i < mcf->tables.nelts; i++) {
1206         if ((src == table->src && dst == table->dst)
1207             || (src == table->dst && dst == table->src))
1208             {
1209                 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1210                     "duplicate \\\"charset_map\\\" between \"
1211                     \"%V\\\" and \"%V\\\"\", &value[1], &value[2]);
1212                 return NGX_CONF_ERROR;
1213             }
1214     }
1215
1216     table = ngx_array_push(&mcf->tables);
1217     if (table == NULL) {
1218         return NGX_CONF_ERROR;
1219     }
1220
1221     table->src = src;
1222     table->dst = dst;
1223
1224     if (ngx_strcasecmp(value[2].data, (u_char *) "utf-8") == 0) {
1225         table->src2dst = ngx_pcalloc(cf->pool, 256 * NGX_UTF_LEN);
1226         if (table->src2dst == NULL) {
1227             return NGX_CONF_ERROR;
1228         }
1229
1230         table->dst2src = ngx_pcalloc(cf->pool, 256 * sizeof(void *));
1231         if (table->dst2src == NULL) {
1232             return NGX_CONF_ERROR;
1233         }
1234
1235         dst2src = ngx_pcalloc(cf->pool, 256);
1236         if (dst2src == NULL) {
1237             return NGX_CONF_ERROR;
1238         }
1239
1240         pp = (u_char **) &table->dst2src[0];
1241         pp[0] = dst2src;
1242
1243         for (i = 0; i < 128; i++) {
1244             p = &table->src2dst[i * NGX_UTF_LEN];
1245             p[0] = '\\1';
1246             p[1] = (u_char) i;
1247             dst2src[i] = (u_char) i;
1248         }
1249
1250         for (/* void */; i < 256; i++) {
1251             p = &table->src2dst[i * NGX_UTF_LEN];
1252             p[0] = '\\1';
1253             p[1] = '?';
1254         }
1255     }
1256     } else {
1257         table->src2dst = ngx_palloc(cf->pool, 256);

```

```

1258     if (table->src2dst == NULL) {
1259         return NGX\_CONF\_ERROR;
1260     }
1261
1262     table->dst2src = ngx\_palloc(cf->pool, 256);
1263     if (table->dst2src == NULL) {
1264         return NGX\_CONF\_ERROR;
1265     }
1266
1267     for (i = 0; i < 128; i++) {
1268         table->src2dst[i] = (u_char) i;
1269         table->dst2src[i] = (u_char) i;
1270     }
1271
1272     for (/* void */; i < 256; i++) {
1273         table->src2dst[i] = '?';
1274         table->dst2src[i] = '?';
1275     }
1276 }
1277
1278 charset = mcf->charsets.elts;
1279
1280 ctx.table = table;
1281 ctx.charset = &charset[dst];
1282 ctx.characters = 0;
1283
1284 pvcf = *cf;
1285 cf->ctx = &ctx;
1286 cf->handler = ngx\_http\_charset\_map;
1287 cf->handler_conf = conf;
1288
1289 rv = ngx\_conf\_parse(cf, NULL);
1290
1291 *cf = pvcf;
1292
1293 if (ctx.characters) {
1294     n = ctx.charset->length;
1295     ctx.charset->length /= ctx.characters;
1296
1297     if (((n * 10) / ctx.characters) % 10 > 4) {
1298         ctx.charset->length++;
1299     }
1300 }
1301
1302 return rv;
1303 }
1304
1305
1306 static char *
1307 ngx\_http\_charset\_map(ngx\_conf\_t *cf, ngx\_command\_t *dummy, void *conf)
1308 {
1309     u_char                *p, *dst2src, **pp;
1310     uint32_t              n;
1311     ngx\_int\_t              src, dst;
1312     ngx\_str\_t              *value;
1313     ngx\_uint\_t              i;
1314     ngx\_http\_charset\_tables\_t *table;
1315     ngx\_http\_charset\_conf\_ctx\_t *ctx;
1316
1317     if (cf->args->nelts != 2) {
1318         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0, "invalid parameters number");
1319         return NGX\_CONF\_ERROR;
1320     }
1321
1322     value = cf->args->elts;
1323
1324     src = ngx\_hextoi(value[0].data, value[0].len);
1325     if (src == NGX\_ERROR || src > 255) {
1326         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
1327             "invalid value \"%V\"", &value[0]);
1328         return NGX\_CONF\_ERROR;
1329     }
1330
1331     ctx = cf->ctx;
1332     table = ctx->table;
1333

```



```

1334     if (ctx->charset->utf8) {
1335         p = &table->src2dst[src * NGX\_UTF\_LEN];
1336
1337         *p++ = (u_char) (value[1].len / 2);
1338
1339         for (i = 0; i < value[1].len; i += 2) {
1340             dst = ngx\_hextoi(&value[1].data[i], 2);
1341             if (dst == NGX\_ERROR || dst > 255) {
1342                 ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
1343                     "invalid value \"%V\"", &value[1]);
1344                 return NGX\_CONF\_ERROR;
1345             }
1346
1347             *p++ = (u_char) dst;
1348         }
1349
1350         i /= 2;
1351
1352         ctx->charset->length += i;
1353         ctx->characters++;
1354
1355         p = &table->src2dst[src * NGX\_UTF\_LEN] + 1;
1356
1357         n = ngx\_utf8\_decode(&p, i);
1358
1359         if (n > 0xffff) {
1360             ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
1361                 "invalid value \"%V\"", &value[1]);
1362             return NGX\_CONF\_ERROR;
1363         }
1364
1365         pp = (u_char **) &table->dst2src[0];
1366
1367         dst2src = pp[n >> 8];
1368
1369         if (dst2src == NULL) {
1370             dst2src = ngx\_palloc(cf->pool, 256);
1371             if (dst2src == NULL) {
1372                 return NGX\_CONF\_ERROR;
1373             }
1374
1375             pp[n >> 8] = dst2src;
1376         }
1377
1378         dst2src[n & 0xff] = (u_char) src;
1379
1380     } else {
1381         dst = ngx\_hextoi(value[1].data, value[1].len);
1382         if (dst == NGX\_ERROR || dst > 255) {
1383             ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
1384                 "invalid value \"%V\"", &value[1]);
1385             return NGX\_CONF\_ERROR;
1386         }
1387
1388         table->src2dst[src] = (u_char) dst;
1389         table->dst2src[dst] = (u_char) src;
1390     }
1391
1392     return NGX\_CONF\_OK;
1393 }
1394
1395
1396 static char *
1397 ngx\_http\_set\_charset\_slot(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
1398 {
1399     char *p = conf;
1400
1401     ngx\_int\_t *cp;
1402     ngx\_str\_t *value, var;
1403     ngx\_http\_charset\_main\_conf\_t *mcf;
1404
1405     cp = (ngx\_int\_t *) (p + cmd->offset);
1406
1407     if (*cp != NGX\_CONF\_UNSET) {
1408         return "is duplicate";
1409     }

```

```

1410 value = cf->args->elts;
1411
1412
1413 if (cmd->offset == offsetof(ngx\_http\_charset\_loc\_conf\_t, charset)
1414     && ngx\_strcmp(value[1].data, "off") == 0)
1415 {
1416     *cp = NGX\_HTTP\_CHARSET\_OFF;
1417     return NGX\_CONF\_OK;
1418 }
1419
1420
1421 if (value[1].data[0] == '$') {
1422     var.len = value[1].len - 1;
1423     var.data = value[1].data + 1;
1424
1425     *cp = ngx\_http\_get\_variable\_index(cf, &var);
1426
1427     if (*cp == NGX\_ERROR) {
1428         return NGX\_CONF\_ERROR;
1429     }
1430
1431     *cp += NGX\_HTTP\_CHARSET\_VAR;
1432
1433     return NGX\_CONF\_OK;
1434 }
1435
1436 mcf = ngx\_http\_conf\_get\_module\_main\_conf(cf,
1437     ngx\_http\_charset\_filter\_module);
1438
1439 *cp = ngx\_http\_add\_charset(&mcf->charsets, &value[1]);
1440 if (*cp == NGX\_ERROR) {
1441     return NGX\_CONF\_ERROR;
1442 }
1443
1444 return NGX\_CONF\_OK;
1445 }
1446
1447
1448 static ngx\_int\_t
1449 ngx\_http\_add\_charset(ngx\_array\_t *charsets, ngx\_str\_t *name)
1450 {
1451     ngx\_uint\_t i;
1452     ngx\_http\_charset\_t *c;
1453
1454     c = charsets->elts;
1455     for (i = 0; i < charsets->nelts; i++) {
1456         if (name->len != c[i].name.len) {
1457             continue;
1458         }
1459
1460         if (ngx\_strcasecmp(name->data, c[i].name.data) == 0) {
1461             break;
1462         }
1463     }
1464
1465     if (i < charsets->nelts) {
1466         return i;
1467     }
1468
1469     c = ngx\_array\_push(charsets);
1470     if (c == NULL) {
1471         return NGX\_ERROR;
1472     }
1473
1474     c->tables = NULL;
1475     c->name = *name;
1476     c->length = 0;
1477
1478     if (ngx\_strcasecmp(name->data, (u_char *) "utf-8") == 0) {
1479         c->utf8 = 1;
1480
1481     } else {
1482         c->utf8 = 0;
1483     }
1484
1485     return i;

```

```

1486 }
1487
1488
1489 static void *
1490 ngx_http_charset_create_main_conf(ngx_conf_t *cf)
1491 {
1492     ngx_http_charset_main_conf_t *mcf;
1493
1494     mcf = ngx_palloc(cf->pool, sizeof(ngx_http_charset_main_conf_t));
1495     if (mcf == NULL) {
1496         return NULL;
1497     }
1498
1499     if (ngx_array_init(&mcf->charsets, cf->pool, 2, sizeof(ngx_http_charset_t))
1500         != NGX_OK)
1501     {
1502         return NULL;
1503     }
1504
1505     if (ngx_array_init(&mcf->tables, cf->pool, 1,
1506         sizeof(ngx_http_charset_tables_t))
1507         != NGX_OK)
1508     {
1509         return NULL;
1510     }
1511
1512     if (ngx_array_init(&mcf->recodes, cf->pool, 2,
1513         sizeof(ngx_http_charset_recode_t))
1514         != NGX_OK)
1515     {
1516         return NULL;
1517     }
1518
1519     return mcf;
1520 }
1521
1522
1523 static void *
1524 ngx_http_charset_create_loc_conf(ngx_conf_t *cf)
1525 {
1526     ngx_http_charset_loc_conf_t *lcf;
1527
1528     lcf = ngx_palloc(cf->pool, sizeof(ngx_http_charset_loc_conf_t));
1529     if (lcf == NULL) {
1530         return NULL;
1531     }
1532
1533     /*
1534      * set by ngx_palloc():
1535      *
1536      *     lcf->types = { NULL };
1537      *     lcf->types_keys = NULL;
1538      */
1539
1540     lcf->charset = NGX_CONF_UNSET;
1541     lcf->source_charset = NGX_CONF_UNSET;
1542     lcf->override_charset = NGX_CONF_UNSET;
1543
1544     return lcf;
1545 }
1546
1547
1548 static char *
1549 ngx_http_charset_merge_loc_conf(ngx_conf_t *cf, void *parent, void *child)
1550 {
1551     ngx_http_charset_loc_conf_t *prev = parent;
1552     ngx_http_charset_loc_conf_t *conf = child;
1553
1554     ngx_uint_t i;
1555     ngx_http_charset_recode_t *recode;
1556     ngx_http_charset_main_conf_t *mcf;
1557
1558     if (ngx_http_merge_types(cf, &conf->types_keys, &conf->types,
1559         &prev->types_keys, &prev->types,
1560         ngx_http_charset_default_types)
1561         != NGX_OK)

```

```

1562 {
1563     return NGX\_CONF\_ERROR;
1564 }
1565
1566 ngx\_conf\_merge\_value(conf->override_charset, prev->override_charset, 0);
1567 ngx\_conf\_merge\_value(conf->charset, prev->charset, NGX\_HTTP\_CHARSET\_OFF);
1568 ngx\_conf\_merge\_value(conf->source_charset, prev->source_charset,
1569     NGX\_HTTP\_CHARSET\_OFF);
1570
1571 if (conf->charset == NGX\_HTTP\_CHARSET\_OFF
1572     || conf->source_charset == NGX\_HTTP\_CHARSET\_OFF
1573     || conf->charset == conf->source_charset)
1574 {
1575     return NGX\_CONF\_OK;
1576 }
1577
1578 if (conf->source_charset >= NGX\_HTTP\_CHARSET\_VAR
1579     || conf->charset >= NGX\_HTTP\_CHARSET\_VAR)
1580 {
1581     return NGX\_CONF\_OK;
1582 }
1583
1584 mcf = ngx\_http\_conf\_get\_module\_main\_conf(cf,
1585     ngx\_http\_charset\_filter\_module);
1586
1587 recode = mcf->recodes.elts;
1588 for (i = 0; i < mcf->recodes.nelts; i++) {
1589     if (conf->source_charset == recode[i].src
1590         && conf->charset == recode[i].dst)
1591     {
1592         return NGX\_CONF\_OK;
1593     }
1594 }
1595
1596 recode = ngx\_array\_push(&mcf->recodes);
1597 if (recode == NULL) {
1598     return NGX\_CONF\_ERROR;
1599 }
1600
1601 recode->src = conf->source_charset;
1602 recode->dst = conf->charset;
1603
1604 return NGX\_CONF\_OK;
1605 }
1606
1607 static ngx\_int\_t
1608 ngx\_http\_charset\_postconfiguration(ngx\_conf\_t *cf)
1609 {
1610     u_char                **src, **dst;
1611     ngx\_int\_t              c;
1612     ngx\_uint\_t             i, t;
1613     ngx\_http\_charset\_t     *charset;
1614     ngx\_http\_charset\_recode\_t *recode;
1615     ngx\_http\_charset\_tables\_t *tables;
1616     ngx\_http\_charset\_main\_conf\_t *mcf;
1617
1618     mcf = ngx\_http\_conf\_get\_module\_main\_conf(cf,
1619         ngx\_http\_charset\_filter\_module);
1620
1621     recode = mcf->recodes.elts;
1622     tables = mcf->tables.elts;
1623     charset = mcf->charsets.elts;
1624
1625     for (i = 0; i < mcf->recodes.nelts; i++) {
1626
1627         c = recode[i].src;
1628
1629         for (t = 0; t < mcf->tables.nelts; t++) {
1630
1631             if (c == tables[t].src && recode[i].dst == tables[t].dst) {
1632                 goto next;
1633             }
1634
1635             if (c == tables[t].dst && recode[i].dst == tables[t].src) {
1636                 goto next;
1637             }

```

```

1638     }
1639
1640     ngx_log_error(NGX_LOG_EMERG, cf->log, 0,
1641                 "no \"charset_map\" between the charsets \"%V\" and \"%V\"",
1642                 &charset[c].name, &charset[recode[i].dst].name);
1643     return NGX_ERROR;
1644
1645 next:
1646     continue;
1647 }
1648
1649
1650 for (t = 0; t < mcf->tables.nelts; t++) {
1651
1652     src = charset[tables[t].src].tables;
1653
1654     if (src == NULL) {
1655         src = ngx_palloc(cf->pool, sizeof(u_char *) * mcf->charsets.nelts);
1656         if (src == NULL) {
1657             return NGX_ERROR;
1658         }
1659
1660         charset[tables[t].src].tables = src;
1661     }
1662
1663     dst = charset[tables[t].dst].tables;
1664
1665     if (dst == NULL) {
1666         dst = ngx_palloc(cf->pool, sizeof(u_char *) * mcf->charsets.nelts);
1667         if (dst == NULL) {
1668             return NGX_ERROR;
1669         }
1670
1671         charset[tables[t].dst].tables = dst;
1672     }
1673
1674     src[tables[t].dst] = tables[t].src2dst;
1675     dst[tables[t].src] = tables[t].dst2src;
1676 }
1677
1678 ngx_http_next_header_filter = ngx_http_top_header_filter;
1679 ngx_http_top_header_filter = ngx_http_charset_header_filter;
1680
1681 ngx_http_next_body_filter = ngx_http_top_body_filter;
1682 ngx_http_top_body_filter = ngx_http_charset_body_filter;
1683
1684 return NGX_OK;
1685 }

```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_chunked\_filter\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_chunked\\_filter\\_module](#)
- [ngx\\_http\\_chunked\\_filter\\_module\\_ctx](#)
- [ngx\\_http\\_next\\_body\\_filter](#)
- [ngx\\_http\\_next\\_header\\_filter](#)

## Data types defined

- [ngx\\_http\\_chunked\\_filter\\_ctx\\_t](#)

## Functions defined

- [ngx\\_http\\_chunked\\_body\\_filter](#)
- [ngx\\_http\\_chunked\\_filter\\_init](#)
- [ngx\\_http\\_chunked\\_header\\_filter](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 typedef struct {
14     ngx_chain_t      *free;
15     ngx_chain_t      *busy;
16 } ngx_http_chunked_filter_ctx_t;
17
18
19 static ngx_int_t ngx_http_chunked_filter_init(ngx_conf_t *cf);
20
21
22 static ngx_http_module_t ngx_http_chunked_filter_module_ctx = {
23     NULL,                          /* preconfiguration */
24     ngx_http_chunked_filter_init,   /* postconfiguration */
25
26     NULL,                            /* create main configuration */
27     NULL,                            /* init main configuration */
28
29     NULL,                            /* create server configuration */
30     NULL,                            /* merge server configuration */
31
32     NULL,                            /* create location configuration */
33     NULL,                            /* merge location configuration */
34 };
35
36
37 ngx_module_t ngx_http_chunked_filter_module = {
38     NGX_MODULE_V1,
39     &ngx_http_chunked_filter_module_ctx, /* module context */
```

```

40     NULL,                                /* module directives */
41     NGX_HTTP_MODULE,                    /* module type */
42     NULL,                                /* init master */
43     NULL,                                /* init module */
44     NULL,                                /* init process */
45     NULL,                                /* init thread */
46     NULL,                                /* exit thread */
47     NULL,                                /* exit process */
48     NULL,                                /* exit master */
49     NGX_MODULE_V1_PADDING
50 };
51
52
53 static ngx_http_output_header_filter_pt ngx_http_next_header_filter;
54 static ngx_http_output_body_filter_pt  ngx_http_next_body_filter;
55
56
57 static ngx_int_t
58 ngx_http_chunked_header_filter(ngx_http_request_t *r)
59 {
60     ngx_http_core_loc_conf_t    *clcf;
61     ngx_http_chunked_filter_ctx_t *ctx;
62
63     if (r->headers_out.status == NGX_HTTP_NOT_MODIFIED
64         || r->headers_out.status == NGX_HTTP_NO_CONTENT
65         || r->headers_out.status < NGX_HTTP_OK
66         || r != r->main
67         || (r->method & NGX_HTTP_HEAD))
68     {
69         return ngx_http_next_header_filter(r);
70     }
71
72     if (r->headers_out.content_length_n == -1) {
73         if (r->http_version < NGX_HTTP_VERSION_11) {
74             r->keepalive = 0;
75         }
76     } else {
77         clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
78
79         if (clcf->chunked_transfer_encoding) {
80             r->chunked = 1;
81
82             ctx = ngx_palloc(r->pool,
83                             sizeof(ngx_http_chunked_filter_ctx_t));
84             if (ctx == NULL) {
85                 return NGX_ERROR;
86             }
87
88             ngx_http_set_ctx(r, ctx, ngx_http_chunked_filter_module);
89
90         } else {
91             r->keepalive = 0;
92         }
93     }
94 }
95
96 return ngx_http_next_header_filter(r);
97 }
98
99
100 static ngx_int_t
101 ngx_http_chunked_body_filter(ngx_http_request_t *r, ngx_chain_t *in)
102 {
103     u_char                *chunk;
104     off_t                 size;
105     ngx_int_t            rc;
106     ngx_buf_t           *b;
107     ngx_chain_t        *out, *cl, *tl, **ll;
108     ngx_http_chunked_filter_ctx_t *ctx;
109
110     if (in == NULL || !r->chunked || r->header_only) {
111         return ngx_http_next_body_filter(r, in);
112     }
113
114     ctx = ngx_http_get_module_ctx(r, ngx_http_chunked_filter_module);
115

```

```

116 out = NULL;
117 ll = &out;
118
119 size = 0;
120 cl = in;
121
122 for ( ;; ) {
123     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
124                 "http chunk: %d", ngx_buf_size(cl->buf));
125
126     size += ngx_buf_size(cl->buf);
127
128     if (cl->buf->flush
129         || cl->buf->sync
130         || ngx_buf_in_memory(cl->buf)
131         || cl->buf->in_file)
132     {
133         tl = ngx_alloc_chain_link(r->pool);
134         if (tl == NULL) {
135             return NGX_ERROR;
136         }
137
138         tl->buf = cl->buf;
139         *ll = tl;
140         ll = &tl->next;
141     }
142
143     if (cl->next == NULL) {
144         break;
145     }
146
147     cl = cl->next;
148 }
149
150 if (size) {
151     tl = ngx_chain_get_free_buf(r->pool, &ctx->free);
152     if (tl == NULL) {
153         return NGX_ERROR;
154     }
155
156     b = tl->buf;
157     chunk = b->start;
158
159     if (chunk == NULL) {
160         /* the "0000000000000000" is 64-bit hexadecimal string */
161
162         chunk = ngx_palloc(r->pool, sizeof("0000000000000000" CRLF) - 1);
163         if (chunk == NULL) {
164             return NGX_ERROR;
165         }
166
167         b->start = chunk;
168         b->end = chunk + sizeof("0000000000000000" CRLF) - 1;
169     }
170
171     b->tag = (ngx_buf_tag_t) &ngx_http_chunked_filter_module;
172     b->memory = 0;
173     b->temporary = 1;
174     b->pos = chunk;
175     b->last = ngx_sprintf(chunk, "%x0" CRLF, size);
176
177     tl->next = out;
178     out = tl;
179 }
180
181 if (cl->buf->last_buf) {
182     tl = ngx_chain_get_free_buf(r->pool, &ctx->free);
183     if (tl == NULL) {
184         return NGX_ERROR;
185     }
186
187     b = tl->buf;
188
189     b->tag = (ngx_buf_tag_t) &ngx_http_chunked_filter_module;
190     b->temporary = 0;
191     b->memory = 1;

```



```

192     b->last_buf = 1;
193     b->pos = (u_char *) CRLF "0" CRLF CRLF;
194     b->last = b->pos + 7;
195
196     cl->buf->last_buf = 0;
197
198     *ll = t1;
199
200     if (size == 0) {
201         b->pos += 2;
202     }
203
204 } else if (size > 0) {
205     t1 = ngx\_chain\_get\_free\_buf(r->pool, &ctx->free);
206     if (t1 == NULL) {
207         return NGX\_ERROR;
208     }
209
210     b = t1->buf;
211
212     b->tag = (ngx\_buf\_tag\_t) &ngx\_http\_chunked\_filter\_module;
213     b->temporary = 0;
214     b->memory = 1;
215     b->pos = (u_char *) CRLF;
216     b->last = b->pos + 2;
217
218     *ll = t1;
219
220 } else {
221     *ll = NULL;
222 }
223
224 rc = ngx\_http\_next\_body\_filter(r, out);
225
226 ngx\_chain\_update\_chains(r->pool, &ctx->free, &ctx->busy, &out,
227                          (ngx\_buf\_tag\_t) &ngx\_http\_chunked\_filter\_module);
228
229 return rc;
230 }
231
232
233 static ngx\_int\_t
234 ngx\_http\_chunked\_filter\_init(ngx\_conf\_t *cf)
235 {
236     ngx\_http\_next\_header\_filter = ngx\_http\_top\_header\_filter;
237     ngx\_http\_top\_header\_filter = ngx\_http\_chunked\_header\_filter;
238
239     ngx\_http\_next\_body\_filter = ngx\_http\_top\_body\_filter;
240     ngx\_http\_top\_body\_filter = ngx\_http\_chunked\_body\_filter;
241
242     return NGX\_OK;
243 }

```

[One Level Up](#)

[Top Level](#)

## src/http/modules/nginx\_http\_dav\_module.c - nginx-1.7.10

### Global variables defined

- [ngx\\_http\\_dav\\_commands](#)
- [ngx\\_http\\_dav\\_methods\\_mask](#)
- [ngx\\_http\\_dav\\_module](#)
- [ngx\\_http\\_dav\\_module\\_ctx](#)

### Data types defined

- [ngx\\_http\\_dav\\_copy\\_ctx\\_t](#)
- [ngx\\_http\\_dav\\_loc\\_conf\\_t](#)

### Functions defined

- [ngx\\_http\\_dav\\_copy\\_dir](#)
- [ngx\\_http\\_dav\\_copy\\_dir\\_time](#)
- [ngx\\_http\\_dav\\_copy\\_move\\_handler](#)
- [ngx\\_http\\_dav\\_copy\\_tree\\_file](#)
- [ngx\\_http\\_dav\\_create\\_loc\\_conf](#)
- [ngx\\_http\\_dav\\_delete\\_dir](#)
- [ngx\\_http\\_dav\\_delete\\_file](#)
- [ngx\\_http\\_dav\\_delete\\_handler](#)
- [ngx\\_http\\_dav\\_delete\\_path](#)
- [ngx\\_http\\_dav\\_depth](#)
- [ngx\\_http\\_dav\\_error](#)
- [ngx\\_http\\_dav\\_handler](#)
- [ngx\\_http\\_dav\\_init](#)
- [ngx\\_http\\_dav\\_location](#)
- [ngx\\_http\\_dav\\_merge\\_loc\\_conf](#)
- [ngx\\_http\\_dav\\_mkcol\\_handler](#)
- [ngx\\_http\\_dav\\_noop](#)
- [ngx\\_http\\_dav\\_put\\_handler](#)

### Macros defined

- [NGX\\_HTTP\\_DAV\\_INFINITY\\_DEPTH](#)

- [NGX HTTP DAV INVALID DEPTH](#)
- [NGX HTTP DAV NO DEPTH](#)
- [NGX HTTP DAV OFF](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 #define NGX_HTTP_DAV_OFF                2
14
15
16 #define NGX_HTTP_DAV_NO_DEPTH           -3
17 #define NGX_HTTP_DAV_INVALID_DEPTH     -2
18 #define NGX_HTTP_DAV_INFINITY_DEPTH    -1
19
20
21 typedef struct {
22     ngx_uint_t  methods;
23     ngx_uint_t  access;
24     ngx_uint_t  min_delete_depth;
25     ngx_flag_t  create_full_put_path;
26 } ngx_http_dav_loc_conf_t;
27
28
29 typedef struct {
30     ngx_str_t   path;
31     size_t      len;
32 } ngx_http_dav_copy_ctx_t;
33
34
35 static ngx_int_t ngx_http_dav_handler(ngx_http_request_t *r);
36
37 static void ngx_http_dav_put_handler(ngx_http_request_t *r);
38
39 static ngx_int_t ngx_http_dav_delete_handler(ngx_http_request_t *r);
40 static ngx_int_t ngx_http_dav_delete_path(ngx_http_request_t *r,
41     ngx_str_t *path, ngx_uint_t dir);
42 static ngx_int_t ngx_http_dav_delete_dir(ngx_tree_ctx_t *ctx, ngx_str_t *path);
43 static ngx_int_t ngx_http_dav_delete_file(ngx_tree_ctx_t *ctx, ngx_str_t *path);
44 static ngx_int_t ngx_http_dav_noop(ngx_tree_ctx_t *ctx, ngx_str_t *path);
45
46 static ngx_int_t ngx_http_dav_mkcol_handler(ngx_http_request_t *r,
47     ngx_http_dav_loc_conf_t *dlcf);
48
49 static ngx_int_t ngx_http_dav_copy_move_handler(ngx_http_request_t *r);
50 static ngx_int_t ngx_http_dav_copy_dir(ngx_tree_ctx_t *ctx, ngx_str_t *path);
51 static ngx_int_t ngx_http_dav_copy_dir_time(ngx_tree_ctx_t *ctx,
52     ngx_str_t *path);
53 static ngx_int_t ngx_http_dav_copy_tree_file(ngx_tree_ctx_t *ctx,
54     ngx_str_t *path);
55
56 static ngx_int_t ngx_http_dav_depth(ngx_http_request_t *r, ngx_int_t dflt);
57 static ngx_int_t ngx_http_dav_error(ngx_log_t *log, ngx_err_t err,
58     ngx_int_t not_found, char *failed, u_char *path);
59 static ngx_int_t ngx_http_dav_location(ngx_http_request_t *r, u_char *path);
60 static void *ngx_http_dav_create_loc_conf(ngx_conf_t *cf);
61 static char *ngx_http_dav_merge_loc_conf(ngx_conf_t *cf,
62     void *parent, void *child);
63 static ngx_int_t ngx_http_dav_init(ngx_conf_t *cf);
64
65

```

```

66 static ngx_conf_bitmask_t ngx_http_dav_methods_mask[] = {
67     { ngx_string("off"), NGX_HTTP_DAV_OFF },
68     { ngx_string("put"), NGX_HTTP_PUT },
69     { ngx_string("delete"), NGX_HTTP_DELETE },
70     { ngx_string("mkcol"), NGX_HTTP_MKCOL },
71     { ngx_string("copy"), NGX_HTTP_COPY },
72     { ngx_string("move"), NGX_HTTP_MOVE },
73     { ngx_null_string, 0 }
74 };
75
76
77 static ngx_command_t ngx_http_dav_commands[] = {
78
79     { ngx_string("dav_methods"),
80       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_1MORE,
81       ngx_conf_set_bitmask_slot,
82       NGX_HTTP_LOC_CONF_OFFSET,
83       offsetof(ngx_http_dav_loc_conf_t, methods),
84       &ngx_http_dav_methods_mask },
85
86     { ngx_string("create_full_put_path"),
87       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
88       ngx_conf_set_flag_slot,
89       NGX_HTTP_LOC_CONF_OFFSET,
90       offsetof(ngx_http_dav_loc_conf_t, create_full_put_path),
91       NULL },
92
93     { ngx_string("min_delete_depth"),
94       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
95       ngx_conf_set_num_slot,
96       NGX_HTTP_LOC_CONF_OFFSET,
97       offsetof(ngx_http_dav_loc_conf_t, min_delete_depth),
98       NULL },
99
100    { ngx_string("dav_access"),
101      NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE123,
102      ngx_conf_set_access_slot,
103      NGX_HTTP_LOC_CONF_OFFSET,
104      offsetof(ngx_http_dav_loc_conf_t, access),
105      NULL },
106
107      ngx_null_command
108 };
109
110
111 static ngx_http_module_t ngx_http_dav_module_ctx = {
112     NULL, /* preconfiguration */
113     ngx_http_dav_init, /* postconfiguration */
114
115     NULL, /* create main configuration */
116     NULL, /* init main configuration */
117
118     NULL, /* create server configuration */
119     NULL, /* merge server configuration */
120
121     ngx_http_dav_create_loc_conf, /* create location configuration */
122     ngx_http_dav_merge_loc_conf /* merge location configuration */
123 };
124
125
126 ngx_module_t ngx_http_dav_module = {
127     NGX_MODULE_V1,
128     &ngx_http_dav_module_ctx, /* module context */
129     ngx_http_dav_commands, /* module directives */
130     NGX_HTTP_MODULE, /* module type */
131     NULL, /* init master */
132     NULL, /* init module */
133     NULL, /* init process */
134     NULL, /* init thread */
135     NULL, /* exit thread */
136     NULL, /* exit process */
137     NULL, /* exit master */
138     NGX_MODULE_V1_PADDING
139 };
140
141

```

```

142 static ngx_int_t
143 ngx_http_dav_handler(ngx_http_request_t *r)
144 {
145     ngx_int_t          rc;
146     ngx_http_dav_loc_conf_t *dlcf;
147
148     dlcf = ngx_http_get_module_loc_conf(r, ngx_http_dav_module);
149
150     if (!(r->method & dlcf->methods)) {
151         return NGX_DECLINED;
152     }
153
154     switch (r->method) {
155
156     case NGX_HTTP_PUT:
157
158         if (r->uri.data[r->uri.len - 1] == '/') {
159             ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
160                 "cannot PUT to a collection");
161             return NGX_HTTP_CONFLICT;
162         }
163
164         r->request_body_in_file_only = 1;
165         r->request_body_in_persistent_file = 1;
166         r->request_body_in_clean_file = 1;
167         r->request_body_file_group_access = 1;
168         r->request_body_file_log_level = 0;
169
170         rc = ngx_http_read_client_request_body(r, ngx_http_dav_put_handler);
171
172         if (rc >= NGX_HTTP_SPECIAL_RESPONSE) {
173             return rc;
174         }
175
176         return NGX_DONE;
177
178     case NGX_HTTP_DELETE:
179
180         return ngx_http_dav_delete_handler(r);
181
182     case NGX_HTTP_MKCOL:
183
184         return ngx_http_dav_mkcol_handler(r, dlcf);
185
186     case NGX_HTTP_COPY:
187
188         return ngx_http_dav_copy_move_handler(r);
189
190     case NGX_HTTP_MOVE:
191
192         return ngx_http_dav_copy_move_handler(r);
193     }
194
195     return NGX_DECLINED;
196 }
197
198
199 static void
200 ngx_http_dav_put_handler(ngx_http_request_t *r)
201 {
202     size_t          root;
203     time_t         date;
204     ngx_str_t      *temp, path;
205     ngx_uint_t     status;
206     ngx_file_info_t fi;
207     ngx_ext_rename_file_t ext;
208     ngx_http_dav_loc_conf_t *dlcf;
209
210     if (r->request_body == NULL || r->request_body->temp_file == NULL) {
211         ngx_http_finalize_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
212         return;
213     }
214
215     if (ngx_http_map_uri_to_path(r, &path, &root, 0) == NULL) {
216         ngx_http_finalize_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
217         return;

```

```

218 }
219
220 path.len--;
221
222 ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
223     "http put filename: \"%s\"", path.data);
224
225 temp = &r->request_body->temp_file->file.name;
226
227 if (ngx\_file\_info(path.data, &fi) == NGX\_FILE\_ERROR) {
228     status = NGX\_HTTP\_CREATED;
229
230 } else {
231     status = NGX\_HTTP\_NO\_CONTENT;
232
233     if (ngx\_is\_dir(&fi)) {
234         ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, NGX\_EISDIR,
235             "\"%s\" could not be created", path.data);
236
237         if (ngx\_delete\_file(temp->data) == NGX\_FILE\_ERROR) {
238             ngx\_log\_error(NGX\_LOG\_CRIT, r->connection->log, ngx\_errno,
239                 ngx\_delete\_file\_n " \"%s\" failed",
240                 temp->data);
241         }
242
243         ngx\_http\_finalize\_request(r, NGX\_HTTP\_CONFLICT);
244         return;
245     }
246 }
247
248 dlcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_dav\_module);
249
250 ext.access = dlcf->access;
251 ext.path_access = dlcf->access;
252 ext.time = -1;
253 ext.create_path = dlcf->create_full_put_path;
254 ext.delete_file = 1;
255 ext.log = r->connection->log;
256
257 if (r->headers_in.date) {
258     date = ngx\_http\_parse\_time(r->headers_in.date->value.data,
259         r->headers_in.date->value.len);
260
261     if (date != NGX\_ERROR) {
262         ext.time = date;
263         ext.fd = r->request_body->temp_file->file.fd;
264     }
265 }
266
267 if (ngx\_ext\_rename\_file(temp, &path, &ext) != NGX\_OK) {
268     ngx\_http\_finalize\_request(r, NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);
269     return;
270 }
271
272 if (status == NGX\_HTTP\_CREATED) {
273     if (ngx\_http\_dav\_location(r, path.data) != NGX\_OK) {
274         ngx\_http\_finalize\_request(r, NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);
275         return;
276     }
277
278     r->headers_out.content_length_n = 0;
279 }
280
281 r->headers_out.status = status;
282 r->header_only = 1;
283
284 ngx\_http\_finalize\_request(r, ngx\_http\_send\_header(r));
285 return;
286 }
287
288
289 static ngx\_int\_t
290 ngx\_http\_dav\_delete\_handler(ngx\_http\_request\_t *r)
291 {
292     size\_t          root;
293     ngx\_err\_t     err;

```

```

294     ngx_int_t         rc, depth;
295     ngx_uint_t        i, d, dir;
296     ngx_str_t         path;
297     ngx_file_info_t   fi;
298     ngx_http_dav_loc_conf_t *dlcf;
299
300     if (r->headers_in.content_length_n > 0) {
301         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
302             "DELETE with body is unsupported");
303         return NGX_HTTP_UNSUPPORTED_MEDIA_TYPE;
304     }
305
306     dlcf = ngx_http_get_module_loc_conf(r, ngx_http_dav_module);
307
308     if (dlcf->min_delete_depth) {
309         d = 0;
310
311         for (i = 0; i < r->uri.len; /* void */) {
312             if (r->uri.data[i++] == '/') {
313                 if (++d >= dlcf->min_delete_depth && i < r->uri.len) {
314                     goto ok;
315                 }
316             }
317         }
318
319         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
320             "insufficient URI depth:%i to DELETE", d);
321         return NGX_HTTP_CONFLICT;
322     }
323
324 ok:
325
326     if (ngx_http_map_uri_to_path(r, &path, &root, 0) == NULL) {
327         return NGX_HTTP_INTERNAL_SERVER_ERROR;
328     }
329
330     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
331         "http delete filename: \"%s\"", path.data);
332
333     if (ngx_link_info(path.data, &fi) == NGX_FILE_ERROR) {
334         err = ngx_errno;
335
336         rc = (err == NGX_ENOTDIR) ? NGX_HTTP_CONFLICT : NGX_HTTP_NOT_FOUND;
337
338         return ngx_http_dav_error(r->connection->log, err,
339             rc, ngx_link_info_n, path.data);
340     }
341
342     if (ngx_is_dir(&fi)) {
343
344         if (r->uri.data[r->uri.len - 1] != '/') {
345             ngx_log_error(NGX_LOG_ERR, r->connection->log, NGX_EISDIR,
346                 "DELETE \"%s\" failed", path.data);
347             return NGX_HTTP_CONFLICT;
348         }
349
350         depth = ngx_http_dav_depth(r, NGX_HTTP_DAV_INFINITY_DEPTH);
351
352         if (depth != NGX_HTTP_DAV_INFINITY_DEPTH) {
353             ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
354                 "\"Depth\" header must be infinity");
355             return NGX_HTTP_BAD_REQUEST;
356         }
357
358         path.len -= 2; /* omit "\0" */
359
360         dir = 1;
361     } else {
362
363         /*
364          * we do not need to test (r->uri.data[r->uri.len - 1] == '/')
365          * because ngx_link_info("/file/") returned NGX_ENOTDIR above
366          */
367
368         depth = ngx_http_dav_depth(r, 0);

```

```

370     if (depth != 0 && depth != NGX\_HTTP\_DAV\_INFINITY\_DEPTH) {
371         ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, 0,
372             "\"Depth\" header must be 0 or infinity");
373         return NGX\_HTTP\_BAD\_REQUEST;
374     }
375 }
376
377     dir = 0;
378 }
379
380 rc = ngx\_http\_dav\_delete\_path(r, &path, dir);
381
382 if (rc == NGX\_OK) {
383     return NGX\_HTTP\_NO\_CONTENT;
384 }
385
386 return rc;
387 }
388
389
390 static ngx\_int\_t
391 ngx\_http\_dav\_delete\_path(ngx\_http\_request\_t *r, ngx\_str\_t *path, ngx\_uint\_t dir)
392 {
393     char                *failed;
394     ngx\_tree\_ctx\_t     tree;
395
396     if (dir) {
397
398         tree.init_handler = NULL;
399         tree.file_handler = ngx\_http\_dav\_delete\_file;
400         tree.pre_tree_handler = ngx\_http\_dav\_noop;
401         tree.post_tree_handler = ngx\_http\_dav\_delete\_dir;
402         tree.spec_handler = ngx\_http\_dav\_delete\_file;
403         tree.data = NULL;
404         tree.alloc = 0;
405         tree.log = r->connection->log;
406
407         /* TODO: 207 */
408
409         if (ngx\_walk\_tree(&tree, path) != NGX\_OK) {
410             return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
411         }
412
413         if (ngx\_delete\_dir(path->data) != NGX\_FILE\_ERROR) {
414             return NGX\_OK;
415         }
416
417         failed = ngx\_delete\_dir\_n;
418
419     } else {
420
421         if (ngx\_delete\_file(path->data) != NGX\_FILE\_ERROR) {
422             return NGX\_OK;
423         }
424
425         failed = ngx\_delete\_file\_n;
426     }
427
428     return ngx\_http\_dav\_error(r->connection->log, ngx\_errno,
429         NGX\_HTTP\_NOT\_FOUND, failed, path->data);
430 }
431
432
433 static ngx\_int\_t
434 ngx\_http\_dav\_delete\_dir(ngx\_tree\_ctx\_t *ctx, ngx\_str\_t *path)
435 {
436     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, ctx->log, 0,
437         "http delete dir: \"%s\"", path->data);
438
439     if (ngx\_delete\_dir(path->data) == NGX\_FILE\_ERROR) {
440
441         /* TODO: add to 207 */
442
443         (void) ngx\_http\_dav\_error(ctx->log, ngx\_errno, 0, ngx\_delete\_dir\_n,
444             path->data);
445     }

```



```

446     return NGX_OK;
447 }
448 }
449
450
451 static ngx_int_t
452 ngx_http_dav_delete_file(ngx_tree_ctx_t *ctx, ngx_str_t *path)
453 {
454     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, ctx->log, 0,
455         "http delete file: \"%s\"", path->data);
456
457     if (ngx_delete_file(path->data) == NGX_FILE_ERROR) {
458
459         /* TODO: add to 207 */
460
461         (void) ngx_http_dav_error(ctx->log, ngx_errno, 0, ngx_delete_file_n,
462             path->data);
463     }
464
465     return NGX_OK;
466 }
467
468
469 static ngx_int_t
470 ngx_http_dav_noop(ngx_tree_ctx_t *ctx, ngx_str_t *path)
471 {
472     return NGX_OK;
473 }
474
475
476 static ngx_int_t
477 ngx_http_dav_mkcol_handler(ngx_http_request_t *r, ngx_http_dav_loc_conf_t *dlcf)
478 {
479     u_char    *p;
480     size_t    root;
481     ngx_str_t path;
482
483     if (r->headers_in.content_length_n > 0) {
484         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
485             "MKCOL with body is unsupported");
486         return NGX_HTTP_UNSUPPORTED_MEDIA_TYPE;
487     }
488
489     if (r->uri.data[r->uri.len - 1] != '/') {
490         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
491             "MKCOL can create a collection only");
492         return NGX_HTTP_CONFLICT;
493     }
494
495     p = ngx_http_map_uri_to_path(r, &path, &root, 0);
496     if (p == NULL) {
497         return NGX_HTTP_INTERNAL_SERVER_ERROR;
498     }
499
500     *(p - 1) = '\\0';
501     r->uri.len--;
502
503     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
504         "http mkcol path: \"%s\"", path.data);
505
506     if (ngx_create_dir(path.data, ngx_dir_access(dlcf->access))
507         != NGX_FILE_ERROR)
508     {
509         if (ngx_http_dav_location(r, path.data) != NGX_OK) {
510             return NGX_HTTP_INTERNAL_SERVER_ERROR;
511         }
512
513         return NGX_HTTP_CREATED;
514     }
515
516     return ngx_http_dav_error(r->connection->log, ngx_errno,
517         NGX_HTTP_CONFLICT, ngx_create_dir_n, path.data);
518 }
519
520
521 static ngx_int_t

```

```

522 ngx_http_dav_copy_move_handler(ngx_http_request_t *r)
523 {
524     u_char          *p, *host, *last, ch;
525     size_t          len, root;
526     ngx_err_t       err;
527     ngx_int_t       rc, depth;
528     ngx_uint_t      overwrite, slash, dir, flags;
529     ngx_str_t       path, uri, duri, args;
530     ngx_tree_ctx_t  tree;
531     ngx_copy_file_t cf;
532     ngx_file_info_t fi;
533     ngx_table_elt_t *dest, *over;
534     ngx_ext_rename_file_t ext;
535     ngx_http_dav_copy_ctx_t copy;
536     ngx_http_dav_loc_conf_t *dlcf;
537
538     if (r->headers_in.content_length_n > 0) {
539         return NGX_HTTP_UNSUPPORTED_MEDIA_TYPE;
540     }
541
542     dest = r->headers_in.destination;
543
544     if (dest == NULL) {
545         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
546             "client sent no \"Destination\" header");
547         return NGX_HTTP_BAD_REQUEST;
548     }
549
550     p = dest->value.data;
551     /* there is always '\0' even after empty header value */
552     if (p[0] == '/') {
553         last = p + dest->value.len;
554         goto destination_done;
555     }
556
557     len = r->headers_in.server.len;
558
559     if (len == 0) {
560         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
561             "client sent no \"Host\" header");
562         return NGX_HTTP_BAD_REQUEST;
563     }
564
565     #if (NGX_HTTP_SSL)
566
567     if (r->connection->ssl) {
568         if (ngx_strncmp(dest->value.data, "https://", sizeof("https://") - 1)
569             != 0)
570             {
571                 goto invalid_destination;
572             }
573
574         host = dest->value.data + sizeof("https://") - 1;
575
576     } else
577     #endif
578     {
579         if (ngx_strncmp(dest->value.data, "http://", sizeof("http://") - 1)
580             != 0)
581             {
582                 goto invalid_destination;
583             }
584
585         host = dest->value.data + sizeof("http://") - 1;
586     }
587
588     if (ngx_strncmp(host, r->headers_in.server.data, len) != 0) {
589         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
590             "\"Destination\" URI \"%V\" is handled by "
591             "different repository than the source URI",
592             &dest->value);
593         return NGX_HTTP_BAD_REQUEST;
594     }
595
596     last = dest->value.data + dest->value.len;
597

```

```

598     for (p = host + len; p < last; p++) {
599         if (*p == '/') {
600             goto destination_done;
601         }
602     }
603
604 invalid_destination:
605
606     ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
607         "client sent invalid \"Destination\" header: \"%V\"",
608         &dest->value);
609     return NGX_HTTP_BAD_REQUEST;
610
611 destination_done:
612
613     duri.len = last - p;
614     duri.data = p;
615     flags = NGX_HTTP_LOG_UNSAFE;
616
617     if (ngx_http_parse_unsafe_uri(r, &duri, &args, &flags) != NGX_OK) {
618         goto invalid_destination;
619     }
620
621     if ((r->uri.data[r->uri.len - 1] == '/' && *(last - 1) != '/')
622         || (r->uri.data[r->uri.len - 1] != '/' && *(last - 1) == '/'))
623     {
624         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
625             "both URI \"%V\" and \"Destination\" URI \"%V\" "
626             "should be either collections or non-collections",
627             &r->uri, &dest->value);
628         return NGX_HTTP_CONFLICT;
629     }
630
631     depth = ngx_http_dav_depth(r, NGX_HTTP_DAV_INFINITY_DEPTH);
632
633     if (depth != NGX_HTTP_DAV_INFINITY_DEPTH) {
634
635         if (r->method == NGX_HTTP_COPY) {
636             if (depth != 0) {
637                 ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
638                     "\"Depth\" header must be 0 or infinity");
639                 return NGX_HTTP_BAD_REQUEST;
640             }
641
642         } else {
643             ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
644                 "\"Depth\" header must be infinity");
645             return NGX_HTTP_BAD_REQUEST;
646         }
647     }
648
649     over = r->headers_in.overwrite;
650
651     if (over) {
652         if (over->value.len == 1) {
653             ch = over->value.data[0];
654
655             if (ch == 'T' || ch == 't') {
656                 overwrite = 1;
657                 goto overwrite_done;
658             }
659
660             if (ch == 'E' || ch == 'f') {
661                 overwrite = 0;
662                 goto overwrite_done;
663             }
664
665         }
666
667         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
668             "client sent invalid \"Overwrite\" header: \"%V\"",
669             &over->value);
670         return NGX_HTTP_BAD_REQUEST;
671     }
672
673     overwrite = 1;

```

```

674
675 overwrite_done:
676
677     if (ngx_http_map_uri_to_path(r, &path, &root, 0) == NULL) {
678         return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
679     }
680
681     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
682         "http copy from: \"%s\"", path.data);
683
684     uri = r->uri;
685     r->uri = duri;
686
687     if (ngx_http_map_uri_to_path(r, &copy.path, &root, 0) == NULL) {
688         return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
689     }
690
691     r->uri = uri;
692
693     copy.path.len--; /* omit "\0" */
694
695     if (copy.path.data[copy.path.len - 1] == '/') {
696         slash = 1;
697         copy.path.len--;
698         copy.path.data[copy.path.len] = '\0';
699
700     } else {
701         slash = 0;
702     }
703
704     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
705         "http copy to: \"%s\"", copy.path.data);
706
707     if (ngx_link_info(copy.path.data, &fi) == NGX\_FILE\_ERROR) {
708         err = ngx_errno;
709
710         if (err != NGX\_ENOENT) {
711             return ngx_http_dav_error(r->connection->log, err,
712                 NGX\_HTTP\_NOT\_FOUND, ngx_link_info_n,
713                 copy.path.data);
714         }
715
716         /* destination does not exist */
717
718         overwrite = 0;
719         dir = 0;
720
721     } else {
722
723         /* destination exists */
724
725         if (ngx_is_dir(&fi) && !slash) {
726             ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, 0,
727                 "\"%V\" could not be %Ved to collection \"%V\"",
728                 &r->uri, &r->method_name, &dest->value);
729             return NGX\_HTTP\_CONFLICT;
730         }
731
732         if (!overwrite) {
733             ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, NGX\_EEXIST,
734                 "\"%s\" could not be created", copy.path.data);
735             return NGX\_HTTP\_PRECONDITION\_FAILED;
736         }
737
738         dir = ngx_is_dir(&fi);
739     }
740
741     if (ngx_link_info(path.data, &fi) == NGX\_FILE\_ERROR) {
742         return ngx_http_dav_error(r->connection->log, ngx_errno,
743             NGX\_HTTP\_NOT\_FOUND, ngx_link_info_n,
744             path.data);
745     }
746
747     if (ngx_is_dir(&fi)) {
748
749         if (r->uri.data[r->uri.len - 1] != '/') {

```

```

750     ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
751                 "\"%V\" is collection", &r->uri);
752     return NGX_HTTP_BAD_REQUEST;
753 }
754
755 if (overwrite) {
756     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
757                 "http delete: \"%s\"", copy.path.data);
758
759     rc = ngx_http_dav_delete_path(r, &copy.path, dir);
760
761     if (rc != NGX_OK) {
762         return rc;
763     }
764 }
765 }
766
767 if (ngx_is_dir(&fi)) {
768
769     path.len -= 2; /* omit "/\0" */
770
771     if (r->method == NGX_HTTP_MOVE) {
772         if (ngx_rename_file(path.data, copy.path.data) != NGX_FILE_ERROR) {
773             return NGX_HTTP_CREATED;
774         }
775     }
776
777     if (ngx_create_dir(copy.path.data, ngx_file_access(&fi))
778         == NGX_FILE_ERROR)
779     {
780         return ngx_http_dav_error(r->connection->log, ngx_errno,
781                                 NGX_HTTP_NOT_FOUND,
782                                 ngx_create_dir_n, copy.path.data);
783     }
784
785     copy.len = path.len;
786
787     tree.init_handler = NULL;
788     tree.file_handler = ngx_http_dav_copy_tree_file;
789     tree.pre_tree_handler = ngx_http_dav_copy_dir;
790     tree.post_tree_handler = ngx_http_dav_copy_dir_time;
791     tree.spec_handler = ngx_http_dav_noop;
792     tree.data = &copy;
793     tree.alloc = 0;
794     tree.log = r->connection->log;
795
796     if (ngx_walk_tree(&tree, &path) == NGX_OK) {
797
798         if (r->method == NGX_HTTP_MOVE) {
799             rc = ngx_http_dav_delete_path(r, &path, 1);
800
801             if (rc != NGX_OK) {
802                 return rc;
803             }
804         }
805
806         return NGX_HTTP_CREATED;
807     }
808
809 } else {
810
811     if (r->method == NGX_HTTP_MOVE) {
812
813         dlcf = ngx_http_get_module_loc_conf(r, ngx_http_dav_module);
814
815         ext.access = 0;
816         ext.path_access = dlcf->access;
817         ext.time = -1;
818         ext.create_path = 1;
819         ext.delete_file = 0;
820         ext.log = r->connection->log;
821
822         if (ngx_ext_rename_file(&path, &copy.path, &ext) == NGX_OK) {
823             return NGX_HTTP_NO_CONTENT;
824         }
825

```

```

826         return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
827     }
828
829     dlcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_dav\_module);
830
831     cf.size = ngx\_file\_size(&fi);
832     cf.buf_size = 0;
833     cf.access = dlcf->access;
834     cf.time = ngx\_file\_mtime(&fi);
835     cf.log = r->connection->log;
836
837     if (ngx\_copy\_file(path.data, copy.path.data, &cf) == NGX\_OK) {
838         return NGX\_HTTP\_NO\_CONTENT;
839     }
840 }
841
842 return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
843 }
844
845
846 static ngx\_int\_t
847 ngx\_http\_dav\_copy\_dir(ngx\_tree\_ctx\_t *ctx, ngx\_str\_t *path)
848 {
849     u_char                *p, *dir;
850     size_t                len;
851     ngx\_http\_dav\_copy\_ctx\_t *copy;
852
853     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, ctx->log, 0,
854                 "http copy dir: \"%s\"", path->data);
855
856     copy = ctx->data;
857
858     len = copy->path.len + path->len;
859
860     dir = ngx\_alloc(len + 1, ctx->log);
861     if (dir == NULL) {
862         return NGX\_ABORT;
863     }
864
865     p = ngx\_cpymem(dir, copy->path.data, copy->path.len);
866     (void) ngx\_cpystrn(p, path->data + copy->len, path->len - copy->len + 1);
867
868     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, ctx->log, 0,
869                 "http copy dir to: \"%s\"", dir);
870
871     if (ngx\_create\_dir(dir, ngx\_dir\_access(ctx->access)) == NGX\_FILE\_ERROR) {
872         (void) ngx\_http\_dav\_error(ctx->log, ngx\_errno, 0, ngx\_create\_dir\_n,
873                                 dir);
874     }
875
876     ngx\_free(dir);
877
878     return NGX\_OK;
879 }
880
881
882 static ngx\_int\_t
883 ngx\_http\_dav\_copy\_dir\_time(ngx\_tree\_ctx\_t *ctx, ngx\_str\_t *path)
884 {
885     u_char                *p, *dir;
886     size_t                len;
887     ngx\_http\_dav\_copy\_ctx\_t *copy;
888
889     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, ctx->log, 0,
890                 "http copy dir time: \"%s\"", path->data);
891
892     copy = ctx->data;
893
894     len = copy->path.len + path->len;
895
896     dir = ngx\_alloc(len + 1, ctx->log);
897     if (dir == NULL) {
898         return NGX\_ABORT;
899     }
900
901     p = ngx\_cpymem(dir, copy->path.data, copy->path.len);

```

```

902     (void) ngx_cpysrn(p, path->data + copy->len, path->len - copy->len + 1);
903
904     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, ctx->log, 0,
905                 "http copy dir time to: \"%s\"", dir);
906
907     #if (NGX_WIN32)
908     {
909         ngx_fd_t fd;
910
911         fd = ngx_open_file(dir, NGX_FILE_RDWR, NGX_FILE_OPEN, 0);
912
913         if (fd == NGX_INVALID_FILE) {
914             (void) ngx_http_dav_error(ctx->log, ngx_errno, 0, ngx_open_file_n, dir);
915             goto failed;
916         }
917
918         if (ngx_set_file_time(NULL, fd, ctx->mtime) != NGX_OK) {
919             ngx_log_error(NGX_LOG_ALERT, ctx->log, ngx_errno,
920                 ngx_set_file_time_n " \"%s\" failed", dir);
921         }
922
923         if (ngx_close_file(fd) == NGX_FILE_ERROR) {
924             ngx_log_error(NGX_LOG_ALERT, ctx->log, ngx_errno,
925                 ngx_close_file_n " \"%s\" failed", dir);
926         }
927     }
928
929     failed:
930
931     #else
932
933     if (ngx_set_file_time(dir, 0, ctx->mtime) != NGX_OK) {
934         ngx_log_error(NGX_LOG_ALERT, ctx->log, ngx_errno,
935             ngx_set_file_time_n " \"%s\" failed", dir);
936     }
937
938     #endif
939
940     ngx_free(dir);
941
942     return NGX_OK;
943 }
944
945
946 static ngx_int_t
947 ngx_http_dav_copy_tree_file(ngx_tree_ctx_t *ctx, ngx_str_t *path)
948 {
949     u_char          *p, *file;
950     size_t          len;
951     ngx_copy_file_t cf;
952     ngx_http_dav_copy_ctx_t *copy;
953
954     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, ctx->log, 0,
955                 "http copy file: \"%s\"", path->data);
956
957     copy = ctx->data;
958
959     len = copy->path.len + path->len;
960
961     file = ngx_alloc(len + 1, ctx->log);
962     if (file == NULL) {
963         return NGX_ABORT;
964     }
965
966     p = ngx_cpymem(file, copy->path.data, copy->path.len);
967     (void) ngx_cpysrn(p, path->data + copy->len, path->len - copy->len + 1);
968
969     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, ctx->log, 0,
970                 "http copy file to: \"%s\"", file);
971
972     cf.size = ctx->size;
973     cf.buf_size = 0;
974     cf.access = ctx->access;
975     cf.time = ctx->mtime;
976     cf.log = ctx->log;
977

```

```

978     (void) ngx_copy_file(path->data, file, &cf);
979
980     ngx_free(file);
981
982     return NGX_OK;
983 }
984
985
986 static ngx_int_t
987 ngx_http_dav_depth(ngx_http_request_t *r, ngx_int_t dflt)
988 {
989     ngx_table_elt_t *depth;
990
991     depth = r->headers_in.depth;
992
993     if (depth == NULL) {
994         return dflt;
995     }
996
997     if (depth->value.len == 1) {
998         if (depth->value.data[0] == '0') {
999             return 0;
1000         }
1001
1002         if (depth->value.data[0] == '1') {
1003             return 1;
1004         }
1005     }
1006
1007     } else {
1008         if (depth->value.len == sizeof("infinity") - 1
1009             && ngx_strcmp(depth->value.data, "infinity") == 0)
1010         {
1011             return NGX_HTTP_DAV_INFINITY_DEPTH;
1012         }
1013     }
1014
1015     ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1016         "client sent invalid \"Depth\" header: \"%V\"",
1017         &depth->value);
1018
1019     return NGX_HTTP_DAV_INVALID_DEPTH;
1020 }
1021
1022
1023
1024 static ngx_int_t
1025 ngx_http_dav_error(ngx_log_t *log, ngx_err_t err, ngx_int_t not_found,
1026     char *failed, u_char *path)
1027 {
1028     ngx_int_t rc;
1029     ngx_uint_t level;
1030
1031     if (err == NGX_ENOENT || err == NGX_ENOTDIR || err == NGX_ENAMETOOLONG) {
1032         level = NGX_LOG_ERR;
1033         rc = not_found;
1034     }
1035     } else if (err == NGX_EACCES || err == NGX_EPERM) {
1036         level = NGX_LOG_ERR;
1037         rc = NGX_HTTP_FORBIDDEN;
1038     }
1039     } else if (err == NGX_EEXIST) {
1040         level = NGX_LOG_ERR;
1041         rc = NGX_HTTP_NOT_ALLOWED;
1042     }
1043     } else if (err == NGX_ENOSPC) {
1044         level = NGX_LOG_CRIT;
1045         rc = NGX_HTTP_INSUFFICIENT_STORAGE;
1046     }
1047     } else {
1048         level = NGX_LOG_CRIT;
1049         rc = NGX_HTTP_INTERNAL_SERVER_ERROR;
1050     }
1051
1052     ngx_log_error(level, log, err, "%s \"%s\" failed", failed, path);
1053

```





```
1130     ngx\_conf\_merge\_uint\_value(conf->access, prev->access, 0600);
1131
1132
1133     ngx\_conf\_merge\_value(conf->create_full_put_path,
1134         prev->create_full_put_path, 0);
1135
1136     return NGX\_CONF\_OK;
1137 }
1138
1139
1140 static ngx\_int\_t
1141 ngx\_http\_dav\_init(ngx\_conf\_t *cf)
1142 {
1143     ngx\_http\_handler\_pt *h;
1144     ngx\_http\_core\_main\_conf\_t *cmcf;
1145
1146     cmcf = ngx\_http\_conf\_get\_module\_main\_conf(cf, ngx\_http\_core\_module);
1147
1148     h = ngx\_array\_push(&cmcf->phases[NGX_HTTP_CONTENT_PHASE].handlers);
1149     if (h == NULL) {
1150         return NGX\_ERROR;
1151     }
1152
1153     *h = ngx\_http\_dav\_handler;
1154
1155     return NGX\_OK;
1156 }
```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_empty\_gif\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_empty\\_gif](#)
- [ngx\\_http\\_empty\\_gif\\_commands](#)
- [ngx\\_http\\_empty\\_gif\\_module](#)
- [ngx\\_http\\_empty\\_gif\\_module\\_ctx](#)
- [ngx\\_http\\_gif\\_type](#)

## Functions defined

- [ngx\\_http\\_empty\\_gif](#)
- [ngx\\_http\\_empty\\_gif\\_handler](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7 #include <ngx_config.h>
8 #include <ngx_core.h>
9 #include <ngx_http.h>
10
11
12 static char *ngx_http_empty_gif(ngx_conf_t *cf, ngx_command_t *cmd,
13     void *conf);
14
15 static ngx_command_t  ngx_http_empty_gif_commands[] = {
16
17     { ngx_string("empty_gif"),
18       NGX_HTTP_LOC_CONF|NGX_CONF_NOARGS,
19       ngx_http_empty_gif,
20       0,
21       0,
22       NULL },
23
24     ngx_null_command
25 };
26
27
28 /* the minimal single pixel transparent GIF, 43 bytes */
29
30 static u_char  ngx_empty_gif[] = {
31
32     'G', 'I', 'F', '8', '9', 'a', /* header */
33
34     /* logical screen descriptor */
35     0x01, 0x00, /* logical screen width */
36     0x01, 0x00, /* logical screen height */
37     0x80, /* global 1-bit color table */
38     0x01, /* background color #1 */
39     0x00, /* no aspect ratio */
40
41     /* global color table */
42     0x00, 0x00, 0x00, /* #0: black */
43     0xff, 0xff, 0xff, /* #1: white */
44
45     /* graphic control extension */
```

```

46     0x21,                /* extension introducer */
47     0xf9,                /* graphic control label */
48     0x04,                /* block size */
49     0x01,                /* transparent color is given, */
50                             /* no disposal specified, */
51                             /* user input is not expected */
52     0x00, 0x00,          /* delay time */
53     0x01,                /* transparent color #1 */
54     0x00,                /* block terminator */
55
56                             /* image descriptor */
57     0x2c,                /* image separator */
58     0x00, 0x00,          /* image left position */
59     0x00, 0x00,          /* image top position */
60     0x01, 0x00,          /* image width */
61     0x01, 0x00,          /* image height */
62     0x00,                /* no local color table, no interlaced */
63
64                             /* table based image data */
65     0x02,                /* LZW minimum code size, */
66                             /* must be at least 2-bit */
67     0x02,                /* block size */
68     0x4c, 0x01,          /* compressed bytes 01_001_100, 0000000_1 */
69                             /* 100: clear code */
70                             /* 001: 1 */
71                             /* 101: end of information code */
72     0x00,                /* block terminator */
73
74     0x3B                 /* trailer */
75 };
76
77
78 static ngx_http_module_t ngx_http_empty_gif_module_ctx = {
79     NULL,                /* preconfiguration */
80     NULL,                /* postconfiguration */
81
82     NULL,                /* create main configuration */
83     NULL,                /* init main configuration */
84
85     NULL,                /* create server configuration */
86     NULL,                /* merge server configuration */
87
88     NULL,                /* create location configuration */
89     NULL,                /* merge location configuration */
90 };
91
92
93 ngx_module_t ngx_http_empty_gif_module = {
94     NGX_MODULE_V1,
95     &ngx_http_empty_gif_module_ctx, /* module context */
96     ngx_http_empty_gif_commands, /* module directives */
97     NGX_HTTP_MODULE, /* module type */
98     NULL, /* init master */
99     NULL, /* init module */
100    NULL, /* init process */
101    NULL, /* init thread */
102    NULL, /* exit thread */
103    NULL, /* exit process */
104    NULL, /* exit master */
105     NGX_MODULE_V1_PADDING
106 };
107
108
109 static ngx_str_t ngx_http_gif_type = ngx_string("image/gif");
110
111
112 static ngx_int_t
113 ngx_http_empty_gif_handler(ngx_http_request_t *r)
114 {
115     ngx_http_complex_value_t cv;
116
117     if (!(r->method & (NGX_HTTP_GET|NGX_HTTP_HEAD))) {
118         return NGX_HTTP_NOT_ALLOWED;
119     }
120
121     ngx_memzero(&cv, sizeof(ngx_http_complex_value_t));

```

```
122     cv.value.len = sizeof(ngx\_empty\_gif);
123     cv.value.data = ngx\_empty\_gif;
124     r->headers_out.last_modified_time = 23349600;
125
126     return ngx\_http\_send\_response(r, NGX\_HTTP\_OK, &ngx\_http\_gif\_type, &cv);
127 }
128
129
130
131 static char *
132 ngx\_http\_empty\_gif(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
133 {
134     ngx\_http\_core\_loc\_conf\_t *clcf;
135
136     clcf = ngx\_http\_conf\_get\_module\_loc\_conf(cf, ngx\_http\_core\_module);
137     clcf->handler = ngx\_http\_empty\_gif\_handler;
138
139     return NGX\_CONF\_OK;
140 }
```

[One Level Up](#)

[Top Level](#)

## src/http/modules/nginx\_http\_fastcgi\_module.c - nginx-1.7.10

### Global variables defined

- [ngx\\_http\\_fastcgi\\_cache\\_headers](#)
- [ngx\\_http\\_fastcgi\\_commands](#)
- [ngx\\_http\\_fastcgi\\_hide\\_headers](#)
- [ngx\\_http\\_fastcgi\\_lowat\\_post](#)
- [ngx\\_http\\_fastcgi\\_module](#)
- [ngx\\_http\\_fastcgi\\_module](#)
- [ngx\\_http\\_fastcgi\\_module\\_ctx](#)
- [ngx\\_http\\_fastcgi\\_next\\_upstream\\_masks](#)
- [ngx\\_http\\_fastcgi\\_request\\_start](#)
- [ngx\\_http\\_fastcgi\\_temp\\_path](#)
- [ngx\\_http\\_fastcgi\\_vars](#)

### Data types defined

- [ngx\\_http\\_fastcgi\\_begin\\_request\\_t](#)
- [ngx\\_http\\_fastcgi\\_ctx\\_t](#)
- [ngx\\_http\\_fastcgi\\_header\\_small\\_t](#)
- [ngx\\_http\\_fastcgi\\_header\\_t](#)
- [ngx\\_http\\_fastcgi\\_loc\\_conf\\_t](#)
- [ngx\\_http\\_fastcgi\\_main\\_conf\\_t](#)
- [ngx\\_http\\_fastcgi\\_params\\_t](#)
- [ngx\\_http\\_fastcgi\\_request\\_start\\_t](#)
- [ngx\\_http\\_fastcgi\\_split\\_part\\_t](#)
- [ngx\\_http\\_fastcgi\\_state\\_e](#)

### Functions defined

- [ngx\\_http\\_fastcgi\\_abort\\_request](#)
- [ngx\\_http\\_fastcgi\\_add\\_variables](#)
- [ngx\\_http\\_fastcgi\\_cache](#)
- [ngx\\_http\\_fastcgi\\_cache\\_key](#)
- [ngx\\_http\\_fastcgi\\_create\\_key](#)
- [ngx\\_http\\_fastcgi\\_create\\_loc\\_conf](#)

- [ngx\\_http\\_fastcgi\\_create\\_main\\_conf](#)
- [ngx\\_http\\_fastcgi\\_create\\_request](#)
- [ngx\\_http\\_fastcgi\\_eval](#)
- [ngx\\_http\\_fastcgi\\_finalize\\_request](#)
- [ngx\\_http\\_fastcgi\\_handler](#)
- [ngx\\_http\\_fastcgi\\_init\\_params](#)
- [ngx\\_http\\_fastcgi\\_input\\_filter](#)
- [ngx\\_http\\_fastcgi\\_input\\_filter\\_init](#)
- [ngx\\_http\\_fastcgi\\_lowat\\_check](#)
- [ngx\\_http\\_fastcgi\\_merge\\_loc\\_conf](#)
- [ngx\\_http\\_fastcgi\\_non\\_buffered\\_filter](#)
- [ngx\\_http\\_fastcgi\\_pass](#)
- [ngx\\_http\\_fastcgi\\_path\\_info\\_variable](#)
- [ngx\\_http\\_fastcgi\\_process\\_header](#)
- [ngx\\_http\\_fastcgi\\_process\\_record](#)
- [ngx\\_http\\_fastcgi\\_reinit\\_request](#)
- [ngx\\_http\\_fastcgi\\_script\\_name\\_variable](#)
- [ngx\\_http\\_fastcgi\\_split](#)
- [ngx\\_http\\_fastcgi\\_split\\_path\\_info](#)
- [ngx\\_http\\_fastcgi\\_store](#)

## Macros defined

- [NGX\\_HTTP\\_FASTCGI\\_ABORT\\_REQUEST](#)
- [NGX\\_HTTP\\_FASTCGI\\_BEGIN\\_REQUEST](#)
- [NGX\\_HTTP\\_FASTCGI\\_DATA](#)
- [NGX\\_HTTP\\_FASTCGI\\_END\\_REQUEST](#)
- [NGX\\_HTTP\\_FASTCGI\\_KEEP\\_CONN](#)
- [NGX\\_HTTP\\_FASTCGI\\_PARAMS](#)
- [NGX\\_HTTP\\_FASTCGI\\_RESPONDER](#)
- [NGX\\_HTTP\\_FASTCGI\\_STDERR](#)
- [NGX\\_HTTP\\_FASTCGI\\_STDIN](#)
- [NGX\\_HTTP\\_FASTCGI\\_STDOUT](#)

## Source code

```

2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 typedef struct {
14     ngx_array_t             caches; /* ngx_http_file_cache_t * */
15 } ngx_http_fastcgi_main_conf_t;
16
17
18 typedef struct {
19     ngx_array_t             *flushes;
20     ngx_array_t             *lengths;
21     ngx_array_t             *values;
22     ngx_uint_t              number;
23     ngx_hash_t              hash;
24 } ngx_http_fastcgi_params_t;
25
26
27 typedef struct {
28     ngx_http_upstream_conf_t upstream;
29
30     ngx_str_t               index;
31
32     ngx_http_fastcgi_params_t params;
33 #if (NGX_HTTP_CACHE)
34     ngx_http_fastcgi_params_t params_cache;
35 #endif
36
37     ngx_array_t             *params_source;
38     ngx_array_t             *catch_stderr;
39
40     ngx_array_t             *fastcgi_lengths;
41     ngx_array_t             *fastcgi_values;
42
43     ngx_flag_t              keep_conn;
44
45 #if (NGX_HTTP_CACHE)
46     ngx_http_complex_value_t cache_key;
47 #endif
48
49 #if (NGX_PCRE)
50     ngx_regex_t             *split_regex;
51     ngx_str_t               split_name;
52 #endif
53 } ngx_http_fastcgi_loc_conf_t;
54
55
56 typedef enum {
57     ngx_http_fastcgi_st_version = 0,
58     ngx_http_fastcgi_st_type,
59     ngx_http_fastcgi_st_request_id_hi,
60     ngx_http_fastcgi_st_request_id_lo,
61     ngx_http_fastcgi_st_content_length_hi,
62     ngx_http_fastcgi_st_content_length_lo,
63     ngx_http_fastcgi_st_padding_length,
64     ngx_http_fastcgi_st_reserved,
65     ngx_http_fastcgi_st_data,
66     ngx_http_fastcgi_st_padding
67 } ngx_http_fastcgi_state_e;
68
69
70 typedef struct {
71     u_char                   *start;
72     u_char                   *end;
73 } ngx_http_fastcgi_split_part_t;
74
75
76 typedef struct {
77     ngx_http_fastcgi_state_e state;

```



```

78     u_char                *pos;
79     u_char                *last;
80     ngx_uint_t           type;
81     size_t               length;
82     size_t               padding;
83
84     unsigned              fastcgi_stdout:1;
85     unsigned              large_stderr:1;
86
87     ngx_array_t          *split_parts;
88
89     ngx_str_t            script_name;
90     ngx_str_t            path_info;
91 } ngx_http_fastcgi_ctx_t;
92
93
94 #define NGX_HTTP_FASTCGI_RESPONDER      1
95
96 #define NGX_HTTP_FASTCGI_KEEP_CONN     1
97
98 #define NGX_HTTP_FASTCGI_BEGIN_REQUEST  1
99 #define NGX_HTTP_FASTCGI_ABORT_REQUEST  2
100 #define NGX_HTTP_FASTCGI_END_REQUEST    3
101 #define NGX_HTTP_FASTCGI_PARAMS        4
102 #define NGX_HTTP_FASTCGI_STDIN         5
103 #define NGX_HTTP_FASTCGI_STDOUT        6
104 #define NGX_HTTP_FASTCGI_STDERR        7
105 #define NGX_HTTP_FASTCGI_DATA          8
106
107
108 typedef struct {
109     u_char  version;
110     u_char  type;
111     u_char  request_id_hi;
112     u_char  request_id_lo;
113     u_char  content_length_hi;
114     u_char  content_length_lo;
115     u_char  padding_length;
116     u_char  reserved;
117 } ngx_http_fastcgi_header_t;
118
119
120 typedef struct {
121     u_char  role_hi;
122     u_char  role_lo;
123     u_char  flags;
124     u_char  reserved[5];
125 } ngx_http_fastcgi_begin_request_t;
126
127
128 typedef struct {
129     u_char  version;
130     u_char  type;
131     u_char  request_id_hi;
132     u_char  request_id_lo;
133 } ngx_http_fastcgi_header_small_t;
134
135
136 typedef struct {
137     ngx_http_fastcgi_header_t  h0;
138     ngx_http_fastcgi_begin_request_t  br;
139     ngx_http_fastcgi_header_small_t  h1;
140 } ngx_http_fastcgi_request_start_t;
141
142
143 static ngx_int_t ngx_http_fastcgi_eval(ngx_http_request_t *r,
144     ngx_http_fastcgi_loc_conf_t *flcf);
145 #if (NGX_HTTP_CACHE)
146 static ngx_int_t ngx_http_fastcgi_create_key(ngx_http_request_t *r);
147 #endif
148 static ngx_int_t ngx_http_fastcgi_create_request(ngx_http_request_t *r);
149 static ngx_int_t ngx_http_fastcgi_reinit_request(ngx_http_request_t *r);
150 static ngx_int_t ngx_http_fastcgi_process_header(ngx_http_request_t *r);
151 static ngx_int_t ngx_http_fastcgi_input_filter_init(void *data);
152 static ngx_int_t ngx_http_fastcgi_input_filter(ngx_event_pipe_t *p,
153     ngx_buf_t *buf);

```

```

154 static ngx_int_t ngx_http_fastcgi_non_buffered_filter(void *data,
155     ssize_t bytes);
156 static ngx_int_t ngx_http_fastcgi_process_record(ngx_http_request_t *r,
157     ngx_http_fastcgi_ctx_t *f);
158 static void ngx_http_fastcgi_abort_request(ngx_http_request_t *r);
159 static void ngx_http_fastcgi_finalize_request(ngx_http_request_t *r,
160     ngx_int_t rc);
161
162 static ngx_int_t ngx_http_fastcgi_add_variables(ngx_conf_t *cf);
163 static void *ngx_http_fastcgi_create_main_conf(ngx_conf_t *cf);
164 static void *ngx_http_fastcgi_create_loc_conf(ngx_conf_t *cf);
165 static char *ngx_http_fastcgi_merge_loc_conf(ngx_conf_t *cf,
166     void *parent, void *child);
167 static ngx_int_t ngx_http_fastcgi_init_params(ngx_conf_t *cf,
168     ngx_http_fastcgi_loc_conf_t *conf, ngx_http_fastcgi_params_t *params,
169     ngx_keyval_t *default_params);
170
171 static ngx_int_t ngx_http_fastcgi_script_name_variable(ngx_http_request_t *r,
172     ngx_http_variable_value_t *v, uintptr_t data);
173 static ngx_int_t ngx_http_fastcgi_path_info_variable(ngx_http_request_t *r,
174     ngx_http_variable_value_t *v, uintptr_t data);
175 static ngx_http_fastcgi_ctx_t *ngx_http_fastcgi_split(ngx_http_request_t *r,
176     ngx_http_fastcgi_loc_conf_t *flcf);
177
178 static char *ngx_http_fastcgi_pass(ngx_conf_t *cf, ngx_command_t *cmd,
179     void *conf);
180 static char *ngx_http_fastcgi_split_path_info(ngx_conf_t *cf,
181     ngx_command_t *cmd, void *conf);
182 static char *ngx_http_fastcgi_store(ngx_conf_t *cf, ngx_command_t *cmd,
183     void *conf);
184 #if (NGX_HTTP_CACHE)
185 static char *ngx_http_fastcgi_cache(ngx_conf_t *cf, ngx_command_t *cmd,
186     void *conf);
187 static char *ngx_http_fastcgi_cache_key(ngx_conf_t *cf, ngx_command_t *cmd,
188     void *conf);
189 #endif
190
191 static char *ngx_http_fastcgi_lowat_check(ngx_conf_t *cf, void *post,
192     void *data);
193
194
195 static ngx_conf_post_t  ngx_http_fastcgi_lowat_post =
196     { ngx_http_fastcgi_lowat_check };
197
198
199 static ngx_conf_bitmask_t  ngx_http_fastcgi_next_upstream_masks[] = {
200     { ngx_string("error"), NGX_HTTP_UPSTREAM_FT_ERROR },
201     { ngx_string("timeout"), NGX_HTTP_UPSTREAM_FT_TIMEOUT },
202     { ngx_string("invalid_header"), NGX_HTTP_UPSTREAM_FT_INVALID_HEADER },
203     { ngx_string("http_500"), NGX_HTTP_UPSTREAM_FT_HTTP_500 },
204     { ngx_string("http_503"), NGX_HTTP_UPSTREAM_FT_HTTP_503 },
205     { ngx_string("http_403"), NGX_HTTP_UPSTREAM_FT_HTTP_403 },
206     { ngx_string("http_404"), NGX_HTTP_UPSTREAM_FT_HTTP_404 },
207     { ngx_string("updating"), NGX_HTTP_UPSTREAM_FT_UPDATING },
208     { ngx_string("off"), NGX_HTTP_UPSTREAM_FT_OFF },
209     { ngx_null_string, 0 }
210 };
211
212
213 ngx_module_t  ngx_http_fastcgi_module;
214
215
216 static ngx_command_t  ngx_http_fastcgi_commands[] = {
217
218     { ngx_string("fastcgi_pass"),
219         NGX_HTTP_LOC_CONF|NGX_HTTP_LIF_CONF|NGX_CONF_TAKE1,
220         ngx_http_fastcgi_pass,
221         NGX_HTTP_LOC_CONF_OFFSET,
222         0,
223         NULL },
224
225     { ngx_string("fastcgi_index"),
226         NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
227         ngx_conf_set_str_slot,
228         NGX_HTTP_LOC_CONF_OFFSET,
229         offsetof(ngx_http_fastcgi_loc_conf_t, index),

```

```

230     NULL },
231
232 { ngx_string("fastcgi_split_path_info"),
233     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF TAKE1,
234     ngx\_http\_fastcgi\_split\_path\_info,
235     NGX\_HTTP\_LOC\_CONF\_OFFSET,
236     0,
237     NULL },
238
239 { ngx_string("fastcgi_store"),
240     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF TAKE1,
241     ngx\_http\_fastcgi\_store,
242     NGX\_HTTP\_LOC\_CONF\_OFFSET,
243     0,
244     NULL },
245
246 { ngx_string("fastcgi_store_access"),
247     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF TAKE123,
248     ngx\_conf\_set\_access\_slot,
249     NGX\_HTTP\_LOC\_CONF\_OFFSET,
250     offsetof\(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.store\_access\),
251     NULL },
252
253 { ngx_string("fastcgi_buffering"),
254     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF FLAG,
255     ngx\_conf\_set\_flag\_slot,
256     NGX\_HTTP\_LOC\_CONF\_OFFSET,
257     offsetof\(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.buffering\),
258     NULL },
259
260 { ngx_string("fastcgi_ignore_client_abort"),
261     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF FLAG,
262     ngx\_conf\_set\_flag\_slot,
263     NGX\_HTTP\_LOC\_CONF\_OFFSET,
264     offsetof\(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.ignore\_client\_abort\),
265     NULL },
266
267 { ngx_string("fastcgi_bind"),
268     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF TAKE1,
269     ngx\_http\_upstream\_bind\_set\_slot,
270     NGX\_HTTP\_LOC\_CONF\_OFFSET,
271     offsetof\(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.local\),
272     NULL },
273
274 { ngx_string("fastcgi_connect_timeout"),
275     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF TAKE1,
276     ngx\_conf\_set\_msec\_slot,
277     NGX\_HTTP\_LOC\_CONF\_OFFSET,
278     offsetof\(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.connect\_timeout\),
279     NULL },
280
281 { ngx_string("fastcgi_send_timeout"),
282     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF TAKE1,
283     ngx\_conf\_set\_msec\_slot,
284     NGX\_HTTP\_LOC\_CONF\_OFFSET,
285     offsetof\(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.send\_timeout\),
286     NULL },
287
288 { ngx_string("fastcgi_send_lowat"),
289     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF TAKE1,
290     ngx\_conf\_set\_size\_slot,
291     NGX\_HTTP\_LOC\_CONF\_OFFSET,
292     offsetof\(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.send\_lowat\),
293     &ngx\_http\_fastcgi\_lowat\_post },
294
295 { ngx_string("fastcgi_buffer_size"),
296     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF TAKE1,
297     ngx\_conf\_set\_size\_slot,
298     NGX\_HTTP\_LOC\_CONF\_OFFSET,
299     offsetof\(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.buffer\_size\),
300     NULL },
301
302 { ngx_string("fastcgi_pass_request_headers"),
303     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF FLAG,
304     ngx\_conf\_set\_flag\_slot,
305     NGX\_HTTP\_LOC\_CONF\_OFFSET,

```

```

306     offsetof(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.pass_request_headers),
307     NULL },
308
309 { ngx\_string\("fastcgi\_pass\_request\_body"\),
310   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_FLAG,
311   ngx\_conf\_set\_flag\_slot,
312   NGX\_HTTP\_LOC\_CONF\_OFFSET,
313   offsetof(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.pass_request_body),
314   NULL },
315
316 { ngx\_string\("fastcgi\_intercept\_errors"\),
317   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_FLAG,
318   ngx\_conf\_set\_flag\_slot,
319   NGX\_HTTP\_LOC\_CONF\_OFFSET,
320   offsetof(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.intercept_errors),
321   NULL },
322
323 { ngx\_string\("fastcgi\_read\_timeout"\),
324   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
325   ngx\_conf\_set\_msec\_slot,
326   NGX\_HTTP\_LOC\_CONF\_OFFSET,
327   offsetof(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.read_timeout),
328   NULL },
329
330 { ngx\_string\("fastcgi\_buffers"\),
331   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE2,
332   ngx\_conf\_set\_bufs\_slot,
333   NGX\_HTTP\_LOC\_CONF\_OFFSET,
334   offsetof(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.bufs),
335   NULL },
336
337 { ngx\_string\("fastcgi\_busy\_buffers\_size"\),
338   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
339   ngx\_conf\_set\_size\_slot,
340   NGX\_HTTP\_LOC\_CONF\_OFFSET,
341   offsetof(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.busy_buffers_size_conf),
342   NULL },
343
344 { ngx\_string\("fastcgi\_force\_ranges"\),
345   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_FLAG,
346   ngx\_conf\_set\_flag\_slot,
347   NGX\_HTTP\_LOC\_CONF\_OFFSET,
348   offsetof(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.force_ranges),
349   NULL },
350
351 { ngx\_string\("fastcgi\_limit\_rate"\),
352   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
353   ngx\_conf\_set\_size\_slot,
354   NGX\_HTTP\_LOC\_CONF\_OFFSET,
355   offsetof(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.limit_rate),
356   NULL },
357
358 #if (NGX_HTTP_CACHE)
359
360 { ngx\_string\("fastcgi\_cache"\),
361   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
362   ngx\_http\_fastcgi\_cache,
363   NGX\_HTTP\_LOC\_CONF\_OFFSET,
364   0,
365   NULL },
366
367 { ngx\_string\("fastcgi\_cache\_key"\),
368   NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
369   ngx\_http\_fastcgi\_cache\_key,
370   NGX\_HTTP\_LOC\_CONF\_OFFSET,
371   0,
372   NULL },
373
374 { ngx\_string\("fastcgi\_cache\_path"\),
375   NGX\_HTTP\_MAIN\_CONF|NGX\_CONF\_2MORE,
376   ngx\_http\_file\_cache\_set\_slot,
377   NGX\_HTTP\_MAIN\_CONF\_OFFSET,
378   offsetof(ngx\_http\_fastcgi\_main\_conf\_t, caches),
379   &ngx\_http\_fastcgi\_module },
380
381 { ngx\_string\("fastcgi\_cache\_bypass"\),

```

```

382 NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_1MORE,
383 ngx\_http\_set\_predicate\_slot,
384 NGX\_HTTP\_LOC\_CONF\_OFFSET,
385 offsetof\(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.cache\_bypass\),
386 NULL },
387
388 { ngx\_string\("fastcgi\_no\_cache"\),
389 NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_1MORE,
390 ngx\_http\_set\_predicate\_slot,
391 NGX\_HTTP\_LOC\_CONF\_OFFSET,
392 offsetof\(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.no\_cache\),
393 NULL },
394
395 { ngx\_string\("fastcgi\_cache\_valid"\),
396 NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_1MORE,
397 ngx\_http\_file\_cache\_valid\_set\_slot,
398 NGX\_HTTP\_LOC\_CONF\_OFFSET,
399 offsetof\(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.cache\_valid\),
400 NULL },
401
402 { ngx\_string\("fastcgi\_cache\_min\_uses"\),
403 NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
404 ngx\_conf\_set\_num\_slot,
405 NGX\_HTTP\_LOC\_CONF\_OFFSET,
406 offsetof\(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.cache\_min\_uses\),
407 NULL },
408
409 { ngx\_string\("fastcgi\_cache\_use\_stale"\),
410 NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_1MORE,
411 ngx\_conf\_set\_bitmask\_slot,
412 NGX\_HTTP\_LOC\_CONF\_OFFSET,
413 offsetof\(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.cache\_use\_stale\),
414 &ngx\_http\_fastcgi\_next\_upstream\_masks },
415
416 { ngx\_string\("fastcgi\_cache\_methods"\),
417 NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_1MORE,
418 ngx\_conf\_set\_bitmask\_slot,
419 NGX\_HTTP\_LOC\_CONF\_OFFSET,
420 offsetof\(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.cache\_methods\),
421 &ngx\_http\_upstream\_cache\_method\_mask },
422
423 { ngx\_string\("fastcgi\_cache\_lock"\),
424 NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_FLAG,
425 ngx\_conf\_set\_flag\_slot,
426 NGX\_HTTP\_LOC\_CONF\_OFFSET,
427 offsetof\(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.cache\_lock\),
428 NULL },
429
430 { ngx\_string\("fastcgi\_cache\_lock\_timeout"\),
431 NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
432 ngx\_conf\_set\_msec\_slot,
433 NGX\_HTTP\_LOC\_CONF\_OFFSET,
434 offsetof\(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.cache\_lock\_timeout\),
435 NULL },
436
437 { ngx\_string\("fastcgi\_cache\_lock\_age"\),
438 NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
439 ngx\_conf\_set\_msec\_slot,
440 NGX\_HTTP\_LOC\_CONF\_OFFSET,
441 offsetof\(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.cache\_lock\_age\),
442 NULL },
443
444 { ngx\_string\("fastcgi\_cache\_revalidate"\),
445 NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_FLAG,
446 ngx\_conf\_set\_flag\_slot,
447 NGX\_HTTP\_LOC\_CONF\_OFFSET,
448 offsetof\(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.cache\_revalidate\),
449 NULL },
450
451 #endif
452
453 { ngx\_string\("fastcgi\_temp\_path"\),
454 NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1234,
455 ngx\_conf\_set\_path\_slot,
456 NGX\_HTTP\_LOC\_CONF\_OFFSET,
457 offsetof\(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.temp\_path\),

```

```

458     NULL },
459
460 { ngx_string("fastcgi_max_temp_file_size"),
461     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF TAKE1,
462     ngx\_conf\_set\_size\_slot,
463     NGX\_HTTP\_LOC\_CONF\_OFFSET,
464     offsetof(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.max_temp_file_size_conf),
465     NULL },
466
467 { ngx_string("fastcgi_temp_file_write_size"),
468     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF TAKE1,
469     ngx\_conf\_set\_size\_slot,
470     NGX\_HTTP\_LOC\_CONF\_OFFSET,
471     offsetof(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.temp_file_write_size_conf),
472     NULL },
473
474 { ngx_string("fastcgi_next_upstream"),
475     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF 1MORE,
476     ngx\_conf\_set\_bitmask\_slot,
477     NGX\_HTTP\_LOC\_CONF\_OFFSET,
478     offsetof(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.next_upstream),
479     &ngx\_http\_fastcgi\_next\_upstream\_masks },
480
481 { ngx_string("fastcgi_next_upstream_tries"),
482     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF TAKE1,
483     ngx\_conf\_set\_num\_slot,
484     NGX\_HTTP\_LOC\_CONF\_OFFSET,
485     offsetof(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.next_upstream_tries),
486     NULL },
487
488 { ngx_string("fastcgi_next_upstream_timeout"),
489     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF TAKE1,
490     ngx\_conf\_set\_msec\_slot,
491     NGX\_HTTP\_LOC\_CONF\_OFFSET,
492     offsetof(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.next_upstream_timeout),
493     NULL },
494
495 { ngx_string("fastcgi_param"),
496     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF TAKE23,
497     ngx\_http\_upstream\_param\_set\_slot,
498     NGX\_HTTP\_LOC\_CONF\_OFFSET,
499     offsetof(ngx\_http\_fastcgi\_loc\_conf\_t, params_source),
500     NULL },
501
502 { ngx_string("fastcgi_pass_header"),
503     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF TAKE1,
504     ngx\_conf\_set\_str\_array\_slot,
505     NGX\_HTTP\_LOC\_CONF\_OFFSET,
506     offsetof(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.pass_headers),
507     NULL },
508
509 { ngx_string("fastcgi_hide_header"),
510     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF TAKE1,
511     ngx\_conf\_set\_str\_array\_slot,
512     NGX\_HTTP\_LOC\_CONF\_OFFSET,
513     offsetof(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.hide_headers),
514     NULL },
515
516 { ngx_string("fastcgi_ignore_headers"),
517     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF 1MORE,
518     ngx\_conf\_set\_bitmask\_slot,
519     NGX\_HTTP\_LOC\_CONF\_OFFSET,
520     offsetof(ngx\_http\_fastcgi\_loc\_conf\_t, upstream.ignore_headers),
521     &ngx\_http\_upstream\_ignore\_headers\_masks },
522
523 { ngx_string("fastcgi_catch_stderr"),
524     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF TAKE1,
525     ngx\_conf\_set\_str\_array\_slot,
526     NGX\_HTTP\_LOC\_CONF\_OFFSET,
527     offsetof(ngx\_http\_fastcgi\_loc\_conf\_t, catch_stderr),
528     NULL },
529
530 { ngx_string("fastcgi_keep_conn"),
531     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF FLAG,
532     ngx\_conf\_set\_flag\_slot,
533     NGX\_HTTP\_LOC\_CONF\_OFFSET,

```

```

534     offsetof(ngx_http_fastcgi_loc_conf_t, keep_conn),
535     NULL },
536
537     ngx_null_command
538 };
539
540
541 static ngx_http_module_t ngx_http_fastcgi_module_ctx = {
542     ngx_http_fastcgi_add_variables,      /* preconfiguration */
543     NULL,                                  /* postconfiguration */
544
545     ngx_http_fastcgi_create_main_conf,  /* create main configuration */
546     NULL,                                  /* init main configuration */
547
548     NULL,                                  /* create server configuration */
549     NULL,                                  /* merge server configuration */
550
551     ngx_http_fastcgi_create_loc_conf,   /* create location configuration */
552     ngx_http_fastcgi_merge_loc_conf    /* merge location configuration */
553 };
554
555
556 ngx_module_t ngx_http_fastcgi_module = {
557     NGX_MODULE_V1,
558     &ngx_http_fastcgi_module_ctx,      /* module context */
559     ngx_http_fastcgi_commands,      /* module directives */
560     NGX_HTTP_MODULE,                /* module type */
561     NULL,                               /* init master */
562     NULL,                               /* init module */
563     NULL,                               /* init process */
564     NULL,                               /* init thread */
565     NULL,                               /* exit thread */
566     NULL,                               /* exit process */
567     NULL,                               /* exit master */
568     NGX_MODULE_V1_PADDING
569 };
570
571
572 static ngx_http_fastcgi_request_start_t ngx_http_fastcgi_request_start = {
573     { 1,                                  /* version */
574       NGX_HTTP_FASTCGI_BEGIN_REQUEST,  /* type */
575       0,                                  /* request_id_hi */
576       1,                                  /* request_id_lo */
577       0,                                  /* content_length_hi */
578       sizeof(ngx_http_fastcgi_begin_request_t), /* content_length_lo */
579       0,                                  /* padding_length */
580       0 },                                /* reserved */
581
582     { 0,                                  /* role_hi */
583       NGX_HTTP_FASTCGI_RESPONDER,      /* role_lo */
584       0, /* NGX_HTTP_FASTCGI_KEEP_CONN */ /* flags */
585       { 0, 0, 0, 0, 0 } },              /* reserved[5] */
586
587     { 1,                                  /* version */
588       NGX_HTTP_FASTCGI_PARAMS,        /* type */
589       0,                                  /* request_id_hi */
590       1 },                                /* request_id_lo */
591
592 };
593
594
595 static ngx_http_variable_t ngx_http_fastcgi_vars[] = {
596
597     { ngx_string("fastcgi_script_name"), NULL,
598       ngx_http_fastcgi_script_name_variable, 0,
599       NGX_HTTP_VAR_NOCACHEABLE|NGX_HTTP_VAR_NOHASH, 0 },
600
601     { ngx_string("fastcgi_path_info"), NULL,
602       ngx_http_fastcgi_path_info_variable, 0,
603       NGX_HTTP_VAR_NOCACHEABLE|NGX_HTTP_VAR_NOHASH, 0 },
604
605     { ngx_null_string, NULL, NULL, 0, 0, 0 }
606 };
607
608
609 static ngx_str_t ngx_http_fastcgi_hide_headers[] = {

```

```

610     ngx_string("Status"),
611     ngx_string("X-Accel-Expires"),
612     ngx_string("X-Accel-Redirect"),
613     ngx_string("X-Accel-Limit-Rate"),
614     ngx_string("X-Accel-Buffering"),
615     ngx_string("X-Accel-Charset"),
616     ngx_null_string
617 };
618
619 #if (NGX_HTTP_CACHE)
620
621 static ngx_keyval_t ngx_http_fastcgi_cache_headers[] = {
622     { ngx_string("HTTP_IF_MODIFIED_SINCE"),
623       ngx_string("$upstream_cache_last_modified") },
624     { ngx_string("HTTP_IF_UNMODIFIED_SINCE"), ngx_string("") },
625     { ngx_string("HTTP_IF_NONE_MATCH"), ngx_string("$upstream_cache_etag") },
626     { ngx_string("HTTP_IF_MATCH"), ngx_string("") },
627     { ngx_string("HTTP_RANGE"), ngx_string("") },
628     { ngx_string("HTTP_IF_RANGE"), ngx_string("") },
629     { ngx_null_string, ngx_null_string }
630 };
631 };
632
633 #endif
634
635
636 static ngx_path_init_t ngx_http_fastcgi_temp_path = {
637     ngx_string(NGX_HTTP_FASTCGI_TEMP_PATH), { 1, 2, 0 }
638 };
639
640
641 static ngx_int_t
642 ngx_http_fastcgi_handler(ngx_http_request_t *r)
643 {
644     ngx_int_t          rc;
645     ngx_http_upstream_t *u;
646     ngx_http_fastcgi_ctx_t *f;
647     ngx_http_fastcgi_loc_conf_t *flcf;
648 #if (NGX_HTTP_CACHE)
649     ngx_http_fastcgi_main_conf_t *fmcf;
650 #endif
651
652     if (ngx_http_upstream_create(r) != NGX_OK) {
653         return NGX_HTTP_INTERNAL_SERVER_ERROR;
654     }
655
656     f = ngx_palloc(r->pool, sizeof(ngx_http_fastcgi_ctx_t));
657     if (f == NULL) {
658         return NGX_HTTP_INTERNAL_SERVER_ERROR;
659     }
660
661     ngx_http_set_ctx(r, f, ngx_http_fastcgi_module);
662
663     flcf = ngx_http_get_module_loc_conf(r, ngx_http_fastcgi_module);
664
665     if (flcf->fastcgi_lengths) {
666         if (ngx_http_fastcgi_eval(r, flcf) != NGX_OK) {
667             return NGX_HTTP_INTERNAL_SERVER_ERROR;
668         }
669     }
670
671     u = r->upstream;
672
673     ngx_str_set(&u->schema, "fastcgi://");
674     u->output.tag = (ngx_buf_tag_t) &ngx_http_fastcgi_module;
675
676     u->conf = &flcf->upstream;
677
678 #if (NGX_HTTP_CACHE)
679     fmcf = ngx_http_get_module_main_conf(r, ngx_http_fastcgi_module);
680
681     u->caches = &fmcf->caches;
682     u->create_key = ngx_http_fastcgi_create_key;
683 #endif
684
685     u->create_request = ngx_http_fastcgi_create_request;

```



```

686     u->reinit_request = ngx_http_fastcgi_reinit_request;
687     u->process_header = ngx_http_fastcgi_process_header;
688     u->abort_request = ngx_http_fastcgi_abort_request;
689     u->finalize_request = ngx_http_fastcgi_finalize_request;
690     r->state = 0;
691
692     u->buffering = flcf->upstream.buffering;
693
694     u->pipe = ngx_palloc(r->pool, sizeof(ngx_event_pipe_t));
695     if (u->pipe == NULL) {
696         return NGX_HTTP_INTERNAL_SERVER_ERROR;
697     }
698
699     u->pipe->input_filter = ngx_http_fastcgi_input_filter;
700     u->pipe->input_ctx = r;
701
702     u->input_filter_init = ngx_http_fastcgi_input_filter_init;
703     u->input_filter = ngx_http_fastcgi_non_buffered_filter;
704     u->input_filter_ctx = r;
705
706     rc = ngx_http_read_client_request_body(r, ngx_http_upstream_init);
707
708     if (rc >= NGX_HTTP_SPECIAL_RESPONSE) {
709         return rc;
710     }
711
712     return NGX_DONE;
713 }
714
715
716 static ngx_int_t
717 ngx_http_fastcgi_eval(ngx_http_request_t *r, ngx_http_fastcgi_loc_conf_t *flcf)
718 {
719     ngx_url_t          url;
720     ngx_http_upstream_t *u;
721
722     ngx_memzero(&url, sizeof(ngx_url_t));
723
724     if (ngx_http_script_run(r, &url.url, flcf->fastcgi_lengths->elts, 0,
725                             flcf->fastcgi_values->elts)
726         == NULL)
727     {
728         return NGX_ERROR;
729     }
730
731     url.no_resolve = 1;
732
733     if (ngx_parse_url(r->pool, &url) != NGX_OK) {
734         if (url.err) {
735             ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
736                          "%s in upstream \"%V\"", url.err, &url.url);
737         }
738
739         return NGX_ERROR;
740     }
741
742     u = r->upstream;
743
744     u->resolved = ngx_palloc(r->pool, sizeof(ngx_http_upstream_resolved_t));
745     if (u->resolved == NULL) {
746         return NGX_ERROR;
747     }
748
749     if (url.addrs && url.addrs[0].sockaddr) {
750         u->resolved->sockaddr = url.addrs[0].sockaddr;
751         u->resolved->socklen = url.addrs[0].socklen;
752         u->resolved->naddrs = 1;
753         u->resolved->host = url.addrs[0].name;
754     } else {
755         u->resolved->host = url.host;
756         u->resolved->port = url.port;
757         u->resolved->no_port = url.no_port;
758     }
759
760     return NGX_OK;
761

```

```

762 }
763
764
765 #if (NGX_HTTP_CACHE)
766
767 static ngx_int_t
768 ngx_http_fastcgi_create_key(ngx_http_request_t *r)
769 {
770     ngx_str_t          *key;
771     ngx_http_fastcgi_loc_conf_t *flcf;
772
773     key = ngx_array_push(&r->cache->keys);
774     if (key == NULL) {
775         return NGX_ERROR;
776     }
777
778     flcf = ngx_http_get_module_loc_conf(r, ngx_http_fastcgi_module);
779
780     if (ngx_http_complex_value(r, &flcf->cache_key, key) != NGX_OK) {
781         return NGX_ERROR;
782     }
783
784     return NGX_OK;
785 }
786
787 #endif
788
789
790 static ngx_int_t
791 ngx_http_fastcgi_create_request(ngx_http_request_t *r)
792 {
793     off_t                file_pos;
794     u_char               ch, *pos, *lowercase_key;
795     size_t               size, len, key_len, val_len, padding,
796                         allocated;
797     ngx_uint_t           i, n, next, hash, skip_empty, header_params;
798     ngx_buf_t            *b;
799     ngx_chain_t          *cl, *body;
800     ngx_list_part_t      *part;
801     ngx_table_elt_t      *header, **ignored;
802     ngx_http_script_code_pt code;
803     ngx_http_script_engine_t e, le;
804     ngx_http_fastcgi_header_t *h;
805     ngx_http_fastcgi_params_t *params;
806     ngx_http_fastcgi_loc_conf_t *flcf;
807     ngx_http_script_len_code_pt lcode;
808
809     len = 0;
810     header_params = 0;
811     ignored = NULL;
812
813     flcf = ngx_http_get_module_loc_conf(r, ngx_http_fastcgi_module);
814
815     #if (NGX_HTTP_CACHE)
816     params = r->upstream->cacheable ? &flcf->params_cache : &flcf->params;
817     #else
818     params = &flcf->params;
819     #endif
820
821     if (params->lengths) {
822         ngx_memzero(&le, sizeof(ngx_http_script_engine_t));
823
824         ngx_http_script_flush_no_cacheable_variables(r, params->flushes);
825         le.flushed = 1;
826
827         le.ip = params->lengths->elts;
828         le.request = r;
829
830         while (*(uintptr_t *) le.ip) {
831
832             lcode = *(ngx_http_script_len_code_pt *) le.ip;
833             key_len = lcode(&le);
834
835             lcode = *(ngx_http_script_len_code_pt *) le.ip;
836             skip_empty = lcode(&le);
837

```

```

838     for (val_len = 0; *(uintptr_t *) le.ip; val_len += lcode(&le)) {
839         lcode = *(ngx_http_script_len_code_pt *) le.ip;
840     }
841     le.ip += sizeof(uintptr_t);
842
843     if (skip_empty && val_len == 0) {
844         continue;
845     }
846
847     len += 1 + key_len + ((val_len > 127) ? 4 : 1) + val_len;
848 }
849 }
850
851 if (flcf->upstream.pass_request_headers) {
852     allocated = 0;
853     lowercase_key = NULL;
854
855     if (params->number) {
856         n = 0;
857         part = &r->headers_in.headers.part;
858
859         while (part) {
860             n += part->nelts;
861             part = part->next;
862         }
863
864         ignored = ngx_palloc(r->pool, n * sizeof(void *));
865         if (ignored == NULL) {
866             return NGX_ERROR;
867         }
868     }
869
870     part = &r->headers_in.headers.part;
871     header = part->elts;
872
873     for (i = 0; /* void */; i++) {
874
875         if (i >= part->nelts) {
876             if (part->next == NULL) {
877                 break;
878             }
879
880             part = part->next;
881             header = part->elts;
882             i = 0;
883         }
884
885         if (params->number) {
886             if (allocated < header[i].key.len) {
887                 allocated = header[i].key.len + 16;
888                 lowercase_key = ngx_pnalloc(r->pool, allocated);
889                 if (lowercase_key == NULL) {
890                     return NGX_ERROR;
891                 }
892             }
893         }
894
895         hash = 0;
896
897         for (n = 0; n < header[i].key.len; n++) {
898             ch = header[i].key.data[n];
899
900             if (ch >= 'A' && ch <= 'Z') {
901                 ch |= 0x20;
902             } else if (ch == '-') {
903                 ch = '_';
904             }
905
906             hash = ngx_hash(hash, ch);
907             lowercase_key[n] = ch;
908         }
909
910         if (ngx_hash_find(&params->hash, hash, lowercase_key, n)) {
911             ignored[header_params++] = &header[i];
912             continue;
913         }

```

```

914         }
915
916         n += sizeof("HTTP_") - 1;
917
918     } else {
919         n = sizeof("HTTP_") - 1 + header[i].key.len;
920     }
921
922     len += ((n > 127) ? 4 : 1) + ((header[i].value.len > 127) ? 4 : 1)
923         + n + header[i].value.len;
924 }
925 }
926
927
928 if (len > 65535) {
929     ngx_log_error(NGX_LOG_ALERT, r->connection->log, 0,
930         "fastcgi request record is too big: %uz", len);
931     return NGX_ERROR;
932 }
933
934
935 padding = 8 - len % 8;
936 padding = (padding == 8) ? 0 : padding;
937
938
939 size = sizeof(ngx_http_fastcgi_header_t)
940     + sizeof(ngx_http_fastcgi_begin_request_t)
941
942     + sizeof(ngx_http_fastcgi_header_t) /* NGX_HTTP_FASTCGI_PARAMS */
943     + len + padding
944     + sizeof(ngx_http_fastcgi_header_t) /* NGX_HTTP_FASTCGI_PARAMS */
945
946     + sizeof(ngx_http_fastcgi_header_t); /* NGX_HTTP_FASTCGI_STDIN */
947
948
949 b = ngx_create_temp_buf(r->pool, size);
950 if (b == NULL) {
951     return NGX_ERROR;
952 }
953
954 cl = ngx_alloc_chain_link(r->pool);
955 if (cl == NULL) {
956     return NGX_ERROR;
957 }
958
959 cl->buf = b;
960
961 ngx_http_fastcgi_request_start.br.flags =
962     flcf->keep_conn ? NGX_HTTP_FASTCGI_KEEP_CONN : 0;
963
964 ngx_memcpy(b->pos, &ngx_http_fastcgi_request_start,
965     sizeof(ngx_http_fastcgi_request_start_t));
966
967 h = (ngx_http_fastcgi_header_t *)
968     (b->pos + sizeof(ngx_http_fastcgi_header_t)
969     + sizeof(ngx_http_fastcgi_begin_request_t));
970
971 h->content_length_hi = (u_char) ((len >> 8) & 0xff);
972 h->content_length_lo = (u_char) (len & 0xff);
973 h->padding_length = (u_char) padding;
974 h->reserved = 0;
975
976 b->last = b->pos + sizeof(ngx_http_fastcgi_header_t)
977     + sizeof(ngx_http_fastcgi_begin_request_t)
978     + sizeof(ngx_http_fastcgi_header_t);
979
980
981 if (params->lengths) {
982     ngx_memzero(&e, sizeof(ngx_http_script_engine_t));
983
984     e.ip = params->values->elts;
985     e.pos = b->last;
986     e.request = r;
987     e.flushed = 1;
988
989     le.ip = params->lengths->elts;

```

```

990
991 while (*(uintptr_t *) le.ip) {
992
993     lcode = *(ngx_http_script_len_code_pt *) le.ip;
994     key_len = (u_char) lcode(&le);
995
996     lcode = *(ngx_http_script_len_code_pt *) le.ip;
997     skip_empty = lcode(&le);
998
999     for (val_len = 0; *(uintptr_t *) le.ip; val_len += lcode(&le)) {
1000         lcode = *(ngx_http_script_len_code_pt *) le.ip;
1001     }
1002     le.ip += sizeof(uintptr_t);
1003
1004     if (skip_empty && val_len == 0) {
1005         e.skip = 1;
1006
1007         while (*(uintptr_t *) e.ip) {
1008             code = *(ngx_http_script_code_pt *) e.ip;
1009             code((ngx_http_script_engine_t *) &e);
1010         }
1011         e.ip += sizeof(uintptr_t);
1012
1013         e.skip = 0;
1014
1015         continue;
1016     }
1017
1018     *e.pos++ = (u_char) key_len;
1019
1020     if (val_len > 127) {
1021         *e.pos++ = (u_char) (((val_len >> 24) & 0x7f) | 0x80);
1022         *e.pos++ = (u_char) ((val_len >> 16) & 0xff);
1023         *e.pos++ = (u_char) ((val_len >> 8) & 0xff);
1024         *e.pos++ = (u_char) (val_len & 0xff);
1025
1026     } else {
1027         *e.pos++ = (u_char) val_len;
1028     }
1029
1030     while (*(uintptr_t *) e.ip) {
1031         code = *(ngx_http_script_code_pt *) e.ip;
1032         code((ngx_http_script_engine_t *) &e);
1033     }
1034     e.ip += sizeof(uintptr_t);
1035
1036     ngx_log_debug4(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1037                  "fastcgi param: \"%s: %s\"",
1038                  key_len, e.pos - (key_len + val_len),
1039                  val_len, e.pos - val_len);
1040 }
1041
1042 b->last = e.pos;
1043 }
1044
1045
1046 if (flcf->upstream.pass_request_headers) {
1047
1048     part = &r->headers_in.headers.part;
1049     header = part->elts;
1050
1051     for (i = 0; /* void */; i++) {
1052
1053         if (i >= part->nelts) {
1054             if (part->next == NULL) {
1055                 break;
1056             }
1057
1058             part = part->next;
1059             header = part->elts;
1060             i = 0;
1061         }
1062
1063         for (n = 0; n < header_params; n++) {
1064             if (&header[i] == ignored[n]) {
1065                 goto next;

```

```

1066     }
1067 }
1068
1069 key_len = sizeof("HTTP_") - 1 + header[i].key.len;
1070 if (key_len > 127) {
1071     *b->last++ = (u_char) (((key_len >> 24) & 0x7f) | 0x80);
1072     *b->last++ = (u_char) ((key_len >> 16) & 0xff);
1073     *b->last++ = (u_char) ((key_len >> 8) & 0xff);
1074     *b->last++ = (u_char) (key_len & 0xff);
1075
1076 } else {
1077     *b->last++ = (u_char) key_len;
1078 }
1079
1080 val_len = header[i].value.len;
1081 if (val_len > 127) {
1082     *b->last++ = (u_char) (((val_len >> 24) & 0x7f) | 0x80);
1083     *b->last++ = (u_char) ((val_len >> 16) & 0xff);
1084     *b->last++ = (u_char) ((val_len >> 8) & 0xff);
1085     *b->last++ = (u_char) (val_len & 0xff);
1086
1087 } else {
1088     *b->last++ = (u_char) val_len;
1089 }
1090
1091 b->last = ngx_cpymem(b->last, "HTTP_", sizeof("HTTP_") - 1);
1092
1093 for (n = 0; n < header[i].key.len; n++) {
1094     ch = header[i].key.data[n];
1095
1096     if (ch >= 'a' && ch <= 'z') {
1097         ch &= ~0x20;
1098
1099     } else if (ch == '-') {
1100         ch = '_';
1101     }
1102
1103     *b->last++ = ch;
1104 }
1105
1106 b->last = ngx_copy(b->last, header[i].value.data, val_len);
1107
1108 ngx_log_debug4(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1109     "fastcgi param: \"%s: %s\"",
1110     key_len, b->last - (key_len + val_len),
1111     val_len, b->last - val_len);
1112 next:
1113
1114     continue;
1115 }
1116 }
1117
1118
1119 if (padding) {
1120     ngx_memzero(b->last, padding);
1121     b->last += padding;
1122 }
1123
1124
1125 h = (ngx_http_fastcgi_header_t *) b->last;
1126 b->last += sizeof(ngx_http_fastcgi_header_t);
1127
1128 h->version = 1;
1129 h->type = NGX_HTTP_FASTCGI_PARAMS;
1130 h->request_id_hi = 0;
1131 h->request_id_lo = 1;
1132 h->content_length_hi = 0;
1133 h->content_length_lo = 0;
1134 h->padding_length = 0;
1135 h->reserved = 0;
1136
1137 h = (ngx_http_fastcgi_header_t *) b->last;
1138 b->last += sizeof(ngx_http_fastcgi_header_t);
1139
1140 if (flcf->upstream.pass_request_body) {
1141     body = r->upstream->request_bufs;

```

```

1142     r->upstream->request_bufs = cl;
1143
1144     #if (NGX_SUPPRESS_WARN)
1145         file_pos = 0;
1146         pos = NULL;
1147     #endif
1148
1149     while (body) {
1150
1151         if (body->buf->in_file) {
1152             file_pos = body->buf->file_pos;
1153
1154         } else {
1155             pos = body->buf->pos;
1156         }
1157
1158         next = 0;
1159
1160         do {
1161             b = ngx_alloc_buf(r->pool);
1162             if (b == NULL) {
1163                 return NGX_ERROR;
1164             }
1165
1166             ngx_memcpy(b, body->buf, sizeof(ngx_buf_t));
1167
1168             if (body->buf->in_file) {
1169                 b->file_pos = file_pos;
1170                 file_pos += 32 * 1024;
1171
1172                 if (file_pos >= body->buf->file_last) {
1173                     file_pos = body->buf->file_last;
1174                     next = 1;
1175                 }
1176
1177                 b->file_last = file_pos;
1178                 len = (ngx_uint_t) (file_pos - b->file_pos);
1179
1180             } else {
1181                 b->pos = pos;
1182                 b->start = pos;
1183                 pos += 32 * 1024;
1184
1185                 if (pos >= body->buf->last) {
1186                     pos = body->buf->last;
1187                     next = 1;
1188                 }
1189
1190                 b->last = pos;
1191                 len = (ngx_uint_t) (pos - b->pos);
1192             }
1193
1194             padding = 8 - len % 8;
1195             padding = (padding == 8) ? 0 : padding;
1196
1197             h->version = 1;
1198             h->type = NGX_HTTP_FASTCGI_STDIN;
1199             h->request_id_hi = 0;
1200             h->request_id_lo = 1;
1201             h->content_length_hi = (u_char) ((len >> 8) & 0xff);
1202             h->content_length_lo = (u_char) (len & 0xff);
1203             h->padding_length = (u_char) padding;
1204             h->reserved = 0;
1205
1206             cl->next = ngx_alloc_chain_link(r->pool);
1207             if (cl->next == NULL) {
1208                 return NGX_ERROR;
1209             }
1210
1211             cl = cl->next;
1212             cl->buf = b;
1213
1214             b = ngx_create_temp_buf(r->pool,
1215                                     sizeof(ngx_http_fastcgi_header_t)
1216                                     + padding);
1217             if (b == NULL) {

```

```

1218         return NGX\_ERROR;
1219     }
1220
1221     if (padding) {
1222         ngx\_memzero(b->last, padding);
1223         b->last += padding;
1224     }
1225
1226     h = (ngx\_http\_fastcgi\_header\_t *) b->last;
1227     b->last += sizeof(ngx\_http\_fastcgi\_header\_t);
1228
1229     cl->next = ngx\_alloc\_chain\_link(r->pool);
1230     if (cl->next == NULL) {
1231         return NGX\_ERROR;
1232     }
1233
1234     cl = cl->next;
1235     cl->buf = b;
1236
1237     } while (!next);
1238
1239     body = body->next;
1240 }
1241
1242 } else {
1243     r->upstream->request_bufs = cl;
1244 }
1245
1246 h->version = 1;
1247 h->type = NGX\_HTTP\_FASTCGI\_STDIN;
1248 h->request_id_hi = 0;
1249 h->request_id_lo = 1;
1250 h->content_length_hi = 0;
1251 h->content_length_lo = 0;
1252 h->padding_length = 0;
1253 h->reserved = 0;
1254
1255 cl->next = NULL;
1256
1257 return NGX\_OK;
1258 }
1259
1260
1261 static ngx\_int\_t
1262 ngx\_http\_fastcgi\_reinit\_request(ngx\_http\_request\_t *r)
1263 {
1264     ngx\_http\_fastcgi\_ctx\_t *f;
1265
1266     f = ngx\_http\_get\_module\_ctx(r, ngx\_http\_fastcgi\_module);
1267
1268     if (f == NULL) {
1269         return NGX\_OK;
1270     }
1271
1272     f->state = ngx\_http\_fastcgi\_st\_version;
1273     f->fastcgi_stdout = 0;
1274     f->large_stderr = 0;
1275
1276     if (f->split_parts) {
1277         f->split_parts->nelts = 0;
1278     }
1279
1280     r->state = 0;
1281
1282     return NGX\_OK;
1283 }
1284
1285
1286 static ngx\_int\_t
1287 ngx\_http\_fastcgi\_process\_header(ngx\_http\_request\_t *r)
1288 {
1289     u_char                *p, *msg, *start, *last,
1290                          *part_start, *part_end;
1291     size_t                size;
1292     ngx\_str\_t            *status_line, *pattern;
1293     ngx\_int\_t            rc, status;

```



```

1294 ngx\_buf\_t buf;
1295 ngx\_uint\_t i;
1296 ngx\_table\_elt\_t *h;
1297 ngx\_http\_upstream\_t *u;
1298 ngx\_http\_fastcgi\_ctx\_t *f;
1299 ngx\_http\_upstream\_header\_t *hh;
1300 ngx\_http\_fastcgi\_loc\_conf\_t *flcf;
1301 ngx\_http\_fastcgi\_split\_part\_t *part;
1302 ngx\_http\_upstream\_main\_conf\_t *umcf;
1303
1304 f = ngx\_http\_get\_module\_ctx(r, ngx\_http\_fastcgi\_module);
1305
1306 umcf = ngx\_http\_get\_module\_main\_conf(r, ngx\_http\_upstream\_module);
1307
1308 u = r->upstream;
1309
1310 for ( ;; ) {
1311
1312     if (f->state < ngx_http_fastcgi_st_data) {
1313
1314         f->pos = u->buffer.pos;
1315         f->last = u->buffer.last;
1316
1317         rc = ngx\_http\_fastcgi\_process\_record(r, f);
1318
1319         u->buffer.pos = f->pos;
1320         u->buffer.last = f->last;
1321
1322         if (rc == NGX\_AGAIN) {
1323             return NGX\_AGAIN;
1324         }
1325
1326         if (rc == NGX\_ERROR) {
1327             return NGX\_HTTP\_UPSTREAM\_INVALID\_HEADER;
1328         }
1329
1330         if (f->type != NGX\_HTTP\_FASTCGI\_STDOUT
1331             && f->type != NGX\_HTTP\_FASTCGI\_STDERR)
1332         {
1333             ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, 0,
1334                 "upstream sent unexpected FastCGI record: %d",
1335                 f->type);
1336
1337             return NGX\_HTTP\_UPSTREAM\_INVALID\_HEADER;
1338         }
1339
1340         if (f->type == NGX\_HTTP\_FASTCGI\_STDOUT && f->length == 0) {
1341             ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, 0,
1342                 "upstream prematurely closed FastCGI stdout");
1343
1344             return NGX\_HTTP\_UPSTREAM\_INVALID\_HEADER;
1345         }
1346     }
1347
1348     if (f->state == ngx_http_fastcgi_st_padding) {
1349
1350         if (u->buffer.pos + f->padding < u->buffer.last) {
1351             f->state = ngx_http_fastcgi_st_version;
1352             u->buffer.pos += f->padding;
1353
1354             continue;
1355         }
1356
1357         if (u->buffer.pos + f->padding == u->buffer.last) {
1358             f->state = ngx_http_fastcgi_st_version;
1359             u->buffer.pos = u->buffer.last;
1360
1361             return NGX\_AGAIN;
1362         }
1363
1364         f->padding -= u->buffer.last - u->buffer.pos;
1365         u->buffer.pos = u->buffer.last;
1366
1367         return NGX\_AGAIN;
1368     }
1369

```

```

1370
1371 /* f->state == ngx_http_fastcgi_st_data */
1372
1373 if (f->type == NGX\_HTTP\_FASTCGI\_STDERR) {
1374
1375     if (f->length) {
1376         msg = u->buffer.pos;
1377
1378         if (u->buffer.pos + f->length <= u->buffer.last) {
1379             u->buffer.pos += f->length;
1380             f->length = 0;
1381             f->state = ngx_http_fastcgi_st_padding;
1382
1383         } else {
1384             f->length -= u->buffer.last - u->buffer.pos;
1385             u->buffer.pos = u->buffer.last;
1386         }
1387
1388         for (p = u->buffer.pos - 1; msg < p; p--) {
1389             if (*p != LF && *p != CR && *p != '.' && *p != ' ') {
1390                 break;
1391             }
1392         }
1393
1394         p++;
1395
1396         ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, 0,
1397             "FastCGI sent in stderr: \"%s\"", p - msg, msg);
1398
1399         flcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_fastcgi\_module);
1400
1401         if (flcf->catch_stderr) {
1402             pattern = flcf->catch_stderr->elts;
1403
1404             for (i = 0; i < flcf->catch_stderr->nelts; i++) {
1405                 if (ngx\_strnstr(msg, (char *) pattern[i].data,
1406                     p - msg)
1407                     != NULL)
1408                 {
1409                     return NGX\_HTTP\_UPSTREAM\_INVALID\_HEADER;
1410                 }
1411             }
1412         }
1413
1414         if (u->buffer.pos == u->buffer.last) {
1415
1416             if (!f->fastcgi_stdout) {
1417
1418                 /*
1419                  * the special handling the large number
1420                  * of the PHP warnings to not allocate memory
1421                  */
1422
1423                 #if (NGX\_HTTP\_CACHE)
1424                 if (r->cache) {
1425                     u->buffer.pos = u->buffer.start
1426                         + r->cache->header_start;
1427                 } else {
1428                     u->buffer.pos = u->buffer.start;
1429                 }
1430                 #else
1431                 u->buffer.pos = u->buffer.start;
1432                 #endif
1433
1434                 u->buffer.last = u->buffer.pos;
1435                 f->large_stderr = 1;
1436             }
1437
1438             return NGX\_AGAIN;
1439         }
1440     } else {
1441         f->state = ngx_http_fastcgi_st_padding;
1442     }
1443
1444     continue;
1445 }

```

```

1446
1447
1448     /* f->type == NGX_HTTP_FASTCGI_STDOUT */
1449
1450 #if (NGX_HTTP_CACHE)
1451
1452     if (f->large_stderr && r->cache) {
1453         u_char                *start;
1454         ssize_t                len;
1455         ngx_http_fastcgi_header_t *fh;
1456
1457         start = u->buffer.start + r->cache->header_start;
1458
1459         len = u->buffer.pos - start - 2 * sizeof(ngx_http_fastcgi_header_t);
1460
1461         /*
1462          * A tail of large stderr output before HTTP header is placed
1463          * in a cache file without a FastCGI record header.
1464          * To workaround it we put a dummy FastCGI record header at the
1465          * start of the stderr output or update r->cache_header_start,
1466          * if there is no enough place for the record header.
1467          */
1468
1469         if (len >= 0) {
1470             fh = (ngx_http_fastcgi_header_t *) start;
1471             fh->version = 1;
1472             fh->type = NGX_HTTP_FASTCGI_STDERR;
1473             fh->request_id_hi = 0;
1474             fh->request_id_lo = 1;
1475             fh->content_length_hi = (u_char) ((len >> 8) & 0xff);
1476             fh->content_length_lo = (u_char) (len & 0xff);
1477             fh->padding_length = 0;
1478             fh->reserved = 0;
1479
1480         } else {
1481             r->cache->header_start += u->buffer.pos - start
1482                 - sizeof(ngx_http_fastcgi_header_t);
1483         }
1484
1485         f->large_stderr = 0;
1486     }
1487 #endif
1488
1489     f->fastcgi_stdout = 1;
1490
1491     start = u->buffer.pos;
1492
1493     if (u->buffer.pos + f->length < u->buffer.last) {
1494         /*
1495          * set u->buffer.last to the end of the FastCGI record data
1496          * for ngx_http_parse_header_line()
1497          */
1498
1499         last = u->buffer.last;
1500         u->buffer.last = u->buffer.pos + f->length;
1501
1502     } else {
1503         last = NULL;
1504     }
1505
1506     for ( ;; ) {
1507
1508         part_start = u->buffer.pos;
1509         part_end = u->buffer.last;
1510
1511         rc = ngx_http_parse_header_line(r, &u->buffer, 1);
1512
1513         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1514             "http fastcgi parser: %d", rc);
1515
1516         if (rc == NGX_AGAIN) {
1517             break;
1518         }
1519     }
1520 }
1521

```

```

1522 if (rc == NGX\_OK) {
1523
1524     /* a header line has been parsed successfully */
1525
1526     h = ngx\_list\_push(&u->headers_in.headers);
1527     if (h == NULL) {
1528         return NGX\_ERROR;
1529     }
1530
1531     if (f->split_parts && f->split_parts->nelts) {
1532
1533         part = f->split_parts->elts;
1534         size = u->buffer.pos - part_start;
1535
1536         for (i = 0; i < f->split_parts->nelts; i++) {
1537             size += part[i].end - part[i].start;
1538         }
1539
1540         p = ngx\_pnalloc(r->pool, size);
1541         if (p == NULL) {
1542             return NGX\_ERROR;
1543         }
1544
1545         buf.pos = p;
1546
1547         for (i = 0; i < f->split_parts->nelts; i++) {
1548             p = ngx\_cpymem(p, part[i].start,
1549                 part[i].end - part[i].start);
1550         }
1551
1552         p = ngx\_cpymem(p, part_start, u->buffer.pos - part_start);
1553
1554         buf.last = p;
1555
1556         f->split_parts->nelts = 0;
1557
1558         rc = ngx\_http\_parse\_header\_line(r, &buf, 1);
1559
1560         if (rc != NGX\_OK) {
1561             ngx\_log\_error(NGX\_LOG\_ALERT, r->connection->log, 0,
1562                 "invalid header after joining "
1563                 "FastCGI records");
1564             return NGX\_ERROR;
1565         }
1566
1567         h->key.len = r->header_name_end - r->header_name_start;
1568         h->key.data = r->header_name_start;
1569         h->key.data[h->key.len] = '\0';
1570
1571         h->value.len = r->header_end - r->header_start;
1572         h->value.data = r->header_start;
1573         h->value.data[h->value.len] = '\0';
1574
1575         h->lowercase_key = ngx\_pnalloc(r->pool, h->key.len);
1576         if (h->lowercase_key == NULL) {
1577             return NGX\_ERROR;
1578         }
1579     } else {
1580
1581         h->key.len = r->header_name_end - r->header_name_start;
1582         h->value.len = r->header_end - r->header_start;
1583
1584         h->key.data = ngx\_pnalloc(r->pool,
1585             h->key.len + 1 + h->value.len + 1
1586             + h->key.len);
1587         if (h->key.data == NULL) {
1588             return NGX\_ERROR;
1589         }
1590     }
1591
1592     h->value.data = h->key.data + h->key.len + 1;
1593     h->lowercase_key = h->key.data + h->key.len + 1
1594         + h->value.len + 1;
1595
1596     ngx\_memcpy(h->key.data, r->header_name_start, h->key.len);
1597     h->key.data[h->key.len] = '\0';

```

```

1598         ngx_memcpy(h->value.data, r->header_start, h->value.len);
1599         h->value.data[h->value.len] = '\0';
1600     }
1601
1602     h->hash = r->header_hash;
1603
1604     if (h->key.len == r->lowercase_index) {
1605         ngx_memcpy(h->lowercase_key, r->lowercase_header, h->key.len);
1606
1607     } else {
1608         ngx_strlow(h->lowercase_key, h->key.data, h->key.len);
1609     }
1610
1611     hh = ngx_hash_find(&umcf->headers_in_hash, h->hash,
1612                      h->lowercase_key, h->key.len);
1613
1614     if (hh && hh->handler(r, h, hh->offset) != NGX_OK) {
1615         return NGX_ERROR;
1616     }
1617
1618     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1619                  "http fastcgi header: \"%V: %V\"",
1620                  &h->key, &h->value);
1621
1622     if (u->buffer.pos < u->buffer.last) {
1623         continue;
1624     }
1625
1626     /* the end of the FastCGI record */
1627
1628     break;
1629 }
1630
1631 if (rc == NGX_HTTP_PARSE_HEADER_DONE) {
1632
1633     /* a whole header has been parsed successfully */
1634
1635     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1636                  "http fastcgi header done");
1637
1638     if (u->headers_in.status) {
1639         status_line = &u->headers_in.status->value;
1640
1641         status = ngx_atoi(status_line->data, 3);
1642
1643         if (status == NGX_ERROR) {
1644             ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1645                          "upstream sent invalid status \"%V\"",
1646                          status_line);
1647             return NGX_HTTP_UPSTREAM_INVALID_HEADER;
1648         }
1649
1650         u->headers_in.status_n = status;
1651         u->headers_in.status_line = *status_line;
1652
1653     } else if (u->headers_in.location) {
1654         u->headers_in.status_n = 302;
1655         ngx_str_set(&u->headers_in.status_line,
1656                  "302 Moved Temporarily");
1657
1658     } else {
1659         u->headers_in.status_n = 200;
1660         ngx_str_set(&u->headers_in.status_line, "200 OK");
1661     }
1662
1663     if (u->state && u->state->status == 0) {
1664         u->state->status = u->headers_in.status_n;
1665     }
1666
1667     break;
1668 }
1669
1670 /* there was error while a header line parsing */
1671
1672 ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1673              "upstream sent invalid header");

```

```

1674         return NGX\_HTTP\_UPSTREAM\_INVALID\_HEADER;
1675     }
1676
1677
1678     if (last) {
1679         u->buffer.last = last;
1680     }
1681
1682     f->length -= u->buffer.pos - start;
1683
1684     if (f->length == 0) {
1685         f->state = ngx_http_fastcgi_st_padding;
1686     }
1687
1688     if (rc == NGX\_HTTP\_PARSE\_HEADER\_DONE) {
1689         return NGX\_OK;
1690     }
1691
1692     if (rc == NGX\_OK) {
1693         continue;
1694     }
1695
1696     /* rc == NGX\_AGAIN */
1697
1698     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
1699                 "upstream split a header line in FastCGI records");
1700
1701     if (f->split_parts == NULL) {
1702         f->split_parts = ngx\_array\_create(r->pool, 1,
1703                                         sizeof(ngx\_http\_fastcgi\_split\_part\_t));
1704         if (f->split_parts == NULL) {
1705             return NGX\_ERROR;
1706         }
1707     }
1708
1709     part = ngx\_array\_push(f->split_parts);
1710     if (part == NULL) {
1711         return NGX\_ERROR;
1712     }
1713
1714     part->start = part_start;
1715     part->end = part_end;
1716
1717     if (u->buffer.pos < u->buffer.last) {
1718         continue;
1719     }
1720
1721     return NGX\_AGAIN;
1722 }
1723 }
1724
1725
1726 static ngx\_int\_t
1727 ngx\_http\_fastcgi\_input\_filter\_init(void *data)
1728 {
1729     ngx\_http\_request\_t *r = data;
1730     ngx\_http\_fastcgi\_loc\_conf\_t *flcf;
1731
1732     flcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_fastcgi\_module);
1733
1734     r->upstream->pipe->length = flcf->keep_conn ?
1735                             (off_t) sizeof(ngx\_http\_fastcgi\_header\_t) : -1;
1736
1737     return NGX\_OK;
1738 }
1739
1740
1741 static ngx\_int\_t
1742 ngx\_http\_fastcgi\_input\_filter(ngx\_event\_pipe\_t *p, ngx\_buf\_t *buf)
1743 {
1744     u_char *m, *msg;
1745     ngx\_int\_t rc;
1746     ngx\_buf\_t *b, **prev;
1747     ngx\_chain\_t *cl;
1748     ngx\_http\_request\_t *r;
1749     ngx\_http\_fastcgi\_ctx\_t *f;

```

```

1750 ngx_http_fastcgi_loc_conf_t *flcf;
1751
1752 if (buf->pos == buf->last) {
1753     return NGX_OK;
1754 }
1755
1756 r = p->input_ctx;
1757 f = ngx_http_get_module_ctx(r, ngx_http_fastcgi_module);
1758 flcf = ngx_http_get_module_loc_conf(r, ngx_http_fastcgi_module);
1759
1760 b = NULL;
1761 prev = &buf->shadow;
1762
1763 f->pos = buf->pos;
1764 f->last = buf->last;
1765
1766 for ( ;; ) {
1767     if (f->state < ngx_http_fastcgi_st_data) {
1768
1769         rc = ngx_http_fastcgi_process_record(r, f);
1770
1771         if (rc == NGX_AGAIN) {
1772             break;
1773         }
1774
1775         if (rc == NGX_ERROR) {
1776             return NGX_ERROR;
1777         }
1778
1779         if (f->type == NGX_HTTP_FASTCGI_STDOUT && f->length == 0) {
1780             f->state = ngx_http_fastcgi_st_padding;
1781
1782             if (!flcf->keep_conn) {
1783                 p->upstream_done = 1;
1784             }
1785
1786             ngx_log_debug0(NGX_LOG_DEBUG_HTTP, p->log, 0,
1787                 "http fastcgi closed stdout");
1788
1789             continue;
1790         }
1791
1792         if (f->type == NGX_HTTP_FASTCGI_END_REQUEST) {
1793
1794             ngx_log_debug0(NGX_LOG_DEBUG_HTTP, p->log, 0,
1795                 "http fastcgi sent end request");
1796
1797             if (!flcf->keep_conn) {
1798                 p->upstream_done = 1;
1799                 break;
1800             }
1801
1802             continue;
1803         }
1804     }
1805
1806     if (f->state == ngx_http_fastcgi_st_padding) {
1807
1808         if (f->type == NGX_HTTP_FASTCGI_END_REQUEST) {
1809
1810             if (f->pos + f->padding < f->last) {
1811                 p->upstream_done = 1;
1812                 break;
1813             }
1814
1815             if (f->pos + f->padding == f->last) {
1816                 p->upstream_done = 1;
1817                 r->upstream->keepalive = 1;
1818                 break;
1819             }
1820         }
1821
1822         f->padding -= f->last - f->pos;
1823
1824         break;
1825     }

```

```

1826     if (f->pos + f->padding < f->last) {
1827         f->state = ngx_http_fastcgi_st_version;
1828         f->pos += f->padding;
1829
1830
1831         continue;
1832     }
1833
1834     if (f->pos + f->padding == f->last) {
1835         f->state = ngx_http_fastcgi_st_version;
1836
1837         break;
1838     }
1839
1840     f->padding -= f->last - f->pos;
1841
1842     break;
1843 }
1844
1845
1846 /* f->state == ngx_http_fastcgi_st_data */
1847
1848 if (f->type == NGX\_HTTP\_FASTCGI\_STDERR) {
1849
1850     if (f->length) {
1851
1852         if (f->pos == f->last) {
1853             break;
1854         }
1855
1856         msg = f->pos;
1857
1858         if (f->pos + f->length <= f->last) {
1859             f->pos += f->length;
1860             f->length = 0;
1861             f->state = ngx_http_fastcgi_st_padding;
1862
1863         } else {
1864             f->length -= f->last - f->pos;
1865             f->pos = f->last;
1866         }
1867
1868         for (m = f->pos - 1; msg < m; m--) {
1869             if (*m != LF && *m != CR && *m != '.' && *m != ' ') {
1870                 break;
1871             }
1872         }
1873
1874         ngx\_log\_error(NGX\_LOG\_ERR, p->log, 0,
1875             "FastCGI sent in stderr: \"%*s\"",
1876             m + 1 - msg, msg);
1877
1878     } else {
1879         f->state = ngx_http_fastcgi_st_padding;
1880     }
1881
1882     continue;
1883 }
1884
1885 if (f->type == NGX\_HTTP\_FASTCGI\_END\_REQUEST) {
1886
1887     if (f->pos + f->length <= f->last) {
1888         f->state = ngx_http_fastcgi_st_padding;
1889         f->pos += f->length;
1890
1891         continue;
1892     }
1893
1894     f->length -= f->last - f->pos;
1895
1896     break;
1897 }
1898
1899
1900 /* f->type == NGX\_HTTP\_FASTCGI\_STDOUT */
1901

```



```

1902     if (f->pos == f->last) {
1903         break;
1904     }
1905
1906     cl = ngx_chain_get_free_buf(p->pool, &p->free);
1907     if (cl == NULL) {
1908         return NGX_ERROR;
1909     }
1910
1911     b = cl->buf;
1912
1913     ngx_memzero(b, sizeof(ngx_buf_t));
1914
1915     b->pos = f->pos;
1916     b->start = buf->start;
1917     b->end = buf->end;
1918     b->tag = p->tag;
1919     b->temporary = 1;
1920     b->recycled = 1;
1921
1922     *prev = b;
1923     prev = &b->shadow;
1924
1925     if (p->in) {
1926         *p->last_in = cl;
1927     } else {
1928         p->in = cl;
1929     }
1930     p->last_in = &cl->next;
1931
1932
1933     /* STUB */ b->num = buf->num;
1934
1935     ngx_log_debug2(NGX_LOG_DEBUG_EVENT, p->log, 0,
1936                  "input buf #%d %p", b->num, b->pos);
1937
1938     if (f->pos + f->length <= f->last) {
1939         f->state = ngx_http_fastcgi_st_padding;
1940         f->pos += f->length;
1941         b->last = f->pos;
1942
1943         continue;
1944     }
1945
1946     f->length -= f->last - f->pos;
1947
1948     b->last = f->last;
1949
1950     break;
1951 }
1952
1953 if (flcf->keep_conn) {
1954
1955     /* set p->length, minimal amount of data we want to see */
1956
1957     if (f->state < ngx_http_fastcgi_st_data) {
1958         p->length = 1;
1959     } else if (f->state == ngx_http_fastcgi_st_padding) {
1960         p->length = f->padding;
1961     } else {
1962         /* ngx_http_fastcgi_st_data */
1963
1964         p->length = f->length;
1965     }
1966 }
1967
1968 if (b) {
1969     b->shadow = buf;
1970     b->last_shadow = 1;
1971
1972     ngx_log_debug2(NGX_LOG_DEBUG_EVENT, p->log, 0,
1973                  "input buf %p %z", b->pos, b->last - b->pos);
1974
1975

```

```

1978     return NGX_OK;
1979 }
1980
1981 /* there is no data record in the buf, add it to free chain */
1982
1983 if (ngx_event_pipe_add_free_buf(p, buf) != NGX_OK) {
1984     return NGX_ERROR;
1985 }
1986
1987 return NGX_OK;
1988 }
1989
1990
1991 static ngx_int_t
1992 ngx_http_fastcgi_non_buffered_filter(void *data, ssize_t bytes)
1993 {
1994     u_char          *m, *msg;
1995     ngx_int_t      rc;
1996     ngx_buf_t     *b, *buf;
1997     ngx_chain_t   *cl, **ll;
1998     ngx_http_request_t *r;
1999     ngx_http_upstream_t *u;
2000     ngx_http_fastcgi_ctx_t *f;
2001
2002     r = data;
2003     f = ngx_http_get_module_ctx(r, ngx_http_fastcgi_module);
2004
2005     u = r->upstream;
2006     buf = &u->buffer;
2007
2008     buf->pos = buf->last;
2009     buf->last += bytes;
2010
2011     for (cl = u->out_bufs, ll = &u->out_bufs; cl; cl = cl->next) {
2012         ll = &cl->next;
2013     }
2014
2015     f->pos = buf->pos;
2016     f->last = buf->last;
2017
2018     for ( ;; ) {
2019         if (f->state < ngx_http_fastcgi_st_data) {
2020
2021             rc = ngx_http_fastcgi_process_record(r, f);
2022
2023             if (rc == NGX_AGAIN) {
2024                 break;
2025             }
2026
2027             if (rc == NGX_ERROR) {
2028                 return NGX_ERROR;
2029             }
2030
2031             if (f->type == NGX_HTTP_FASTCGI_STDOUT && f->length == 0) {
2032                 f->state = ngx_http_fastcgi_st_padding;
2033
2034                 ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
2035                     "http fastcgi closed stdout");
2036
2037                 continue;
2038             }
2039         }
2040
2041         if (f->state == ngx_http_fastcgi_st_padding) {
2042
2043             if (f->type == NGX_HTTP_FASTCGI_END_REQUEST) {
2044
2045                 if (f->pos + f->padding < f->last) {
2046                     u->length = 0;
2047                     break;
2048                 }
2049
2050                 if (f->pos + f->padding == f->last) {
2051                     u->length = 0;
2052                     u->keepalive = 1;
2053                     break;

```

```

2054     }
2055
2056     f->padding -= f->last - f->pos;
2057
2058     break;
2059 }
2060
2061 if (f->pos + f->padding < f->last) {
2062     f->state = ngx_http_fastcgi_st_version;
2063     f->pos += f->padding;
2064
2065     continue;
2066 }
2067
2068 if (f->pos + f->padding == f->last) {
2069     f->state = ngx_http_fastcgi_st_version;
2070
2071     break;
2072 }
2073
2074 f->padding -= f->last - f->pos;
2075
2076 break;
2077 }
2078
2079
2080 /* f->state == ngx_http_fastcgi_st_data */
2081
2082 if (f->type == NGX\_HTTP\_FASTCGI\_STDERR) {
2083
2084     if (f->length) {
2085
2086         if (f->pos == f->last) {
2087             break;
2088         }
2089
2090         msg = f->pos;
2091
2092         if (f->pos + f->length <= f->last) {
2093             f->pos += f->length;
2094             f->length = 0;
2095             f->state = ngx_http_fastcgi_st_padding;
2096
2097         } else {
2098             f->length -= f->last - f->pos;
2099             f->pos = f->last;
2100         }
2101
2102         for (m = f->pos - 1; msg < m; m--) {
2103             if (*m != LF && *m != CR && *m != '.' && *m != ' ') {
2104                 break;
2105             }
2106         }
2107
2108         ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, 0,
2109             "FastCGI sent in stderr: \"%*s\"",
2110             m + 1 - msg, msg);
2111
2112     } else {
2113         f->state = ngx_http_fastcgi_st_padding;
2114     }
2115
2116     continue;
2117 }
2118
2119 if (f->type == NGX\_HTTP\_FASTCGI\_END\_REQUEST) {
2120
2121     if (f->pos + f->length <= f->last) {
2122         f->state = ngx_http_fastcgi_st_padding;
2123         f->pos += f->length;
2124
2125         continue;
2126     }
2127
2128     f->length -= f->last - f->pos;
2129

```

```

2130         break;
2131     }
2132
2133     /* f->type == NGX\_HTTP\_FASTCGI\_STDOUT */
2134
2135     if (f->pos == f->last) {
2136         break;
2137     }
2138
2139     cl = ngx\_chain\_get\_free\_buf(r->pool, &u->free_bufs);
2140     if (cl == NULL) {
2141         return NGX\_ERROR;
2142     }
2143
2144     *ll = cl;
2145     ll = &cl->next;
2146
2147     b = cl->buf;
2148
2149     b->flush = 1;
2150     b->memory = 1;
2151
2152     b->pos = f->pos;
2153     b->tag = u->output.tag;
2154
2155     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
2156                 "http fastcgi output buf %p", b->pos);
2157
2158     if (f->pos + f->length <= f->last) {
2159         f->state = ngx_http_fastcgi_st_padding;
2160         f->pos += f->length;
2161         b->last = f->pos;
2162
2163         continue;
2164     }
2165
2166     f->length -= f->last - f->pos;
2167     b->last = f->last;
2168
2169     break;
2170 }
2171
2172 /* provide continuous buffer for subrequests in memory */
2173
2174 if (r->subrequest_in_memory) {
2175     cl = u->out_bufs;
2176
2177     if (cl) {
2178         buf->pos = cl->buf->pos;
2179     }
2180
2181     buf->last = buf->pos;
2182
2183     for (cl = u->out_bufs; cl; cl = cl->next) {
2184         ngx\_log\_debug3(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
2185                     "http fastcgi in memory %p-%p %uz",
2186                     cl->buf->pos, cl->buf->last, ngx\_buf\_size(cl->buf));
2187
2188         if (buf->last == cl->buf->pos) {
2189             buf->last = cl->buf->last;
2190             continue;
2191         }
2192
2193         buf->last = ngx\_movemem(buf->last, cl->buf->pos,
2194                             cl->buf->last - cl->buf->pos);
2195
2196         cl->buf->pos = buf->last - (cl->buf->last - cl->buf->pos);
2197         cl->buf->last = buf->last;
2198     }
2199 }
2200
2201 return NGX\_OK;
2202 }
2203
2204 }
2205

```

```

2206 static ngx_int_t
2207 ngx_http_fastcgi_process_record(ngx_http_request_t *r,
2208 ngx_http_fastcgi_ctx_t *f)
2209 {
2210     u_char          ch, *p;
2211     ngx_http_fastcgi_state_e  state;
2212
2213     state = f->state;
2214
2215     for (p = f->pos; p < f->last; p++) {
2216
2217         ch = *p;
2218
2219         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
2220             "http fastcgi record byte: %02Xd", ch);
2221
2222         switch (state) {
2223
2224         case ngx_http_fastcgi_st_version:
2225             if (ch != 1) {
2226                 ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
2227                     "upstream sent unsupported FastCGI "
2228                     "protocol version: %d", ch);
2229                 return NGX_ERROR;
2230             }
2231             state = ngx_http_fastcgi_st_type;
2232             break;
2233
2234         case ngx_http_fastcgi_st_type:
2235             switch (ch) {
2236             case NGX_HTTP_FASTCGI_STDOUT:
2237             case NGX_HTTP_FASTCGI_STDERR:
2238             case NGX_HTTP_FASTCGI_END_REQUEST:
2239                 f->type = (ngx_uint_t) ch;
2240                 break;
2241             default:
2242                 ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
2243                     "upstream sent invalid FastCGI "
2244                     "record type: %d", ch);
2245                 return NGX_ERROR;
2246             }
2247             state = ngx_http_fastcgi_st_request_id_hi;
2248             break;
2249
2250         /* we support the single request per connection */
2251
2252         case ngx_http_fastcgi_st_request_id_hi:
2253             if (ch != 0) {
2254                 ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
2255                     "upstream sent unexpected FastCGI "
2256                     "request id high byte: %d", ch);
2257                 return NGX_ERROR;
2258             }
2259             state = ngx_http_fastcgi_st_request_id_lo;
2260             break;
2261
2262         case ngx_http_fastcgi_st_request_id_lo:
2263             if (ch != 1) {
2264                 ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
2265                     "upstream sent unexpected FastCGI "
2266                     "request id low byte: %d", ch);
2267                 return NGX_ERROR;
2268             }
2269             state = ngx_http_fastcgi_st_content_length_hi;
2270             break;
2271
2272         case ngx_http_fastcgi_st_content_length_hi:
2273             f->length = ch << 8;
2274             state = ngx_http_fastcgi_st_content_length_lo;
2275             break;
2276
2277         case ngx_http_fastcgi_st_content_length_lo:
2278             f->length |= (size_t) ch;
2279             state = ngx_http_fastcgi_st_padding_length;
2280
2281

```

```

2282         break;
2283
2284     case ngx_http_fastcgi_st_padding_length:
2285         f->padding = (size_t) ch;
2286         state = ngx_http_fastcgi_st_reserved;
2287         break;
2288
2289     case ngx_http_fastcgi_st_reserved:
2290         state = ngx_http_fastcgi_st_data;
2291
2292         ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
2293                 "http fastcgi record length: %z", f->length);
2294
2295         f->pos = p + 1;
2296         f->state = state;
2297
2298         return NGX\_OK;
2299
2300         /* suppress warning */
2301     case ngx_http_fastcgi_st_data:
2302     case ngx_http_fastcgi_st_padding:
2303         break;
2304     }
2305 }
2306
2307 f->state = state;
2308
2309 return NGX\_AGAIN;
2310 }
2311
2312
2313 static void
2314 ngx\_http\_fastcgi\_abort\_request(ngx\_http\_request\_t *r)
2315 {
2316     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
2317                   "abort http fastcgi request");
2318
2319     return;
2320 }
2321
2322
2323 static void
2324 ngx\_http\_fastcgi\_finalize\_request(ngx\_http\_request\_t *r, ngx\_int\_t rc)
2325 {
2326     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
2327                   "finalize http fastcgi request");
2328
2329     return;
2330 }
2331
2332
2333 static ngx\_int\_t
2334 ngx\_http\_fastcgi\_add\_variables(ngx\_conf\_t *cf)
2335 {
2336     ngx\_http\_variable\_t *var, *v;
2337
2338     for (v = ngx\_http\_fastcgi\_vars; v->name.len; v++) {
2339         var = ngx\_http\_add\_variable(cf, &v->name, v->flags);
2340         if (var == NULL) {
2341             return NGX\_ERROR;
2342         }
2343
2344         var->get_handler = v->get_handler;
2345         var->data = v->data;
2346     }
2347
2348     return NGX\_OK;
2349 }
2350
2351
2352 static void *
2353 ngx\_http\_fastcgi\_create\_main\_conf(ngx\_conf\_t *cf)
2354 {
2355     ngx\_http\_fastcgi\_main\_conf\_t *conf;
2356
2357     conf = ngx\_palloc(cf->pool, sizeof(ngx\_http\_fastcgi\_main\_conf\_t));

```

```

2358     if (conf == NULL) {
2359         return NULL;
2360     }
2361
2362     #if (NGX_HTTP_CACHE)
2363     if (ngx_array_init(&conf->caches, cf->pool, 4,
2364         sizeof(ngx_http_file_cache_t *)))
2365         != NGX_OK)
2366     {
2367         return NULL;
2368     }
2369     #endif
2370
2371     return conf;
2372 }
2373
2374
2375 static void *
2376 ngx_http_fastcgi_create_loc_conf(ngx_conf_t *cf)
2377 {
2378     ngx_http_fastcgi_loc_conf_t *conf;
2379
2380     conf = ngx_palloc(cf->pool, sizeof(ngx_http_fastcgi_loc_conf_t));
2381     if (conf == NULL) {
2382         return NULL;
2383     }
2384
2385     /*
2386      * set by ngx_palloc():
2387      *
2388      *     conf->upstream.bufs.num = 0;
2389      *     conf->upstream.ignore_headers = 0;
2390      *     conf->upstream.next_upstream = 0;
2391      *     conf->upstream.cache_zone = NULL;
2392      *     conf->upstream.cache_use_stale = 0;
2393      *     conf->upstream.cache_methods = 0;
2394      *     conf->upstream.temp_path = NULL;
2395      *     conf->upstream.hide_headers_hash = { NULL, 0 };
2396      *     conf->upstream.uri = { 0, NULL };
2397      *     conf->upstream.location = NULL;
2398      *     conf->upstream.store_lengths = NULL;
2399      *     conf->upstream.store_values = NULL;
2400      *
2401      *     conf->index.len = { 0, NULL };
2402      */
2403
2404     conf->upstream.store = NGX_CONF_UNSET;
2405     conf->upstream.store_access = NGX_CONF_UNSET_UINT;
2406     conf->upstream.next_upstream_tries = NGX_CONF_UNSET_UINT;
2407     conf->upstream.buffering = NGX_CONF_UNSET;
2408     conf->upstream.ignore_client_abort = NGX_CONF_UNSET;
2409     conf->upstream.force_ranges = NGX_CONF_UNSET;
2410
2411     conf->upstream.local = NGX_CONF_UNSET_PTR;
2412
2413     conf->upstream.connect_timeout = NGX_CONF_UNSET_MSEC;
2414     conf->upstream.send_timeout = NGX_CONF_UNSET_MSEC;
2415     conf->upstream.read_timeout = NGX_CONF_UNSET_MSEC;
2416     conf->upstream.next_upstream_timeout = NGX_CONF_UNSET_MSEC;
2417
2418     conf->upstream.send_lowat = NGX_CONF_UNSET_SIZE;
2419     conf->upstream.buffer_size = NGX_CONF_UNSET_SIZE;
2420     conf->upstream.limit_rate = NGX_CONF_UNSET_SIZE;
2421
2422     conf->upstream.busy_buffers_size_conf = NGX_CONF_UNSET_SIZE;
2423     conf->upstream.max_temp_file_size_conf = NGX_CONF_UNSET_SIZE;
2424     conf->upstream.temp_file_write_size_conf = NGX_CONF_UNSET_SIZE;
2425
2426     conf->upstream.pass_request_headers = NGX_CONF_UNSET;
2427     conf->upstream.pass_request_body = NGX_CONF_UNSET;
2428
2429     #if (NGX_HTTP_CACHE)
2430     conf->upstream.cache = NGX_CONF_UNSET;
2431     conf->upstream.cache_min_uses = NGX_CONF_UNSET_UINT;
2432     conf->upstream.cache_bypass = NGX_CONF_UNSET_PTR;
2433     conf->upstream.no_cache = NGX_CONF_UNSET_PTR;

```

```

2434     conf->upstream.cache_valid = NGX\_CONF\_UNSET\_PTR;
2435     conf->upstream.cache_lock = NGX\_CONF\_UNSET;
2436     conf->upstream.cache_lock_timeout = NGX\_CONF\_UNSET\_MSEC;
2437     conf->upstream.cache_lock_age = NGX\_CONF\_UNSET\_MSEC;
2438     conf->upstream.cache_revalidate = NGX\_CONF\_UNSET;
2439 #endif
2440
2441     conf->upstream.hide_headers = NGX\_CONF\_UNSET\_PTR;
2442     conf->upstream.pass_headers = NGX\_CONF\_UNSET\_PTR;
2443
2444     conf->upstream.intercept_errors = NGX\_CONF\_UNSET;
2445
2446     /* "fastcgi_cyclic_temp_file" is disabled */
2447     conf->upstream.cyclic_temp_file = 0;
2448
2449     conf->upstream.change_buffering = 1;
2450
2451     conf->catch_stderr = NGX\_CONF\_UNSET\_PTR;
2452
2453     conf->keep_conn = NGX\_CONF\_UNSET;
2454
2455     ngx\_str\_set(&conf->upstream.module, "fastcgi");
2456
2457     return conf;
2458 }
2459
2460
2461 static char *
2462 ngx_http_fastcgi_merge_loc_conf(ngx\_conf\_t *cf, void *parent, void *child)
2463 {
2464     ngx\_http\_fastcgi\_loc\_conf\_t *prev = parent;
2465     ngx\_http\_fastcgi\_loc\_conf\_t *conf = child;
2466
2467     size_t                size;
2468     ngx\_int\_t             rc;
2469     ngx\_hash\_init\_t      hash;
2470     ngx\_http\_core\_loc\_conf\_t *clcf;
2471
2472 #if (NGX_HTTP_CACHE)
2473
2474     if (conf->upstream.store > 0) {
2475         conf->upstream.cache = 0;
2476     }
2477
2478     if (conf->upstream.cache > 0) {
2479         conf->upstream.store = 0;
2480     }
2481
2482 #endif
2483
2484     if (conf->upstream.store == NGX\_CONF\_UNSET) {
2485         ngx\_conf\_merge\_value(conf->upstream.store,
2486             prev->upstream.store, 0);
2487
2488         conf->upstream.store_lengths = prev->upstream.store_lengths;
2489         conf->upstream.store_values = prev->upstream.store_values;
2490     }
2491
2492     ngx\_conf\_merge\_uint\_value(conf->upstream.store_access,
2493         prev->upstream.store_access, 0600);
2494
2495     ngx\_conf\_merge\_uint\_value(conf->upstream.next_upstream_tries,
2496         prev->upstream.next_upstream_tries, 0);
2497
2498     ngx\_conf\_merge\_value(conf->upstream.buffering,
2499         prev->upstream.buffering, 1);
2500
2501     ngx\_conf\_merge\_value(conf->upstream.ignore_client_abort,
2502         prev->upstream.ignore_client_abort, 0);
2503
2504     ngx\_conf\_merge\_value(conf->upstream.force_ranges,
2505         prev->upstream.force_ranges, 0);
2506
2507     ngx\_conf\_merge\_ptr\_value(conf->upstream.local,
2508         prev->upstream.local, NULL);
2509

```



```

2510 ngx\_conf\_merge\_msec\_value(conf->upstream.connect_timeout,
2511     prev->upstream.connect_timeout, 60000);
2512
2513 ngx\_conf\_merge\_msec\_value(conf->upstream.send_timeout,
2514     prev->upstream.send_timeout, 60000);
2515
2516 ngx\_conf\_merge\_msec\_value(conf->upstream.read_timeout,
2517     prev->upstream.read_timeout, 60000);
2518
2519 ngx\_conf\_merge\_msec\_value(conf->upstream.next_upstream_timeout,
2520     prev->upstream.next_upstream_timeout, 0);
2521
2522 ngx\_conf\_merge\_size\_value(conf->upstream.send_lowat,
2523     prev->upstream.send_lowat, 0);
2524
2525 ngx\_conf\_merge\_size\_value(conf->upstream.buffer_size,
2526     prev->upstream.buffer_size,
2527     (size_t) ngx\_pagesize);
2528
2529 ngx\_conf\_merge\_size\_value(conf->upstream.limit_rate,
2530     prev->upstream.limit_rate, 0);
2531
2532
2533 ngx\_conf\_merge\_bufs\_value(conf->upstream.bufs, prev->upstream.bufs,
2534     8, ngx\_pagesize);
2535
2536 if (conf->upstream.bufs.num < 2) {
2537     ngx\_conf\_log\_error(NGX_LOG_EMERG, cf, 0,
2538         "there must be at least 2 \"fastcgi_buffers\");
2539     return NGX\_CONF\_ERROR;
2540 }
2541
2542
2543 size = conf->upstream.buffer_size;
2544 if (size < conf->upstream.bufs.size) {
2545     size = conf->upstream.bufs.size;
2546 }
2547
2548
2549 ngx\_conf\_merge\_size\_value(conf->upstream.busy_buffers_size_conf,
2550     prev->upstream.busy_buffers_size_conf,
2551     NGX\_CONF\_UNSET\_SIZE);
2552
2553 if (conf->upstream.busy_buffers_size_conf == NGX\_CONF\_UNSET\_SIZE) {
2554     conf->upstream.busy_buffers_size = 2 * size;
2555 } else {
2556     conf->upstream.busy_buffers_size =
2557         conf->upstream.busy_buffers_size_conf;
2558 }
2559
2560 if (conf->upstream.busy_buffers_size < size) {
2561     ngx\_conf\_log\_error(NGX_LOG_EMERG, cf, 0,
2562         "\"fastcgi_busy_buffers_size\" must be equal to or greater than \"
2563         \"the maximum of the value of \"fastcgi_buffer_size\" and \"
2564         \"one of the \"fastcgi_buffers\");
2565
2566     return NGX\_CONF\_ERROR;
2567 }
2568
2569 if (conf->upstream.busy_buffers_size
2570     > (conf->upstream.bufs.num - 1) * conf->upstream.bufs.size)
2571 {
2572     ngx\_conf\_log\_error(NGX_LOG_EMERG, cf, 0,
2573         "\"fastcgi_busy_buffers_size\" must be less than \"
2574         \"the size of all \"fastcgi_buffers\" minus one buffer");
2575
2576     return NGX\_CONF\_ERROR;
2577 }
2578
2579
2580 ngx\_conf\_merge\_size\_value(conf->upstream.temp_file_write_size_conf,
2581     prev->upstream.temp_file_write_size_conf,
2582     NGX\_CONF\_UNSET\_SIZE);
2583
2584 if (conf->upstream.temp_file_write_size_conf == NGX\_CONF\_UNSET\_SIZE) {
2585     conf->upstream.temp_file_write_size = 2 * size;

```

```

2586 } else {
2587     conf->upstream.temp_file_write_size =
2588         conf->upstream.temp_file_write_size_conf;
2589 }
2590
2591 if (conf->upstream.temp_file_write_size < size) {
2592     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
2593         "\"fastcgi_temp_file_write_size\" must be equal to or greater "
2594         "than the maximum of the value of \"fastcgi_buffer_size\" and "
2595         "one of the \"fastcgi_buffers\"");
2596
2597     return NGX_CONF_ERROR;
2598 }
2599
2600 ngx_conf_merge_size_value(conf->upstream.max_temp_file_size_conf,
2601     prev->upstream.max_temp_file_size_conf,
2602     NGX_CONF_UNSET_SIZE);
2603
2604 if (conf->upstream.max_temp_file_size_conf == NGX_CONF_UNSET_SIZE) {
2605     conf->upstream.max_temp_file_size = 1024 * 1024 * 1024;
2606 } else {
2607     conf->upstream.max_temp_file_size =
2608         conf->upstream.max_temp_file_size_conf;
2609 }
2610
2611 if (conf->upstream.max_temp_file_size != 0
2612     && conf->upstream.max_temp_file_size < size)
2613 {
2614     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
2615         "\"fastcgi_max_temp_file_size\" must be equal to zero to disable "
2616         "temporary files usage or must be equal to or greater than "
2617         "the maximum of the value of \"fastcgi_buffer_size\" and "
2618         "one of the \"fastcgi_buffers\"");
2619
2620     return NGX_CONF_ERROR;
2621 }
2622
2623 ngx_conf_merge_bitmask_value(conf->upstream.ignore_headers,
2624     prev->upstream.ignore_headers,
2625     NGX_CONF_BITMASK_SET);
2626
2627 ngx_conf_merge_bitmask_value(conf->upstream.next_upstream,
2628     prev->upstream.next_upstream,
2629     (NGX_CONF_BITMASK_SET
2630      |NGX_HTTP_UPSTREAM_FT_ERROR
2631      |NGX_HTTP_UPSTREAM_FT_TIMEOUT));
2632
2633 if (conf->upstream.next_upstream & NGX_HTTP_UPSTREAM_FT_OFF) {
2634     conf->upstream.next_upstream = NGX_CONF_BITMASK_SET
2635         |NGX_HTTP_UPSTREAM_FT_OFF;
2636 }
2637
2638 if (ngx_conf_merge_path_value(cf, &conf->upstream.temp_path,
2639     prev->upstream.temp_path,
2640     &ngx_http_fastcgi_temp_path)
2641     != NGX_OK)
2642 {
2643     return NGX_CONF_ERROR;
2644 }
2645
2646 #if (NGX_HTTP_CACHE)
2647
2648 if (conf->upstream.cache == NGX_CONF_UNSET) {
2649     ngx_conf_merge_value(conf->upstream.cache,
2650         prev->upstream.cache, 0);
2651
2652     conf->upstream.cache_zone = prev->upstream.cache_zone;
2653     conf->upstream.cache_value = prev->upstream.cache_value;
2654 }
2655
2656 if (conf->upstream.cache_zone && conf->upstream.cache_zone->data == NULL) {
2657     ngx_shm_zone_t *shm_zone;
2658 }
2659
2660
2661

```

```

2662     shm_zone = conf->upstream.cache_zone;
2663
2664     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
2665         "\"fastcgi_cache\" zone \"%V\" is unknown",
2666         &shm_zone->shm.name);
2667
2668     return NGX_CONF_ERROR;
2669 }
2670
2671 ngx_conf_merge_uint_value(conf->upstream.cache_min_uses,
2672     prev->upstream.cache_min_uses, 1);
2673
2674 ngx_conf_merge_bitmask_value(conf->upstream.cache_use_stale,
2675     prev->upstream.cache_use_stale,
2676     (NGX_CONF_BITMASK_SET
2677     |NGX_HTTP_UPSTREAM_FT_OFF));
2678
2679 if (conf->upstream.cache_use_stale & NGX_HTTP_UPSTREAM_FT_OFF) {
2680     conf->upstream.cache_use_stale = NGX_CONF_BITMASK_SET
2681         |NGX_HTTP_UPSTREAM_FT_OFF;
2682 }
2683
2684 if (conf->upstream.cache_use_stale & NGX_HTTP_UPSTREAM_FT_ERROR) {
2685     conf->upstream.cache_use_stale |= NGX_HTTP_UPSTREAM_FT_NOLIVE;
2686 }
2687
2688 if (conf->upstream.cache_methods == 0) {
2689     conf->upstream.cache_methods = prev->upstream.cache_methods;
2690 }
2691
2692 conf->upstream.cache_methods |= NGX_HTTP_GET|NGX_HTTP_HEAD;
2693
2694 ngx_conf_merge_ptr_value(conf->upstream.cache_bypass,
2695     prev->upstream.cache_bypass, NULL);
2696
2697 ngx_conf_merge_ptr_value(conf->upstream.no_cache,
2698     prev->upstream.no_cache, NULL);
2699
2700 ngx_conf_merge_ptr_value(conf->upstream.cache_valid,
2701     prev->upstream.cache_valid, NULL);
2702
2703 if (conf->cache_key.value.data == NULL) {
2704     conf->cache_key = prev->cache_key;
2705 }
2706
2707 if (conf->upstream.cache && conf->cache_key.value.data == NULL) {
2708     ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
2709         "no \"fastcgi_cache_key\" for \"fastcgi_cache\"");
2710 }
2711
2712 ngx_conf_merge_value(conf->upstream.cache_lock,
2713     prev->upstream.cache_lock, 0);
2714
2715 ngx_conf_merge_msec_value(conf->upstream.cache_lock_timeout,
2716     prev->upstream.cache_lock_timeout, 5000);
2717
2718 ngx_conf_merge_msec_value(conf->upstream.cache_lock_age,
2719     prev->upstream.cache_lock_age, 5000);
2720
2721 ngx_conf_merge_value(conf->upstream.cache_revalidate,
2722     prev->upstream.cache_revalidate, 0);
2723
2724 #endif
2725
2726 ngx_conf_merge_value(conf->upstream.pass_request_headers,
2727     prev->upstream.pass_request_headers, 1);
2728 ngx_conf_merge_value(conf->upstream.pass_request_body,
2729     prev->upstream.pass_request_body, 1);
2730
2731 ngx_conf_merge_value(conf->upstream.intercept_errors,
2732     prev->upstream.intercept_errors, 0);
2733
2734 ngx_conf_merge_ptr_value(conf->catch_stderr, prev->catch_stderr, NULL);
2735
2736 ngx_conf_merge_value(conf->keep_conn, prev->keep_conn, 0);
2737

```

```

2738     ngx_conf_merge_str_value(conf->index, prev->index, "");
2739
2740
2741     hash.max_size = 512;
2742     hash.bucket_size = ngx_align(64, ngx_cacheline_size);
2743     hash.name = "fastcgi_hide_headers_hash";
2744
2745     if (ngx_http_upstream_hide_headers_hash(cf, &conf->upstream,
2746         &prev->upstream, ngx_http_fastcgi_hide_headers, &hash)
2747         != NGX_OK)
2748     {
2749         return NGX_CONF_ERROR;
2750     }
2751
2752     clcf = ngx_http_conf_get_module_loc_conf(cf, ngx_http_core_module);
2753
2754     if (clcf->noname
2755         && conf->upstream.upstream == NULL && conf->fastcgi_lengths == NULL)
2756     {
2757         conf->upstream.upstream = prev->upstream.upstream;
2758         conf->fastcgi_lengths = prev->fastcgi_lengths;
2759         conf->fastcgi_values = prev->fastcgi_values;
2760     }
2761
2762     if (clcf->lmt_excpt && clcf->handler == NULL
2763         && (conf->upstream.upstream || conf->fastcgi_lengths))
2764     {
2765         clcf->handler = ngx_http_fastcgi_handler;
2766     }
2767
2768 #if (NGX_PCRE)
2769     if (conf->split_regex == NULL) {
2770         conf->split_regex = prev->split_regex;
2771         conf->split_name = prev->split_name;
2772     }
2773 #endif
2774
2775     if (conf->params_source == NULL) {
2776         conf->params = prev->params;
2777 #if (NGX_HTTP_CACHE)
2778         conf->params_cache = prev->params_cache;
2779 #endif
2780         conf->params_source = prev->params_source;
2781     }
2782
2783     rc = ngx_http_fastcgi_init_params(cf, conf, &conf->params, NULL);
2784     if (rc != NGX_OK) {
2785         return NGX_CONF_ERROR;
2786     }
2787
2788 #if (NGX_HTTP_CACHE)
2789
2790     if (conf->upstream.cache) {
2791         rc = ngx_http_fastcgi_init_params(cf, conf, &conf->params_cache,
2792             ngx_http_fastcgi_cache_headers);
2793         if (rc != NGX_OK) {
2794             return NGX_CONF_ERROR;
2795         }
2796     }
2797 #endif
2798 #endif
2799
2800     return NGX_CONF_OK;
2801 }
2802
2803
2804 static ngx_int_t
2805 ngx_http_fastcgi_init_params(ngx_conf_t *cf, ngx_http_fastcgi_loc_conf_t *conf,
2806     ngx_http_fastcgi_params_t *params, ngx_keyval_t *default_params)
2807 {
2808     u_char                *p;
2809     size_t                size;
2810     uintptr_t             *code;
2811     ngx_uint_t            i, nsrc;
2812     ngx_array_t            headers_names, params_merged;
2813     ngx_keyval_t          *h;

```

```

2814 ngx\_hash\_key\_t *hk;
2815 ngx\_hash\_init\_t hash;
2816 ngx\_http\_upstream\_param\_t *src, *s;
2817 ngx\_http\_script\_compile\_t sc;
2818 ngx\_http\_script\_copy\_code\_t *copy;
2819
2820 if (params->hash.buckets) {
2821     return NGX\_OK;
2822 }
2823
2824 if (conf->params_source == NULL && default_params == NULL) {
2825     params->hash.buckets = (void *) 1;
2826     return NGX\_OK;
2827 }
2828
2829 params->lengths = ngx\_array\_create(cf->pool, 64, 1);
2830 if (params->lengths == NULL) {
2831     return NGX\_ERROR;
2832 }
2833
2834 params->values = ngx\_array\_create(cf->pool, 512, 1);
2835 if (params->values == NULL) {
2836     return NGX\_ERROR;
2837 }
2838
2839 if (ngx\_array\_init(&headers_names, cf->temp_pool, 4, sizeof(ngx\_hash\_key\_t))
2840     != NGX\_OK)
2841 {
2842     return NGX\_ERROR;
2843 }
2844
2845 if (conf->params_source) {
2846     src = conf->params_source->elts;
2847     nsrc = conf->params_source->nelts;
2848
2849 } else {
2850     src = NULL;
2851     nsrc = 0;
2852 }
2853
2854 if (default_params) {
2855     if (ngx\_array\_init(&params_merged, cf->temp_pool, 4,
2856         sizeof(ngx\_http\_upstream\_param\_t))
2857         != NGX\_OK)
2858     {
2859         return NGX\_ERROR;
2860     }
2861
2862     for (i = 0; i < nsrc; i++) {
2863
2864         s = ngx\_array\_push(&params_merged);
2865         if (s == NULL) {
2866             return NGX\_ERROR;
2867         }
2868
2869         *s = src[i];
2870     }
2871
2872     h = default_params;
2873
2874     while (h->key.len) {
2875
2876         src = params_merged.elts;
2877         nsrc = params_merged.nelts;
2878
2879         for (i = 0; i < nsrc; i++) {
2880             if (ngx\_strcasecmp(h->key.data, src[i].key.data) == 0) {
2881                 goto next;
2882             }
2883         }
2884
2885         s = ngx\_array\_push(&params_merged);
2886         if (s == NULL) {
2887             return NGX\_ERROR;
2888         }
2889

```

```

2890     s->key = h->key;
2891     s->value = h->value;
2892     s->skip_empty = 1;
2893
2894     next:
2895
2896         h++;
2897     }
2898
2899     src = params_merged.elts;
2900     nsrc = params_merged.nelts;
2901 }
2902
2903 for (i = 0; i < nsrc; i++) {
2904
2905     if (src[i].key.len > sizeof("HTTP_") - 1
2906         && ngx_strncmp(src[i].key.data, "HTTP_", sizeof("HTTP_") - 1) == 0)
2907     {
2908         hk = ngx_array_push(&headers_names);
2909         if (hk == NULL) {
2910             return NGX_ERROR;
2911         }
2912
2913         hk->key.len = src[i].key.len - 5;
2914         hk->key.data = src[i].key.data + 5;
2915         hk->key_hash = ngx_hash_key_lc(hk->key.data, hk->key.len);
2916         hk->value = (void *) 1;
2917
2918         if (src[i].value.len == 0) {
2919             continue;
2920         }
2921     }
2922
2923     copy = ngx_array_push_n(params->lengths,
2924                             sizeof(ngx_http_script_copy_code_t));
2925     if (copy == NULL) {
2926         return NGX_ERROR;
2927     }
2928
2929     copy->code = (ngx_http_script_code_pt) ngx_http_script_copy_len_code;
2930     copy->len = src[i].key.len;
2931
2932     copy = ngx_array_push_n(params->lengths,
2933                             sizeof(ngx_http_script_copy_code_t));
2934     if (copy == NULL) {
2935         return NGX_ERROR;
2936     }
2937
2938     copy->code = (ngx_http_script_code_pt) ngx_http_script_copy_len_code;
2939     copy->len = src[i].skip_empty;
2940
2941
2942     size = (sizeof(ngx_http_script_copy_code_t)
2943             + src[i].key.len + sizeof(uintptr_t) - 1)
2944           & ~(sizeof(uintptr_t) - 1);
2945
2946     copy = ngx_array_push_n(params->values, size);
2947     if (copy == NULL) {
2948         return NGX_ERROR;
2949     }
2950
2951     copy->code = ngx_http_script_copy_code;
2952     copy->len = src[i].key.len;
2953
2954     p = (u_char *) copy + sizeof(ngx_http_script_copy_code_t);
2955     ngx_memcpy(p, src[i].key.data, src[i].key.len);
2956
2957     ngx_memzero(&sc, sizeof(ngx_http_script_compile_t));
2958
2959     sc.cf = cf;
2960     sc.source = &src[i].value;
2961     sc.flushes = &params->flushes;
2962     sc.lengths = &params->lengths;
2963     sc.values = &params->values;
2964
2965

```

```

2966     if (ngx_http_script_compile(&sc) != NGX_OK) {
2967         return NGX_ERROR;
2968     }
2969
2970     code = ngx_array_push_n(params->lengths, sizeof(uintptr_t));
2971     if (code == NULL) {
2972         return NGX_ERROR;
2973     }
2974
2975     *code = (uintptr_t) NULL;
2976
2977
2978     code = ngx_array_push_n(params->values, sizeof(uintptr_t));
2979     if (code == NULL) {
2980         return NGX_ERROR;
2981     }
2982
2983     *code = (uintptr_t) NULL;
2984 }
2985
2986 code = ngx_array_push_n(params->lengths, sizeof(uintptr_t));
2987 if (code == NULL) {
2988     return NGX_ERROR;
2989 }
2990
2991 *code = (uintptr_t) NULL;
2992
2993 params->number = headers_names.nelts;
2994
2995 hash.hash = &params->hash;
2996 hash.key = ngx_hash_key_lc;
2997 hash.max_size = 512;
2998 hash.bucket_size = 64;
2999 hash.name = "fastcgi_params_hash";
3000 hash.pool = cf->pool;
3001 hash.temp_pool = NULL;
3002
3003 return ngx_hash_init(&hash, headers_names.elts, headers_names.nelts);
3004 }
3005
3006
3007 static ngx_int_t
3008 ngx_http_fastcgi_script_name_variable(ngx_http_request_t *r,
3009     ngx_http_variable_value_t *v, uintptr_t data)
3010 {
3011     u_char *p;
3012     ngx_http_fastcgi_ctx_t *f;
3013     ngx_http_fastcgi_loc_conf_t *flcf;
3014
3015     flcf = ngx_http_get_module_loc_conf(r, ngx_http_fastcgi_module);
3016
3017     f = ngx_http_fastcgi_split(r, flcf);
3018
3019     if (f == NULL) {
3020         return NGX_ERROR;
3021     }
3022
3023     if (f->script_name.len == 0
3024         || f->script_name.data[f->script_name.len - 1] != '/')
3025     {
3026         v->len = f->script_name.len;
3027         v->valid = 1;
3028         v->no_cacheable = 0;
3029         v->not_found = 0;
3030         v->data = f->script_name.data;
3031
3032         return NGX_OK;
3033     }
3034
3035     v->len = f->script_name.len + flcf->index.len;
3036
3037     v->data = ngx_pnalloc(r->pool, v->len);
3038     if (v->data == NULL) {
3039         return NGX_ERROR;
3040     }
3041

```

```

3042     p = ngx_copy(v->data, f->script_name.data, f->script_name.len);
3043     ngx_memcpy(p, flcf->index.data, flcf->index.len);
3044
3045     return NGX_OK;
3046 }
3047
3048
3049 static ngx_int_t
3050 ngx_http_fastcgi_path_info_variable(ngx_http_request_t *r,
3051     ngx_http_variable_value_t *v, uintptr_t data)
3052 {
3053     ngx_http_fastcgi_ctx_t *f;
3054     ngx_http_fastcgi_loc_conf_t *flcf;
3055
3056     flcf = ngx_http_get_module_loc_conf(r, ngx_http_fastcgi_module);
3057
3058     f = ngx_http_fastcgi_split(r, flcf);
3059
3060     if (f == NULL) {
3061         return NGX_ERROR;
3062     }
3063
3064     v->len = f->path_info.len;
3065     v->valid = 1;
3066     v->no_cacheable = 0;
3067     v->not_found = 0;
3068     v->data = f->path_info.data;
3069
3070     return NGX_OK;
3071 }
3072
3073
3074 static ngx_http_fastcgi_ctx_t *
3075 ngx_http_fastcgi_split(ngx_http_request_t *r, ngx_http_fastcgi_loc_conf_t *flcf)
3076 {
3077     ngx_http_fastcgi_ctx_t *f;
3078     #if (NGX_PCRE)
3079     ngx_int_t n;
3080     int captures[(1 + 2) * 3];
3081
3082     f = ngx_http_get_module_ctx(r, ngx_http_fastcgi_module);
3083
3084     if (f == NULL) {
3085         f = ngx_palloc(r->pool, sizeof(ngx_http_fastcgi_ctx_t));
3086         if (f == NULL) {
3087             return NULL;
3088         }
3089
3090         ngx_http_set_ctx(r, f, ngx_http_fastcgi_module);
3091     }
3092
3093     if (f->script_name.len) {
3094         return f;
3095     }
3096
3097     if (flcf->split_regex == NULL) {
3098         f->script_name = r->uri;
3099         return f;
3100     }
3101
3102     n = ngx_regex_exec(flcf->split_regex, &r->uri, captures, (1 + 2) * 3);
3103
3104     if (n >= 0) { /* match */
3105         f->script_name.len = captures[3] - captures[2];
3106         f->script_name.data = r->uri.data + captures[2];
3107
3108         f->path_info.len = captures[5] - captures[4];
3109         f->path_info.data = r->uri.data + captures[4];
3110
3111         return f;
3112     }
3113
3114     if (n == NGX_REGEX_NO_MATCHED) {
3115         f->script_name = r->uri;
3116         return f;
3117     }

```



```

3118     ngx_log_error(NGX_LOG_ALERT, r->connection->log, 0,
3119         ngx_regex_exec_n " failed: %i on \"%V\" using \"%V\"",
3120         n, &r->uri, &flcf->split_name);
3121     return NULL;
3122
3123 #else
3124
3125     f = ngx_http_get_module_ctx(r, ngx_http_fastcgi_module);
3126
3127     if (f == NULL) {
3128         f = ngx_palloc(r->pool, sizeof(ngx_http_fastcgi_ctx_t));
3129         if (f == NULL) {
3130             return NULL;
3131         }
3132     }
3133     ngx_http_set_ctx(r, f, ngx_http_fastcgi_module);
3134 }
3135
3136 f->script_name = r->uri;
3137
3138 return f;
3139
3140 #endif
3141 }
3142
3143 static char *
3144 ngx_http_fastcgi_pass(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
3145 {
3146     ngx_http_fastcgi_loc_conf_t *flcf = conf;
3147
3148     ngx_url_t          u;
3149     ngx_str_t          *value, *url;
3150     ngx_uint_t          n;
3151     ngx_http_core_loc_conf_t *clcf;
3152     ngx_http_script_compile_t sc;
3153
3154     if (flcf->upstream.upstream || flcf->fastcgi_lengths) {
3155         return "is duplicate";
3156     }
3157
3158     clcf = ngx_http_conf_get_module_loc_conf(cf, ngx_http_core_module);
3159
3160     clcf->handler = ngx_http_fastcgi_handler;
3161
3162     if (clcf->name.data[clcf->name.len - 1] == '/') {
3163         clcf->auto_redirect = 1;
3164     }
3165
3166     value = cf->args->elts;
3167
3168     url = &value[1];
3169
3170     n = ngx_http_script_variables_count(url);
3171
3172     if (n) {
3173         ngx_memzero(&sc, sizeof(ngx_http_script_compile_t));
3174
3175         sc.cf = cf;
3176         sc.source = url;
3177         sc.lengths = &flcf->fastcgi_lengths;
3178         sc.values = &flcf->fastcgi_values;
3179         sc.variables = n;
3180         sc.complete_lengths = 1;
3181         sc.complete_values = 1;
3182
3183         if (ngx_http_script_compile(&sc) != NGX_OK) {
3184             return NGX_CONF_ERROR;
3185         }
3186
3187         return NGX_CONF_OK;
3188     }
3189
3190     return NGX_CONF_OK;
3191 }
3192
3193 ngx_memzero(&u, sizeof(ngx_url_t));

```

```

3194     u.url = value[1];
3195     u.no_resolve = 1;
3196
3197
3198     flcf->upstream.upstream = ngx_http_upstream_add(cf, &u, 0);
3199     if (flcf->upstream.upstream == NULL) {
3200         return NGX_CONF_ERROR;
3201     }
3202
3203     return NGX_CONF_OK;
3204 }
3205
3206
3207 static char *
3208 ngx_http_fastcgi_split_path_info(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
3209 {
3210     #if (NGX_PCRE)
3211         ngx_http_fastcgi_loc_conf_t *flcf = conf;
3212
3213         ngx_str_t          *value;
3214         ngx_regex_compile_t rc;
3215         u_char              errstr[NGX_MAX_CONF_ERRSTR];
3216
3217         value = cf->args->elts;
3218
3219         flcf->split_name = value[1];
3220
3221         ngx_memzero(&rc, sizeof(ngx_regex_compile_t));
3222
3223         rc.pattern = value[1];
3224         rc.pool = cf->pool;
3225         rc.err.len = NGX_MAX_CONF_ERRSTR;
3226         rc.err.data = errstr;
3227
3228         if (ngx_regex_compile(&rc) != NGX_OK) {
3229             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0, "%V", &rc.err);
3230             return NGX_CONF_ERROR;
3231         }
3232
3233         if (rc.captures != 2) {
3234             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
3235                 "pattern \"%V\" must have 2 captures", &value[1]);
3236             return NGX_CONF_ERROR;
3237         }
3238
3239         flcf->split_regex = rc.regex;
3240
3241         return NGX_CONF_OK;
3242
3243     #else
3244
3245         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
3246             "\"%V\" requires PCRE library", &cmd->name);
3247         return NGX_CONF_ERROR;
3248
3249     #endif
3250 }
3251
3252
3253 static char *
3254 ngx_http_fastcgi_store(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
3255 {
3256     ngx_http_fastcgi_loc_conf_t *flcf = conf;
3257
3258     ngx_str_t          *value;
3259     ngx_http_script_compile_t sc;
3260
3261     if (flcf->upstream.store != NGX_CONF_UNSET) {
3262         return "is duplicate";
3263     }
3264
3265     value = cf->args->elts;
3266
3267     if (ngx_strcmp(value[1].data, "off") == 0) {
3268         flcf->upstream.store = 0;
3269         return NGX_CONF_OK;

```

```

3270     }
3271
3272 #if (NGX_HTTP_CACHE)
3273     if (flcf->upstream.cache > 0) {
3274         return "is incompatible with \"fastcgi_cache\"";
3275     }
3276 #endif
3277
3278     flcf->upstream.store = 1;
3279
3280     if (ngx_strcmp(value[1].data, "on") == 0) {
3281         return NGX_CONF_OK;
3282     }
3283
3284     /* include the terminating '\0' into script */
3285     value[1].len++;
3286
3287     ngx_memzero(&sc, sizeof(ngx_http_script_compile_t));
3288
3289     sc.cf = cf;
3290     sc.source = &value[1];
3291     sc.lengths = &flcf->upstream.store_lengths;
3292     sc.values = &flcf->upstream.store_values;
3293     sc.variables = ngx_http_script_variables_count(&value[1]);
3294     sc.complete_lengths = 1;
3295     sc.complete_values = 1;
3296
3297     if (ngx_http_script_compile(&sc) != NGX_OK) {
3298         return NGX_CONF_ERROR;
3299     }
3300
3301     return NGX_CONF_OK;
3302 }
3303
3304 #if (NGX_HTTP_CACHE)
3305
3306 static char *
3307 ngx_http_fastcgi_cache(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
3308 {
3309     ngx_http_fastcgi_loc_conf_t *flcf = conf;
3310
3311     ngx_str_t          *value;
3312     ngx_http_complex_value_t  cv;
3313     ngx_http_compile_complex_value_t  ccv;
3314
3315     value = cf->args->elts;
3316
3317     if (flcf->upstream.cache != NGX_CONF_UNSET) {
3318         return "is duplicate";
3319     }
3320
3321     if (ngx_strcmp(value[1].data, "off") == 0) {
3322         flcf->upstream.cache = 0;
3323         return NGX_CONF_OK;
3324     }
3325
3326     if (flcf->upstream.store > 0) {
3327         return "is incompatible with \"fastcgi_store\"";
3328     }
3329
3330     flcf->upstream.cache = 1;
3331
3332     ngx_memzero(&ccv, sizeof(ngx_http_compile_complex_value_t));
3333
3334     ccv.cf = cf;
3335     ccv.value = &value[1];
3336     ccv.complex_value = &cv;
3337
3338     if (ngx_http_compile_complex_value(&ccv) != NGX_OK) {
3339         return NGX_CONF_ERROR;
3340     }
3341
3342     if (cv.lengths != NULL) {
3343         flcf->upstream.cache_value = ngx_palloc(cf->pool,

```

```

3346         sizeof(ngx\_http\_complex\_value\_t));
3347     if (flcf->upstream.cache_value == NULL) {
3348         return NGX\_CONF\_ERROR;
3349     }
3350
3351     *flcf->upstream.cache_value = cv;
3352
3353     return NGX\_CONF\_OK;
3354 }
3355
3356 flcf->upstream.cache_zone = ngx\_shared\_memory\_add(cf, &value[1], 0,
3357         &ngx\_http\_fastcgi\_module);
3358 if (flcf->upstream.cache_zone == NULL) {
3359     return NGX\_CONF\_ERROR;
3360 }
3361
3362 return NGX\_CONF\_OK;
3363 }
3364
3365
3366 static char *
3367 ngx\_http\_fastcgi\_cache\_key(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
3368 {
3369     ngx\_http\_fastcgi\_loc\_conf\_t *flcf = conf;
3370
3371     ngx\_str\_t *value;
3372     ngx\_http\_compile\_complex\_value\_t ccv;
3373
3374     value = cf->args->elts;
3375
3376     if (flcf->cache_key.value.data) {
3377         return "is duplicate";
3378     }
3379
3380     ngx\_memzero(&ccv, sizeof(ngx\_http\_compile\_complex\_value\_t));
3381
3382     ccv.cf = cf;
3383     ccv.value = &value[1];
3384     ccv.complex_value = &flcf->cache_key;
3385
3386     if (ngx\_http\_compile\_complex\_value(&ccv) != NGX\_OK) {
3387         return NGX\_CONF\_ERROR;
3388     }
3389
3390     return NGX\_CONF\_OK;
3391 }
3392
3393 #endif
3394
3395
3396 static char *
3397 ngx\_http\_fastcgi\_lowat\_check(ngx\_conf\_t *cf, void *post, void *data)
3398 {
3399     #if (NGX\_FREEBSD)
3400         ssize_t *np = data;
3401
3402         if ((u_long) *np >= ngx\_freebsd\_net\_inet\_tcp\_sendspace) {
3403             ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
3404                 "\ fastcgi_send_lowat\" must be less than %d "
3405                 "(sysctl net.inet.tcp.sendspace)",
3406                 ngx\_freebsd\_net\_inet\_tcp\_sendspace);
3407
3408             return NGX\_CONF\_ERROR;
3409         }
3410
3411     #elif !(NGX\_HAVE\_SO\_SNDLOWAT)
3412         ssize_t *np = data;
3413
3414         ngx\_conf\_log\_error(NGX\_LOG\_WARN, cf, 0,
3415             "\ fastcgi_send_lowat\" is not supported, ignored");
3416
3417         *np = 0;
3418
3419     #endif
3420
3421     return NGX\_CONF\_OK;

```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_flv\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_flv\\_header](#)
- [ngx\\_http\\_flv\\_commands](#)
- [ngx\\_http\\_flv\\_module](#)
- [ngx\\_http\\_flv\\_module\\_ctx](#)

## Functions defined

- [ngx\\_http\\_flv](#)
- [ngx\\_http\\_flv\\_handler](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7 #include <ngx_config.h>
8 #include <ngx_core.h>
9 #include <ngx_http.h>
10
11
12 static char *ngx_http_flv(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
13
14 static ngx_command_t  ngx_http_flv_commands[] = {
15
16     { ngx_string("flv"),
17       NGX_HTTP_LOC_CONF|NGX_CONF_NOARGS,
18       ngx_http_flv,
19       0,
20       0,
21       NULL },
22
23     ngx_null_command
24 };
25
26
27 static u_char  ngx_flv_header[] = "FLV\x1\x5\x0\x0\x0\x9\x0\x0\x0";
28
29
30 static ngx_http_module_t  ngx_http_flv_module_ctx = {
31     NULL,                               /* preconfiguration */
32     NULL,                               /* postconfiguration */
33
34     NULL,                               /* create main configuration */
35     NULL,                               /* init main configuration */
36
37     NULL,                               /* create server configuration */
38     NULL,                               /* merge server configuration */
39
40     NULL,                               /* create location configuration */
41     NULL,                               /* merge location configuration */
42 };
43
44
45 ngx_module_t  ngx_http_flv_module = {
46     NGX_MODULE_V1,
47     &ngx_http_flv_module_ctx,          /* module context */
48     ngx_http_flv_commands,            /* module directives */
```

```

49     NGX\_HTTP\_MODULE,                /* module type */
50     NULL,                          /* init master */
51     NULL,                          /* init module */
52     NULL,                          /* init process */
53     NULL,                          /* init thread */
54     NULL,                          /* exit thread */
55     NULL,                          /* exit process */
56     NULL,                          /* exit master */
57     NGX\_MODULE\_V1\_PADDING
58 };
59
60
61 static ngx\_int\_t
62 ngx\_http\_flv\_handler(ngx\_http\_request\_t *r)
63 {
64     u_char                *last;
65     off_t                 start, len;
66     size_t                root;
67     ngx\_int\_t             rc;
68     ngx\_uint\_t          level, i;
69     ngx\_str\_t           path, value;
70     ngx\_log\_t           *log;
71     ngx\_buf\_t           *b;
72     ngx\_chain\_t         out[2];
73     ngx\_open\_file\_info\_t of;
74     ngx\_http\_core\_loc\_conf\_t *clcf;
75
76     if (!(r->method & (NGX\_HTTP\_GET|NGX\_HTTP\_HEAD))) {
77         return NGX\_HTTP\_NOT\_ALLOWED;
78     }
79
80     if (r->uri.data[r->uri.len - 1] == '/') {
81         return NGX\_DECLINED;
82     }
83
84     rc = ngx\_http\_discard\_request\_body(r);
85
86     if (rc != NGX\_OK) {
87         return rc;
88     }
89
90     last = ngx\_http\_map\_uri\_to\_path(r, &path, &root, 0);
91     if (last == NULL) {
92         return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
93     }
94
95     log = r->connection->log;
96
97     path.len = last - path.data;
98
99     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, log, 0,
100                "http flv filename: \"%V\"", &path);
101
102     clcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_core\_module);
103
104     ngx\_memzero(&of, sizeof(ngx\_open\_file\_info\_t));
105
106     of.read_ahead = clcf->read_ahead;
107     of.directio = clcf->directio;
108     of.valid = clcf->open_file_cache_valid;
109     of.min_uses = clcf->open_file_cache_min_uses;
110     of.errors = clcf->open_file_cache_errors;
111     of.events = clcf->open_file_cache_events;
112
113     if (ngx\_http\_set\_disable\_symlinks(r, clcf, &path, &of) != NGX\_OK) {
114         return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
115     }
116
117     if (ngx\_open\_cached\_file(clcf->open_file_cache, &path, &of, r->pool)
118         != NGX\_OK)
119     {
120         switch (of.err) {
121
122             case 0:
123                 return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
124

```

```

125     case NGX\_ENOENT:
126     case NGX\_ENOTDIR:
127     case NGX\_ENAMETOOLONG:
128
129         level = NGX\_LOG\_ERR;
130         rc = NGX\_HTTP\_NOT\_FOUND;
131         break;
132
133     case NGX\_EACCES:
134 #if (NGX\_HAVE\_OPENAT)
135     case NGX\_EMLINK:
136     case NGX\_ELOOP:
137 #endif
138
139         level = NGX\_LOG\_ERR;
140         rc = NGX\_HTTP\_FORBIDDEN;
141         break;
142
143     default:
144
145         level = NGX\_LOG\_CRIT;
146         rc = NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
147         break;
148     }
149
150     if (rc != NGX\_HTTP\_NOT\_FOUND || clcf->log_not_found) {
151         ngx\_log\_error(level, log, of.err,
152                     "%s \"%s\" failed", of.failed, path.data);
153     }
154
155     return rc;
156 }
157
158 if (!of.is_file) {
159
160     if (ngx\_close\_file(of.fd) == NGX\_FILE\_ERROR) {
161         ngx\_log\_error(NGX\_LOG\_ALERT, log, ngx\_errno,
162                     ngx\_close\_file\_n " \"%s\" failed", path.data);
163     }
164
165     return NGX\_DECLINED;
166 }
167
168 r->root_tested = !r->error_page;
169
170 start = 0;
171 len = of.size;
172 i = 1;
173
174 if (r->args.len) {
175
176     if (ngx\_http\_arg(r, (u_char *) "start", 5, &value) == NGX\_OK) {
177
178         start = ngx\_atoof(value.data, value.len);
179
180         if (start == NGX\_ERROR || start >= len) {
181             start = 0;
182         }
183
184         if (start) {
185             len = sizeof(ngx\_flv\_header) - 1 + len - start;
186             i = 0;
187         }
188     }
189 }
190
191 log->action = "sending flv to client";
192
193 r->headers_out.status = NGX\_HTTP\_OK;
194 r->headers_out.content_length_n = len;
195 r->headers_out.last_modified_time = of.mtime;
196
197 if (ngx\_http\_set\_etag(r) != NGX\_OK) {
198     return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
199 }
200

```



```

201     if (ngx_http_set_content_type(r) != NGX_OK) {
202         return NGX_HTTP_INTERNAL_SERVER_ERROR;
203     }
204
205     if (i == 0) {
206         b = ngx_palloc(r->pool, sizeof(ngx_buf_t));
207         if (b == NULL) {
208             return NGX_HTTP_INTERNAL_SERVER_ERROR;
209         }
210
211         b->pos = ngx_flv_header;
212         b->last = ngx_flv_header + sizeof(ngx_flv_header) - 1;
213         b->memory = 1;
214
215         out[0].buf = b;
216         out[0].next = &out[1];
217     }
218
219     b = ngx_palloc(r->pool, sizeof(ngx_buf_t));
220     if (b == NULL) {
221         return NGX_HTTP_INTERNAL_SERVER_ERROR;
222     }
223
224     b->file = ngx_palloc(r->pool, sizeof(ngx_file_t));
225     if (b->file == NULL) {
226         return NGX_HTTP_INTERNAL_SERVER_ERROR;
227     }
228
229     r->allow_ranges = 1;
230
231     rc = ngx_http_send_header(r);
232
233     if (rc == NGX_ERROR || rc > NGX_OK || r->header_only) {
234         return rc;
235     }
236
237     b->file_pos = start;
238     b->file_last = of.size;
239
240     b->in_file = b->file_last ? 1 : 0;
241     b->last_buf = (r == r->main) ? 1 : 0;
242     b->last_in_chain = 1;
243
244     b->file->fd = of.fd;
245     b->file->name = path;
246     b->file->log = log;
247     b->file->directio = of.is_directio;
248
249     out[1].buf = b;
250     out[1].next = NULL;
251
252     return ngx_http_output_filter(r, &out[i]);
253 }
254
255
256 static char *
257 ngx_http_flv(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
258 {
259     ngx_http_core_loc_conf_t *clcf;
260
261     clcf = ngx_http_conf_get_module_loc_conf(cf, ngx_http_core_module);
262     clcf->handler = ngx_http_flv_handler;
263
264     return NGX_CONF_OK;
265 }

```

[One Level Up](#)

[Top Level](#)

## src/http/modules/nginx\_http\_geo\_module.c - nginx-1.7.10

### Global variables defined

- [ngx\\_http\\_geo\\_commands](#)
- [ngx\\_http\\_geo\\_header](#)
- [ngx\\_http\\_geo\\_module](#)
- [ngx\\_http\\_geo\\_module\\_ctx](#)

### Data types defined

- [ngx\\_http\\_geo\\_conf\\_ctx\\_t](#)
- [ngx\\_http\\_geo\\_ctx\\_t](#)
- [ngx\\_http\\_geo\\_header\\_t](#)
- [ngx\\_http\\_geo\\_high\\_ranges\\_t](#)
- [ngx\\_http\\_geo\\_range\\_t](#)
- [ngx\\_http\\_geo\\_trees\\_t](#)
- [ngx\\_http\\_geo\\_variable\\_value\\_node\\_t](#)

### Functions defined

- [ngx\\_http\\_geo](#)
- [ngx\\_http\\_geo\\_add\\_proxy](#)
- [ngx\\_http\\_geo\\_add\\_range](#)
- [ngx\\_http\\_geo\\_addr](#)
- [ngx\\_http\\_geo\\_block](#)
- [ngx\\_http\\_geo\\_cidr](#)
- [ngx\\_http\\_geo\\_cidr\\_add](#)
- [ngx\\_http\\_geo\\_cidr\\_value](#)
- [ngx\\_http\\_geo\\_cidr\\_variable](#)
- [ngx\\_http\\_geo\\_copy\\_values](#)
- [ngx\\_http\\_geo\\_create\\_binary\\_base](#)
- [ngx\\_http\\_geo\\_delete\\_range](#)
- [ngx\\_http\\_geo\\_include](#)
- [ngx\\_http\\_geo\\_include\\_binary\\_base](#)
- [ngx\\_http\\_geo\\_range](#)
- [ngx\\_http\\_geo\\_range\\_variable](#)

- [ngx\\_http\\_geo\\_real\\_addr](#)
- [ngx\\_http\\_geo\\_value](#)

## Source code

```

1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 typedef struct {
14     ngx_http_variable_value_t    *value;
15     u_short                       start;
16     u_short                       end;
17 } ngx_http_geo_range_t;
18
19
20 typedef struct {
21     ngx_radix_tree_t              *tree;
22     #if (NGX_HAVE_INET6)
23     ngx_radix_tree_t              *tree6;
24     #endif
25 } ngx_http_geo_trees_t;
26
27
28 typedef struct {
29     ngx_http_geo_range_t          **low;
30     ngx_http_variable_value_t     *default_value;
31 } ngx_http_geo_high_ranges_t;
32
33
34 typedef struct {
35     ngx_str_node_t                 sn;
36     ngx_http_variable_value_t     *value;
37     size_t                         offset;
38 } ngx_http_geo_variable_value_node_t;
39
40
41 typedef struct {
42     ngx_http_variable_value_t     *value;
43     ngx_str_t                       *net;
44     ngx_http_geo_high_ranges_t     high;
45     ngx_radix_tree_t              *tree;
46     #if (NGX_HAVE_INET6)
47     ngx_radix_tree_t              *tree6;
48     #endif
49     ngx_rbtree_t                   rbtree;
50     ngx_rbtree_node_t             sentinel;
51     ngx_array_t                    *proxies;
52     ngx_pool_t                     *pool;
53     ngx_pool_t                     *temp_pool;
54
55     size_t                          data_size;
56
57     ngx_str_t                       include_name;
58     ngx_uint_t                      includes;
59     ngx_uint_t                      entries;
60
61     unsigned                        ranges:1;
62     unsigned                        outside_entries:1;
63     unsigned                        allow_binary_include:1;
64     unsigned                        binary_include:1;
65     unsigned                        proxy_recursive:1;
66 } ngx_http_geo_conf_ctx_t;
67
68

```

```

69 typedef struct {
70     union {
71         ngx_http_geo_trees_t      trees;
72         ngx_http_geo_high_ranges_t high;
73     } u;
74
75     ngx_array_t                    *proxies;
76     unsigned                       proxy_recursive:1;
77
78     ngx_int_t                      index;
79 } ngx_http_geo_ctx_t;
80
81
82 static ngx_int_t ngx_http_geo_addr(ngx_http_request_t *r,
83     ngx_http_geo_ctx_t *ctx, ngx_addr_t *addr);
84 static ngx_int_t ngx_http_geo_real_addr(ngx_http_request_t *r,
85     ngx_http_geo_ctx_t *ctx, ngx_addr_t *addr);
86 static char *ngx_http_geo_block(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
87 static char *ngx_http_geo(ngx_conf_t *cf, ngx_command_t *dummy, void *conf);
88 static char *ngx_http_geo_range(ngx_conf_t *cf, ngx_http_geo_conf_ctx_t *ctx,
89     ngx_str_t *value);
90 static char *ngx_http_geo_add_range(ngx_conf_t *cf,
91     ngx_http_geo_conf_ctx_t *ctx, in_addr_t start, in_addr_t end);
92 static ngx_uint_t ngx_http_geo_delete_range(ngx_conf_t *cf,
93     ngx_http_geo_conf_ctx_t *ctx, in_addr_t start, in_addr_t end);
94 static char *ngx_http_geo_cidr(ngx_conf_t *cf, ngx_http_geo_conf_ctx_t *ctx,
95     ngx_str_t *value);
96 static char *ngx_http_geo_cidr_add(ngx_conf_t *cf, ngx_http_geo_conf_ctx_t *ctx,
97     ngx_cidr_t *cidr, ngx_str_t *value, ngx_str_t *net);
98 static ngx_http_variable_value_t *ngx_http_geo_value(ngx_conf_t *cf,
99     ngx_http_geo_conf_ctx_t *ctx, ngx_str_t *value);
100 static char *ngx_http_geo_add_proxy(ngx_conf_t *cf,
101     ngx_http_geo_conf_ctx_t *ctx, ngx_cidr_t *cidr);
102 static ngx_int_t ngx_http_geo_cidr_value(ngx_conf_t *cf, ngx_str_t *net,
103     ngx_cidr_t *cidr);
104 static char *ngx_http_geo_include(ngx_conf_t *cf, ngx_http_geo_conf_ctx_t *ctx,
105     ngx_str_t *name);
106 static ngx_int_t ngx_http_geo_include_binary_base(ngx_conf_t *cf,
107     ngx_http_geo_conf_ctx_t *ctx, ngx_str_t *name);
108 static void ngx_http_geo_create_binary_base(ngx_http_geo_conf_ctx_t *ctx);
109 static u_char *ngx_http_geo_copy_values(u_char *base, u_char *p,
110     ngx_rbtree_node_t *node, ngx_rbtree_node_t *sentinel);
111
112
113 static ngx_command_t  ngx_http_geo_commands[] = {
114
115     { ngx_string("geo"),
116       NGX_HTTP_MAIN_CONF|NGX_CONF_BLOCK|NGX_CONF_TAKE12,
117       ngx_http_geo_block,
118       NGX_HTTP_MAIN_CONF_OFFSET,
119       0,
120       NULL },
121
122     ngx_null_command
123 };
124
125
126 static ngx_http_module_t  ngx_http_geo_module_ctx = {
127     NULL,                                  /* preconfiguration */
128     NULL,                                  /* postconfiguration */
129
130     NULL,                                  /* create main configuration */
131     NULL,                                  /* init main configuration */
132
133     NULL,                                  /* create server configuration */
134     NULL,                                  /* merge server configuration */
135
136     NULL,                                  /* create location configuration */
137     NULL,                                  /* merge location configuration */
138 };
139
140
141 ngx_module_t  ngx_http_geo_module = {
142     NGX_MODULE_V1,
143     &ngx_http_geo_module_ctx,            /* module context */
144     ngx_http_geo_commands,              /* module directives */

```

```

145     NGX_HTTP_MODULE,                /* module type */
146     NULL,                            /* init master */
147     NULL,                            /* init module */
148     NULL,                            /* init process */
149     NULL,                            /* init thread */
150     NULL,                            /* exit thread */
151     NULL,                            /* exit process */
152     NULL,                            /* exit master */
153     NGX_MODULE_V1_PADDING
154 };
155
156 typedef struct {
157     u_char    GEORNG[6];
158     u_char    version;
159     u_char    ptr_size;
160     uint32_t  endianness;
161     uint32_t  crc32;
162 } ngx_http_geo_header_t;
163
164
165 static ngx_http_geo_header_t ngx_http_geo_header = {
166     { 'G', 'E', 'O', 'R', 'N', 'G' }, 0, sizeof(void *), 0x12345678, 0
167 };
168
169
170
171 /* geo range is AF_INET only */
172
173 static ngx_int_t
174 ngx_http_geo_cidr_variable(ngx_http_request_t *r, ngx_http_variable_value_t *v,
175     uintptr_t data)
176 {
177     ngx_http_geo_ctx_t *ctx = (ngx_http_geo_ctx_t *) data;
178
179     in_addr_t    inaddr;
180     ngx_addr_t   addr;
181     struct sockaddr_in *sin;
182     ngx_http_variable_value_t *vv;
183     #if (NGX_HAVE_INET6)
184     u_char      *p;
185     struct in6_addr *inaddr6;
186     #endif
187
188     if (ngx_http_geo_addr(r, ctx, &addr) != NGX_OK) {
189         vv = (ngx_http_variable_value_t *)
190             ngx_radix32tree_find(ctx->u.trees.tree, INADDR_NONE);
191         goto done;
192     }
193
194     switch (addr.sockaddr->sa_family) {
195     #if (NGX_HAVE_INET6)
196     case AF_INET6:
197         inaddr6 = &((struct sockaddr_in6 *) addr.sockaddr)->sin6_addr;
198         p = inaddr6->s6_addr;
199
200         if (IN6_IS_ADDR_V4MAPPED(inaddr6)) {
201             inaddr = p[12] << 24;
202             inaddr += p[13] << 16;
203             inaddr += p[14] << 8;
204             inaddr += p[15];
205
206             vv = (ngx_http_variable_value_t *)
207                 ngx_radix32tree_find(ctx->u.trees.tree, inaddr);
208         } else {
209             vv = (ngx_http_variable_value_t *)
210                 ngx_radix128tree_find(ctx->u.trees.tree6, p);
211         }
212     }
213
214     break;
215     #endif
216
217     default: /* AF_INET */
218         sin = (struct sockaddr_in *) addr.sockaddr;
219         inaddr = ntohl(sin->sin_addr.s_addr);
220

```

```

221     vv = (ngx\_http\_variable\_value\_t *)
222         ngx\_radix32tree\_find(ctx->u.trees.tree, inaddr);
223
224     break;
225 }
226
227 done:
228
229     *v = *vv;
230
231     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
232         "http geo: %v", v);
233
234     return NGX\_OK;
235 }
236
237
238
239 static ngx\_int\_t
240 ngx\_http\_geo\_range\_variable(ngx\_http\_request\_t *r, ngx\_http\_variable\_value\_t *v,
241     uintptr\_t data)
242 {
243     ngx\_http\_geo\_ctx\_t *ctx = (ngx\_http\_geo\_ctx\_t *) data;
244
245     in\_addr\_t         inaddr;
246     ngx\_addr\_t       addr;
247     ngx\_uint\_t       n;
248     struct sockaddr\_in *sin;
249     ngx\_http\_geo\_range\_t *range;
250 #if (NGX\_HAVE\_INET6)
251     u\_char           *p;
252     struct in6\_addr  *inaddr6;
253 #endif
254
255     *v = ctx->u.high.default_value;
256
257     if (ngx\_http\_geo\_addr(r, ctx, &addr) == NGX\_OK) {
258
259         switch (addr.sockaddr->sa_family) {
260
261 #if (NGX\_HAVE\_INET6)
262         case AF\_INET6:
263             inaddr6 = &((struct sockaddr\_in6 *) addr.sockaddr)->sin6_addr;
264
265             if (IN6\_IS\_ADDR\_V4MAPPED(inaddr6)) {
266                 p = inaddr6->s6_addr;
267
268                 inaddr = p[12] << 24;
269                 inaddr += p[13] << 16;
270                 inaddr += p[14] << 8;
271                 inaddr += p[15];
272
273             } else {
274                 inaddr = INADDR\_NONE;
275             }
276
277             break;
278 #endif
279
280         default: /* AF\_INET */
281             sin = (struct sockaddr\_in *) addr.sockaddr;
282             inaddr = ntohl(sin->sin_addr.s_addr);
283             break;
284         }
285
286     } else {
287         inaddr = INADDR\_NONE;
288     }
289
290     if (ctx->u.high.low) {
291         range = ctx->u.high.low[inaddr >> 16];
292
293         if (range) {
294             n = inaddr & 0xffff;
295             do {
296                 if (n >= (ngx\_uint\_t) range->start

```

```

297         && n <= (ngx_uint_t) range->end)
298     {
299         *v = *range->value;
300         break;
301     }
302     } while ((++range)->value);
303 }
304 }
305
306 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
307     "http geo: %v", v);
308
309 return NGX_OK;
310 }
311
312
313 static ngx_int_t
314 ngx_http_geo_addr(ngx_http_request_t *r, ngx_http_geo_ctx_t *ctx,
315     ngx_addr_t *addr)
316 {
317     ngx_array_t *xfwd;
318
319     if (ngx_http_geo_real_addr(r, ctx, addr) != NGX_OK) {
320         return NGX_ERROR;
321     }
322
323     xfwd = &r->headers_in.x_forwarded_for;
324
325     if (xfwd->nelts > 0 && ctx->proxies != NULL) {
326         (void) ngx_http_get_forwarded_addr(r, addr, xfwd, NULL,
327             ctx->proxies, ctx->proxy_recursive);
328     }
329
330     return NGX_OK;
331 }
332
333
334 static ngx_int_t
335 ngx_http_geo_real_addr(ngx_http_request_t *r, ngx_http_geo_ctx_t *ctx,
336     ngx_addr_t *addr)
337 {
338     ngx_http_variable_value_t *v;
339
340     if (ctx->index == -1) {
341         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
342             "http geo started: %v", &r->connection->addr_text);
343
344         addr->sockaddr = r->connection->sockaddr;
345         addr->socklen = r->connection->socklen;
346         /* addr->name = r->connection->addr_text; */
347
348         return NGX_OK;
349     }
350
351     v = ngx_http_get_flushed_variable(r, ctx->index);
352
353     if (v == NULL || v->not_found) {
354         ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
355             "http geo not found");
356
357         return NGX_ERROR;
358     }
359
360     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
361         "http geo started: %v", v);
362
363     if (ngx_parse_addr(r->pool, addr, v->data, v->len) == NGX_OK) {
364         return NGX_OK;
365     }
366
367     return NGX_ERROR;
368 }
369
370
371 static char *
372 ngx_http_geo_block(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)

```

```

373 {
374     char                *rv;
375     size_t              len;
376     ngx_str_t           *value, name;
377     ngx_uint_t          i;
378     ngx_conf_t          save;
379     ngx_pool_t          *pool;
380     ngx_array_t         *a;
381     ngx_http_variable_t *var;
382     ngx_http_geo_ctx_t  *geo;
383     ngx_http_geo_conf_ctx_t  ctx;
384     #if (NGX_HAVE_INET6)
385     static struct in6_addr  zero;
386     #endif
387
388     value = cf->args->elts;
389
390     geo = ngx_palloc(cf->pool, sizeof(ngx_http_geo_ctx_t));
391     if (geo == NULL) {
392         return NGX_CONF_ERROR;
393     }
394
395     name = value[1];
396
397     if (name.data[0] != '$') {
398         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
399             "invalid variable name \"%V\"", &name);
400         return NGX_CONF_ERROR;
401     }
402
403     name.len--;
404     name.data++;
405
406     if (cf->args->nelts == 3) {
407
408         geo->index = ngx_http_get_variable_index(cf, &name);
409         if (geo->index == NGX_ERROR) {
410             return NGX_CONF_ERROR;
411         }
412
413         name = value[2];
414
415         if (name.data[0] != '$') {
416             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
417                 "invalid variable name \"%V\"", &name);
418             return NGX_CONF_ERROR;
419         }
420
421         name.len--;
422         name.data++;
423
424     } else {
425         geo->index = -1;
426     }
427
428     var = ngx_http_add_variable(cf, &name, NGX_HTTP_VAR_CHANGEABLE);
429     if (var == NULL) {
430         return NGX_CONF_ERROR;
431     }
432
433     pool = ngx_create_pool(NGX_DEFAULT_POOL_SIZE, cf->log);
434     if (pool == NULL) {
435         return NGX_CONF_ERROR;
436     }
437
438     ngx_memzero(&ctx, sizeof(ngx_http_geo_conf_ctx_t));
439
440     ctx.temp_pool = ngx_create_pool(NGX_DEFAULT_POOL_SIZE, cf->log);
441     if (ctx.temp_pool == NULL) {
442         return NGX_CONF_ERROR;
443     }
444
445     ngx_rbtree_init(&ctx.rbtree, &ctx.sentinel, ngx_str_rbtree_insert_value);
446
447     ctx.pool = cf->pool;
448     ctx.data_size = sizeof(ngx_http_geo_header_t)

```



```

449         + sizeof(ngx\_http\_variable\_value\_t)
450         + 0x10000 * sizeof(ngx\_http\_geo\_range\_t *);
451 ctx.allow_binary_include = 1;
452
453 save = *cf;
454 cf->pool = pool;
455 cf->ctx = &ctx;
456 cf->handler = ngx\_http\_geo;
457 cf->handler_conf = conf;
458
459 rv = ngx\_conf\_parse(cf, NULL);
460
461 *cf = save;
462
463 geo->proxies = ctx.proxies;
464 geo->proxy_recursive = ctx.proxy_recursive;
465
466 if (ctx.ranges) {
467     if (ctx.high.low && !ctx.binary_include) {
468         for (i = 0; i < 0x10000; i++) {
469             a = (ngx\_array\_t *) ctx.high.low[i];
470
471             if (a == NULL || a->nelts == 0) {
472                 continue;
473             }
474
475             len = a->nelts * sizeof(ngx\_http\_geo\_range\_t);
476
477             ctx.high.low[i] = ngx\_palloc(cf->pool, len + sizeof(void *));
478             if (ctx.high.low[i] == NULL) {
479                 return NGX\_CONF\_ERROR;
480             }
481
482             ngx\_memcpy(ctx.high.low[i], a->elts, len);
483             ctx.high.low[i][a->nelts].value = NULL;
484             ctx.data_size += len + sizeof(void *);
485         }
486
487         if (ctx.allow_binary_include
488             && !ctx.outside_entries
489             && ctx.entries > 100000
490             && ctx.includes == 1)
491         {
492             ngx\_http\_geo\_create\_binary\_base(&ctx);
493         }
494
495     }
496
497     if (ctx.high.default_value == NULL) {
498         ctx.high.default_value = &ngx\_http\_variable\_null\_value;
499     }
500
501     geo->u.high = ctx.high;
502
503     var->get_handler = ngx\_http\_geo\_range\_variable;
504     var->data = (uintptr\_t) geo;
505
506     ngx\_destroy\_pool(ctx.temp_pool);
507     ngx\_destroy\_pool(pool);
508
509 } else {
510     if (ctx.tree == NULL) {
511         ctx.tree = ngx\_radix\_tree\_create(cf->pool, -1);
512         if (ctx.tree == NULL) {
513             return NGX\_CONF\_ERROR;
514         }
515     }
516
517     geo->u.trees.tree = ctx.tree;
518
519 #if (NGX_HAVE_INET6)
520     if (ctx.tree6 == NULL) {
521         ctx.tree6 = ngx\_radix\_tree\_create(cf->pool, -1);
522         if (ctx.tree6 == NULL) {
523             return NGX\_CONF\_ERROR;
524         }
525     }

```

```

525     }
526
527     geo->u.trees.tree6 = ctx.tree6;
528 #endif
529
530     var->get_handler = ngx_http_geo_cidr_variable;
531     var->data = (uintptr_t) geo;
532
533     ngx_destroy_pool(ctx.temp_pool);
534     ngx_destroy_pool(pool);
535
536     if (ngx_radix32tree_insert(ctx.tree, 0, 0,
537                               (uintptr_t) &ngx_http_variable_null_value)
538         == NGX_ERROR)
539     {
540         return NGX_CONF_ERROR;
541     }
542
543     /* NGX_BUSY is okay (default was set explicitly) */
544
545 #if (NGX_HAVE_INET6)
546     if (ngx_radix128tree_insert(ctx.tree6, zero.s6_addr, zero.s6_addr,
547                               (uintptr_t) &ngx_http_variable_null_value)
548         == NGX_ERROR)
549     {
550         return NGX_CONF_ERROR;
551     }
552 #endif
553     }
554
555     return rv;
556 }
557
558
559 static char *
560 ngx_http_geo(ngx_conf_t *cf, ngx_command_t *dummy, void *conf)
561 {
562     char          *rv;
563     ngx_str_t     *value;
564     ngx_cidr_t    cidr;
565     ngx_http_geo_conf_ctx_t *ctx;
566
567     ctx = cf->ctx;
568
569     value = cf->args->elts;
570
571     if (cf->args->nelts == 1) {
572
573         if (ngx_strcmp(value[0].data, "ranges") == 0) {
574
575             if (ctx->tree
576 #if (NGX_HAVE_INET6)
577                 || ctx->tree6
578 #endif
579             )
580             {
581                 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
582                                   "the \"ranges\" directive must be "
583                                   "the first directive inside \"geo\" block");
584                 goto failed;
585             }
586
587             ctx->ranges = 1;
588
589             rv = NGX_CONF_OK;
590
591             goto done;
592         }
593
594         else if (ngx_strcmp(value[0].data, "proxy_recursive") == 0) {
595             ctx->proxy_recursive = 1;
596             rv = NGX_CONF_OK;
597             goto done;
598         }
599     }
600 }

```

```

601     if (cf->args->nelts != 2) {
602         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
603             "invalid number of the geo parameters");
604         goto failed;
605     }
606
607     if (ngx_strcmp(value[0].data, "include") == 0) {
608         rv = ngx_http_geo_include(cf, ctx, &value[1]);
609
610         goto done;
611     }
612
613     } else if (ngx_strcmp(value[0].data, "proxy") == 0) {
614
615         if (ngx_http_geo_cidr_value(cf, &value[1], &cidr) != NGX_OK) {
616             goto failed;
617         }
618
619         rv = ngx_http_geo_add_proxy(cf, ctx, &cidr);
620
621         goto done;
622     }
623
624     if (ctx->ranges) {
625         rv = ngx_http_geo_range(cf, ctx, value);
626     } else {
627         rv = ngx_http_geo_cidr(cf, ctx, value);
628     }
629 }
630
631 done:
632
633     ngx_reset_pool(cf->pool);
634
635     return rv;
636
637 failed:
638
639     ngx_reset_pool(cf->pool);
640
641     return NGX_CONF_ERROR;
642 }
643
644
645 static char *
646 ngx_http_geo_range(ngx_conf_t *cf, ngx_http_geo_conf_ctx_t *ctx,
647     ngx_str_t *value)
648 {
649     u_char      *p, *last;
650     in_addr_t    start, end;
651     ngx_str_t    *net;
652     ngx_uint_t   del;
653
654     if (ngx_strcmp(value[0].data, "default") == 0) {
655
656         if (ctx->high.default_value) {
657             ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
658                 "duplicate default geo range value: \"%V\", old value: \"%v\"",
659                 &value[1], ctx->high.default_value);
660         }
661
662         ctx->high.default_value = ngx_http_geo_value(cf, ctx, &value[1]);
663         if (ctx->high.default_value == NULL) {
664             return NGX_CONF_ERROR;
665         }
666
667         return NGX_CONF_OK;
668     }
669
670     if (ctx->binary_include) {
671         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
672             "binary geo range base \"%s\" cannot be mixed with usual entries",
673             ctx->include_name.data);
674         return NGX_CONF_ERROR;
675     }
676 }

```

```

677     if (ctx->high.low == NULL) {
678         ctx->high.low = ngx_pcalloc(ctx->pool,
679             0x10000 * sizeof(ngx_http_geo_range_t *));
680         if (ctx->high.low == NULL) {
681             return NGX_CONF_ERROR;
682         }
683     }
684
685     ctx->entries++;
686     ctx->outside_entries = 1;
687
688     if (ngx_strcmp(value[0].data, "delete") == 0) {
689         net = &value[1];
690         del = 1;
691     }
692     else {
693         net = &value[0];
694         del = 0;
695     }
696
697     last = net->data + net->len;
698
699     p = ngx_strlchr(net->data, last, '-');
700
701     if (p == NULL) {
702         goto invalid;
703     }
704
705     start = ngx_inet_addr(net->data, p - net->data);
706
707     if (start == INADDR_NONE) {
708         goto invalid;
709     }
710
711     start = ntohl(start);
712
713     p++;
714
715     end = ngx_inet_addr(p, last - p);
716
717     if (end == INADDR_NONE) {
718         goto invalid;
719     }
720
721     end = ntohl(end);
722
723     if (start > end) {
724         goto invalid;
725     }
726
727     if (del) {
728         if (ngx_http_geo_delete_range(cf, ctx, start, end) {
729             ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
730                 "no address range \"%V\" to delete", net);
731         }
732
733         return NGX_CONF_OK;
734     }
735
736     ctx->value = ngx_http_geo_value(cf, ctx, &value[1]);
737
738     if (ctx->value == NULL) {
739         return NGX_CONF_ERROR;
740     }
741
742     ctx->net = net;
743
744     return ngx_http_geo_add_range(cf, ctx, start, end);
745
746 invalid:
747
748     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0, "invalid range \"%V\"", net);
749
750     return NGX_CONF_ERROR;
751 }
752

```

```

753
754 /* the add procedure is optimized to add a growing up sequence */
755
756 static char *
757 ngx_http_geo_add_range(ngx_conf_t *cf, ngx_http_geo_conf_ctx_t *ctx,
758     in_addr_t start, in_addr_t end)
759 {
760     in_addr_t      n;
761     ngx_uint_t     h, i, s, e;
762     ngx_array_t    *a;
763     ngx_http_geo_range_t *range;
764
765     for (n = start; n <= end; n = (n + 0x10000) & 0xffff0000) {
766
767         h = n >> 16;
768
769         if (n == start) {
770             s = n & 0xffff;
771         } else {
772             s = 0;
773         }
774
775         if ((n | 0xffff) > end) {
776             e = end & 0xffff;
777
778         } else {
779             e = 0xffff;
780         }
781
782         a = (ngx_array_t *) ctx->high.low[h];
783
784         if (a == NULL) {
785             a = ngx_array_create(ctx->temp_pool, 64,
786                 sizeof(ngx_http_geo_range_t));
787             if (a == NULL) {
788                 return NGX_CONF_ERROR;
789             }
790
791             ctx->high.low[h] = (ngx_http_geo_range_t *) a;
792         }
793
794         i = a->nelts;
795         range = a->elts;
796
797         while (i) {
798
799             i--;
800
801             if (e < (ngx_uint_t) range[i].start) {
802                 continue;
803             }
804
805             if (s > (ngx_uint_t) range[i].end) {
806
807                 /* add after the range */
808
809                 range = ngx_array_push(a);
810                 if (range == NULL) {
811                     return NGX_CONF_ERROR;
812                 }
813
814                 range = a->elts;
815
816                 ngx_memmove(&range[i + 2], &range[i + 1],
817                     (a->nelts - 2 - i) * sizeof(ngx_http_geo_range_t));
818
819                 range[i + 1].start = (u_short) s;
820                 range[i + 1].end = (u_short) e;
821                 range[i + 1].value = ctx->value;
822
823                 goto next;
824             }
825
826             if (s == (ngx_uint_t) range[i].start
827                 && e == (ngx_uint_t) range[i].end)
828                 {

```

```

829     ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
830         "duplicate range \"%V\"", value: \"%V\"", old value: \"%V\"",
831         ctx->net, ctx->value, range[i].value);
832
833     range[i].value = ctx->value;
834
835     goto next;
836 }
837
838 if (s > (ngx_uint_t) range[i].start
839     && e < (ngx_uint_t) range[i].end)
840 {
841     /* split the range and insert the new one */
842
843     range = ngx_array_push(a);
844     if (range == NULL) {
845         return NGX_CONF_ERROR;
846     }
847
848     range = ngx_array_push(a);
849     if (range == NULL) {
850         return NGX_CONF_ERROR;
851     }
852
853     range = a->elts;
854
855     ngx_memmove(&range[i + 3], &range[i + 1],
856         (a->nelts - 3 - i) * sizeof(ngx_http_geo_range_t));
857
858     range[i + 2].start = (u_short) (e + 1);
859     range[i + 2].end = range[i].end;
860     range[i + 2].value = range[i].value;
861
862     range[i + 1].start = (u_short) s;
863     range[i + 1].end = (u_short) e;
864     range[i + 1].value = ctx->value;
865
866     range[i].end = (u_short) (s - 1);
867
868     goto next;
869 }
870
871 if (s == (ngx_uint_t) range[i].start
872     && e < (ngx_uint_t) range[i].end)
873 {
874     /* shift the range start and insert the new range */
875
876     range = ngx_array_push(a);
877     if (range == NULL) {
878         return NGX_CONF_ERROR;
879     }
880
881     range = a->elts;
882
883     ngx_memmove(&range[i + 1], &range[i],
884         (a->nelts - 1 - i) * sizeof(ngx_http_geo_range_t));
885
886     range[i + 1].start = (u_short) (e + 1);
887
888     range[i].start = (u_short) s;
889     range[i].end = (u_short) e;
890     range[i].value = ctx->value;
891
892     goto next;
893 }
894
895 if (s > (ngx_uint_t) range[i].start
896     && e == (ngx_uint_t) range[i].end)
897 {
898     /* shift the range end and insert the new range */
899
900     range = ngx_array_push(a);
901     if (range == NULL) {
902         return NGX_CONF_ERROR;
903     }
904

```

```

905     range = a->elts;
906
907     ngx_memmove(&range[i + 2], &range[i + 1],
908               (a->nelts - 2 - i) * sizeof(ngx_http_geo_range_t));
909
910     range[i + 1].start = (u_short) s;
911     range[i + 1].end = (u_short) e;
912     range[i + 1].value = ctx->value;
913
914     range[i].end = (u_short) (s - 1);
915
916     goto next;
917 }
918
919 s = (ngx_uint_t) range[i].start;
920 e = (ngx_uint_t) range[i].end;
921
922 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
923                  "range \"%V\" overlaps \"%d.%d.%d.%d-%d.%d.%d.%d\"",
924                  ctx->net,
925                  h >> 8, h & 0xff, s >> 8, s & 0xff,
926                  h >> 8, h & 0xff, e >> 8, e & 0xff);
927
928     return NGX_CONF_ERROR;
929 }
930
931 /* add the first range */
932
933 range = ngx_array_push(a);
934 if (range == NULL) {
935     return NGX_CONF_ERROR;
936 }
937
938 range->start = (u_short) s;
939 range->end = (u_short) e;
940 range->value = ctx->value;
941
942 next:
943
944     continue;
945 }
946
947 return NGX_CONF_OK;
948 }
949
950
951 static ngx_uint_t
952 ngx_http_geo_delete_range(ngx_conf_t *cf, ngx_http_geo_conf_ctx_t *ctx,
953                          in_addr_t start, in_addr_t end)
954 {
955     in_addr_t          n;
956     ngx_uint_t        h, i, s, e, warn;
957     ngx_array_t        *a;
958     ngx_http_geo_range_t *range;
959
960     warn = 0;
961
962     for (n = start; n <= end; n += 0x10000) {
963
964         h = n >> 16;
965
966         if (n == start) {
967             s = n & 0xffff;
968         } else {
969             s = 0;
970         }
971
972         if ((n | 0xffff) > end) {
973             e = end & 0xffff;
974         } else {
975             e = 0xffff;
976         }
977
978         a = (ngx_array_t *) ctx->high.low[h];
979
980

```

```

981     if (a == NULL) {
982         warn = 1;
983         continue;
984     }
985
986     range = a->elts;
987     for (i = 0; i < a->nelts; i++) {
988
989         if (s == (ngx_uint_t) range[i].start
990             && e == (ngx_uint_t) range[i].end)
991             {
992                 ngx_memmove(&range[i], &range[i + 1],
993                             (a->nelts - 1 - i) * sizeof(ngx_http_geo_range_t));
994
995                 a->nelts--;
996
997                 break;
998             }
999
1000         if (s != (ngx_uint_t) range[i].start
1001             && e != (ngx_uint_t) range[i].end)
1002             {
1003                 continue;
1004             }
1005
1006         warn = 1;
1007     }
1008 }
1009
1010 return warn;
1011 }
1012
1013
1014 static char *
1015 ngx_http_geo_cidr(ngx_conf_t *cf, ngx_http_geo_conf_ctx_t *ctx,
1016                  ngx_str_t *value)
1017 {
1018     char      *rv;
1019     ngx_int_t  rc, del;
1020     ngx_str_t  *net;
1021     ngx_cidr_t  cidr;
1022
1023     if (ctx->tree == NULL) {
1024         ctx->tree = ngx_radix_tree_create(ctx->pool, -1);
1025         if (ctx->tree == NULL) {
1026             return NGX_CONF_ERROR;
1027         }
1028     }
1029
1030 #if (NGX_HAVE_INET6)
1031     if (ctx->tree6 == NULL) {
1032         ctx->tree6 = ngx_radix_tree_create(ctx->pool, -1);
1033         if (ctx->tree6 == NULL) {
1034             return NGX_CONF_ERROR;
1035         }
1036     }
1037 #endif
1038
1039     if (ngx_strcmp(value[0].data, "default") == 0) {
1040         cidr.family = AF_INET;
1041         cidr.u.in.addr = 0;
1042         cidr.u.in.mask = 0;
1043
1044         rv = ngx_http_geo_cidr_add(cf, ctx, &cidr, &value[1], &value[0]);
1045
1046         if (rv != NGX_CONF_OK) {
1047             return rv;
1048         }
1049
1050 #if (NGX_HAVE_INET6)
1051         cidr.family = AF_INET6;
1052         ngx_memzero(&cidr.u.in6, sizeof(ngx_in6_cidr_t));
1053
1054         rv = ngx_http_geo_cidr_add(cf, ctx, &cidr, &value[1], &value[0]);
1055
1056         if (rv != NGX_CONF_OK) {

```



```

1057         return rv;
1058     }
1059 #endif
1060
1061     return NGX_CONF_OK;
1062 }
1063
1064 if (ngx_strcmp(value[0].data, "delete") == 0) {
1065     net = &value[1];
1066     del = 1;
1067 } else {
1068     net = &value[0];
1069     del = 0;
1070 }
1071
1072
1073 if (ngx_http_geo_cidr_value(cf, net, &cidr) != NGX_OK) {
1074     return NGX_CONF_ERROR;
1075 }
1076
1077 if (cidr.family == AF_INET) {
1078     cidr.u.in.addr = ntohl(cidr.u.in.addr);
1079     cidr.u.in.mask = ntohl(cidr.u.in.mask);
1080 }
1081
1082 if (del) {
1083     switch (cidr.family) {
1084
1085 #if (NGX_HAVE_INET6)
1086         case AF_INET6:
1087             rc = ngx_radix128tree_delete(ctx->tree6,
1088                                     cidr.u.in6.addr.s6_addr,
1089                                     cidr.u.in6.mask.s6_addr);
1090             break;
1091 #endif
1092
1093         default: /* AF_INET */
1094             rc = ngx_radix32tree_delete(ctx->tree, cidr.u.in.addr,
1095                                         cidr.u.in.mask);
1096             break;
1097     }
1098
1099     if (rc != NGX_OK) {
1100         ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
1101                             "no network \"%V\" to delete", net);
1102     }
1103
1104     return NGX_CONF_OK;
1105 }
1106
1107 return ngx_http_geo_cidr_add(cf, ctx, &cidr, &value[1], net);
1108 }
1109
1110
1111 static char *
1112 ngx_http_geo_cidr_add(ngx_conf_t *cf, ngx_http_geo_conf_ctx_t *ctx,
1113 ngx_cidr_t *cidr, ngx_str_t *value, ngx_str_t *net)
1114 {
1115     ngx_int_t rc;
1116     ngx_http_variable_value_t *val, *old;
1117
1118     val = ngx_http_geo_value(cf, ctx, value);
1119
1120     if (val == NULL) {
1121         return NGX_CONF_ERROR;
1122     }
1123
1124     switch (cidr->family) {
1125
1126 #if (NGX_HAVE_INET6)
1127         case AF_INET6:
1128             rc = ngx_radix128tree_insert(ctx->tree6, cidr->u.in6.addr.s6_addr,
1129                                         cidr->u.in6.mask.s6_addr,
1130                                         (uintptr_t) val);
1131
1132             if (rc == NGX_OK) {

```

```

1133     return NGX_CONF_OK;
1134 }
1135
1136 if (rc == NGX_ERROR) {
1137     return NGX_CONF_ERROR;
1138 }
1139
1140 /* rc == NGX_BUSY */
1141
1142 old = (ngx_http_variable_value_t *)
1143     ngx_radix128tree_find(ctx->tree6,
1144         cidr->u.in6.addr.s6_addr);
1145
1146 ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
1147     "duplicate network \"%V\", value: \"%v\", old value: \"%v\"",
1148     net, val, old);
1149
1150 rc = ngx_radix128tree_delete(ctx->tree6,
1151     cidr->u.in6.addr.s6_addr,
1152     cidr->u.in6.mask.s6_addr);
1153
1154 if (rc == NGX_ERROR) {
1155     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0, "invalid radix tree");
1156     return NGX_CONF_ERROR;
1157 }
1158
1159 rc = ngx_radix128tree_insert(ctx->tree6, cidr->u.in6.addr.s6_addr,
1160     cidr->u.in6.mask.s6_addr,
1161     (uintptr_t) val);
1162
1163 break;
1164 #endif
1165
1166 default: /* AF_INET */
1167     rc = ngx_radix32tree_insert(ctx->tree, cidr->u.in.addr,
1168         cidr->u.in.mask, (uintptr_t) val);
1169
1170 if (rc == NGX_OK) {
1171     return NGX_CONF_OK;
1172 }
1173
1174 if (rc == NGX_ERROR) {
1175     return NGX_CONF_ERROR;
1176 }
1177
1178 /* rc == NGX_BUSY */
1179
1180 old = (ngx_http_variable_value_t *)
1181     ngx_radix32tree_find(ctx->tree, cidr->u.in.addr);
1182
1183 ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
1184     "duplicate network \"%V\", value: \"%v\", old value: \"%v\"",
1185     net, val, old);
1186
1187 rc = ngx_radix32tree_delete(ctx->tree,
1188     cidr->u.in.addr, cidr->u.in.mask);
1189
1190 if (rc == NGX_ERROR) {
1191     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0, "invalid radix tree");
1192     return NGX_CONF_ERROR;
1193 }
1194
1195 rc = ngx_radix32tree_insert(ctx->tree, cidr->u.in.addr,
1196     cidr->u.in.mask, (uintptr_t) val);
1197
1198 break;
1199 }
1200
1201 if (rc == NGX_OK) {
1202     return NGX_CONF_OK;
1203 }
1204
1205 return NGX_CONF_ERROR;
1206 }
1207
1208

```

```

1209 static ngx_http_variable_value_t *
1210 ngx_http_geo_value(ngx_conf_t *cf, ngx_http_geo_conf_ctx_t *ctx,
1211 ngx_str_t *value)
1212 {
1213     uint32_t                hash;
1214     ngx_http_variable_value_t *val;
1215     ngx_http_geo_variable_value_node_t *gvv;
1216
1217     hash = ngx_crc32_long(value->data, value->len);
1218
1219     gvvn = (ngx_http_geo_variable_value_node_t *)
1220             ngx_str_rbtrees_lookup(&ctx->rbtree, value, hash);
1221
1222     if (gvvn) {
1223         return gvvn->value;
1224     }
1225
1226     val = ngx_palloc(ctx->pool, sizeof(ngx_http_variable_value_t));
1227     if (val == NULL) {
1228         return NULL;
1229     }
1230
1231     val->len = value->len;
1232     val->data = ngx_pstrdup(ctx->pool, value);
1233     if (val->data == NULL) {
1234         return NULL;
1235     }
1236
1237     val->valid = 1;
1238     val->no_cacheable = 0;
1239     val->not_found = 0;
1240
1241     gvvn = ngx_palloc(ctx->temp_pool,
1242                     sizeof(ngx_http_geo_variable_value_node_t));
1243     if (gvvn == NULL) {
1244         return NULL;
1245     }
1246
1247     gvvn->sn.node.key = hash;
1248     gvvn->sn.str.len = val->len;
1249     gvvn->sn.str.data = val->data;
1250     gvvn->value = val;
1251     gvvn->offset = 0;
1252
1253     ngx_rbtrees_insert(&ctx->rbtree, &gvvn->sn.node);
1254
1255     ctx->data_size += ngx_align(sizeof(ngx_http_variable_value_t) + value->len,
1256                               sizeof(void *));
1257
1258     return val;
1259 }
1260
1261 static char *
1262 ngx_http_geo_add_proxy(ngx_conf_t *cf, ngx_http_geo_conf_ctx_t *ctx,
1263 ngx_cidr_t *cidr)
1264 {
1265     ngx_cidr_t *c;
1266
1267     if (ctx->proxies == NULL) {
1268         ctx->proxies = ngx_array_create(ctx->pool, 4, sizeof(ngx_cidr_t));
1269         if (ctx->proxies == NULL) {
1270             return NGX_CONF_ERROR;
1271         }
1272     }
1273
1274     c = ngx_array_push(ctx->proxies);
1275     if (c == NULL) {
1276         return NGX_CONF_ERROR;
1277     }
1278
1279     *c = *cidr;
1280
1281     return NGX_CONF_OK;
1282 }
1283
1284

```

```

1285 static ngx_int_t
1286 ngx_http_geo_cidr_value(ngx_conf_t *cf, ngx_str_t *net, ngx_cidr_t *cidr)
1287 {
1288     ngx_int_t rc;
1289
1290     if (ngx_strcmp(net->data, "255.255.255.255") == 0) {
1291         cidr->family = AF_INET;
1292         cidr->u.in.addr = 0xffffffff;
1293         cidr->u.in.mask = 0xffffffff;
1294
1295         return NGX_OK;
1296     }
1297
1298     rc = ngx_ptocidr(net, cidr);
1299
1300     if (rc == NGX_ERROR) {
1301         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0, "invalid network \"%V\"", net);
1302         return NGX_ERROR;
1303     }
1304
1305     if (rc == NGX_DONE) {
1306         ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
1307             "low address bits of %V are meaningless", net);
1308     }
1309
1310     return NGX_OK;
1311 }
1312
1313
1314
1315 static char *
1316 ngx_http_geo_include(ngx_conf_t *cf, ngx_http_geo_conf_ctx_t *ctx,
1317     ngx_str_t *name)
1318 {
1319     char *rv;
1320     ngx_str_t file;
1321
1322     file.len = name->len + 4;
1323     file.data = ngx_pnalloc(ctx->temp_pool, name->len + 5);
1324     if (file.data == NULL) {
1325         return NGX_CONF_ERROR;
1326     }
1327
1328     ngx_sprintf(file.data, "%V.bin%Z", name);
1329
1330     if (ngx_conf_full_name(cf->cycle, &file, 1) != NGX_OK) {
1331         return NGX_CONF_ERROR;
1332     }
1333
1334     if (ctx->ranges) {
1335         ngx_log_debug1(NGX_LOG_DEBUG_CORE, cf->log, 0, "include %s", file.data);
1336
1337         switch (ngx_http_geo_include_binary_base(cf, ctx, &file)) {
1338             case NGX_OK:
1339                 return NGX_CONF_OK;
1340             case NGX_ERROR:
1341                 return NGX_CONF_ERROR;
1342             default:
1343                 break;
1344         }
1345     }
1346
1347     file.len -= 4;
1348     file.data[file.len] = '\0';
1349
1350     ctx->include_name = file;
1351
1352     if (ctx->outside_entries) {
1353         ctx->allow_binary_include = 0;
1354     }
1355
1356     ngx_log_debug1(NGX_LOG_DEBUG_CORE, cf->log, 0, "include %s", file.data);
1357
1358     rv = ngx_conf_parse(cf, &file);
1359
1360     ctx->includes++;

```

```

1361     ctx->outside_entries = 0;
1362
1363     return rv;
1364 }
1365
1366
1367 static ngx_int_t
1368 ngx_http_geo_include_binary_base(ngx_conf_t *cf, ngx_http_geo_conf_ctx_t *ctx,
1369     ngx_str_t *name)
1370 {
1371     u_char          *base, ch;
1372     time_t          mtime;
1373     size_t          size, len;
1374     ssize_t         n;
1375     uint32_t        crc32;
1376     ngx_err_t       err;
1377     ngx_int_t       rc;
1378     ngx_uint_t      i;
1379     ngx_file_t      file;
1380     ngx_file_info_t fi;
1381     ngx_http_geo_range_t *range, **ranges;
1382     ngx_http_geo_header_t *header;
1383     ngx_http_variable_value_t *vv;
1384
1385     ngx_memzero(&file, sizeof(ngx_file_t));
1386     file.name = *name;
1387     file.log = cf->log;
1388
1389     file.fd = ngx_open_file(name->data, NGX_FILE_RDONLY, 0, 0);
1390     if (file.fd == NGX_INVALID_FILE) {
1391         err = ngx_errno;
1392         if (err != NGX_ENOENT) {
1393             ngx_conf_log_error(NGX_LOG_CRIT, cf, err,
1394                 ngx_open_file_n "\'%s\' failed", name->data);
1395         }
1396         return NGX_DECLINED;
1397     }
1398
1399     if (ctx->outside_entries) {
1400         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1401             "binary geo range base \''%s\' cannot be mixed with usual entries",
1402             name->data);
1403         rc = NGX_ERROR;
1404         goto done;
1405     }
1406
1407     if (ctx->binary_include) {
1408         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1409             "second binary geo range base \''%s\' cannot be mixed with \''%s\'",
1410             name->data, ctx->include_name.data);
1411         rc = NGX_ERROR;
1412         goto done;
1413     }
1414
1415     if (ngx_fd_info(file.fd, &fi) == NGX_FILE_ERROR) {
1416         ngx_conf_log_error(NGX_LOG_CRIT, cf, ngx_errno,
1417             ngx_fd_info_n "\'%s\' failed", name->data);
1418         goto failed;
1419     }
1420
1421     size = (size_t) ngx_file_size(&fi);
1422     mtime = ngx_file_mtime(&fi);
1423
1424     ch = name->data[name->len - 4];
1425     name->data[name->len - 4] = '\0';
1426
1427     if (ngx_file_info(name->data, &fi) == NGX_FILE_ERROR) {
1428         ngx_conf_log_error(NGX_LOG_CRIT, cf, ngx_errno,
1429             ngx_file_info_n "\'%s\' failed", name->data);
1430         goto failed;
1431     }
1432
1433     name->data[name->len - 4] = ch;
1434
1435     if (mtime < ngx_file_mtime(&fi)) {
1436         ngx_conf_log_error(NGX_LOG_WARN, cf, 0,

```

```

1437         "stale binary geo range base \"%s\"", name->data);
1438     goto failed;
1439 }
1440
1441 base = ngx_palloc(ctx->pool, size);
1442 if (base == NULL) {
1443     goto failed;
1444 }
1445
1446 n = ngx_read_file(&file, base, size, 0);
1447
1448 if (n == NGX_ERROR) {
1449     ngx_conf_log_error(NGX_LOG_CRIT, cf, ngx_errno,
1450         ngx_read_file_n " \"%s\" failed", name->data);
1451     goto failed;
1452 }
1453
1454 if ((size_t) n != size) {
1455     ngx_conf_log_error(NGX_LOG_CRIT, cf, 0,
1456         ngx_read_file_n " \"%s\" returned only %z bytes instead of %z",
1457         name->data, n, size);
1458     goto failed;
1459 }
1460
1461 header = (ngx_http_geo_header_t *) base;
1462
1463 if (size < 16 || ngx_memcmp(&ngx_http_geo_header, header, 12) != 0) {
1464     ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
1465         "incompatible binary geo range base \"%s\"", name->data);
1466     goto failed;
1467 }
1468
1469 ngx_crc32_init(crc32);
1470
1471 vv = (ngx_http_variable_value_t *) (base + sizeof(ngx_http_geo_header_t));
1472
1473 while (vv->data) {
1474     len = ngx_align(sizeof(ngx_http_variable_value_t) + vv->len,
1475         sizeof(void *));
1476     ngx_crc32_update(&crc32, (u_char *) vv, len);
1477     vv->data += (size_t) base;
1478     vv = (ngx_http_variable_value_t *) ((u_char *) vv + len);
1479 }
1480 ngx_crc32_update(&crc32, (u_char *) vv, sizeof(ngx_http_variable_value_t));
1481 vv++;
1482
1483 ranges = (ngx_http_geo_range_t **) vv;
1484
1485 for (i = 0; i < 0x10000; i++) {
1486     ngx_crc32_update(&crc32, (u_char *) &ranges[i], sizeof(void *));
1487     if (ranges[i]) {
1488         ranges[i] = (ngx_http_geo_range_t *)
1489             ((u_char *) ranges[i] + (size_t) base);
1490     }
1491 }
1492
1493 range = (ngx_http_geo_range_t *) &ranges[0x10000];
1494
1495 while ((u_char *) range < base + size) {
1496     while (range->value) {
1497         ngx_crc32_update(&crc32, (u_char *) range,
1498             sizeof(ngx_http_geo_range_t));
1499         range->value = (ngx_http_variable_value_t *)
1500             ((u_char *) range->value + (size_t) base);
1501         range++;
1502     }
1503     ngx_crc32_update(&crc32, (u_char *) range, sizeof(void *));
1504     range = (ngx_http_geo_range_t *) ((u_char *) range + sizeof(void *));
1505 }
1506
1507 ngx_crc32_final(crc32);
1508
1509 if (crc32 != header->crc32) {
1510     ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
1511         "CRC32 mismatch in binary geo range base \"%s\"", name->data);
1512     goto failed;

```

```

1513     }
1514
1515     ngx\_conf\_log\_error(NGX_LOG_NOTICE, cf, 0,
1516         "using binary geo range base \"%s\"", name->data);
1517
1518     ctx->include_name = *name;
1519     ctx->binary_include = 1;
1520     ctx->high.low = ranges;
1521     rc = NGX\_OK;
1522
1523     goto done;
1524
1525 failed:
1526
1527     rc = NGX\_DECLINED;
1528
1529 done:
1530
1531     if (ngx\_close\_file(file.fd) == NGX\_FILE\_ERROR) {
1532         ngx\_log\_error(NGX_LOG_ALERT, cf->log, ngx\_errno,
1533             ngx\_close\_file\_n " \"%s\" failed", name->data);
1534     }
1535
1536     return rc;
1537 }
1538
1539
1540 static void
1541 ngx\_http\_geo\_create\_binary\_base(ngx\_http\_geo\_conf\_ctx\_t *ctx)
1542 {
1543     u_char                *p;
1544     uint32_t              hash;
1545     ngx\_str\_t             s;
1546     ngx\_uint\_t           i;
1547     ngx\_file\_mapping\_t   fm;
1548     ngx\_http\_geo\_range\_t *r, *range, **ranges;
1549     ngx\_http\_geo\_header\_t *header;
1550     ngx\_http\_geo\_variable\_value\_node\_t *gvvn;
1551
1552     fm.name = ngx\_pnalloc(ctx->temp_pool, ctx->include_name.len + 5);
1553     if (fm.name == NULL) {
1554         return;
1555     }
1556
1557     ngx\_sprintf(fm.name, "%V.bin%Z", &ctx->include_name);
1558
1559     fm.size = ctx->data_size;
1560     fm.log = ctx->pool->log;
1561
1562     ngx\_log\_error(NGX_LOG_NOTICE, fm.log, 0,
1563         "creating binary geo range base \"%s\"", fm.name);
1564
1565     if (ngx\_create\_file\_mapping(&fm) != NGX\_OK) {
1566         return;
1567     }
1568
1569     p = ngx\_cpymem(fm.addr, &ngx\_http\_geo\_header,
1570         sizeof(ngx\_http\_geo\_header\_t));
1571
1572     p = ngx\_http\_geo\_copy\_values(fm.addr, p, ctx->rbtree.root,
1573         ctx->rbtree.sentinel);
1574
1575     p += sizeof(ngx\_http\_variable\_value\_t);
1576
1577     ranges = (ngx\_http\_geo\_range\_t **) p;
1578
1579     p += 0x10000 * sizeof(ngx\_http\_geo\_range\_t *);
1580
1581     for (i = 0; i < 0x10000; i++) {
1582         r = ctx->high.low[i];
1583         if (r == NULL) {
1584             continue;
1585         }
1586
1587         range = (ngx\_http\_geo\_range\_t *) p;
1588         ranges[i] = (ngx\_http\_geo\_range\_t *) (p - (u_char *) fm.addr);

```

```

1589
1590     do {
1591         s.len = r->value->len;
1592         s.data = r->value->data;
1593         hash = ngx\_crc32\_long(s.data, s.len);
1594         gvvn = (ngx\_http\_geo\_variable\_value\_node\_t *)
1595             ngx\_str\_rbtrees\_lookup(&ctx->rbtree, &s, hash);
1596
1597         range->value = (ngx\_http\_variable\_value\_t *) gvvn->offset;
1598         range->start = r->start;
1599         range->end = r->end;
1600         range++;
1601
1602     } while ((++r)->value);
1603
1604     range->value = NULL;
1605
1606     p = (u_char *) range + sizeof\(void \*\);
1607 }
1608
1609 header = fm.addr;
1610 header->crc32 = ngx\_crc32\_long((u_char *) fm.addr
1611     + sizeof\(ngx\_http\_geo\_header\_t\),
1612     fm.size - sizeof\(ngx\_http\_geo\_header\_t\));
1613
1614 ngx\_close\_file\_mapping(&fm);
1615 }
1616
1617
1618 static u_char *
1619 ngx\_http\_geo\_copy\_values(u_char *base, u_char *p, ngx\_rbtrees\_node\_t *node,
1620     ngx\_rbtrees\_node\_t *sentinel)
1621 {
1622     ngx\_http\_variable\_value\_t *vv;
1623     ngx\_http\_geo\_variable\_value\_node\_t *gvvn;
1624
1625     if (node == sentinel) {
1626         return p;
1627     }
1628
1629     gvvn = (ngx\_http\_geo\_variable\_value\_node\_t *) node;
1630     gvvn->offset = p - base;
1631
1632     vv = (ngx\_http\_variable\_value\_t *) p;
1633     *vv = *gvvn->value;
1634     p += sizeof\(ngx\_http\_variable\_value\_t\);
1635     vv->data = (u_char *) (p - base);
1636
1637     p = ngx\_cpymem(p, gvvn->sn.str.data, gvvn->sn.str.len);
1638
1639     p = ngx\_align\_ptr(p, sizeof\(void \*\));
1640
1641     p = ngx\_http\_geo\_copy\_values(base, p, node->left, sentinel);
1642
1643     return ngx\_http\_geo\_copy\_values(base, p, node->right, sentinel);
1644 }

```

[One Level Up](#)

[Top Level](#)



# src/core/nginx\_radix\_tree.h - nginx-1.7.10

## Data types defined

- [ngx\\_radix\\_node\\_s](#)
- [ngx\\_radix\\_node\\_t](#)
- [ngx\\_radix\\_tree\\_t](#)

## Macros defined

- [NGX\\_RADIX\\_NO\\_VALUE](#)
- [\\_NGX\\_RADIX\\_TREE\\_H\\_INCLUDED](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_RADIX\_TREE\_H\_INCLUDED
9 #define \_NGX\_RADIX\_TREE\_H\_INCLUDED
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 #define NGX\_RADIX\_NO\_VALUE (uintptr_t) -1
17
18 typedef struct ngx\_radix\_node\_s ngx\_radix\_node\_t;
19
20 struct ngx\_radix\_node\_s {
21     ngx\_radix\_node\_t *right;
22     ngx\_radix\_node\_t *left;
23     ngx\_radix\_node\_t *parent;
24     uintptr_t value;
25 };
26
27
28 typedef struct {
29     ngx\_radix\_node\_t *root;
30     ngx\_pool\_t *pool;
31     ngx\_radix\_node\_t *free;
32     char *start;
33     size_t size;
34 } ngx\_radix\_tree\_t;
35
36
37 ngx\_radix\_tree\_t *ngx\_radix\_tree\_create(ngx\_pool\_t *pool,
38     ngx\_int\_t preallocate);
39
40 ngx\_int\_t ngx\_radix32tree\_insert(ngx\_radix\_tree\_t *tree,
41     uint32_t key, uint32_t mask, uintptr_t value);
42 ngx\_int\_t ngx\_radix32tree\_delete(ngx\_radix\_tree\_t *tree,
43     uint32_t key, uint32_t mask);
44 uintptr_t ngx\_radix32tree\_find(ngx\_radix\_tree\_t *tree, uint32_t key);
45
46 #if (NGX_HAVE_INET6)
47 ngx\_int\_t ngx\_radix128tree\_insert(ngx\_radix\_tree\_t *tree,
48     u_char *key, u_char *mask, uintptr_t value);
49 ngx\_int\_t ngx\_radix128tree\_delete(ngx\_radix\_tree\_t *tree,
50     u_char *key, u_char *mask);
```

```
51 uintptr_t ngx_radix128tree_find(ngx_radix_tree_t *tree, u_char *key);
52 #endif
53
54
55 #endif /* NGX_RADIX_TREE_H_INCLUDED */
```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_radix\_tree.c - nginx-1.7.10

### Functions defined

- [ngx\\_radix128tree\\_delete](#)
- [ngx\\_radix128tree\\_find](#)
- [ngx\\_radix128tree\\_insert](#)
- [ngx\\_radix32tree\\_delete](#)
- [ngx\\_radix32tree\\_find](#)
- [ngx\\_radix32tree\\_insert](#)
- [ngx\\_radix\\_alloc](#)
- [ngx\\_radix\\_tree\\_create](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10
11
12 static ngx_radix_node_t *ngx_radix_alloc(ngx_radix_tree_t *tree);
13
14
15 ngx_radix_tree_t *
16 ngx_radix_tree_create(ngx_pool_t *pool, ngx_int_t preallocate)
17 {
18     uint32_t      key, mask, inc;
19     ngx_radix_tree_t *tree;
20
21     tree = ngx_palloc(pool, sizeof(ngx_radix_tree_t));
22     if (tree == NULL) {
23         return NULL;
24     }
25
26     tree->pool = pool;
27     tree->free = NULL;
28     tree->start = NULL;
29     tree->size = 0;
30
31     tree->root = ngx_radix_alloc(tree);
32     if (tree->root == NULL) {
33         return NULL;
34     }
35
36     tree->root->right = NULL;
37     tree->root->left = NULL;
38     tree->root->parent = NULL;
39     tree->root->value = NGX_RADIX_NO_VALUE;
40
41     if (preallocate == 0) {
42         return tree;
43     }
44
45     /*
46      * Preallocation of first nodes : 0, 1, 00, 01, 10, 11, 000, 001, etc.
47      * increases TLB hits even if for first lookup iterations.
```

```

48  * On 32-bit platforms the 7 preallocated bits takes continuous 4K,
49  * 8 - 8K, 9 - 16K, etc. On 64-bit platforms the 6 preallocated bits
50  * takes continuous 4K, 7 - 8K, 8 - 16K, etc. There is no sense to
51  * to preallocate more than one page, because further preallocation
52  * distributes the only bit per page. Instead, a random insertion
53  * may distribute several bits per page.
54  *
55  * Thus, by default we preallocate maximum
56  *   6 bits on amd64 (64-bit platform and 4K pages)
57  *   7 bits on i386 (32-bit platform and 4K pages)
58  *   7 bits on sparc64 in 64-bit mode (8K pages)
59  *   8 bits on sparc64 in 32-bit mode (8K pages)
60  */
61
62  if (preallocate == -1) {
63      switch (ngx_pagesize / sizeof(ngx_radix_node_t)) {
64
65          /* amd64 */
66          case 128:
67              preallocate = 6;
68              break;
69
70          /* i386, sparc64 */
71          case 256:
72              preallocate = 7;
73              break;
74
75          /* sparc64 in 32-bit mode */
76          default:
77              preallocate = 8;
78      }
79  }
80
81  mask = 0;
82  inc = 0x80000000;
83
84  while (preallocate--) {
85
86      key = 0;
87      mask >>= 1;
88      mask |= 0x80000000;
89
90      do {
91          if (ngx_radix32tree_insert(tree, key, mask, NGX_RADIX_NO_VALUE)
92              != NGX_OK)
93              {
94                  return NULL;
95              }
96
97              key += inc;
98
99      } while (key);
100
101      inc >>= 1;
102  }
103
104  return tree;
105 }
106
107
108 ngx_int_t
109 ngx_radix32tree_insert(ngx_radix_tree_t *tree, uint32_t key, uint32_t mask,
110                      uintptr_t value)
111 {
112     uint32_t      bit;
113     ngx_radix_node_t *node, *next;
114
115     bit = 0x80000000;
116
117     node = tree->root;
118     next = tree->root;
119
120     while (bit & mask) {
121         if (key & bit) {
122             next = node->right;

```

```

124     } else {
125         next = node->left;
126     }
127
128     if (next == NULL) {
129         break;
130     }
131
132     bit >>= 1;
133     node = next;
134 }
135
136 if (next) {
137     if (node->value != NGX\_RADIX\_NO\_VALUE) {
138         return NGX\_BUSY;
139     }
140
141     node->value = value;
142     return NGX\_OK;
143 }
144
145 while (bit & mask) {
146     next = ngx\_radix\_alloc(tree);
147     if (next == NULL) {
148         return NGX\_ERROR;
149     }
150
151     next->right = NULL;
152     next->left = NULL;
153     next->parent = node;
154     next->value = NGX\_RADIX\_NO\_VALUE;
155
156     if (key & bit) {
157         node->right = next;
158
159     } else {
160         node->left = next;
161     }
162
163     bit >>= 1;
164     node = next;
165 }
166
167 node->value = value;
168
169 return NGX\_OK;
170 }
171
172
173 ngx\_int\_t
174 ngx\_radix32tree\_delete(ngx\_radix\_tree\_t *tree, uint32\_t key, uint32\_t mask)
175 {
176     uint32\_t         bit;
177     ngx\_radix\_node\_t *node;
178
179     bit = 0x80000000;
180     node = tree->root;
181
182     while (node && (bit & mask)) {
183         if (key & bit) {
184             node = node->right;
185
186         } else {
187             node = node->left;
188         }
189
190         bit >>= 1;
191     }
192
193     if (node == NULL) {
194         return NGX\_ERROR;
195     }
196
197     if (node->right || node->left) {
198         if (node->value != NGX\_RADIX\_NO\_VALUE) {
199             node->value = NGX\_RADIX\_NO\_VALUE;

```

```

200         return NGX\_OK;
201     }
202
203     return NGX\_ERROR;
204 }
205
206 for ( ;; ) {
207     if (node->parent->right == node) {
208         node->parent->right = NULL;
209
210     } else {
211         node->parent->left = NULL;
212     }
213
214     node->right = tree->free;
215     tree->free = node;
216
217     node = node->parent;
218
219     if (node->right || node->left) {
220         break;
221     }
222
223     if (node->value != NGX\_RADIX\_NO\_VALUE) {
224         break;
225     }
226
227     if (node->parent == NULL) {
228         break;
229     }
230 }
231
232 return NGX\_OK;
233 }
234
235
236 uintptr\_t
237 ngx\_radix32tree\_find(ngx\_radix\_tree\_t *tree, uint32\_t key)
238 {
239     uint32\_t         bit;
240     uintptr\_t        value;
241     ngx\_radix\_node\_t *node;
242
243     bit = 0x80000000;
244     value = NGX\_RADIX\_NO\_VALUE;
245     node = tree->root;
246
247     while (node) {
248         if (node->value != NGX\_RADIX\_NO\_VALUE) {
249             value = node->value;
250         }
251
252         if (key & bit) {
253             node = node->right;
254
255         } else {
256             node = node->left;
257         }
258
259         bit >>= 1;
260     }
261
262     return value;
263 }
264
265
266 #if (NGX\_HAVE\_INET6)
267
268 ngx\_int\_t
269 ngx\_radix128tree\_insert(ngx\_radix\_tree\_t *tree, u\_char *key, u\_char *mask,
270 uintptr\_t value)
271 {
272     u\_char           bit;
273     ngx\_uint\_t       i;
274     ngx\_radix\_node\_t *node, *next;
275

```

```

276     i = 0;
277     bit = 0x80;
278
279     node = tree->root;
280     next = tree->root;
281
282     while (bit & mask[i]) {
283         if (key[i] & bit) {
284             next = node->right;
285
286         } else {
287             next = node->left;
288         }
289
290         if (next == NULL) {
291             break;
292         }
293
294         bit >>= 1;
295         node = next;
296
297         if (bit == 0) {
298             if (++i == 16) {
299                 break;
300             }
301
302             bit = 0x80;
303         }
304     }
305
306     if (next) {
307         if (node->value != NGX\_RADIX\_NO\_VALUE) {
308             return NGX\_BUSY;
309         }
310
311         node->value = value;
312         return NGX\_OK;
313     }
314
315     while (bit & mask[i]) {
316         next = ngx\_radix\_alloc(tree);
317         if (next == NULL) {
318             return NGX\_ERROR;
319         }
320
321         next->right = NULL;
322         next->left = NULL;
323         next->parent = node;
324         next->value = NGX\_RADIX\_NO\_VALUE;
325
326         if (key[i] & bit) {
327             node->right = next;
328
329         } else {
330             node->left = next;
331         }
332
333         bit >>= 1;
334         node = next;
335
336         if (bit == 0) {
337             if (++i == 16) {
338                 break;
339             }
340
341             bit = 0x80;
342         }
343     }
344
345     node->value = value;
346
347     return NGX\_OK;
348 }
349
350
351 ngx\_int\_t

```

```

352 ngx_radix128tree_delete(ngx_radix_tree_t *tree, u_char *key, u_char *mask)
353 {
354     u_char          bit;
355     ngx_uint_t      i;
356     ngx_radix_node_t *node;
357
358     i = 0;
359     bit = 0x80;
360     node = tree->root;
361
362     while (node && (bit & mask[i])) {
363         if (key[i] & bit) {
364             node = node->right;
365
366         } else {
367             node = node->left;
368         }
369
370         bit >>= 1;
371
372         if (bit == 0) {
373             if (++i == 16) {
374                 break;
375             }
376
377             bit = 0x80;
378         }
379     }
380
381     if (node == NULL) {
382         return NGX_ERROR;
383     }
384
385     if (node->right || node->left) {
386         if (node->value != NGX_RADIX_NO_VALUE) {
387             node->value = NGX_RADIX_NO_VALUE;
388             return NGX_OK;
389         }
390
391         return NGX_ERROR;
392     }
393
394     for ( ;; ) {
395         if (node->parent->right == node) {
396             node->parent->right = NULL;
397
398         } else {
399             node->parent->left = NULL;
400         }
401
402         node->right = tree->free;
403         tree->free = node;
404
405         node = node->parent;
406
407         if (node->right || node->left) {
408             break;
409         }
410
411         if (node->value != NGX_RADIX_NO_VALUE) {
412             break;
413         }
414
415         if (node->parent == NULL) {
416             break;
417         }
418     }
419
420     return NGX_OK;
421 }
422
423
424 uintptr_t
425 ngx_radix128tree_find(ngx_radix_tree_t *tree, u_char *key)
426 {
427     u_char          bit;

```



```

428     uintptr_t     value;
429     ngx_uint_t   i;
430     ngx_radix_node_t *node;
431
432     i = 0;
433     bit = 0x80;
434     value = NGX_RADIX_NO_VALUE;
435     node = tree->root;
436
437     while (node) {
438         if (node->value != NGX_RADIX_NO_VALUE) {
439             value = node->value;
440         }
441
442         if (key[i] & bit) {
443             node = node->right;
444         } else {
445             node = node->left;
446         }
447     }
448
449     bit >>= 1;
450
451     if (bit == 0) {
452         i++;
453         bit = 0x80;
454     }
455 }
456
457     return value;
458 }
459
460 #endif
461
462
463 static ngx_radix_node_t *
464 ngx_radix_alloc(ngx_radix_tree_t *tree)
465 {
466     ngx_radix_node_t *p;
467
468     if (tree->free) {
469         p = tree->free;
470         tree->free = tree->free->right;
471         return p;
472     }
473
474     if (tree->size < sizeof(ngx_radix_node_t)) {
475         tree->start = ngx_pmemalign(tree->pool, ngx_pagesize, ngx_pagesize);
476         if (tree->start == NULL) {
477             return NULL;
478         }
479
480         tree->size = ngx_pagesize;
481     }
482
483     p = (ngx_radix_node_t *) tree->start;
484     tree->start += sizeof(ngx_radix_node_t);
485     tree->size -= sizeof(ngx_radix_node_t);
486
487     return p;
488 }

```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_geoiip\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_geoiip\\_commands](#)
- [ngx\\_http\\_geoiip\\_country\\_functions](#)
- [ngx\\_http\\_geoiip\\_country\\_v6\\_functions](#)
- [ngx\\_http\\_geoiip\\_module](#)
- [ngx\\_http\\_geoiip\\_module\\_ctx](#)
- [ngx\\_http\\_geoiip\\_vars](#)

## Data types defined

- [ngx\\_http\\_geoiip\\_conf\\_t](#)
- [ngx\\_http\\_geoiip\\_var\\_t](#)
- [ngx\\_http\\_geoiip\\_variable\\_handler\\_pt](#)
- [ngx\\_http\\_geoiip\\_variable\\_handler\\_v6\\_pt](#)

## Functions defined

- [ngx\\_http\\_geoiip\\_add\\_variables](#)
- [ngx\\_http\\_geoiip\\_addr](#)
- [ngx\\_http\\_geoiip\\_addr\\_v6](#)
- [ngx\\_http\\_geoiip\\_cidr\\_value](#)
- [ngx\\_http\\_geoiip\\_city](#)
- [ngx\\_http\\_geoiip\\_city\\_float\\_variable](#)
- [ngx\\_http\\_geoiip\\_city\\_int\\_variable](#)
- [ngx\\_http\\_geoiip\\_city\\_variable](#)
- [ngx\\_http\\_geoiip\\_cleanup](#)
- [ngx\\_http\\_geoiip\\_country](#)
- [ngx\\_http\\_geoiip\\_country\\_variable](#)
- [ngx\\_http\\_geoiip\\_create\\_conf](#)
- [ngx\\_http\\_geoiip\\_get\\_city\\_record](#)
- [ngx\\_http\\_geoiip\\_init\\_conf](#)
- [ngx\\_http\\_geoiip\\_org](#)
- [ngx\\_http\\_geoiip\\_org\\_variable](#)
- [ngx\\_http\\_geoiip\\_proxy](#)

- [ngx\\_http\\_geoip\\_region\\_name\\_variable](#)

## Macros defined

- [NGX\\_GEOIP\\_COUNTRY\\_CODE](#)
- [NGX\\_GEOIP\\_COUNTRY\\_CODE3](#)
- [NGX\\_GEOIP\\_COUNTRY\\_NAME](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12 #include <GeoIP.h>
13 #include <GeoIPCity.h>
14
15
16 #define NGX_GEOIP_COUNTRY_CODE 0
17 #define NGX_GEOIP_COUNTRY_CODE3 1
18 #define NGX_GEOIP_COUNTRY_NAME 2
19
20
21 typedef struct {
22     GeoIP      *country;
23     GeoIP      *org;
24     GeoIP      *city;
25     ngx_array_t *proxies; /* array of ngx_cidr_t */
26     ngx_flag_t proxy_recursive;
27 #if (NGX_HAVE_GEOIP_V6)
28     unsigned   country_v6:1;
29     unsigned   org_v6:1;
30     unsigned   city_v6:1;
31 #endif
32 } ngx_http_geoip_conf_t;
33
34
35 typedef struct {
36     ngx_str_t  *name;
37     uintptr_t  data;
38 } ngx_http_geoip_var_t;
39
40
41 typedef const char *(*ngx_http_geoip_variable_handler_pt)(GeoIP *,
42     u_long addr);
43
44
45 ngx_http_geoip_variable_handler_pt ngx_http_geoip_country_functions[] = {
46     GeoIP_country_code_by_ipnum,
47     GeoIP_country_code3_by_ipnum,
48     GeoIP_country_name_by_ipnum,
49 };
50
51
52 #if (NGX_HAVE_GEOIP_V6)
53
54 typedef const char *(*ngx_http_geoip_variable_handler_v6_pt)(GeoIP *,
55     geoipv6_t addr);
56
57
58 ngx_http_geoip_variable_handler_v6_pt ngx_http_geoip_country_v6_functions[] = {
59     GeoIP_country_code_by_ipnum_v6,
60     GeoIP_country_code3_by_ipnum_v6,
```

```

61     GeoIP_country_name_by_ipnum_v6,
62 };
63
64 #endif
65
66
67 static ngx_int_t ngx_http_geoip_country_variable(ngx_http_request_t *r,
68     ngx_http_variable_value_t *v, uintptr_t data);
69 static ngx_int_t ngx_http_geoip_org_variable(ngx_http_request_t *r,
70     ngx_http_variable_value_t *v, uintptr_t data);
71 static ngx_int_t ngx_http_geoip_city_variable(ngx_http_request_t *r,
72     ngx_http_variable_value_t *v, uintptr_t data);
73 static ngx_int_t ngx_http_geoip_region_name_variable(ngx_http_request_t *r,
74     ngx_http_variable_value_t *v, uintptr_t data);
75 static ngx_int_t ngx_http_geoip_city_float_variable(ngx_http_request_t *r,
76     ngx_http_variable_value_t *v, uintptr_t data);
77 static ngx_int_t ngx_http_geoip_city_int_variable(ngx_http_request_t *r,
78     ngx_http_variable_value_t *v, uintptr_t data);
79 static GeoIPRecord *ngx_http_geoip_get_city_record(ngx_http_request_t *r);
80
81 static ngx_int_t ngx_http_geoip_add_variables(ngx_conf_t *cf);
82 static void *ngx_http_geoip_create_conf(ngx_conf_t *cf);
83 static char *ngx_http_geoip_init_conf(ngx_conf_t *cf, void *conf);
84 static char *ngx_http_geoip_country(ngx_conf_t *cf, ngx_command_t *cmd,
85     void *conf);
86 static char *ngx_http_geoip_org(ngx_conf_t *cf, ngx_command_t *cmd,
87     void *conf);
88 static char *ngx_http_geoip_city(ngx_conf_t *cf, ngx_command_t *cmd,
89     void *conf);
90 static char *ngx_http_geoip_proxy(ngx_conf_t *cf, ngx_command_t *cmd,
91     void *conf);
92 static ngx_int_t ngx_http_geoip_cidr_value(ngx_conf_t *cf, ngx_str_t *net,
93     ngx_cidr_t *cidr);
94 static void ngx_http_geoip_cleanup(void *data);
95
96
97 static ngx_command_t  ngx_http_geoip_commands[] = {
98
99     { ngx_string("geoip_country"),
100     NGX_HTTP_MAIN_CONF|NGX_CONF_TAKE12,
101     ngx_http_geoip_country,
102     NGX_HTTP_MAIN_CONF_OFFSET,
103     0,
104     NULL },
105
106     { ngx_string("geoip_org"),
107     NGX_HTTP_MAIN_CONF|NGX_CONF_TAKE12,
108     ngx_http_geoip_org,
109     NGX_HTTP_MAIN_CONF_OFFSET,
110     0,
111     NULL },
112
113     { ngx_string("geoip_city"),
114     NGX_HTTP_MAIN_CONF|NGX_CONF_TAKE12,
115     ngx_http_geoip_city,
116     NGX_HTTP_MAIN_CONF_OFFSET,
117     0,
118     NULL },
119
120     { ngx_string("geoip_proxy"),
121     NGX_HTTP_MAIN_CONF|NGX_CONF_TAKE1,
122     ngx_http_geoip_proxy,
123     NGX_HTTP_MAIN_CONF_OFFSET,
124     0,
125     NULL },
126
127     { ngx_string("geoip_proxy_recursive"),
128     NGX_HTTP_MAIN_CONF|NGX_CONF_FLAG,
129     ngx_conf_set_flag_slot,
130     NGX_HTTP_MAIN_CONF_OFFSET,
131     offsetof(ngx_http_geoip_conf_t, proxy_recursive),
132     NULL },
133
134     ngx_null_command
135 };
136

```

```

137
138 static ngx_http_module_t ngx_http_geoip_module_ctx = {
139     ngx_http_geoip_add_variables,      /* preconfiguration */
140     NULL,                               /* postconfiguration */
141
142     ngx_http_geoip_create_conf,        /* create main configuration */
143     ngx_http_geoip_init_conf,         /* init main configuration */
144
145     NULL,                               /* create server configuration */
146     NULL,                               /* merge server configuration */
147
148     NULL,                               /* create location configuration */
149     NULL,                               /* merge location configuration */
150 };
151
152
153 ngx_module_t ngx_http_geoip_module = {
154     NGX_MODULE_V1,
155     &ngx_http_geoip_module_ctx,        /* module context */
156     ngx_http_geoip_commands,          /* module directives */
157     NGX_HTTP_MODULE,                  /* module type */
158     NULL,                              /* init master */
159     NULL,                              /* init module */
160     NULL,                              /* init process */
161     NULL,                              /* init thread */
162     NULL,                              /* exit thread */
163     NULL,                              /* exit process */
164     NULL,                              /* exit master */
165     NGX_MODULE_V1_PADDING
166 };
167
168
169 static ngx_http_variable_t ngx_http_geoip_vars[] = {
170
171     { ngx_string("geoip_country_code"), NULL,
172       ngx_http_geoip_country_variable,
173       NGX_GEOIP_COUNTRY_CODE, 0, 0 },
174
175     { ngx_string("geoip_country_code3"), NULL,
176       ngx_http_geoip_country_variable,
177       NGX_GEOIP_COUNTRY_CODE3, 0, 0 },
178
179     { ngx_string("geoip_country_name"), NULL,
180       ngx_http_geoip_country_variable,
181       NGX_GEOIP_COUNTRY_NAME, 0, 0 },
182
183     { ngx_string("geoip_org"), NULL,
184       ngx_http_geoip_org_variable,
185       0, 0, 0 },
186
187     { ngx_string("geoip_city_continent_code"), NULL,
188       ngx_http_geoip_city_variable,
189       offsetof(GeoIPRecord, continent_code), 0, 0 },
190
191     { ngx_string("geoip_city_country_code"), NULL,
192       ngx_http_geoip_city_variable,
193       offsetof(GeoIPRecord, country_code), 0, 0 },
194
195     { ngx_string("geoip_city_country_code3"), NULL,
196       ngx_http_geoip_city_variable,
197       offsetof(GeoIPRecord, country_code3), 0, 0 },
198
199     { ngx_string("geoip_city_country_name"), NULL,
200       ngx_http_geoip_city_variable,
201       offsetof(GeoIPRecord, country_name), 0, 0 },
202
203     { ngx_string("geoip_region"), NULL,
204       ngx_http_geoip_city_variable,
205       offsetof(GeoIPRecord, region), 0, 0 },
206
207     { ngx_string("geoip_region_name"), NULL,
208       ngx_http_geoip_region_name_variable,
209       0, 0, 0 },
210
211     { ngx_string("geoip_city"), NULL,
212       ngx_http_geoip_city_variable,

```

```

213     offsetof(GeoIPRecord, city), 0, 0 },
214
215     { ngx_string("geoip_postal_code"), NULL,
216       ngx_http_geoip_city_variable,
217       offsetof(GeoIPRecord, postal_code), 0, 0 },
218
219     { ngx_string("geoip_latitude"), NULL,
220       ngx_http_geoip_city_float_variable,
221       offsetof(GeoIPRecord, latitude), 0, 0 },
222
223     { ngx_string("geoip_longitude"), NULL,
224       ngx_http_geoip_city_float_variable,
225       offsetof(GeoIPRecord, longitude), 0, 0 },
226
227     { ngx_string("geoip_dma_code"), NULL,
228       ngx_http_geoip_city_int_variable,
229       offsetof(GeoIPRecord, dma_code), 0, 0 },
230
231     { ngx_string("geoip_area_code"), NULL,
232       ngx_http_geoip_city_int_variable,
233       offsetof(GeoIPRecord, area_code), 0, 0 },
234
235     { ngx_null_string, NULL, NULL, 0, 0, 0 }
236 };
237
238
239 static u_long
240 ngx_http_geoip_addr(ngx_http_request_t *r, ngx_http_geoip_conf_t *gcf)
241 {
242     ngx_addr_t      addr;
243     ngx_array_t     *xfwd;
244     struct sockaddr_in *sin;
245
246     addr.sockaddr = r->connection->sockaddr;
247     addr.socklen = r->connection->socklen;
248     /* addr.name = r->connection->addr_text; */
249
250     xfwd = &r->headers_in.x_forwarded_for;
251
252     if (xfwd->nelts > 0 && gcf->proxies != NULL) {
253         (void) ngx_http_get_forwarded_addr(r, &addr, xfwd, NULL,
254                                           gcf->proxies, gcf->proxy_recursive);
255     }
256
257     #if (NGX_HAVE_INET6)
258
259     if (addr.sockaddr->sa_family == AF_INET6) {
260         u_char      *p;
261         in_addr_t    inaddr;
262         struct in6_addr *inaddr6;
263
264         inaddr6 = &((struct sockaddr_in6 *) addr.sockaddr)->sin6_addr;
265
266         if (IN6_IS_ADDR_V4MAPPED(inaddr6)) {
267             p = inaddr6->s6_addr;
268
269             inaddr = p[12] << 24;
270             inaddr += p[13] << 16;
271             inaddr += p[14] << 8;
272             inaddr += p[15];
273
274             return inaddr;
275         }
276     }
277
278     #endif
279
280     if (addr.sockaddr->sa_family != AF_INET) {
281         return INADDR_NONE;
282     }
283
284     sin = (struct sockaddr_in *) addr.sockaddr;
285     return ntohl(sin->sin_addr.s_addr);
286 }
287
288

```

```

289 #if (NGX_HAVE_GEOIP_V6)
290
291 static geoipv6_t
292 ngx_http_geoip_addr_v6(ngx_http_request_t *r, ngx_http_geoip_conf_t *gcf)
293 {
294     ngx_addr_t      addr;
295     ngx_array_t     *xfwd;
296     in_addr_t       addr4;
297     struct in6_addr  addr6;
298     struct sockaddr_in *sin;
299     struct sockaddr_in6 *sin6;
300
301     addr.sockaddr = r->connection->sockaddr;
302     addr.socklen = r->connection->socklen;
303     /* addr.name = r->connection->addr_text; */
304
305     xfwd = &r->headers_in.x_forwarded_for;
306
307     if (xfwd->nelts > 0 && gcf->proxies != NULL) {
308         (void) ngx_http_get_forwarded_addr(r, &addr, xfwd, NULL,
309             gcf->proxies, gcf->proxy_recursive);
310     }
311
312     switch (addr.sockaddr->sa_family) {
313
314     case AF_INET:
315         /* Produce IPv4-mapped IPv6 address. */
316         sin = (struct sockaddr_in *) addr.sockaddr;
317         addr4 = ntohl(sin->sin_addr.s_addr);
318
319         ngx_memzero(&addr6, sizeof(struct in6_addr));
320         addr6.s6_addr[10] = 0xff;
321         addr6.s6_addr[11] = 0xff;
322         addr6.s6_addr[12] = addr4 >> 24;
323         addr6.s6_addr[13] = addr4 >> 16;
324         addr6.s6_addr[14] = addr4 >> 8;
325         addr6.s6_addr[15] = addr4;
326         return addr6;
327
328     case AF_INET6:
329         sin6 = (struct sockaddr_in6 *) addr.sockaddr;
330         return sin6->sin6_addr;
331
332     default:
333         return in6addr_any;
334     }
335 }
336
337 #endif
338
339
340 static ngx_int_t
341 ngx_http_geoip_country_variable(ngx_http_request_t *r,
342     ngx_http_variable_value_t *v, uintptr_t data)
343 {
344     ngx_http_geoip_variable_handler_pt handler =
345         ngx_http_geoip_country_functions[data];
346     #if (NGX_HAVE_GEOIP_V6)
347     ngx_http_geoip_variable_handler_v6_pt handler_v6 =
348         ngx_http_geoip_country_v6_functions[data];
349     #endif
350
351     const char      *val;
352     ngx_http_geoip_conf_t *gcf;
353
354     gcf = ngx_http_get_module_main_conf(r, ngx_http_geoip_module);
355
356     if (gcf->country == NULL) {
357         goto not_found;
358     }
359
360     #if (NGX_HAVE_GEOIP_V6)
361     val = gcf->country_v6
362         ? handler_v6(gcf->country, ngx_http_geoip_addr_v6(r, gcf))
363         : handler(gcf->country, ngx_http_geoip_addr(r, gcf));
364     #else

```

```

365     val = handler(gcf->country, ngx_http_geoip_addr(r, gcf));
366 #endif
367
368     if (val == NULL) {
369         goto not_found;
370     }
371
372     v->len = ngx_strlen(val);
373     v->valid = 1;
374     v->no_cacheable = 0;
375     v->not_found = 0;
376     v->data = (u_char *) val;
377
378     return NGX_OK;
379
380 not_found:
381
382     v->not_found = 1;
383
384     return NGX_OK;
385 }
386
387
388 static ngx_int_t
389 ngx_http_geoip_org_variable(ngx_http_request_t *r,
390     ngx_http_variable_value_t *v, uintptr_t data)
391 {
392     size_t      len;
393     char        *val;
394     ngx_http_geoip_conf_t *gcf;
395
396     gcf = ngx_http_get_module_main_conf(r, ngx_http_geoip_module);
397
398     if (gcf->org == NULL) {
399         goto not_found;
400     }
401
402 #if (NGX_HAVE_GEOIP_V6)
403     val = gcf->org_v6
404         ? GeoIP_name_by_ipnum_v6(gcf->org,
405             ngx_http_geoip_addr_v6(r, gcf))
406         : GeoIP_name_by_ipnum(gcf->org,
407             ngx_http_geoip_addr(r, gcf));
408 #else
409     val = GeoIP_name_by_ipnum(gcf->org, ngx_http_geoip_addr(r, gcf));
410 #endif
411
412     if (val == NULL) {
413         goto not_found;
414     }
415
416     len = ngx_strlen(val);
417     v->data = ngx_pnalloc(r->pool, len);
418     if (v->data == NULL) {
419         ngx_free(val);
420         return NGX_ERROR;
421     }
422
423     ngx_memcpy(v->data, val, len);
424
425     v->len = len;
426     v->valid = 1;
427     v->no_cacheable = 0;
428     v->not_found = 0;
429
430     ngx_free(val);
431
432     return NGX_OK;
433
434 not_found:
435
436     v->not_found = 1;
437
438     return NGX_OK;
439 }
440

```



```

441 static ngx_int_t
442 ngx_http_geoip_city_variable(ngx_http_request_t *r,
443     ngx_http_variable_value_t *v, uintptr_t data)
444 {
445     char        *val;
446     size_t      len;
447     GeoIPRecord *gr;
448
449     gr = ngx_http_geoip_get_city_record(r);
450     if (gr == NULL) {
451         goto not_found;
452     }
453
454     val = *(char **) ((char *) gr + data);
455     if (val == NULL) {
456         goto no_value;
457     }
458
459     len = ngx_strlen(val);
460     v->data = ngx_pnalloc(r->pool, len);
461     if (v->data == NULL) {
462         GeoIPRecord_delete(gr);
463         return NGX_ERROR;
464     }
465
466     ngx_memcpy(v->data, val, len);
467
468     v->len = len;
469     v->valid = 1;
470     v->no_cacheable = 0;
471     v->not_found = 0;
472
473     GeoIPRecord_delete(gr);
474
475     return NGX_OK;
476
477 no_value:
478     GeoIPRecord_delete(gr);
479
480 not_found:
481     v->not_found = 1;
482
483     return NGX_OK;
484 }
485
486 static ngx_int_t
487 ngx_http_geoip_region_name_variable(ngx_http_request_t *r,
488     ngx_http_variable_value_t *v, uintptr_t data)
489 {
490     size_t      len;
491     const char  *val;
492     GeoIPRecord *gr;
493
494     gr = ngx_http_geoip_get_city_record(r);
495     if (gr == NULL) {
496         goto not_found;
497     }
498
499     val = GeoIP_region_name_by_code(gr->country_code, gr->region);
500
501     GeoIPRecord_delete(gr);
502
503     if (val == NULL) {
504         goto not_found;
505     }
506
507     len = ngx_strlen(val);
508     v->data = ngx_pnalloc(r->pool, len);
509     if (v->data == NULL) {
510         return NGX_ERROR;
511     }
512
513     return NGX_OK;
514 }
515
516

```

```

517     ngx_memcpy(v->data, val, len);
518
519     v->len = len;
520     v->valid = 1;
521     v->no_cacheable = 0;
522     v->not_found = 0;
523
524     return NGX_OK;
525
526 not_found:
527
528     v->not_found = 1;
529
530     return NGX_OK;
531 }
532
533
534 static ngx_int_t
535 ngx_http_geopip_city_float_variable(ngx_http_request_t *r,
536     ngx_http_variable_value_t *v, uintptr_t data)
537 {
538     float        val;
539     GeoIPRecord  *gr;
540
541     gr = ngx_http_geopip_get_city_record(r);
542     if (gr == NULL) {
543         v->not_found = 1;
544         return NGX_OK;
545     }
546
547     v->data = ngx_pnalloc(r->pool, NGX_INT64_LEN + 5);
548     if (v->data == NULL) {
549         GeoIPRecord_delete(gr);
550         return NGX_ERROR;
551     }
552
553     val = *(float *) ((char *) gr + data);
554
555     v->len = ngx_sprintf(v->data, "%.4f", val) - v->data;
556     v->valid = 1;
557     v->no_cacheable = 0;
558     v->not_found = 0;
559
560     GeoIPRecord_delete(gr);
561
562     return NGX_OK;
563 }
564
565
566 static ngx_int_t
567 ngx_http_geopip_city_int_variable(ngx_http_request_t *r,
568     ngx_http_variable_value_t *v, uintptr_t data)
569 {
570     int        val;
571     GeoIPRecord  *gr;
572
573     gr = ngx_http_geopip_get_city_record(r);
574     if (gr == NULL) {
575         v->not_found = 1;
576         return NGX_OK;
577     }
578
579     v->data = ngx_pnalloc(r->pool, NGX_INT64_LEN);
580     if (v->data == NULL) {
581         GeoIPRecord_delete(gr);
582         return NGX_ERROR;
583     }
584
585     val = *(int *) ((char *) gr + data);
586
587     v->len = ngx_sprintf(v->data, "%d", val) - v->data;
588     v->valid = 1;
589     v->no_cacheable = 0;
590     v->not_found = 0;
591
592     GeoIPRecord_delete(gr);

```

```

593     return NGX\_OK;
594 }
595
596
597
598 static GeoIPRecord *
599 ngx\_http\_geoip\_get\_city\_record(ngx\_http\_request\_t *r)
600 {
601     ngx\_http\_geoip\_conf\_t *gcf;
602
603     gcf = ngx\_http\_get\_module\_main\_conf(r, ngx\_http\_geoip\_module);
604
605     if (gcf->city) {
606 #if (NGX\_HAVE\_GEOIP\_V6)
607         return gcf->city_v6
608             ? GeoIP\_record\_by\_ipnum\_v6(gcf->city,
609                 ngx\_http\_geoip\_addr\_v6(r, gcf))
610             : GeoIP\_record\_by\_ipnum(gcf->city,
611                 ngx\_http\_geoip\_addr(r, gcf));
612 #else
613         return GeoIP\_record\_by\_ipnum(gcf->city, ngx\_http\_geoip\_addr(r, gcf));
614 #endif
615     }
616
617     return NULL;
618 }
619
620
621 static ngx\_int\_t
622 ngx\_http\_geoip\_add\_variables(ngx\_conf\_t *cf)
623 {
624     ngx\_http\_variable\_t *var, *v;
625
626     for (v = ngx\_http\_geoip\_vars; v->name.len; v++) {
627         var = ngx\_http\_add\_variable(cf, &v->name, v->flags);
628         if (var == NULL) {
629             return NGX\_ERROR;
630         }
631
632         var->get_handler = v->get_handler;
633         var->data = v->data;
634     }
635
636     return NGX\_OK;
637 }
638
639
640 static void *
641 ngx\_http\_geoip\_create\_conf(ngx\_conf\_t *cf)
642 {
643     ngx\_pool\_cleanup\_t *cfn;
644     ngx\_http\_geoip\_conf\_t *conf;
645
646     conf = ngx\_palloc(cf->pool, sizeof(ngx\_http\_geoip\_conf\_t));
647     if (conf == NULL) {
648         return NULL;
649     }
650
651     conf->proxy_recursive = NGX\_CONF\_UNSET;
652
653     cfn = ngx\_pool\_cleanup\_add(cf->pool, 0);
654     if (cfn == NULL) {
655         return NULL;
656     }
657
658     cfn->handler = ngx\_http\_geoip\_cleanup;
659     cfn->data = conf;
660
661     return conf;
662 }
663
664
665 static char *
666 ngx\_http\_geoip\_init\_conf(ngx\_conf\_t *cf, void *conf)
667 {
668     ngx\_http\_geoip\_conf\_t *gcf = conf;

```

```

669     ngx_conf_init_value(gcf->proxy_recursive, 0);
670 }
671 return NGX_CONF_OK;
672 }
673
674
675
676 static char *
677 ngx_http_geoip_country(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
678 {
679     ngx_http_geoip_conf_t *gcf = conf;
680
681     ngx_str_t *value;
682
683     if (gcf->country) {
684         return "is duplicate";
685     }
686
687     value = cf->args->elts;
688
689     gcf->country = GeoIP_open((char *) value[1].data, GEOIP_MEMORY_CACHE);
690
691     if (gcf->country == NULL) {
692         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
693             "GeoIP_open(\"%V\") failed", &value[1]);
694
695         return NGX_CONF_ERROR;
696     }
697
698     if (cf->args->nelts == 3) {
699         if (ngx_strcmp(value[2].data, "utf8") == 0) {
700             GeoIP_set_charset(gcf->country, GEOIP_CHARSET_UTF8);
701
702         } else {
703             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
704                 "invalid parameter \"%V\"", &value[2]);
705             return NGX_CONF_ERROR;
706         }
707     }
708
709     switch (gcf->country->databaseType) {
710
711     case GEOIP_COUNTRY_EDITION:
712
713         return NGX_CONF_OK;
714
715     #if (NGX_HAVE_GEOIP_V6)
716     case GEOIP_COUNTRY_EDITION_V6:
717
718         gcf->country_v6 = 1;
719         return NGX_CONF_OK;
720     #endif
721
722     default:
723         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
724             "invalid GeoIP database \"%V\" type:%d",
725             &value[1], gcf->country->databaseType);
726         return NGX_CONF_ERROR;
727     }
728 }
729
730
731 static char *
732 ngx_http_geoip_org(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
733 {
734     ngx_http_geoip_conf_t *gcf = conf;
735
736     ngx_str_t *value;
737
738     if (gcf->org) {
739         return "is duplicate";
740     }
741
742     value = cf->args->elts;
743
744     gcf->org = GeoIP_open((char *) value[1].data, GEOIP_MEMORY_CACHE);

```

```

745
746 if (gcf->org == NULL) {
747     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
748         "GeoIP_open(\"%V\") failed", &value[1]);
749
750     return NGX_CONF_ERROR;
751 }
752
753 if (cf->args->nelts == 3) {
754     if (ngx_strcmp(value[2].data, "utf8") == 0) {
755         GeoIP_set_charset(gcf->org, GEOIP_CHARSET_UTF8);
756
757     } else {
758         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
759             "invalid parameter \"%V\"", &value[2]);
760         return NGX_CONF_ERROR;
761     }
762 }
763
764 switch (gcf->org->databaseType) {
765
766     case GEOIP_ISP_EDITION:
767     case GEOIP_ORG_EDITION:
768     case GEOIP_DOMAIN_EDITION:
769     case GEOIP_ASNUM_EDITION:
770
771         return NGX_CONF_OK;
772
773 #if (NGX_HAVE_GEOIP_V6)
774     case GEOIP_ISP_EDITION_V6:
775     case GEOIP_ORG_EDITION_V6:
776     case GEOIP_DOMAIN_EDITION_V6:
777     case GEOIP_ASNUM_EDITION_V6:
778
779         gcf->org_v6 = 1;
780         return NGX_CONF_OK;
781 #endif
782
783     default:
784         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
785             "invalid GeoIP database \"%V\" type:%d",
786             &value[1], gcf->org->databaseType);
787         return NGX_CONF_ERROR;
788 }
789 }
790
791
792 static char *
793 ngx_http_geoip_city(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
794 {
795     ngx_http_geoip_conf_t *gcf = conf;
796
797     ngx_str_t *value;
798
799     if (gcf->city) {
800         return "is duplicate";
801     }
802
803     value = cf->args->elts;
804
805     gcf->city = GeoIP_open((char *) value[1].data, GEOIP_MEMORY_CACHE);
806
807     if (gcf->city == NULL) {
808         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
809             "GeoIP_open(\"%V\") failed", &value[1]);
810
811         return NGX_CONF_ERROR;
812     }
813
814     if (cf->args->nelts == 3) {
815         if (ngx_strcmp(value[2].data, "utf8") == 0) {
816             GeoIP_set_charset(gcf->city, GEOIP_CHARSET_UTF8);
817
818         } else {
819             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
820                 "invalid parameter \"%V\"", &value[2]);

```

```

821         return NGX\_CONF\_ERROR;
822     }
823 }
824
825 switch (gcf->city->databaseType) {
826
827     case GEOIP_CITY_EDITION_REV0:
828     case GEOIP_CITY_EDITION_REV1:
829
830         return NGX\_CONF\_OK;
831
832     #if (NGX_HAVE_GEOIP_V6)
833     case GEOIP_CITY_EDITION_REV0_V6:
834     case GEOIP_CITY_EDITION_REV1_V6:
835
836         gcf->city_v6 = 1;
837         return NGX\_CONF\_OK;
838     #endif
839
840     default:
841         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
842             "invalid GeoIP City database \"%V\" type:%d",
843             &value[1], gcf->city->databaseType);
844         return NGX\_CONF\_ERROR;
845 }
846 }
847
848
849 static char *
850 ngx\_http\_geoip\_proxy(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
851 {
852     ngx\_http\_geoip\_conf\_t *gcf = conf;
853
854     ngx\_str\_t *value;
855     ngx\_cidr\_t cidr, *c;
856
857     value = cf->args->elts;
858
859     if (ngx\_http\_geoip\_cidr\_value(cf, &value[1], &cidr) != NGX\_OK) {
860         return NGX\_CONF\_ERROR;
861     }
862
863     if (gcf->proxies == NULL) {
864         gcf->proxies = ngx\_array\_create(cf->pool, 4, sizeof(ngx\_cidr\_t));
865         if (gcf->proxies == NULL) {
866             return NGX\_CONF\_ERROR;
867         }
868     }
869
870     c = ngx\_array\_push(gcf->proxies);
871     if (c == NULL) {
872         return NGX\_CONF\_ERROR;
873     }
874
875     *c = cidr;
876
877     return NGX\_CONF\_OK;
878 }
879
880 static ngx\_int\_t
881 ngx\_http\_geoip\_cidr\_value(ngx\_conf\_t *cf, ngx\_str\_t *net, ngx\_cidr\_t *cidr)
882 {
883     ngx\_int\_t rc;
884
885     if (ngx\_strcmp(net->data, "255.255.255.255") == 0) {
886         cidr->family = AF_INET;
887         cidr->u.in.addr = 0xffffffff;
888         cidr->u.in.mask = 0xffffffff;
889
890         return NGX\_OK;
891     }
892
893     rc = ngx\_ptocidr(net, cidr);
894
895     if (rc == NGX\_ERROR) {
896         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0, "invalid network \"%V\"", net);

```

```
897     return NGX\_ERROR;
898 }
899
900 if (rc == NGX\_DONE) {
901     ngx\_conf\_log\_error(NGX\_LOG\_WARN, cf, 0,
902         "low address bits of %V are meaningless", net);
903 }
904
905 return NGX\_OK;
906 }
907
908
909 static void
910 ngx\_http\_geoip\_cleanup(void *data)
911 {
912     ngx\_http\_geoip\_conf\_t *gcf = data;
913
914     if (gcf->country) {
915         GeoIP_delete(gcf->country);
916     }
917
918     if (gcf->org) {
919         GeoIP_delete(gcf->org);
920     }
921
922     if (gcf->city) {
923         GeoIP_delete(gcf->city);
924     }
925 }
```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_gunzip\_filter\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_gunzip\\_filter\\_commands](#)
- [ngx\\_http\\_gunzip\\_filter\\_module](#)
- [ngx\\_http\\_gunzip\\_filter\\_module\\_ctx](#)
- [ngx\\_http\\_next\\_body\\_filter](#)
- [ngx\\_http\\_next\\_header\\_filter](#)

## Data types defined

- [ngx\\_http\\_gunzip\\_conf\\_t](#)
- [ngx\\_http\\_gunzip\\_ctx\\_t](#)

## Functions defined

- [ngx\\_http\\_gunzip\\_body\\_filter](#)
- [ngx\\_http\\_gunzip\\_create\\_conf](#)
- [ngx\\_http\\_gunzip\\_filter\\_add\\_data](#)
- [ngx\\_http\\_gunzip\\_filter\\_alloc](#)
- [ngx\\_http\\_gunzip\\_filter\\_free](#)
- [ngx\\_http\\_gunzip\\_filter\\_get\\_buf](#)
- [ngx\\_http\\_gunzip\\_filter\\_inflate](#)
- [ngx\\_http\\_gunzip\\_filter\\_inflate\\_end](#)
- [ngx\\_http\\_gunzip\\_filter\\_inflate\\_start](#)
- [ngx\\_http\\_gunzip\\_filter\\_init](#)
- [ngx\\_http\\_gunzip\\_header\\_filter](#)
- [ngx\\_http\\_gunzip\\_merge\\_conf](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Maxim Dounin
5  * Copyright (C) Nginx, Inc.
6  */
7
8
9 #include <ngx_config.h>
10 #include <ngx_core.h>
11 #include <ngx_http.h>
12
13 #include <zlib.h>
14
```



```

15
16 typedef struct {
17     ngx_flag_t      enable;
18     ngx_bufs_t      bufs;
19 } ngx_http_gunzip_conf_t;
20
21
22 typedef struct {
23     ngx_chain_t      *in;
24     ngx_chain_t      *free;
25     ngx_chain_t      *busy;
26     ngx_chain_t      *out;
27     ngx_chain_t      **last_out;
28
29     ngx_buf_t        *in_buf;
30     ngx_buf_t        *out_buf;
31     ngx_int_t        bufs;
32
33     unsigned         started:1;
34     unsigned         flush:4;
35     unsigned         redo:1;
36     unsigned         done:1;
37     unsigned         nomem:1;
38
39     z_stream          zstream;
40     ngx_http_request_t *request;
41 } ngx_http_gunzip_ctx_t;
42
43
44 static ngx_int_t ngx_http_gunzip_filter_inflate_start(ngx_http_request_t *r,
45     ngx_http_gunzip_ctx_t *ctx);
46 static ngx_int_t ngx_http_gunzip_filter_add_data(ngx_http_request_t *r,
47     ngx_http_gunzip_ctx_t *ctx);
48 static ngx_int_t ngx_http_gunzip_filter_get_buf(ngx_http_request_t *r,
49     ngx_http_gunzip_ctx_t *ctx);
50 static ngx_int_t ngx_http_gunzip_filter_inflate(ngx_http_request_t *r,
51     ngx_http_gunzip_ctx_t *ctx);
52 static ngx_int_t ngx_http_gunzip_filter_inflate_end(ngx_http_request_t *r,
53     ngx_http_gunzip_ctx_t *ctx);
54
55 static void *ngx_http_gunzip_filter_alloc(void *opaque, u_int items,
56     u_int size);
57 static void ngx_http_gunzip_filter_free(void *opaque, void *address);
58
59 static ngx_int_t ngx_http_gunzip_filter_init(ngx_conf_t *cf);
60 static void *ngx_http_gunzip_create_conf(ngx_conf_t *cf);
61 static char *ngx_http_gunzip_merge_conf(ngx_conf_t *cf,
62     void *parent, void *child);
63
64
65 static ngx_command_t  ngx_http_gunzip_filter_commands[] = {
66
67     { ngx_string("gunzip"),
68       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
69       ngx_conf_set_flag_slot,
70       NGX_HTTP_LOC_CONF_OFFSET,
71       offsetof(ngx_http_gunzip_conf_t, enable),
72       NULL },
73
74     { ngx_string("gunzip_buffers"),
75       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE2,
76       ngx_conf_set_bufs_slot,
77       NGX_HTTP_LOC_CONF_OFFSET,
78       offsetof(ngx_http_gunzip_conf_t, bufs),
79       NULL },
80
81     ngx_null_command
82 };
83
84
85 static ngx_http_module_t  ngx_http_gunzip_filter_module_ctx = {
86     NULL, /* preconfiguration */
87     ngx_http_gunzip_filter_init, /* postconfiguration */
88
89     NULL, /* create main configuration */
90     NULL, /* init main configuration */

```

```

91     NULL,                                /* create server configuration */
92     NULL,                                /* merge server configuration */
93
94     ngx_http_gunzip_create_conf,         /* create location configuration */
95     ngx_http_gunzip_merge_conf          /* merge location configuration */
96 };
97
98
99
100 ngx_module_t ngx_http_gunzip_filter_module = {
101     NGX_MODULE_V1,
102     &ngx_http_gunzip_filter_module_ctx, /* module context */
103     ngx_http_gunzip_filter_commands,    /* module directives */
104     NGX_HTTP_MODULE,                    /* module type */
105     NULL,                                /* init master */
106     NULL,                                /* init module */
107     NULL,                                /* init process */
108     NULL,                                /* init thread */
109     NULL,                                /* exit thread */
110     NULL,                                /* exit process */
111     NULL,                                /* exit master */
112     NGX_MODULE_V1_PADDING
113 };
114
115
116 static ngx_http_output_header_filter_pt ngx_http_next_header_filter;
117 static ngx_http_output_body_filter_pt  ngx_http_next_body_filter;
118
119
120 static ngx_int_t
121 ngx_http_gunzip_header_filter(ngx_http_request_t *r)
122 {
123     ngx_http_gunzip_ctx_t  *ctx;
124     ngx_http_gunzip_conf_t *conf;
125
126     conf = ngx_http_get_module_loc_conf(r, ngx_http_gunzip_filter_module);
127
128     /* TODO support multiple content-codings */
129     /* TODO always gunzip - due to configuration or module request */
130     /* TODO ignore content encoding? */
131
132     if (!conf->enable
133         || r->headers_out.content_encoding == NULL
134         || r->headers_out.content_encoding->value.len != 4
135         || ngx_strncasecmp(r->headers_out.content_encoding->value.data,
136                          (u_char *) "gzip", 4) != 0)
137     {
138         return ngx_http_next_header_filter(r);
139     }
140
141     r->gzip_vary = 1;
142
143     if (!r->gzip_tested) {
144         if (ngx_http_gzip_ok(r) == NGX_OK) {
145             return ngx_http_next_header_filter(r);
146         }
147     }
148     else if (r->gzip_ok) {
149         return ngx_http_next_header_filter(r);
150     }
151
152     ctx = ngx_palloc(r->pool, sizeof(ngx_http_gunzip_ctx_t));
153     if (ctx == NULL) {
154         return NGX_ERROR;
155     }
156
157     ngx_http_set_ctx(r, ctx, ngx_http_gunzip_filter_module);
158
159     ctx->request = r;
160
161     r->filter_need_in_memory = 1;
162
163     r->headers_out.content_encoding->hash = 0;
164     r->headers_out.content_encoding = NULL;
165
166     ngx_http_clear_content_length(r);

```

```

167     ngx_http_clear_accept_ranges(r);
168     ngx_http_weak_etag(r);
169
170     return ngx_http_next_header_filter(r);
171 }
172
173
174 static ngx_int_t
175 ngx_http_gunzip_body_filter(ngx_http_request_t *r, ngx_chain_t *in)
176 {
177     int rc;
178     ngx_uint_t flush;
179     ngx_chain_t *cl;
180     ngx_http_gunzip_ctx_t *ctx;
181
182     ctx = ngx_http_get_module_ctx(r, ngx_http_gunzip_filter_module);
183
184     if (ctx == NULL || ctx->done) {
185         return ngx_http_next_body_filter(r, in);
186     }
187
188     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
189                  "http gunzip filter");
190
191     if (!ctx->started) {
192         if (ngx_http_gunzip_filter_inflate_start(r, ctx) != NGX_OK) {
193             goto failed;
194         }
195     }
196
197     if (in) {
198         if (ngx_chain_add_copy(r->pool, &ctx->in, in) != NGX_OK) {
199             goto failed;
200         }
201     }
202
203     if (ctx->nomem) {
204
205         /* flush busy buffers */
206
207         if (ngx_http_next_body_filter(r, NULL) == NGX_ERROR) {
208             goto failed;
209         }
210
211         cl = NULL;
212
213         ngx_chain_update_chains(r->pool, &ctx->free, &ctx->busy, &cl,
214                               (ngx_buf_tag_t) &ngx_http_gunzip_filter_module);
215         ctx->nomem = 0;
216         flush = 0;
217     } else {
218         flush = ctx->busy ? 1 : 0;
219     }
220
221     for ( ;; ) {
222
223         /* cycle while we can write to a client */
224
225         for ( ;; ) {
226
227             /* cycle while there is data to feed zlib and ... */
228
229             rc = ngx_http_gunzip_filter_add_data(r, ctx);
230
231             if (rc == NGX_DECLINED) {
232                 break;
233             }
234
235             if (rc == NGX_AGAIN) {
236                 continue;
237             }
238
239
240
241             /* ... there are buffers to write zlib output */
242

```

```

243     rc = ngx_http_gunzip_filter_get_buf(r, ctx);
244
245     if (rc == NGX_DECLINED) {
246         break;
247     }
248
249     if (rc == NGX_ERROR) {
250         goto failed;
251     }
252
253     rc = ngx_http_gunzip_filter_inflate(r, ctx);
254
255     if (rc == NGX_OK) {
256         break;
257     }
258
259     if (rc == NGX_ERROR) {
260         goto failed;
261     }
262
263     /* rc == NGX_AGAIN */
264 }
265
266 if (ctx->out == NULL && !flush) {
267     return ctx->busy ? NGX_AGAIN : NGX_OK;
268 }
269
270 rc = ngx_http_next_body_filter(r, ctx->out);
271
272 if (rc == NGX_ERROR) {
273     goto failed;
274 }
275
276 ngx_chain_update_chains(r->pool, &ctx->free, &ctx->busy, &ctx->out,
277     (ngx_buf_tag_t) &ngx_http_gunzip_filter_module);
278 ctx->last_out = &ctx->out;
279
280 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
281     "gunzip out: %p", ctx->out);
282
283 ctx->nomem = 0;
284 flush = 0;
285
286 if (ctx->done) {
287     return rc;
288 }
289 }
290
291 /* unreachable */
292
293 failed:
294
295     ctx->done = 1;
296
297     return NGX_ERROR;
298 }
299
300
301 static ngx_int_t
302 ngx_http_gunzip_filter_inflate_start(ngx_http_request_t *r,
303     ngx_http_gunzip_ctx_t *ctx)
304 {
305     int rc;
306
307     ctx->zstream.next_in = Z_NULL;
308     ctx->zstream.avail_in = 0;
309
310     ctx->zstream.zalloc = ngx_http_gunzip_filter_alloc;
311     ctx->zstream.zfree = ngx_http_gunzip_filter_free;
312     ctx->zstream.opaque = ctx;
313
314     /* windowBits +16 to decode gzip, zlib 1.2.0.4+ */
315     rc = inflateInit2(&ctx->zstream, MAX_WBITS + 16);
316
317     if (rc != Z_OK) {
318         ngx_log_error(NGX_LOG_ALERT, r->connection->log, 0,

```

```

319         "inflateInit2() failed: %d", rc);
320     return NGX\_ERROR;
321 }
322
323 ctx->started = 1;
324
325 ctx->last_out = &ctx->out;
326 ctx->flush = Z_NO_FLUSH;
327
328 return NGX\_OK;
329 }
330
331
332 static ngx\_int\_t
333 ngx\_http\_gunzip\_filter\_add\_data(ngx\_http\_request\_t *r,
334 ngx\_http\_gunzip\_ctx\_t *ctx)
335 {
336     if (ctx->zstream.avail_in || ctx->flush != Z_NO_FLUSH || ctx->redo) {
337         return NGX\_OK;
338     }
339
340     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
341         "gunzip in: %p", ctx->in);
342
343     if (ctx->in == NULL) {
344         return NGX\_DECLINED;
345     }
346
347     ctx->in_buf = ctx->in->buf;
348     ctx->in = ctx->in->next;
349
350     ctx->zstream.next_in = ctx->in_buf->pos;
351     ctx->zstream.avail_in = ctx->in_buf->last - ctx->in_buf->pos;
352
353     ngx\_log\_debug3(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
354         "gunzip in_buf:%p ni:%p ai:%ud",
355         ctx->in_buf,
356         ctx->zstream.next_in, ctx->zstream.avail_in);
357
358     if (ctx->in_buf->last_buf || ctx->in_buf->last_in_chain) {
359         ctx->flush = Z_FINISH;
360     }
361     else if (ctx->in_buf->flush) {
362         ctx->flush = Z_SYNC_FLUSH;
363     }
364     else if (ctx->zstream.avail_in == 0) {
365         /* ctx->flush == Z_NO_FLUSH */
366         return NGX\_AGAIN;
367     }
368
369     return NGX\_OK;
370 }
371
372
373 static ngx\_int\_t
374 ngx\_http\_gunzip\_filter\_get\_buf(ngx\_http\_request\_t *r,
375 ngx\_http\_gunzip\_ctx\_t *ctx)
376 {
377     ngx\_http\_gunzip\_conf\_t *conf;
378
379     if (ctx->zstream.avail_out) {
380         return NGX\_OK;
381     }
382
383     conf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_gunzip\_filter\_module);
384
385     if (ctx->free) {
386         ctx->out_buf = ctx->free->buf;
387         ctx->free = ctx->free->next;
388
389         ctx->out_buf->flush = 0;
390     }
391     else if (ctx->bufs < conf->bufs.num) {
392
393         ctx->out_buf = ngx\_create\_temp\_buf(r->pool, conf->bufs.size);
394         if (ctx->out_buf == NULL) {

```

```

395     return NGX\_ERROR;
396 }
397
398     ctx->out_buf->tag = (ngx\_buf\_tag\_t) &ngx\_http\_gunzip\_filter\_module;
399     ctx->out_buf->recycled = 1;
400     ctx->bufs++;
401
402 } else {
403     ctx->nomem = 1;
404     return NGX\_DECLINED;
405 }
406
407     ctx->zstream.next_out = ctx->out_buf->pos;
408     ctx->zstream.avail_out = conf->bufs.size;
409
410     return NGX\_OK;
411 }
412
413
414 static ngx\_int\_t
415 ngx\_http\_gunzip\_filter\_inflate(ngx\_http\_request\_t *r,
416 ngx\_http\_gunzip\_ctx\_t *ctx)
417 {
418     int         rc;
419     ngx\_buf\_t   *b;
420     ngx\_chain\_t *cl;
421
422     ngx\_log\_debug6(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
423         "inflate in: ni:%p no:%p ai:%ud ao:%ud fl:%d redo:%d",
424         ctx->zstream.next_in, ctx->zstream.next_out,
425         ctx->zstream.avail_in, ctx->zstream.avail_out,
426         ctx->flush, ctx->redo);
427
428     rc = inflate(&ctx->zstream, ctx->flush);
429
430     if (rc != Z_OK && rc != Z_STREAM_END && rc != Z_BUF_ERROR) {
431         ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, 0,
432             "inflate() failed: %d, %d", ctx->flush, rc);
433         return NGX\_ERROR;
434     }
435
436     ngx\_log\_debug5(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
437         "inflate out: ni:%p no:%p ai:%ud ao:%ud rc:%d",
438         ctx->zstream.next_in, ctx->zstream.next_out,
439         ctx->zstream.avail_in, ctx->zstream.avail_out,
440         rc);
441
442     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
443         "gunzip in_buf:%p pos:%p",
444         ctx->in_buf, ctx->in_buf->pos);
445
446     if (ctx->zstream.next_in) {
447         ctx->in_buf->pos = ctx->zstream.next_in;
448
449         if (ctx->zstream.avail_in == 0) {
450             ctx->zstream.next_in = NULL;
451         }
452     }
453
454     ctx->out_buf->last = ctx->zstream.next_out;
455
456     if (ctx->zstream.avail_out == 0) {
457
458         /* zlib wants to output some more data */
459
460         cl = ngx\_alloc\_chain\_link(r->pool);
461         if (cl == NULL) {
462             return NGX\_ERROR;
463         }
464
465         cl->buf = ctx->out_buf;
466         cl->next = NULL;
467         *ctx->last_out = cl;
468         ctx->last_out = &cl->next;
469
470         ctx->redo = 1;

```

```

471     return NGX\_AGAIN;
472 }
473
474 ctx->redo = 0;
475
476 if (ctx->flush == Z_SYNC_FLUSH) {
477     ctx->flush = Z_NO_FLUSH;
478
479     cl = ngx\_alloc\_chain\_link(r->pool);
480     if (cl == NULL) {
481         return NGX\_ERROR;
482     }
483
484     b = ctx->out_buf;
485
486     if (ngx\_buf\_size(b) == 0) {
487         b = ngx\_calloc\_buf(ctx->request->pool);
488         if (b == NULL) {
489             return NGX\_ERROR;
490         }
491     } else {
492         ctx->zstream.avail_out = 0;
493     }
494
495     b->flush = 1;
496
497     cl->buf = b;
498     cl->next = NULL;
499     *ctx->last_out = cl;
500     ctx->last_out = &cl->next;
501
502     return NGX\_OK;
503 }
504
505 if (ctx->flush == Z_FINISH && ctx->zstream.avail_in == 0) {
506     if (rc != Z_STREAM_END) {
507         ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, 0,
508             "inflate() returned %d on response end", rc);
509         return NGX\_ERROR;
510     }
511
512     if (ngx\_http\_gunzip\_filter\_inflate\_end(r, ctx) != NGX\_OK) {
513         return NGX\_ERROR;
514     }
515
516     return NGX\_OK;
517 }
518
519 if (rc == Z_STREAM_END && ctx->zstream.avail_in > 0) {
520     rc = inflateReset(&ctx->zstream);
521
522     if (rc != Z_OK) {
523         ngx\_log\_error(NGX\_LOG\_ALERT, r->connection->log, 0,
524             "inflateReset() failed: %d", rc);
525         return NGX\_ERROR;
526     }
527
528     ctx->redo = 1;
529
530     return NGX\_AGAIN;
531 }
532
533 if (ctx->in == NULL) {
534     b = ctx->out_buf;
535
536     if (ngx\_buf\_size(b) == 0) {
537         return NGX\_OK;
538     }
539 }

```

```

547     cl = ngx_alloc_chain_link(r->pool);
548     if (cl == NULL) {
549         return NGX_ERROR;
550     }
551
552     ctx->zstream.avail_out = 0;
553
554     cl->buf = b;
555     cl->next = NULL;
556     *ctx->last_out = cl;
557     ctx->last_out = &cl->next;
558
559     return NGX_OK;
560 }
561
562 return NGX_AGAIN;
563 }
564
565
566 static ngx_int_t
567 ngx_http_gunzip_filter_inflate_end(ngx_http_request_t *r,
568     ngx_http_gunzip_ctx_t *ctx)
569 {
570     int rc;
571     ngx_buf_t *b;
572     ngx_chain_t *cl;
573
574     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
575         "gunzip inflate end");
576
577     rc = inflateEnd(&ctx->zstream);
578
579     if (rc != Z_OK) {
580         ngx_log_error(NGX_LOG_ALERT, r->connection->log, 0,
581             "inflateEnd() failed: %d", rc);
582         return NGX_ERROR;
583     }
584
585     b = ctx->out_buf;
586
587     if (ngx_buf_size(b) == 0) {
588
589         b = ngx_calloc_buf(ctx->request->pool);
590         if (b == NULL) {
591             return NGX_ERROR;
592         }
593     }
594
595     cl = ngx_alloc_chain_link(r->pool);
596     if (cl == NULL) {
597         return NGX_ERROR;
598     }
599
600     cl->buf = b;
601     cl->next = NULL;
602     *ctx->last_out = cl;
603     ctx->last_out = &cl->next;
604
605     b->last_buf = (r == r->main) ? 1 : 0;
606     b->last_in_chain = 1;
607     b->sync = 1;
608
609     ctx->done = 1;
610
611     return NGX_OK;
612 }
613
614
615 static void *
616 ngx_http_gunzip_filter_alloc(void *opaque, u_int items, u_int size)
617 {
618     ngx_http_gunzip_ctx_t *ctx = opaque;
619
620     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, ctx->request->connection->log, 0,
621         "gunzip alloc: n:%ud s:%ud",
622         items, size);

```



```

623     return ngx_palloc(ctx->request->pool, items * size);
624 }
625
626
627
628 static void
629 ngx_http_gunzip_filter_free(void *opaque, void *address)
630 {
631     #if 0
632         ngx_http_gunzip_ctx_t *ctx = opaque;
633
634         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, ctx->request->connection->log, 0,
635             "gunzip free: %p", address);
636     #endif
637 }
638
639
640 static void *
641 ngx_http_gunzip_create_conf(ngx_conf_t *cf)
642 {
643     ngx_http_gunzip_conf_t *conf;
644
645     conf = ngx_palloc(cf->pool, sizeof(ngx_http_gunzip_conf_t));
646     if (conf == NULL) {
647         return NULL;
648     }
649
650     /*
651      * set by ngx_palloc():
652      *
653      *     conf->bufs.num = 0;
654      */
655
656     conf->enable = NGX_CONF_UNSET;
657
658     return conf;
659 }
660
661
662 static char *
663 ngx_http_gunzip_merge_conf(ngx_conf_t *cf, void *parent, void *child)
664 {
665     ngx_http_gunzip_conf_t *prev = parent;
666     ngx_http_gunzip_conf_t *conf = child;
667
668     ngx_conf_merge_value(conf->enable, prev->enable, 0);
669
670     ngx_conf_merge_bufs_value(conf->bufs, prev->bufs,
671         (128 * 1024) / ngx_pagesize, ngx_pagesize);
672
673     return NGX_CONF_OK;
674 }
675
676
677 static ngx_int_t
678 ngx_http_gunzip_filter_init(ngx_conf_t *cf)
679 {
680     ngx_http_next_header_filter = ngx_http_top_header_filter;
681     ngx_http_top_header_filter = ngx_http_gunzip_header_filter;
682
683     ngx_http_next_body_filter = ngx_http_top_body_filter;
684     ngx_http_top_body_filter = ngx_http_gunzip_body_filter;
685
686     return NGX_OK;
687 }

```

[One Level Up](#)

[Top Level](#)

## src/http/modules/nginx\_http\_gzip\_filter\_module.c - nginx-1.7.10

### Global variables defined

- [ngx\\_http\\_gzip\\_comp\\_level\\_bounds](#)
- [ngx\\_http\\_gzip\\_filter\\_commands](#)
- [ngx\\_http\\_gzip\\_filter\\_module](#)
- [ngx\\_http\\_gzip\\_filter\\_module\\_ctx](#)
- [ngx\\_http\\_gzip\\_hash\\_p](#)
- [ngx\\_http\\_gzip\\_ratio](#)
- [ngx\\_http\\_gzip\\_window\\_p](#)
- [ngx\\_http\\_next\\_body\\_filter](#)
- [ngx\\_http\\_next\\_header\\_filter](#)

### Data types defined

- [gztrailer](#)
- [gztrailer](#)
- [ngx\\_http\\_gzip\\_conf\\_t](#)
- [ngx\\_http\\_gzip\\_ctx\\_t](#)

### Functions defined

- [ngx\\_http\\_gzip\\_add\\_variables](#)
- [ngx\\_http\\_gzip\\_body\\_filter](#)
- [ngx\\_http\\_gzip\\_create\\_conf](#)
- [ngx\\_http\\_gzip\\_filter\\_add\\_data](#)
- [ngx\\_http\\_gzip\\_filter\\_alloc](#)
- [ngx\\_http\\_gzip\\_filter\\_buffer](#)
- [ngx\\_http\\_gzip\\_filter\\_deflate](#)
- [ngx\\_http\\_gzip\\_filter\\_deflate\\_end](#)
- [ngx\\_http\\_gzip\\_filter\\_deflate\\_start](#)
- [ngx\\_http\\_gzip\\_filter\\_free](#)
- [ngx\\_http\\_gzip\\_filter\\_free\\_copy\\_buf](#)
- [ngx\\_http\\_gzip\\_filter\\_get\\_buf](#)
- [ngx\\_http\\_gzip\\_filter\\_gzheader](#)
- [ngx\\_http\\_gzip\\_filter\\_init](#)

- [ngx\\_http\\_gzip\\_filter\\_memory](#)
- [ngx\\_http\\_gzip\\_hash](#)
- [ngx\\_http\\_gzip\\_header\\_filter](#)
- [ngx\\_http\\_gzip\\_merge\\_conf](#)
- [ngx\\_http\\_gzip\\_ratio\\_variable](#)
- [ngx\\_http\\_gzip\\_window](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12 #include <zlib.h>
13
14
15 typedef struct {
16     ngx_flag_t      enable;
17     ngx_flag_t      no_buffer;
18
19     ngx_hash_t      types;
20
21     ngx_bufs_t      bufs;
22
23     size_t          postpone_gzipping;
24     ngx_int_t       level;
25     size_t          wbits;
26     size_t          memlevel;
27     ssize_t         min_length;
28
29     ngx_array_t     *types_keys;
30 } ngx_http_gzip_conf_t;
31
32
33 typedef struct {
34     ngx_chain_t     *in;
35     ngx_chain_t     *free;
36     ngx_chain_t     *busy;
37     ngx_chain_t     *out;
38     ngx_chain_t     **last_out;
39
40     ngx_chain_t     *copied;
41     ngx_chain_t     *copy_buf;
42
43     ngx_buf_t       *in_buf;
44     ngx_buf_t       *out_buf;
45     ngx_int_t       bufs;
46
47     void            *preallocated;
48     char            *free_mem;
49     ngx_uint_t      allocated;
50
51     int             wbits;
52     int             memlevel;
53
54     unsigned        flush:4;
55     unsigned        redo:1;
56     unsigned        done:1;
57     unsigned        nomem:1;
58     unsigned        gzheader:1;
59     unsigned        buffering:1;

```

```

60
61     size_t          zin;
62     size_t          zout;
63
64     uint32_t        crc32;
65     z_stream         zstream;
66     ngx_http_request_t *request;
67 } ngx_http_gzip_ctx_t;
68
69
70 #if (NGX_HAVE_LITTLE_ENDIAN && NGX_HAVE_NONALIGNED)
71
72 struct gztrailer {
73     uint32_t  crc32;
74     uint32_t  zlen;
75 };
76
77 #else /* NGX_HAVE_BIG_ENDIAN || !NGX_HAVE_NONALIGNED */
78
79 struct gztrailer {
80     u_char  crc32[4];
81     u_char  zlen[4];
82 };
83
84 #endif
85
86
87 static void ngx_http_gzip_filter_memory(ngx_http_request_t *r,
88     ngx_http_gzip_ctx_t *ctx);
89 static ngx_int_t ngx_http_gzip_filter_buffer(ngx_http_gzip_ctx_t *ctx,
90     ngx_chain_t *in);
91 static ngx_int_t ngx_http_gzip_filter_deflate_start(ngx_http_request_t *r,
92     ngx_http_gzip_ctx_t *ctx);
93 static ngx_int_t ngx_http_gzip_filter_gzheader(ngx_http_request_t *r,
94     ngx_http_gzip_ctx_t *ctx);
95 static ngx_int_t ngx_http_gzip_filter_add_data(ngx_http_request_t *r,
96     ngx_http_gzip_ctx_t *ctx);
97 static ngx_int_t ngx_http_gzip_filter_get_buf(ngx_http_request_t *r,
98     ngx_http_gzip_ctx_t *ctx);
99 static ngx_int_t ngx_http_gzip_filter_deflate(ngx_http_request_t *r,
100     ngx_http_gzip_ctx_t *ctx);
101 static ngx_int_t ngx_http_gzip_filter_deflate_end(ngx_http_request_t *r,
102     ngx_http_gzip_ctx_t *ctx);
103
104 static void *ngx_http_gzip_filter_alloc(void *opaque, u_int items,
105     u_int size);
106 static void ngx_http_gzip_filter_free(void *opaque, void *address);
107 static void ngx_http_gzip_filter_free_copy_buf(ngx_http_request_t *r,
108     ngx_http_gzip_ctx_t *ctx);
109
110 static ngx_int_t ngx_http_gzip_add_variables(ngx_conf_t *cf);
111 static ngx_int_t ngx_http_gzip_ratio_variable(ngx_http_request_t *r,
112     ngx_http_variable_value_t *v, uintptr_t data);
113
114 static ngx_int_t ngx_http_gzip_filter_init(ngx_conf_t *cf);
115 static void *ngx_http_gzip_create_conf(ngx_conf_t *cf);
116 static char *ngx_http_gzip_merge_conf(ngx_conf_t *cf,
117     void *parent, void *child);
118 static char *ngx_http_gzip_window(ngx_conf_t *cf, void *post, void *data);
119 static char *ngx_http_gzip_hash(ngx_conf_t *cf, void *post, void *data);
120
121
122 static ngx_conf_num_bounds_t  ngx_http_gzip_comp_level_bounds = {
123     ngx_conf_check_num_bounds, 1, 9
124 };
125
126 static ngx_conf_post_handler_pt  ngx_http_gzip_window_p = ngx_http_gzip_window;
127 static ngx_conf_post_handler_pt  ngx_http_gzip_hash_p = ngx_http_gzip_hash;
128
129
130 static ngx_command_t  ngx_http_gzip_filter_commands[] = {
131
132     { ngx_string("gzip"),
133       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_HTTP_LIF_CONF
134         |NGX_CONF_FLAG,
135       ngx_conf_set_flag_slot,

```

```

136     NGX_HTTP_LOC_CONF_OFFSET,
137     offsetof(ngx_http_gzip_conf_t, enable),
138     NULL },
139
140 { ngx_string("gzip_buffers"),
141   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE2,
142   ngx_conf_set_bufs_slot,
143   NGX_HTTP_LOC_CONF_OFFSET,
144   offsetof(ngx_http_gzip_conf_t, bufs),
145   NULL },
146
147 { ngx_string("gzip_types"),
148   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_1MORE,
149   ngx_http_types_slot,
150   NGX_HTTP_LOC_CONF_OFFSET,
151   offsetof(ngx_http_gzip_conf_t, types_keys),
152   &ngx_http_html_default_types[0] },
153
154 { ngx_string("gzip_comp_level"),
155   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
156   ngx_conf_set_num_slot,
157   NGX_HTTP_LOC_CONF_OFFSET,
158   offsetof(ngx_http_gzip_conf_t, level),
159   &ngx_http_gzip_comp_level_bounds },
160
161 { ngx_string("gzip_window"),
162   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
163   ngx_conf_set_size_slot,
164   NGX_HTTP_LOC_CONF_OFFSET,
165   offsetof(ngx_http_gzip_conf_t, wbits),
166   &ngx_http_gzip_window_p },
167
168 { ngx_string("gzip_hash"),
169   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
170   ngx_conf_set_size_slot,
171   NGX_HTTP_LOC_CONF_OFFSET,
172   offsetof(ngx_http_gzip_conf_t, memlevel),
173   &ngx_http_gzip_hash_p },
174
175 { ngx_string("postpone_gzipping"),
176   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
177   ngx_conf_set_size_slot,
178   NGX_HTTP_LOC_CONF_OFFSET,
179   offsetof(ngx_http_gzip_conf_t, postpone_gzipping),
180   NULL },
181
182 { ngx_string("gzip_no_buffer"),
183   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
184   ngx_conf_set_flag_slot,
185   NGX_HTTP_LOC_CONF_OFFSET,
186   offsetof(ngx_http_gzip_conf_t, no_buffer),
187   NULL },
188
189 { ngx_string("gzip_min_length"),
190   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
191   ngx_conf_set_size_slot,
192   NGX_HTTP_LOC_CONF_OFFSET,
193   offsetof(ngx_http_gzip_conf_t, min_length),
194   NULL },
195
196     ngx_null_command
197 };
198
199
200 static ngx_http_module_t ngx_http_gzip_filter_module_ctx = {
201     ngx_http_gzip_add_variables,          /* preconfiguration */
202     ngx_http_gzip_filter_init,          /* postconfiguration */
203
204     NULL,                                  /* create main configuration */
205     NULL,                                  /* init main configuration */
206
207     NULL,                                  /* create server configuration */
208     NULL,                                  /* merge server configuration */
209
210     ngx_http_gzip_create_conf,          /* create location configuration */
211     ngx_http_gzip_merge_conf          /* merge location configuration */

```

```

212 };
213
214
215 ngx_module_t ngx_http_gzip_filter_module = {
216     NGX_MODULE_V1,
217     &ngx_http_gzip_filter_module_ctx,      /* module context */
218     ngx_http_gzip_filter_commands,        /* module directives */
219     NGX_HTTP_MODULE,                      /* module type */
220     NULL,                                  /* init master */
221     NULL,                                  /* init module */
222     NULL,                                  /* init process */
223     NULL,                                  /* init thread */
224     NULL,                                  /* exit thread */
225     NULL,                                  /* exit process */
226     NULL,                                  /* exit master */
227     NGX_MODULE_V1_PADDING
228 };
229
230
231 static ngx_str_t ngx_http_gzip_ratio = ngx_string("gzip_ratio");
232
233 static ngx_http_output_header_filter_pt ngx_http_next_header_filter;
234 static ngx_http_output_body_filter_pt  ngx_http_next_body_filter;
235
236
237 static ngx_int_t
238 ngx_http_gzip_header_filter(ngx_http_request_t *r)
239 {
240     ngx_table_elt_t    *h;
241     ngx_http_gzip_ctx_t    *ctx;
242     ngx_http_gzip_conf_t  *conf;
243
244     conf = ngx_http_get_module_loc_conf(r, ngx_http_gzip_filter_module);
245
246     if (!conf->enable
247         || (r->headers_out.status != NGX_HTTP_OK
248             && r->headers_out.status != NGX_HTTP_FORBIDDEN
249             && r->headers_out.status != NGX_HTTP_NOT_FOUND)
250         || (r->headers_out.content_encoding
251             && r->headers_out.content_encoding->value.len)
252         || (r->headers_out.content_length_n != -1
253             && r->headers_out.content_length_n < conf->min_length)
254         || ngx_http_test_content_type(r, &conf->types) == NULL
255         || r->header_only)
256     {
257         return ngx_http_next_header_filter(r);
258     }
259
260     r->gzip_vary = 1;
261
262     #if (NGX_HTTP_DEGRADATION)
263     {
264         ngx_http_core_loc_conf_t    *clcf;
265
266         clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
267
268         if (clcf->gzip_disable_degradation && ngx_http_degraded(r)) {
269             return ngx_http_next_header_filter(r);
270         }
271     }
272     #endif
273
274     if (!r->gzip_tested) {
275         if (ngx_http_gzip_ok(r) != NGX_OK) {
276             return ngx_http_next_header_filter(r);
277         }
278     }
279
280     } else if (!r->gzip_ok) {
281         return ngx_http_next_header_filter(r);
282     }
283
284     ctx = ngx_palloc(r->pool, sizeof(ngx_http_gzip_ctx_t));
285     if (ctx == NULL) {
286         return NGX_ERROR;
287     }

```

```

288     ngx_http_set_ctx(r, ctx, ngx_http_gzip_filter_module);
289
290     ctx->request = r;
291     ctx->buffering = (conf->postpone_gzipping != 0);
292
293     ngx_http_gzip_filter_memory(r, ctx);
294
295     h = ngx_list_push(&r->headers_out.headers);
296     if (h == NULL) {
297         return NGX_ERROR;
298     }
299
300     h->hash = 1;
301     ngx_str_set(&h->key, "Content-Encoding");
302     ngx_str_set(&h->value, "gzip");
303     r->headers_out.content_encoding = h;
304
305     r->main_filter_need_in_memory = 1;
306
307     ngx_http_clear_content_length(r);
308     ngx_http_clear_accept_ranges(r);
309     ngx_http_weak_etag(r);
310
311     return ngx_http_next_header_filter(r);
312 }
313
314
315 static ngx_int_t
316 ngx_http_gzip_body_filter(ngx_http_request_t *r, ngx_chain_t *in)
317 {
318     int                rc;
319     ngx_uint_t        flush;
320     ngx_chain_t        *cl;
321     ngx_http_gzip_ctx_t *ctx;
322
323     ctx = ngx_http_get_module_ctx(r, ngx_http_gzip_filter_module);
324
325     if (ctx == NULL || ctx->done || r->header_only) {
326         return ngx_http_next_body_filter(r, in);
327     }
328
329     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
330                  "http gzip filter");
331
332     if (ctx->buffering) {
333
334         /*
335          * With default memory settings zlib starts to output gzipped data
336          * only after it has got about 90K, so it makes sense to allocate
337          * zlib memory (200-400K) only after we have enough data to compress.
338          * Although we copy buffers, nevertheless for not big responses
339          * this allows to allocate zlib memory, to compress and to output
340          * the response in one step using hot CPU cache.
341          */
342
343         if (in) {
344             switch (ngx_http_gzip_filter_buffer(ctx, in)) {
345
346                 case NGX_OK:
347                     return NGX_OK;
348
349                 case NGX_DONE:
350                     in = NULL;
351                     break;
352
353                 default: /* NGX_ERROR */
354                     goto failed;
355             }
356
357         } else {
358             ctx->buffering = 0;
359         }
360     }
361
362     if (ctx->preallocated == NULL) {
363         if (ngx_http_gzip_filter_deflate_start(r, ctx) != NGX_OK) {

```

```

364         goto failed;
365     }
366 }
367
368 if (in) {
369     if (ngx\_chain\_add\_copy(r->pool, &ctx->in, in) != NGX\_OK) {
370         goto failed;
371     }
372
373     r->connection->buffered |= NGX\_HTTP\_GZIP\_BUFFERED;
374 }
375
376 if (ctx->nomem) {
377     /* flush busy buffers */
378
379     if (ngx\_http\_next\_body\_filter(r, NULL) == NGX\_ERROR) {
380         goto failed;
381     }
382 }
383
384     cl = NULL;
385
386     ngx\_chain\_update\_chains(r->pool, &ctx->free, &ctx->busy, &cl,
387         (ngx\_buf\_tag\_t) &ngx\_http\_gzip\_filter\_module);
388     ctx->nomem = 0;
389     flush = 0;
390
391 } else {
392     flush = ctx->busy ? 1 : 0;
393 }
394
395 for ( ;; ) {
396
397     /* cycle while we can write to a client */
398
399     for ( ;; ) {
400
401         /* cycle while there is data to feed zlib and ... */
402
403         rc = ngx\_http\_gzip\_filter\_add\_data(r, ctx);
404
405         if (rc == NGX\_DECLINED) {
406             break;
407         }
408
409         if (rc == NGX\_AGAIN) {
410             continue;
411         }
412
413         /* ... there are buffers to write zlib output */
414
415         rc = ngx\_http\_gzip\_filter\_get\_buf(r, ctx);
416
417         if (rc == NGX\_DECLINED) {
418             break;
419         }
420
421         if (rc == NGX\_ERROR) {
422             goto failed;
423         }
424
425
426         rc = ngx\_http\_gzip\_filter\_deflate(r, ctx);
427
428         if (rc == NGX\_OK) {
429             break;
430         }
431
432         if (rc == NGX\_ERROR) {
433             goto failed;
434         }
435
436         /* rc == NGX_AGAIN */
437     }
438 }
439

```



```

440     if (ctx->out == NULL && !flush) {
441         ngx_http_gzip_filter_free_copy_buf(r, ctx);
442     }
443     return ctx->busy ? NGX_AGAIN : NGX_OK;
444 }
445
446 if (!ctx->gzheader) {
447     if (ngx_http_gzip_filter_gzheader(r, ctx) != NGX_OK) {
448         goto failed;
449     }
450 }
451
452 rc = ngx_http_next_body_filter(r, ctx->out);
453
454 if (rc == NGX_ERROR) {
455     goto failed;
456 }
457
458 ngx_http_gzip_filter_free_copy_buf(r, ctx);
459
460 ngx_chain_update_chains(r->pool, &ctx->free, &ctx->busy, &ctx->out,
461     (ngx_buf_tag_t) &ngx_http_gzip_filter_module);
462 ctx->last_out = &ctx->out;
463
464 ctx->nomem = 0;
465 flush = 0;
466
467 if (ctx->done) {
468     return rc;
469 }
470 }
471
472 /* unreachable */
473
474 failed:
475
476 ctx->done = 1;
477
478 if (ctx->preallocated) {
479     deflateEnd(&ctx->zstream);
480
481     ngx_pfree(r->pool, ctx->preallocated);
482 }
483
484 ngx_http_gzip_filter_free_copy_buf(r, ctx);
485
486 return NGX_ERROR;
487 }
488
489
490 static void
491 ngx_http_gzip_filter_memory(ngx_http_request_t *r, ngx_http_gzip_ctx_t *ctx)
492 {
493     int                wbits, memlevel;
494     ngx_http_gzip_conf_t *conf;
495
496     conf = ngx_http_get_module_loc_conf(r, ngx_http_gzip_filter_module);
497
498     wbits = conf->wbits;
499     memlevel = conf->memlevel;
500
501     if (r->headers_out.content_length_n > 0) {
502
503         /* the actual zlib window size is smaller by 262 bytes */
504
505         while (r->headers_out.content_length_n < ((1 << (wbits - 1)) - 262)) {
506             wbits--;
507             memlevel--;
508         }
509
510         if (memlevel < 1) {
511             memlevel = 1;
512         }
513     }
514
515     ctx->wbits = wbits;

```

```

516     ctx->memlevel = memlevel;
517
518     /*
519     * We preallocate a memory for zlib in one buffer (200K-400K), this
520     * decreases a number of malloc() and free() calls and also probably
521     * decreases a number of syscalls (sbrk()/mmap()) and so on).
522     * Besides we free the memory as soon as a gzipping will complete
523     * and do not wait while a whole response will be sent to a client.
524     *
525     * 8K is for zlib deflate_state, it takes
526     * *) 5816 bytes on i386 and sparc64 (32-bit mode)
527     * *) 5920 bytes on amd64 and sparc64
528     */
529
530     ctx->allocated = 8192 + (1 << (wbits + 2)) + (1 << (memlevel + 9));
531 }
532
533
534 static ngx_int_t
535 ngx_http_gzip_filter_buffer(ngx_http_gzip_ctx_t *ctx, ngx_chain_t *in)
536 {
537     size_t          size, buffered;
538     ngx_buf_t      *b, *buf;
539     ngx_chain_t    *cl, **ll;
540     ngx_http_request_t *r;
541     ngx_http_gzip_conf_t *conf;
542
543     r = ctx->request;
544
545     r->connection->buffered |= NGX_HTTP_GZIP_BUFFERED;
546
547     buffered = 0;
548     ll = &ctx->in;
549
550     for (cl = ctx->in; cl; cl = cl->next) {
551         buffered += cl->buf->last - cl->buf->pos;
552         ll = &cl->next;
553     }
554
555     conf = ngx_http_get_module_loc_conf(r, ngx_http_gzip_filter_module);
556
557     while (in) {
558         cl = ngx_alloc_chain_link(r->pool);
559         if (cl == NULL) {
560             return NGX_ERROR;
561         }
562
563         b = in->buf;
564
565         size = b->last - b->pos;
566         buffered += size;
567
568         if (b->flush || b->last_buf || buffered > conf->postpone_gzipping) {
569             ctx->buffering = 0;
570         }
571
572         if (ctx->buffering && size) {
573
574             buf = ngx_create_temp_buf(r->pool, size);
575             if (buf == NULL) {
576                 return NGX_ERROR;
577             }
578
579             buf->last = ngx_cpymem(buf->pos, b->pos, size);
580             b->pos = b->last;
581
582             buf->last_buf = b->last_buf;
583             buf->tag = (ngx_buf_tag_t) &ngx_http_gzip_filter_module;
584
585             cl->buf = buf;
586
587         } else {
588             cl->buf = b;
589         }
590
591         *ll = cl;

```

```

592     ll = &cl->next;
593     in = in->next;
594 }
595
596 *ll = NULL;
597
598 return ctx->buffering ? NGX\_OK : NGX\_DONE;
599 }
600
601
602 static ngx\_int\_t
603 ngx\_http\_gzip\_filter\_deflate\_start(ngx\_http\_request\_t *r,
604 ngx\_http\_gzip\_ctx\_t *ctx)
605 {
606     int rc;
607     ngx\_http\_gzip\_conf\_t *conf;
608
609     conf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_gzip\_filter\_module);
610
611     ctx->preallocated = ngx\_palloc(r->pool, ctx->allocated);
612     if (ctx->preallocated == NULL) {
613         return NGX\_ERROR;
614     }
615
616     ctx->free_mem = ctx->preallocated;
617
618     ctx->zstream.zalloc = ngx\_http\_gzip\_filter\_alloc;
619     ctx->zstream.zfree = ngx\_http\_gzip\_filter\_free;
620     ctx->zstream.opaque = ctx;
621
622     rc = deflateInit2(&ctx->zstream, (int) conf->level, Z_DEFLATED,
623                     - ctx->wbits, ctx->memlevel, Z_DEFAULT_STRATEGY);
624
625     if (rc != Z_OK) {
626         ngx\_log\_error(NGX\_LOG\_ALERT, r->connection->log, 0,
627                     "deflateInit2() failed: %d", rc);
628         return NGX\_ERROR;
629     }
630
631     ctx->last_out = &ctx->out;
632     ctx->crc32 = crc32(0L, Z_NULL, 0);
633     ctx->flush = Z_NO_FLUSH;
634
635     return NGX\_OK;
636 }
637
638
639 static ngx\_int\_t
640 ngx\_http\_gzip\_filter\_gzheader(ngx\_http\_request\_t *r, ngx\_http\_gzip\_ctx\_t *ctx)
641 {
642     ngx\_buf\_t *b;
643     ngx\_chain\_t *cl;
644     static u_char gzheader[10] =
645         { 0x1f, 0x8b, Z_DEFLATED, 0, 0, 0, 0, 0, 0, 3 };
646
647     b = ngx\_palloc(r->pool, sizeof(ngx\_buf\_t));
648     if (b == NULL) {
649         return NGX\_ERROR;
650     }
651
652     b->memory = 1;
653     b->pos = gzheader;
654     b->last = b->pos + 10;
655
656     cl = ngx\_alloc\_chain\_link(r->pool);
657     if (cl == NULL) {
658         return NGX\_ERROR;
659     }
660
661     cl->buf = b;
662     cl->next = ctx->out;
663     ctx->out = cl;
664
665     ctx->gzheader = 1;
666
667     return NGX\_OK;

```

```

668 }
669
670
671 static ngx_int_t
672 ngx_http_gzip_filter_add_data(ngx_http_request_t *r, ngx_http_gzip_ctx_t *ctx)
673 {
674     if (ctx->zstream.avail_in || ctx->flush != Z_NO_FLUSH || ctx->redo) {
675         return NGX_OK;
676     }
677
678     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
679                 "gzip in: %p", ctx->in);
680
681     if (ctx->in == NULL) {
682         return NGX_DECLINED;
683     }
684
685     if (ctx->copy_buf) {
686
687         /*
688          * to avoid CPU cache trashing we do not free() just quit buf,
689          * but postpone free()ing after zlib compressing and data output
690          */
691
692         ctx->copy_buf->next = ctx->copied;
693         ctx->copied = ctx->copy_buf;
694         ctx->copy_buf = NULL;
695     }
696
697     ctx->in_buf = ctx->in->buf;
698
699     if (ctx->in_buf->tag == (ngx_buf_tag_t) &ngx_http_gzip_filter_module) {
700         ctx->copy_buf = ctx->in;
701     }
702
703     ctx->in = ctx->in->next;
704
705     ctx->zstream.next_in = ctx->in_buf->pos;
706     ctx->zstream.avail_in = ctx->in_buf->last - ctx->in_buf->pos;
707
708     ngx_log_debug3(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
709                 "gzip in_buf:%p ni:%p ai:%ud",
710                 ctx->in_buf,
711                 ctx->zstream.next_in, ctx->zstream.avail_in);
712
713     if (ctx->in_buf->last_buf) {
714         ctx->flush = Z_FINISH;
715     } else if (ctx->in_buf->flush) {
716         ctx->flush = Z_SYNC_FLUSH;
717     }
718
719     if (ctx->zstream.avail_in) {
720
721         ctx->crc32 = crc32(ctx->crc32, ctx->zstream.next_in,
722                         ctx->zstream.avail_in);
723
724     } else if (ctx->flush == Z_NO_FLUSH) {
725         return NGX_AGAIN;
726     }
727
728     return NGX_OK;
729 }
730 }
731
732
733 static ngx_int_t
734 ngx_http_gzip_filter_get_buf(ngx_http_request_t *r, ngx_http_gzip_ctx_t *ctx)
735 {
736     ngx_http_gzip_conf_t *conf;
737
738     if (ctx->zstream.avail_out) {
739         return NGX_OK;
740     }
741
742     conf = ngx_http_get_module_loc_conf(r, ngx_http_gzip_filter_module);
743

```

```

744     if (ctx->free) {
745         ctx->out_buf = ctx->free->buf;
746         ctx->free = ctx->free->next;
747     }
748     } else if (ctx->bufs < conf->bufs.num) {
749
750         ctx->out_buf = ngx\_create\_temp\_buf(r->pool, conf->bufs.size);
751         if (ctx->out_buf == NULL) {
752             return NGX\_ERROR;
753         }
754
755         ctx->out_buf->tag = (ngx\_buf\_tag\_t) &ngx\_http\_gzip\_filter\_module;
756         ctx->out_buf->recycled = 1;
757         ctx->bufs++;
758     }
759     } else {
760         ctx->nomem = 1;
761         return NGX\_DECLINED;
762     }
763
764     ctx->zstream.next_out = ctx->out_buf->pos;
765     ctx->zstream.avail_out = conf->bufs.size;
766
767     return NGX\_OK;
768 }
769
770
771 static ngx\_int\_t
772 ngx\_http\_gzip\_filter\_deflate(ngx\_http\_request\_t *r, ngx\_http\_gzip\_ctx\_t *ctx)
773 {
774     int                rc;
775     ngx\_buf\_t         *b;
776     ngx\_chain\_t       *cl;
777     ngx\_http\_gzip\_conf\_t *conf;
778
779     ngx\_log\_debug6(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
780                 "deflate in: ni:%p no:%p ai:%ud ao:%ud fl:%d redo:%d",
781                 ctx->zstream.next_in, ctx->zstream.next_out,
782                 ctx->zstream.avail_in, ctx->zstream.avail_out,
783                 ctx->flush, ctx->redo);
784
785     rc = deflate(&ctx->zstream, ctx->flush);
786
787     if (rc != Z_OK && rc != Z_STREAM_END && rc != Z_BUF_ERROR) {
788         ngx\_log\_error(NGX\_LOG\_ALERT, r->connection->log, 0,
789                     "deflate() failed: %d, %d", ctx->flush, rc);
790         return NGX\_ERROR;
791     }
792
793     ngx\_log\_debug5(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
794                 "deflate out: ni:%p no:%p ai:%ud ao:%ud rc:%d",
795                 ctx->zstream.next_in, ctx->zstream.next_out,
796                 ctx->zstream.avail_in, ctx->zstream.avail_out,
797                 rc);
798
799     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
800                 "gzip in_buf:%p pos:%p",
801                 ctx->in_buf, ctx->in_buf->pos);
802
803     if (ctx->zstream.next_in) {
804         ctx->in_buf->pos = ctx->zstream.next_in;
805
806         if (ctx->zstream.avail_in == 0) {
807             ctx->zstream.next_in = NULL;
808         }
809     }
810
811     ctx->out_buf->last = ctx->zstream.next_out;
812
813     if (ctx->zstream.avail_out == 0) {
814
815         /* zlib wants to output some more gzipped data */
816
817         cl = ngx\_alloc\_chain\_link(r->pool);
818         if (cl == NULL) {
819             return NGX\_ERROR;

```

```

820     }
821
822     cl->buf = ctx->out_buf;
823     cl->next = NULL;
824     *ctx->last_out = cl;
825     ctx->last_out = &cl->next;
826
827     ctx->redo = 1;
828
829     return NGX\_AGAIN;
830 }
831
832 ctx->redo = 0;
833
834 if (ctx->flush == Z_SYNC_FLUSH) {
835     ctx->flush = Z_NO_FLUSH;
836
837     cl = ngx\_alloc\_chain\_link(r->pool);
838     if (cl == NULL) {
839         return NGX\_ERROR;
840     }
841
842     b = ctx->out_buf;
843
844     if (ngx\_buf\_size(b) == 0) {
845         b = ngx\_calloc\_buf(ctx->request->pool);
846         if (b == NULL) {
847             return NGX\_ERROR;
848         }
849     } else {
850         ctx->zstream.avail_out = 0;
851     }
852
853     b->flush = 1;
854
855     cl->buf = b;
856     cl->next = NULL;
857     *ctx->last_out = cl;
858     ctx->last_out = &cl->next;
859
860     r->connection->buffered &= ~NGX\_HTTP\_GZIP\_BUFFERED;
861
862     return NGX\_OK;
863 }
864
865 if (rc == Z_STREAM_END) {
866     if (ngx\_http\_gzip\_filter\_deflate\_end(r, ctx) != NGX\_OK) {
867         return NGX\_ERROR;
868     }
869
870     return NGX\_OK;
871 }
872
873 conf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_gzip\_filter\_module);
874
875 if (conf->no_buffer && ctx->in == NULL) {
876     cl = ngx\_alloc\_chain\_link(r->pool);
877     if (cl == NULL) {
878         return NGX\_ERROR;
879     }
880
881     cl->buf = ctx->out_buf;
882     cl->next = NULL;
883     *ctx->last_out = cl;
884     ctx->last_out = &cl->next;
885
886     return NGX\_OK;
887 }
888
889 return NGX\_AGAIN;
890 }
891
892 }

```

```

896
897
898 static ngx_int_t
899 ngx_http_gzip_filter_deflate_end(ngx_http_request_t *r,
900     ngx_http_gzip_ctx_t *ctx)
901 {
902     int                rc;
903     ngx_buf_t         *b;
904     ngx_chain_t       *cl;
905     struct gztrailer  *trailer;
906
907     ctx->zin = ctx->zstream.total_in;
908     ctx->zout = 10 + ctx->zstream.total_out + 8;
909
910     rc = deflateEnd(&ctx->zstream);
911
912     if (rc != Z_OK) {
913         ngx_log_error(NGX_LOG_ALERT, r->connection->log, 0,
914             "deflateEnd() failed: %d", rc);
915         return NGX_ERROR;
916     }
917
918     ngx_pfree(r->pool, ctx->preallocated);
919
920     cl = ngx_alloc_chain_link(r->pool);
921     if (cl == NULL) {
922         return NGX_ERROR;
923     }
924
925     cl->buf = ctx->out_buf;
926     cl->next = NULL;
927     *ctx->last_out = cl;
928     ctx->last_out = &cl->next;
929
930     if (ctx->zstream.avail_out >= 8) {
931         trailer = (struct gztrailer *) ctx->out_buf->last;
932         ctx->out_buf->last += 8;
933         ctx->out_buf->last_buf = 1;
934     } else {
935         b = ngx_create_temp_buf(r->pool, 8);
936         if (b == NULL) {
937             return NGX_ERROR;
938         }
939
940         b->last_buf = 1;
941
942         cl = ngx_alloc_chain_link(r->pool);
943         if (cl == NULL) {
944             return NGX_ERROR;
945         }
946
947         cl->buf = b;
948         cl->next = NULL;
949         *ctx->last_out = cl;
950         ctx->last_out = &cl->next;
951         trailer = (struct gztrailer *) b->pos;
952         b->last += 8;
953     }
954 }
955
956 #if (NGX_HAVE_LITTLE_ENDIAN && NGX_HAVE_NONALIGNED)
957
958     trailer->crc32 = ctx->crc32;
959     trailer->zlen = ctx->zin;
960
961 #else
962
963     trailer->crc32[0] = (u_char) (ctx->crc32 & 0xff);
964     trailer->crc32[1] = (u_char) ((ctx->crc32 >> 8) & 0xff);
965     trailer->crc32[2] = (u_char) ((ctx->crc32 >> 16) & 0xff);
966     trailer->crc32[3] = (u_char) ((ctx->crc32 >> 24) & 0xff);
967
968     trailer->zlen[0] = (u_char) (ctx->zin & 0xff);
969     trailer->zlen[1] = (u_char) ((ctx->zin >> 8) & 0xff);
970     trailer->zlen[2] = (u_char) ((ctx->zin >> 16) & 0xff);
971     trailer->zlen[3] = (u_char) ((ctx->zin >> 24) & 0xff);

```

```

972
973 #endif
974
975     ctx->zstream.avail_in = 0;
976     ctx->zstream.avail_out = 0;
977
978     ctx->done = 1;
979
980     r->connection->buffered &= ~NGX_HTTP_GZIP_BUFFERED;
981
982     return NGX_OK;
983 }
984
985
986 static void *
987 ngx_http_gzip_filter_alloc(void *opaque, u_int items, u_int size)
988 {
989     ngx_http_gzip_ctx_t *ctx = opaque;
990
991     void *p;
992     ngx_uint_t alloc;
993
994     alloc = items * size;
995
996     if (alloc % 512 != 0 && alloc < 8192) {
997
998         /*
999          * The zlib deflate_state allocation, it takes about 6K,
1000          * we allocate 8K. Other allocations are divisible by 512.
1001          */
1002
1003         alloc = 8192;
1004     }
1005
1006     if (alloc <= ctx->allocated) {
1007         p = ctx->free_mem;
1008         ctx->free_mem += alloc;
1009         ctx->allocated -= alloc;
1010
1011         ngx_log_debug4(NGX_LOG_DEBUG_HTTP, ctx->request->connection->log, 0,
1012             "gzip alloc: n:%ud s:%ud a:%ud p:%p",
1013             items, size, alloc, p);
1014
1015         return p;
1016     }
1017
1018     ngx_log_error(NGX_LOG_ALERT, ctx->request->connection->log, 0,
1019         "gzip filter failed to use preallocated memory: %ud of %ud",
1020         items * size, ctx->allocated);
1021
1022     p = ngx_palloc(ctx->request->pool, items * size);
1023
1024     return p;
1025 }
1026
1027
1028 static void
1029 ngx_http_gzip_filter_free(void *opaque, void *address)
1030 {
1031     #if 0
1032         ngx_http_gzip_ctx_t *ctx = opaque;
1033
1034         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, ctx->request->connection->log, 0,
1035             "gzip free: %p", address);
1036     #endif
1037 }
1038
1039
1040 static void
1041 ngx_http_gzip_filter_free_copy_buf(ngx_http_request_t *r,
1042     ngx_http_gzip_ctx_t *ctx)
1043 {
1044     ngx_chain_t *cl;
1045
1046     for (cl = ctx->copied; cl; cl = cl->next) {
1047         ngx_pfree(r->pool, cl->buf->start);

```



```

1048     }
1049
1050     ctx->copied = NULL;
1051 }
1052
1053
1054 static ngx_int_t
1055 ngx_http_gzip_add_variables(ngx_conf_t *cf)
1056 {
1057     ngx_http_variable_t *var;
1058
1059     var = ngx_http_add_variable(cf, &ngx_http_gzip_ratio, NGX_HTTP_VAR_NOHASH);
1060     if (var == NULL) {
1061         return NGX_ERROR;
1062     }
1063
1064     var->get_handler = ngx_http_gzip_ratio_variable;
1065
1066     return NGX_OK;
1067 }
1068
1069
1070 static ngx_int_t
1071 ngx_http_gzip_ratio_variable(ngx_http_request_t *r,
1072     ngx_http_variable_value_t *v, uintptr_t data)
1073 {
1074     ngx_uint_t          zint, zfrac;
1075     ngx_http_gzip_ctx_t *ctx;
1076
1077     v->valid = 1;
1078     v->no_cacheable = 0;
1079     v->not_found = 0;
1080
1081     ctx = ngx_http_get_module_ctx(r, ngx_http_gzip_filter_module);
1082
1083     if (ctx == NULL || ctx->zout == 0) {
1084         v->not_found = 1;
1085         return NGX_OK;
1086     }
1087
1088     v->data = ngx_pnalloc(r->pool, NGX_INT32_LEN + 3);
1089     if (v->data == NULL) {
1090         return NGX_ERROR;
1091     }
1092
1093     zint = (ngx_uint_t) (ctx->zin / ctx->zout);
1094     zfrac = (ngx_uint_t) ((ctx->zin * 100 / ctx->zout) % 100);
1095
1096     if ((ctx->zin * 1000 / ctx->zout) % 10 > 4) {
1097         /* the rounding, e.g., 2.125 to 2.13 */
1098
1099         zfrac++;
1100
1101         if (zfrac > 99) {
1102             zint++;
1103             zfrac = 0;
1104         }
1105     }
1106 }
1107
1108 v->len = ngx_sprintf(v->data, "%ui.%02ui", zint, zfrac) - v->data;
1109
1110 return NGX_OK;
1111 }
1112
1113
1114 static void *
1115 ngx_http_gzip_create_conf(ngx_conf_t *cf)
1116 {
1117     ngx_http_gzip_conf_t *conf;
1118
1119     conf = ngx_pcalloc(cf->pool, sizeof(ngx_http_gzip_conf_t));
1120     if (conf == NULL) {
1121         return NULL;
1122     }
1123 }

```

```

1124  /*
1125  * set by ngx\_palloc\(\):
1126  *
1127  *     conf->bufs.num = 0;
1128  *     conf->types = { NULL };
1129  *     conf->types_keys = NULL;
1130  */
1131
1132  conf->enable = NGX\_CONF\_UNSET;
1133  conf->no_buffer = NGX\_CONF\_UNSET;
1134
1135  conf->postpone_gzipping = NGX\_CONF\_UNSET\_SIZE;
1136  conf->level = NGX\_CONF\_UNSET;
1137  conf->wbits = NGX\_CONF\_UNSET\_SIZE;
1138  conf->memlevel = NGX\_CONF\_UNSET\_SIZE;
1139  conf->min_length = NGX\_CONF\_UNSET;
1140
1141  return conf;
1142 }
1143
1144
1145 static char *
1146 ngx_http_gzip_merge_conf(ngx\_conf\_t *cf, void *parent, void *child)
1147 {
1148     ngx\_http\_gzip\_conf\_t *prev = parent;
1149     ngx\_http\_gzip\_conf\_t *conf = child;
1150
1151     ngx\_conf\_merge\_value(conf->enable, prev->enable, 0);
1152     ngx\_conf\_merge\_value(conf->no_buffer, prev->no_buffer, 0);
1153
1154     ngx\_conf\_merge\_bufs\_value(conf->bufs, prev->bufs,
1155                               (128 * 1024) / ngx\_pagesize, ngx\_pagesize);
1156
1157     ngx\_conf\_merge\_size\_value(conf->postpone_gzipping, prev->postpone_gzipping,
1158                               0);
1159     ngx\_conf\_merge\_value(conf->level, prev->level, 1);
1160     ngx\_conf\_merge\_size\_value(conf->wbits, prev->wbits, MAX_WBITS);
1161     ngx\_conf\_merge\_size\_value(conf->memlevel, prev->memlevel,
1162                               MAX_MEM_LEVEL - 1);
1163     ngx\_conf\_merge\_value(conf->min_length, prev->min_length, 20);
1164
1165     if (ngx\_http\_merge\_types(cf, &conf->types_keys, &conf->types,
1166                             &prev->types_keys, &prev->types,
1167                             ngx\_http\_html\_default\_types)
1168         != NGX\_OK)
1169     {
1170         return NGX\_CONF\_ERROR;
1171     }
1172
1173     return NGX\_CONF\_OK;
1174 }
1175
1176
1177 static ngx\_int\_t
1178 ngx_http_gzip_filter_init(ngx\_conf\_t *cf)
1179 {
1180     ngx\_http\_next\_header\_filter = ngx\_http\_top\_header\_filter;
1181     ngx\_http\_top\_header\_filter = ngx\_http\_gzip\_header\_filter;
1182
1183     ngx\_http\_next\_body\_filter = ngx\_http\_top\_body\_filter;
1184     ngx\_http\_top\_body\_filter = ngx\_http\_gzip\_body\_filter;
1185
1186     return NGX\_OK;
1187 }
1188
1189
1190 static char *
1191 ngx_http_gzip_window(ngx\_conf\_t *cf, void *post, void *data)
1192 {
1193     size\_t *np = data;
1194
1195     size\_t wbits, wsize;
1196
1197     wbits = 15;
1198
1199     for (wsize = 32 * 1024; wsize > 256; wsize >>= 1) {

```

```
1200         if (wsize == *np) {
1201             *np = wbits;
1202
1203             return NGX\_CONF\_OK;
1204         }
1205
1206         wbits--;
1207     }
1208
1209     return "must be 512, 1k, 2k, 4k, 8k, 16k, or 32k";
1210 }
1211
1212
1213 static char *
1214 ngx_http_gzip_hash(ngx\_conf\_t *cf, void *post, void *data)
1215 {
1216     size_t *np = data;
1217
1218     size_t memlevel, hsize;
1219
1220     memlevel = 9;
1221
1222     for (hsize = 128 * 1024; hsize > 256; hsize >>= 1) {
1223
1224         if (hsize == *np) {
1225             *np = memlevel;
1226
1227             return NGX\_CONF\_OK;
1228         }
1229
1230         memlevel--;
1231     }
1232
1233     return "must be 512, 1k, 2k, 4k, 8k, 16k, 32k, 64k, or 128k";
1234 }
1235 }
```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_gzip\_static\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_gzip\\_static](#)
- [ngx\\_http\\_gzip\\_static\\_commands](#)
- [ngx\\_http\\_gzip\\_static\\_module](#)
- [ngx\\_http\\_gzip\\_static\\_module\\_ctx](#)

## Data types defined

- [ngx\\_http\\_gzip\\_static\\_conf\\_t](#)

## Functions defined

- [ngx\\_http\\_gzip\\_static\\_create\\_conf](#)
- [ngx\\_http\\_gzip\\_static\\_handler](#)
- [ngx\\_http\\_gzip\\_static\\_init](#)
- [ngx\\_http\\_gzip\\_static\\_merge\\_conf](#)

## Macros defined

- [NGX\\_HTTP\\_GZIP\\_STATIC\\_ALWAYS](#)
- [NGX\\_HTTP\\_GZIP\\_STATIC\\_OFF](#)
- [NGX\\_HTTP\\_GZIP\\_STATIC\\_ON](#)

## Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 #define NGX_HTTP_GZIP_STATIC_OFF    0
14 #define NGX_HTTP_GZIP_STATIC_ON    1
15 #define NGX_HTTP_GZIP_STATIC_ALWAYS 2
16
17
18 typedef struct {
19     ngx\_uint\_t enable;
20 } ngx\_http\_gzip\_static\_conf\_t;
21
22
23 static ngx\_int\_t ngx\_http\_gzip\_static\_handler(ngx\_http\_request\_t *r);
24 static void *ngx\_http\_gzip\_static\_create\_conf(ngx\_conf\_t *cf);
25 static char *ngx\_http\_gzip\_static\_merge\_conf(ngx\_conf\_t *cf, void *parent,
26     void *child);
27 static ngx\_int\_t ngx\_http\_gzip\_static\_init(ngx\_conf\_t *cf);
```

```

28
29
30 static ngx_conf_enum_t ngx_http_gzip_static[] = {
31     { ngx_string("off"), NGX_HTTP_GZIP_STATIC_OFF },
32     { ngx_string("on"), NGX_HTTP_GZIP_STATIC_ON },
33     { ngx_string("always"), NGX_HTTP_GZIP_STATIC_ALWAYS },
34     { ngx_null_string, 0 }
35 };
36
37
38 static ngx_command_t ngx_http_gzip_static_commands[] = {
39
40     { ngx_string("gzip_static"),
41       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF TAKE1,
42       ngx_conf_set_enum_slot,
43       NGX_HTTP_LOC_CONF_OFFSET,
44       offsetof(ngx_http_gzip_static_conf_t, enable),
45       &ngx_http_gzip_static },
46
47     ngx_null_command
48 };
49
50
51 ngx_http_module_t ngx_http_gzip_static_module_ctx = {
52     NULL, /* preconfiguration */
53     ngx_http_gzip_static_init, /* postconfiguration */
54
55     NULL, /* create main configuration */
56     NULL, /* init main configuration */
57
58     NULL, /* create server configuration */
59     NULL, /* merge server configuration */
60
61     ngx_http_gzip_static_create_conf, /* create location configuration */
62     ngx_http_gzip_static_merge_conf /* merge location configuration */
63 };
64
65
66 ngx_module_t ngx_http_gzip_static_module = {
67     NGX_MODULE_V1,
68     &ngx_http_gzip_static_module_ctx, /* module context */
69     ngx_http_gzip_static_commands, /* module directives */
70     NGX_HTTP_MODULE, /* module type */
71     NULL, /* init master */
72     NULL, /* init module */
73     NULL, /* init process */
74     NULL, /* init thread */
75     NULL, /* exit thread */
76     NULL, /* exit process */
77     NULL, /* exit master */
78     NGX_MODULE_V1_PADDING
79 };
80
81
82 static ngx_int_t
83 ngx_http_gzip_static_handler(ngx_http_request_t *r)
84 {
85     u_char *p;
86     size_t root;
87     ngx_str_t path;
88     ngx_int_t rc;
89     ngx_uint_t level;
90     ngx_log_t *log;
91     ngx_buf_t *b;
92     ngx_chain_t out;
93     ngx_table_elt_t *h;
94     ngx_open_file_info_t of;
95     ngx_http_core_loc_conf_t *clcf;
96     ngx_http_gzip_static_conf_t *gzcf;
97
98     if (!(r->method & (NGX_HTTP_GET|NGX_HTTP_HEAD))) {
99         return NGX_DECLINED;
100     }
101
102     if (r->uri.data[r->uri.len - 1] == '/') {
103         return NGX_DECLINED;

```

```

104 }
105
106 gzcf = ngx_http_get_module_loc_conf(r, ngx_http_gzip_static_module);
107
108 if (gzcf->enable == NGX_HTTP_GZIP_STATIC_OFF) {
109     return NGX_DECLINED;
110 }
111
112 if (gzcf->enable == NGX_HTTP_GZIP_STATIC_ON) {
113     rc = ngx_http_gzip_ok(r);
114 }
115 } else {
116     /* always */
117     rc = NGX_OK;
118 }
119
120 clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
121
122 if (!clcf->gzip_vary && rc != NGX_OK) {
123     return NGX_DECLINED;
124 }
125
126 log = r->connection->log;
127
128 p = ngx_http_map_uri_to_path(r, &path, &root, sizeof(".gz") - 1);
129 if (p == NULL) {
130     return NGX_HTTP_INTERNAL_SERVER_ERROR;
131 }
132
133 *p++ = '.';
134 *p++ = 'g';
135 *p++ = 'z';
136 *p = '\0';
137
138 path.len = p - path.data;
139
140 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, log, 0,
141              "http filename: \"%s\"", path.data);
142
143 ngx_memzero(&of, sizeof(ngx_open_file_info_t));
144
145 of.read_ahead = clcf->read_ahead;
146 of.directio = clcf->directio;
147 of.valid = clcf->open_file_cache_valid;
148 of.min_uses = clcf->open_file_cache_min_uses;
149 of.errors = clcf->open_file_cache_errors;
150 of.events = clcf->open_file_cache_events;
151
152 if (ngx_http_set_disable_symlinks(r, clcf, &path, &of) != NGX_OK) {
153     return NGX_HTTP_INTERNAL_SERVER_ERROR;
154 }
155
156 if (ngx_open_cached_file(clcf->open_file_cache, &path, &of, r->pool)
157     != NGX_OK)
158 {
159     switch (of.err) {
160
161     case 0:
162         return NGX_HTTP_INTERNAL_SERVER_ERROR;
163
164     case NGX_ENOENT:
165     case NGX_ENOTDIR:
166     case NGX_ENAMETOOLONG:
167
168         return NGX_DECLINED;
169
170     case NGX_EACCES:
171 #if (NGX_HAVE_OPENAT)
172     case NGX_EMLINK:
173     case NGX_ELOOP:
174 #endif
175
176         level = NGX_LOG_ERR;
177         break;
178
179     default:

```

```

180         level = NGX_LOG_CRIT;
181         break;
182     }
183
184     ngx_log_error(level, log, of.err,
185                 "%s \"%s\" failed", of.failed, path.data);
186
187     return NGX_DECLINED;
188 }
189
190 if (gzcf->enable == NGX_HTTP_GZIP_STATIC_ON) {
191     r->gzip_vary = 1;
192
193     if (rc != NGX_OK) {
194         return NGX_DECLINED;
195     }
196 }
197
198 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, log, 0, "http static fd: %d", of.fd);
199
200 if (of.is_dir) {
201     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, log, 0, "http dir");
202     return NGX_DECLINED;
203 }
204
205 #if !(NGX_WIN32) /* the not regular files are probably Unix specific */
206
207 if (!of.is_file) {
208     ngx_log_error(NGX_LOG_CRIT, log, 0,
209                 "\"%s\" is not a regular file", path.data);
210
211     return NGX_HTTP_NOT_FOUND;
212 }
213
214 #endif
215
216 r->root_tested = !r->error_page;
217
218 rc = ngx_http_discard_request_body(r);
219
220 if (rc != NGX_OK) {
221     return rc;
222 }
223
224 log->action = "sending response to client";
225
226 r->headers_out.status = NGX_HTTP_OK;
227 r->headers_out.content_length_n = of.size;
228 r->headers_out.last_modified_time = of.mtime;
229
230 if (ngx_http_set_etag(r) != NGX_OK) {
231     return NGX_HTTP_INTERNAL_SERVER_ERROR;
232 }
233
234 if (ngx_http_set_content_type(r) != NGX_OK) {
235     return NGX_HTTP_INTERNAL_SERVER_ERROR;
236 }
237
238 h = ngx_list_push(&r->headers_out.headers);
239 if (h == NULL) {
240     return NGX_ERROR;
241 }
242
243 h->hash = 1;
244 ngx_str_set(&h->key, "Content-Encoding");
245 ngx_str_set(&h->value, "gzip");
246 r->headers_out.content_encoding = h;
247
248 /* we need to allocate all before the header would be sent */
249
250 b = ngx_palloc(r->pool, sizeof(ngx_buf_t));
251 if (b == NULL) {
252     return NGX_HTTP_INTERNAL_SERVER_ERROR;
253 }
254
255

```

```

256     b->file = ngx_palloc(r->pool, sizeof(ngx_file_t));
257     if (b->file == NULL) {
258         return NGX_HTTP_INTERNAL_SERVER_ERROR;
259     }
260
261     rc = ngx_http_send_header(r);
262
263     if (rc == NGX_ERROR || rc > NGX_OK || r->header_only) {
264         return rc;
265     }
266
267     b->file_pos = 0;
268     b->file_last = of.size;
269
270     b->in_file = b->file_last ? 1 : 0;
271     b->last_buf = (r == r->main) ? 1 : 0;
272     b->last_in_chain = 1;
273
274     b->file->fd = of.fd;
275     b->file->name = path;
276     b->file->log = log;
277     b->file->directio = of.is_directio;
278
279     out.buf = b;
280     out.next = NULL;
281
282     return ngx_http_output_filter(r, &out);
283 }
284
285
286 static void *
287 ngx_http_gzip_static_create_conf(ngx_conf_t *cf)
288 {
289     ngx_http_gzip_static_conf_t *conf;
290
291     conf = ngx_palloc(cf->pool, sizeof(ngx_http_gzip_static_conf_t));
292     if (conf == NULL) {
293         return NULL;
294     }
295
296     conf->enable = NGX_CONF_UNSET_UINT;
297
298     return conf;
299 }
300
301
302 static char *
303 ngx_http_gzip_static_merge_conf(ngx_conf_t *cf, void *parent, void *child)
304 {
305     ngx_http_gzip_static_conf_t *prev = parent;
306     ngx_http_gzip_static_conf_t *conf = child;
307
308     ngx_conf_merge_uint_value(conf->enable, prev->enable,
309                               NGX_HTTP_GZIP_STATIC_OFF);
310
311     return NGX_CONF_OK;
312 }
313
314
315 static ngx_int_t
316 ngx_http_gzip_static_init(ngx_conf_t *cf)
317 {
318     ngx_http_handler_pt *h;
319     ngx_http_core_main_conf_t *cmcf;
320
321     cmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_core_module);
322
323     h = ngx_array_push(&cmcf->phases[NGX_HTTP_CONTENT_PHASE].handlers);
324     if (h == NULL) {
325         return NGX_ERROR;
326     }
327
328     *h = ngx_http_gzip_static_handler;
329
330     return NGX_OK;
331 }

```



[One Level Up](#)

[Top Level](#)

## src/http/modules/nginx\_http\_headers\_filter\_module.c - nginx-1.7.10

### Global variables defined

- [ngx\\_http\\_headers\\_filter\\_commands](#)
- [ngx\\_http\\_headers\\_filter\\_module](#)
- [ngx\\_http\\_headers\\_filter\\_module\\_ctx](#)
- [ngx\\_http\\_next\\_header\\_filter](#)
- [ngx\\_http\\_set\\_headers](#)

### Data types defined

- [ngx\\_http\\_expires\\_t](#)
- [ngx\\_http\\_header\\_val\\_s](#)
- [ngx\\_http\\_header\\_val\\_t](#)
- [ngx\\_http\\_headers\\_conf\\_t](#)
- [ngx\\_http\\_set\\_header\\_pt](#)
- [ngx\\_http\\_set\\_header\\_t](#)

### Functions defined

- [ngx\\_http\\_add\\_cache\\_control](#)
- [ngx\\_http\\_add\\_header](#)
- [ngx\\_http\\_headers\\_add](#)
- [ngx\\_http\\_headers\\_create\\_conf](#)
- [ngx\\_http\\_headers\\_expires](#)
- [ngx\\_http\\_headers\\_filter](#)
- [ngx\\_http\\_headers\\_filter\\_init](#)
- [ngx\\_http\\_headers\\_merge\\_conf](#)
- [ngx\\_http\\_parse\\_expires](#)
- [ngx\\_http\\_set\\_expires](#)
- [ngx\\_http\\_set\\_last\\_modified](#)
- [ngx\\_http\\_set\\_response\\_header](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
```

```

6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 typedef struct ngx_http_header_val_s ngx_http_header_val_t;
14
15 typedef ngx_int_t (*ngx_http_set_header_pt)(ngx_http_request_t *r,
16     ngx_http_header_val_t *hv, ngx_str_t *value);
17
18
19 typedef struct {
20     ngx_str_t          name;
21     ngx_uint_t        offset;
22     ngx_http_set_header_pt handler;
23 } ngx_http_set_header_t;
24
25
26 struct ngx_http_header_val_s {
27     ngx_http_complex_value_t value;
28     ngx_str_t                key;
29     ngx_http_set_header_pt  handler;
30     ngx_uint_t              offset;
31     ngx_uint_t              always; /* unsigned always:1 */
32 };
33
34
35 typedef enum {
36     NGX_HTTP_EXPIRES_OFF,
37     NGX_HTTP_EXPIRES_EPOCH,
38     NGX_HTTP_EXPIRES_MAX,
39     NGX_HTTP_EXPIRES_ACCESS,
40     NGX_HTTP_EXPIRES_MODIFIED,
41     NGX_HTTP_EXPIRES_DAILY,
42     NGX_HTTP_EXPIRES_UNSET
43 } ngx_http_expires_t;
44
45
46 typedef struct {
47     ngx_http_expires_t expires;
48     time_t             expires_time;
49     ngx_http_complex_value_t *expires_value;
50     ngx_array_t        *headers;
51 } ngx_http_headers_conf_t;
52
53
54 static ngx_int_t ngx_http_set_expires(ngx_http_request_t *r,
55     ngx_http_headers_conf_t *conf);
56 static ngx_int_t ngx_http_parse_expires(ngx_str_t *value,
57     ngx_http_expires_t *expires, time_t *expires_time, char **err);
58 static ngx_int_t ngx_http_add_cache_control(ngx_http_request_t *r,
59     ngx_http_header_val_t *hv, ngx_str_t *value);
60 static ngx_int_t ngx_http_add_header(ngx_http_request_t *r,
61     ngx_http_header_val_t *hv, ngx_str_t *value);
62 static ngx_int_t ngx_http_set_last_modified(ngx_http_request_t *r,
63     ngx_http_header_val_t *hv, ngx_str_t *value);
64 static ngx_int_t ngx_http_set_response_header(ngx_http_request_t *r,
65     ngx_http_header_val_t *hv, ngx_str_t *value);
66
67 static void *ngx_http_headers_create_conf(ngx_conf_t *cf);
68 static char *ngx_http_headers_merge_conf(ngx_conf_t *cf,
69     void *parent, void *child);
70 static ngx_int_t ngx_http_headers_filter_init(ngx_conf_t *cf);
71 static char *ngx_http_headers_expires(ngx_conf_t *cf, ngx_command_t *cmd,
72     void *conf);
73 static char *ngx_http_headers_add(ngx_conf_t *cf, ngx_command_t *cmd,
74     void *conf);
75
76
77 static ngx_http_set_header_t ngx_http_set_headers[] = {
78
79     { ngx_string("Cache-Control"), 0, ngx_http_add_cache_control },
80
81     { ngx_string("Last-Modified"),

```

```

82         offsetof(ngx_http_headers_out_t, last_modified),
83         ngx_http_set_last_modified },
84
85     { ngx_string("ETag"),
86       offsetof(ngx_http_headers_out_t, etag),
87       ngx_http_set_response_header },
88
89     { ngx_null_string, 0, NULL }
90 };
91
92
93 static ngx_command_t ngx_http_headers_filter_commands[] = {
94
95     { ngx_string("expires"),
96       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_HTTP_LIF_CONF
97       |NGX_CONF_TAKE12,
98       ngx_http_headers_expires,
99       NGX_HTTP_LOC_CONF_OFFSET,
100      0,
101      NULL},
102
103     { ngx_string("add_header"),
104       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_HTTP_LIF_CONF
105       |NGX_CONF_TAKE23,
106       ngx_http_headers_add,
107       NGX_HTTP_LOC_CONF_OFFSET,
108       0,
109       NULL},
110
111     ngx_null_command
112 };
113
114
115 static ngx_http_module_t ngx_http_headers_filter_module_ctx = {
116     NULL,                                     /* preconfiguration */
117     ngx_http_headers_filter_init,           /* postconfiguration */
118
119     NULL,                                     /* create main configuration */
120     NULL,                                     /* init main configuration */
121
122     NULL,                                     /* create server configuration */
123     NULL,                                     /* merge server configuration */
124
125     ngx_http_headers_create_conf,          /* create location configuration */
126     ngx_http_headers_merge_conf,          /* merge location configuration */
127 };
128
129
130 ngx_module_t ngx_http_headers_filter_module = {
131     NGX_MODULE_V1,
132     &ngx_http_headers_filter_module_ctx,    /* module context */
133     ngx_http_headers_filter_commands,      /* module directives */
134     NGX_HTTP_MODULE,                      /* module type */
135     NULL,                                    /* init master */
136     NULL,                                    /* init module */
137     NULL,                                    /* init process */
138     NULL,                                    /* init thread */
139     NULL,                                    /* exit thread */
140     NULL,                                    /* exit process */
141     NULL,                                    /* exit master */
142     NGX_MODULE_V1_PADDING
143 };
144
145
146 static ngx_http_output_header_filter_pt ngx_http_next_header_filter;
147
148
149 static ngx_int_t
150 ngx_http_headers_filter(ngx_http_request_t *r)
151 {
152     ngx_str_t          value;
153     ngx_uint_t         i, safe_status;
154     ngx_http_header_val_t *h;
155     ngx_http_headers_conf_t *conf;
156
157     conf = ngx_http_get_module_loc_conf(r, ngx_http_headers_filter_module);

```

```

158     if ((conf->expires == NGX_HTTP_EXPIRES_OFF && conf->headers == NULL)
159         || r != r->main)
160     {
161         return ngx_http_next_header_filter(r);
162     }
163
164     switch (r->headers_out.status) {
165
166     case NGX_HTTP_OK:
167     case NGX_HTTP_CREATED:
168     case NGX_HTTP_NO_CONTENT:
169     case NGX_HTTP_PARTIAL_CONTENT:
170     case NGX_HTTP_MOVED_PERMANENTLY:
171     case NGX_HTTP_MOVED_TEMPORARILY:
172     case NGX_HTTP_SEE_OTHER:
173     case NGX_HTTP_NOT_MODIFIED:
174     case NGX_HTTP_TEMPORARY_REDIRECT:
175         safe_status = 1;
176         break;
177
178     default:
179         safe_status = 0;
180         break;
181     }
182
183     if (conf->expires != NGX_HTTP_EXPIRES_OFF && safe_status) {
184         if (ngx_http_set_expires(r, conf) != NGX_OK) {
185             return NGX_ERROR;
186         }
187     }
188
189     if (conf->headers) {
190         h = conf->headers->elts;
191         for (i = 0; i < conf->headers->nelts; i++) {
192
193             if (!safe_status && !h[i].always) {
194                 continue;
195             }
196
197             if (ngx_http_complex_value(r, &h[i].value, &value) != NGX_OK) {
198                 return NGX_ERROR;
199             }
200
201             if (h[i].handler(r, &h[i], &value) != NGX_OK) {
202                 return NGX_ERROR;
203             }
204         }
205     }
206
207     return ngx_http_next_header_filter(r);
208 }
209
210
211 static ngx_int_t
212 ngx_http_set_expires(ngx_http_request_t *r, ngx_http_headers_conf_t *conf)
213 {
214     char                *err;
215     size_t              len;
216     time_t              now, expires_time, max_age;
217     ngx_str_t          value;
218     ngx_int_t          rc;
219     ngx_uint_t         i;
220     ngx_table_elt_t    *e, *cc, **ccp;
221     ngx_http_expires_t expires;
222
223     expires = conf->expires;
224     expires_time = conf->expires_time;
225
226     if (conf->expires_value != NULL) {
227         if (ngx_http_complex_value(r, conf->expires_value, &value) != NGX_OK) {
228             return NGX_ERROR;
229         }
230     }
231
232     rc = ngx_http_parse_expires(&value, &expires, &expires_time, &err);

```

```

234     if (rc != NGX\_OK) {
235         return NGX\_OK;
236     }
237
238
239     if (expires == NGX_HTTP_EXPIRES_OFF) {
240         return NGX\_OK;
241     }
242 }
243
244 e = r->headers_out.expires;
245
246 if (e == NULL) {
247
248     e = ngx\_list\_push(&r->headers_out.headers);
249     if (e == NULL) {
250         return NGX\_ERROR;
251     }
252
253     r->headers_out.expires = e;
254
255     e->hash = 1;
256     ngx\_str\_set(&e->key, "Expires");
257 }
258
259 len = sizeof("Mon, 28 Sep 1970 06:00:00 GMT");
260 e->value.len = len - 1;
261
262 ccp = r->headers_out.cache_control.elts;
263
264 if (ccp == NULL) {
265
266     if (ngx\_array\_init(&r->headers_out.cache_control, r->pool,
267         1, sizeof(ngx\_table\_elt\_t *)))
268         != NGX\_OK)
269     {
270         return NGX\_ERROR;
271     }
272
273     ccp = ngx\_array\_push(&r->headers_out.cache_control);
274     if (ccp == NULL) {
275         return NGX\_ERROR;
276     }
277
278     cc = ngx\_list\_push(&r->headers_out.headers);
279     if (cc == NULL) {
280         return NGX\_ERROR;
281     }
282
283     cc->hash = 1;
284     ngx\_str\_set(&cc->key, "Cache-Control");
285     *ccp = cc;
286
287 } else {
288     for (i = 1; i < r->headers_out.cache_control.nelts; i++) {
289         ccp[i]->hash = 0;
290     }
291
292     cc = ccp[0];
293 }
294
295 if (expires == NGX_HTTP_EXPIRES_EPOCH) {
296     e->value.data = (u_char *) "Thu, 01 Jan 1970 00:00:01 GMT";
297     ngx\_str\_set(&cc->value, "no-cache");
298     return NGX\_OK;
299 }
300
301 if (expires == NGX_HTTP_EXPIRES_MAX) {
302     e->value.data = (u_char *) "Thu, 31 Dec 2037 23:55:55 GMT";
303     /* 10 years */
304     ngx\_str\_set(&cc->value, "max-age=315360000");
305     return NGX\_OK;
306 }
307
308 e->value.data = ngx\_pnalloc(r->pool, len);
309 if (e->value.data == NULL) {

```

```

310     return NGX\_ERROR;
311 }
312
313 if (expires_time == 0 && expires != NGX_HTTP_EXPIRES_DAILY) {
314     ngx\_memcpy(e->value.data, ngx\_cached\_http\_time.data,
315               ngx\_cached\_http\_time.len + 1);
316     ngx\_str\_set(&cc->value, "max-age=0");
317     return NGX\_OK;
318 }
319
320 now = ngx\_time();
321
322 if (expires == NGX_HTTP_EXPIRES_DAILY) {
323     expires_time = ngx\_next\_time(expires_time);
324     max_age = expires_time - now;
325 }
326 else if (expires == NGX_HTTP_EXPIRES_ACCESS
327         || r->headers_out.last_modified_time == -1)
328 {
329     max_age = expires_time;
330     expires_time += now;
331 }
332 else {
333     expires_time += r->headers_out.last_modified_time;
334     max_age = expires_time - now;
335 }
336
337 ngx\_http\_time(e->value.data, expires_time);
338
339 if (conf->expires_time < 0 || max_age < 0) {
340     ngx\_str\_set(&cc->value, "no-cache");
341     return NGX\_OK;
342 }
343
344 cc->value.data = ngx\_pnalloc(r->pool,
345                             sizeof("max-age=") + NGX_TIME_T_LEN + 1);
346 if (cc->value.data == NULL) {
347     return NGX\_ERROR;
348 }
349
350 cc->value.len = ngx\_sprintf(cc->value.data, "max-age=%T", max_age)
351                   - cc->value.data;
352
353 return NGX\_OK;
354 }
355
356
357 static ngx\_int\_t
358 ngx\_http\_parse\_expires(ngx\_str\_t *value, ngx\_http\_expires\_t *expires,
359                       time\_t *expires_time, char **err)
360 {
361     ngx\_uint\_t minus;
362
363     if (*expires != NGX_HTTP_EXPIRES_MODIFIED) {
364
365         if (value->len == 5 && ngx\_strncmp(value->data, "epoch", 5) == 0) {
366             *expires = NGX_HTTP_EXPIRES_EPOCH;
367             return NGX\_OK;
368         }
369
370         if (value->len == 3 && ngx\_strncmp(value->data, "max", 3) == 0) {
371             *expires = NGX_HTTP_EXPIRES_MAX;
372             return NGX\_OK;
373         }
374
375         if (value->len == 3 && ngx\_strncmp(value->data, "off", 3) == 0) {
376             *expires = NGX_HTTP_EXPIRES_OFF;
377             return NGX\_OK;
378         }
379     }
380
381     if (value->data[0] == '@') {
382         value->data++;
383         value->len--;
384         minus = 0;
385     }

```

```

386     if (*expires == NGX_HTTP_EXPIRES_MODIFIED) {
387         *err = "daily time cannot be used with \"modified\" parameter";
388         return NGX_ERROR;
389     }
390
391     *expires = NGX_HTTP_EXPIRES_DAILY;
392
393 } else if (value->data[0] == '+') {
394     value->data++;
395     value->len--;
396     minus = 0;
397
398 } else if (value->data[0] == '-') {
399     value->data++;
400     value->len--;
401     minus = 1;
402
403 } else {
404     minus = 0;
405 }
406
407 *expires_time = ngx_parse_time(value, 1);
408
409 if (*expires_time == (time_t) NGX_ERROR) {
410     *err = "invalid value";
411     return NGX_ERROR;
412 }
413
414 if (*expires == NGX_HTTP_EXPIRES_DAILY
415     && *expires_time > 24 * 60 * 60)
416 {
417     *err = "daily time value must be less than 24 hours";
418     return NGX_ERROR;
419 }
420
421 if (minus) {
422     *expires_time = - *expires_time;
423 }
424
425 return NGX_OK;
426 }
427
428
429 static ngx_int_t
430 ngx_http_add_header(ngx_http_request_t *r, ngx_http_header_val_t *hv,
431     ngx_str_t *value)
432 {
433     ngx_table_elt_t *h;
434
435     if (value->len) {
436         h = ngx_list_push(&r->headers_out.headers);
437         if (h == NULL) {
438             return NGX_ERROR;
439         }
440
441         h->hash = 1;
442         h->key = hv->key;
443         h->value = *value;
444     }
445
446     return NGX_OK;
447 }
448
449
450 static ngx_int_t
451 ngx_http_add_cache_control(ngx_http_request_t *r, ngx_http_header_val_t *hv,
452     ngx_str_t *value)
453 {
454     ngx_table_elt_t *cc, **ccp;
455
456     if (value->len == 0) {
457         return NGX_OK;
458     }
459
460     ccp = r->headers_out.cache_control.elts;
461

```



```

462     if (ccp == NULL) {
463
464         if (ngx\_array\_init(&r->headers_out.cache_control, r->pool,
465             1, sizeof(ngx\_table\_elt\_t *)))
466             != NGX\_OK)
467         {
468             return NGX\_ERROR;
469         }
470     }
471
472     ccp = ngx\_array\_push(&r->headers_out.cache_control);
473     if (ccp == NULL) {
474         return NGX\_ERROR;
475     }
476
477     cc = ngx\_list\_push(&r->headers_out.headers);
478     if (cc == NULL) {
479         return NGX\_ERROR;
480     }
481
482     cc->hash = 1;
483     ngx\_str\_set(&cc->key, "Cache-Control");
484     cc->value = *value;
485
486     *ccp = cc;
487
488     return NGX\_OK;
489 }
490
491
492 static ngx\_int\_t
493 ngx\_http\_set\_last\_modified(ngx\_http\_request\_t *r, ngx\_http\_header\_val\_t *hv,
494     ngx\_str\_t *value)
495 {
496     if (ngx\_http\_set\_response\_header(r, hv, value) != NGX\_OK) {
497         return NGX\_ERROR;
498     }
499
500     r->headers_out.last_modified_time =
501         (value->len) ? ngx\_http\_parse\_time(value->data, value->len) : -1;
502
503     return NGX\_OK;
504 }
505
506
507 static ngx\_int\_t
508 ngx\_http\_set\_response\_header(ngx\_http\_request\_t *r, ngx\_http\_header\_val\_t *hv,
509     ngx\_str\_t *value)
510 {
511     ngx\_table\_elt\_t *h, **old;
512
513     old = (ngx\_table\_elt\_t **) ((char *) &r->headers_out + hv->offset);
514
515     if (value->len == 0) {
516         if (*old) {
517             (*old)->hash = 0;
518             *old = NULL;
519         }
520
521         return NGX\_OK;
522     }
523
524     if (*old) {
525         h = *old;
526
527     } else {
528         h = ngx\_list\_push(&r->headers_out.headers);
529         if (h == NULL) {
530             return NGX\_ERROR;
531         }
532
533         *old = h;
534     }
535
536     h->hash = 1;
537     h->key = hv->key;

```

```

538     h->value = *value;
539
540     return NGX\_OK;
541 }
542
543
544 static void *
545 ngx_http_headers_create_conf(ngx\_conf\_t *cf)
546 {
547     ngx\_http\_headers\_conf\_t *conf;
548
549     conf = ngx\_palloc(cf->pool, sizeof(ngx\_http\_headers\_conf\_t));
550     if (conf == NULL) {
551         return NULL;
552     }
553
554     /*
555      * set by ngx\_palloc\(\):
556      *
557      *     conf->headers = NULL;
558      *     conf->expires_time = 0;
559      *     conf->expires_value = NULL;
560      */
561
562     conf->expires = NGX_HTTP_EXPIRES_UNSET;
563
564     return conf;
565 }
566
567
568 static char *
569 ngx_http_headers_merge_conf(ngx\_conf\_t *cf, void *parent, void *child)
570 {
571     ngx\_http\_headers\_conf\_t *prev = parent;
572     ngx\_http\_headers\_conf\_t *conf = child;
573
574     if (conf->expires == NGX_HTTP_EXPIRES_UNSET) {
575         conf->expires = prev->expires;
576         conf->expires_time = prev->expires_time;
577         conf->expires_value = prev->expires_value;
578
579         if (conf->expires == NGX_HTTP_EXPIRES_UNSET) {
580             conf->expires = NGX_HTTP_EXPIRES_OFF;
581         }
582     }
583
584     if (conf->headers == NULL) {
585         conf->headers = prev->headers;
586     }
587
588     return NGX\_CONF\_OK;
589 }
590
591
592 static ngx\_int\_t
593 ngx_http_headers_filter_init(ngx\_conf\_t *cf)
594 {
595     ngx\_http\_next\_header\_filter = ngx\_http\_top\_header\_filter;
596     ngx\_http\_top\_header\_filter = ngx\_http\_headers\_filter;
597
598     return NGX\_OK;
599 }
600
601
602 static char *
603 ngx_http_headers_expires(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
604 {
605     ngx\_http\_headers\_conf\_t *hcf = conf;
606
607     char                                *err;
608     ngx\_str\_t                          *value;
609     ngx\_int\_t                          rc;
610     ngx\_uint\_t                          n;
611     ngx\_http\_complex\_value\_t           cv;
612     ngx\_http\_compile\_complex\_value\_t   ccv;
613

```

```

614     if (hcf->expires != NGX_HTTP_EXPIRES_UNSET) {
615         return "is duplicate";
616     }
617
618     value = cf->args->elts;
619
620     if (cf->args->nelts == 2) {
621         hcf->expires = NGX_HTTP_EXPIRES_ACCESS;
622
623         n = 1;
624     } else { /* cf->args->nelts == 3 */
625
626         if (ngx_strcmp(value[1].data, "modified") != 0) {
627             return "invalid value";
628         }
629
630         hcf->expires = NGX_HTTP_EXPIRES_MODIFIED;
631
632         n = 2;
633     }
634
635     ngx_memzero(&ccv, sizeof(ngx_http_compile_complex_value_t));
636
637     ccv.cf = cf;
638     ccv.value = &value[n];
639     ccv.complex_value = &cv;
640
641     if (ngx_http_compile_complex_value(&ccv) != NGX_OK) {
642         return NGX_CONF_ERROR;
643     }
644
645     if (cv.lengths != NULL) {
646
647         hcf->expires_value = ngx_palloc(cf->pool,
648                                         sizeof(ngx_http_complex_value_t));
649         if (hcf->expires_value == NULL) {
650             return NGX_CONF_ERROR;
651         }
652
653         *hcf->expires_value = cv;
654
655         return NGX_CONF_OK;
656     }
657
658     rc = ngx_http_parse_expires(&value[n], &hcf->expires, &hcf->expires_time,
659                                &err);
660
661     if (rc != NGX_OK) {
662         return err;
663     }
664
665     return NGX_CONF_OK;
666 }
667
668
669 static char *
670 ngx_http_headers_add(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
671 {
672     ngx_http_headers_conf_t *hcf = conf;
673
674     ngx_str_t          *value;
675     ngx_uint_t         i;
676     ngx_http_header_val_t *hv;
677     ngx_http_set_header_t *set;
678     ngx_http_compile_complex_value_t ccv;
679
680     value = cf->args->elts;
681
682     if (hcf->headers == NULL) {
683         hcf->headers = ngx_array_create(cf->pool, 1,
684                                       sizeof(ngx_http_header_val_t));
685
686         if (hcf->headers == NULL) {
687             return NGX_CONF_ERROR;
688         }
689     }

```

```

690     hv = ngx_array_push(hcf->headers);
691     if (hv == NULL) {
692         return NGX_CONF_ERROR;
693     }
694
695     hv->key = value[1];
696     hv->handler = ngx_http_add_header;
697     hv->offset = 0;
698     hv->always = 0;
699
700     set = ngx_http_set_headers;
701     for (i = 0; set[i].name.len; i++) {
702         if (ngx_strcasecmp(value[1].data, set[i].name.data) != 0) {
703             continue;
704         }
705
706         hv->offset = set[i].offset;
707         hv->handler = set[i].handler;
708
709         break;
710     }
711
712     if (value[2].len == 0) {
713         ngx_memzero(&hv->value, sizeof(ngx_http_complex_value_t));
714         return NGX_CONF_OK;
715     }
716
717     ngx_memzero(&ccv, sizeof(ngx_http_compile_complex_value_t));
718
719     ccv.cf = cf;
720     ccv.value = &value[2];
721     ccv.complex_value = &hv->value;
722
723     if (ngx_http_compile_complex_value(&ccv) != NGX_OK) {
724         return NGX_CONF_ERROR;
725     }
726
727     if (cf->args->nelts == 3) {
728         return NGX_CONF_OK;
729     }
730
731     if (ngx_strcmp(value[3].data, "always") != 0) {
732         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
733             "invalid parameter \"%V\"", &value[3]);
734         return NGX_CONF_ERROR;
735     }
736
737     hv->always = 1;
738
739     return NGX_CONF_OK;
740 }

```

[One Level Up](#)

[Top Level](#)

## src/http/modules/nginx\_http\_image\_filter\_module.c - nginx-1.7.10

### Global variables defined

- [ngx\\_http\\_image\\_filter\\_commands](#)
- [ngx\\_http\\_image\\_filter\\_module](#)
- [ngx\\_http\\_image\\_filter\\_module\\_ctx](#)
- [ngx\\_http\\_image\\_types](#)
- [ngx\\_http\\_next\\_body\\_filter](#)
- [ngx\\_http\\_next\\_header\\_filter](#)

### Data types defined

- [ngx\\_http\\_image\\_filter\\_conf\\_t](#)
- [ngx\\_http\\_image\\_filter\\_ctx\\_t](#)

### Functions defined

- [ngx\\_http\\_image\\_asis](#)
- [ngx\\_http\\_image\\_body\\_filter](#)
- [ngx\\_http\\_image\\_cleanup](#)
- [ngx\\_http\\_image\\_filter](#)
- [ngx\\_http\\_image\\_filter\\_create\\_conf](#)
- [ngx\\_http\\_image\\_filter\\_get\\_value](#)
- [ngx\\_http\\_image\\_filter\\_init](#)
- [ngx\\_http\\_image\\_filter\\_jpeg\\_quality](#)
- [ngx\\_http\\_image\\_filter\\_merge\\_conf](#)
- [ngx\\_http\\_image\\_filter\\_sharpen](#)
- [ngx\\_http\\_image\\_filter\\_value](#)
- [ngx\\_http\\_image\\_header\\_filter](#)
- [ngx\\_http\\_image\\_json](#)
- [ngx\\_http\\_image\\_length](#)
- [ngx\\_http\\_image\\_new](#)
- [ngx\\_http\\_image\\_out](#)
- [ngx\\_http\\_image\\_process](#)
- [ngx\\_http\\_image\\_read](#)
- [ngx\\_http\\_image\\_resize](#)

- [ngx\\_http\\_image\\_send](#)
- [ngx\\_http\\_image\\_size](#)
- [ngx\\_http\\_image\\_source](#)
- [ngx\\_http\\_image\\_test](#)

## Macros defined

- [NGX\\_HTTP\\_IMAGE\\_BUFFERED](#)
- [NGX\\_HTTP\\_IMAGE\\_CROP](#)
- [NGX\\_HTTP\\_IMAGE\\_DONE](#)
- [NGX\\_HTTP\\_IMAGE\\_GIF](#)
- [NGX\\_HTTP\\_IMAGE\\_JPEG](#)
- [NGX\\_HTTP\\_IMAGE\\_NONE](#)
- [NGX\\_HTTP\\_IMAGE\\_OFF](#)
- [NGX\\_HTTP\\_IMAGE\\_PASS](#)
- [NGX\\_HTTP\\_IMAGE\\_PNG](#)
- [NGX\\_HTTP\\_IMAGE\\_PROCESS](#)
- [NGX\\_HTTP\\_IMAGE\\_READ](#)
- [NGX\\_HTTP\\_IMAGE\\_RESIZE](#)
- [NGX\\_HTTP\\_IMAGE\\_ROTATE](#)
- [NGX\\_HTTP\\_IMAGE\\_SIZE](#)
- [NGX\\_HTTP\\_IMAGE\\_START](#)
- [NGX\\_HTTP\\_IMAGE\\_TEST](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12 #include <gd.h>
13
14
15 #define NGX_HTTP_IMAGE_OFF          0
16 #define NGX_HTTP_IMAGE_TEST        1
17 #define NGX_HTTP_IMAGE_SIZE        2
18 #define NGX_HTTP_IMAGE_RESIZE      3
19 #define NGX_HTTP_IMAGE_CROP        4
20 #define NGX_HTTP_IMAGE_ROTATE      5
21
22
23 #define NGX_HTTP_IMAGE_START        0
```

```

24 #define NGX_HTTP_IMAGE_READ 1
25 #define NGX_HTTP_IMAGE_PROCESS 2
26 #define NGX_HTTP_IMAGE_PASS 3
27 #define NGX_HTTP_IMAGE_DONE 4
28
29
30 #define NGX_HTTP_IMAGE_NONE 0
31 #define NGX_HTTP_IMAGE_JPEG 1
32 #define NGX_HTTP_IMAGE_GIF 2
33 #define NGX_HTTP_IMAGE_PNG 3
34
35
36 #define NGX_HTTP_IMAGE_BUFFERED 0x08
37
38
39 typedef struct {
40     ngx_uint_t filter;
41     ngx_uint_t width;
42     ngx_uint_t height;
43     ngx_uint_t angle;
44     ngx_uint_t jpeg_quality;
45     ngx_uint_t sharpen;
46
47     ngx_flag_t transparency;
48     ngx_flag_t interlace;
49
50     ngx_http_complex_value_t *wcv;
51     ngx_http_complex_value_t *hcv;
52     ngx_http_complex_value_t *acv;
53     ngx_http_complex_value_t *jqcv;
54     ngx_http_complex_value_t *shcv;
55
56     size_t buffer_size;
57 } ngx_http_image_filter_conf_t;
58
59
60 typedef struct {
61     u_char *image;
62     u_char *last;
63
64     size_t length;
65
66     ngx_uint_t width;
67     ngx_uint_t height;
68     ngx_uint_t max_width;
69     ngx_uint_t max_height;
70     ngx_uint_t angle;
71
72     ngx_uint_t phase;
73     ngx_uint_t type;
74     ngx_uint_t force;
75 } ngx_http_image_filter_ctx_t;
76
77
78 static ngx_int_t ngx_http_image_send(ngx_http_request_t *r,
79     ngx_http_image_filter_ctx_t *ctx, ngx_chain_t *in);
80 static ngx_uint_t ngx_http_image_test(ngx_http_request_t *r, ngx_chain_t *in);
81 static ngx_int_t ngx_http_image_read(ngx_http_request_t *r, ngx_chain_t *in);
82 static ngx_buf_t *ngx_http_image_process(ngx_http_request_t *r);
83 static ngx_buf_t *ngx_http_image_json(ngx_http_request_t *r,
84     ngx_http_image_filter_ctx_t *ctx);
85 static ngx_buf_t *ngx_http_image_asis(ngx_http_request_t *r,
86     ngx_http_image_filter_ctx_t *ctx);
87 static void ngx_http_image_length(ngx_http_request_t *r, ngx_buf_t *b);
88 static ngx_int_t ngx_http_image_size(ngx_http_request_t *r,
89     ngx_http_image_filter_ctx_t *ctx);
90
91 static ngx_buf_t *ngx_http_image_resize(ngx_http_request_t *r,
92     ngx_http_image_filter_ctx_t *ctx);
93 static gdImagePtr ngx_http_image_source(ngx_http_request_t *r,
94     ngx_http_image_filter_ctx_t *ctx);
95 static gdImagePtr ngx_http_image_new(ngx_http_request_t *r, int w, int h,
96     int colors);
97 static u_char *ngx_http_image_out(ngx_http_request_t *r, ngx_uint_t type,
98     gdImagePtr img, int *size);
99 static void ngx_http_image_cleanup(void *data);

```

```

100 static ngx_uint_t ngx_http_image_filter_get_value(ngx_http_request_t *r,
101     ngx_http_complex_value_t *cv, ngx_uint_t v);
102 static ngx_uint_t ngx_http_image_filter_value(ngx_str_t *value);
103
104
105 static void *ngx_http_image_filter_create_conf(ngx_conf_t *cf);
106 static char *ngx_http_image_filter_merge_conf(ngx_conf_t *cf, void *parent,
107     void *child);
108 static char *ngx_http_image_filter(ngx_conf_t *cf, ngx_command_t *cmd,
109     void *conf);
110 static char *ngx_http_image_filter_jpeg_quality(ngx_conf_t *cf,
111     ngx_command_t *cmd, void *conf);
112 static char *ngx_http_image_filter_sharpen(ngx_conf_t *cf, ngx_command_t *cmd,
113     void *conf);
114 static ngx_int_t ngx_http_image_filter_init(ngx_conf_t *cf);
115
116
117 static ngx_command_t  ngx_http_image_filter_commands[] = {
118
119     { ngx_string("image_filter"),
120       NGX_HTTP_LOC_CONF|NGX_CONF_TAKE123,
121       ngx_http_image_filter,
122       NGX_HTTP_LOC_CONF_OFFSET,
123       0,
124       NULL },
125
126     { ngx_string("image_filter_jpeg_quality"),
127       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
128       ngx_http_image_filter_jpeg_quality,
129       NGX_HTTP_LOC_CONF_OFFSET,
130       0,
131       NULL },
132
133     { ngx_string("image_filter_sharpen"),
134       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
135       ngx_http_image_filter_sharpen,
136       NGX_HTTP_LOC_CONF_OFFSET,
137       0,
138       NULL },
139
140     { ngx_string("image_filter_transparency"),
141       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
142       ngx_conf_set_flag_slot,
143       NGX_HTTP_LOC_CONF_OFFSET,
144       offsetof(ngx_http_image_filter_conf_t, transparency),
145       NULL },
146
147     { ngx_string("image_filter_interlace"),
148       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
149       ngx_conf_set_flag_slot,
150       NGX_HTTP_LOC_CONF_OFFSET,
151       offsetof(ngx_http_image_filter_conf_t, interlace),
152       NULL },
153
154     { ngx_string("image_filter_buffer"),
155       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
156       ngx_conf_set_size_slot,
157       NGX_HTTP_LOC_CONF_OFFSET,
158       offsetof(ngx_http_image_filter_conf_t, buffer_size),
159       NULL },
160
161     ngx_null_command
162 };
163
164
165 static ngx_http_module_t  ngx_http_image_filter_module_ctx = {
166     NULL,                                     /* preconfiguration */
167     ngx_http_image_filter_init,               /* postconfiguration */
168
169     NULL,                                     /* create main configuration */
170     NULL,                                     /* init main configuration */
171
172     NULL,                                     /* create server configuration */
173     NULL,                                     /* merge server configuration */
174
175     ngx_http_image_filter_create_conf,        /* create location configuration */

```



```

176 ngx\_http\_image\_filter\_merge\_conf      /* merge location configuration */
177 };
178
179
180 ngx\_module\_t ngx\_http\_image\_filter\_module = {
181     NGX\_MODULE\_V1,
182     &ngx\_http\_image\_filter\_module\_ctx,    /* module context */
183     ngx\_http\_image\_filter\_commands,    /* module directives */
184     NGX\_HTTP\_MODULE,                    /* module type */
185     NULL,                                /* init master */
186     NULL,                                /* init module */
187     NULL,                                /* init process */
188     NULL,                                /* init thread */
189     NULL,                                /* exit thread */
190     NULL,                                /* exit process */
191     NULL,                                /* exit master */
192     NGX\_MODULE\_V1\_PADDING
193 };
194
195
196 static ngx\_http\_output\_header\_filter\_pt ngx\_http\_next\_header\_filter;
197 static ngx\_http\_output\_body\_filter\_pt ngx\_http\_next\_body\_filter;
198
199
200 static ngx\_str\_t ngx\_http\_image\_types[] = {
201     ngx\_string("image/jpeg"),
202     ngx\_string("image/gif"),
203     ngx\_string("image/png")
204 };
205
206
207 static ngx\_int\_t
208 ngx\_http\_image\_header\_filter(ngx\_http\_request\_t *r)
209 {
210     off\_t len;
211     ngx\_http\_image\_filter\_ctx\_t *ctx;
212     ngx\_http\_image\_filter\_conf\_t *conf;
213
214     if (r->headers_out.status == NGX\_HTTP\_NOT\_MODIFIED) {
215         return ngx\_http\_next\_header\_filter(r);
216     }
217
218     ctx = ngx\_http\_get\_module\_ctx(r, ngx\_http\_image\_filter\_module);
219
220     if (ctx) {
221         ngx\_http\_set\_ctx(r, NULL, ngx\_http\_image\_filter\_module);
222         return ngx\_http\_next\_header\_filter(r);
223     }
224
225     conf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_image\_filter\_module);
226
227     if (conf->filter == NGX\_HTTP\_IMAGE\_OFF) {
228         return ngx\_http\_next\_header\_filter(r);
229     }
230
231     if (r->headers_out.content_type.len
232         >= sizeof("multipart/x-mixed-replace") - 1
233         && ngx\_strncasecmp(r->headers_out.content_type.data,
234                          (u_char *) "multipart/x-mixed-replace",
235                          sizeof("multipart/x-mixed-replace") - 1)
236         == 0)
237     {
238         ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, 0,
239                     "image filter: multipart/x-mixed-replace response");
240
241         return NGX\_ERROR;
242     }
243
244     ctx = ngx\_palloc(r->pool, sizeof(ngx\_http\_image\_filter\_ctx\_t));
245     if (ctx == NULL) {
246         return NGX\_ERROR;
247     }
248
249     ngx\_http\_set\_ctx(r, ctx, ngx\_http\_image\_filter\_module);
250
251     len = r->headers_out.content_length_n;

```

```

252
253 if (len != -1 && len > (off_t) conf->buffer_size){
254     ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
255         "image filter: too big response: %0", len);
256
257     return NGX_HTTP_UNSUPPORTED_MEDIA_TYPE;
258 }
259
260 if (len == -1) {
261     ctx->length = conf->buffer_size;
262
263 } else {
264     ctx->length = (size_t) len;
265 }
266
267 if (r->headers_out.refresh) {
268     r->headers_out.refresh->hash = 0;
269 }
270
271 r->main_filter_need_in_memory = 1;
272 r->allow_ranges = 0;
273
274 return NGX_OK;
275 }
276
277
278 static ngx_int_t
279 ngx_http_image_body_filter(ngx_http_request_t *r, ngx_chain_t *in)
280 {
281     ngx_int_t          rc;
282     ngx_str_t          *ct;
283     ngx_chain_t        out;
284     ngx_http_image_filter_ctx_t *ctx;
285     ngx_http_image_filter_conf_t *conf;
286
287     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0, "image filter");
288
289     if (in == NULL) {
290         return ngx_http_next_body_filter(r, in);
291     }
292
293     ctx = ngx_http_get_module_ctx(r, ngx_http_image_filter_module);
294
295     if (ctx == NULL) {
296         return ngx_http_next_body_filter(r, in);
297     }
298
299     switch (ctx->phase) {
300
301     case NGX_HTTP_IMAGE_START:
302
303         ctx->type = ngx_http_image_test(r, in);
304
305         conf = ngx_http_get_module_loc_conf(r, ngx_http_image_filter_module);
306
307         if (ctx->type == NGX_HTTP_IMAGE_NONE) {
308
309             if (conf->filter == NGX_HTTP_IMAGE_SIZE) {
310                 out.buf = ngx_http_image_json(r, NULL);
311
312                 if (out.buf) {
313                     out.next = NULL;
314                     ctx->phase = NGX_HTTP_IMAGE_DONE;
315
316                     return ngx_http_image_send(r, ctx, &out);
317                 }
318             }
319
320             return ngx_http_filter_finalize_request(r,
321                 &ngx_http_image_filter_module,
322                 NGX_HTTP_UNSUPPORTED_MEDIA_TYPE);
323         }
324
325         /* override content type */
326
327         ct = &ngx_http_image_types[ctx->type - 1];

```

```

328     r->headers_out.content_type_len = ct->len;
329     r->headers_out.content_type = *ct;
330     r->headers_out.content_type_lowcase = NULL;
331
332     if (conf->filter == NGX\_HTTP\_IMAGE\_TEST) {
333         ctx->phase = NGX\_HTTP\_IMAGE\_PASS;
334
335         return ngx\_http\_image\_send(r, ctx, in);
336     }
337
338     ctx->phase = NGX\_HTTP\_IMAGE\_READ;
339
340     /* fall through */
341
342     case NGX\_HTTP\_IMAGE\_READ:
343
344         rc = ngx\_http\_image\_read(r, in);
345
346         if (rc == NGX\_AGAIN) {
347             return NGX\_OK;
348         }
349
350         if (rc == NGX\_ERROR) {
351             return ngx\_http\_filter\_finalize\_request(r,
352                                                     &ngx\_http\_image\_filter\_module,
353                                                     NGX\_HTTP\_UNSUPPORTED\_MEDIA\_TYPE);
354         }
355
356         /* fall through */
357
358     case NGX\_HTTP\_IMAGE\_PROCESS:
359
360         out.buf = ngx\_http\_image\_process(r);
361
362         if (out.buf == NULL) {
363             return ngx\_http\_filter\_finalize\_request(r,
364                                                     &ngx\_http\_image\_filter\_module,
365                                                     NGX\_HTTP\_UNSUPPORTED\_MEDIA\_TYPE);
366         }
367
368         out.next = NULL;
369         ctx->phase = NGX\_HTTP\_IMAGE\_PASS;
370
371         return ngx\_http\_image\_send(r, ctx, &out);
372
373     case NGX\_HTTP\_IMAGE\_PASS:
374
375         return ngx\_http\_next\_body\_filter(r, in);
376
377     default: /* NGX_HTTP_IMAGE_DONE */
378
379         rc = ngx\_http\_next\_body\_filter(r, NULL);
380
381         /* NGX_ERROR resets any pending data */
382         return (rc == NGX\_OK) ? NGX\_ERROR : rc;
383     }
384 }
385
386
387 static ngx\_int\_t
388 ngx\_http\_image\_send(ngx\_http\_request\_t *r, ngx\_http\_image\_filter\_ctx\_t *ctx,
389 ngx\_chain\_t *in)
390 {
391     ngx\_int\_t rc;
392
393     rc = ngx\_http\_next\_header\_filter(r);
394
395     if (rc == NGX\_ERROR || rc > NGX\_OK || r->header_only) {
396         return NGX\_ERROR;
397     }
398
399     rc = ngx\_http\_next\_body\_filter(r, in);
400
401     if (ctx->phase == NGX\_HTTP\_IMAGE\_DONE) {
402         /* NGX_ERROR resets any pending data */
403         return (rc == NGX\_OK) ? NGX\_ERROR : rc;

```

```

404     }
405
406     return rc;
407 }
408
409
410 static ngx_uint_t
411 ngx_http_image_test(ngx_http_request_t *r, ngx_chain_t *in)
412 {
413     u_char *p;
414
415     p = in->buf->pos;
416
417     if (in->buf->last - p < 16) {
418         return NGX_HTTP_IMAGE_NONE;
419     }
420
421     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
422                  "image filter: \"%c%c\"", p[0], p[1]);
423
424     if (p[0] == 0xff && p[1] == 0xd8) {
425
426         /* JPEG */
427
428         return NGX_HTTP_IMAGE_JPEG;
429
430     } else if (p[0] == 'G' && p[1] == 'I' && p[2] == 'F' && p[3] == '8'
431              && p[5] == 'a')
432     {
433         if (p[4] == '9' || p[4] == '7') {
434             /* GIF */
435             return NGX_HTTP_IMAGE_GIF;
436         }
437
438     } else if (p[0] == 0x89 && p[1] == 'P' && p[2] == 'N' && p[3] == 'G'
439              && p[4] == 0x0d && p[5] == 0x0a && p[6] == 0x1a && p[7] == 0x0a)
440     {
441         /* PNG */
442
443         return NGX_HTTP_IMAGE_PNG;
444     }
445
446     return NGX_HTTP_IMAGE_NONE;
447 }
448
449
450 static ngx_int_t
451 ngx_http_image_read(ngx_http_request_t *r, ngx_chain_t *in)
452 {
453     u_char *p;
454     size_t size, rest;
455     ngx_buf_t *b;
456     ngx_chain_t *cl;
457     ngx_http_image_filter_ctx_t *ctx;
458
459     ctx = ngx_http_get_module_ctx(r, ngx_http_image_filter_module);
460
461     if (ctx->image == NULL) {
462         ctx->image = ngx_palloc(r->pool, ctx->length);
463         if (ctx->image == NULL) {
464             return NGX_ERROR;
465         }
466
467         ctx->last = ctx->image;
468     }
469
470     p = ctx->last;
471
472     for (cl = in; cl; cl = cl->next) {
473
474         b = cl->buf;
475         size = b->last - b->pos;
476
477         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
478                      "image buf: %uz", size);
479

```

```

480     rest = ctx->image + ctx->length - p;
481
482     if (size > rest) {
483         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
484             "image filter: too big response");
485         return NGX_ERROR;
486     }
487
488     p = ngx_cpymem(p, b->pos, size);
489     b->pos += size;
490
491     if (b->last_buf) {
492         ctx->last = p;
493         return NGX_OK;
494     }
495 }
496
497 ctx->last = p;
498 r->connection->buffered |= NGX_HTTP_IMAGE_BUFFERED;
499
500 return NGX_AGAIN;
501 }
502
503
504 static ngx_buf_t *
505 ngx_http_image_process(ngx_http_request_t *r)
506 {
507     ngx_int_t          rc;
508     ngx_http_image_filter_ctx_t *ctx;
509     ngx_http_image_filter_conf_t *conf;
510
511     r->connection->buffered &= ~NGX_HTTP_IMAGE_BUFFERED;
512
513     ctx = ngx_http_get_module_ctx(r, ngx_http_image_filter_module);
514
515     rc = ngx_http_image_size(r, ctx);
516
517     conf = ngx_http_get_module_loc_conf(r, ngx_http_image_filter_module);
518
519     if (conf->filter == NGX_HTTP_IMAGE_SIZE) {
520         return ngx_http_image_json(r, rc == NGX_OK ? ctx : NULL);
521     }
522
523     ctx->angle = ngx_http_image_filter_get_value(r, conf->acv, conf->angle);
524
525     if (conf->filter == NGX_HTTP_IMAGE_ROTATE) {
526
527         if (ctx->angle != 90 && ctx->angle != 180 && ctx->angle != 270) {
528             return NULL;
529         }
530
531         return ngx_http_image_resize(r, ctx);
532     }
533
534     ctx->max_width = ngx_http_image_filter_get_value(r, conf->wcv, conf->width);
535     if (ctx->max_width == 0) {
536         return NULL;
537     }
538
539     ctx->max_height = ngx_http_image_filter_get_value(r, conf->hcv,
540                                                     conf->height);
541
542     if (ctx->max_height == 0) {
543         return NULL;
544     }
545
546     if (rc == NGX_OK
547         && ctx->width <= ctx->max_width
548         && ctx->height <= ctx->max_height
549         && ctx->angle == 0
550         && !ctx->force)
551     {
552         return ngx_http_image_asis(r, ctx);
553     }
554
555     return ngx_http_image_resize(r, ctx);
556 }

```

```

556
557
558 static ngx_buf_t *
559 ngx_http_image_json(ngx_http_request_t *r, ngx_http_image_filter_ctx_t *ctx)
560 {
561     size_t    len;
562     ngx_buf_t *b;
563
564     b = ngx_palloc(r->pool, sizeof(ngx_buf_t));
565     if (b == NULL) {
566         return NULL;
567     }
568
569     b->memory = 1;
570     b->last_buf = 1;
571
572     ngx_http_clean_header(r);
573
574     r->headers_out.status = NGX_HTTP_OK;
575     r->headers_out.content_type_len = sizeof("application/json") - 1;
576     ngx_str_set(&r->headers_out.content_type, "application/json");
577     r->headers_out.content_type_lowercase = NULL;
578
579     if (ctx == NULL) {
580         b->pos = (u_char *) "{}" CRLF;
581         b->last = b->pos + sizeof("{}" CRLF) - 1;
582
583         ngx_http_image_length(r, b);
584
585         return b;
586     }
587
588     len = sizeof("{ \"img\" : "
589                "{ \"width\": , \"height\": , \"type\": \"jpeg\" } }" CRLF) - 1
590         + 2 * NGX_SIZE_T_LEN;
591
592     b->pos = ngx_palloc(r->pool, len);
593     if (b->pos == NULL) {
594         return NULL;
595     }
596
597     b->last = ngx_sprintf(b->pos,
598                          "{ \"img\" : "
599                          "{ \"width\": %uz, "
600                          " \"height\": %uz, "
601                          " \"type\": \"%s\" } }" CRLF,
602                          ctx->width, ctx->height,
603                          ngx_http_image_types[ctx->type - 1].data + 6);
604
605     ngx_http_image_length(r, b);
606
607     return b;
608 }
609
610
611 static ngx_buf_t *
612 ngx_http_image_asis(ngx_http_request_t *r, ngx_http_image_filter_ctx_t *ctx)
613 {
614     ngx_buf_t *b;
615
616     b = ngx_palloc(r->pool, sizeof(ngx_buf_t));
617     if (b == NULL) {
618         return NULL;
619     }
620
621     b->pos = ctx->image;
622     b->last = ctx->last;
623     b->memory = 1;
624     b->last_buf = 1;
625
626     ngx_http_image_length(r, b);
627
628     return b;
629 }
630
631

```

```

632 static void
633 ngx_http_image_length(ngx_http_request_t *r, ngx_buf_t *b)
634 {
635     r->headers_out.content_length_n = b->last - b->pos;
636
637     if (r->headers_out.content_length) {
638         r->headers_out.content_length->hash = 0;
639     }
640
641     r->headers_out.content_length = NULL;
642 }
643
644
645 static ngx_int_t
646 ngx_http_image_size(ngx_http_request_t *r, ngx_http_image_filter_ctx_t *ctx)
647 {
648     u_char      *p, *last;
649     size_t      len, app;
650     ngx_uint_t  width, height;
651
652     p = ctx->image;
653
654     switch (ctx->type) {
655
656     case NGX_HTTP_IMAGE_JPEG:
657
658         p += 2;
659         last = ctx->image + ctx->length - 10;
660         width = 0;
661         height = 0;
662         app = 0;
663
664         while (p < last) {
665
666             if (p[0] == 0xff && p[1] != 0xff) {
667
668                 ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
669                     "JPEG: %02xd %02xd", p[0], p[1]);
670
671                 p++;
672
673                 if ((*p == 0xc0 || *p == 0xc1 || *p == 0xc2 || *p == 0xc3
674                     || *p == 0xc9 || *p == 0xca || *p == 0xcb)
675                     && (width == 0 || height == 0))
676                 {
677                     width = p[6] * 256 + p[7];
678                     height = p[4] * 256 + p[5];
679                 }
680
681                 ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
682                     "JPEG: %02xd %02xd", p[1], p[2]);
683
684                 len = p[1] * 256 + p[2];
685
686                 if (*p >= 0xe1 && *p <= 0xef) {
687                     /* application data, e.g., EXIF, Adobe XMP, etc. */
688                     app += len;
689                 }
690
691                 p += len;
692
693                 continue;
694             }
695
696             p++;
697         }
698
699         if (width == 0 || height == 0) {
700             return NGX_DECLINED;
701         }
702
703         if (ctx->length / 20 < app) {
704             /* force conversion if application data consume more than 5% */
705             ctx->force = 1;
706             ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
707                 "app data size: %uz", app);

```

```

708     }
709
710     break;
711
712     case NGX\_HTTP\_IMAGE\_GIF:
713
714         if (ctx->length < 10) {
715             return NGX\_DECLINED;
716         }
717
718         width = p[7] * 256 + p[6];
719         height = p[9] * 256 + p[8];
720
721         break;
722
723     case NGX\_HTTP\_IMAGE\_PNG:
724
725         if (ctx->length < 24) {
726             return NGX\_DECLINED;
727         }
728
729         width = p[18] * 256 + p[19];
730         height = p[22] * 256 + p[23];
731
732         break;
733
734     default:
735
736         return NGX\_DECLINED;
737 }
738
739 ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
740              "image size: %d x %d", width, height);
741
742     ctx->width = width;
743     ctx->height = height;
744
745     return NGX\_OK;
746 }
747
748
749 static ngx\_buf\_t *
750 ngx\_http\_image\_resize(ngx\_http\_request\_t *r, ngx\_http\_image\_filter\_ctx\_t *ctx)
751 {
752     int                sx, sy, dx, dy, ox, oy, ax, ay, size,
753                     colors, palette, transparent, sharpen,
754                     red, green, blue, t;
755     u_char            *out;
756     ngx\_buf\_t         *b;
757     ngx\_uint\_t        resize;
758     gdImagePtr        src, dst;
759     ngx\_pool\_cleanup\_t *cln;
760     ngx\_http\_image\_filter\_conf\_t *conf;
761
762     src = ngx\_http\_image\_source(r, ctx);
763
764     if (src == NULL) {
765         return NULL;
766     }
767
768     sx = gdImageSX(src);
769     sy = gdImageSY(src);
770
771     conf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_image\_filter\_module);
772
773     if (!ctx->force
774         && ctx->angle == 0
775         && (ngx\_uint\_t) sx <= ctx->max_width
776         && (ngx\_uint\_t) sy <= ctx->max_height)
777     {
778         gdImageDestroy(src);
779         return ngx\_http\_image\_asis(r, ctx);
780     }
781
782     colors = gdImageColorsTotal(src);
783

```



```

784     if (colors && conf->transparency) {
785         transparent = gdImageGetTransparent(src);
786
787         if (transparent != -1) {
788             palette = colors;
789             red = gdImageRed(src, transparent);
790             green = gdImageGreen(src, transparent);
791             blue = gdImageBlue(src, transparent);
792
793             goto transparent;
794         }
795     }
796
797     palette = 0;
798     transparent = -1;
799     red = 0;
800     green = 0;
801     blue = 0;
802
803 transparent:
804
805     gdImageColorTransparent(src, -1);
806
807     dx = sx;
808     dy = sy;
809
810     if (conf->filter == NGX\_HTTP\_IMAGE\_RESIZE) {
811
812         if ((ngx\_uint\_t) dx > ctx->max_width) {
813             dy = dy * ctx->max_width / dx;
814             dy = dy ? dy : 1;
815             dx = ctx->max_width;
816         }
817
818         if ((ngx\_uint\_t) dy > ctx->max_height) {
819             dx = dx * ctx->max_height / dy;
820             dx = dx ? dx : 1;
821             dy = ctx->max_height;
822         }
823
824         resize = 1;
825
826     } else if (conf->filter == NGX\_HTTP\_IMAGE\_ROTATE) {
827
828         resize = 0;
829
830     } else { /* NGX\_HTTP\_IMAGE\_CROP */
831
832         resize = 0;
833
834         if ((double) dx / dy < (double) ctx->max_width / ctx->max_height) {
835             if ((ngx\_uint\_t) dx > ctx->max_width) {
836                 dy = dy * ctx->max_width / dx;
837                 dy = dy ? dy : 1;
838                 dx = ctx->max_width;
839                 resize = 1;
840             }
841
842         } else {
843             if ((ngx\_uint\_t) dy > ctx->max_height) {
844                 dx = dx * ctx->max_height / dy;
845                 dx = dx ? dx : 1;
846                 dy = ctx->max_height;
847                 resize = 1;
848             }
849         }
850     }
851
852     if (resize) {
853         dst = ngx\_http\_image\_new(r, dx, dy, palette);
854         if (dst == NULL) {
855             gdImageDestroy(src);
856             return NULL;
857         }
858
859         if (colors == 0) {

```

```

860         gdImageSaveAlpha(dst, 1);
861         gdImageAlphaBlending(dst, 0);
862     }
863
864     gdImageCopyResampled(dst, src, 0, 0, 0, 0, dx, dy, sx, sy);
865
866     if (colors) {
867         gdImageTrueColorToPalette(dst, 1, 256);
868     }
869
870     gdImageDestroy(src);
871
872 } else {
873     dst = src;
874 }
875
876 if (ctx->angle) {
877     src = dst;
878
879     ax = (dx % 2 == 0) ? 1 : 0;
880     ay = (dy % 2 == 0) ? 1 : 0;
881
882     switch (ctx->angle) {
883
884     case 90:
885     case 270:
886         dst = ngx_http_image_new(r, dy, dx, palette);
887         if (dst == NULL) {
888             gdImageDestroy(src);
889             return NULL;
890         }
891         if (ctx->angle == 90) {
892             ox = dy / 2 + ay;
893             oy = dx / 2 - ax;
894
895         } else {
896             ox = dy / 2 - ay;
897             oy = dx / 2 + ax;
898         }
899
900         gdImageCopyRotated(dst, src, ox, oy, 0, 0,
901                             dx + ax, dy + ay, ctx->angle);
902         gdImageDestroy(src);
903
904         t = dx;
905         dx = dy;
906         dy = t;
907         break;
908
909     case 180:
910         dst = ngx_http_image_new(r, dx, dy, palette);
911         if (dst == NULL) {
912             gdImageDestroy(src);
913             return NULL;
914         }
915         gdImageCopyRotated(dst, src, dx / 2 - ax, dy / 2 - ay, 0, 0,
916                             dx + ax, dy + ay, ctx->angle);
917         gdImageDestroy(src);
918         break;
919     }
920 }
921
922 if (conf->filter == NGX_HTTP_IMAGE_CROP) {
923
924     src = dst;
925
926     if ((ngx_uint_t) dx > ctx->max_width) {
927         ox = dx - ctx->max_width;
928
929     } else {
930         ox = 0;
931     }
932
933     if ((ngx_uint_t) dy > ctx->max_height) {
934         oy = dy - ctx->max_height;
935

```

```

936     } else {
937         oy = 0;
938     }
939
940     if (ox || oy) {
941         dst = ngx_http_image_new(r, dx - ox, dy - oy, colors);
942
943         if (dst == NULL) {
944             gdImageDestroy(src);
945             return NULL;
946         }
947
948         ox /= 2;
949         oy /= 2;
950
951         ngx_log_debug4(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
952             "image crop: %d x %d @ %d x %d",
953             dx, dy, ox, oy);
954
955         if (colors == 0) {
956             gdImageSaveAlpha(dst, 1);
957             gdImageAlphaBlending(dst, 0);
958         }
959
960         gdImageCopy(dst, src, 0, 0, ox, oy, dx - ox, dy - oy);
961
962         if (colors) {
963             gdImageTrueColorToPalette(dst, 1, 256);
964         }
965
966         gdImageDestroy(src);
967     }
968 }
969
970
971 if (transparent != -1 && colors) {
972     gdImageColorTransparent(dst, gdImageColorExact(dst, red, green, blue));
973 }
974
975 sharpen = ngx_http_image_filter_get_value(r, conf->shcv, conf->sharpen);
976 if (sharpen > 0) {
977     gdImageSharpen(dst, sharpen);
978 }
979
980 gdImageInterlace(dst, (int) conf->interlace);
981
982 out = ngx_http_image_out(r, ctx->type, dst, &size);
983
984 ngx_log_debug3(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
985     "image: %d x %d %d", sx, sy, colors);
986
987 gdImageDestroy(dst);
988 ngx_pfree(r->pool, ctx->image);
989
990 if (out == NULL) {
991     return NULL;
992 }
993
994 cIn = ngx_pool_cleanup_add(r->pool, 0);
995 if (cIn == NULL) {
996     gdFree(out);
997     return NULL;
998 }
999
1000 b = ngx_pcalloc(r->pool, sizeof(ngx_buf_t));
1001 if (b == NULL) {
1002     gdFree(out);
1003     return NULL;
1004 }
1005
1006 cIn->handler = ngx_http_image_cleanup;
1007 cIn->data = out;
1008
1009 b->pos = out;
1010 b->last = out + size;
1011 b->memory = 1;

```

```

1012     b->last_buf = 1;
1013
1014     ngx_http_image_length(r, b);
1015     ngx_http_weak_etag(r);
1016
1017     return b;
1018 }
1019
1020
1021 static gdImagePtr
1022 ngx_http_image_source(ngx_http_request_t *r, ngx_http_image_filter_ctx_t *ctx)
1023 {
1024     char *failed;
1025     gdImagePtr img;
1026
1027     img = NULL;
1028
1029     switch (ctx->type) {
1030
1031     case NGX_HTTP_IMAGE_JPEG:
1032         img = gdImageCreateFromJpegPtr(ctx->length, ctx->image);
1033         failed = "gdImageCreateFromJpegPtr() failed";
1034         break;
1035
1036     case NGX_HTTP_IMAGE_GIF:
1037         img = gdImageCreateFromGifPtr(ctx->length, ctx->image);
1038         failed = "gdImageCreateFromGifPtr() failed";
1039         break;
1040
1041     case NGX_HTTP_IMAGE_PNG:
1042         img = gdImageCreateFromPngPtr(ctx->length, ctx->image);
1043         failed = "gdImageCreateFromPngPtr() failed";
1044         break;
1045
1046     default:
1047         failed = "unknown image type";
1048         break;
1049     }
1050
1051     if (img == NULL) {
1052         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0, failed);
1053     }
1054
1055     return img;
1056 }
1057
1058
1059 static gdImagePtr
1060 ngx_http_image_new(ngx_http_request_t *r, int w, int h, int colors)
1061 {
1062     gdImagePtr img;
1063
1064     if (colors == 0) {
1065         img = gdImageCreateTrueColor(w, h);
1066
1067         if (img == NULL) {
1068             ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1069                 "gdImageCreateTrueColor() failed");
1070             return NULL;
1071         }
1072     } else {
1073         img = gdImageCreate(w, h);
1074
1075         if (img == NULL) {
1076             ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1077                 "gdImageCreate() failed");
1078             return NULL;
1079         }
1080     }
1081
1082     return img;
1083 }
1084
1085
1086
1087 static u_char *

```

```

1088 ngx_http_image_out(ngx_http_request_t *r, ngx_uint_t type, gdImagePtr img,
1089 int *size)
1090 {
1091     char *failed;
1092     u_char *out;
1093     ngx_int_t jq;
1094     ngx_http_image_filter_conf_t *conf;
1095
1096     out = NULL;
1097
1098     switch (type) {
1099
1100     case NGX_HTTP_IMAGE_JPEG:
1101         conf = ngx_http_get_module_loc_conf(r, ngx_http_image_filter_module);
1102
1103         jq = ngx_http_image_filter_get_value(r, conf->jqcv, conf->jpeg_quality);
1104         if (jq <= 0) {
1105             return NULL;
1106         }
1107
1108         out = gdImageJpegPtr(img, size, jq);
1109         failed = "gdImageJpegPtr() failed";
1110         break;
1111
1112     case NGX_HTTP_IMAGE_GIF:
1113         out = gdImageGifPtr(img, size);
1114         failed = "gdImageGifPtr() failed";
1115         break;
1116
1117     case NGX_HTTP_IMAGE_PNG:
1118         out = gdImagePngPtr(img, size);
1119         failed = "gdImagePngPtr() failed";
1120         break;
1121
1122     default:
1123         failed = "unknown image type";
1124         break;
1125     }
1126
1127     if (out == NULL) {
1128         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0, failed);
1129     }
1130
1131     return out;
1132 }
1133
1134
1135 static void
1136 ngx_http_image_cleanup(void *data)
1137 {
1138     gdFree(data);
1139 }
1140
1141
1142 static ngx_uint_t
1143 ngx_http_image_filter_get_value(ngx_http_request_t *r,
1144 ngx_http_complex_value_t *cv, ngx_uint_t v)
1145 {
1146     ngx_str_t val;
1147
1148     if (cv == NULL) {
1149         return v;
1150     }
1151
1152     if (ngx_http_complex_value(r, cv, &val) != NGX_OK) {
1153         return 0;
1154     }
1155
1156     return ngx_http_image_filter_value(&val);
1157 }
1158
1159
1160 static ngx_uint_t
1161 ngx_http_image_filter_value(ngx_str_t *value)
1162 {
1163     ngx_int_t n;

```

```

1164     if (value->len == 1 && value->data[0] == '-') {
1165         return (ngx_uint_t) -1;
1166     }
1167 }
1168
1169 n = ngx_atoi(value->data, value->len);
1170
1171 if (n > 0) {
1172     return (ngx_uint_t) n;
1173 }
1174
1175 return 0;
1176 }
1177
1178
1179 static void *
1180 ngx_http_image_filter_create_conf(ngx_conf_t *cf)
1181 {
1182     ngx_http_image_filter_conf_t *conf;
1183
1184     conf = ngx_palloc(cf->pool, sizeof(ngx_http_image_filter_conf_t));
1185     if (conf == NULL) {
1186         return NULL;
1187     }
1188
1189     /*
1190     * set by ngx_palloc():
1191     *
1192     *     conf->width = 0;
1193     *     conf->height = 0;
1194     *     conf->angle = 0;
1195     *     conf->wcv = NULL;
1196     *     conf->hcv = NULL;
1197     *     conf->acv = NULL;
1198     *     conf->jqcv = NULL;
1199     *     conf->shcv = NULL;
1200     */
1201
1202     conf->filter = NGX_CONF_UNSET_UINT;
1203     conf->jpeg_quality = NGX_CONF_UNSET_UINT;
1204     conf->sharpen = NGX_CONF_UNSET_UINT;
1205     conf->transparency = NGX_CONF_UNSET;
1206     conf->interlace = NGX_CONF_UNSET;
1207     conf->buffer_size = NGX_CONF_UNSET_SIZE;
1208
1209     return conf;
1210 }
1211
1212
1213 static char *
1214 ngx_http_image_filter_merge_conf(ngx_conf_t *cf, void *parent, void *child)
1215 {
1216     ngx_http_image_filter_conf_t *prev = parent;
1217     ngx_http_image_filter_conf_t *conf = child;
1218
1219     if (conf->filter == NGX_CONF_UNSET_UINT) {
1220
1221         if (prev->filter == NGX_CONF_UNSET_UINT) {
1222             conf->filter = NGX_HTTP_IMAGE_OFF;
1223
1224         } else {
1225             conf->filter = prev->filter;
1226             conf->width = prev->width;
1227             conf->height = prev->height;
1228             conf->angle = prev->angle;
1229             conf->wcv = prev->wcv;
1230             conf->hcv = prev->hcv;
1231             conf->acv = prev->acv;
1232         }
1233     }
1234
1235     if (conf->jpeg_quality == NGX_CONF_UNSET_UINT) {
1236
1237         /* 75 is libjpeg default quality */
1238         ngx_conf_merge_uint_value(conf->jpeg_quality, prev->jpeg_quality, 75);
1239     }

```

```

1240     if (conf->jqcv == NULL) {
1241         conf->jqcv = prev->jqcv;
1242     }
1243 }
1244
1245 if (conf->sharpen == NGX\_CONF\_UNSET\_UINT) {
1246     ngx\_conf\_merge\_uint\_value(conf->sharpen, prev->sharpen, 0);
1247
1248     if (conf->shcv == NULL) {
1249         conf->shcv = prev->shcv;
1250     }
1251 }
1252
1253 ngx\_conf\_merge\_value(conf->transparency, prev->transparency, 1);
1254
1255 ngx\_conf\_merge\_value(conf->interlace, prev->interlace, 0);
1256
1257 ngx\_conf\_merge\_size\_value(conf->buffer_size, prev->buffer_size,
1258     1 * 1024 * 1024);
1259
1260 return NGX\_CONF\_OK;
1261 }
1262
1263
1264 static char *
1265 ngx\_http\_image\_filter(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
1266 {
1267     ngx\_http\_image\_filter\_conf\_t *imcf = conf;
1268
1269     ngx\_str\_t          *value;
1270     ngx\_int\_t         n;
1271     ngx\_uint\_t        i;
1272     ngx\_http\_complex\_value\_t cv;
1273     ngx\_http\_compile\_complex\_value\_t ccv;
1274
1275     value = cf->args->elts;
1276
1277     i = 1;
1278
1279     if (cf->args->nelts == 2) {
1280         if (ngx\_strcmp(value[i].data, "off") == 0) {
1281             imcf->filter = NGX\_HTTP\_IMAGE\_OFF;
1282
1283         } else if (ngx\_strcmp(value[i].data, "test") == 0) {
1284             imcf->filter = NGX\_HTTP\_IMAGE\_TEST;
1285
1286         } else if (ngx\_strcmp(value[i].data, "size") == 0) {
1287             imcf->filter = NGX\_HTTP\_IMAGE\_SIZE;
1288
1289         } else {
1290             goto failed;
1291         }
1292
1293         return NGX\_CONF\_OK;
1294
1295     } else if (cf->args->nelts == 3) {
1296
1297         if (ngx\_strcmp(value[i].data, "rotate") == 0) {
1298             if (imcf->filter != NGX\_HTTP\_IMAGE\_RESIZE
1299                 && imcf->filter != NGX\_HTTP\_IMAGE\_CROP)
1300             {
1301                 imcf->filter = NGX\_HTTP\_IMAGE\_ROTATE;
1302             }
1303
1304             ngx\_memzero(&ccv, sizeof(ngx\_http\_compile\_complex\_value\_t));
1305
1306             ccv.cf = cf;
1307             ccv.value = &value[++i];
1308             ccv.complex_value = &cv;
1309
1310             if (ngx\_http\_compile\_complex\_value(&ccv) != NGX\_OK) {
1311                 return NGX\_CONF\_ERROR;
1312             }
1313
1314             if (cv.lengths == NULL) {
1315                 n = ngx\_http\_image\_filter\_value(&value[i]);

```

```

1316         if (n != 90 && n != 180 && n != 270){
1317             goto failed;
1318         }
1319     }
1320
1321     imcf->angle = (ngx_uint_t) n;
1322
1323     } else {
1324         imcf->acv = ngx_palloc(cf->pool,
1325                               sizeof(ngx_http_complex_value_t));
1326         if (imcf->acv == NULL) {
1327             return NGX_CONF_ERROR;
1328         }
1329
1330         *imcf->acv = cv;
1331     }
1332
1333     return NGX_CONF_OK;
1334
1335     } else {
1336         goto failed;
1337     }
1338 }
1339
1340 if (ngx_strcmp(value[i].data, "resize") == 0) {
1341     imcf->filter = NGX_HTTP_IMAGE_RESIZE;
1342
1343 } else if (ngx_strcmp(value[i].data, "crop") == 0) {
1344     imcf->filter = NGX_HTTP_IMAGE_CROP;
1345
1346 } else {
1347     goto failed;
1348 }
1349
1350 ngx_memzero(&ccv, sizeof(ngx_http_compile_complex_value_t));
1351
1352 ccv.cf = cf;
1353 ccv.value = &value[++i];
1354 ccv.complex_value = &cv;
1355
1356 if (ngx_http_compile_complex_value(&ccv) != NGX_OK) {
1357     return NGX_CONF_ERROR;
1358 }
1359
1360 if (cv.lengths == NULL) {
1361     n = ngx_http_image_filter_value(&value[i]);
1362
1363     if (n == 0) {
1364         goto failed;
1365     }
1366
1367     imcf->width = (ngx_uint_t) n;
1368
1369 } else {
1370     imcf->wcv = ngx_palloc(cf->pool, sizeof(ngx_http_complex_value_t));
1371     if (imcf->wcv == NULL) {
1372         return NGX_CONF_ERROR;
1373     }
1374
1375     *imcf->wcv = cv;
1376 }
1377
1378 ngx_memzero(&ccv, sizeof(ngx_http_compile_complex_value_t));
1379
1380 ccv.cf = cf;
1381 ccv.value = &value[++i];
1382 ccv.complex_value = &cv;
1383
1384 if (ngx_http_compile_complex_value(&ccv) != NGX_OK) {
1385     return NGX_CONF_ERROR;
1386 }
1387
1388 if (cv.lengths == NULL) {
1389     n = ngx_http_image_filter_value(&value[i]);
1390
1391     if (n == 0) {

```



```

1392     goto failed;
1393 }
1394
1395     imcf->height = (ngx_uint_t) n;
1396
1397 } else {
1398     imcf->hcv = ngx_palloc(cf->pool, sizeof(ngx_http_complex_value_t));
1399     if (imcf->hcv == NULL) {
1400         return NGX_CONF_ERROR;
1401     }
1402
1403     *imcf->hcv = cv;
1404 }
1405
1406     return NGX_CONF_OK;
1407
1408 failed:
1409
1410     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0, "invalid parameter \"%V\"",
1411         &value[i]);
1412
1413     return NGX_CONF_ERROR;
1414 }
1415
1416
1417 static char *
1418 ngx_http_image_filter_jpeg_quality(ngx_conf_t *cf, ngx_command_t *cmd,
1419     void *conf)
1420 {
1421     ngx_http_image_filter_conf_t *imcf = conf;
1422
1423     ngx_str_t                *value;
1424     ngx_int_t                n;
1425     ngx_http_complex_value_t cv;
1426     ngx_http_compile_complex_value_t ccv;
1427
1428     value = cf->args->elts;
1429
1430     ngx_memzero(&ccv, sizeof(ngx_http_compile_complex_value_t));
1431
1432     ccv.cf = cf;
1433     ccv.value = &value[1];
1434     ccv.complex_value = &cv;
1435
1436     if (ngx_http_compile_complex_value(&ccv) != NGX_OK) {
1437         return NGX_CONF_ERROR;
1438     }
1439
1440     if (cv.lengths == NULL) {
1441         n = ngx_http_image_filter_value(&value[1]);
1442
1443         if (n <= 0) {
1444             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1445                 "invalid value \"%V\"", &value[1]);
1446             return NGX_CONF_ERROR;
1447         }
1448
1449         imcf->jpeg_quality = (ngx_uint_t) n;
1450
1451     } else {
1452         imcf->jqcv = ngx_palloc(cf->pool, sizeof(ngx_http_complex_value_t));
1453         if (imcf->jqcv == NULL) {
1454             return NGX_CONF_ERROR;
1455         }
1456
1457         *imcf->jqcv = cv;
1458     }
1459
1460     return NGX_CONF_OK;
1461 }
1462
1463
1464 static char *
1465 ngx_http_image_filter_sharpen(ngx_conf_t *cf, ngx_command_t *cmd,
1466     void *conf)
1467 {

```

```

1468 ngx\_http\_image\_filter\_conf\_t *imcf = conf;
1469
1470 ngx\_str\_t *value;
1471 ngx\_int\_t n;
1472 ngx\_http\_complex\_value\_t cv;
1473 ngx\_http\_compile\_complex\_value\_t ccv;
1474
1475 value = cf->args->elts;
1476
1477 ngx\_memzero(&ccv, sizeof(ngx\_http\_compile\_complex\_value\_t));
1478
1479 ccv.cf = cf;
1480 ccv.value = &value[1];
1481 ccv.complex_value = &cv;
1482
1483 if (ngx\_http\_compile\_complex\_value(&ccv) != NGX\_OK) {
1484     return NGX\_CONF\_ERROR;
1485 }
1486
1487 if (cv.lengths == NULL) {
1488     n = ngx\_http\_image\_filter\_value(&value[1]);
1489
1490     if (n < 0) {
1491         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
1492             "invalid value \"%V\"", &value[1]);
1493         return NGX\_CONF\_ERROR;
1494     }
1495
1496     imcf->sharpen = (ngx\_uint\_t) n;
1497
1498 } else {
1499     imcf->shcv = ngx\_palloc(cf->pool, sizeof(ngx\_http\_complex\_value\_t));
1500     if (imcf->shcv == NULL) {
1501         return NGX\_CONF\_ERROR;
1502     }
1503
1504     *imcf->shcv = cv;
1505 }
1506
1507 return NGX\_CONF\_OK;
1508 }
1509
1510
1511 static ngx\_int\_t
1512 ngx\_http\_image\_filter\_init(ngx\_conf\_t *cf)
1513 {
1514     ngx\_http\_next\_header\_filter = ngx\_http\_top\_header\_filter;
1515     ngx\_http\_top\_header\_filter = ngx\_http\_image\_header\_filter;
1516
1517     ngx\_http\_next\_body\_filter = ngx\_http\_top\_body\_filter;
1518     ngx\_http\_top\_body\_filter = ngx\_http\_image\_body\_filter;
1519
1520     return NGX\_OK;
1521 }

```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_index\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_index\\_commands](#)
- [ngx\\_http\\_index\\_module](#)
- [ngx\\_http\\_index\\_module\\_ctx](#)

## Data types defined

- [ngx\\_http\\_index\\_loc\\_conf\\_t](#)
- [ngx\\_http\\_index\\_t](#)

## Functions defined

- [ngx\\_http\\_index\\_create\\_loc\\_conf](#)
- [ngx\\_http\\_index\\_error](#)
- [ngx\\_http\\_index\\_handler](#)
- [ngx\\_http\\_index\\_init](#)
- [ngx\\_http\\_index\\_merge\\_loc\\_conf](#)
- [ngx\\_http\\_index\\_set\\_index](#)
- [ngx\\_http\\_index\\_test\\_dir](#)

## Macros defined

- [NGX\\_HTTP\\_DEFAULT\\_INDEX](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 typedef struct {
14     ngx_str_t          name;
15     ngx_array_t        *lengths;
16     ngx_array_t        *values;
17 } ngx_http_index_t;
18
19
20 typedef struct {
21     ngx_array_t        *indices; /* array of ngx_http_index_t */
22     size_t              max_index_len;
23 } ngx_http_index_loc_conf_t;
24
```

```

25
26 #define NGX_HTTP_DEFAULT_INDEX    "index.html"
27
28
29 static ngx_int_t ngx_http_index_test_dir(ngx_http_request_t *r,
30     ngx_http_core_loc_conf_t *clcf, u_char *path, u_char *last);
31 static ngx_int_t ngx_http_index_error(ngx_http_request_t *r,
32     ngx_http_core_loc_conf_t *clcf, u_char *file, ngx_err_t err);
33
34 static ngx_int_t ngx_http_index_init(ngx_conf_t *cf);
35 static void *ngx_http_index_create_loc_conf(ngx_conf_t *cf);
36 static char *ngx_http_index_merge_loc_conf(ngx_conf_t *cf,
37     void *parent, void *child);
38 static char *ngx_http_index_set_index(ngx_conf_t *cf, ngx_command_t *cmd,
39     void *conf);
40
41
42 static ngx_command_t  ngx_http_index_commands[] = {
43
44     { ngx_string("index"),
45       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_1MORE,
46       ngx_http_index_set_index,
47       NGX_HTTP_LOC_CONF_OFFSET,
48       0,
49       NULL },
50
51     ngx_null_command
52 };
53
54
55 static ngx_http_module_t  ngx_http_index_module_ctx = {
56     NULL,                          /* preconfiguration */
57     ngx_http_index_init,            /* postconfiguration */
58
59     NULL,                            /* create main configuration */
60     NULL,                            /* init main configuration */
61
62     NULL,                            /* create server configuration */
63     NULL,                            /* merge server configuration */
64
65     ngx_http_index_create_loc_conf, /* create location configuration */
66     ngx_http_index_merge_loc_conf   /* merge location configuration */
67 };
68
69
70 ngx_module_t  ngx_http_index_module = {
71     NGX_MODULE_V1,
72     &ngx_http_index_module_ctx,     /* module context */
73     ngx_http_index_commands,        /* module directives */
74     NGX_HTTP_MODULE,                /* module type */
75     NULL,                            /* init master */
76     NULL,                            /* init module */
77     NULL,                            /* init process */
78     NULL,                            /* init thread */
79     NULL,                            /* exit thread */
80     NULL,                            /* exit process */
81     NULL,                            /* exit master */
82     NGX_MODULE_V1_PADDING
83 };
84
85
86 /*
87  * Try to open/test the first index file before the test of directory
88  * existence because valid requests should prevail over invalid ones.
89  * If open()/stat() of a file will fail then stat() of a directory
90  * should be faster because kernel may have already cached some data.
91  * Besides, Win32 may return ERROR_PATH_NOT_FOUND (NGX_ENOTDIR) at once.
92  * Unix has ENOTDIR error; however, it's less helpful than Win32's one:
93  * it only indicates that path points to a regular file, not a directory.
94  */
95
96 static ngx_int_t
97 ngx_http_index_handler(ngx_http_request_t *r)
98 {
99     u_char          *p, *name;
100     size_t          len, root, reserve, allocated;

```

```

101     ngx_int_t          rc;
102     ngx_str_t          path, uri;
103     ngx_uint_t         i, dir_tested;
104     ngx_http_index_t   *index;
105     ngx_open_file_info_t of;
106     ngx_http_script_code_pt code;
107     ngx_http_script_engine_t e;
108     ngx_http_core_loc_conf_t *clcf;
109     ngx_http_index_loc_conf_t *ilcf;
110     ngx_http_script_len_code_pt lcode;
111
112     if (r->uri.data[r->uri.len - 1] != '/') {
113         return NGX_DECLINED;
114     }
115
116     if (!(r->method & (NGX_HTTP_GET|NGX_HTTP_HEAD|NGX_HTTP_POST))) {
117         return NGX_DECLINED;
118     }
119
120     ilcf = ngx_http_get_module_loc_conf(r, ngx_http_index_module);
121     clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
122
123     allocated = 0;
124     root = 0;
125     dir_tested = 0;
126     name = NULL;
127     /* suppress MSVC warning */
128     path.data = NULL;
129
130     index = ilcf->indices->elts;
131     for (i = 0; i < ilcf->indices->nelts; i++) {
132
133         if (index[i].lengths == NULL) {
134
135             if (index[i].name.data[0] == '/') {
136                 return ngx_http_internal_redirect(r, &index[i].name, &r->args);
137             }
138
139             reserve = ilcf->max_index_len;
140             len = index[i].name.len;
141
142         } else {
143             ngx_memzero(&e, sizeof(ngx_http_script_engine_t));
144
145             e.ip = index[i].lengths->elts;
146             e.request = r;
147             e.flushed = 1;
148
149             /* 1 is for terminating '\0' as in static names */
150             len = 1;
151
152             while (*(uintptr_t *) e.ip) {
153                 lcode = *(ngx_http_script_len_code_pt *) e.ip;
154                 len += lcode(&e);
155             }
156
157             /* 16 bytes are preallocation */
158
159             reserve = len + 16;
160         }
161
162         if (reserve > allocated) {
163
164             name = ngx_http_map_uri_to_path(r, &path, &root, reserve);
165             if (name == NULL) {
166                 return NGX_ERROR;
167             }
168
169             allocated = path.data + path.len - name;
170         }
171
172         if (index[i].values == NULL) {
173
174             /* index[i].name.len includes the terminating '\0' */
175
176             ngx_memcpy(name, index[i].name.data, index[i].name.len);

```

```

177     path.len = (name + index[i].name.len - 1) - path.data;
178
179
180 } else {
181     e.ip = index[i].values->elts;
182     e.pos = name;
183
184     while (*(uintptr_t *) e.ip) {
185         code = *(ngx_http_script_code_pt *) e.ip;
186         code((ngx_http_script_engine_t *) &e);
187     }
188
189     if (*name == '/') {
190         uri.len = len - 1;
191         uri.data = name;
192         return ngx_http_internal_redirect(r, &uri, &r->args);
193     }
194
195     path.len = e.pos - path.data;
196
197     *e.pos = '\0';
198 }
199
200 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
201               "open index \"%V\"", &path);
202
203 ngx_memzero(&of, sizeof(ngx_open_file_info_t));
204
205 of.read_ahead = clcf->read_ahead;
206 of.directio = clcf->directio;
207 of.valid = clcf->open_file_cache_valid;
208 of.min_uses = clcf->open_file_cache_min_uses;
209 of.test_only = 1;
210 of.errors = clcf->open_file_cache_errors;
211 of.events = clcf->open_file_cache_events;
212
213 if (ngx_http_set_disable_symlinks(r, clcf, &path, &of) != NGX_OK) {
214     return NGX_HTTP_INTERNAL_SERVER_ERROR;
215 }
216
217 if (ngx_open_cached_file(clcf->open_file_cache, &path, &of, r->pool)
218     != NGX_OK)
219 {
220     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, of.err,
221                   "%s \"%s\" failed", of.failed, path.data);
222
223     if (of.err == 0) {
224         return NGX_HTTP_INTERNAL_SERVER_ERROR;
225     }
226
227 #if (NGX_HAVE_OPENAT)
228     if (of.err == NGX_EMLINK
229         || of.err == NGX_ELOOP)
230     {
231         return NGX_HTTP_FORBIDDEN;
232     }
233 #endif
234
235     if (of.err == NGX_ENOTDIR
236         || of.err == NGX_ENAMETOOLONG
237         || of.err == NGX_EACCES)
238     {
239         return ngx_http_index_error(r, clcf, path.data, of.err);
240     }
241
242     if (!dir_tested) {
243         rc = ngx_http_index_test_dir(r, clcf, path.data, name - 1);
244
245         if (rc != NGX_OK) {
246             return rc;
247         }
248
249         dir_tested = 1;
250     }
251
252     if (of.err == NGX_ENOENT) {

```

```

253         continue;
254     }
255
256     ngx_log_error(NGX_LOG_CRIT, r->connection->log, of.err,
257                 "%s \\\"%s\\\" failed", of.failed, path.data);
258
259     return NGX_HTTP_INTERNAL_SERVER_ERROR;
260 }
261
262 uri.len = r->uri.len + len - 1;
263
264 if (!clcf->alias) {
265     uri.data = path.data + root;
266
267 } else {
268     uri.data = ngx_pnalloc(r->pool, uri.len);
269     if (uri.data == NULL) {
270         return NGX_HTTP_INTERNAL_SERVER_ERROR;
271     }
272
273     p = ngx_copy(uri.data, r->uri.data, r->uri.len);
274     ngx_memcpy(p, name, len - 1);
275 }
276
277 return ngx_http_internal_redirect(r, &uri, &r->args);
278 }
279
280 return NGX_DECLINED;
281 }
282
283
284 static ngx_int_t
285 ngx_http_index_test_dir(ngx_http_request_t *r, ngx_http_core_loc_conf_t *clcf,
286 u_char *path, u_char *last)
287 {
288     u_char          c;
289     ngx_str_t       dir;
290     ngx_open_file_info_t of;
291
292     c = *last;
293     if (c != '/' || path == last) {
294         /* "alias" without trailing slash */
295         c = *(++last);
296     }
297     *last = '\0';
298
299     dir.len = last - path;
300     dir.data = path;
301
302     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
303                  "http index check dir: \"%V\"", &dir);
304
305     ngx_memzero(&of, sizeof(ngx_open_file_info_t));
306
307     of.test_dir = 1;
308     of.test_only = 1;
309     of.valid = clcf->open_file_cache_valid;
310     of.errors = clcf->open_file_cache_errors;
311
312     if (ngx_http_set_disable_symlinks(r, clcf, &dir, &of) != NGX_OK) {
313         return NGX_HTTP_INTERNAL_SERVER_ERROR;
314     }
315
316     if (ngx_open_cached_file(clcf->open_file_cache, &dir, &of, r->pool)
317         != NGX_OK)
318     {
319         if (of.err) {
320
321 #if (NGX_HAVE_OPENAT)
322             if (of.err == NGX_EMLINK
323                 || of.err == NGX_ELOOP)
324             {
325                 return NGX_HTTP_FORBIDDEN;
326             }
327 #endif
328

```

```

329     if (of.err == NGX\_ENOENT) {
330         *last = c;
331         return ngx\_http\_index\_error(r, clcf, dir.data, NGX\_ENOENT);
332     }
333
334     if (of.err == NGX\_EACCES) {
335
336         *last = c;
337
338         /*
339          * ngx\_http\_index\_test\_dir\(\) is called after the first index
340          * file testing has returned an error distinct from NGX\_EACCES.
341          * This means that directory searching is allowed.
342          */
343
344         return NGX\_OK;
345     }
346
347     ngx\_log\_error(NGX\_LOG\_CRIT, r->connection->log, of.err,
348                 "%s \"%s\" failed", of.failed, dir.data);
349 }
350
351 return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
352 }
353
354 *last = c;
355
356 if (of.is_dir) {
357     return NGX\_OK;
358 }
359
360 ngx\_log\_error(NGX\_LOG\_ALERT, r->connection->log, 0,
361               "\"%s\" is not a directory", dir.data);
362
363 return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
364 }
365
366
367 static ngx\_int\_t
368 ngx\_http\_index\_error(ngx\_http\_request\_t *r, ngx\_http\_core\_loc\_conf\_t *clcf,
369                     u\_char *file, ngx\_err\_t err)
370 {
371     if (err == NGX\_EACCES) {
372         ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, err,
373                     "\"%s\" is forbidden", file);
374
375         return NGX\_HTTP\_FORBIDDEN;
376     }
377
378     if (clcf->log_not_found) {
379         ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, err,
380                     "\"%s\" is not found", file);
381     }
382
383     return NGX\_HTTP\_NOT\_FOUND;
384 }
385
386
387 static void *
388 ngx\_http\_index\_create\_loc\_conf(ngx\_conf\_t *cf)
389 {
390     ngx\_http\_index\_loc\_conf\_t *conf;
391
392     conf = ngx\_palloc(cf->pool, sizeof(ngx\_http\_index\_loc\_conf\_t));
393     if (conf == NULL) {
394         return NULL;
395     }
396
397     conf->indices = NULL;
398     conf->max_index_len = 0;
399
400     return conf;
401 }
402
403
404 static char *

```



```

405 ngx_http_index_merge_loc_conf(ngx_conf_t *cf, void *parent, void *child)
406 {
407     ngx_http_index_loc_conf_t *prev = parent;
408     ngx_http_index_loc_conf_t *conf = child;
409
410     ngx_http_index_t *index;
411
412     if (conf->indices == NULL) {
413         conf->indices = prev->indices;
414         conf->max_index_len = prev->max_index_len;
415     }
416
417     if (conf->indices == NULL) {
418         conf->indices = ngx_array_create(cf->pool, 1, sizeof(ngx_http_index_t));
419         if (conf->indices == NULL) {
420             return NGX_CONF_ERROR;
421         }
422
423         index = ngx_array_push(conf->indices);
424         if (index == NULL) {
425             return NGX_CONF_ERROR;
426         }
427
428         index->name.len = sizeof(NGX_HTTP_DEFAULT_INDEX);
429         index->name.data = (u_char *) NGX_HTTP_DEFAULT_INDEX;
430         index->lengths = NULL;
431         index->values = NULL;
432
433         conf->max_index_len = sizeof(NGX_HTTP_DEFAULT_INDEX);
434
435         return NGX_CONF_OK;
436     }
437
438     return NGX_CONF_OK;
439 }
440
441
442 static ngx_int_t
443 ngx_http_index_init(ngx_conf_t *cf)
444 {
445     ngx_http_handler_pt *h;
446     ngx_http_core_main_conf_t *cmcf;
447
448     cmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_core_module);
449
450     h = ngx_array_push(&cmcf->phases[NGX_HTTP_CONTENT_PHASE].handlers);
451     if (h == NULL) {
452         return NGX_ERROR;
453     }
454
455     *h = ngx_http_index_handler;
456
457     return NGX_OK;
458 }
459
460
461 /* TODO: warn about duplicate indices */
462
463 static char *
464 ngx_http_index_set_index(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
465 {
466     ngx_http_index_loc_conf_t *ilcf = conf;
467
468     ngx_str_t *value;
469     ngx_uint_t i, n;
470     ngx_http_index_t *index;
471     ngx_http_script_compile_t sc;
472
473     if (ilcf->indices == NULL) {
474         ilcf->indices = ngx_array_create(cf->pool, 2, sizeof(ngx_http_index_t));
475         if (ilcf->indices == NULL) {
476             return NGX_CONF_ERROR;
477         }
478     }
479
480     value = cf->args->elts;

```

```

481 for (i = 1; i < cf->args->nelts; i++) {
482
483     if (value[i].data[0] == '/' && i != cf->args->nelts - 1) {
484         ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
485             "only the last index in \"index\" directive \"
486             \"should be absolute");
487     }
488
489     if (value[i].len == 0) {
490         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
491             "index \"%V\" in \"index\" directive is invalid",
492             &value[1]);
493         return NGX_CONF_ERROR;
494     }
495
496     index = ngx_array_push(ilcf->indices);
497     if (index == NULL) {
498         return NGX_CONF_ERROR;
499     }
500
501     index->name.len = value[i].len;
502     index->name.data = value[i].data;
503     index->lengths = NULL;
504     index->values = NULL;
505
506     n = ngx_http_script_variables_count(&value[i]);
507
508     if (n == 0) {
509         if (ilcf->max_index_len < index->name.len) {
510             ilcf->max_index_len = index->name.len;
511         }
512
513         if (index->name.data[0] == '/') {
514             continue;
515         }
516
517         /* include the terminating '\0' to the length to use ngx_memcpy() */
518         index->name.len++;
519
520         continue;
521     }
522
523     ngx_memzero(&sc, sizeof(ngx_http_script_compile_t));
524
525     sc.cf = cf;
526     sc.source = &value[i];
527     sc.lengths = &index->lengths;
528     sc.values = &index->values;
529     sc.variables = n;
530     sc.complete_lengths = 1;
531     sc.complete_values = 1;
532
533     if (ngx_http_script_compile(&sc) != NGX_OK) {
534         return NGX_CONF_ERROR;
535     }
536 }
537
538 return NGX_CONF_OK;
539 }
540

```

[One Level Up](#)

[Top Level](#)

## src/http/modules/nginx\_http\_log\_module.c - nginx-1.7.10

### Global variables defined

- [ngx\\_http\\_access\\_log](#)
- [ngx\\_http\\_combined\\_fmt](#)
- [ngx\\_http\\_log\\_commands](#)
- [ngx\\_http\\_log\\_module](#)
- [ngx\\_http\\_log\\_module\\_ctx](#)
- [ngx\\_http\\_log\\_vars](#)

### Data types defined

- [ngx\\_http\\_log\\_buf\\_t](#)
- [ngx\\_http\\_log\\_fmt\\_t](#)
- [ngx\\_http\\_log\\_loc\\_conf\\_t](#)
- [ngx\\_http\\_log\\_main\\_conf\\_t](#)
- [ngx\\_http\\_log\\_op\\_getlen\\_pt](#)
- [ngx\\_http\\_log\\_op\\_run\\_pt](#)
- [ngx\\_http\\_log\\_op\\_s](#)
- [ngx\\_http\\_log\\_op\\_t](#)
- [ngx\\_http\\_log\\_script\\_t](#)
- [ngx\\_http\\_log\\_t](#)
- [ngx\\_http\\_log\\_var\\_t](#)

### Functions defined

- [ngx\\_http\\_log\\_body\\_bytes\\_sent](#)
- [ngx\\_http\\_log\\_bytes\\_sent](#)
- [ngx\\_http\\_log\\_compile\\_format](#)
- [ngx\\_http\\_log\\_copy\\_long](#)
- [ngx\\_http\\_log\\_copy\\_short](#)
- [ngx\\_http\\_log\\_create\\_loc\\_conf](#)
- [ngx\\_http\\_log\\_create\\_main\\_conf](#)
- [ngx\\_http\\_log\\_escape](#)
- [ngx\\_http\\_log\\_flush](#)
- [ngx\\_http\\_log\\_flush\\_handler](#)

- [ngx\\_http\\_log\\_gzip](#)
- [ngx\\_http\\_log\\_gzip\\_alloc](#)
- [ngx\\_http\\_log\\_gzip\\_free](#)
- [ngx\\_http\\_log\\_handler](#)
- [ngx\\_http\\_log\\_init](#)
- [ngx\\_http\\_log\\_iso8601](#)
- [ngx\\_http\\_log\\_merge\\_loc\\_conf](#)
- [ngx\\_http\\_log\\_msec](#)
- [ngx\\_http\\_log\\_open\\_file\\_cache](#)
- [ngx\\_http\\_log\\_pipe](#)
- [ngx\\_http\\_log\\_request\\_length](#)
- [ngx\\_http\\_log\\_request\\_time](#)
- [ngx\\_http\\_log\\_script\\_write](#)
- [ngx\\_http\\_log\\_set\\_format](#)
- [ngx\\_http\\_log\\_set\\_log](#)
- [ngx\\_http\\_log\\_status](#)
- [ngx\\_http\\_log\\_time](#)
- [ngx\\_http\\_log\\_variable](#)
- [ngx\\_http\\_log\\_variable\\_compile](#)
- [ngx\\_http\\_log\\_variable\\_getlen](#)
- [ngx\\_http\\_log\\_write](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12 #if (NGX_ZLIB)
13 #include <zlib.h>
14 #endif
15
16
17 typedef struct ngx_http_log_op_s ngx_http_log_op_t;
18
19 typedef u_char *(*ngx_http_log_op_run_pt) (ngx_http_request_t *r, u_char *buf,
20     ngx_http_log_op_t *op);
21
22 typedef size_t (*ngx_http_log_op_getlen_pt) (ngx_http_request_t *r,
23     uintptr_t data);
24
25
```

```

26 struct ngx_http_log_op_s {
27     size_t                len;
28     ngx_http_log_op_getlen_pt getlen;
29     ngx_http_log_op_run_pt  run;
30     uintptr_t             data;
31 };
32
33
34 typedef struct {
35     ngx_str_t             name;
36     ngx_array_t          *flushes;
37     ngx_array_t          *ops;           /* array of ngx_http_log_op_t */
38 } ngx_http_log_fmt_t;
39
40
41 typedef struct {
42     ngx_array_t          formats;       /* array of ngx_http_log_fmt_t */
43     ngx_uint_t          combined_used; /* unsigned combined_used:1 */
44 } ngx_http_log_main_conf_t;
45
46
47 typedef struct {
48     u_char              *start;
49     u_char              *pos;
50     u_char              *last;
51
52     ngx_event_t        *event;
53     ngx_msec_t         flush;
54     ngx_int_t          gzip;
55 } ngx_http_log_buf_t;
56
57
58 typedef struct {
59     ngx_array_t          *lengths;
60     ngx_array_t          *values;
61 } ngx_http_log_script_t;
62
63
64 typedef struct {
65     ngx_open_file_t     *file;
66     ngx_http_log_script_t *script;
67     time_t              disk_full_time;
68     time_t              error_log_time;
69     ngx_syslog_peer_t   *syslog_peer;
70     ngx_http_log_fmt_t  *format;
71     ngx_http_complex_value_t *filter;
72 } ngx_http_log_t;
73
74
75 typedef struct {
76     ngx_array_t          *logs;         /* array of ngx_http_log_t */
77
78     ngx_open_file_cache_t *open_file_cache;
79     time_t              open_file_cache_valid;
80     ngx_uint_t          open_file_cache_min_uses;
81
82     ngx_uint_t          off;           /* unsigned off:1 */
83 } ngx_http_log_loc_conf_t;
84
85
86 typedef struct {
87     ngx_str_t             name;
88     size_t               len;
89     ngx_http_log_op_run_pt run;
90 } ngx_http_log_var_t;
91
92
93 static void ngx_http_log_write(ngx_http_request_t *r, ngx_http_log_t *log,
94     u_char *buf, size_t len);
95 static ssize_t ngx_http_log_script_write(ngx_http_request_t *r,
96     ngx_http_log_script_t *script, u_char **name, u_char *buf, size_t len);
97
98 #if (NGX_ZLIB)
99 static ssize_t ngx_http_log_gzip(ngx_fd_t fd, u_char *buf, size_t len,
100     ngx_int_t level, ngx_log_t *log);
101

```

```

102 static void ngx\_http\_log\_gzip\_alloc(void *opaque, u_int items, u_int size);
103 static void ngx\_http\_log\_gzip\_free(void *opaque, void *address);
104 #endif
105
106 static void ngx\_http\_log\_flush(ngx\_open\_file\_t *file, ngx\_log\_t *log);
107 static void ngx\_http\_log\_flush\_handler(ngx\_event\_t *ev);
108
109 static u_char ngx\_http\_log\_pipe(ngx\_http\_request\_t *r, u_char *buf,
110     ngx\_http\_log\_op\_t *op);
111 static u_char ngx\_http\_log\_time(ngx\_http\_request\_t *r, u_char *buf,
112     ngx\_http\_log\_op\_t *op);
113 static u_char ngx\_http\_log\_iso8601(ngx\_http\_request\_t *r, u_char *buf,
114     ngx\_http\_log\_op\_t *op);
115 static u_char ngx\_http\_log\_msec(ngx\_http\_request\_t *r, u_char *buf,
116     ngx\_http\_log\_op\_t *op);
117 static u_char ngx\_http\_log\_request\_time(ngx\_http\_request\_t *r, u_char *buf,
118     ngx\_http\_log\_op\_t *op);
119 static u_char ngx\_http\_log\_status(ngx\_http\_request\_t *r, u_char *buf,
120     ngx\_http\_log\_op\_t *op);
121 static u_char ngx\_http\_log\_bytes\_sent(ngx\_http\_request\_t *r, u_char *buf,
122     ngx\_http\_log\_op\_t *op);
123 static u_char ngx\_http\_log\_body\_bytes\_sent(ngx\_http\_request\_t *r,
124     u_char *buf, ngx\_http\_log\_op\_t *op);
125 static u_char ngx\_http\_log\_request\_length(ngx\_http\_request\_t *r, u_char *buf,
126     ngx\_http\_log\_op\_t *op);
127
128 static ngx\_int\_t ngx\_http\_log\_variable\_compile(ngx\_conf\_t *cf,
129     ngx\_http\_log\_op\_t *op, ngx\_str\_t *value);
130 static size_t ngx\_http\_log\_variable\_getlen(ngx\_http\_request\_t *r,
131     uintptr_t data);
132 static u_char ngx\_http\_log\_variable(ngx\_http\_request\_t *r, u_char *buf,
133     ngx\_http\_log\_op\_t *op);
134 static uintptr_t ngx\_http\_log\_escape(u_char *dst, u_char *src, size_t size);
135
136
137 static void ngx\_http\_log\_create\_main\_conf(ngx\_conf\_t *cf);
138 static void ngx\_http\_log\_create\_loc\_conf(ngx\_conf\_t *cf);
139 static char ngx\_http\_log\_merge\_loc\_conf(ngx\_conf\_t *cf, void *parent,
140     void *child);
141 static char ngx\_http\_log\_set\_log(ngx\_conf\_t *cf, ngx\_command\_t *cmd,
142     void *conf);
143 static char ngx\_http\_log\_set\_format(ngx\_conf\_t *cf, ngx\_command\_t *cmd,
144     void *conf);
145 static char ngx\_http\_log\_compile\_format(ngx\_conf\_t *cf,
146     ngx\_array\_t *flushes, ngx\_array\_t *ops, ngx\_array\_t *args, ngx\_uint\_t s);
147 static char ngx\_http\_log\_open\_file\_cache(ngx\_conf\_t *cf, ngx\_command\_t *cmd,
148     void *conf);
149 static ngx\_int\_t ngx\_http\_log\_init(ngx\_conf\_t *cf);
150
151
152 static ngx\_command\_t ngx\_http\_log\_commands[] = {
153
154     { ngx\_string("log_format"),
155       NGX\_HTTP\_MAIN\_CONF|NGX\_CONF\_2MORE,
156       ngx\_http\_log\_set\_format,
157       NGX\_HTTP\_MAIN\_CONF\_OFFSET,
158       0,
159       NULL },
160
161     { ngx\_string("access_log"),
162       NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_HTTP\_LIF\_CONF
163       |NGX\_HTTP\_LMT\_CONF|NGX\_CONF\_1MORE,
164       ngx\_http\_log\_set\_log,
165       NGX\_HTTP\_LOC\_CONF\_OFFSET,
166       0,
167       NULL },
168
169     { ngx\_string("open_log_file_cache"),
170       NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1234,
171       ngx\_http\_log\_open\_file\_cache,
172       NGX\_HTTP\_LOC\_CONF\_OFFSET,
173       0,
174       NULL },
175
176     ngx\_null\_command
177 };

```

```

178
179
180 static ngx_http_module_t ngx_http_log_module_ctx = {
181     NULL,                                     /* preconfiguration */
182     ngx_http_log_init,                       /* postconfiguration */
183
184     ngx_http_log_create_main_conf,          /* create main configuration */
185     NULL,                                   /* init main configuration */
186
187     NULL,                                   /* create server configuration */
188     NULL,                                   /* merge server configuration */
189
190     ngx_http_log_create_loc_conf,          /* create location configuration */
191     ngx_http_log_merge_loc_conf          /* merge location configuration */
192 };
193
194
195 ngx_module_t ngx_http_log_module = {
196     NGX_MODULE_V1,
197     &ngx_http_log_module_ctx,             /* module context */
198     ngx_http_log_commands,               /* module directives */
199     NGX_HTTP_MODULE,                     /* module type */
200     NULL,                                 /* init master */
201     NULL,                                 /* init module */
202     NULL,                                 /* init process */
203     NULL,                                 /* init thread */
204     NULL,                                 /* exit thread */
205     NULL,                                 /* exit process */
206     NULL,                                 /* exit master */
207     NGX_MODULE_V1_PADDING
208 };
209
210
211 static ngx_str_t ngx_http_access_log = ngx_string(NGX_HTTP_LOG_PATH);
212
213
214 static ngx_str_t ngx_http_combined_fmt =
215     ngx_string("$remote_addr - $remote_user [$time_local] "
216               "\"$request\" $status $body_bytes_sent "
217               "\"$http_referer\" \"$http_user_agent\"");
218
219
220 static ngx_http_log_var_t ngx_http_log_vars[] = {
221     { ngx_string("pipe"), 1, ngx_http_log_pipe },
222     { ngx_string("time_local"), sizeof("28/Sep/1970:12:00:00 +0600") - 1,
223       ngx_http_log_time },
224     { ngx_string("time_iso8601"), sizeof("1970-09-28T12:00:00+06:00") - 1,
225       ngx_http_log_iso8601 },
226     { ngx_string("msec"), NGX_TIME_T_LEN + 4, ngx_http_log_msec },
227     { ngx_string("request_time"), NGX_TIME_T_LEN + 4,
228       ngx_http_log_request_time },
229     { ngx_string("status"), NGX_INT_T_LEN, ngx_http_log_status },
230     { ngx_string("bytes_sent"), NGX_OFF_T_LEN, ngx_http_log_bytes_sent },
231     { ngx_string("body_bytes_sent"), NGX_OFF_T_LEN,
232       ngx_http_log_body_bytes_sent },
233     { ngx_string("request_length"), NGX_SIZE_T_LEN,
234       ngx_http_log_request_length },
235
236     { ngx_null_string, 0, NULL }
237 };
238
239
240 static ngx_int_t
241 ngx_http_log_handler(ngx_http_request_t *r)
242 {
243     u_char          *line, *p;
244     size_t          len, size;
245     ssize_t         n;
246     ngx_str_t       val;
247     ngx_uint_t      i, l;
248     ngx_http_log_t  *log;
249     ngx_http_log_op_t *op;
250     ngx_http_log_buf_t *buffer;
251     ngx_http_log_loc_conf_t *lcf;
252
253     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,

```

```

254         "http log handler");
255
256 lcf = ngx_http_get_module_loc_conf(r, ngx_http_log_module);
257
258 if (lcf->off) {
259     return NGX_OK;
260 }
261
262 log = lcf->logs->elts;
263 for (l = 0; l < lcf->logs->nelts; l++) {
264
265     if (log[l].filter) {
266         if (ngx_http_complex_value(r, log[l].filter, &val) != NGX_OK) {
267             return NGX_ERROR;
268         }
269
270         if (val.len == 0 || (val.len == 1 && val.data[0] == '0')) {
271             continue;
272         }
273     }
274
275     if (ngx_time() == log[l].disk_full_time) {
276
277         /*
278          * on FreeBSD writing to a full filesystem with enabled softupdates
279          * may block process for much longer time than writing to non-full
280          * filesystem, so we skip writing to a log for one second
281          */
282
283         continue;
284     }
285
286     ngx_http_script_flush_no_cacheable_variables(r, log[l].format->flushes);
287
288     len = 0;
289     op = log[l].format->ops->elts;
290     for (i = 0; i < log[l].format->ops->nelts; i++) {
291         if (op[i].len == 0) {
292             len += op[i].getlen(r, op[i].data);
293
294         } else {
295             len += op[i].len;
296         }
297     }
298
299     if (log[l].syslog_peer) {
300
301         /* length of syslog's PRI and HEADER message parts */
302         len += sizeof("<255>Jan 01 00:00:00 ") - 1
303             + ngx_cycle->hostname.len + 1
304             + log[l].syslog_peer->tag.len + 2;
305
306         goto alloc_line;
307     }
308
309     len += NGX_LINEFEED_SIZE;
310
311     buffer = log[l].file ? log[l].file->data : NULL;
312
313     if (buffer) {
314
315         if (len > (size_t) (buffer->last - buffer->pos)) {
316
317             ngx_http_log_write(r, &log[l], buffer->start,
318                 buffer->pos - buffer->start);
319
320             buffer->pos = buffer->start;
321         }
322
323         if (len <= (size_t) (buffer->last - buffer->pos)) {
324
325             p = buffer->pos;
326
327             if (buffer->event && p == buffer->start) {
328                 ngx_add_timer(buffer->event, buffer->flush);
329             }

```



```

330         for (i = 0; i < log[l].format->ops->nelts; i++) {
331             p = op[i].run(r, p, &op[i]);
332         }
333
334         ngx_linefeed(p);
335
336         buffer->pos = p;
337
338         continue;
339     }
340
341     if (buffer->event && buffer->event->timer_set) {
342         ngx_del_timer(buffer->event);
343     }
344 }
345
346 alloc_line:
347
348     line = ngx_pnalloc(r->pool, len);
349     if (line == NULL) {
350         return NGX_ERROR;
351     }
352
353     p = line;
354
355     if (log[l].syslog_peer) {
356         p = ngx_syslog_add_header(log[l].syslog_peer, line);
357     }
358
359     for (i = 0; i < log[l].format->ops->nelts; i++) {
360         p = op[i].run(r, p, &op[i]);
361     }
362
363     if (log[l].syslog_peer) {
364         size = p - line;
365
366         n = ngx_syslog_send(log[l].syslog_peer, line, size);
367
368         if (n < 0) {
369             ngx_log_error(NGX_LOG_WARN, r->connection->log, 0,
370                 "send() to syslog failed");
371         } else if ((size_t) n != size) {
372             ngx_log_error(NGX_LOG_WARN, r->connection->log, 0,
373                 "send() to syslog has written only %z of %uz",
374                 n, size);
375         }
376     }
377
378     continue;
379 }
380
381     ngx_linefeed(p);
382
383     ngx_http_log_write(r, &log[l], line, p - line);
384 }
385
386 return NGX_OK;
387 }
388
389
390
391
392 static void
393 ngx_http_log_write(ngx_http_request_t *r, ngx_http_log_t *log, u_char *buf,
394     size_t len)
395 {
396     u_char          *name;
397     time_t          now;
398     ssize_t         n;
399     ngx_err_t       err;
400     #if (NGX_ZLIB)
401     ngx_http_log_buf_t *buffer;
402     #endif
403
404     if (log->script == NULL) {
405         name = log->file->name.data;

```

```

406
407 #if (NGX_ZLIB)
408     buffer = log->file->data;
409
410     if (buffer && buffer->gzip) {
411         n = ngx_http_log_gzip(log->file->fd, buf, len, buffer->gzip,
412                               r->connection->log);
413     } else {
414         n = ngx_write_fd(log->file->fd, buf, len);
415     }
416 #else
417     n = ngx_write_fd(log->file->fd, buf, len);
418 #endif
419
420 } else {
421     name = NULL;
422     n = ngx_http_log_script_write(r, log->script, &name, buf, len);
423 }
424
425 if (n == (ssize_t) len) {
426     return;
427 }
428
429 now = ngx_time();
430
431 if (n == -1) {
432     err = ngx_errno;
433
434     if (err == NGX_ENOSPC) {
435         log->disk_full_time = now;
436     }
437
438     if (now - log->error_log_time > 59) {
439         ngx_log_error(NGX_LOG_ALERT, r->connection->log, err,
440                       ngx_write_fd_n " to \"%s\" failed", name);
441
442         log->error_log_time = now;
443     }
444
445     return;
446 }
447
448 if (now - log->error_log_time > 59) {
449     ngx_log_error(NGX_LOG_ALERT, r->connection->log, 0,
450                 ngx_write_fd_n " to \"%s\" was incomplete: %z of %uz",
451                 name, n, len);
452
453     log->error_log_time = now;
454 }
455 }
456
457
458 static ssize_t
459 ngx_http_log_script_write(ngx_http_request_t *r, ngx_http_log_script_t *script,
460                          u_char **name, u_char *buf, size_t len)
461 {
462     size_t          root;
463     ssize_t         n;
464     ngx_str_t       log, path;
465     ngx_open_file_info_t of;
466     ngx_http_log_loc_conf_t *llcf;
467     ngx_http_core_loc_conf_t *clcf;
468
469     clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
470
471     if (!r->root_tested) {
472
473         /* test root directory existence */
474
475         if (ngx_http_map_uri_to_path(r, &path, &root, 0) == NULL) {
476             /* simulate successful logging */
477             return len;
478         }
479
480         path.data[root] = '\0';
481

```

```

482     ngx_memzero(&of, sizeof(ngx_open_file_info_t));
483
484     of.valid = clcf->open_file_cache_valid;
485     of.min_uses = clcf->open_file_cache_min_uses;
486     of.test_dir = 1;
487     of.test_only = 1;
488     of.errors = clcf->open_file_cache_errors;
489     of.events = clcf->open_file_cache_events;
490
491     if (ngx_http_set_disable_symlinks(r, clcf, &path, &of) != NGX_OK) {
492         /* simulate successful logging */
493         return len;
494     }
495
496     if (ngx_open_cached_file(clcf->open_file_cache, &path, &of, r->pool)
497         != NGX_OK)
498     {
499         if (of.err == 0) {
500             /* simulate successful logging */
501             return len;
502         }
503
504         ngx_log_error(NGX_LOG_ERR, r->connection->log, of.err,
505             "testing \"%s\" existence failed", path.data);
506
507         /* simulate successful logging */
508         return len;
509     }
510
511     if (!of.is_dir) {
512         ngx_log_error(NGX_LOG_ERR, r->connection->log, NGX_ENOTDIR,
513             "testing \"%s\" existence failed", path.data);
514
515         /* simulate successful logging */
516         return len;
517     }
518 }
519
520 if (ngx_http_script_run(r, &log, script->lengths->elts, 1,
521     script->values->elts)
522     == NULL)
523 {
524     /* simulate successful logging */
525     return len;
526 }
527
528 log.data[log.len - 1] = '\0';
529 *name = log.data;
530
531 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
532     "http log \"%s\"", log.data);
533
534 llcf = ngx_http_get_module_loc_conf(r, ngx_http_log_module);
535
536 ngx_memzero(&of, sizeof(ngx_open_file_info_t));
537
538 of.log = 1;
539 of.valid = llcf->open_file_cache_valid;
540 of.min_uses = llcf->open_file_cache_min_uses;
541 of.directio = NGX_OPEN_FILE_DIRECTIO_OFF;
542
543 if (ngx_http_set_disable_symlinks(r, clcf, &log, &of) != NGX_OK) {
544     /* simulate successful logging */
545     return len;
546 }
547
548 if (ngx_open_cached_file(llcf->open_file_cache, &log, &of, r->pool)
549     != NGX_OK)
550 {
551     ngx_log_error(NGX_LOG_CRIT, r->connection->log, ngx_errno,
552         "%s \"%s\" failed", of.failed, log.data);
553     /* simulate successful logging */
554     return len;
555 }
556
557 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,

```

```

558         "http log #%d", of.fd);
559
560     n = ngx_write_fd(of.fd, buf, len);
561
562     return n;
563 }
564
565
566 #if (NGX_ZLIB)
567
568 static ssize_t
569 ngx_http_log_gzip(ngx_fd_t fd, u_char *buf, size_t len, ngx_int_t level,
570                 ngx_log_t *log)
571 {
572     int          rc, wbits, memlevel;
573     u_char      *out;
574     size_t      size;
575     ssize_t     n;
576     z_stream     zstream;
577     ngx_err_t   err;
578     ngx_pool_t  *pool;
579
580     wbits = MAX_WBITS;
581     memlevel = MAX_MEM_LEVEL - 1;
582
583     while ((ssize_t) len < ((1 << (wbits - 1)) - 262)) {
584         wbits--;
585         memlevel--;
586     }
587
588     /*
589      * This is a formula from deflateBound() for conservative upper bound of
590      * compressed data plus 18 bytes of gzip wrapper.
591      */
592
593     size = len + ((len + 7) >> 3) + ((len + 63) >> 6) + 5 + 18;
594
595     ngx_memzero(&zstream, sizeof(z_stream));
596
597     pool = ngx_create_pool(256, log);
598     if (pool == NULL) {
599         /* simulate successful logging */
600         return len;
601     }
602
603     pool->log = log;
604
605     zstream.zalloc = ngx_http_log_gzip_alloc;
606     zstream.zfree = ngx_http_log_gzip_free;
607     zstream.opaque = pool;
608
609     out = ngx_pnalloc(pool, size);
610     if (out == NULL) {
611         goto done;
612     }
613
614     zstream.next_in = buf;
615     zstream.avail_in = len;
616     zstream.next_out = out;
617     zstream.avail_out = size;
618
619     rc = deflateInit2(&zstream, (int) level, Z_DEFLATED, wbits + 16, memlevel,
620                    Z_DEFAULT_STRATEGY);
621
622     if (rc != Z_OK) {
623         ngx_log_error(NGX_LOG_ALERT, log, 0, "deflateInit2() failed: %d", rc);
624         goto done;
625     }
626
627     ngx_log_debug4(NGX_LOG_DEBUG_HTTP, log, 0,
628                  "deflate in: ni:%p no:%p ai:%ud ao:%ud",
629                  zstream.next_in, zstream.next_out,
630                  zstream.avail_in, zstream.avail_out);
631
632     rc = deflate(&zstream, Z_FINISH);
633

```

```

634     if (rc != Z_STREAM_END) {
635         ngx_log_error(NGX_LOG_ALERT, log, 0,
636             "deflate(Z_FINISH) failed: %d", rc);
637         goto done;
638     }
639
640     ngx_log_debug5(NGX_LOG_DEBUG_HTTP, log, 0,
641         "deflate out: ni:%p no:%p ai:%ud ao:%ud rc:%d",
642         zstream.next_in, zstream.next_out,
643         zstream.avail_in, zstream.avail_out,
644         rc);
645
646     size -= zstream.avail_out;
647
648     rc = deflateEnd(&zstream);
649
650     if (rc != Z_OK) {
651         ngx_log_error(NGX_LOG_ALERT, log, 0, "deflateEnd() failed: %d", rc);
652         goto done;
653     }
654
655     n = ngx_write_fd(fd, out, size);
656
657     if (n != (ssize_t) size) {
658         err = (n == -1) ? ngx_errno : 0;
659
660         ngx_destroy_pool(pool);
661
662         ngx_set_errno(err);
663         return -1;
664     }
665
666 done:
667
668     ngx_destroy_pool(pool);
669
670     /* simulate successful logging */
671     return len;
672 }
673
674
675 static void *
676 ngx_http_log_gzip_alloc(void *opaque, u_int items, u_int size)
677 {
678     ngx_pool_t *pool = opaque;
679
680     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, pool->log, 0,
681         "gzip alloc: n:%ud s:%ud", items, size);
682
683     return ngx_palloc(pool, items * size);
684 }
685
686
687 static void
688 ngx_http_log_gzip_free(void *opaque, void *address)
689 {
690     #if 0
691         ngx_pool_t *pool = opaque;
692
693         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, pool->log, 0, "gzip free: %p", address);
694     #endif
695 }
696
697 #endif
698
699
700 static void
701 ngx_http_log_flush(ngx_open_file_t *file, ngx_log_t *log)
702 {
703     size_t          len;
704     ssize_t         n;
705     ngx_http_log_buf_t *buffer;
706
707     buffer = file->data;
708
709     len = buffer->pos - buffer->start;

```

```

710     if (len == 0) {
711         return;
712     }
713 }
714
715 #if (NGX_ZLIB)
716     if (buffer->gzip) {
717         n = ngx_http_log_gzip(file->fd, buffer->start, len, buffer->gzip, log);
718     } else {
719         n = ngx_write_fd(file->fd, buffer->start, len);
720     }
721 #else
722     n = ngx_write_fd(file->fd, buffer->start, len);
723 #endif
724
725     if (n == -1) {
726         ngx_log_error(NGX_LOG_ALERT, log, ngx_errno,
727                     ngx_write_fd_n " to \"%s\" failed",
728                     file->name.data);
729     } else if ((size_t) n != len) {
730         ngx_log_error(NGX_LOG_ALERT, log, 0,
731                     ngx_write_fd_n " to \"%s\" was incomplete: %z of %uz",
732                     file->name.data, n, len);
733     }
734 }
735
736     buffer->pos = buffer->start;
737
738     if (buffer->event && buffer->event->timer_set) {
739         ngx_del_timer(buffer->event);
740     }
741 }
742
743
744 static void
745 ngx_http_log_flush_handler(ngx_event_t *ev)
746 {
747     ngx_open_file_t    *file;
748     ngx_http_log_buf_t *buffer;
749
750     ngx_log_debug0(NGX_LOG_DEBUG_EVENT, ev->log, 0,
751                  "http log buffer flush handler");
752
753     if (ev->timedout) {
754         ngx_http_log_flush(ev->data, ev->log);
755         return;
756     }
757
758     /* cancel the flush timer for graceful shutdown */
759
760     file = ev->data;
761     buffer = file->data;
762
763     buffer->event = NULL;
764 }
765
766
767 static u_char *
768 ngx_http_log_copy_short(ngx_http_request_t *r, u_char *buf,
769                        ngx_http_log_op_t *op)
770 {
771     size_t    len;
772     uintptr_t data;
773
774     len = op->len;
775     data = op->data;
776
777     while (len-- > 0) {
778         *buf++ = (u_char) (data & 0xff);
779         data >>= 8;
780     }
781
782     return buf;
783 }
784
785

```

```

786 static u_char *
787 ngx_http_log_copy_long(ngx_http_request_t *r, u_char *buf,
788     ngx_http_log_op_t *op)
789 {
790     return ngx_cpymem(buf, (u_char *) op->data, op->len);
791 }
792
793
794 static u_char *
795 ngx_http_log_pipe(ngx_http_request_t *r, u_char *buf, ngx_http_log_op_t *op)
796 {
797     if (r->pipeline) {
798         *buf = 'p';
799     } else {
800         *buf = '.';
801     }
802
803     return buf + 1;
804 }
805
806
807 static u_char *
808 ngx_http_log_time(ngx_http_request_t *r, u_char *buf, ngx_http_log_op_t *op)
809 {
810     return ngx_cpymem(buf, ngx_cached_http_log_time.data,
811         ngx_cached_http_log_time.len);
812 }
813
814 static u_char *
815 ngx_http_log_iso8601(ngx_http_request_t *r, u_char *buf, ngx_http_log_op_t *op)
816 {
817     return ngx_cpymem(buf, ngx_cached_http_log_iso8601.data,
818         ngx_cached_http_log_iso8601.len);
819 }
820
821 static u_char *
822 ngx_http_log_msec(ngx_http_request_t *r, u_char *buf, ngx_http_log_op_t *op)
823 {
824     ngx_time_t *tp;
825
826     tp = ngx_timeofday();
827
828     return ngx_sprintf(buf, "%T.%03M", tp->sec, tp->msec);
829 }
830
831
832 static u_char *
833 ngx_http_log_request_time(ngx_http_request_t *r, u_char *buf,
834     ngx_http_log_op_t *op)
835 {
836     ngx_time_t *tp;
837     ngx_msec_int_t ms;
838
839     tp = ngx_timeofday();
840
841     ms = (ngx_msec_int_t)
842         ((tp->sec - r->start_sec) * 1000 + (tp->msec - r->start_msec));
843     ms = ngx_max(ms, 0);
844
845     return ngx_sprintf(buf, "%T.%03M", (time_t) ms / 1000, ms % 1000);
846 }
847
848
849 static u_char *
850 ngx_http_log_status(ngx_http_request_t *r, u_char *buf, ngx_http_log_op_t *op)
851 {
852     ngx_uint_t status;
853
854     if (r->err_status) {
855         status = r->err_status;
856     } else if (r->headers_out.status) {
857         status = r->headers_out.status;
858     } else if (r->http_version == NGX_HTTP_VERSION_9) {
859         status = 9;
860     }
861 }

```

```

862     } else {
863         status = 0;
864     }
865 }
866
867 return ngx_sprintf(buf, "%03ui", status);
868 }
869
870
871 static u_char *
872 ngx_http_log_bytes_sent(ngx_http_request_t *r, u_char *buf,
873     ngx_http_log_op_t *op)
874 {
875     return ngx_sprintf(buf, "%0", r->connection->sent);
876 }
877
878
879 /*
880  * although there is a real $body_bytes_sent variable,
881  * this log operation code function is more optimized for logging
882  */
883
884 static u_char *
885 ngx_http_log_body_bytes_sent(ngx_http_request_t *r, u_char *buf,
886     ngx_http_log_op_t *op)
887 {
888     off_t length;
889
890     length = r->connection->sent - r->header_size;
891
892     if (length > 0) {
893         return ngx_sprintf(buf, "%0", length);
894     }
895
896     *buf = '0';
897
898     return buf + 1;
899 }
900
901
902 static u_char *
903 ngx_http_log_request_length(ngx_http_request_t *r, u_char *buf,
904     ngx_http_log_op_t *op)
905 {
906     return ngx_sprintf(buf, "%0", r->request_length);
907 }
908
909
910 static ngx_int_t
911 ngx_http_log_variable_compile(ngx_conf_t *cf, ngx_http_log_op_t *op,
912     ngx_str_t *value)
913 {
914     ngx_int_t index;
915
916     index = ngx_http_get_variable_index(cf, value);
917     if (index == NGX_ERROR) {
918         return NGX_ERROR;
919     }
920
921     op->len = 0;
922     op->getlen = ngx_http_log_variable_getlen;
923     op->run = ngx_http_log_variable;
924     op->data = index;
925
926     return NGX_OK;
927 }
928
929
930 static size_t
931 ngx_http_log_variable_getlen(ngx_http_request_t *r, uintptr_t data)
932 {
933     uintptr_t len;
934     ngx_http_variable_value_t *value;
935
936     value = ngx_http_get_indexed_variable(r, data);
937

```



```

938     if (value == NULL || value->not_found) {
939         return 1;
940     }
941
942     len = ngx_http_log_escape(NULL, value->data, value->len);
943
944     value->escape = len ? 1 : 0;
945
946     return value->len + len * 3;
947 }
948
949
950 static u_char *
951 ngx_http_log_variable(ngx_http_request_t *r, u_char *buf, ngx_http_log_op_t *op)
952 {
953     ngx_http_variable_value_t *value;
954
955     value = ngx_http_get_indexed_variable(r, op->data);
956
957     if (value == NULL || value->not_found) {
958         *buf = '-';
959         return buf + 1;
960     }
961
962     if (value->escape == 0) {
963         return ngx_cpymem(buf, value->data, value->len);
964     } else {
965         return (u_char *) ngx_http_log_escape(buf, value->data, value->len);
966     }
967 }
968 }
969
970
971 static uintptr_t
972 ngx_http_log_escape(u_char *dst, u_char *src, size_t size)
973 {
974     ngx_uint_t    n;
975     static u_char  hex[] = "0123456789ABCDEF";
976
977     static uint32_t  escape[] = {
978         0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
979
980         /* ?>= < ;:98 7654 3210 /.-, +*)('&%"$ #"! */
981         0x00000004, /* 0000 0000 0000 0000 0000 0000 0000 0100 */
982
983         /* _^]\ [ZYX WVUT SRQP ONML KJIH GFED CBA@ */
984         0x10000000, /* 0001 0000 0000 0000 0000 0000 0000 0000 */
985
986         /* ~}| {zyx wvut srqp onml kjih gfed cba` */
987         0x80000000, /* 1000 0000 0000 0000 0000 0000 0000 0000 */
988
989         0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
990         0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
991         0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
992         0xffffffff, /* 1111 1111 1111 1111 1111 1111 1111 1111 */
993     };
994
995
996     if (dst == NULL) {
997
998         /* find the number of the characters to be escaped */
999
1000         n = 0;
1001
1002         while (size) {
1003             if (escape[*src >> 5] & (1 << (*src & 0x1f))) {
1004                 n++;
1005             }
1006             src++;
1007             size--;
1008         }
1009
1010         return (uintptr_t) n;
1011     }
1012
1013     while (size) {

```

```

1014     if (escape[*src >> 5] & (1 << (*src & 0x1f))) {
1015         *dst++ = '\\';
1016         *dst++ = 'x';
1017         *dst++ = hex[*src >> 4];
1018         *dst++ = hex[*src & 0xf];
1019         src++;
1020
1021     } else {
1022         *dst++ = *src++;
1023     }
1024     size--;
1025 }
1026
1027 return (uintptr_t) dst;
1028 }
1029
1030
1031 static void *
1032 ngx_http_log_create_main_conf(ngx_conf_t *cf)
1033 {
1034     ngx_http_log_main_conf_t *conf;
1035
1036     ngx_http_log_fmt_t *fmt;
1037
1038     conf = ngx_palloc(cf->pool, sizeof(ngx_http_log_main_conf_t));
1039     if (conf == NULL) {
1040         return NULL;
1041     }
1042
1043     if (ngx_array_init(&conf->formats, cf->pool, 4, sizeof(ngx_http_log_fmt_t))
1044         != NGX_OK)
1045     {
1046         return NULL;
1047     }
1048
1049     fmt = ngx_array_push(&conf->formats);
1050     if (fmt == NULL) {
1051         return NULL;
1052     }
1053
1054     ngx_str_set(&fmt->name, "combined");
1055
1056     fmt->flushes = NULL;
1057
1058     fmt->ops = ngx_array_create(cf->pool, 16, sizeof(ngx_http_log_op_t));
1059     if (fmt->ops == NULL) {
1060         return NULL;
1061     }
1062
1063     return conf;
1064 }
1065
1066
1067 static void *
1068 ngx_http_log_create_loc_conf(ngx_conf_t *cf)
1069 {
1070     ngx_http_log_loc_conf_t *conf;
1071
1072     conf = ngx_palloc(cf->pool, sizeof(ngx_http_log_loc_conf_t));
1073     if (conf == NULL) {
1074         return NULL;
1075     }
1076
1077     conf->open_file_cache = NGX_CONF_UNSET_PTR;
1078
1079     return conf;
1080 }
1081
1082
1083 static char *
1084 ngx_http_log_merge_loc_conf(ngx_conf_t *cf, void *parent, void *child)
1085 {
1086     ngx_http_log_loc_conf_t *prev = parent;
1087     ngx_http_log_loc_conf_t *conf = child;
1088
1089     ngx_http_log_t *log;

```

```

1090     ngx_http_log_fmt_t      *fmt;
1091     ngx_http_log_main_conf_t *lmcf;
1092
1093     if (conf->open_file_cache == NGX\_CONF\_UNSET\_PTR) {
1094
1095         conf->open_file_cache = prev->open_file_cache;
1096         conf->open_file_cache_valid = prev->open_file_cache_valid;
1097         conf->open_file_cache_min_uses = prev->open_file_cache_min_uses;
1098
1099         if (conf->open_file_cache == NGX\_CONF\_UNSET\_PTR) {
1100             conf->open_file_cache = NULL;
1101         }
1102     }
1103
1104     if (conf->logs || conf->off) {
1105         return NGX\_CONF\_OK;
1106     }
1107
1108     conf->logs = prev->logs;
1109     conf->off = prev->off;
1110
1111     if (conf->logs || conf->off) {
1112         return NGX\_CONF\_OK;
1113     }
1114
1115     conf->logs = ngx\_array\_create(cf->pool, 2, sizeof\(ngx\_http\_log\_t\));
1116     if (conf->logs == NULL) {
1117         return NGX\_CONF\_ERROR;
1118     }
1119
1120     log = ngx\_array\_push(conf->logs);
1121     if (log == NULL) {
1122         return NGX\_CONF\_ERROR;
1123     }
1124
1125     ngx\_memzero(log, sizeof\(ngx\_http\_log\_t\));
1126
1127     log->file = ngx\_conf\_open\_file(cf->cycle, &ngx\_http\_access\_log);
1128     if (log->file == NULL) {
1129         return NGX\_CONF\_ERROR;
1130     }
1131
1132     lmcf = ngx\_http\_conf\_get\_module\_main\_conf(cf, ngx\_http\_log\_module);
1133     fmt = lmcf->formats.elts;
1134
1135     /* the default "combined" format */
1136     log->format = &fmt[0];
1137     lmcf->combined_used = 1;
1138
1139     return NGX\_CONF\_OK;
1140 }
1141
1142
1143 static char *
1144 ngx_http_log_set_log(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
1145 {
1146     ngx\_http\_log\_loc\_conf\_t *llcf = conf;
1147
1148     ssize\_t                size;
1149     ngx\_int\_t             gzip;
1150     ngx\_uint\_t           i, n;
1151     ngx\_msec\_t           flush;
1152     ngx\_str\_t            *value, name, s;
1153     ngx\_http\_log\_t       *log;
1154     ngx\_syslog\_peer\_t    *peer;
1155     ngx\_http\_log\_buf\_t   *buffer;
1156     ngx\_http\_log\_fmt\_t   *fmt;
1157     ngx\_http\_log\_main\_conf\_t *lmcf;
1158     ngx\_http\_script\_compile\_t sc;
1159     ngx\_http\_compile\_complex\_value\_t ccv;
1160
1161     value = cf->args->elts;
1162
1163     if (ngx\_strcmp(value[1].data, "off") == 0) {
1164         llcf->off = 1;
1165         if (cf->args->nelts == 2) {

```

```

1166     return NGX_CONF_OK;
1167 }
1168
1169     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1170         "invalid parameter \"%v\"", &value[2]);
1171     return NGX_CONF_ERROR;
1172 }
1173
1174 if (llcf->logs == NULL) {
1175     llcf->logs = ngx_array_create(cf->pool, 2, sizeof(ngx_http_log_t));
1176     if (llcf->logs == NULL) {
1177         return NGX_CONF_ERROR;
1178     }
1179 }
1180
1181 lmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_log_module);
1182
1183 log = ngx_array_push(llcf->logs);
1184 if (log == NULL) {
1185     return NGX_CONF_ERROR;
1186 }
1187
1188 ngx_memzero(log, sizeof(ngx_http_log_t));
1189
1190
1191 if (ngx_strcmp(value[1].data, "syslog:", 7) == 0) {
1192
1193     peer = ngx_palloc(cf->pool, sizeof(ngx_syslog_peer_t));
1194     if (peer == NULL) {
1195         return NGX_CONF_ERROR;
1196     }
1197
1198     if (ngx_syslog_process_conf(cf, peer) != NGX_CONF_OK) {
1199         return NGX_CONF_ERROR;
1200     }
1201
1202     log->syslog_peer = peer;
1203
1204     goto process_formats;
1205 }
1206
1207 n = ngx_http_script_variables_count(&value[1]);
1208
1209 if (n == 0) {
1210     log->file = ngx_conf_open_file(cf->cycle, &value[1]);
1211     if (log->file == NULL) {
1212         return NGX_CONF_ERROR;
1213     }
1214 } else {
1215     if (ngx_conf_full_name(cf->cycle, &value[1], 0) != NGX_OK) {
1216         return NGX_CONF_ERROR;
1217     }
1218
1219     log->script = ngx_palloc(cf->pool, sizeof(ngx_http_log_script_t));
1220     if (log->script == NULL) {
1221         return NGX_CONF_ERROR;
1222     }
1223
1224     ngx_memzero(&sc, sizeof(ngx_http_script_compile_t));
1225
1226     sc.cf = cf;
1227     sc.source = &value[1];
1228     sc.lengths = &log->script->lengths;
1229     sc.values = &log->script->values;
1230     sc.variables = n;
1231     sc.complete_lengths = 1;
1232     sc.complete_values = 1;
1233
1234     if (ngx_http_script_compile(&sc) != NGX_OK) {
1235         return NGX_CONF_ERROR;
1236     }
1237 }
1238 }
1239
1240 process_formats:
1241

```

```

1242 if (cf->args->nelts >= 3) {
1243     name = value[2];
1244
1245     if (ngx_strcmp(name.data, "combined") == 0) {
1246         lmcf->combined_used = 1;
1247     }
1248
1249 } else {
1250     ngx_str_set(&name, "combined");
1251     lmcf->combined_used = 1;
1252 }
1253
1254 fmt = lmcf->formats.elts;
1255 for (i = 0; i < lmcf->formats.nelts; i++) {
1256     if (fmt[i].name.len == name.len
1257         && ngx_strcasecmp(fmt[i].name.data, name.data) == 0)
1258     {
1259         log->format = &fmt[i];
1260         break;
1261     }
1262 }
1263
1264 if (log->format == NULL) {
1265     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1266         "unknown log format \"%V\"", &name);
1267     return NGX_CONF_ERROR;
1268 }
1269
1270 size = 0;
1271 flush = 0;
1272 gzip = 0;
1273
1274 for (i = 3; i < cf->args->nelts; i++) {
1275
1276     if (ngx_strncmp(value[i].data, "buffer=", 7) == 0) {
1277         s.len = value[i].len - 7;
1278         s.data = value[i].data + 7;
1279
1280         size = ngx_parse_size(&s);
1281
1282         if (size == NGX_ERROR || size == 0) {
1283             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1284                 "invalid buffer size \"%V\"", &s);
1285             return NGX_CONF_ERROR;
1286         }
1287
1288         continue;
1289     }
1290
1291     if (ngx_strncmp(value[i].data, "flush=", 6) == 0) {
1292         s.len = value[i].len - 6;
1293         s.data = value[i].data + 6;
1294
1295         flush = ngx_parse_time(&s, 0);
1296
1297         if (flush == (ngx_msec_t) NGX_ERROR || flush == 0) {
1298             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1299                 "invalid flush time \"%V\"", &s);
1300             return NGX_CONF_ERROR;
1301         }
1302
1303         continue;
1304     }
1305
1306     if (ngx_strncmp(value[i].data, "gzip", 4) == 0
1307         && (value[i].len == 4 || value[i].data[4] == '='))
1308     {
1309 #if (NGX_ZLIB)
1310         if (size == 0) {
1311             size = 64 * 1024;
1312         }
1313
1314         if (value[i].len == 4) {
1315             gzip = Z_BEST_SPEED;
1316             continue;
1317         }

```

```

1318     s.len = value[i].len - 5;
1319     s.data = value[i].data + 5;
1320
1321     gzip = ngx_atoi(s.data, s.len);
1322
1323     if (gzip < 1 || gzip > 9) {
1324         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1325             "invalid compression level \"%V\"", &s);
1326         return NGX_CONF_ERROR;
1327     }
1328
1329     continue;
1330
1331 #else
1332     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1333         "nginx was built without zlib support");
1334     return NGX_CONF_ERROR;
1335 #endif
1336 }
1337
1338 if (ngx_strcmp(value[i].data, "if=", 3) == 0) {
1339     s.len = value[i].len - 3;
1340     s.data = value[i].data + 3;
1341
1342     ngx_memzero(&ccv, sizeof(ngx_http_compile_complex_value_t));
1343
1344     ccv.cf = cf;
1345     ccv.value = &s;
1346     ccv.complex_value = ngx_palloc(cf->pool,
1347         sizeof(ngx_http_compile_complex_value_t));
1348     if (ccv.complex_value == NULL) {
1349         return NGX_CONF_ERROR;
1350     }
1351
1352     if (ngx_http_compile_complex_value(&ccv) != NGX_OK) {
1353         return NGX_CONF_ERROR;
1354     }
1355
1356     log->filter = ccv.complex_value;
1357
1358     continue;
1359 }
1360
1361 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1362     "invalid parameter \"%V\"", &value[i]);
1363 return NGX_CONF_ERROR;
1364 }
1365
1366 if (flush && size == 0) {
1367     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1368         "no buffer is defined for access_log \"%V\"",
1369         &value[1]);
1370     return NGX_CONF_ERROR;
1371 }
1372
1373 if (size) {
1374     if (log->script) {
1375         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1376             "buffered logs cannot have variables in name");
1377         return NGX_CONF_ERROR;
1378     }
1379
1380     if (log->syslog_peer) {
1381         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1382             "logs to syslog cannot be buffered");
1383         return NGX_CONF_ERROR;
1384     }
1385
1386     if (log->file->data) {
1387         buffer = log->file->data;
1388
1389         if (buffer->last - buffer->start != size
1390             || buffer->flush != flush
1391             || buffer->gzip != gzip)

```

```

1394     {
1395         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1396             "access_log \"%V\" already defined "
1397             "with conflicting parameters",
1398             &value[1]);
1399         return NGX_CONF_ERROR;
1400     }
1401
1402     return NGX_CONF_OK;
1403 }
1404
1405 buffer = ngx_pcalloc(cf->pool, sizeof(ngx_http_log_buf_t));
1406 if (buffer == NULL) {
1407     return NGX_CONF_ERROR;
1408 }
1409
1410 buffer->start = ngx_pnalloc(cf->pool, size);
1411 if (buffer->start == NULL) {
1412     return NGX_CONF_ERROR;
1413 }
1414
1415 buffer->pos = buffer->start;
1416 buffer->last = buffer->start + size;
1417
1418 if (flush) {
1419     buffer->event = ngx_pcalloc(cf->pool, sizeof(ngx_event_t));
1420     if (buffer->event == NULL) {
1421         return NGX_CONF_ERROR;
1422     }
1423
1424     buffer->event->data = log->file;
1425     buffer->event->handler = ngx_http_log_flush_handler;
1426     buffer->event->log = &cf->cycle->new_log;
1427     buffer->event->cancelable = 1;
1428
1429     buffer->flush = flush;
1430 }
1431
1432 buffer->gzip = gzip;
1433
1434 log->file->flush = ngx_http_log_flush;
1435 log->file->data = buffer;
1436 }
1437
1438 return NGX_CONF_OK;
1439 }
1440
1441
1442 static char *
1443 ngx_http_log_set_format(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
1444 {
1445     ngx_http_log_main_conf_t *lmcf = conf;
1446
1447     ngx_str_t          *value;
1448     ngx_uint_t        i;
1449     ngx_http_log_fmt_t *fmt;
1450
1451     value = cf->args->elts;
1452
1453     fmt = lmcf->formats.elts;
1454     for (i = 0; i < lmcf->formats.nelts; i++) {
1455         if (fmt[i].name.len == value[1].len
1456             && ngx_strcmp(fmt[i].name.data, value[1].data) == 0)
1457         {
1458             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1459                 "duplicate \"log_format\" name \"%V\"",
1460                 &value[1]);
1461             return NGX_CONF_ERROR;
1462         }
1463     }
1464
1465     fmt = ngx_array_push(&lmcf->formats);
1466     if (fmt == NULL) {
1467         return NGX_CONF_ERROR;
1468     }
1469 }

```

```

1470     fmt->name = value[1];
1471
1472     fmt->flushes = ngx_array_create(cf->pool, 4, sizeof(ngx_int_t));
1473     if (fmt->flushes == NULL) {
1474         return NGX_CONF_ERROR;
1475     }
1476
1477     fmt->ops = ngx_array_create(cf->pool, 16, sizeof(ngx_http_log_op_t));
1478     if (fmt->ops == NULL) {
1479         return NGX_CONF_ERROR;
1480     }
1481
1482     return ngx_http_log_compile_format(cf, fmt->flushes, fmt->ops, cf->args, 2);
1483 }
1484
1485
1486 static char *
1487 ngx_http_log_compile_format(ngx_conf_t *cf, ngx_array_t *flushes,
1488     ngx_array_t *ops, ngx_array_t *args, ngx_uint_t s)
1489 {
1490     u_char          *data, *p, ch;
1491     size_t          i, len;
1492     ngx_str_t       *value, var;
1493     ngx_int_t       *flush;
1494     ngx_uint_t      bracket;
1495     ngx_http_log_op_t *op;
1496     ngx_http_log_var_t *v;
1497
1498     value = args->elts;
1499
1500     for ( /* void */ ; s < args->nelts; s++) {
1501         i = 0;
1502
1503         while (i < value[s].len) {
1504
1505             op = ngx_array_push(ops);
1506             if (op == NULL) {
1507                 return NGX_CONF_ERROR;
1508             }
1509
1510             data = &value[s].data[i];
1511
1512             if (value[s].data[i] == '$') {
1513                 if (++i == value[s].len) {
1514                     goto invalid;
1515                 }
1516             }
1517
1518             if (value[s].data[i] == '{') {
1519                 bracket = 1;
1520
1521                 if (++i == value[s].len) {
1522                     goto invalid;
1523                 }
1524
1525                 var.data = &value[s].data[i];
1526
1527             } else {
1528                 bracket = 0;
1529                 var.data = &value[s].data[i];
1530             }
1531
1532             for (var.len = 0; i < value[s].len; i++, var.len++) {
1533                 ch = value[s].data[i];
1534
1535                 if (ch == '}' && bracket) {
1536                     i++;
1537                     bracket = 0;
1538                     break;
1539                 }
1540
1541                 if ((ch >= 'A' && ch <= 'Z')
1542                     || (ch >= 'a' && ch <= 'z')
1543                     || (ch >= '0' && ch <= '9')
1544                     || ch == '_')

```



```

1546         {
1547             continue;
1548         }
1549
1550         break;
1551     }
1552
1553     if (bracket) {
1554         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1555             "the closing bracket in \"%V\" "
1556             "variable is missing", &var);
1557         return NGX_CONF_ERROR;
1558     }
1559
1560     if (var.len == 0) {
1561         goto invalid;
1562     }
1563
1564     for (v = ngx_http_log_vars; v->name.len; v++) {
1565
1566         if (v->name.len == var.len
1567             && ngx_strncmp(v->name.data, var.data, var.len) == 0)
1568         {
1569             op->len = v->len;
1570             op->getlen = NULL;
1571             op->run = v->run;
1572             op->data = 0;
1573
1574             goto found;
1575         }
1576     }
1577
1578     if (ngx_http_log_variable_compile(cf, op, &var) != NGX_OK) {
1579         return NGX_CONF_ERROR;
1580     }
1581
1582     if (flushes) {
1583
1584         flush = ngx_array_push(flushes);
1585         if (flush == NULL) {
1586             return NGX_CONF_ERROR;
1587         }
1588
1589         *flush = op->data; /* variable index */
1590     }
1591
1592     found:
1593
1594     continue;
1595 }
1596
1597 i++;
1598
1599 while (i < value[s].len && value[s].data[i] != '$') {
1600     i++;
1601 }
1602
1603 len = &value[s].data[i] - data;
1604
1605 if (len) {
1606
1607     op->len = len;
1608     op->getlen = NULL;
1609
1610     if (len <= sizeof(uintptr_t)) {
1611         op->run = ngx_http_log_copy_short;
1612         op->data = 0;
1613
1614         while (len--) {
1615             op->data <= 8;
1616             op->data |= data[len];
1617         }
1618
1619     } else {
1620         op->run = ngx_http_log_copy_long;
1621

```

```

1622         p = ngx_pnalloc(cf->pool, len);
1623         if (p == NULL) {
1624             return NGX_CONF_ERROR;
1625         }
1626
1627         ngx_memcpy(p, data, len);
1628         op->data = (uintptr_t) p;
1629     }
1630 }
1631 }
1632 }
1633
1634 return NGX_CONF_OK;
1635
1636 invalid:
1637
1638     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0, "invalid parameter \"%s\"", data);
1639
1640     return NGX_CONF_ERROR;
1641 }
1642
1643
1644 static char *
1645 ngx_http_log_open_file_cache(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
1646 {
1647     ngx_http_log_loc_conf_t *llcf = conf;
1648
1649     time_t      inactive, valid;
1650     ngx_str_t   *value, s;
1651     ngx_int_t   max, min_uses;
1652     ngx_uint_t  i;
1653
1654     if (llcf->open_file_cache != NGX_CONF_UNSET_PTR) {
1655         return "is duplicate";
1656     }
1657
1658     value = cf->args->elts;
1659
1660     max = 0;
1661     inactive = 10;
1662     valid = 60;
1663     min_uses = 1;
1664
1665     for (i = 1; i < cf->args->nelts; i++) {
1666
1667         if (ngx_strncmp(value[i].data, "max=", 4) == 0) {
1668
1669             max = ngx_atoi(value[i].data + 4, value[i].len - 4);
1670             if (max == NGX_ERROR) {
1671                 goto failed;
1672             }
1673
1674             continue;
1675         }
1676
1677         if (ngx_strncmp(value[i].data, "inactive=", 9) == 0) {
1678
1679             s.len = value[i].len - 9;
1680             s.data = value[i].data + 9;
1681
1682             inactive = ngx_parse_time(&s, 1);
1683             if (inactive == (time_t) NGX_ERROR) {
1684                 goto failed;
1685             }
1686
1687             continue;
1688         }
1689
1690         if (ngx_strncmp(value[i].data, "min_uses=", 9) == 0) {
1691
1692             min_uses = ngx_atoi(value[i].data + 9, value[i].len - 9);
1693             if (min_uses == NGX_ERROR) {
1694                 goto failed;
1695             }
1696
1697             continue;

```

```

1698     }
1699
1700     if (ngx_strncmp(value[i].data, "valid=", 6) == 0) {
1701         s.len = value[i].len - 6;
1702         s.data = value[i].data + 6;
1703
1704         valid = ngx_parse_time(&s, 1);
1705         if (valid == (time_t) NGX_ERROR) {
1706             goto failed;
1707         }
1708     }
1709
1710     continue;
1711 }
1712
1713 if (ngx_strcmp(value[i].data, "off") == 0) {
1714
1715     llcf->open_file_cache = NULL;
1716
1717     continue;
1718 }
1719
1720 failed:
1721
1722     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1723         "invalid \"open_log_file_cache\" parameter \"%V\"",
1724         &value[i]);
1725     return NGX_CONF_ERROR;
1726 }
1727
1728 if (llcf->open_file_cache == NULL) {
1729     return NGX_CONF_OK;
1730 }
1731
1732 if (max == 0) {
1733     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1734         "\"open_log_file_cache\" must have \"max\" parameter");
1735     return NGX_CONF_ERROR;
1736 }
1737
1738 llcf->open_file_cache = ngx_open_file_cache_init(cf->pool, max, inactive);
1739
1740 if (llcf->open_file_cache) {
1741
1742     llcf->open_file_cache_valid = valid;
1743     llcf->open_file_cache_min_uses = min_uses;
1744
1745     return NGX_CONF_OK;
1746 }
1747
1748 return NGX_CONF_ERROR;
1749 }
1750
1751 static ngx_int_t
1752 ngx_http_log_init(ngx_conf_t *cf)
1753 {
1754     {
1755         ngx_str_t          *value;
1756         ngx_array_t      a;
1757         ngx_http_handler_pt *h;
1758         ngx_http_log_fmt_t *fmt;
1759         ngx_http_log_main_conf_t *lmcf;
1760         ngx_http_core_main_conf_t *cmcf;
1761
1762         lmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_log_module);
1763
1764         if (lmcf->combined_used) {
1765             if (ngx_array_init(&a, cf->pool, 1, sizeof(ngx_str_t)) != NGX_OK) {
1766                 return NGX_ERROR;
1767             }
1768
1769             value = ngx_array_push(&a);
1770             if (value == NULL) {
1771                 return NGX_ERROR;
1772             }
1773

```

```
1774     *value = ngx\_http\_combined\_fmt;  
1775     fmt = lmcf->formats.elts;  
1776  
1777     if (ngx\_http\_log\_compile\_format(cf, NULL, fmt->ops, &a, 0)  
1778         != NGX\_CONF\_OK)  
1779     {  
1780         return NGX\_ERROR;  
1781     }  
1782 }  
1783  
1784 cmcf = ngx\_http\_conf\_get\_module\_main\_conf(cf, ngx\_http\_core\_module);  
1785  
1786 h = ngx\_array\_push(&cmcf->phases[NGX_HTTP_LOG_PHASE].handlers);  
1787 if (h == NULL) {  
1788     return NGX\_ERROR;  
1789 }  
1790  
1791 *h = ngx\_http\_log\_handler;  
1792  
1793 return NGX\_OK;  
1794 }
```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_map\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_map\\_commands](#)
- [ngx\\_http\\_map\\_module](#)
- [ngx\\_http\\_map\\_module\\_ctx](#)

## Data types defined

- [ngx\\_http\\_map\\_conf\\_ctx\\_t](#)
- [ngx\\_http\\_map\\_conf\\_t](#)
- [ngx\\_http\\_map\\_ctx\\_t](#)

## Functions defined

- [ngx\\_http\\_map](#)
- [ngx\\_http\\_map\\_block](#)
- [ngx\\_http\\_map\\_cmp\\_dns\\_wildcards](#)
- [ngx\\_http\\_map\\_create\\_conf](#)
- [ngx\\_http\\_map\\_variable](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 typedef struct {
14     ngx_uint_t          hash_max_size;
15     ngx_uint_t          hash_bucket_size;
16 } ngx_http_map_conf_t;
17
18
19 typedef struct {
20     ngx_hash_keys_arrays_t  keys;
21
22     ngx_array_t             *values_hash;
23     ngx_array_t             var_values;
24 #if (NGX_PCRE)
25     ngx_array_t             regexes;
26 #endif
27
28     ngx_http_variable_value_t *default_value;
29     ngx_conf_t              *cf;
30     ngx_uint_t              hostnames;    /* unsigned hostnames:1 */
31 } ngx_http_map_conf_ctx_t;
32
33
```

```

34 typedef struct {
35     ngx_http_map_t      map;
36     ngx_http_complex_value_t value;
37     ngx_http_variable_value_t *default_value;
38     ngx_uint_t          hostnames;      /* unsigned hostnames:1 */
39 } ngx_http_map_ctx_t;
40
41
42 static int ngx_libc cdecl ngx_http_map_cmp_dns_wildcards(const void *one,
43     const void *two);
44 static void *ngx_http_map_create_conf(ngx_conf_t *cf);
45 static char *ngx_http_map_block(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
46 static char *ngx_http_map(ngx_conf_t *cf, ngx_command_t *dummy, void *conf);
47
48
49 static ngx_command_t  ngx_http_map_commands[] = {
50
51     { ngx_string("map"),
52       NGX_HTTP_MAIN_CONF|NGX_CONF_BLOCK|NGX_CONF_TAKE2,
53       ngx_http_map_block,
54       NGX_HTTP_MAIN_CONF_OFFSET,
55       0,
56       NULL },
57
58     { ngx_string("map_hash_max_size"),
59       NGX_HTTP_MAIN_CONF|NGX_CONF_TAKE1,
60       ngx_conf_set_num_slot,
61       NGX_HTTP_MAIN_CONF_OFFSET,
62       offsetof(ngx_http_map_conf_t, hash_max_size),
63       NULL },
64
65     { ngx_string("map_hash_bucket_size"),
66       NGX_HTTP_MAIN_CONF|NGX_CONF_TAKE1,
67       ngx_conf_set_num_slot,
68       NGX_HTTP_MAIN_CONF_OFFSET,
69       offsetof(ngx_http_map_conf_t, hash_bucket_size),
70       NULL },
71
72     ngx_null_command
73 };
74
75
76 static ngx_http_module_t  ngx_http_map_module_ctx = {
77     NULL,                          /* preconfiguration */
78     NULL,                          /* postconfiguration */
79
80     ngx_http_map_create_conf,       /* create main configuration */
81     NULL,                          /* init main configuration */
82
83     NULL,                          /* create server configuration */
84     NULL,                          /* merge server configuration */
85
86     NULL,                          /* create location configuration */
87     NULL,                          /* merge location configuration */
88 };
89
90
91 ngx_module_t  ngx_http_map_module = {
92     NGX_MODULE_V1,
93     &ngx_http_map_module_ctx,      /* module context */
94     ngx_http_map_commands,         /* module directives */
95     NGX_HTTP_MODULE,              /* module type */
96     NULL,                         /* init master */
97     NULL,                         /* init module */
98     NULL,                         /* init process */
99     NULL,                         /* init thread */
100    NULL,                         /* exit thread */
101    NULL,                         /* exit process */
102    NULL,                         /* exit master */
103    NGX_MODULE_V1_PADDING
104 };
105
106
107 static ngx_int_t
108 ngx_http_map_variable(ngx_http_request_t *r, ngx_http_variable_value_t *v,
109     uintptr_t data)

```

```

110 {
111     ngx\_http\_map\_ctx\_t *map = (ngx\_http\_map\_ctx\_t *)data;
112
113     ngx\_str\_t val;
114     ngx\_http\_variable\_value\_t *value;
115
116     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
117         "http map started");
118
119     if (ngx\_http\_complex\_value(r, &map->value, &val) != NGX\_OK) {
120         return NGX\_ERROR;
121     }
122
123     if (map->hostnames && val.len > 0 && val.data[val.len - 1] == '.') {
124         val.len--;
125     }
126
127     value = ngx\_http\_map\_find(r, &map->map, &val);
128
129     if (value == NULL) {
130         value = map->default_value;
131     }
132
133     if (!value->valid) {
134         value = ngx\_http\_get\_flushed\_variable(r, (uintptr\_t) value->data);
135
136         if (value == NULL || value->not_found) {
137             value = &ngx\_http\_variable\_null\_value;
138         }
139     }
140
141     *v = *value;
142
143     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
144         "http map: \"%v\" \"%v\"", &val, v);
145
146     return NGX\_OK;
147 }
148
149
150 static void *
151 ngx\_http\_map\_create\_conf(ngx\_conf\_t *cf)
152 {
153     ngx\_http\_map\_conf\_t *mcf;
154
155     mcf = ngx\_palloc(cf->pool, sizeof(ngx\_http\_map\_conf\_t));
156     if (mcf == NULL) {
157         return NULL;
158     }
159
160     mcf->hash_max_size = NGX\_CONF\_UNSET\_UINT;
161     mcf->hash_bucket_size = NGX\_CONF\_UNSET\_UINT;
162
163     return mcf;
164 }
165
166
167 static char *
168 ngx\_http\_map\_block(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
169 {
170     ngx\_http\_map\_conf\_t *mcf = conf;
171
172     char *rv;
173     ngx\_str\_t *value, name;
174     ngx\_conf\_t save;
175     ngx\_pool\_t *pool;
176     ngx\_hash\_init\_t hash;
177     ngx\_http\_map\_ctx\_t *map;
178     ngx\_http\_variable\_t *var;
179     ngx\_http\_map\_conf\_ctx\_t ctx;
180     ngx\_http\_compile\_complex\_value\_t ccv;
181
182     if (mcf->hash_max_size == NGX\_CONF\_UNSET\_UINT) {
183         mcf->hash_max_size = 2048;
184     }
185

```

```

186     if (mcf->hash_bucket_size == NGX\_CONF\_UNSET\_UINT){
187         mcf->hash_bucket_size = ngx\_cacheline\_size;
188
189     } else {
190         mcf->hash_bucket_size = ngx\_align(mcf->hash_bucket_size,
191                                         ngx\_cacheline\_size);
192     }
193
194     map = ngx\_palloc(cf->pool, sizeof(ngx\_http\_map\_ctx\_t));
195     if (map == NULL) {
196         return NGX\_CONF\_ERROR;
197     }
198
199     value = cf->args->elts;
200
201     ngx\_memzero(&ccv, sizeof(ngx\_http\_compile\_complex\_value\_t));
202
203     ccv.cf = cf;
204     ccv.value = &value[1];
205     ccv.complex_value = &map->value;
206
207     if (ngx\_http\_compile\_complex\_value(&ccv) != NGX\_OK) {
208         return NGX\_CONF\_ERROR;
209     }
210
211     name = value[2];
212
213     if (name.data[0] != '$') {
214         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
215                            "invalid variable name \"%V\"", &name);
216         return NGX\_CONF\_ERROR;
217     }
218
219     name.len--;
220     name.data++;
221
222     var = ngx\_http\_add\_variable(cf, &name, NGX\_HTTP\_VAR\_CHANGEABLE);
223     if (var == NULL) {
224         return NGX\_CONF\_ERROR;
225     }
226
227     var->get_handler = ngx\_http\_map\_variable;
228     var->data = (uintptr_t) map;
229
230     pool = ngx\_create\_pool(NGX\_DEFAULT\_POOL\_SIZE, cf->log);
231     if (pool == NULL) {
232         return NGX\_CONF\_ERROR;
233     }
234
235     ctx.keys.pool = cf->pool;
236     ctx.keys.temp_pool = pool;
237
238     if (ngx\_hash\_keys\_array\_init(&ctx.keys, NGX\_HASH\_LARGE) != NGX\_OK) {
239         ngx\_destroy\_pool(pool);
240         return NGX\_CONF\_ERROR;
241     }
242
243     ctx.values_hash = ngx\_palloc(pool, sizeof(ngx\_array\_t) * ctx.keys.hsize);
244     if (ctx.values_hash == NULL) {
245         ngx\_destroy\_pool(pool);
246         return NGX\_CONF\_ERROR;
247     }
248
249     if (ngx\_array\_init(&ctx.var_values, cf->pool, 2,
250                      sizeof(ngx\_http\_variable\_value\_t))
251         != NGX\_OK)
252     {
253         ngx\_destroy\_pool(pool);
254         return NGX\_CONF\_ERROR;
255     }
256
257     #if (NGX\_PCRE)
258     if (ngx\_array\_init(&ctx.regexes, cf->pool, 2, sizeof(ngx\_http\_map\_regex\_t))
259         != NGX\_OK)
260     {
261         ngx\_destroy\_pool(pool);

```



```

262     return NGX\_CONF\_ERROR;
263 }
264 #endif
265
266 ctx.default_value = NULL;
267 ctx.cf = &save;
268 ctx.hostnames = 0;
269
270 save = *cf;
271 cf->pool = pool;
272 cf->ctx = &ctx;
273 cf->handler = ngx\_http\_map;
274 cf->handler_conf = conf;
275
276 rv = ngx\_conf\_parse(cf, NULL);
277
278 *cf = save;
279
280 if (rv != NGX\_CONF\_OK) {
281     ngx\_destroy\_pool(pool);
282     return rv;
283 }
284
285 map->default_value = ctx.default_value ? ctx.default_value:
286     &ngx\_http\_variable\_null\_value;
287
288 map->hostnames = ctx.hostnames;
289
290 hash.key = ngx\_hash\_key\_lc;
291 hash.max_size = mcf->hash_max_size;
292 hash.bucket_size = mcf->hash_bucket_size;
293 hash.name = "map_hash";
294 hash.pool = cf->pool;
295
296 if (ctx.keys.keys.nelts) {
297     hash.hash = &map->map.hash.hash;
298     hash.temp_pool = NULL;
299
300     if (ngx\_hash\_init(&hash, ctx.keys.keys.elts, ctx.keys.keys.nelts)
301         != NGX\_OK)
302     {
303         ngx\_destroy\_pool(pool);
304         return NGX\_CONF\_ERROR;
305     }
306 }
307
308 if (ctx.keys.dns_wc_head.nelts) {
309
310     ngx\_qsort(ctx.keys.dns_wc_head.elts,
311             (size_t) ctx.keys.dns_wc_head.nelts,
312             sizeof(ngx\_hash\_key\_t), ngx\_http\_map\_cmp\_dns\_wildcards);
313
314     hash.hash = NULL;
315     hash.temp_pool = pool;
316
317     if (ngx\_hash\_wildcard\_init(&hash, ctx.keys.dns_wc_head.elts,
318                             ctx.keys.dns_wc_head.nelts)
319         != NGX\_OK)
320     {
321         ngx\_destroy\_pool(pool);
322         return NGX\_CONF\_ERROR;
323     }
324
325     map->map.hash.wc_head = (ngx\_hash\_wildcard\_t *) hash.hash;
326 }
327
328 if (ctx.keys.dns_wc_tail.nelts) {
329
330     ngx\_qsort(ctx.keys.dns_wc_tail.elts,
331             (size_t) ctx.keys.dns_wc_tail.nelts,
332             sizeof(ngx\_hash\_key\_t), ngx\_http\_map\_cmp\_dns\_wildcards);
333
334     hash.hash = NULL;
335     hash.temp_pool = pool;
336
337     if (ngx\_hash\_wildcard\_init(&hash, ctx.keys.dns_wc_tail.elts,

```

```

338                                     ctx.keys.dns_wc_tail.nelts)
339         != NGX\_OK)
340     {
341         ngx\_destroy\_pool(pool);
342         return NGX\_CONF\_ERROR;
343     }
344
345     map->map.hash.wc_tail = (ngx\_hash\_wildcard\_t *) hash.hash;
346 }
347
348 #if (NGX\_PCRE)
349
350     if (ctx.regexes.nelts) {
351         map->map.regex = ctx.regexes.elts;
352         map->map.nregex = ctx.regexes.nelts;
353     }
354
355 #endif
356
357     ngx\_destroy\_pool(pool);
358
359     return rv;
360 }
361
362
363 static int ngx\_libc\_cdecl
364 ngx\_http\_map\_cmp\_dns\_wildcards(const void *one, const void *two)
365 {
366     ngx\_hash\_key\_t *first, *second;
367
368     first = (ngx\_hash\_key\_t *) one;
369     second = (ngx\_hash\_key\_t *) two;
370
371     return ngx\_dns\_strcmp(first->key.data, second->key.data);
372 }
373
374
375 static char *
376 ngx\_http\_map(ngx\_conf\_t *cf, ngx\_command\_t *dummy, void *conf)
377 {
378     ngx\_int\_t                rc, index;
379     ngx\_str\_t                *value, name;
380     ngx\_uint\_t                i, key;
381     ngx\_http\_map\_conf\_ctx\_t *ctx;
382     ngx\_http\_variable\_value\_t *var, **vp;
383
384     ctx = cf->ctx;
385
386     value = cf->args->elts;
387
388     if (cf->args->nelts == 1
389         && ngx\_strcmp(value[0].data, "hostnames") == 0)
390     {
391         ctx->hostnames = 1;
392         return NGX\_CONF\_OK;
393     }
394     } else if (cf->args->nelts != 2) {
395         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
396             "invalid number of the map parameters");
397         return NGX\_CONF\_ERROR;
398     }
399
400     if (ngx\_strcmp(value[0].data, "include") == 0) {
401         return ngx\_conf\_include(cf, dummy, conf);
402     }
403
404     if (value[1].data[0] == '$') {
405         name = value[1];
406         name.len--;
407         name.data++;
408
409         index = ngx\_http\_get\_variable\_index(ctx->cf, &name);
410         if (index == NGX\_ERROR) {
411             return NGX\_CONF\_ERROR;
412         }
413     }

```

```

414     var = ctx->var_values.elts;
415
416     for (i = 0; i < ctx->var_values.nelts; i++) {
417         if (index == (intptr_t) var[i].data) {
418             var = &var[i];
419             goto found;
420         }
421     }
422
423     var = ngx_array_push(&ctx->var_values);
424     if (var == NULL) {
425         return NGX_CONF_ERROR;
426     }
427
428     var->valid = 0;
429     var->no_cacheable = 0;
430     var->not_found = 0;
431     var->len = 0;
432     var->data = (u_char *) (intptr_t) index;
433
434     goto found;
435 }
436
437 key = 0;
438
439 for (i = 0; i < value[1].len; i++) {
440     key = ngx_hash(key, value[1].data[i]);
441 }
442
443 key %= ctx->keys.hsize;
444
445 vp = ctx->values_hash[key].elts;
446
447 if (vp) {
448     for (i = 0; i < ctx->values_hash[key].nelts; i++) {
449         if (value[1].len != (size_t) vp[i]->len) {
450             continue;
451         }
452
453         if (ngx_strncmp(value[1].data, vp[i]->data, value[1].len) == 0) {
454             var = vp[i];
455             goto found;
456         }
457     }
458 } else {
459     if (ngx_array_init(&ctx->values_hash[key], cf->pool, 4,
460         sizeof(ngx_http_variable_value_t *))
461         != NGX_OK)
462     {
463         return NGX_CONF_ERROR;
464     }
465 }
466
467 var = ngx_palloc(ctx->keys.pool, sizeof(ngx_http_variable_value_t));
468 if (var == NULL) {
469     return NGX_CONF_ERROR;
470 }
471
472 var->len = value[1].len;
473 var->data = ngx_pstrdup(ctx->keys.pool, &value[1]);
474 if (var->data == NULL) {
475     return NGX_CONF_ERROR;
476 }
477
478 var->valid = 1;
479 var->no_cacheable = 0;
480 var->not_found = 0;
481
482 vp = ngx_array_push(&ctx->values_hash[key]);
483 if (vp == NULL) {
484     return NGX_CONF_ERROR;
485 }
486
487 *vp = var;
488
489

```

```

490 found:
491
492     if (ngx_strcmp(value[0].data, "default") == 0) {
493
494         if (ctx->default_value) {
495             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
496                 "duplicate default map parameter");
497             return NGX_CONF_ERROR;
498         }
499
500         ctx->default_value = var;
501
502         return NGX_CONF_OK;
503     }
504
505 #if (NGX_PCRE)
506
507     if (value[0].len && value[0].data[0] == '~') {
508         ngx_regex_compile_t    rc;
509         ngx_http_map_regex_t   *regex;
510         u_char                  errstr[NGX_MAX_CONF_ERRSTR];
511
512         regex = ngx_array_push(&ctx->regexes);
513         if (regex == NULL) {
514             return NGX_CONF_ERROR;
515         }
516
517         value[0].len--;
518         value[0].data++;
519
520         ngx_memzero(&rc, sizeof(ngx_regex_compile_t));
521
522         if (value[0].data[0] == '*') {
523             value[0].len--;
524             value[0].data++;
525             rc.options = NGX_REGEX_CASELESS;
526         }
527
528         rc.pattern = value[0];
529         rc.err.len = NGX_MAX_CONF_ERRSTR;
530         rc.err.data = errstr;
531
532         regex->regex = ngx_http_regex_compile(ctx->cf, &rc);
533         if (regex->regex == NULL) {
534             return NGX_CONF_ERROR;
535         }
536
537         regex->value = var;
538
539         return NGX_CONF_OK;
540     }
541
542 #endif
543
544     if (value[0].len && value[0].data[0] == '\\') {
545         value[0].len--;
546         value[0].data++;
547     }
548
549     rc = ngx_hash_add_key(&ctx->keys, &value[0], var,
550         (ctx->hostnames) ? NGX_HASH_WILDCARD_KEY : 0);
551
552     if (rc == NGX_OK) {
553         return NGX_CONF_OK;
554     }
555
556     if (rc == NGX_DECLINED) {
557         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
558             "invalid hostname or wildcard \"%V\"", &value[0]);
559     }
560
561     if (rc == NGX_BUSY) {
562         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
563             "conflicting parameter \"%V\"", &value[0]);
564     }
565

```

```
566 return NGX\_CONF\_ERROR;  
567 }
```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_memcached\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_memcached\\_commands](#)
- [ngx\\_http\\_memcached\\_end](#)
- [ngx\\_http\\_memcached\\_key](#)
- [ngx\\_http\\_memcached\\_module](#)
- [ngx\\_http\\_memcached\\_module\\_ctx](#)
- [ngx\\_http\\_memcached\\_next\\_upstream\\_masks](#)

## Data types defined

- [ngx\\_http\\_memcached\\_ctx\\_t](#)
- [ngx\\_http\\_memcached\\_loc\\_conf\\_t](#)

## Functions defined

- [ngx\\_http\\_memcached\\_abort\\_request](#)
- [ngx\\_http\\_memcached\\_create\\_loc\\_conf](#)
- [ngx\\_http\\_memcached\\_create\\_request](#)
- [ngx\\_http\\_memcached\\_filter](#)
- [ngx\\_http\\_memcached\\_filter\\_init](#)
- [ngx\\_http\\_memcached\\_finalize\\_request](#)
- [ngx\\_http\\_memcached\\_handler](#)
- [ngx\\_http\\_memcached\\_merge\\_loc\\_conf](#)
- [ngx\\_http\\_memcached\\_pass](#)
- [ngx\\_http\\_memcached\\_process\\_header](#)
- [ngx\\_http\\_memcached\\_reinit\\_request](#)

## Macros defined

- [NGX\\_HTTP\\_MEMCACHED\\_END](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
```

```

10 #include <ngx_http.h>
11
12
13 typedef struct {
14     ngx_http_upstream_conf_t    upstream;
15     ngx_int_t                    index;
16     ngx_uint_t                   gzip_flag;
17 } ngx_http_memcached_loc_conf_t;
18
19
20 typedef struct {
21     size_t                        rest;
22     ngx_http_request_t            *request;
23     ngx_str_t                      key;
24 } ngx_http_memcached_ctx_t;
25
26
27 static ngx_int_t ngx_http_memcached_create_request(ngx_http_request_t *r);
28 static ngx_int_t ngx_http_memcached_reinit_request(ngx_http_request_t *r);
29 static ngx_int_t ngx_http_memcached_process_header(ngx_http_request_t *r);
30 static ngx_int_t ngx_http_memcached_filter_init(void *data);
31 static ngx_int_t ngx_http_memcached_filter(void *data, ssize_t bytes);
32 static void ngx_http_memcached_abort_request(ngx_http_request_t *r);
33 static void ngx_http_memcached_finalize_request(ngx_http_request_t *r,
34     ngx_int_t rc);
35
36 static void *ngx_http_memcached_create_loc_conf(ngx_conf_t *cf);
37 static char *ngx_http_memcached_merge_loc_conf(ngx_conf_t *cf,
38     void *parent, void *child);
39
40 static char *ngx_http_memcached_pass(ngx_conf_t *cf, ngx_command_t *cmd,
41     void *conf);
42
43
44 static ngx_conf_bitmask_t ngx_http_memcached_next_upstream_masks[] = {
45     { ngx_string("error"), NGX_HTTP_UPSTREAM_FT_ERROR },
46     { ngx_string("timeout"), NGX_HTTP_UPSTREAM_FT_TIMEOUT },
47     { ngx_string("invalid_response"), NGX_HTTP_UPSTREAM_FT_INVALID_HEADER },
48     { ngx_string("not_found"), NGX_HTTP_UPSTREAM_FT_HTTP_404 },
49     { ngx_string("off"), NGX_HTTP_UPSTREAM_FT_OFF },
50     { ngx_null_string, 0 }
51 };
52
53
54 static ngx_command_t ngx_http_memcached_commands[] = {
55
56     { ngx_string("memcached_pass"),
57         NGX_HTTP_LOC_CONF|NGX_HTTP_LIF_CONF|NGX_CONF_TAKE1,
58         ngx_http_memcached_pass,
59         NGX_HTTP_LOC_CONF_OFFSET,
60         0,
61         NULL },
62
63     { ngx_string("memcached_bind"),
64         NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
65         ngx_http_upstream_bind_set_slot,
66         NGX_HTTP_LOC_CONF_OFFSET,
67         offsetof(ngx_http_memcached_loc_conf_t, upstream.local),
68         NULL },
69
70     { ngx_string("memcached_connect_timeout"),
71         NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
72         ngx_conf_set_msec_slot,
73         NGX_HTTP_LOC_CONF_OFFSET,
74         offsetof(ngx_http_memcached_loc_conf_t, upstream.connect_timeout),
75         NULL },
76
77     { ngx_string("memcached_send_timeout"),
78         NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
79         ngx_conf_set_msec_slot,
80         NGX_HTTP_LOC_CONF_OFFSET,
81         offsetof(ngx_http_memcached_loc_conf_t, upstream.send_timeout),
82         NULL },
83
84     { ngx_string("memcached_buffer_size"),
85         NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,

```

```

86     ngx_conf_set_size_slot,
87     NGX_HTTP_LOC_CONF_OFFSET,
88     offsetof(ngx_http_memcached_loc_conf_t, upstream.buffer_size),
89     NULL },
90
91     { ngx_string("memcached_read_timeout"),
92       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
93       ngx_conf_set_msec_slot,
94       NGX_HTTP_LOC_CONF_OFFSET,
95       offsetof(ngx_http_memcached_loc_conf_t, upstream.read_timeout),
96       NULL },
97
98     { ngx_string("memcached_next_upstream"),
99       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_1MORE,
100      ngx_conf_set_bitmask_slot,
101      NGX_HTTP_LOC_CONF_OFFSET,
102      offsetof(ngx_http_memcached_loc_conf_t, upstream.next_upstream),
103      &ngx_http_memcached_next_upstream_masks },
104
105     { ngx_string("memcached_next_upstream_tries"),
106       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
107       ngx_conf_set_num_slot,
108       NGX_HTTP_LOC_CONF_OFFSET,
109       offsetof(ngx_http_memcached_loc_conf_t, upstream.next_upstream_tries),
110       NULL },
111
112     { ngx_string("memcached_next_upstream_timeout"),
113       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
114       ngx_conf_set_msec_slot,
115       NGX_HTTP_LOC_CONF_OFFSET,
116       offsetof(ngx_http_memcached_loc_conf_t, upstream.next_upstream_timeout),
117       NULL },
118
119     { ngx_string("memcached_gzip_flag"),
120       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
121       ngx_conf_set_num_slot,
122       NGX_HTTP_LOC_CONF_OFFSET,
123       offsetof(ngx_http_memcached_loc_conf_t, gzip_flag),
124       NULL },
125
126     ngx_null_command
127 };
128
129
130 static ngx_http_module_t  ngx_http_memcached_module_ctx = {
131     NULL,                                     /* preconfiguration */
132     NULL,                                     /* postconfiguration */
133
134     NULL,                                     /* create main configuration */
135     NULL,                                     /* init main configuration */
136
137     NULL,                                     /* create server configuration */
138     NULL,                                     /* merge server configuration */
139
140     ngx_http_memcached_create_loc_conf,      /* create location configuration */
141     ngx_http_memcached_merge_loc_conf       /* merge location configuration */
142 };
143
144
145 ngx_module_t  ngx_http_memcached_module = {
146     NGX_MODULE_V1,
147     &ngx_http_memcached_module_ctx,         /* module context */
148     ngx_http_memcached_commands,           /* module directives */
149     NGX_HTTP_MODULE,                       /* module type */
150     NULL,                                   /* init master */
151     NULL,                                   /* init module */
152     NULL,                                   /* init process */
153     NULL,                                   /* init thread */
154     NULL,                                   /* exit thread */
155     NULL,                                   /* exit process */
156     NULL,                                   /* exit master */
157     NGX_MODULE_V1_PADDING
158 };
159
160
161 static ngx_str_t  ngx_http_memcached_key = ngx_string("memcached_key");

```



```

162
163
164 #define NGX_HTTP_MEMCACHED_END (sizeof(ngx_http_memcached_end) - 1)
165 static u_char ngx_http_memcached_end[] = CRLF "END" CRLE;
166
167
168 static ngx_int_t
169 ngx_http_memcached_handler(ngx_http_request_t *r)
170 {
171     ngx_int_t rc;
172     ngx_http_upstream_t *u;
173     ngx_http_memcached_ctx_t *ctx;
174     ngx_http_memcached_loc_conf_t *mlcf;
175
176     if (!(r->method & (NGX_HTTP_GET|NGX_HTTP_HEAD))) {
177         return NGX_HTTP_NOT_ALLOWED;
178     }
179
180     rc = ngx_http_discard_request_body(r);
181
182     if (rc != NGX_OK) {
183         return rc;
184     }
185
186     if (ngx_http_set_content_type(r) != NGX_OK) {
187         return NGX_HTTP_INTERNAL_SERVER_ERROR;
188     }
189
190     if (ngx_http_upstream_create(r) != NGX_OK) {
191         return NGX_HTTP_INTERNAL_SERVER_ERROR;
192     }
193
194     u = r->upstream;
195
196     ngx_str_set(&u->schema, "memcached://");
197     u->output.tag = (ngx_buf_tag_t) &ngx_http_memcached_module;
198
199     mlcf = ngx_http_get_module_loc_conf(r, ngx_http_memcached_module);
200
201     u->conf = &mlcf->upstream;
202
203     u->create_request = ngx_http_memcached_create_request;
204     u->reinit_request = ngx_http_memcached_reinit_request;
205     u->process_header = ngx_http_memcached_process_header;
206     u->abort_request = ngx_http_memcached_abort_request;
207     u->finalize_request = ngx_http_memcached_finalize_request;
208
209     ctx = ngx_palloc(r->pool, sizeof(ngx_http_memcached_ctx_t));
210     if (ctx == NULL) {
211         return NGX_HTTP_INTERNAL_SERVER_ERROR;
212     }
213
214     ctx->request = r;
215
216     ngx_http_set_ctx(r, ctx, ngx_http_memcached_module);
217
218     u->input_filter_init = ngx_http_memcached_filter_init;
219     u->input_filter = ngx_http_memcached_filter;
220     u->input_filter_ctx = ctx;
221
222     r->main->count++;
223
224     ngx_http_upstream_init(r);
225
226     return NGX_DONE;
227 }
228
229
230 static ngx_int_t
231 ngx_http_memcached_create_request(ngx_http_request_t *r)
232 {
233     size_t len;
234     uintptr_t escape;
235     ngx_buf_t *b;
236     ngx_chain_t *cl;
237     ngx_http_memcached_ctx_t *ctx;

```

```

238 ngx_http_variable_value_t *vv;
239 ngx_http_memcached_loc_conf_t *mlcf;
240
241 mlcf = ngx_http_get_module_loc_conf(r, ngx_http_memcached_module);
242
243 vv = ngx_http_get_indexed_variable(r, mlcf->index);
244
245 if (vv == NULL || vv->not_found || vv->len == 0) {
246     ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
247                 "the \"$memcached_key\" variable is not set");
248     return NGX_ERROR;
249 }
250
251 escape = 2 * ngx_escape_uri(NULL, vv->data, vv->len, NGX_ESCAPE_MEMCACHED);
252
253 len = sizeof("get ") - 1 + vv->len + escape + sizeof(CRLF) - 1;
254
255 b = ngx_create_temp_buf(r->pool, len);
256 if (b == NULL) {
257     return NGX_ERROR;
258 }
259
260 cl = ngx_alloc_chain_link(r->pool);
261 if (cl == NULL) {
262     return NGX_ERROR;
263 }
264
265 cl->buf = b;
266 cl->next = NULL;
267
268 r->upstream->request_bufs = cl;
269
270 *b->last++ = 'g'; *b->last++ = 'e'; *b->last++ = 't'; *b->last++ = ' ';
271
272 ctx = ngx_http_get_module_ctx(r, ngx_http_memcached_module);
273
274 ctx->key.data = b->last;
275
276 if (escape == 0) {
277     b->last = ngx_copy(b->last, vv->data, vv->len);
278 } else {
279     b->last = (u_char *) ngx_escape_uri(b->last, vv->data, vv->len,
280                                         NGX_ESCAPE_MEMCACHED);
281 }
282
283 ctx->key.len = b->last - ctx->key.data;
284
285 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
286               "http memcached request: \"%V\"", &ctx->key);
287
288 *b->last++ = CR; *b->last++ = LF;
289
290 return NGX_OK;
291 }
292
293
294
295 static ngx_int_t
296 ngx_http_memcached_reinit_request(ngx_http_request_t *r)
297 {
298     return NGX_OK;
299 }
300
301
302 static ngx_int_t
303 ngx_http_memcached_process_header(ngx_http_request_t *r)
304 {
305     u_char *p, *start;
306     ngx_str_t line;
307     ngx_uint_t flags;
308     ngx_table_elt_t *h;
309     ngx_http_upstream_t *u;
310     ngx_http_memcached_ctx_t *ctx;
311     ngx_http_memcached_loc_conf_t *mlcf;
312
313     u = r->upstream;

```

```

314     for (p = u->buffer.pos; p < u->buffer.last; p++) {
315         if (*p == LF) {
316             goto found;
317         }
318     }
319 }
320
321 return NGX_AGAIN;
322
323 found:
324
325     line.data = u->buffer.pos;
326     line.len = p - u->buffer.pos;
327
328     if (line.len == 0 || *(p - 1) != CR) {
329         goto no_valid;
330     }
331
332     *p = '\0';
333     line.len--;
334
335     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
336                  "memcached: \"%V\"", &line);
337
338     p = u->buffer.pos;
339
340     ctx = ngx_http_get_module_ctx(r, ngx_http_memcached_module);
341     mlcf = ngx_http_get_module_loc_conf(r, ngx_http_memcached_module);
342
343     if (ngx_strncmp(p, "VALUE ", sizeof("VALUE ") - 1) == 0) {
344
345         p += sizeof("VALUE ") - 1;
346
347         if (ngx_strncmp(p, ctx->key.data, ctx->key.len) != 0) {
348             ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
349                          "memcached sent invalid key in response \"%V\" "
350                          "for key \"%V\"",
351                          &line, &ctx->key);
352
353             return NGX_HTTP_UPSTREAM_INVALID_HEADER;
354         }
355
356         p += ctx->key.len;
357
358         if (*p++ != ' ') {
359             goto no_valid;
360         }
361
362         /* flags */
363
364         start = p;
365
366         while (*p) {
367             if (*p++ == ' ') {
368                 if (mlcf->gzip_flag) {
369                     goto flags;
370                 } else {
371                     goto length;
372                 }
373             }
374         }
375
376         goto no_valid;
377
378     flags:
379
380         flags = ngx_atoi(start, p - start - 1);
381
382         if (flags == (ngx_uint_t) NGX_ERROR) {
383             ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
384                          "memcached sent invalid flags in response \"%V\" "
385                          "for key \"%V\"",
386                          &line, &ctx->key);
387             return NGX_HTTP_UPSTREAM_INVALID_HEADER;
388         }
389

```

```

390     if (flags & mlcf->gzip_flag) {
391         h = ngx_list_push(&r->headers_out.headers);
392         if (h == NULL) {
393             return NGX_ERROR;
394         }
395
396         h->hash = 1;
397         ngx_str_set(&h->key, "Content-Encoding");
398         ngx_str_set(&h->value, "gzip");
399         r->headers_out.content_encoding = h;
400     }
401
402     length:
403
404     start = p;
405     p = line.data + line.len;
406
407     u->headers_in.content_length_n = ngx_atoof(start, p - start);
408     if (u->headers_in.content_length_n == NGX_ERROR) {
409         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
410             "memcached sent invalid length in response \"%V\" "
411             "for key \"%V\"",
412             &line, &ctx->key);
413         return NGX_HTTP_UPSTREAM_INVALID_HEADER;
414     }
415
416     u->headers_in.status_n = 200;
417     u->state->status = 200;
418     u->buffer.pos = p + sizeof(CRLF) - 1;
419
420     return NGX_OK;
421 }
422
423 if (ngx_strcmp(p, "END\x0d") == 0) {
424     ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
425         "key: \"%V\" was not found by memcached", &ctx->key);
426
427     u->headers_in.content_length_n = 0;
428     u->headers_in.status_n = 404;
429     u->state->status = 404;
430     u->buffer.pos = p + sizeof("END" CRLF) - 1;
431     u->keepalive = 1;
432
433     return NGX_OK;
434 }
435
436 no_valid:
437
438     ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
439         "memcached sent invalid response: \"%V\"", &line);
440
441     return NGX_HTTP_UPSTREAM_INVALID_HEADER;
442 }
443
444
445 static ngx_int_t
446 ngx_http_memcached_filter_init(void *data)
447 {
448     ngx_http_memcached_ctx_t *ctx = data;
449
450     ngx_http_upstream_t *u;
451
452     u = ctx->request->upstream;
453
454     if (u->headers_in.status_n != 404) {
455         u->length = u->headers_in.content_length_n + NGX_HTTP_MEMCACHED_END;
456         ctx->rest = NGX_HTTP_MEMCACHED_END;
457     } else {
458         u->length = 0;
459     }
460 }
461
462     return NGX_OK;
463 }
464
465

```

```

466 static ngx_int_t
467 ngx_http_memcached_filter(void *data, ssize_t bytes)
468 {
469     ngx_http_memcached_ctx_t *ctx = data;
470
471     u_char                *last;
472     ngx_buf_t             *b;
473     ngx_chain_t           *cl, **ll;
474     ngx_http_upstream_t   *u;
475
476     u = ctx->request->upstream;
477     b = &u->buffer;
478
479     if (u->length == (ssize_t) ctx->rest) {
480
481         if (ngx_strncmp(b->last,
482                        ngx_http_memcached_end + NGX_HTTP_MEMCACHED_END - ctx->rest,
483                        bytes)
484             != 0)
485         {
486             ngx_log_error(NGX_LOG_ERR, ctx->request->connection->log, 0,
487                           "memcached sent invalid trailer");
488
489             u->length = 0;
490             ctx->rest = 0;
491
492             return NGX_OK;
493         }
494
495         u->length -= bytes;
496         ctx->rest -= bytes;
497
498         if (u->length == 0) {
499             u->keepalive = 1;
500         }
501
502         return NGX_OK;
503     }
504
505     for (cl = u->out_bufs, ll = &u->out_bufs; cl; cl = cl->next) {
506         ll = &cl->next;
507     }
508
509     cl = ngx_chain_get_free_buf(ctx->request->pool, &u->free_bufs);
510     if (cl == NULL) {
511         return NGX_ERROR;
512     }
513
514     cl->buf->flush = 1;
515     cl->buf->memory = 1;
516
517     *ll = cl;
518
519     last = b->last;
520     cl->buf->pos = last;
521     b->last += bytes;
522     cl->buf->last = b->last;
523     cl->buf->tag = u->output.tag;
524
525     ngx_log_debug4(NGX_LOG_DEBUG_HTTP, ctx->request->connection->log, 0,
526                  "memcached filter bytes:%z size:%z length:%z rest:%z",
527                  bytes, b->last - b->pos, u->length, ctx->rest);
528
529     if (bytes <= (ssize_t) (u->length - NGX_HTTP_MEMCACHED_END)) {
530         u->length -= bytes;
531         return NGX_OK;
532     }
533
534     last += (size_t) (u->length - NGX_HTTP_MEMCACHED_END);
535
536     if (ngx_strncmp(last, ngx_http_memcached_end, b->last - last) != 0) {
537         ngx_log_error(NGX_LOG_ERR, ctx->request->connection->log, 0,
538                       "memcached sent invalid trailer");
539
540         b->last = last;
541         cl->buf->last = last;

```

```

542     u->length = 0;
543     ctx->rest = 0;
544
545     return NGX_OK;
546 }
547
548 ctx->rest -= b->last - last;
549 b->last = last;
550 cl->buf->last = last;
551 u->length = ctx->rest;
552
553 if (u->length == 0) {
554     u->keepalive = 1;
555 }
556
557 return NGX_OK;
558 }
559
560
561 static void
562 ngx_http_memcached_abort_request(ngx_http_request_t *r)
563 {
564     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
565                  "abort http memcached request");
566     return;
567 }
568
569
570 static void
571 ngx_http_memcached_finalize_request(ngx_http_request_t *r, ngx_int_t rc)
572 {
573     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
574                  "finalize http memcached request");
575     return;
576 }
577
578
579 static void *
580 ngx_http_memcached_create_loc_conf(ngx_conf_t *cf)
581 {
582     ngx_http_memcached_loc_conf_t *conf;
583
584     conf = ngx_palloc(cf->pool, sizeof(ngx_http_memcached_loc_conf_t));
585     if (conf == NULL) {
586         return NULL;
587     }
588
589     /*
590     * set by ngx_palloc():
591     *
592     *     conf->upstream.bufs.num = 0;
593     *     conf->upstream.next_upstream = 0;
594     *     conf->upstream.temp_path = NULL;
595     *     conf->upstream.uri = { 0, NULL };
596     *     conf->upstream.location = NULL;
597     */
598
599     conf->upstream.local = NGX_CONF_UNSET_PTR;
600     conf->upstream.next_upstream_tries = NGX_CONF_UNSET_UINT;
601     conf->upstream.connect_timeout = NGX_CONF_UNSET_MSEC;
602     conf->upstream.send_timeout = NGX_CONF_UNSET_MSEC;
603     conf->upstream.read_timeout = NGX_CONF_UNSET_MSEC;
604     conf->upstream.next_upstream_timeout = NGX_CONF_UNSET_MSEC;
605
606     conf->upstream.buffer_size = NGX_CONF_UNSET_SIZE;
607
608     /* the hardcoded values */
609     conf->upstream.cyclic_temp_file = 0;
610     conf->upstream.buffering = 0;
611     conf->upstream.ignore_client_abort = 0;
612     conf->upstream.send_lowat = 0;
613     conf->upstream.bufs.num = 0;
614     conf->upstream.busy_buffers_size = 0;
615     conf->upstream.max_temp_file_size = 0;
616     conf->upstream.temp_file_write_size = 0;
617     conf->upstream.intercept_errors = 1;

```

```

618     conf->upstream.intercept_404 = 1;
619     conf->upstream.pass_request_headers = 0;
620     conf->upstream.pass_request_body = 0;
621
622     conf->index = NGX\_CONF\_UNSET;
623     conf->gzip_flag = NGX\_CONF\_UNSET\_UINT;
624
625     return conf;
626 }
627
628
629 static char *
630 ngx_http_memcached_merge_loc_conf(ngx\_conf\_t *cf, void *parent, void *child)
631 {
632     ngx\_http\_memcached\_loc\_conf\_t *prev = parent;
633     ngx\_http\_memcached\_loc\_conf\_t *conf = child;
634
635     ngx\_conf\_merge\_ptr\_value(conf->upstream.local,
636                             prev->upstream.local, NULL);
637
638     ngx\_conf\_merge\_uint\_value(conf->upstream.next_upstream_tries,
639                             prev->upstream.next_upstream_tries, 0);
640
641     ngx\_conf\_merge\_msec\_value(conf->upstream.connect_timeout,
642                             prev->upstream.connect_timeout, 60000);
643
644     ngx\_conf\_merge\_msec\_value(conf->upstream.send_timeout,
645                             prev->upstream.send_timeout, 60000);
646
647     ngx\_conf\_merge\_msec\_value(conf->upstream.read_timeout,
648                             prev->upstream.read_timeout, 60000);
649
650     ngx\_conf\_merge\_msec\_value(conf->upstream.next_upstream_timeout,
651                             prev->upstream.next_upstream_timeout, 0);
652
653     ngx\_conf\_merge\_size\_value(conf->upstream.buffer_size,
654                             prev->upstream.buffer_size,
655                             (size_t) ngx\_pagesize);
656
657     ngx\_conf\_merge\_bitmask\_value(conf->upstream.next_upstream,
658                             prev->upstream.next_upstream,
659                             (NGX\_CONF\_BITMASK\_SET
660                              |NGX\_HTTP\_UPSTREAM\_FT\_ERROR
661                              |NGX\_HTTP\_UPSTREAM\_FT\_TIMEOUT));
662
663     if (conf->upstream.next_upstream & NGX\_HTTP\_UPSTREAM\_FT\_OFF) {
664         conf->upstream.next_upstream = NGX\_CONF\_BITMASK\_SET
665                                     |NGX\_HTTP\_UPSTREAM\_FT\_OFF;
666     }
667
668     if (conf->upstream.upstream == NULL) {
669         conf->upstream.upstream = prev->upstream.upstream;
670     }
671
672     if (conf->index == NGX\_CONF\_UNSET) {
673         conf->index = prev->index;
674     }
675
676     ngx\_conf\_merge\_uint\_value(conf->gzip_flag, prev->gzip_flag, 0);
677
678     return NGX\_CONF\_OK;
679 }
680
681
682 static char *
683 ngx_http_memcached_pass(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
684 {
685     ngx\_http\_memcached\_loc\_conf\_t *mlcf = conf;
686
687     ngx\_str\_t *value;
688     ngx\_url\_t u;
689     ngx\_http\_core\_loc\_conf\_t *clcf;
690
691     if (mlcf->upstream.upstream) {
692         return "is duplicate";
693     }

```

```
694 value = cf->args->elts;
695
696
697 ngx\_memzero(&u, sizeof\(ngx\_url\_t\));
698
699 u.url = value[1];
700 u.no_resolve = 1;
701
702 mlcf->upstream.upstream = ngx\_http\_upstream\_add(cf, &u, 0);
703 if (mlcf->upstream.upstream == NULL) {
704     return NGX\_CONF\_ERROR;
705 }
706
707 clcf = ngx\_http\_conf\_get\_module\_loc\_conf(cf, ngx\_http\_core\_module);
708
709 clcf->handler = ngx\_http\_memcached\_handler;
710
711 if (clcf->name.data[clcf->name.len - 1] == '/') {
712     clcf->auto_redirect = 1;
713 }
714
715 mlcf->index = ngx\_http\_get\_variable\_index(cf, &ngx\_http\_memcached\_key);
716
717 if (mlcf->index == NGX\_ERROR) {
718     return NGX\_CONF\_ERROR;
719 }
720
721 return NGX\_CONF\_OK;
722 }
```

[One Level Up](#)

[Top Level](#)



## src/http/modules/nginx\_http\_mp4\_module.c - nginx-1.7.10

### Global variables defined

- [ngx\\_http\\_mp4\\_atoms](#)
- [ngx\\_http\\_mp4\\_commands](#)
- [ngx\\_http\\_mp4\\_mdia\\_atoms](#)
- [ngx\\_http\\_mp4\\_minf\\_atoms](#)
- [ngx\\_http\\_mp4\\_module](#)
- [ngx\\_http\\_mp4\\_module\\_ctx](#)
- [ngx\\_http\\_mp4\\_moov\\_atoms](#)
- [ngx\\_http\\_mp4\\_stbl\\_atoms](#)
- [ngx\\_http\\_mp4\\_trak\\_atoms](#)

### Data types defined

- [ngx\\_http\\_mp4\\_atom\\_handler\\_t](#)
- [ngx\\_http\\_mp4\\_conf\\_t](#)
- [ngx\\_http\\_mp4\\_file\\_t](#)
- [ngx\\_http\\_mp4\\_stss\\_atom\\_t](#)
- [ngx\\_http\\_mp4\\_trak\\_t](#)
- [ngx\\_mp4\\_atom\\_header64\\_t](#)
- [ngx\\_mp4\\_atom\\_header\\_t](#)
- [ngx\\_mp4\\_co64\\_atom\\_t](#)
- [ngx\\_mp4\\_ctts\\_atom\\_t](#)
- [ngx\\_mp4\\_ctts\\_entry\\_t](#)
- [ngx\\_mp4\\_mdhd64\\_atom\\_t](#)
- [ngx\\_mp4\\_mdhd\\_atom\\_t](#)
- [ngx\\_mp4\\_mvhd64\\_atom\\_t](#)
- [ngx\\_mp4\\_mvhd\\_atom\\_t](#)
- [ngx\\_mp4\\_stco\\_atom\\_t](#)
- [ngx\\_mp4\\_stsc\\_atom\\_t](#)
- [ngx\\_mp4\\_stsc\\_entry\\_t](#)
- [ngx\\_mp4\\_stsd\\_atom\\_t](#)
- [ngx\\_mp4\\_stsz\\_atom\\_t](#)

- [ngx\\_mp4\\_stts\\_atom\\_t](#)
- [ngx\\_mp4\\_stts\\_entry\\_t](#)
- [ngx\\_mp4\\_tkhd64\\_atom\\_t](#)
- [ngx\\_mp4\\_tkhd\\_atom\\_t](#)

## Functions defined

- [ngx\\_http\\_mp4](#)
- [ngx\\_http\\_mp4\\_adjust\\_co64\\_atom](#)
- [ngx\\_http\\_mp4\\_adjust\\_stco\\_atom](#)
- [ngx\\_http\\_mp4\\_create\\_conf](#)
- [ngx\\_http\\_mp4\\_crop\\_ctts\\_data](#)
- [ngx\\_http\\_mp4\\_crop\\_stsc\\_data](#)
- [ngx\\_http\\_mp4\\_crop\\_stss\\_data](#)
- [ngx\\_http\\_mp4\\_crop\\_stts\\_data](#)
- [ngx\\_http\\_mp4\\_handler](#)
- [ngx\\_http\\_mp4\\_merge\\_conf](#)
- [ngx\\_http\\_mp4\\_process](#)
- [ngx\\_http\\_mp4\\_read](#)
- [ngx\\_http\\_mp4\\_read\\_atom](#)
- [ngx\\_http\\_mp4\\_read\\_cmov\\_atom](#)
- [ngx\\_http\\_mp4\\_read\\_co64\\_atom](#)
- [ngx\\_http\\_mp4\\_read\\_ctts\\_atom](#)
- [ngx\\_http\\_mp4\\_read\\_dinf\\_atom](#)
- [ngx\\_http\\_mp4\\_read\\_ftyp\\_atom](#)
- [ngx\\_http\\_mp4\\_read\\_hdlr\\_atom](#)
- [ngx\\_http\\_mp4\\_read\\_mdat\\_atom](#)
- [ngx\\_http\\_mp4\\_read\\_mdhd\\_atom](#)
- [ngx\\_http\\_mp4\\_read\\_mdia\\_atom](#)
- [ngx\\_http\\_mp4\\_read\\_minf\\_atom](#)
- [ngx\\_http\\_mp4\\_read\\_moov\\_atom](#)
- [ngx\\_http\\_mp4\\_read\\_mvhd\\_atom](#)
- [ngx\\_http\\_mp4\\_read\\_smhd\\_atom](#)
- [ngx\\_http\\_mp4\\_read\\_stbl\\_atom](#)
- [ngx\\_http\\_mp4\\_read\\_stco\\_atom](#)

- [ngx\\_http\\_mp4\\_read\\_stsc\\_atom](#)
- [ngx\\_http\\_mp4\\_read\\_stsd\\_atom](#)
- [ngx\\_http\\_mp4\\_read\\_stss\\_atom](#)
- [ngx\\_http\\_mp4\\_read\\_stsz\\_atom](#)
- [ngx\\_http\\_mp4\\_read\\_stts\\_atom](#)
- [ngx\\_http\\_mp4\\_read\\_tkhd\\_atom](#)
- [ngx\\_http\\_mp4\\_read\\_trak\\_atom](#)
- [ngx\\_http\\_mp4\\_read\\_vmhd\\_atom](#)
- [ngx\\_http\\_mp4\\_update\\_co64\\_atom](#)
- [ngx\\_http\\_mp4\\_update\\_ctts\\_atom](#)
- [ngx\\_http\\_mp4\\_update\\_mdat\\_atom](#)
- [ngx\\_http\\_mp4\\_update\\_mdia\\_atom](#)
- [ngx\\_http\\_mp4\\_update\\_minf\\_atom](#)
- [ngx\\_http\\_mp4\\_update\\_stbl\\_atom](#)
- [ngx\\_http\\_mp4\\_update\\_stco\\_atom](#)
- [ngx\\_http\\_mp4\\_update\\_stsc\\_atom](#)
- [ngx\\_http\\_mp4\\_update\\_stss\\_atom](#)
- [ngx\\_http\\_mp4\\_update\\_stsz\\_atom](#)
- [ngx\\_http\\_mp4\\_update\\_stts\\_atom](#)
- [ngx\\_http\\_mp4\\_update\\_trak\\_atom](#)

## Macros defined

- [NGX\\_HTTP\\_MP4\\_CO64\\_ATOM](#)
- [NGX\\_HTTP\\_MP4\\_CO64\\_DATA](#)
- [NGX\\_HTTP\\_MP4\\_CTTS\\_ATOM](#)
- [NGX\\_HTTP\\_MP4\\_CTTS\\_DATA](#)
- [NGX\\_HTTP\\_MP4\\_DINF\\_ATOM](#)
- [NGX\\_HTTP\\_MP4\\_HDLR\\_ATOM](#)
- [NGX\\_HTTP\\_MP4\\_LAST\\_ATOM](#)
- [NGX\\_HTTP\\_MP4\\_MDHD\\_ATOM](#)
- [NGX\\_HTTP\\_MP4\\_MDIA\\_ATOM](#)
- [NGX\\_HTTP\\_MP4\\_MINF\\_ATOM](#)
- [NGX\\_HTTP\\_MP4\\_MOOV\\_BUFFER\\_EXCESS](#)
- [NGX\\_HTTP\\_MP4\\_SMHD\\_ATOM](#)

- [NGX HTTP MP4 STBL ATOM](#)
- [NGX HTTP MP4 STCO ATOM](#)
- [NGX HTTP MP4 STCO DATA](#)
- [NGX HTTP MP4 STSC ATOM](#)
- [NGX HTTP MP4 STSC DATA](#)
- [NGX HTTP MP4 STSC END](#)
- [NGX HTTP MP4 STSC START](#)
- [NGX HTTP MP4 STSD ATOM](#)
- [NGX HTTP MP4 STSS ATOM](#)
- [NGX HTTP MP4 STSS DATA](#)
- [NGX HTTP MP4 STSZ ATOM](#)
- [NGX HTTP MP4 STSZ DATA](#)
- [NGX HTTP MP4 STTS ATOM](#)
- [NGX HTTP MP4 STTS DATA](#)
- [NGX HTTP MP4 TKHD ATOM](#)
- [NGX HTTP MP4 TRAK ATOM](#)
- [NGX HTTP MP4 VMHD ATOM](#)
- [ngx\\_mp4\\_atom\\_data](#)
- [ngx\\_mp4\\_atom\\_data\\_size](#)
- [ngx\\_mp4\\_atom\\_header](#)
- [ngx\\_mp4\\_atom\\_next](#)
- [ngx\\_mp4\\_get\\_32value](#)
- [ngx\\_mp4\\_get\\_64value](#)
- [ngx\\_mp4\\_last\\_trak](#)
- [ngx\\_mp4\\_set\\_32value](#)
- [ngx\\_mp4\\_set\\_64value](#)
- [ngx\\_mp4\\_set\\_atom\\_name](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7 #include <ngx_config.h>
8 #include <ngx_core.h>
9 #include <ngx_http.h>
10
11
```

```

12 #define NGX_HTTP_MP4_TRAK_ATOM      0
13 #define NGX_HTTP_MP4_TKHD_ATOM      1
14 #define NGX_HTTP_MP4_MDIA_ATOM      2
15 #define NGX_HTTP_MP4_MDHD_ATOM      3
16 #define NGX_HTTP_MP4_HDLR_ATOM      4
17 #define NGX_HTTP_MP4_MINF_ATOM      5
18 #define NGX_HTTP_MP4_VMHD_ATOM      6
19 #define NGX_HTTP_MP4_SMHD_ATOM      7
20 #define NGX_HTTP_MP4_DINF_ATOM      8
21 #define NGX_HTTP_MP4_STBL_ATOM      9
22 #define NGX_HTTP_MP4_STSD_ATOM     10
23 #define NGX_HTTP_MP4_STTS_ATOM     11
24 #define NGX_HTTP_MP4_STTS_DATA     12
25 #define NGX_HTTP_MP4_STSS_ATOM     13
26 #define NGX_HTTP_MP4_STSS_DATA     14
27 #define NGX_HTTP_MP4_CTTS_ATOM     15
28 #define NGX_HTTP_MP4_CTTS_DATA     16
29 #define NGX_HTTP_MP4_STSC_ATOM     17
30 #define NGX_HTTP_MP4_STSC_START     18
31 #define NGX_HTTP_MP4_STSC_DATA     19
32 #define NGX_HTTP_MP4_STSC_END      20
33 #define NGX_HTTP_MP4_STSZ_ATOM     21
34 #define NGX_HTTP_MP4_STSZ_DATA     22
35 #define NGX_HTTP_MP4_STCO_ATOM     23
36 #define NGX_HTTP_MP4_STCO_DATA     24
37 #define NGX_HTTP_MP4_C064_ATOM     25
38 #define NGX_HTTP_MP4_C064_DATA     26
39
40 #define NGX_HTTP_MP4_LAST_ATOM      NGX\_HTTP\_MP4\_C064\_DATA
41
42
43 typedef struct {
44     size_t          buffer_size;
45     size_t          max_buffer_size;
46 } ngx_http_mp4_conf_t;
47
48
49 typedef struct {
50     u_char          chunk[4];
51     u_char          samples[4];
52     u_char          id[4];
53 } ngx_mp4_stsc_entry_t;
54
55
56 typedef struct {
57     uint32_t        timescale;
58     uint32_t        time_to_sample_entries;
59     uint32_t        sample_to_chunk_entries;
60     uint32_t        sync_samples_entries;
61     uint32_t        composition_offset_entries;
62     uint32_t        sample_sizes_entries;
63     uint32_t        chunks;
64
65     ngx\_uint\_t    start_sample;
66     ngx\_uint\_t    end_sample;
67     ngx\_uint\_t    start_chunk;
68     ngx\_uint\_t    end_chunk;
69     ngx\_uint\_t    start_chunk_samples;
70     ngx\_uint\_t    end_chunk_samples;
71     uint64_t        start_chunk_samples_size;
72     uint64_t        end_chunk_samples_size;
73     off_t           start_offset;
74     off_t           end_offset;
75
76     size_t          tkhd_size;
77     size_t          mdhd_size;
78     size_t          hdlr_size;
79     size_t          vmhd_size;
80     size_t          smhd_size;
81     size_t          dinf_size;
82     size_t          size;
83
84     ngx\_chain\_t    out[NGX\_HTTP\_MP4\_LAST\_ATOM + 1];
85
86     ngx\_buf\_t      trak_atom_buf;
87     ngx\_buf\_t      tkhd_atom_buf;

```

```

88     ngx_buf_t      mdia_atom_buf;
89     ngx_buf_t      mdhd_atom_buf;
90     ngx_buf_t      hdlr_atom_buf;
91     ngx_buf_t      minf_atom_buf;
92     ngx_buf_t      vmhd_atom_buf;
93     ngx_buf_t      smhd_atom_buf;
94     ngx_buf_t      dinf_atom_buf;
95     ngx_buf_t      stbl_atom_buf;
96     ngx_buf_t      stsd_atom_buf;
97     ngx_buf_t      stts_atom_buf;
98     ngx_buf_t      stts_data_buf;
99     ngx_buf_t      stss_atom_buf;
100    ngx_buf_t      stss_data_buf;
101    ngx_buf_t      ctts_atom_buf;
102    ngx_buf_t      ctts_data_buf;
103    ngx_buf_t      stsc_atom_buf;
104    ngx_buf_t      stsc_start_chunk_buf;
105    ngx_buf_t      stsc_end_chunk_buf;
106    ngx_buf_t      stsc_data_buf;
107    ngx_buf_t      stsz_atom_buf;
108    ngx_buf_t      stsz_data_buf;
109    ngx_buf_t      stco_atom_buf;
110    ngx_buf_t      stco_data_buf;
111    ngx_buf_t      co64_atom_buf;
112    ngx_buf_t      co64_data_buf;
113
114    ngx_mp4_stsc_entry_t stsc_start_chunk_entry;
115    ngx_mp4_stsc_entry_t stsc_end_chunk_entry;
116 } ngx_http_mp4_trak_t;
117
118
119 typedef struct {
120     ngx_file_t      file;
121
122     u_char          *buffer;
123     u_char          *buffer_start;
124     u_char          *buffer_pos;
125     u_char          *buffer_end;
126     size_t         buffer_size;
127
128     off_t           offset;
129     off_t           end;
130     off_t           content_length;
131     ngx_uint_t      start;
132     ngx_uint_t      length;
133     uint32_t        timescale;
134     ngx_http_request_t *request;
135     ngx_array_t     trak;
136     ngx_http_mp4_trak_t traks[2];
137
138     size_t         ftyp_size;
139     size_t         moov_size;
140
141     ngx_chain_t    *out;
142     ngx_chain_t    ftyp_atom;
143     ngx_chain_t    moov_atom;
144     ngx_chain_t    mvhd_atom;
145     ngx_chain_t    mdat_atom;
146     ngx_chain_t    mdat_data;
147
148     ngx_buf_t      ftyp_atom_buf;
149     ngx_buf_t      moov_atom_buf;
150     ngx_buf_t      mvhd_atom_buf;
151     ngx_buf_t      mdat_atom_buf;
152     ngx_buf_t      mdat_data_buf;
153
154     u_char         moov_atom_header[8];
155     u_char         mdat_atom_header[16];
156 } ngx_http_mp4_file_t;
157
158
159 typedef struct {
160     char          *name;
161     ngx_int_t     (*handler)(ngx_http_mp4_file_t *mp4,
162                             uint64_t atom_data_size);
163 } ngx_http_mp4_atom_handler_t;

```

```

164
165
166 #define ngx_mp4_atom_header(mp4)    (mp4->buffer_pos - 8)
167 #define ngx_mp4_atom_data(mp4)      mp4->buffer_pos
168 #define ngx_mp4_atom_data_size(t)   (uint64_t) (sizeof(t) - 8)
169
170
171 #define ngx_mp4_atom_next(mp4, n)    \
172     mp4->buffer_pos += (size_t) n;   \
173     mp4->offset += n
174
175
176 #define ngx_mp4_set_atom_name(p, n1, n2, n3, n4)    \
177     ((u_char *) (p))[4] = n1;                       \
178     ((u_char *) (p))[5] = n2;                       \
179     ((u_char *) (p))[6] = n3;                       \
180     ((u_char *) (p))[7] = n4
181
182 #define ngx_mp4_get_32value(p)    \
183     ( ((uint32_t) ((u_char *) (p))[0] << 24)   \
184     + ( ((u_char *) (p))[1] << 16)             \
185     + ( ((u_char *) (p))[2] << 8)              \
186     + ( ((u_char *) (p))[3] ) )
187
188 #define ngx_mp4_set_32value(p, n)    \
189     ((u_char *) (p))[0] = (u_char) ((n) >> 24); \
190     ((u_char *) (p))[1] = (u_char) ((n) >> 16); \
191     ((u_char *) (p))[2] = (u_char) ((n) >> 8);  \
192     ((u_char *) (p))[3] = (u_char) (n)
193
194 #define ngx_mp4_get_64value(p)    \
195     ( ((uint64_t) ((u_char *) (p))[0] << 56)   \
196     + ((uint64_t) ((u_char *) (p))[1] << 48)   \
197     + ((uint64_t) ((u_char *) (p))[2] << 40)   \
198     + ((uint64_t) ((u_char *) (p))[3] << 32)   \
199     + ((uint64_t) ((u_char *) (p))[4] << 24)   \
200     + ( ((u_char *) (p))[5] << 16)             \
201     + ( ((u_char *) (p))[6] << 8)              \
202     + ( ((u_char *) (p))[7] ) )
203
204 #define ngx_mp4_set_64value(p, n)    \
205     ((u_char *) (p))[0] = (u_char) ((uint64_t) (n) >> 56); \
206     ((u_char *) (p))[1] = (u_char) ((uint64_t) (n) >> 48); \
207     ((u_char *) (p))[2] = (u_char) ((uint64_t) (n) >> 40); \
208     ((u_char *) (p))[3] = (u_char) ((uint64_t) (n) >> 32); \
209     ((u_char *) (p))[4] = (u_char) ( (n) >> 24); \
210     ((u_char *) (p))[5] = (u_char) ( (n) >> 16); \
211     ((u_char *) (p))[6] = (u_char) ( (n) >> 8); \
212     ((u_char *) (p))[7] = (u_char) (n)
213
214 #define ngx_mp4_last_trak(mp4)    \
215     &((ngx_http_mp4_trak_t *) mp4->trak.elts)[mp4->trak.nelts - 1]
216
217
218 static ngx_int_t ngx_http_mp4_handler(ngx_http_request_t *r);
219
220 static ngx_int_t ngx_http_mp4_process(ngx_http_mp4_file_t *mp4);
221 static ngx_int_t ngx_http_mp4_read_atom(ngx_http_mp4_file_t *mp4,
222     ngx_http_mp4_atom_handler_t *atom, uint64_t atom_data_size);
223 static ngx_int_t ngx_http_mp4_read(ngx_http_mp4_file_t *mp4, size_t size);
224 static ngx_int_t ngx_http_mp4_read_ftyp_atom(ngx_http_mp4_file_t *mp4,
225     uint64_t atom_data_size);
226 static ngx_int_t ngx_http_mp4_read_moov_atom(ngx_http_mp4_file_t *mp4,
227     uint64_t atom_data_size);
228 static ngx_int_t ngx_http_mp4_read_mdat_atom(ngx_http_mp4_file_t *mp4,
229     uint64_t atom_data_size);
230 static size_t ngx_http_mp4_update_mdat_atom(ngx_http_mp4_file_t *mp4,
231     off_t start_offset, off_t end_offset);
232 static ngx_int_t ngx_http_mp4_read_mvhd_atom(ngx_http_mp4_file_t *mp4,
233     uint64_t atom_data_size);
234 static ngx_int_t ngx_http_mp4_read_trak_atom(ngx_http_mp4_file_t *mp4,
235     uint64_t atom_data_size);
236 static void ngx_http_mp4_update_trak_atom(ngx_http_mp4_file_t *mp4,
237     ngx_http_mp4_trak_t *trak);
238 static ngx_int_t ngx_http_mp4_read_cmov_atom(ngx_http_mp4_file_t *mp4,
239     uint64_t atom_data_size);

```

```

240 static ngx_int_t ngx_http_mp4_read_tkhd_atom(ngx_http_mp4_file_t *mp4,
241     uint64_t atom_data_size);
242 static ngx_int_t ngx_http_mp4_read_mdia_atom(ngx_http_mp4_file_t *mp4,
243     uint64_t atom_data_size);
244 static void ngx_http_mp4_update_mdia_atom(ngx_http_mp4_file_t *mp4,
245     ngx_http_mp4_trak_t *trak);
246 static ngx_int_t ngx_http_mp4_read_mdhd_atom(ngx_http_mp4_file_t *mp4,
247     uint64_t atom_data_size);
248 static ngx_int_t ngx_http_mp4_read_hdlr_atom(ngx_http_mp4_file_t *mp4,
249     uint64_t atom_data_size);
250 static ngx_int_t ngx_http_mp4_read_minf_atom(ngx_http_mp4_file_t *mp4,
251     uint64_t atom_data_size);
252 static void ngx_http_mp4_update_minf_atom(ngx_http_mp4_file_t *mp4,
253     ngx_http_mp4_trak_t *trak);
254 static ngx_int_t ngx_http_mp4_read_dinf_atom(ngx_http_mp4_file_t *mp4,
255     uint64_t atom_data_size);
256 static ngx_int_t ngx_http_mp4_read_vmhd_atom(ngx_http_mp4_file_t *mp4,
257     uint64_t atom_data_size);
258 static ngx_int_t ngx_http_mp4_read_smhd_atom(ngx_http_mp4_file_t *mp4,
259     uint64_t atom_data_size);
260 static ngx_int_t ngx_http_mp4_read_stbl_atom(ngx_http_mp4_file_t *mp4,
261     uint64_t atom_data_size);
262 static void ngx_http_mp4_update_stbl_atom(ngx_http_mp4_file_t *mp4,
263     ngx_http_mp4_trak_t *trak);
264 static ngx_int_t ngx_http_mp4_read_stsd_atom(ngx_http_mp4_file_t *mp4,
265     uint64_t atom_data_size);
266 static ngx_int_t ngx_http_mp4_read_stts_atom(ngx_http_mp4_file_t *mp4,
267     uint64_t atom_data_size);
268 static ngx_int_t ngx_http_mp4_update_stts_atom(ngx_http_mp4_file_t *mp4,
269     ngx_http_mp4_trak_t *trak);
270 static ngx_int_t ngx_http_mp4_crop_stts_data(ngx_http_mp4_file_t *mp4,
271     ngx_http_mp4_trak_t *trak, ngx_uint_t start);
272 static ngx_int_t ngx_http_mp4_read_stss_atom(ngx_http_mp4_file_t *mp4,
273     uint64_t atom_data_size);
274 static ngx_int_t ngx_http_mp4_update_stss_atom(ngx_http_mp4_file_t *mp4,
275     ngx_http_mp4_trak_t *trak);
276 static void ngx_http_mp4_crop_stss_data(ngx_http_mp4_file_t *mp4,
277     ngx_http_mp4_trak_t *trak, ngx_uint_t start);
278 static ngx_int_t ngx_http_mp4_read_ctts_atom(ngx_http_mp4_file_t *mp4,
279     uint64_t atom_data_size);
280 static void ngx_http_mp4_update_ctts_atom(ngx_http_mp4_file_t *mp4,
281     ngx_http_mp4_trak_t *trak);
282 static void ngx_http_mp4_crop_ctts_data(ngx_http_mp4_file_t *mp4,
283     ngx_http_mp4_trak_t *trak, ngx_uint_t start);
284 static ngx_int_t ngx_http_mp4_read_stsc_atom(ngx_http_mp4_file_t *mp4,
285     uint64_t atom_data_size);
286 static ngx_int_t ngx_http_mp4_update_stsc_atom(ngx_http_mp4_file_t *mp4,
287     ngx_http_mp4_trak_t *trak);
288 static ngx_int_t ngx_http_mp4_crop_stsc_data(ngx_http_mp4_file_t *mp4,
289     ngx_http_mp4_trak_t *trak, ngx_uint_t start);
290 static ngx_int_t ngx_http_mp4_read_stsz_atom(ngx_http_mp4_file_t *mp4,
291     uint64_t atom_data_size);
292 static ngx_int_t ngx_http_mp4_update_stsz_atom(ngx_http_mp4_file_t *mp4,
293     ngx_http_mp4_trak_t *trak);
294 static ngx_int_t ngx_http_mp4_read_stco_atom(ngx_http_mp4_file_t *mp4,
295     uint64_t atom_data_size);
296 static ngx_int_t ngx_http_mp4_update_stco_atom(ngx_http_mp4_file_t *mp4,
297     ngx_http_mp4_trak_t *trak);
298 static void ngx_http_mp4_adjust_stco_atom(ngx_http_mp4_file_t *mp4,
299     ngx_http_mp4_trak_t *trak, int32_t adjustment);
300 static ngx_int_t ngx_http_mp4_read_co64_atom(ngx_http_mp4_file_t *mp4,
301     uint64_t atom_data_size);
302 static ngx_int_t ngx_http_mp4_update_co64_atom(ngx_http_mp4_file_t *mp4,
303     ngx_http_mp4_trak_t *trak);
304 static void ngx_http_mp4_adjust_co64_atom(ngx_http_mp4_file_t *mp4,
305     ngx_http_mp4_trak_t *trak, off_t adjustment);
306
307 static char *ngx_http_mp4(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
308 static void *ngx_http_mp4_create_conf(ngx_conf_t *cf);
309 static char *ngx_http_mp4_merge_conf(ngx_conf_t *cf, void *parent, void *child);
310
311
312 static ngx_command_t ngx_http_mp4_commands[] = {
313
314     { ngx_string("mp4"),
315       NGX_HTTP_LOC_CONF|NGX_CONF_NOARGS,

```



```

316     ngx_http_mp4,
317     0,
318     0,
319     NULL },
320
321 { ngx_string("mp4_buffer_size"),
322   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
323   ngx_conf_set_size_slot,
324   NGX_HTTP_LOC_CONF_OFFSET,
325   offsetof(ngx_http_mp4_conf_t, buffer_size),
326   NULL },
327
328 { ngx_string("mp4_max_buffer_size"),
329   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
330   ngx_conf_set_size_slot,
331   NGX_HTTP_LOC_CONF_OFFSET,
332   offsetof(ngx_http_mp4_conf_t, max_buffer_size),
333   NULL },
334
335   ngx_null_command
336 };
337
338
339 static ngx_http_module_t  ngx_http_mp4_module_ctx = {
340     NULL,                          /* preconfiguration */
341     NULL,                          /* postconfiguration */
342
343     NULL,                          /* create main configuration */
344     NULL,                          /* init main configuration */
345
346     NULL,                          /* create server configuration */
347     NULL,                          /* merge server configuration */
348
349     ngx_http_mp4_create_conf,      /* create location configuration */
350     ngx_http_mp4_merge_conf       /* merge location configuration */
351 };
352
353
354 ngx_module_t  ngx_http_mp4_module = {
355     NGX_MODULE_V1,
356     &ngx_http_mp4_module_ctx,      /* module context */
357     ngx_http_mp4_commands,         /* module directives */
358     NGX_HTTP_MODULE,               /* module type */
359     NULL,                          /* init master */
360     NULL,                          /* init module */
361     NULL,                          /* init process */
362     NULL,                          /* init thread */
363     NULL,                          /* exit thread */
364     NULL,                          /* exit process */
365     NULL,                          /* exit master */
366     NGX_MODULE_V1_PADDING
367 };
368
369
370 static ngx_http_mp4_atom_handler_t  ngx_http_mp4_atoms[] = {
371     { "ftyp", ngx_http_mp4_read_ftyp_atom },
372     { "moov", ngx_http_mp4_read_moov_atom },
373     { "mdat", ngx_http_mp4_read_mdat_atom },
374     { NULL, NULL }
375 };
376
377 static ngx_http_mp4_atom_handler_t  ngx_http_mp4_moov_atoms[] = {
378     { "mvhd", ngx_http_mp4_read_mvhd_atom },
379     { "trak", ngx_http_mp4_read_trak_atom },
380     { "cmov", ngx_http_mp4_read_cmov_atom },
381     { NULL, NULL }
382 };
383
384 static ngx_http_mp4_atom_handler_t  ngx_http_mp4_trak_atoms[] = {
385     { "tkhd", ngx_http_mp4_read_tkhd_atom },
386     { "mdia", ngx_http_mp4_read_mdia_atom },
387     { NULL, NULL }
388 };
389
390 static ngx_http_mp4_atom_handler_t  ngx_http_mp4_mdia_atoms[] = {
391     { "mdhd", ngx_http_mp4_read_mdhd_atom },

```

```

392     { "hdlr", ngx_http_mp4_read_hdlr_atom },
393     { "minf", ngx_http_mp4_read_minf_atom },
394     { NULL, NULL }
395 };
396
397 static ngx_http_mp4_atom_handler_t ngx_http_mp4_minf_atoms[] = {
398     { "vmhd", ngx_http_mp4_read_vmhd_atom },
399     { "smhd", ngx_http_mp4_read_smhd_atom },
400     { "dinf", ngx_http_mp4_read_dinf_atom },
401     { "stbl", ngx_http_mp4_read_stbl_atom },
402     { NULL, NULL }
403 };
404
405 static ngx_http_mp4_atom_handler_t ngx_http_mp4_stbl_atoms[] = {
406     { "stsd", ngx_http_mp4_read_stsd_atom },
407     { "stts", ngx_http_mp4_read_stts_atom },
408     { "stss", ngx_http_mp4_read_stss_atom },
409     { "ctts", ngx_http_mp4_read_ctts_atom },
410     { "stsc", ngx_http_mp4_read_stsc_atom },
411     { "stsz", ngx_http_mp4_read_stsz_atom },
412     { "stco", ngx_http_mp4_read_stco_atom },
413     { "co64", ngx_http_mp4_read_co64_atom },
414     { NULL, NULL }
415 };
416
417
418 static ngx_int_t
419 ngx_http_mp4_handler(ngx_http_request_t *r)
420 {
421     u_char          *last;
422     size_t          root;
423     ngx_int_t       rc, start, end;
424     ngx_uint_t      level, length;
425     ngx_str_t       path, value;
426     ngx_log_t       *log;
427     ngx_buf_t       *b;
428     ngx_chain_t     out;
429     ngx_http_mp4_file_t *mp4;
430     ngx_open_file_info_t of;
431     ngx_http_core_loc_conf_t *clcf;
432
433     if (!(r->method & (NGX_HTTP_GET|NGX_HTTP_HEAD))) {
434         return NGX_HTTP_NOT_ALLOWED;
435     }
436
437     if (r->uri.data[r->uri.len - 1] == '/') {
438         return NGX_DECLINED;
439     }
440
441     rc = ngx_http_discard_request_body(r);
442
443     if (rc != NGX_OK) {
444         return rc;
445     }
446
447     last = ngx_http_map_uri_to_path(r, &path, &root, 0);
448     if (last == NULL) {
449         return NGX_HTTP_INTERNAL_SERVER_ERROR;
450     }
451
452     log = r->connection->log;
453
454     path.len = last - path.data;
455
456     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, log, 0,
457                 "http mp4 filename: \"%V\"", &path);
458
459     clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
460
461     ngx_memzero(&of, sizeof(ngx_open_file_info_t));
462
463     of.read_ahead = clcf->read_ahead;
464     of.directio = NGX_MAX_OFF_T_VALUE;
465     of.valid = clcf->open_file_cache_valid;
466     of.min_uses = clcf->open_file_cache_min_uses;
467     of.errors = clcf->open_file_cache_errors;

```

```

468 of.events = clcf->open_file_cache_events;
469
470 if (ngx_http_set_disable_symlinks(r, clcf, &path, &of) != NGX_OK) {
471     return NGX_HTTP_INTERNAL_SERVER_ERROR;
472 }
473
474 if (ngx_open_cached_file(clcf->open_file_cache, &path, &of, r->pool)
475     != NGX_OK)
476 {
477     switch (of.err) {
478
479         case 0:
480             return NGX_HTTP_INTERNAL_SERVER_ERROR;
481
482         case NGX_ENOENT:
483         case NGX_ENOTDIR:
484         case NGX_ENAMETOOLONG:
485
486             level = NGX_LOG_ERR;
487             rc = NGX_HTTP_NOT_FOUND;
488             break;
489
490         case NGX_EACCES:
491 #if (NGX_HAVE_OPENAT)
492         case NGX_EMLINK:
493         case NGX_ELOOP:
494 #endif
495
496             level = NGX_LOG_ERR;
497             rc = NGX_HTTP_FORBIDDEN;
498             break;
499
500         default:
501
502             level = NGX_LOG_CRIT;
503             rc = NGX_HTTP_INTERNAL_SERVER_ERROR;
504             break;
505     }
506
507     if (rc != NGX_HTTP_NOT_FOUND || clcf->log_not_found) {
508         ngx_log_error(level, log, of.err,
509             "%s \"%s\" failed", of.failed, path.data);
510     }
511
512     return rc;
513 }
514
515 if (!of.is_file) {
516
517     if (ngx_close_file(of.fd) == NGX_FILE_ERROR) {
518         ngx_log_error(NGX_LOG_ALERT, log, ngx_errno,
519             ngx_close_file_n " \"%s\" failed", path.data);
520     }
521
522     return NGX_DECLINED;
523 }
524
525 r->root_tested = !r->error_page;
526 r->allow_ranges = 1;
527
528 start = -1;
529 length = 0;
530 r->headers_out.content_length_n = of.size;
531 mp4 = NULL;
532 b = NULL;
533
534 if (r->args.len) {
535
536     if (ngx_http_arg(r, (u_char *) "start", 5, &value) == NGX_OK) {
537
538         /*
539          * A Flash player may send start value with a lot of digits
540          * after dot so strtod() is used instead of atofp(). NaNs and
541          * infinities become negative numbers after (int) conversion.
542          */
543

```

```

544     ngx_set_errno(0);
545     start = (int) (strtod((char *) value.data, NULL) * 1000);
546
547     if (ngx_errno != 0) {
548         start = -1;
549     }
550 }
551
552 if (ngx_http_arg(r, (u_char *) "end", 3, &value) == NGX_OK) {
553
554     ngx_set_errno(0);
555     end = (int) (strtod((char *) value.data, NULL) * 1000);
556
557     if (ngx_errno != 0) {
558         end = -1;
559     }
560
561     if (end > 0) {
562         if (start < 0) {
563             start = 0;
564         }
565
566         if (end > start) {
567             length = end - start;
568         }
569     }
570 }
571 }
572
573 if (start >= 0) {
574     r->single_range = 1;
575
576     mp4 = ngx_palloc(r->pool, sizeof(ngx_http_mp4_file_t));
577     if (mp4 == NULL) {
578         return NGX_HTTP_INTERNAL_SERVER_ERROR;
579     }
580
581     mp4->file.fd = of.fd;
582     mp4->file.name = path;
583     mp4->file.log = r->connection->log;
584     mp4->end = of.size;
585     mp4->start = (ngx_uint_t) start;
586     mp4->length = length;
587     mp4->request = r;
588
589     switch (ngx_http_mp4_process(mp4)) {
590
591     case NGX_DECLINED:
592         if (mp4->buffer) {
593             ngx_pfree(r->pool, mp4->buffer);
594         }
595
596         ngx_pfree(r->pool, mp4);
597         mp4 = NULL;
598
599         break;
600
601     case NGX_OK:
602         r->headers_out.content_length_n = mp4->content_length;
603         break;
604
605     default: /* NGX_ERROR */
606         if (mp4->buffer) {
607             ngx_pfree(r->pool, mp4->buffer);
608         }
609
610         ngx_pfree(r->pool, mp4);
611
612         return NGX_HTTP_INTERNAL_SERVER_ERROR;
613     }
614 }
615
616 log->action = "sending mp4 to client";
617
618 if (clcf->directio <= of.size) {
619

```

```

620  /*
621  * DIRECTIO is set on transfer only
622  * to allow kernel to cache "moov" atom
623  */
624
625  if (ngx_directio_on(of.fd) == NGX_FILE_ERROR) {
626      ngx_log_error(NGX_LOG_ALERT, log, ngx_errno,
627                  ngx_directio_on_n " \"%s\" failed", path.data);
628  }
629
630  of.is_directio = 1;
631
632  if (mp4) {
633      mp4->file.directio = 1;
634  }
635  }
636
637  r->headers_out.status = NGX_HTTP_OK;
638  r->headers_out.last_modified_time = of.mtime;
639
640  if (ngx_http_set_etag(r) != NGX_OK) {
641      return NGX_HTTP_INTERNAL_SERVER_ERROR;
642  }
643
644  if (ngx_http_set_content_type(r) != NGX_OK) {
645      return NGX_HTTP_INTERNAL_SERVER_ERROR;
646  }
647
648  if (mp4 == NULL) {
649      b = ngx_palloc(r->pool, sizeof(ngx_buf_t));
650      if (b == NULL) {
651          return NGX_HTTP_INTERNAL_SERVER_ERROR;
652      }
653
654      b->file = ngx_palloc(r->pool, sizeof(ngx_file_t));
655      if (b->file == NULL) {
656          return NGX_HTTP_INTERNAL_SERVER_ERROR;
657      }
658  }
659
660  rc = ngx_http_send_header(r);
661
662  if (rc == NGX_ERROR || rc > NGX_OK || r->header_only) {
663      return rc;
664  }
665
666  if (mp4) {
667      return ngx_http_output_filter(r, mp4->out);
668  }
669
670  b->file_pos = 0;
671  b->file_last = of.size;
672
673  b->in_file = b->file_last ? 1 : 0;
674  b->last_buf = (r == r->main) ? 1 : 0;
675  b->last_in_chain = 1;
676
677  b->file->fd = of.fd;
678  b->file->name = path;
679  b->file->log = log;
680  b->file->directio = of.is_directio;
681
682  out.buf = b;
683  out.next = NULL;
684
685  return ngx_http_output_filter(r, &out);
686  }
687
688
689  static ngx_int_t
690  ngx_http_mp4_process(ngx_http_mp4_file_t *mp4)
691  {
692      off_t                start_offset, end_offset, adjustment;
693      ngx_int_t           rc;
694      ngx_uint_t         i, j;
695      ngx_chain_t       **prev;

```

```

696 ngx_http_mp4_trak_t *trak;
697 ngx_http_mp4_conf_t *conf;
698
699 ngx_log_debug2(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
700     "mp4 start:%ui, length:%ui", mp4->start, mp4->length);
701
702 conf = ngx_http_get_module_loc_conf(mp4->request, ngx_http_mp4_module);
703
704 mp4->buffer_size = conf->buffer_size;
705
706 rc = ngx_http_mp4_read_atom(mp4, ngx_http_mp4_atoms, mp4->end);
707 if (rc != NGX_OK) {
708     return rc;
709 }
710
711 if (mp4->trak.nelts == 0) {
712     ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
713         "no mp4 trak atoms were found in \"%s\"",
714         mp4->file.name.data);
715     return NGX_ERROR;
716 }
717
718 if (mp4->mdat_atom.buf == NULL) {
719     ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
720         "no mp4 mdat atom was found in \"%s\"",
721         mp4->file.name.data);
722     return NGX_ERROR;
723 }
724
725 prev = &mp4->out;
726
727 if (mp4->ftyp_atom.buf) {
728     *prev = &mp4->ftyp_atom;
729     prev = &mp4->ftyp_atom.next;
730 }
731
732 *prev = &mp4->moov_atom;
733 prev = &mp4->moov_atom.next;
734
735 if (mp4->mvhd_atom.buf) {
736     mp4->moov_size += mp4->mvhd_atom_buf.last - mp4->mvhd_atom_buf.pos;
737     *prev = &mp4->mvhd_atom;
738     prev = &mp4->mvhd_atom.next;
739 }
740
741 start_offset = mp4->end;
742 end_offset = 0;
743 trak = mp4->trak.elts;
744
745 for (i = 0; i < mp4->trak.nelts; i++) {
746     if (ngx_http_mp4_update_stts_atom(mp4, &trak[i]) != NGX_OK) {
747         return NGX_ERROR;
748     }
749
750     if (ngx_http_mp4_update_stss_atom(mp4, &trak[i]) != NGX_OK) {
751         return NGX_ERROR;
752     }
753
754     ngx_http_mp4_update_ctts_atom(mp4, &trak[i]);
755
756     if (ngx_http_mp4_update_stsc_atom(mp4, &trak[i]) != NGX_OK) {
757         return NGX_ERROR;
758     }
759
760     if (ngx_http_mp4_update_stsz_atom(mp4, &trak[i]) != NGX_OK) {
761         return NGX_ERROR;
762     }
763
764     if (trak[i].out[NGX_HTTP_MP4_CO64_DATA].buf) {
765         if (ngx_http_mp4_update_co64_atom(mp4, &trak[i]) != NGX_OK) {
766             return NGX_ERROR;
767         }
768     }
769
770 } else {
771     if (ngx_http_mp4_update_stco_atom(mp4, &trak[i]) != NGX_OK) {

```

```

772         return NGX\_ERROR;
773     }
774 }
775
776 ngx\_http\_mp4\_update\_stbl\_atom(mp4, &trak[i]);
777 ngx\_http\_mp4\_update\_minf\_atom(mp4, &trak[i]);
778 trak[i].size += trak[i].mdhd_size;
779 trak[i].size += trak[i].hdlr_size;
780 ngx\_http\_mp4\_update\_mdia\_atom(mp4, &trak[i]);
781 trak[i].size += trak[i].tkhd_size;
782 ngx\_http\_mp4\_update\_trak\_atom(mp4, &trak[i]);
783
784 mp4->moov_size += trak[i].size;
785
786 if (start_offset > trak[i].start_offset) {
787     start_offset = trak[i].start_offset;
788 }
789
790 if (end_offset < trak[i].end_offset) {
791     end_offset = trak[i].end_offset;
792 }
793
794 *prev = &trak[i].out[NGX\_HTTP\_MP4\_TRAK\_ATOM];
795 prev = &trak[i].out[NGX\_HTTP\_MP4\_TRAK\_ATOM].next;
796
797 for (j = 0; j < NGX\_HTTP\_MP4\_LAST\_ATOM + 1; j++) {
798     if (trak[i].out[j].buf) {
799         *prev = &trak[i].out[j];
800         prev = &trak[i].out[j].next;
801     }
802 }
803 }
804
805 if (end_offset < start_offset) {
806     end_offset = start_offset;
807 }
808
809 mp4->moov_size += 8;
810
811 ngx\_mp4\_set\_32value(mp4->moov_atom_header, mp4->moov_size);
812 ngx\_mp4\_set\_atom\_name(mp4->moov_atom_header, 'm', 'o', 'o', 'v');
813 mp4->content_length += mp4->moov_size;
814
815 *prev = &mp4->mdat_atom;
816
817 if (start_offset > mp4->mdat_data.buf->file_last) {
818     ngx\_log\_error(NGX\_LOG\_ERR, mp4->file.log, 0,
819         "start time is out mp4 mdat atom in \"%s\"",
820         mp4->file.name.data);
821     return NGX\_ERROR;
822 }
823
824 adjustment = mp4->ftyp_size + mp4->moov_size
825     + ngx\_http\_mp4\_update\_mdat\_atom(mp4, start_offset, end_offset)
826     - start_offset;
827
828 ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, mp4->file.log, 0,
829     "mp4 adjustment:%0", adjustment);
830
831 for (i = 0; i < mp4->trak.nelts; i++) {
832     if (trak[i].out[NGX\_HTTP\_MP4\_CO64\_DATA].buf) {
833         ngx\_http\_mp4\_adjust\_co64\_atom(mp4, &trak[i], adjustment);
834     } else {
835         ngx\_http\_mp4\_adjust\_stco\_atom(mp4, &trak[i], (int32_t) adjustment);
836     }
837 }
838
839 return NGX\_OK;
840 }
841
842
843 typedef struct {
844     u_char    size[4];
845     u_char    name[4];
846 } ngx\_mp4\_atom\_header\_t;
847

```

```

848 typedef struct {
849     u_char    size[4];
850     u_char    name[4];
851     u_char    size64[8];
852 } ngx_mp4_atom_header64_t;
853
854
855 static ngx_int_t
856 ngx_http_mp4_read_atom(ngx_http_mp4_file_t *mp4,
857     ngx_http_mp4_atom_handler_t *atom, uint64_t atom_data_size)
858 {
859     off_t      end;
860     size_t     atom_header_size;
861     u_char     *atom_header, *atom_name;
862     uint64_t   atom_size;
863     ngx_int_t  rc;
864     ngx_uint_t n;
865
866     end = mp4->offset + atom_data_size;
867
868     while (mp4->offset < end) {
869
870         if (ngx_http_mp4_read(mp4, sizeof(uint32_t)) != NGX_OK) {
871             return NGX_ERROR;
872         }
873
874         atom_header = mp4->buffer_pos;
875         atom_size = ngx_mp4_get_32value(atom_header);
876         atom_header_size = sizeof(ngx_mp4_atom_header_t);
877
878         if (atom_size == 0) {
879             ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
880                 "mp4 atom end");
881             return NGX_OK;
882         }
883
884         if (atom_size < sizeof(ngx_mp4_atom_header_t)) {
885
886             if (atom_size == 1) {
887
888                 if (ngx_http_mp4_read(mp4, sizeof(ngx_mp4_atom_header64_t))
889                     != NGX_OK)
890                 {
891                     return NGX_ERROR;
892                 }
893
894                 /* 64-bit atom size */
895                 atom_header = mp4->buffer_pos;
896                 atom_size = ngx_mp4_get_64value(atom_header + 8);
897                 atom_header_size = sizeof(ngx_mp4_atom_header64_t);
898
899             } else {
900                 ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
901                     "\\\"%s\\\" mp4 atom is too small:%uL",
902                     mp4->file.name.data, atom_size);
903                 return NGX_ERROR;
904             }
905         }
906
907         if (ngx_http_mp4_read(mp4, sizeof(ngx_mp4_atom_header_t)) != NGX_OK) {
908             return NGX_ERROR;
909         }
910
911         atom_header = mp4->buffer_pos;
912         atom_name = atom_header + sizeof(uint32_t);
913
914         ngx_log_debug4(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
915             "mp4 atom: %*s @%0:%uL",
916             4, atom_name, mp4->offset, atom_size);
917
918         if (atom_size > (uint64_t) (NGX_MAX_OFF_T_VALUE - mp4->offset)
919             || mp4->offset + (off_t) atom_size > end)
920         {
921             ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
922                 "\\\"%s\\\" mp4 atom too large:%uL",
923                 mp4->file.name.data, atom_size);

```



```

924     return NGX\_ERROR;
925 }
926
927 for (n = 0; atom[n].name; n++) {
928     if (ngx\_strncmp(atom_name, atom[n].name, 4) == 0) {
929         ngx\_mp4\_atom\_next(mp4, atom_header_size);
930
931         rc = atom[n].handler(mp4, atom_size - atom_header_size);
932         if (rc != NGX\_OK) {
933             return rc;
934         }
935
936         goto next;
937     }
938 }
939
940 }
941
942 ngx\_mp4\_atom\_next(mp4, atom_size);
943
944 next:
945     continue;
946 }
947
948 return NGX\_OK;
949 }
950
951
952 static ngx\_int\_t
953 ngx\_http\_mp4\_read(ngx\_http\_mp4\_file\_t *mp4, size\_t size)
954 {
955     ssize\_t n;
956
957     if (mp4->buffer_pos + size <= mp4->buffer_end) {
958         return NGX\_OK;
959     }
960
961     if (mp4->offset + (off\_t) mp4->buffer_size > mp4->end) {
962         mp4->buffer_size = (size\_t) (mp4->end - mp4->offset);
963     }
964
965     if (mp4->buffer_size < size) {
966         ngx\_log\_error(NGX\_LOG\_ERR, mp4->file.log, 0,
967             "\"%s\" mp4 file truncated", mp4->file.name.data);
968         return NGX\_ERROR;
969     }
970
971     if (mp4->buffer == NULL) {
972         mp4->buffer = ngx\_palloc(mp4->request->pool, mp4->buffer_size);
973         if (mp4->buffer == NULL) {
974             return NGX\_ERROR;
975         }
976
977         mp4->buffer_start = mp4->buffer;
978     }
979
980     n = ngx\_read\_file(&mp4->file, mp4->buffer_start, mp4->buffer_size,
981         mp4->offset);
982
983     if (n == NGX\_ERROR) {
984         return NGX\_ERROR;
985     }
986
987     if ((size\_t) n != mp4->buffer_size) {
988         ngx\_log\_error(NGX\_LOG\_CRIT, mp4->file.log, 0,
989             ngx\_read\_file\_n " read only %z of %z from \"%s\"",
990             n, mp4->buffer_size, mp4->file.name.data);
991         return NGX\_ERROR;
992     }
993
994     mp4->buffer_pos = mp4->buffer_start;
995     mp4->buffer_end = mp4->buffer_start + mp4->buffer_size;
996
997     return NGX\_OK;
998 }
999

```

```

1000 static ngx_int_t
1001 ngx_http_mp4_read_ftyp_atom(ngx_http_mp4_file_t *mp4, uint64_t atom_data_size)
1002 {
1003     u_char      *ftyp_atom;
1004     size_t      atom_size;
1005     ngx_buf_t   *atom;
1006
1007     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0, "mp4 ftyp atom");
1008
1009     if (atom_data_size > 1024
1010         || ngx_mp4_atom_data(mp4) + (size_t) atom_data_size > mp4->buffer_end)
1011     {
1012         ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
1013             "\\\"%s\\\" mp4 ftyp atom is too large:%uL",
1014             mp4->file.name.data, atom_data_size);
1015         return NGX_ERROR;
1016     }
1017
1018     atom_size = sizeof(ngx_mp4_atom_header_t) + (size_t) atom_data_size;
1019
1020     ftyp_atom = ngx_palloc(mp4->request->pool, atom_size);
1021     if (ftyp_atom == NULL) {
1022         return NGX_ERROR;
1023     }
1024
1025     ngx_mp4_set_32value(ftyp_atom, atom_size);
1026     ngx_mp4_set_atom_name(ftyp_atom, 'f', 't', 'y', 'p');
1027
1028     /*
1029     * only moov atom content is guaranteed to be in mp4->buffer
1030     * during sending response, so ftyp atom content should be copied
1031     */
1032     ngx_memcpy(ftyp_atom + sizeof(ngx_mp4_atom_header_t),
1033         ngx_mp4_atom_data(mp4), (size_t) atom_data_size);
1034
1035     atom = &mp4->ftyp_atom_buf;
1036     atom->temporary = 1;
1037     atom->pos = ftyp_atom;
1038     atom->last = ftyp_atom + atom_size;
1039
1040     mp4->ftyp_atom.buf = atom;
1041     mp4->ftyp_size = atom_size;
1042     mp4->content_length = atom_size;
1043
1044     ngx_mp4_atom_next(mp4, atom_data_size);
1045
1046     return NGX_OK;
1047 }
1048
1049
1050
1051 /*
1052 * Small excess buffer to process atoms after moov atom, mp4->buffer_start
1053 * will be set to this buffer part after moov atom processing.
1054 */
1055 #define NGX_HTTP_MP4_MOOV_BUFFER_EXCESS (4 * 1024)
1056
1057 static ngx_int_t
1058 ngx_http_mp4_read_moov_atom(ngx_http_mp4_file_t *mp4, uint64_t atom_data_size)
1059 {
1060     ngx_int_t      rc;
1061     ngx_uint_t     no_mdat;
1062     ngx_buf_t      *atom;
1063     ngx_http_mp4_conf_t *conf;
1064
1065     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0, "mp4 moov atom");
1066
1067     no_mdat = (mp4->mdat_atom.buf == NULL);
1068
1069     if (no_mdat && mp4->start == 0 && mp4->length == 0) {
1070         /*
1071         * send original file if moov atom resides before
1072         * mdat atom and client requests integral file
1073         */
1074         return NGX_DECLINED;
1075     }

```

```

1076 conf = ngx_http_get_module_loc_conf(mp4->request, ngx_http_mp4_module);
1077
1078
1079 if (atom_data_size > mp4->buffer_size) {
1080
1081     if (atom_data_size > conf->max_buffer_size) {
1082         ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
1083             "\"%s\" mp4 moov atom is too large:%uL, \"
1084             \"you may want to increase mp4_max_buffer_size\",
1085             mp4->file.name.data, atom_data_size);
1086         return NGX_ERROR;
1087     }
1088
1089     ngx_pfree(mp4->request->pool, mp4->buffer);
1090     mp4->buffer = NULL;
1091     mp4->buffer_pos = NULL;
1092     mp4->buffer_end = NULL;
1093
1094     mp4->buffer_size = (size_t) atom_data_size
1095         + NGX_HTTP_MP4_MOOV_BUFFER_EXCESS * no_mdat;
1096 }
1097
1098 if (ngx_http_mp4_read(mp4, (size_t) atom_data_size) != NGX_OK) {
1099     return NGX_ERROR;
1100 }
1101
1102 mp4->trak.elts = &mp4->traks;
1103 mp4->trak.size = sizeof(ngx_http_mp4_trak_t);
1104 mp4->trak.nalloc = 2;
1105 mp4->trak.pool = mp4->request->pool;
1106
1107 atom = &mp4->moov_atom_buf;
1108 atom->temporary = 1;
1109 atom->pos = mp4->moov_atom_header;
1110 atom->last = mp4->moov_atom_header + 8;
1111
1112 mp4->moov_atom.buf = &mp4->moov_atom_buf;
1113
1114 rc = ngx_http_mp4_read_atom(mp4, ngx_http_mp4_moov_atoms, atom_data_size);
1115
1116 ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0, "mp4 moov atom done");
1117
1118 if (no_mdat) {
1119     mp4->buffer_start = mp4->buffer_pos;
1120     mp4->buffer_size = NGX_HTTP_MP4_MOOV_BUFFER_EXCESS;
1121
1122     if (mp4->buffer_start + mp4->buffer_size > mp4->buffer_end) {
1123         mp4->buffer = NULL;
1124         mp4->buffer_pos = NULL;
1125         mp4->buffer_end = NULL;
1126     }
1127
1128 } else {
1129     /* skip atoms after moov atom */
1130     mp4->offset = mp4->end;
1131 }
1132
1133 return rc;
1134 }
1135
1136
1137 static ngx_int_t
1138 ngx_http_mp4_read_mdat_atom(ngx_http_mp4_file_t *mp4, uint64_t atom_data_size)
1139 {
1140     ngx_buf_t *data;
1141
1142     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0, "mp4 mdat atom");
1143
1144     data = &mp4->mdat_data_buf;
1145     data->file = &mp4->file;
1146     data->in_file = 1;
1147     data->last_buf = 1;
1148     data->last_in_chain = 1;
1149     data->file_last = mp4->offset + atom_data_size;
1150
1151     mp4->mdat_atom.buf = &mp4->mdat_atom_buf;

```

```

1152 mp4->mdat_atom.next = &mp4->mdat_data;
1153 mp4->mdat_data.buf = data;
1154
1155 if (mp4->trak.nelts) {
1156     /* skip atoms after mdat atom */
1157     mp4->offset = mp4->end;
1158
1159 } else {
1160     ngx_mp4_atom_next(mp4, atom_data_size);
1161 }
1162
1163 return NGX_OK;
1164 }
1165
1166 static size_t
1167 ngx_http_mp4_update_mdat_atom(ngx_http_mp4_file_t *mp4, off_t start_offset,
1168     off_t end_offset)
1169 {
1170     off_t      atom_data_size;
1171     u_char     *atom_header;
1172     uint32_t   atom_header_size;
1173     uint64_t   atom_size;
1174     ngx_buf_t  *atom;
1175
1176     atom_data_size = end_offset - start_offset;
1177     mp4->mdat_data.buf->file_pos = start_offset;
1178     mp4->mdat_data.buf->file_last = end_offset;
1179
1180     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
1181         "mdat new offset @%0:%0", start_offset, atom_data_size);
1182
1183     atom_header = mp4->mdat_atom_header;
1184
1185     if ((uint64_t) atom_data_size > (uint64_t) 0xffffffff) {
1186         atom_size = 1;
1187         atom_header_size = sizeof(ngx_mp4_atom_header64_t);
1188         ngx_mp4_set_64value(atom_header + sizeof(ngx_mp4_atom_header_t),
1189             sizeof(ngx_mp4_atom_header64_t) + atom_data_size);
1190     } else {
1191         atom_size = sizeof(ngx_mp4_atom_header_t) + atom_data_size;
1192         atom_header_size = sizeof(ngx_mp4_atom_header_t);
1193     }
1194
1195     mp4->content_length += atom_header_size + atom_data_size;
1196
1197     ngx_mp4_set_32value(atom_header, atom_size);
1198     ngx_mp4_set_atom_name(atom_header, 'm', 'd', 'a', 't');
1199
1200     atom = &mp4->mdat_atom_buf;
1201     atom->temporary = 1;
1202     atom->pos = atom_header;
1203     atom->last = atom_header + atom_header_size;
1204
1205     return atom_header_size;
1206 }
1207
1208
1209
1210 typedef struct {
1211     u_char    size[4];
1212     u_char    name[4];
1213     u_char    version[1];
1214     u_char    flags[3];
1215     u_char    creation_time[4];
1216     u_char    modification_time[4];
1217     u_char    timescale[4];
1218     u_char    duration[4];
1219     u_char    rate[4];
1220     u_char    volume[2];
1221     u_char    reserved[10];
1222     u_char    matrix[36];
1223     u_char    preview_time[4];
1224     u_char    preview_duration[4];
1225     u_char    poster_time[4];
1226     u_char    selection_time[4];
1227     u_char    selection_duration[4];

```

```

1228     u_char    current_time[4];
1229     u_char    next_track_id[4];
1230 } ngx_mp4_mvhd_atom_t;
1231
1232 typedef struct {
1233     u_char    size[4];
1234     u_char    name[4];
1235     u_char    version[1];
1236     u_char    flags[3];
1237     u_char    creation_time[8];
1238     u_char    modification_time[8];
1239     u_char    timescale[4];
1240     u_char    duration[8];
1241     u_char    rate[4];
1242     u_char    volume[2];
1243     u_char    reserved[10];
1244     u_char    matrix[36];
1245     u_char    preview_time[4];
1246     u_char    preview_duration[4];
1247     u_char    poster_time[4];
1248     u_char    selection_time[4];
1249     u_char    selection_duration[4];
1250     u_char    current_time[4];
1251     u_char    next_track_id[4];
1252 } ngx_mp4_mvhd64_atom_t;
1253
1254
1255 static ngx_int_t
1256 ngx_http_mp4_read_mvhd_atom(ngx_http_mp4_file_t *mp4, uint64_t atom_data_size)
1257 {
1258     u_char          *atom_header;
1259     size_t          atom_size;
1260     uint32_t        timescale;
1261     uint64_t        duration, start_time, length_time;
1262     ngx_buf_t       *atom;
1263     ngx_mp4_mvhd_atom_t *mvhd_atom;
1264     ngx_mp4_mvhd64_atom_t *mvhd64_atom;
1265
1266     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0, "mp4 mvhd atom");
1267
1268     atom_header = ngx_mp4_atom_header(mp4);
1269     mvhd_atom = (ngx_mp4_mvhd_atom_t *) atom_header;
1270     mvhd64_atom = (ngx_mp4_mvhd64_atom_t *) atom_header;
1271     ngx_mp4_set_atom_name(atom_header, 'm', 'v', 'h', 'd');
1272
1273     if (ngx_mp4_atom_data_size(ngx_mp4_mvhd_atom_t) > atom_data_size) {
1274         ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
1275             "\"%s\" mp4 mvhd atom too small", mp4->file.name.data);
1276         return NGX_ERROR;
1277     }
1278
1279     if (mvhd_atom->version[0] == 0) {
1280         /* version 0: 32-bit duration */
1281         timescale = ngx_mp4_get_32value(mvhd_atom->timescale);
1282         duration = ngx_mp4_get_32value(mvhd_atom->duration);
1283     }
1284     else {
1285         /* version 1: 64-bit duration */
1286
1287         if (ngx_mp4_atom_data_size(ngx_mp4_mvhd64_atom_t) > atom_data_size) {
1288             ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
1289                 "\"%s\" mp4 mvhd atom too small",
1290                 mp4->file.name.data);
1291             return NGX_ERROR;
1292         }
1293
1294         timescale = ngx_mp4_get_32value(mvhd64_atom->timescale);
1295         duration = ngx_mp4_get_64value(mvhd64_atom->duration);
1296     }
1297
1298     mp4->timescale = timescale;
1299
1300     ngx_log_debug3(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
1301         "mvhd timescale:%uD, duration:%uL, time:%.3fs",
1302         timescale, duration, (double) duration / timescale);
1303

```

```

1304 start_time = (uint64_t) mp4->start * timescale / 1000;
1305
1306 if (duration < start_time) {
1307     ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
1308                 "\"%s\" mp4 start time exceeds file duration",
1309                 mp4->file.name.data);
1310     return NGX_ERROR;
1311 }
1312
1313 duration -= start_time;
1314
1315 if (mp4->length) {
1316     length_time = (uint64_t) mp4->length * timescale / 1000;
1317
1318     if (duration > length_time) {
1319         duration = length_time;
1320     }
1321 }
1322
1323 ngx_log_debug2(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
1324              "mvhd new duration:%uL, time:%.3fs",
1325              duration, (double) duration / timescale);
1326
1327 atom_size = sizeof(ngx_mp4_atom_header_t) + (size_t) atom_data_size;
1328 ngx_mp4_set_32value(mvhd_atom->size, atom_size);
1329
1330 if (mvhd_atom->version[0] == 0) {
1331     ngx_mp4_set_32value(mvhd_atom->duration, duration);
1332 }
1333 } else {
1334     ngx_mp4_set_64value(mvhd64_atom->duration, duration);
1335 }
1336
1337 atom = &mp4->mvhd_atom_buf;
1338 atom->temporary = 1;
1339 atom->pos = atom_header;
1340 atom->last = atom_header + atom_size;
1341
1342 mp4->mvhd_atom.buf = atom;
1343
1344 ngx_mp4_atom_next(mp4, atom_data_size);
1345
1346 return NGX_OK;
1347 }
1348
1349
1350 static ngx_int_t
1351 ngx_http_mp4_read_trak_atom(ngx_http_mp4_file_t *mp4, uint64_t atom_data_size)
1352 {
1353     u_char          *atom_header, *atom_end;
1354     off_t           atom_file_end;
1355     ngx_int_t       rc;
1356     ngx_buf_t       *atom;
1357     ngx_http_mp4_trak_t *trak;
1358
1359     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0, "mp4 trak atom");
1360
1361     trak = ngx_array_push(&mp4->trak);
1362     if (trak == NULL) {
1363         return NGX_ERROR;
1364     }
1365
1366     ngx_memzero(trak, sizeof(ngx_http_mp4_trak_t));
1367
1368     atom_header = ngx_mp4_atom_header(mp4);
1369     ngx_mp4_set_atom_name(atom_header, 't', 'r', 'a', 'k');
1370
1371     atom = &trak->trak_atom_buf;
1372     atom->temporary = 1;
1373     atom->pos = atom_header;
1374     atom->last = atom_header + sizeof(ngx_mp4_atom_header_t);
1375
1376     trak->out[NGX_HTTP_MP4_TRAK_ATOM].buf = atom;
1377
1378     atom_end = mp4->buffer_pos + (size_t) atom_data_size;
1379     atom_file_end = mp4->offset + atom_data_size;

```

```

1380 rc = ngx_http_mp4_read_atom(mp4, ngx_http_mp4_trak_atoms, atom_data_size);
1381
1382
1383 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
1384 "mp4 trak atom: %i", rc);
1385
1386 if (rc == NGX_DECLINED) {
1387     /* skip this trak */
1388     ngx_memzero(trak, sizeof(ngx_http_mp4_trak_t));
1389     mp4->trak.nelts--;
1390     mp4->buffer_pos = atom_end;
1391     mp4->offset = atom_file_end;
1392     return NGX_OK;
1393 }
1394
1395 return rc;
1396 }
1397
1398
1399 static void
1400 ngx_http_mp4_update_trak_atom(ngx_http_mp4_file_t *mp4,
1401 ngx_http_mp4_trak_t *trak)
1402 {
1403     ngx_buf_t *atom;
1404
1405     trak->size += sizeof(ngx_mp4_atom_header_t);
1406     atom = &trak->trak_atom_buf;
1407     ngx_mp4_set_32value(atom->pos, trak->size);
1408 }
1409
1410
1411 static ngx_int_t
1412 ngx_http_mp4_read_cmov_atom(ngx_http_mp4_file_t *mp4, uint64_t atom_data_size)
1413 {
1414     ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
1415 "\'%s\' mp4 compressed moov atom (cmov) is not supported",
1416 mp4->file.name.data);
1417
1418     return NGX_ERROR;
1419 }
1420
1421
1422 typedef struct {
1423     u_char size[4];
1424     u_char name[4];
1425     u_char version[1];
1426     u_char flags[3];
1427     u_char creation_time[4];
1428     u_char modification_time[4];
1429     u_char track_id[4];
1430     u_char reserved1[4];
1431     u_char duration[4];
1432     u_char reserved2[8];
1433     u_char layer[2];
1434     u_char group[2];
1435     u_char volume[2];
1436     u_char reserved3[2];
1437     u_char matrix[36];
1438     u_char width[4];
1439     u_char height[4];
1440 } ngx_mp4_tkhd_atom_t;
1441
1442 typedef struct {
1443     u_char size[4];
1444     u_char name[4];
1445     u_char version[1];
1446     u_char flags[3];
1447     u_char creation_time[8];
1448     u_char modification_time[8];
1449     u_char track_id[4];
1450     u_char reserved1[4];
1451     u_char duration[8];
1452     u_char reserved2[8];
1453     u_char layer[2];
1454     u_char group[2];
1455     u_char volume[2];

```

```

1456     u_char     reserved3[2];
1457     u_char     matrix[36];
1458     u_char     width[4];
1459     u_char     height[4];
1460 } ngx_mp4_tkhd64_atom_t;
1461
1462
1463 static ngx_int_t
1464 ngx_http_mp4_read_tkhd_atom(ngx_http_mp4_file_t *mp4, uint64_t atom_data_size)
1465 {
1466     u_char     *atom_header;
1467     size_t     atom_size;
1468     uint64_t   duration, start_time, length_time;
1469     ngx_buf_t  *atom;
1470     ngx_http_mp4_trak_t *trak;
1471     ngx_mp4_tkhd_atom_t *tkhd_atom;
1472     ngx_mp4_tkhd64_atom_t *tkhd64_atom;
1473
1474     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0, "mp4 tkhd atom");
1475
1476     atom_header = ngx_mp4_atom_header(mp4);
1477     tkhd_atom = (ngx_mp4_tkhd_atom_t *) atom_header;
1478     tkhd64_atom = (ngx_mp4_tkhd64_atom_t *) atom_header;
1479     ngx_mp4_set_atom_name(tkhd_atom, 't', 'k', 'h', 'd');
1480
1481     if (ngx_mp4_atom_data_size(ngx_mp4_tkhd_atom_t) > atom_data_size) {
1482         ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
1483             "\\\"%s\\\" mp4 tkhd atom too small", mp4->file.name.data);
1484         return NGX_ERROR;
1485     }
1486
1487     if (tkhd_atom->version[0] == 0) {
1488         /* version 0: 32-bit duration */
1489         duration = ngx_mp4_get_32value(tkhd_atom->duration);
1490
1491     } else {
1492         /* version 1: 64-bit duration */
1493
1494         if (ngx_mp4_atom_data_size(ngx_mp4_tkhd64_atom_t) > atom_data_size) {
1495             ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
1496                 "\\\"%s\\\" mp4 tkhd atom too small",
1497                 mp4->file.name.data);
1498             return NGX_ERROR;
1499         }
1500
1501         duration = ngx_mp4_get_64value(tkhd64_atom->duration);
1502     }
1503
1504     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
1505         "tkhd duration:%uL, time:%.3fs",
1506         duration, (double) duration / mp4->timescale);
1507
1508     start_time = (uint64_t) mp4->start * mp4->timescale / 1000;
1509
1510     if (duration <= start_time) {
1511         ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
1512             "tkhd duration is less than start time");
1513         return NGX_DECLINED;
1514     }
1515
1516     duration -= start_time;
1517
1518     if (mp4->length) {
1519         length_time = (uint64_t) mp4->length * mp4->timescale / 1000;
1520
1521         if (duration > length_time) {
1522             duration = length_time;
1523         }
1524     }
1525
1526     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
1527         "tkhd new duration:%uL, time:%.3fs",
1528         duration, (double) duration / mp4->timescale);
1529
1530     atom_size = sizeof(ngx_mp4_atom_header_t) + (size_t) atom_data_size;
1531

```



```

1532     trak = ngx_mp4_last_trak(mp4);
1533     trak->tkhd_size = atom_size;
1534
1535     ngx_mp4_set_32value(tkhd_atom->size, atom_size);
1536
1537     if (tkhd_atom->version[0] == 0) {
1538         ngx_mp4_set_32value(tkhd_atom->duration, duration);
1539     }
1540     else {
1541         ngx_mp4_set_64value(tkhd64_atom->duration, duration);
1542     }
1543
1544     atom = &trak->tkhd_atom_buf;
1545     atom->temporary = 1;
1546     atom->pos = atom_header;
1547     atom->last = atom_header + atom_size;
1548
1549     trak->out[NGX_HTTP_MP4_TKHD_ATOM].buf = atom;
1550
1551     ngx_mp4_atom_next(mp4, atom_data_size);
1552
1553     return NGX_OK;
1554 }
1555
1556
1557 static ngx_int_t
1558 ngx_http_mp4_read_mdia_atom(ngx_http_mp4_file_t *mp4, uint64_t atom_data_size)
1559 {
1560     u_char          *atom_header;
1561     ngx_buf_t       *atom;
1562     ngx_http_mp4_trak_t *trak;
1563
1564     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0, "process mdia atom");
1565
1566     atom_header = ngx_mp4_atom_header(mp4);
1567     ngx_mp4_set_atom_name(atom_header, 'm', 'd', 'i', 'a');
1568
1569     trak = ngx_mp4_last_trak(mp4);
1570
1571     atom = &trak->mdia_atom_buf;
1572     atom->temporary = 1;
1573     atom->pos = atom_header;
1574     atom->last = atom_header + sizeof(ngx_mp4_atom_header_t);
1575
1576     trak->out[NGX_HTTP_MP4_MDIA_ATOM].buf = atom;
1577
1578     return ngx_http_mp4_read_atom(mp4, ngx_http_mp4_mdia_atoms, atom_data_size);
1579 }
1580
1581
1582 static void
1583 ngx_http_mp4_update_mdia_atom(ngx_http_mp4_file_t *mp4,
1584     ngx_http_mp4_trak_t *trak)
1585 {
1586     ngx_buf_t *atom;
1587
1588     trak->size += sizeof(ngx_mp4_atom_header_t);
1589     atom = &trak->mdia_atom_buf;
1590     ngx_mp4_set_32value(atom->pos, trak->size);
1591 }
1592
1593
1594 typedef struct {
1595     u_char    size[4];
1596     u_char    name[4];
1597     u_char    version[1];
1598     u_char    flags[3];
1599     u_char    creation_time[4];
1600     u_char    modification_time[4];
1601     u_char    timescale[4];
1602     u_char    duration[4];
1603     u_char    language[2];
1604     u_char    quality[2];
1605 } ngx_mp4_mdhd_atom_t;
1606
1607 typedef struct {

```

```

1608     u_char    size[4];
1609     u_char    name[4];
1610     u_char    version[1];
1611     u_char    flags[3];
1612     u_char    creation_time[8];
1613     u_char    modification_time[8];
1614     u_char    timescale[4];
1615     u_char    duration[8];
1616     u_char    language[2];
1617     u_char    quality[2];
1618 } ngx_mp4_mdhd64_atom_t;
1619
1620
1621 static ngx_int_t
1622 ngx_http_mp4_read_mdhd_atom(ngx_http_mp4_file_t *mp4, uint64_t atom_data_size)
1623 {
1624     u_char          *atom_header;
1625     size_t          atom_size;
1626     uint32_t        timescale;
1627     uint64_t        duration, start_time, length_time;
1628     ngx_buf_t       *atom;
1629     ngx_http_mp4_trak_t *trak;
1630     ngx_mp4_mdhd_atom_t *mdhd_atom;
1631     ngx_mp4_mdhd64_atom_t *mdhd64_atom;
1632
1633     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0, "mp4 mdhd atom");
1634
1635     atom_header = ngx_mp4_atom_header(mp4);
1636     mdhd_atom = (ngx_mp4_mdhd_atom_t *) atom_header;
1637     mdhd64_atom = (ngx_mp4_mdhd64_atom_t *) atom_header;
1638     ngx_mp4_set_atom_name(mdhd_atom, 'm', 'd', 'h', 'd');
1639
1640     if (ngx_mp4_atom_data_size(ngx_mp4_mdhd_atom_t) > atom_data_size) {
1641         ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
1642             "\\\"%s\\\" mp4 mdhd atom too small", mp4->file.name.data);
1643         return NGX_ERROR;
1644     }
1645
1646     if (mdhd_atom->version[0] == 0) {
1647         /* version 0: everything is 32-bit */
1648         timescale = ngx_mp4_get_32value(mdhd_atom->timescale);
1649         duration = ngx_mp4_get_32value(mdhd_atom->duration);
1650
1651     } else {
1652         /* version 1: 64-bit duration and 32-bit timescale */
1653
1654         if (ngx_mp4_atom_data_size(ngx_mp4_mdhd64_atom_t) > atom_data_size) {
1655             ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
1656                 "\\\"%s\\\" mp4 mdhd atom too small",
1657                 mp4->file.name.data);
1658             return NGX_ERROR;
1659         }
1660
1661         timescale = ngx_mp4_get_32value(mdhd64_atom->timescale);
1662         duration = ngx_mp4_get_64value(mdhd64_atom->duration);
1663     }
1664
1665     ngx_log_debug3(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
1666         "mdhd timescale:%uD, duration:%uL, time:%.3fs",
1667         timescale, duration, (double) duration / timescale);
1668
1669     start_time = (uint64_t) mp4->start * timescale / 1000;
1670
1671     if (duration <= start_time) {
1672         ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
1673             "mdhd duration is less than start time");
1674         return NGX_DECLINED;
1675     }
1676
1677     duration -= start_time;
1678
1679     if (mp4->length) {
1680         length_time = (uint64_t) mp4->length * timescale / 1000;
1681
1682         if (duration > length_time) {
1683             duration = length_time;

```

```

1684     }
1685 }
1686
1687 ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, mp4->file.log, 0,
1688     "mdhd new duration:%uL, time:%.3fs",
1689     duration, (double) duration / timescale);
1690
1691 atom_size = sizeof(ngx\_mp4\_atom\_header\_t) + (size_t) atom_data_size;
1692
1693 trak = ngx\_mp4\_last\_trak(mp4);
1694 trak->mdhd_size = atom_size;
1695 trak->timescale = timescale;
1696
1697 ngx\_mp4\_set\_32value(mdhd_atom->size, atom_size);
1698
1699 if (mdhd_atom->version[0] == 0) {
1700     ngx\_mp4\_set\_32value(mdhd_atom->duration, duration);
1701
1702 } else {
1703     ngx\_mp4\_set\_64value(mdhd64_atom->duration, duration);
1704 }
1705
1706 atom = &trak->mdhd_atom_buf;
1707 atom->temporary = 1;
1708 atom->pos = atom_header;
1709 atom->last = atom_header + atom_size;
1710
1711 trak->out[NGX\_HTTP\_MP4\_MDHD\_ATOM].buf = atom;
1712
1713 ngx\_mp4\_atom\_next(mp4, atom_data_size);
1714
1715 return NGX\_OK;
1716 }
1717
1718
1719 static ngx\_int\_t
1720 ngx\_http\_mp4\_read\_hdlr\_atom(ngx\_http\_mp4\_file\_t *mp4, uint64\_t atom_data_size)
1721 {
1722     u\_char          *atom_header;
1723     size\_t         atom_size;
1724     ngx\_buf\_t      *atom;
1725     ngx\_http\_mp4\_trak\_t *trak;
1726
1727     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, mp4->file.log, 0, "mp4 hdlr atom");
1728
1729     atom_header = ngx\_mp4\_atom\_header(mp4);
1730     atom_size = sizeof(ngx\_mp4\_atom\_header\_t) + (size_t) atom_data_size;
1731     ngx\_mp4\_set\_32value(atom_header, atom_size);
1732     ngx\_mp4\_set\_atom\_name(atom_header, 'h', 'd', 'l', 'r');
1733
1734     trak = ngx\_mp4\_last\_trak(mp4);
1735
1736     atom = &trak->hdlr_atom_buf;
1737     atom->temporary = 1;
1738     atom->pos = atom_header;
1739     atom->last = atom_header + atom_size;
1740
1741     trak->hdlr_size = atom_size;
1742     trak->out[NGX\_HTTP\_MP4\_HDLR\_ATOM].buf = atom;
1743
1744     ngx\_mp4\_atom\_next(mp4, atom_data_size);
1745
1746     return NGX\_OK;
1747 }
1748
1749
1750 static ngx\_int\_t
1751 ngx\_http\_mp4\_read\_minf\_atom(ngx\_http\_mp4\_file\_t *mp4, uint64\_t atom_data_size)
1752 {
1753     u\_char          *atom_header;
1754     ngx\_buf\_t      *atom;
1755     ngx\_http\_mp4\_trak\_t *trak;
1756
1757     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, mp4->file.log, 0, "process minf atom");
1758
1759     atom_header = ngx\_mp4\_atom\_header(mp4);

```

```

1760     ngx_mp4_set_atom_name(atom_header, 'm', 'i', 'n', 'f');
1761
1762     trak = ngx_mp4_last_trak(mp4);
1763
1764     atom = &trak->minf_atom_buf;
1765     atom->temporary = 1;
1766     atom->pos = atom_header;
1767     atom->last = atom_header + sizeof(ngx_mp4_atom_header_t);
1768
1769     trak->out[NGX_HTTP_MP4_MINF_ATOM].buf = atom;
1770
1771     return ngx_http_mp4_read_atom(mp4, ngx_http_mp4_minf_atoms, atom_data_size);
1772 }
1773
1774
1775 static void
1776 ngx_http_mp4_update_minf_atom(ngx_http_mp4_file_t *mp4,
1777     ngx_http_mp4_trak_t *trak)
1778 {
1779     ngx_buf_t *atom;
1780
1781     trak->size += sizeof(ngx_mp4_atom_header_t)
1782         + trak->vmhd_size
1783         + trak->smhd_size
1784         + trak->dinf_size;
1785     atom = &trak->minf_atom_buf;
1786     ngx_mp4_set_32value(atom->pos, trak->size);
1787 }
1788
1789
1790 static ngx_int_t
1791 ngx_http_mp4_read_vmhd_atom(ngx_http_mp4_file_t *mp4, uint64_t atom_data_size)
1792 {
1793     u_char          *atom_header;
1794     size_t          atom_size;
1795     ngx_buf_t       *atom;
1796     ngx_http_mp4_trak_t *trak;
1797
1798     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0, "mp4 vmhd atom");
1799
1800     atom_header = ngx_mp4_atom_header(mp4);
1801     atom_size = sizeof(ngx_mp4_atom_header_t) + (size_t) atom_data_size;
1802     ngx_mp4_set_32value(atom_header, atom_size);
1803     ngx_mp4_set_atom_name(atom_header, 'v', 'm', 'h', 'd');
1804
1805     trak = ngx_mp4_last_trak(mp4);
1806
1807     atom = &trak->vmhd_atom_buf;
1808     atom->temporary = 1;
1809     atom->pos = atom_header;
1810     atom->last = atom_header + atom_size;
1811
1812     trak->vmhd_size += atom_size;
1813     trak->out[NGX_HTTP_MP4_VMHD_ATOM].buf = atom;
1814
1815     ngx_mp4_atom_next(mp4, atom_data_size);
1816
1817     return NGX_OK;
1818 }
1819
1820
1821 static ngx_int_t
1822 ngx_http_mp4_read_smhd_atom(ngx_http_mp4_file_t *mp4, uint64_t atom_data_size)
1823 {
1824     u_char          *atom_header;
1825     size_t          atom_size;
1826     ngx_buf_t       *atom;
1827     ngx_http_mp4_trak_t *trak;
1828
1829     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0, "mp4 smhd atom");
1830
1831     atom_header = ngx_mp4_atom_header(mp4);
1832     atom_size = sizeof(ngx_mp4_atom_header_t) + (size_t) atom_data_size;
1833     ngx_mp4_set_32value(atom_header, atom_size);
1834     ngx_mp4_set_atom_name(atom_header, 's', 'm', 'h', 'd');
1835

```

```

1836     trak = ngx_mp4_last_trak(mp4);
1837
1838     atom = &trak->smhd_atom_buf;
1839     atom->temporary = 1;
1840     atom->pos = atom_header;
1841     atom->last = atom_header + atom_size;
1842
1843     trak->smhd_size += atom_size;
1844     trak->out[NGX_HTTP_MP4_SMHD_ATOM].buf = atom;
1845
1846     ngx_mp4_atom_next(mp4, atom_data_size);
1847
1848     return NGX_OK;
1849 }
1850
1851
1852 static ngx_int_t
1853 ngx_http_mp4_read_dinf_atom(ngx_http_mp4_file_t *mp4, uint64_t atom_data_size)
1854 {
1855     u_char          *atom_header;
1856     size_t          atom_size;
1857     ngx_buf_t       *atom;
1858     ngx_http_mp4_trak_t *trak;
1859
1860     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0, "mp4 dinf atom");
1861
1862     atom_header = ngx_mp4_atom_header(mp4);
1863     atom_size = sizeof(ngx_mp4_atom_header_t) + (size_t) atom_data_size;
1864     ngx_mp4_set_32value(atom_header, atom_size);
1865     ngx_mp4_set_atom_name(atom_header, 'd', 'i', 'n', 'f');
1866
1867     trak = ngx_mp4_last_trak(mp4);
1868
1869     atom = &trak->dinf_atom_buf;
1870     atom->temporary = 1;
1871     atom->pos = atom_header;
1872     atom->last = atom_header + atom_size;
1873
1874     trak->dinf_size += atom_size;
1875     trak->out[NGX_HTTP_MP4_DINF_ATOM].buf = atom;
1876
1877     ngx_mp4_atom_next(mp4, atom_data_size);
1878
1879     return NGX_OK;
1880 }
1881
1882
1883 static ngx_int_t
1884 ngx_http_mp4_read_stbl_atom(ngx_http_mp4_file_t *mp4, uint64_t atom_data_size)
1885 {
1886     u_char          *atom_header;
1887     ngx_buf_t       *atom;
1888     ngx_http_mp4_trak_t *trak;
1889
1890     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0, "process stbl atom");
1891
1892     atom_header = ngx_mp4_atom_header(mp4);
1893     ngx_mp4_set_atom_name(atom_header, 's', 't', 'b', 'l');
1894
1895     trak = ngx_mp4_last_trak(mp4);
1896
1897     atom = &trak->stbl_atom_buf;
1898     atom->temporary = 1;
1899     atom->pos = atom_header;
1900     atom->last = atom_header + sizeof(ngx_mp4_atom_header_t);
1901
1902     trak->out[NGX_HTTP_MP4_STBL_ATOM].buf = atom;
1903
1904     return ngx_http_mp4_read_atom(mp4, ngx_http_mp4_stbl_atoms, atom_data_size);
1905 }
1906
1907
1908 static void
1909 ngx_http_mp4_update_stbl_atom(ngx_http_mp4_file_t *mp4,
1910     ngx_http_mp4_trak_t *trak)
1911 {

```

```

1912     ngx_buf_t  *atom;
1913
1914     trak->size += sizeof(ngx_mp4_atom_header_t);
1915     atom = &trak->stbl_atom_buf;
1916     ngx_mp4_set_32value(atom->pos, trak->size);
1917 }
1918
1919
1920 typedef struct {
1921     u_char    size[4];
1922     u_char    name[4];
1923     u_char    version[1];
1924     u_char    flags[3];
1925     u_char    entries[4];
1926
1927     u_char    media_size[4];
1928     u_char    media_name[4];
1929 } ngx_mp4_stsd_atom_t;
1930
1931
1932 static ngx_int_t
1933 ngx_http_mp4_read_stsd_atom(ngx_http_mp4_file_t *mp4, uint64_t atom_data_size)
1934 {
1935     u_char          *atom_header, *atom_table;
1936     size_t          atom_size;
1937     ngx_buf_t       *atom;
1938     ngx_mp4_stsd_atom_t *stsd_atom;
1939     ngx_http_mp4_trak_t *trak;
1940
1941     /* sample description atom */
1942
1943     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0, "mp4 stsd atom");
1944
1945     atom_header = ngx_mp4_atom_header(mp4);
1946     stsd_atom = (ngx_mp4_stsd_atom_t *) atom_header;
1947     atom_size = sizeof(ngx_mp4_atom_header_t) + (size_t) atom_data_size;
1948     atom_table = atom_header + atom_size;
1949     ngx_mp4_set_32value(stsd_atom->size, atom_size);
1950     ngx_mp4_set_atom_name(stsd_atom, 's', 't', 's', 'd');
1951
1952     if (ngx_mp4_atom_data_size(ngx_mp4_stsd_atom_t) > atom_data_size) {
1953         ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
1954             "\\\"%s\\\" mp4 stsd atom too small", mp4->file.name.data);
1955         return NGX_ERROR;
1956     }
1957
1958     ngx_log_debug3(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
1959         "stsd entries:%uD, media:%*s",
1960         ngx_mp4_get_32value(stsd_atom->entries),
1961         4, stsd_atom->media_name);
1962
1963     trak = ngx_mp4_last_trak(mp4);
1964
1965     atom = &trak->stsd_atom_buf;
1966     atom->temporary = 1;
1967     atom->pos = atom_header;
1968     atom->last = atom_table;
1969
1970     trak->out[NGX_HTTP_MP4_STSD_ATOM].buf = atom;
1971     trak->size += atom_size;
1972
1973     ngx_mp4_atom_next(mp4, atom_data_size);
1974
1975     return NGX_OK;
1976 }
1977
1978
1979 typedef struct {
1980     u_char    size[4];
1981     u_char    name[4];
1982     u_char    version[1];
1983     u_char    flags[3];
1984     u_char    entries[4];
1985 } ngx_mp4_stts_atom_t;
1986
1987 typedef struct {

```

```

1988     u_char    count[4];
1989     u_char    duration[4];
1990 } ngx_mp4_stts_entry_t;
1991
1992
1993 static ngx_int_t
1994 ngx_http_mp4_read_stts_atom(ngx_http_mp4_file_t *mp4, uint64_t atom_data_size)
1995 {
1996     u_char          *atom_header, *atom_table, *atom_end;
1997     uint32_t        entries;
1998     ngx_buf_t       *atom, *data;
1999     ngx_mp4_stts_atom_t *stts_atom;
2000     ngx_http_mp4_trak_t *trak;
2001
2002     /* time-to-sample atom */
2003
2004     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0, "mp4 stts atom");
2005
2006     atom_header = ngx_mp4_atom_header(mp4);
2007     stts_atom = (ngx_mp4_stts_atom_t *) atom_header;
2008     ngx_mp4_set_atom_name(stts_atom, 's', 't', 't', 's');
2009
2010     if (ngx_mp4_atom_data_size(ngx_mp4_stts_atom_t) > atom_data_size) {
2011         ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
2012             "\\\"%s\\\" mp4 stts atom too small", mp4->file.name.data);
2013         return NGX_ERROR;
2014     }
2015
2016     entries = ngx_mp4_get_32value(stts_atom->entries);
2017
2018     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2019         "mp4 time-to-sample entries:%uD", entries);
2020
2021     if (ngx_mp4_atom_data_size(ngx_mp4_stts_atom_t)
2022         + entries * sizeof(ngx_mp4_stts_entry_t) > atom_data_size)
2023     {
2024         ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
2025             "\\\"%s\\\" mp4 stts atom too small", mp4->file.name.data);
2026         return NGX_ERROR;
2027     }
2028
2029     atom_table = atom_header + sizeof(ngx_mp4_stts_atom_t);
2030     atom_end = atom_table + entries * sizeof(ngx_mp4_stts_entry_t);
2031
2032     trak = ngx_mp4_last_trak(mp4);
2033     trak->time_to_sample_entries = entries;
2034
2035     atom = &trak->stts_atom_buf;
2036     atom->temporary = 1;
2037     atom->pos = atom_header;
2038     atom->last = atom_table;
2039
2040     data = &trak->stts_data_buf;
2041     data->temporary = 1;
2042     data->pos = atom_table;
2043     data->last = atom_end;
2044
2045     trak->out[NGX_HTTP_MP4_STTS_ATOM].buf = atom;
2046     trak->out[NGX_HTTP_MP4_STTS_DATA].buf = data;
2047
2048     ngx_mp4_atom_next(mp4, atom_data_size);
2049
2050     return NGX_OK;
2051 }
2052
2053
2054 static ngx_int_t
2055 ngx_http_mp4_update_stts_atom(ngx_http_mp4_file_t *mp4,
2056     ngx_http_mp4_trak_t *trak)
2057 {
2058     size_t          atom_size;
2059     ngx_buf_t       *atom, *data;
2060     ngx_mp4_stts_atom_t *stts_atom;
2061
2062     /*
2063      * mdia.minf.stbl.stts updating requires trak->timescale

```

```

2064     * from mdia.mdhd atom which may reside after mdia.minf
2065     */
2066
2067     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2068         "mp4 stts atom update");
2069
2070     data = trak->out[NGX_HTTP_MP4_STTS_DATA].buf;
2071
2072     if (data == NULL) {
2073         ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
2074             "no mp4 stts atoms were found in \"%s\"",
2075             mp4->file.name.data);
2076         return NGX_ERROR;
2077     }
2078
2079     if (ngx_http_mp4_crop_stts_data(mp4, trak, 1) != NGX_OK) {
2080         return NGX_ERROR;
2081     }
2082
2083     if (ngx_http_mp4_crop_stts_data(mp4, trak, 0) != NGX_OK) {
2084         return NGX_ERROR;
2085     }
2086
2087     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2088         "time-to-sample entries:%uD", trak->time_to_sample_entries);
2089
2090     atom_size = sizeof(ngx_mp4_stts_atom_t) + (data->last - data->pos);
2091     trak->size += atom_size;
2092
2093     atom = trak->out[NGX_HTTP_MP4_STTS_ATOM].buf;
2094     stts_atom = (ngx_mp4_stts_atom_t *) atom->pos;
2095     ngx_mp4_set_32value(stts_atom->size, atom_size);
2096     ngx_mp4_set_32value(stts_atom->entries, trak->time_to_sample_entries);
2097
2098     return NGX_OK;
2099 }
2100
2101
2102 static ngx_int_t
2103 ngx_http_mp4_crop_stts_data(ngx_http_mp4_file_t *mp4,
2104     ngx_http_mp4_trak_t *trak, ngx_uint_t start)
2105 {
2106     uint32_t          count, duration, rest;
2107     uint64_t          start_time;
2108     ngx_buf_t         *data;
2109     ngx_uint_t        start_sample, entries, start_sec;
2110     ngx_mp4_stts_entry_t *entry, *end;
2111
2112     if (start) {
2113         start_sec = mp4->start;
2114
2115         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2116             "mp4 stts crop start_time:%ui", start_sec);
2117     } else if (mp4->length) {
2118         start_sec = mp4->length;
2119
2120         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2121             "mp4 stts crop end_time:%ui", start_sec);
2122     } else {
2123         return NGX_OK;
2124     }
2125
2126     data = trak->out[NGX_HTTP_MP4_STTS_DATA].buf;
2127
2128     start_time = (uint64_t) start_sec * trak->timescale / 1000;
2129
2130     entries = trak->time_to_sample_entries;
2131     start_sample = 0;
2132     entry = (ngx_mp4_stts_entry_t *) data->pos;
2133     end = (ngx_mp4_stts_entry_t *) data->last;
2134
2135     while (entry < end) {
2136         count = ngx_mp4_get_32value(entry->count);
2137         duration = ngx_mp4_get_32value(entry->duration);

```



```

2140     ngx_log_debug3(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2141                 "time:%uL, count:%uD, duration:%uD",
2142                 start_time, count, duration);
2143
2144
2145     if (start_time < (uint64_t) count * duration) {
2146         start_sample += (ngx_uint_t) (start_time / duration);
2147         rest = (uint32_t) (start_time / duration);
2148         goto found;
2149     }
2150
2151     start_sample += count;
2152     start_time -= count * duration;
2153     entries--;
2154     entry++;
2155 }
2156
2157 if (start) {
2158     ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
2159                 "start time is out mp4 stts samples in \"%s\"",
2160                 mp4->file.name.data);
2161
2162     return NGX_ERROR;
2163 }
2164 else {
2165     trak->end_sample = trak->start_sample + start_sample;
2166
2167     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2168                 "end_sample:%ui", trak->end_sample);
2169
2170     return NGX_OK;
2171 }
2172
2173 found:
2174
2175     if (start) {
2176         ngx_mp4_set_32value(entry->count, count - rest);
2177         data->pos = (u_char *) entry;
2178         trak->time_to_sample_entries = entries;
2179         trak->start_sample = start_sample;
2180
2181         ngx_log_debug2(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2182                     "start_sample:%ui, new count:%uD",
2183                     trak->start_sample, count - rest);
2184     }
2185     else {
2186         ngx_mp4_set_32value(entry->count, rest);
2187         data->last = (u_char *) (entry + 1);
2188         trak->time_to_sample_entries -= entries - 1;
2189         trak->end_sample = trak->start_sample + start_sample;
2190
2191         ngx_log_debug2(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2192                     "end_sample:%ui, new count:%uD",
2193                     trak->end_sample, rest);
2194     }
2195
2196     return NGX_OK;
2197 }
2198
2199
2200 typedef struct {
2201     u_char    size[4];
2202     u_char    name[4];
2203     u_char    version[1];
2204     u_char    flags[3];
2205     u_char    entries[4];
2206 } ngx_http_mp4_stss_atom_t;
2207
2208
2209 static ngx_int_t
2210 ngx_http_mp4_read_stss_atom(ngx_http_mp4_file_t *mp4, uint64_t atom_data_size)
2211 {
2212     u_char          *atom_header, *atom_table, *atom_end;
2213     uint32_t        entries;
2214     ngx_buf_t       *atom, *data;
2215     ngx_http_mp4_trak_t *trak;

```

```

2216 ngx_http_mp4_stss_atom_t *stss_atom;
2217
2218 /* sync samples atom */
2219
2220 ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0, "mp4 stss atom");
2221
2222 atom_header = ngx_mp4_atom_header(mp4);
2223 stss_atom = (ngx_http_mp4_stss_atom_t *) atom_header;
2224 ngx_mp4_set_atom_name(stss_atom, 's', 't', 's', 's');
2225
2226 if (ngx_mp4_atom_data_size(ngx_http_mp4_stss_atom_t) > atom_data_size) {
2227     ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
2228         "\\\"%s\\\" mp4 stss atom too small", mp4->file.name.data);
2229     return NGX_ERROR;
2230 }
2231
2232 entries = ngx_mp4_get_32value(stss_atom->entries);
2233
2234 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2235     "sync sample entries:%uD", entries);
2236
2237 trak = ngx_mp4_last_trak(mp4);
2238 trak->sync_samples_entries = entries;
2239
2240 atom_table = atom_header + sizeof(ngx_http_mp4_stss_atom_t);
2241
2242 atom = &trak->stss_atom_buf;
2243 atom->temporary = 1;
2244 atom->pos = atom_header;
2245 atom->last = atom_table;
2246
2247 if (ngx_mp4_atom_data_size(ngx_http_mp4_stss_atom_t)
2248     + entries * sizeof(uint32_t) > atom_data_size)
2249 {
2250     ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
2251         "\\\"%s\\\" mp4 stss atom too small", mp4->file.name.data);
2252     return NGX_ERROR;
2253 }
2254
2255 atom_end = atom_table + entries * sizeof(uint32_t);
2256
2257 data = &trak->stss_data_buf;
2258 data->temporary = 1;
2259 data->pos = atom_table;
2260 data->last = atom_end;
2261
2262 trak->out[NGX_HTTP_MP4_STSS_ATOM].buf = atom;
2263 trak->out[NGX_HTTP_MP4_STSS_DATA].buf = data;
2264
2265 ngx_mp4_atom_next(mp4, atom_data_size);
2266
2267 return NGX_OK;
2268 }
2269
2270
2271 static ngx_int_t
2272 ngx_http_mp4_update_stss_atom(ngx_http_mp4_file_t *mp4,
2273     ngx_http_mp4_trak_t *trak)
2274 {
2275     size_t          atom_size;
2276     uint32_t        sample, start_sample, *entry, *end;
2277     ngx_buf_t       *atom, *data;
2278     ngx_http_mp4_stss_atom_t *stss_atom;
2279
2280     /*
2281      * mdia.minf.stbl.stss updating requires trak->start_sample
2282      * from mdia.minf.stbl.stts which depends on value from mdia.mdhd
2283      * atom which may reside after mdia.minf
2284      */
2285
2286     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2287         "mp4 stss atom update");
2288
2289     data = trak->out[NGX_HTTP_MP4_STSS_DATA].buf;
2290
2291     if (data == NULL) {

```

```

2292     return NGX_OK;
2293 }
2294
2295 ngx_http_mp4_crop_stss_data(mp4, trak, 1);
2296 ngx_http_mp4_crop_stss_data(mp4, trak, 0);
2297
2298 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2299     "sync sample entries:%uD", trak->sync_samples_entries);
2300
2301 if (trak->sync_samples_entries) {
2302     entry = (uint32_t *) data->pos;
2303     end = (uint32_t *) data->last;
2304
2305     start_sample = trak->start_sample;
2306
2307     while (entry < end) {
2308         sample = ngx_mp4_get_32value(entry);
2309         sample -= start_sample;
2310         ngx_mp4_set_32value(entry, sample);
2311         entry++;
2312     }
2313
2314 } else {
2315     trak->out[NGX_HTTP_MP4_STSS_DATA].buf = NULL;
2316 }
2317
2318 atom_size = sizeof(ngx_http_mp4_stss_atom_t) + (data->last - data->pos);
2319 trak->size += atom_size;
2320
2321 atom = trak->out[NGX_HTTP_MP4_STSS_ATOM].buf;
2322 stss_atom = (ngx_http_mp4_stss_atom_t *) atom->pos;
2323
2324 ngx_mp4_set_32value(stss_atom->size, atom_size);
2325 ngx_mp4_set_32value(stss_atom->entries, trak->sync_samples_entries);
2326
2327 return NGX_OK;
2328 }
2329
2330
2331 static void
2332 ngx_http_mp4_crop_stss_data(ngx_http_mp4_file_t *mp4,
2333 ngx_http_mp4_trak_t *trak, ngx_uint_t start)
2334 {
2335     uint32_t    sample, start_sample, *entry, *end;
2336     ngx_buf_t *data;
2337     ngx_uint_t entries;
2338
2339     /* sync samples starts from 1 */
2340
2341     if (start) {
2342         start_sample = trak->start_sample + 1;
2343
2344         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2345             "mp4 stss crop start_sample:%uD", start_sample);
2346     } else if (mp4->length) {
2347         start_sample = trak->end_sample + 1;
2348
2349         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2350             "mp4 stss crop end_sample:%uD", start_sample);
2351     } else {
2352         return;
2353     }
2354 }
2355
2356 data = trak->out[NGX_HTTP_MP4_STSS_DATA].buf;
2357
2358 entries = trak->sync_samples_entries;
2359 entry = (uint32_t *) data->pos;
2360 end = (uint32_t *) data->last;
2361
2362 while (entry < end) {
2363     sample = ngx_mp4_get_32value(entry);
2364
2365     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2366         "sync:%uD", sample);

```

```

2368     if (sample >= start_sample) {
2369         goto found;
2370     }
2371 }
2372
2373     entries--;
2374     entry++;
2375 }
2376
2377     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2378         "sample is out of mp4 stss atom");
2379
2380 found:
2381
2382     if (start) {
2383         data->pos = (u_char *) entry;
2384         trak->sync_samples_entries = entries;
2385     }
2386     else {
2387         data->last = (u_char *) entry;
2388         trak->sync_samples_entries -= entries;
2389     }
2390 }
2391
2392
2393 typedef struct {
2394     u_char    size[4];
2395     u_char    name[4];
2396     u_char    version[1];
2397     u_char    flags[3];
2398     u_char    entries[4];
2399 } ngx_mp4_ctts_atom_t;
2400
2401 typedef struct {
2402     u_char    count[4];
2403     u_char    offset[4];
2404 } ngx_mp4_ctts_entry_t;
2405
2406
2407 static ngx_int_t
2408 ngx_http_mp4_read_ctts_atom(ngx_http_mp4_file_t *mp4, uint64_t atom_data_size)
2409 {
2410     u_char          *atom_header, *atom_table, *atom_end;
2411     uint32_t        entries;
2412     ngx_buf_t       *atom, *data;
2413     ngx_mp4_ctts_atom_t *ctts_atom;
2414     ngx_http_mp4_trak_t *trak;
2415
2416     /* composition offsets atom */
2417
2418     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0, "mp4 ctts atom");
2419
2420     atom_header = ngx_mp4_atom_header(mp4);
2421     ctts_atom = (ngx_mp4_ctts_atom_t *) atom_header;
2422     ngx_mp4_set_atom_name(ctts_atom, 'c', 't', 't', 's');
2423
2424     if (ngx_mp4_atom_data_size(ngx_mp4_ctts_atom_t) > atom_data_size) {
2425         ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
2426             "\"%s\" mp4 ctts atom too small", mp4->file.name.data);
2427         return NGX_ERROR;
2428     }
2429
2430     entries = ngx_mp4_get_32value(ctts_atom->entries);
2431
2432     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2433         "composition offset entries:%uD", entries);
2434
2435     trak = ngx_mp4_last_trak(mp4);
2436     trak->composition_offset_entries = entries;
2437
2438     atom_table = atom_header + sizeof(ngx_mp4_ctts_atom_t);
2439
2440     atom = &trak->ctts_atom_buf;
2441     atom->temporary = 1;
2442     atom->pos = atom_header;
2443     atom->last = atom_table;

```

```

2444     if (ngx_mp4_atom_data_size(ngx_mp4_ctts_atom_t)
2445         + entries * sizeof(ngx_mp4_ctts_entry_t) > atom_data_size)
2446     {
2447         ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
2448             "\"%s\" mp4 ctts atom too small", mp4->file.name.data);
2449         return NGX_ERROR;
2450     }
2451 }
2452
2453 atom_end = atom_table + entries * sizeof(ngx_mp4_ctts_entry_t);
2454
2455 data = &trak->ctts_data_buf;
2456 data->temporary = 1;
2457 data->pos = atom_table;
2458 data->last = atom_end;
2459
2460 trak->out[NGX_HTTP_MP4_CTTS_ATOM].buf = atom;
2461 trak->out[NGX_HTTP_MP4_CTTS_DATA].buf = data;
2462
2463 ngx_mp4_atom_next(mp4, atom_data_size);
2464
2465 return NGX_OK;
2466 }
2467
2468
2469 static void
2470 ngx_http_mp4_update_ctts_atom(ngx_http_mp4_file_t *mp4,
2471     ngx_http_mp4_trak_t *trak)
2472 {
2473     size_t          atom_size;
2474     ngx_buf_t      *atom, *data;
2475     ngx_mp4_ctts_atom_t *ctts_atom;
2476
2477     /*
2478      * mdia.minf.stbl.ctts updating requires trak->start_sample
2479      * from mdia.minf.stbl.stts which depends on value from mdia.mdhd
2480      * atom which may reside after mdia.minf
2481      */
2482
2483     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2484         "mp4 ctts atom update");
2485
2486     data = trak->out[NGX_HTTP_MP4_CTTS_DATA].buf;
2487
2488     if (data == NULL) {
2489         return;
2490     }
2491
2492     ngx_http_mp4_crop_ctts_data(mp4, trak, 1);
2493     ngx_http_mp4_crop_ctts_data(mp4, trak, 0);
2494
2495     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2496         "composition offset entries:%uD",
2497         trak->composition_offset_entries);
2498
2499     if (trak->composition_offset_entries == 0) {
2500         trak->out[NGX_HTTP_MP4_CTTS_ATOM].buf = NULL;
2501         trak->out[NGX_HTTP_MP4_CTTS_DATA].buf = NULL;
2502         return;
2503     }
2504
2505     atom_size = sizeof(ngx_mp4_ctts_atom_t) + (data->last - data->pos);
2506     trak->size += atom_size;
2507
2508     atom = trak->out[NGX_HTTP_MP4_CTTS_ATOM].buf;
2509     ctts_atom = (ngx_mp4_ctts_atom_t *) atom->pos;
2510
2511     ngx_mp4_set_32value(ctts_atom->size, atom_size);
2512     ngx_mp4_set_32value(ctts_atom->entries, trak->composition_offset_entries);
2513
2514     return;
2515 }
2516
2517
2518 static void
2519 ngx_http_mp4_crop_ctts_data(ngx_http_mp4_file_t *mp4,

```

```

2520     ngx_http_mp4_trak_t *trak, ngx_uint_t start)
2521 {
2522     uint32_t          count, start_sample, rest;
2523     ngx_buf_t        *data;
2524     ngx_uint_t       entries;
2525     ngx_mp4_ctts_entry_t *entry, *end;
2526
2527     /* sync samples starts from 1 */
2528
2529     if (start) {
2530         start_sample = trak->start_sample + 1;
2531
2532         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2533             "mp4 ctts crop start_sample:%uD", start_sample);
2534
2535     } else if (mp4->length) {
2536         start_sample = trak->end_sample - trak->start_sample + 1;
2537
2538         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2539             "mp4 ctts crop end_sample:%uD", start_sample);
2540
2541     } else {
2542         return;
2543     }
2544
2545     data = trak->out[NGX_HTTP_MP4_CTTS_DATA].buf;
2546
2547     entries = trak->composition_offset_entries;
2548     entry = (ngx_mp4_ctts_entry_t *) data->pos;
2549     end = (ngx_mp4_ctts_entry_t *) data->last;
2550
2551     while (entry < end) {
2552         count = ngx_mp4_get_32value(entry->count);
2553
2554         ngx_log_debug3(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2555             "sample:%uD, count:%uD, offset:%uD",
2556             start_sample, count, ngx_mp4_get_32value(entry->offset));
2557
2558         if (start_sample <= count) {
2559             rest = start_sample - 1;
2560             goto found;
2561         }
2562
2563         start_sample -= count;
2564         entries--;
2565         entry++;
2566     }
2567
2568     if (start) {
2569         data->pos = (u_char *) end;
2570         trak->composition_offset_entries = 0;
2571     }
2572
2573     return;
2574
2575 found:
2576
2577     if (start) {
2578         ngx_mp4_set_32value(entry->count, count - rest);
2579         data->pos = (u_char *) entry;
2580         trak->composition_offset_entries = entries;
2581     } else {
2582         ngx_mp4_set_32value(entry->count, rest);
2583         data->last = (u_char *) (entry + 1);
2584         trak->composition_offset_entries -= entries - 1;
2585     }
2586 }
2587
2588
2589
2590 typedef struct {
2591     u_char    size[4];
2592     u_char    name[4];
2593     u_char    version[1];
2594     u_char    flags[3];
2595     u_char    entries[4];

```

```

2596 } ngx_mp4_stsc_atom_t;
2597
2598
2599 static ngx_int_t
2600 ngx_http_mp4_read_stsc_atom(ngx_http_mp4_file_t *mp4, uint64_t atom_data_size)
2601 {
2602     u_char          *atom_header, *atom_table, *atom_end;
2603     uint32_t        entries;
2604     ngx_buf_t       *atom, *data;
2605     ngx_mp4_stsc_atom_t *stsc_atom;
2606     ngx_http_mp4_trak_t *trak;
2607
2608     /* sample-to-chunk atom */
2609
2610     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0, "mp4 stsc atom");
2611
2612     atom_header = ngx_mp4_atom_header(mp4);
2613     stsc_atom = (ngx_mp4_stsc_atom_t *) atom_header;
2614     ngx_mp4_set_atom_name(stsc_atom, 's', 't', 's', 'c');
2615
2616     if (ngx_mp4_atom_data_size(ngx_mp4_stsc_atom_t) > atom_data_size) {
2617         ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
2618             "\\\"%s\\\" mp4 stsc atom too small", mp4->file.name.data);
2619         return NGX_ERROR;
2620     }
2621
2622     entries = ngx_mp4_get_32value(stsc_atom->entries);
2623
2624     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2625         "sample-to-chunk entries:%uD", entries);
2626
2627     if (ngx_mp4_atom_data_size(ngx_mp4_stsc_atom_t)
2628         + entries * sizeof(ngx_mp4_stsc_entry_t) > atom_data_size)
2629     {
2630         ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
2631             "\\\"%s\\\" mp4 stsc atom too small", mp4->file.name.data);
2632         return NGX_ERROR;
2633     }
2634
2635     atom_table = atom_header + sizeof(ngx_mp4_stsc_atom_t);
2636     atom_end = atom_table + entries * sizeof(ngx_mp4_stsc_entry_t);
2637
2638     trak = ngx_mp4_last_trak(mp4);
2639     trak->sample_to_chunk_entries = entries;
2640
2641     atom = &trak->stsc_atom_buf;
2642     atom->temporary = 1;
2643     atom->pos = atom_header;
2644     atom->last = atom_table;
2645
2646     data = &trak->stsc_data_buf;
2647     data->temporary = 1;
2648     data->pos = atom_table;
2649     data->last = atom_end;
2650
2651     trak->out[NGX_HTTP_MP4_STSC_ATOM].buf = atom;
2652     trak->out[NGX_HTTP_MP4_STSC_DATA].buf = data;
2653
2654     ngx_mp4_atom_next(mp4, atom_data_size);
2655
2656     return NGX_OK;
2657 }
2658
2659
2660 static ngx_int_t
2661 ngx_http_mp4_update_stsc_atom(ngx_http_mp4_file_t *mp4,
2662     ngx_http_mp4_trak_t *trak)
2663 {
2664     size_t          atom_size;
2665     uint32_t        chunk;
2666     ngx_buf_t       *atom, *data;
2667     ngx_mp4_stsc_atom_t *stsc_atom;
2668     ngx_mp4_stsc_entry_t *entry, *end;
2669
2670     /*
2671     * mdia.minf.stbl.stsc updating requires trak->start_sample

```

```

2672     * from mdia.minf.stbl.stts which depends on value from mdia.mdhd
2673     * atom which may reside after mdia.minf
2674     */
2675
2676     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2677         "mp4 stsc atom update");
2678
2679     data = trak->out[NGX_HTTP_MP4_STSC_DATA].buf;
2680
2681     if (data == NULL) {
2682         ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
2683             "no mp4 stsc atoms were found in \"%s\"",
2684             mp4->file.name.data);
2685         return NGX_ERROR;
2686     }
2687
2688     if (trak->sample_to_chunk_entries == 0) {
2689         ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
2690             "zero number of entries in stsc atom in \"%s\"",
2691             mp4->file.name.data);
2692         return NGX_ERROR;
2693     }
2694
2695     if (ngx_http_mp4_crop_stsc_data(mp4, trak, 1) != NGX_OK) {
2696         return NGX_ERROR;
2697     }
2698
2699     if (ngx_http_mp4_crop_stsc_data(mp4, trak, 0) != NGX_OK) {
2700         return NGX_ERROR;
2701     }
2702
2703     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2704         "sample-to-chunk entries:%uD",
2705         trak->sample_to_chunk_entries);
2706
2707     entry = (ngx_mp4_stsc_entry_t *) data->pos;
2708     end = (ngx_mp4_stsc_entry_t *) data->last;
2709
2710     while (entry < end) {
2711         chunk = ngx_mp4_get_32value(entry->chunk);
2712         chunk -= trak->start_chunk;
2713         ngx_mp4_set_32value(entry->chunk, chunk);
2714         entry++;
2715     }
2716
2717     atom_size = sizeof(ngx_mp4_stsc_atom_t)
2718         + trak->sample_to_chunk_entries * sizeof(ngx_mp4_stsc_entry_t);
2719
2720     trak->size += atom_size;
2721
2722     atom = trak->out[NGX_HTTP_MP4_STSC_ATOM].buf;
2723     stsc_atom = (ngx_mp4_stsc_atom_t *) atom->pos;
2724
2725     ngx_mp4_set_32value(stsc_atom->size, atom_size);
2726     ngx_mp4_set_32value(stsc_atom->entries, trak->sample_to_chunk_entries);
2727
2728     return NGX_OK;
2729 }
2730
2731
2732 static ngx_int_t
2733 ngx_http_mp4_crop_stsc_data(ngx_http_mp4_file_t *mp4,
2734     ngx_http_mp4_trak_t *trak, ngx_uint_t start)
2735 {
2736     uint32_t          start_sample, chunk, samples, id, next_chunk, n,
2737                     prev_samples;
2738     ngx_buf_t        *data, *buf;
2739     ngx_uint_t        entries, target_chunk, chunk_samples;
2740     ngx_mp4_stsc_entry_t *entry, *end, *first;
2741
2742     entries = trak->sample_to_chunk_entries - 1;
2743
2744     if (start) {
2745         start_sample = (uint32_t) trak->start_sample;
2746
2747         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,

```



```

2748         "mp4 stsc crop start_sample:%uD", start_sample);
2749
2750 } else if (mp4->length) {
2751     start_sample = (uint32_t) (trak->end_sample - trak->start_sample);
2752     samples = 0;
2753
2754     data = trak->out[NGX_HTTP_MP4_STSC_START].buf;
2755
2756     if (data) {
2757         entry = (ngx_mp4_stsc_entry_t *) data->pos;
2758         samples = ngx_mp4_get_32value(entry->samples);
2759         entries--;
2760
2761         if (samples > start_sample) {
2762             samples = start_sample;
2763             ngx_mp4_set_32value(entry->samples, samples);
2764         }
2765
2766         start_sample -= samples;
2767     }
2768
2769     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2770                 "mp4 stsc crop end_sample:%uD, ext_samples:%uD",
2771                 start_sample, samples);
2772
2773 } else {
2774     return NGX_OK;
2775 }
2776
2777 data = trak->out[NGX_HTTP_MP4_STSC_DATA].buf;
2778
2779 entry = (ngx_mp4_stsc_entry_t *) data->pos;
2780 end = (ngx_mp4_stsc_entry_t *) data->last;
2781
2782 chunk = ngx_mp4_get_32value(entry->chunk);
2783 samples = ngx_mp4_get_32value(entry->samples);
2784 id = ngx_mp4_get_32value(entry->id);
2785 prev_samples = 0;
2786 entry++;
2787
2788 while (entry < end) {
2789
2790     next_chunk = ngx_mp4_get_32value(entry->chunk);
2791
2792     ngx_log_debug5(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2793                 "sample:%uD, chunk:%uD, chunks:%uD, "
2794                 "samples:%uD, id:%uD",
2795                 start_sample, chunk, next_chunk - chunk, samples, id);
2796
2797     n = (next_chunk - chunk) * samples;
2798
2799     if (start_sample < n) {
2800         goto found;
2801     }
2802
2803     start_sample -= n;
2804
2805     prev_samples = samples;
2806     chunk = next_chunk;
2807     samples = ngx_mp4_get_32value(entry->samples);
2808     id = ngx_mp4_get_32value(entry->id);
2809     entries--;
2810     entry++;
2811 }
2812
2813 next_chunk = trak->chunks + 1;
2814
2815 ngx_log_debug4(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
2816             "sample:%uD, chunk:%uD, chunks:%uD, samples:%uD",
2817             start_sample, chunk, next_chunk - chunk, samples);
2818
2819 n = (next_chunk - chunk) * samples;
2820
2821 if (start_sample > n) {
2822     ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
2823                 "%s time is out mp4 stsc chunks in \"%s\"",

```

```

2824         start ? "start" : "end", mp4->file.name.data);
2825     return NGX\_ERROR;
2826 }
2827
2828 found:
2829
2830     entries++;
2831     entry--;
2832
2833     if (samples == 0) {
2834         ngx\_log\_error(NGX\_LOG\_ERR, mp4->file.log, 0,
2835             "zero number of samples in \"%s\"",
2836             mp4->file.name.data);
2837         return NGX\_ERROR;
2838     }
2839
2840     target_chunk = chunk - 1;
2841     target_chunk += start_sample / samples;
2842     chunk_samples = start_sample % samples;
2843
2844     if (start) {
2845         data->pos = (u_char *) entry;
2846
2847         trak->sample_to_chunk_entries = entries;
2848         trak->start_chunk = target_chunk;
2849         trak->start_chunk_samples = chunk_samples;
2850
2851         ngx\_mp4\_set\_32value(entry->chunk, trak->start_chunk + 1);
2852
2853         samples -= chunk_samples;
2854
2855         ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, mp4->file.log, 0,
2856             "start_chunk:%ui, start_chunk_samples:%ui",
2857             trak->start_chunk, trak->start_chunk_samples);
2858
2859     } else {
2860         if (start_sample) {
2861             data->last = (u_char *) (entry + 1);
2862             trak->sample_to_chunk_entries -= entries - 1;
2863             trak->end_chunk_samples = samples;
2864
2865         } else {
2866             data->last = (u_char *) entry;
2867             trak->sample_to_chunk_entries -= entries;
2868             trak->end_chunk_samples = prev_samples;
2869         }
2870
2871         if (chunk_samples) {
2872             trak->end_chunk = target_chunk + 1;
2873             trak->end_chunk_samples = chunk_samples;
2874
2875         } else {
2876             trak->end_chunk = target_chunk;
2877         }
2878
2879         samples = chunk_samples;
2880         next_chunk = chunk + 1;
2881
2882         ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, mp4->file.log, 0,
2883             "end_chunk:%ui, end_chunk_samples:%ui",
2884             trak->end_chunk, trak->end_chunk_samples);
2885     }
2886
2887     if (chunk_samples && next_chunk - target_chunk == 2) {
2888
2889         ngx\_mp4\_set\_32value(entry->samples, samples);
2890
2891     } else if (chunk_samples && start) {
2892
2893         first = &trak->stsc_start_chunk_entry;
2894         ngx\_mp4\_set\_32value(first->chunk, 1);
2895         ngx\_mp4\_set\_32value(first->samples, samples);
2896         ngx\_mp4\_set\_32value(first->id, id);
2897
2898         buf = &trak->stsc_start_chunk_buf;
2899         buf->temporary = 1;

```

```

2900     buf->pos = (u_char *) first;
2901     buf->last = (u_char *) first + sizeof(ngx\_mp4\_stsc\_entry\_t);
2902
2903     trak->out[NGX\_HTTP\_MP4\_STSC\_START].buf = buf;
2904
2905     ngx\_mp4\_set\_32value(entry->chunk, trak->start_chunk + 2);
2906
2907     trak->sample_to_chunk_entries++;
2908
2909 } else if (chunk_samples) {
2910
2911     first = &trak->stsc_end_chunk_entry;
2912     ngx\_mp4\_set\_32value(first->chunk, trak->end_chunk - trak->start_chunk);
2913     ngx\_mp4\_set\_32value(first->samples, samples);
2914     ngx\_mp4\_set\_32value(first->id, id);
2915
2916     buf = &trak->stsc_end_chunk_buf;
2917     buf->temporary = 1;
2918     buf->pos = (u_char *) first;
2919     buf->last = (u_char *) first + sizeof(ngx\_mp4\_stsc\_entry\_t);
2920
2921     trak->out[NGX\_HTTP\_MP4\_STSC\_END].buf = buf;
2922
2923     trak->sample_to_chunk_entries++;
2924 }
2925
2926 return NGX\_OK;
2927 }
2928
2929
2930 typedef struct {
2931     u_char    size[4];
2932     u_char    name[4];
2933     u_char    version[1];
2934     u_char    flags[3];
2935     u_char    uniform_size[4];
2936     u_char    entries[4];
2937 } ngx\_mp4\_stsz\_atom\_t;
2938
2939
2940 static ngx\_int\_t
2941 ngx\_http\_mp4\_read\_stsz\_atom(ngx\_http\_mp4\_file\_t *mp4, uint64\_t atom_data_size)
2942 {
2943     u_char          *atom_header, *atom_table, *atom_end;
2944     size\_t          atom_size;
2945     uint32\_t        entries, size;
2946     ngx\_buf\_t      *atom, *data;
2947     ngx\_mp4\_stsz\_atom\_t *stsz_atom;
2948     ngx\_http\_mp4\_trak\_t *trak;
2949
2950     /* sample sizes atom */
2951
2952     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, mp4->file.log, 0, "mp4 stsz atom");
2953
2954     atom_header = ngx\_mp4\_atom\_header(mp4);
2955     stsz_atom = (ngx\_mp4\_stsz\_atom\_t *) atom_header;
2956     ngx\_mp4\_set\_atom\_name(stsz_atom, 's', 't', 's', 'z');
2957
2958     if (ngx\_mp4\_atom\_data\_size(ngx\_mp4\_stsz\_atom\_t) > atom_data_size) {
2959         ngx\_log\_error(NGX\_LOG\_ERR, mp4->file.log, 0,
2960             "\"%s\" mp4 stsz atom too small", mp4->file.name.data);
2961         return NGX\_ERROR;
2962     }
2963
2964     size = ngx\_mp4\_get\_32value(stsz_atom->uniform_size);
2965     entries = ngx\_mp4\_get\_32value(stsz_atom->entries);
2966
2967     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, mp4->file.log, 0,
2968         "sample uniform size:%uD, entries:%uD", size, entries);
2969
2970     trak = ngx\_mp4\_last\_trak(mp4);
2971     trak->sample_sizes_entries = entries;
2972
2973     atom_table = atom_header + sizeof(ngx\_mp4\_stsz\_atom\_t);
2974
2975     atom = &trak->stsz_atom_buf;

```

```

2976 atom->temporary = 1;
2977 atom->pos = atom_header;
2978 atom->last = atom_table;
2979
2980 trak->out[NGX_HTTP_MP4_STSZ_ATOM].buf = atom;
2981
2982 if (size == 0) {
2983     if (ngx_mp4_atom_data_size(ngx_mp4_stsz_atom_t)
2984         + entries * sizeof(uint32_t) > atom_data_size)
2985     {
2986         ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
2987             "\\%s\\" mp4_stsz_atom too small",
2988             mp4->file.name.data);
2989         return NGX_ERROR;
2990     }
2991
2992     atom_end = atom_table + entries * sizeof(uint32_t);
2993
2994     data = &trak->stsz_data_buf;
2995     data->temporary = 1;
2996     data->pos = atom_table;
2997     data->last = atom_end;
2998
2999     trak->out[NGX_HTTP_MP4_STSZ_DATA].buf = data;
3000
3001 } else {
3002     /* if size != 0 then all samples are the same size */
3003     /* TODO : chunk samples */
3004     atom_size = sizeof(ngx_mp4_atom_header_t) + (size_t) atom_data_size;
3005     ngx_mp4_set_32value(atom_header, atom_size);
3006     trak->size += atom_size;
3007 }
3008
3009 ngx_mp4_atom_next(mp4, atom_data_size);
3010
3011 return NGX_OK;
3012 }
3013
3014
3015 static ngx_int_t
3016 ngx_http_mp4_update_stsz_atom(ngx_http_mp4_file_t *mp4,
3017     ngx_http_mp4_trak_t *trak)
3018 {
3019     size_t          atom_size;
3020     uint32_t        *pos, *end, entries;
3021     ngx_buf_t        *atom, *data;
3022     ngx_mp4_stsz_atom_t *stsz_atom;
3023
3024     /*
3025      * mdia.minf.stbl.stsz updating requires trak->start_sample
3026      * from mdia.minf.stbl.stts which depends on value from mdia.mdhd
3027      * atom which may reside after mdia.minf
3028      */
3029
3030     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
3031         "mp4 stsz atom update");
3032
3033     data = trak->out[NGX_HTTP_MP4_STSZ_DATA].buf;
3034
3035     if (data) {
3036         entries = trak->sample_sizes_entries;
3037
3038         if (trak->start_sample > entries) {
3039             ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
3040                 "start time is out mp4 stsz samples in \\%s\\",
3041                 mp4->file.name.data);
3042             return NGX_ERROR;
3043         }
3044
3045         entries -= trak->start_sample;
3046         data->pos += trak->start_sample * sizeof(uint32_t);
3047         end = (uint32_t *) data->pos;
3048
3049         for (pos = end - trak->start_chunk_samples; pos < end; pos++) {
3050             trak->start_chunk_samples_size += ngx_mp4_get_32value(pos);
3051         }

```

```

3052     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
3053     "chunk samples sizes:%uL",
3054     trak->start_chunk_samples_size);
3055
3056
3057     if (mp4->length) {
3058         if (trak->end_sample - trak->start_sample > entries) {
3059             ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
3060             "end time is out mp4 stsz samples in \"%s\"",
3061             mp4->file.name.data);
3062             return NGX_ERROR;
3063         }
3064
3065         entries = trak->end_sample - trak->start_sample;
3066         data->last = data->pos + entries * sizeof(uint32_t);
3067         end = (uint32_t *) data->last;
3068
3069         for (pos = end - trak->end_chunk_samples; pos < end; pos++) {
3070             trak->end_chunk_samples_size += ngx_mp4_get_32value(pos);
3071         }
3072
3073         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
3074         "mp4 stsz end_chunk_samples_size:%uL",
3075         trak->end_chunk_samples_size);
3076     }
3077
3078     atom_size = sizeof(ngx_mp4_stsz_atom_t) + (data->last - data->pos);
3079     trak->size += atom_size;
3080
3081     atom = trak->out[NGX_HTTP_MP4_STSZ_ATOM].buf;
3082     stsz_atom = (ngx_mp4_stsz_atom_t *) atom->pos;
3083
3084     ngx_mp4_set_32value(stsz_atom->size, atom_size);
3085     ngx_mp4_set_32value(stsz_atom->entries, entries);
3086 }
3087
3088 return NGX_OK;
3089 }
3090
3091
3092 typedef struct {
3093     u_char    size[4];
3094     u_char    name[4];
3095     u_char    version[1];
3096     u_char    flags[3];
3097     u_char    entries[4];
3098 } ngx_mp4_stco_atom_t;
3099
3100
3101 static ngx_int_t
3102 ngx_http_mp4_read_stco_atom(ngx_http_mp4_file_t *mp4, uint64_t atom_data_size)
3103 {
3104     u_char          *atom_header, *atom_table, *atom_end;
3105     uint32_t        entries;
3106     ngx_buf_t       *atom, *data;
3107     ngx_mp4_stco_atom_t *stco_atom;
3108     ngx_http_mp4_trak_t *trak;
3109
3110     /* chunk offsets atom */
3111
3112     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0, "mp4 stco atom");
3113
3114     atom_header = ngx_mp4_atom_header(mp4);
3115     stco_atom = (ngx_mp4_stco_atom_t *) atom_header;
3116     ngx_mp4_set_atom_name(stco_atom, 's', 't', 'c', 'o');
3117
3118     if (ngx_mp4_atom_data_size(ngx_mp4_stco_atom_t) > atom_data_size) {
3119         ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
3120         "\"%s\" mp4 stco atom too small", mp4->file.name.data);
3121         return NGX_ERROR;
3122     }
3123
3124     entries = ngx_mp4_get_32value(stco_atom->entries);
3125
3126     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0, "chunks:%uD", entries);
3127

```

```

3128     if (ngx_mp4_atom_data_size(ngx_mp4_stco_atom_t)
3129         + entries * sizeof(uint32_t) > atom_data_size)
3130     {
3131         ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
3132             "\\\"%s\\\" mp4 stco atom too small", mp4->file.name.data);
3133         return NGX_ERROR;
3134     }
3135
3136     atom_table = atom_header + sizeof(ngx_mp4_stco_atom_t);
3137     atom_end = atom_table + entries * sizeof(uint32_t);
3138
3139     trak = ngx_mp4_last_trak(mp4);
3140     trak->chunks = entries;
3141
3142     atom = &trak->stco_atom_buf;
3143     atom->temporary = 1;
3144     atom->pos = atom_header;
3145     atom->last = atom_table;
3146
3147     data = &trak->stco_data_buf;
3148     data->temporary = 1;
3149     data->pos = atom_table;
3150     data->last = atom_end;
3151
3152     trak->out[NGX_HTTP_MP4_STCO_ATOM].buf = atom;
3153     trak->out[NGX_HTTP_MP4_STCO_DATA].buf = data;
3154
3155     ngx_mp4_atom_next(mp4, atom_data_size);
3156
3157     return NGX_OK;
3158 }
3159
3160
3161 static ngx_int_t
3162 ngx_http_mp4_update_stco_atom(ngx_http_mp4_file_t *mp4,
3163     ngx_http_mp4_trak_t *trak)
3164 {
3165     size_t          atom_size;
3166     uint32_t        entries;
3167     ngx_buf_t       *atom, *data;
3168     ngx_mp4_stco_atom_t *stco_atom;
3169
3170     /*
3171      * mdia.minf.stbl.stco updating requires trak->start_chunk
3172      * from mdia.minf.stbl.stsc which depends on value from mdia.mhd
3173      * atom which may reside after mdia.minf
3174      */
3175
3176     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
3177         "mp4 stco atom update");
3178
3179     data = trak->out[NGX_HTTP_MP4_STCO_DATA].buf;
3180
3181     if (data == NULL) {
3182         ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
3183             "no mp4 stco atoms were found in \\\"%s\\\" ",
3184             mp4->file.name.data);
3185         return NGX_ERROR;
3186     }
3187
3188     if (trak->start_chunk > trak->chunks) {
3189         ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
3190             "start time is out mp4 stco chunks in \\\"%s\\\" ",
3191             mp4->file.name.data);
3192         return NGX_ERROR;
3193     }
3194
3195     data->pos += trak->start_chunk * sizeof(uint32_t);
3196
3197     trak->start_offset = ngx_mp4_get_32value(data->pos);
3198     trak->start_offset += trak->start_chunk_samples_size;
3199     ngx_mp4_set_32value(data->pos, trak->start_offset);
3200
3201     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
3202         "start chunk offset:%0", trak->start_offset);
3203

```

```

3204     if (mp4->length) {
3205
3206         if (trak->end_chunk > trak->chunks) {
3207             ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
3208                 "end time is out mp4 stco chunks in \"%s\"",
3209                 mp4->file.name.data);
3210             return NGX_ERROR;
3211         }
3212
3213         entries = trak->end_chunk - trak->start_chunk;
3214         data->last = data->pos + entries * sizeof(uint32_t);
3215
3216         if (entries) {
3217             trak->end_offset =
3218                 ngx_mp4_get_32value(data->last - sizeof(uint32_t));
3219             trak->end_offset += trak->end_chunk_samples_size;
3220
3221             ngx_log_debug1(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
3222                 "end chunk offset:%0", trak->end_offset);
3223         }
3224
3225     } else {
3226         entries = trak->chunks - trak->start_chunk;
3227         trak->end_offset = mp4->mdat_data.buf->file_last;
3228     }
3229
3230     if (entries == 0) {
3231         trak->start_offset = mp4->end;
3232         trak->end_offset = 0;
3233     }
3234
3235     atom_size = sizeof(ngx_mp4_stco_atom_t) + (data->last - data->pos);
3236     trak->size += atom_size;
3237
3238     atom = trak->out[NGX_HTTP_MP4_STCO_ATOM].buf;
3239     stco_atom = (ngx_mp4_stco_atom_t *) atom->pos;
3240
3241     ngx_mp4_set_32value(stco_atom->size, atom_size);
3242     ngx_mp4_set_32value(stco_atom->entries, entries);
3243
3244     return NGX_OK;
3245 }
3246
3247
3248 static void
3249 ngx_http_mp4_adjust_stco_atom(ngx_http_mp4_file_t *mp4,
3250     ngx_http_mp4_trak_t *trak, int32_t adjustment)
3251 {
3252     uint32_t    offset, *entry, *end;
3253     ngx_buf_t  *data;
3254
3255     /*
3256      * moov.trak.mdia.minf.stbl.stco adjustment requires
3257      * minimal start offset of all traks and new moov atom size
3258      */
3259
3260     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
3261         "mp4 stco atom adjustment");
3262
3263     data = trak->out[NGX_HTTP_MP4_STCO_DATA].buf;
3264     entry = (uint32_t *) data->pos;
3265     end = (uint32_t *) data->last;
3266
3267     while (entry < end) {
3268         offset = ngx_mp4_get_32value(entry);
3269         offset += adjustment;
3270         ngx_mp4_set_32value(entry, offset);
3271         entry++;
3272     }
3273 }
3274
3275
3276 typedef struct {
3277     u_char    size[4];
3278     u_char    name[4];
3279     u_char    version[1];

```

```

3280     u_char    flags[3];
3281     u_char    entries[4];
3282 } ngx_mp4_co64_atom_t;
3283
3284
3285 static ngx_int_t
3286 ngx_http_mp4_read_co64_atom(ngx_http_mp4_file_t *mp4, uint64_t atom_data_size)
3287 {
3288     u_char          *atom_header, *atom_table, *atom_end;
3289     uint32_t        entries;
3290     ngx_buf_t       *atom, *data;
3291     ngx_mp4_co64_atom_t *co64_atom;
3292     ngx_http_mp4_trak_t *trak;
3293
3294     /* chunk offsets atom */
3295
3296     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0, "mp4 co64 atom");
3297
3298     atom_header = ngx_mp4_atom_header(mp4);
3299     co64_atom = (ngx_mp4_co64_atom_t *) atom_header;
3300     ngx_mp4_set_atom_name(co64_atom, 'c', 'o', '6', '4');
3301
3302     if (ngx_mp4_atom_data_size(ngx_mp4_co64_atom_t) > atom_data_size) {
3303         ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
3304             "\"%s\" mp4 co64 atom too small", mp4->file.name.data);
3305         return NGX_ERROR;
3306     }
3307
3308     entries = ngx_mp4_get_32value(co64_atom->entries);
3309
3310     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0, "chunks:%uD", entries);
3311
3312     if (ngx_mp4_atom_data_size(ngx_mp4_co64_atom_t)
3313         + entries * sizeof(uint64_t) > atom_data_size)
3314     {
3315         ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
3316             "\"%s\" mp4 co64 atom too small", mp4->file.name.data);
3317         return NGX_ERROR;
3318     }
3319
3320     atom_table = atom_header + sizeof(ngx_mp4_co64_atom_t);
3321     atom_end = atom_table + entries * sizeof(uint64_t);
3322
3323     trak = ngx_mp4_last_trak(mp4);
3324     trak->chunks = entries;
3325
3326     atom = &trak->co64_atom_buf;
3327     atom->temporary = 1;
3328     atom->pos = atom_header;
3329     atom->last = atom_table;
3330
3331     data = &trak->co64_data_buf;
3332     data->temporary = 1;
3333     data->pos = atom_table;
3334     data->last = atom_end;
3335
3336     trak->out[NGX_HTTP_MP4_CO64_ATOM].buf = atom;
3337     trak->out[NGX_HTTP_MP4_CO64_DATA].buf = data;
3338
3339     ngx_mp4_atom_next(mp4, atom_data_size);
3340
3341     return NGX_OK;
3342 }
3343
3344
3345 static ngx_int_t
3346 ngx_http_mp4_update_co64_atom(ngx_http_mp4_file_t *mp4,
3347     ngx_http_mp4_trak_t *trak)
3348 {
3349     size_t          atom_size;
3350     uint64_t        entries;
3351     ngx_buf_t       *atom, *data;
3352     ngx_mp4_co64_atom_t *co64_atom;
3353
3354     /*
3355      * mdia.minf.stbl.co64 updating requires trak->start_chunk

```



```

3356     * from mdia.minf.stbl.stsc which depends on value from mdia.mdhd
3357     * atom which may reside after mdia.minf
3358     */
3359
3360     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
3361                 "mp4 co64 atom update");
3362
3363     data = trak->out[NGX_HTTP_MP4_CO64_DATA].buf;
3364
3365     if (data == NULL) {
3366         ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
3367                     "no mp4 co64 atoms were found in \"%s\"",
3368                     mp4->file.name.data);
3369         return NGX_ERROR;
3370     }
3371
3372     if (trak->start_chunk > trak->chunks) {
3373         ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
3374                     "start time is out mp4 co64 chunks in \"%s\"",
3375                     mp4->file.name.data);
3376         return NGX_ERROR;
3377     }
3378
3379     data->pos += trak->start_chunk * sizeof(uint64_t);
3380
3381     trak->start_offset = ngx_mp4_get_64value(data->pos);
3382     trak->start_offset += trak->start_chunk_samples_size;
3383     ngx_mp4_set_64value(data->pos, trak->start_offset);
3384
3385     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
3386                 "start chunk offset:%0", trak->start_offset);
3387
3388     if (mp4->length) {
3389
3390         if (trak->end_chunk > trak->chunks) {
3391             ngx_log_error(NGX_LOG_ERR, mp4->file.log, 0,
3392                         "end time is out mp4 co64 chunks in \"%s\"",
3393                         mp4->file.name.data);
3394             return NGX_ERROR;
3395         }
3396
3397         entries = trak->end_chunk - trak->start_chunk;
3398         data->last = data->pos + entries * sizeof(uint64_t);
3399
3400         if (entries) {
3401             trak->end_offset =
3402                 ngx_mp4_get_64value(data->last - sizeof(uint64_t));
3403             trak->end_offset += trak->end_chunk_samples_size;
3404
3405             ngx_log_debug1(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
3406                         "end chunk offset:%0", trak->end_offset);
3407         }
3408
3409     } else {
3410         entries = trak->chunks - trak->start_chunk;
3411         trak->end_offset = mp4->mdat_data.buf->file_last;
3412     }
3413
3414     if (entries == 0) {
3415         trak->start_offset = mp4->end;
3416         trak->end_offset = 0;
3417     }
3418
3419     atom_size = sizeof(ngx_mp4_co64_atom_t) + (data->last - data->pos);
3420     trak->size += atom_size;
3421
3422     atom = trak->out[NGX_HTTP_MP4_CO64_ATOM].buf;
3423     co64_atom = (ngx_mp4_co64_atom_t *) atom->pos;
3424
3425     ngx_mp4_set_32value(co64_atom->size, atom_size);
3426     ngx_mp4_set_32value(co64_atom->entries, entries);
3427
3428     return NGX_OK;
3429 }
3430
3431

```

```

3432 static void
3433 ngx_http_mp4_adjust_co64_atom(ngx_http_mp4_file_t *mp4,
3434 ngx_http_mp4_trak_t *trak, off_t adjustment)
3435 {
3436     uint64_t    offset, *entry, *end;
3437     ngx_buf_t   *data;
3438
3439     /*
3440      * moov.trak.mdia.minf.stbl.co64 adjustment requires
3441      * minimal start offset of all traks and new moov atom size
3442      */
3443
3444     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, mp4->file.log, 0,
3445                  "mp4 co64 atom adjustment");
3446
3447     data = trak->out[NGX_HTTP_MP4_CO64_DATA].buf;
3448     entry = (uint64_t *) data->pos;
3449     end = (uint64_t *) data->last;
3450
3451     while (entry < end) {
3452         offset = ngx_mp4_get_64value(entry);
3453         offset += adjustment;
3454         ngx_mp4_set_64value(entry, offset);
3455         entry++;
3456     }
3457 }
3458
3459 static char *
3460 ngx_http_mp4(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
3461 {
3462     ngx_http_core_loc_conf_t *clcf;
3463
3464     clcf = ngx_http_conf_get_module_loc_conf(cf, ngx_http_core_module);
3465     clcf->handler = ngx_http_mp4_handler;
3466
3467     return NGX_CONF_OK;
3468 }
3469
3470
3471 static void *
3472 ngx_http_mp4_create_conf(ngx_conf_t *cf)
3473 {
3474     ngx_http_mp4_conf_t *conf;
3475
3476     conf = ngx_palloc(cf->pool, sizeof(ngx_http_mp4_conf_t));
3477     if (conf == NULL) {
3478         return NULL;
3479     }
3480
3481     conf->buffer_size = NGX_CONF_UNSET_SIZE;
3482     conf->max_buffer_size = NGX_CONF_UNSET_SIZE;
3483
3484     return conf;
3485 }
3486
3487
3488 static char *
3489 ngx_http_mp4_merge_conf(ngx_conf_t *cf, void *parent, void *child)
3490 {
3491     ngx_http_mp4_conf_t *prev = parent;
3492     ngx_http_mp4_conf_t *conf = child;
3493
3494     ngx_conf_merge_size_value(conf->buffer_size, prev->buffer_size, 512 * 1024);
3495     ngx_conf_merge_size_value(conf->max_buffer_size, prev->max_buffer_size,
3496                               10 * 1024 * 1024);
3497
3498     return NGX_CONF_OK;
3499 }
3500

```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_not\_modified\_filter\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_next\\_header\\_filter](#)
- [ngx\\_http\\_not\\_modified\\_filter\\_module](#)
- [ngx\\_http\\_not\\_modified\\_filter\\_module\\_ctx](#)

## Functions defined

- [ngx\\_http\\_not\\_modified\\_filter\\_init](#)
- [ngx\\_http\\_not\\_modified\\_header\\_filter](#)
- [ngx\\_http\\_test\\_if\\_match](#)
- [ngx\\_http\\_test\\_if\\_modified](#)
- [ngx\\_http\\_test\\_if\\_unmodified](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 static ngx_uint_t ngx_http_test_if_unmodified(ngx_http_request_t *r);
14 static ngx_uint_t ngx_http_test_if_modified(ngx_http_request_t *r);
15 static ngx_uint_t ngx_http_test_if_match(ngx_http_request_t *r,
16     ngx_table_elt_t *header, ngx_uint_t weak);
17 static ngx_int_t ngx_http_not_modified_filter_init(ngx_conf_t *cf);
18
19
20 static ngx_http_module_t ngx_http_not_modified_filter_module_ctx = {
21     NULL, /* preconfiguration */
22     ngx_http_not_modified_filter_init, /* postconfiguration */
23
24     NULL, /* create main configuration */
25     NULL, /* init main configuration */
26
27     NULL, /* create server configuration */
28     NULL, /* merge server configuration */
29
30     NULL, /* create location configuration */
31     NULL, /* merge location configuration */
32 };
33
34
35 ngx_module_t ngx_http_not_modified_filter_module = {
36     NGX_MODULE_V1,
37     &ngx_http_not_modified_filter_module_ctx, /* module context */
38     NULL, /* module directives */
39     NGX_HTTP_MODULE, /* module type */
40     NULL, /* init master */
41     NULL, /* init module */
42     NULL, /* init process */
43     NULL, /* init thread */
```

```

44     NULL,                                /* exit thread */
45     NULL,                                /* exit process */
46     NULL,                                /* exit master */
47     NGX_MODULE_V1_PADDING
48 };
49
50
51 static ngx_http_output_header_filter_pt ngx_http_next_header_filter;
52
53
54 static ngx_int_t
55 ngx_http_not_modified_header_filter(ngx_http_request_t *r)
56 {
57     if (r->headers_out.status != NGX_HTTP_OK
58         || r != r->main
59         || r->disable_not_modified)
60     {
61         return ngx_http_next_header_filter(r);
62     }
63
64     if (r->headers_in.if_unmodified_since
65         && !ngx_http_test_if_unmodified(r))
66     {
67         return ngx_http_filter_finalize_request(r, NULL,
68                                                 NGX_HTTP_PRECONDITION_FAILED);
69     }
70
71     if (r->headers_in.if_match
72         && !ngx_http_test_if_match(r, r->headers_in.if_match, 0))
73     {
74         return ngx_http_filter_finalize_request(r, NULL,
75                                                 NGX_HTTP_PRECONDITION_FAILED);
76     }
77
78     if (r->headers_in.if_modified_since || r->headers_in.if_none_match) {
79
80         if (r->headers_in.if_modified_since
81             && ngx_http_test_if_modified(r))
82         {
83             return ngx_http_next_header_filter(r);
84         }
85
86         if (r->headers_in.if_none_match
87             && !ngx_http_test_if_match(r, r->headers_in.if_none_match, 1))
88         {
89             return ngx_http_next_header_filter(r);
90         }
91
92         /* not modified */
93
94         r->headers_out.status = NGX_HTTP_NOT_MODIFIED;
95         r->headers_out.status_line.len = 0;
96         r->headers_out.content_type.len = 0;
97         ngx_http_clear_content_length(r);
98         ngx_http_clear_accept_ranges(r);
99
100        if (r->headers_out.content_encoding) {
101            r->headers_out.content_encoding->hash = 0;
102            r->headers_out.content_encoding = NULL;
103        }
104
105        return ngx_http_next_header_filter(r);
106    }
107
108    return ngx_http_next_header_filter(r);
109 }
110
111
112 static ngx_uint_t
113 ngx_http_test_if_unmodified(ngx_http_request_t *r)
114 {
115     time_t iums;
116
117     if (r->headers_out.last_modified_time == (time_t) -1) {
118         return 0;
119     }

```

```

120     iums = ngx_http_parse_time(r->headers_in.if_unmodified_since->value.data,
121                               r->headers_in.if_unmodified_since->value.len);
122
123
124     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
125                  "http iums:%T lm:%T", iums, r->headers_out.last_modified_time);
126
127     if (iums >= r->headers_out.last_modified_time) {
128         return 1;
129     }
130
131     return 0;
132 }
133
134
135 static ngx_uint_t
136 ngx_http_test_if_modified(ngx_http_request_t *r)
137 {
138     time_t          ims;
139     ngx_http_core_loc_conf_t *clcf;
140
141     if (r->headers_out.last_modified_time == (time_t) -1) {
142         return 1;
143     }
144
145     clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
146
147     if (clcf->if_modified_since == NGX_HTTP_IMS_OFF) {
148         return 1;
149     }
150
151     ims = ngx_http_parse_time(r->headers_in.if_modified_since->value.data,
152                               r->headers_in.if_modified_since->value.len);
153
154     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
155                  "http ims:%T lm:%T", ims, r->headers_out.last_modified_time);
156
157     if (ims == r->headers_out.last_modified_time) {
158         return 0;
159     }
160
161     if (clcf->if_modified_since == NGX_HTTP_IMS_EXACT
162         || ims < r->headers_out.last_modified_time)
163     {
164         return 1;
165     }
166
167     return 0;
168 }
169
170
171 static ngx_uint_t
172 ngx_http_test_if_match(ngx_http_request_t *r, ngx_table_elt_t *header,
173                       ngx_uint_t weak)
174 {
175     u_char    *start, *end, ch;
176     ngx_str_t  etag, *list;
177
178     list = &header->value;
179
180     if (list->len == 1 && list->data[0] == '*') {
181         return 1;
182     }
183
184     if (r->headers_out.etag == NULL) {
185         return 0;
186     }
187
188     etag = r->headers_out.etag->value;
189
190     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
191                  "http im:\"%V\" etag:%V", list, &etag);
192
193     if (weak
194         && etag.len > 2
195         && etag.data[0] == 'w'

```

```

196     && etag.data[1] == '/')
197 {
198     etag.len -= 2;
199     etag.data += 2;
200 }
201
202 start = list->data;
203 end = list->data + list->len;
204
205 while (start < end) {
206
207     if (weak
208         && end - start > 2
209         && start[0] == 'W'
210         && start[1] == '/')
211     {
212         start += 2;
213     }
214
215     if (etag.len > (size_t) (end - start)) {
216         return 0;
217     }
218
219     if (ngx\_strncmp(start, etag.data, etag.len) != 0) {
220         goto skip;
221     }
222
223     start += etag.len;
224
225     while (start < end) {
226         ch = *start;
227
228         if (ch == ' ' || ch == '\t') {
229             start++;
230             continue;
231         }
232
233         break;
234     }
235
236     if (start == end || *start == ',') {
237         return 1;
238     }
239
240 skip:
241
242     while (start < end && *start != ',') { start++; }
243     while (start < end) {
244         ch = *start;
245
246         if (ch == ' ' || ch == '\t' || ch == ',') {
247             start++;
248             continue;
249         }
250
251         break;
252     }
253 }
254
255 return 0;
256 }
257
258
259 static ngx\_int\_t
260 ngx\_http\_not\_modified\_filter\_init(ngx\_conf\_t *cf)
261 {
262     ngx\_http\_next\_header\_filter = ngx\_http\_top\_header\_filter;
263     ngx\_http\_top\_header\_filter = ngx\_http\_not\_modified\_header\_filter;
264
265     return NGX\_OK;
266 }

```

## src/http/modules/nginx\_http\_proxy\_module.c - nginx-1.7.10

### Global variables defined

- [ngx\\_http\\_proxy\\_cache\\_headers](#)
- [ngx\\_http\\_proxy\\_commands](#)
- [ngx\\_http\\_proxy\\_headers](#)
- [ngx\\_http\\_proxy\\_hide\\_headers](#)
- [ngx\\_http\\_proxy\\_http\\_version](#)
- [ngx\\_http\\_proxy\\_lowat\\_post](#)
- [ngx\\_http\\_proxy\\_module](#)
- [ngx\\_http\\_proxy\\_module](#)
- [ngx\\_http\\_proxy\\_module\\_ctx](#)
- [ngx\\_http\\_proxy\\_next\\_upstream\\_masks](#)
- [ngx\\_http\\_proxy\\_ssl\\_protocols](#)
- [ngx\\_http\\_proxy\\_temp\\_path](#)
- [ngx\\_http\\_proxy\\_vars](#)
- [ngx\\_http\\_proxy\\_version](#)
- [ngx\\_http\\_proxy\\_version\\_11](#)

### Data types defined

- [ngx\\_http\\_proxy\\_ctx\\_t](#)
- [ngx\\_http\\_proxy\\_headers\\_t](#)
- [ngx\\_http\\_proxy\\_loc\\_conf\\_t](#)
- [ngx\\_http\\_proxy\\_main\\_conf\\_t](#)
- [ngx\\_http\\_proxy\\_rewrite\\_pt](#)
- [ngx\\_http\\_proxy\\_rewrite\\_s](#)
- [ngx\\_http\\_proxy\\_rewrite\\_t](#)
- [ngx\\_http\\_proxy\\_vars\\_t](#)

### Functions defined

- [ngx\\_http\\_proxy\\_abort\\_request](#)
- [ngx\\_http\\_proxy\\_add\\_variables](#)
- [ngx\\_http\\_proxy\\_add\\_x\\_forwarded\\_for\\_variable](#)
- [ngx\\_http\\_proxy\\_cache](#)

- [ngx http proxy cache key](#)
- [ngx http proxy chunked filter](#)
- [ngx http proxy cookie domain](#)
- [ngx http proxy cookie path](#)
- [ngx http proxy copy filter](#)
- [ngx http proxy create key](#)
- [ngx http proxy create loc conf](#)
- [ngx http proxy create main conf](#)
- [ngx http proxy create request](#)
- [ngx http proxy eval](#)
- [ngx http proxy finalize request](#)
- [ngx http proxy handler](#)
- [ngx http proxy host variable](#)
- [ngx http proxy init headers](#)
- [ngx http proxy input filter init](#)
- [ngx http proxy internal body length variable](#)
- [ngx http proxy lowat check](#)
- [ngx http proxy merge loc conf](#)
- [ngx http proxy non buffered chunked filter](#)
- [ngx http proxy non buffered copy filter](#)
- [ngx http proxy pass](#)
- [ngx http proxy port variable](#)
- [ngx http proxy process header](#)
- [ngx http proxy process status line](#)
- [ngx http proxy redirect](#)
- [ngx http proxy reinit request](#)
- [ngx http proxy rewrite](#)
- [ngx http proxy rewrite complex handler](#)
- [ngx http proxy rewrite cookie](#)
- [ngx http proxy rewrite cookie value](#)
- [ngx http proxy rewrite domain handler](#)
- [ngx http proxy rewrite redirect](#)
- [ngx http proxy rewrite regex](#)



- [ngx\\_http\\_proxy\\_rewrite\\_regex\\_handler](#)
- [ngx\\_http\\_proxy\\_set\\_ssl](#)
- [ngx\\_http\\_proxy\\_set\\_vars](#)
- [ngx\\_http\\_proxy\\_ssl\\_password\\_file](#)
- [ngx\\_http\\_proxy\\_store](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 typedef struct {
14     ngx_array_t          caches; /* ngx_http_file_cache_t * */
15 } ngx_http_proxy_main_conf_t;
16
17
18 typedef struct ngx_http_proxy_rewrite_s ngx_http_proxy_rewrite_t;
19
20 typedef ngx_int_t (*ngx_http_proxy_rewrite_pt)(ngx_http_request_t *r,
21     ngx_table_elt_t *h, size_t prefix, size_t len,
22     ngx_http_proxy_rewrite_t *pr);
23
24 struct ngx_http_proxy_rewrite_s {
25     ngx_http_proxy_rewrite_pt  handler;
26
27     union {
28         ngx_http_complex_value_t  complex;
29 #if (NGX_PCRE)
30         ngx_http_regex_t          *regex;
31 #endif
32     } pattern;
33
34     ngx_http_complex_value_t  replacement;
35 };
36
37
38 typedef struct {
39     ngx_str_t          key_start;
40     ngx_str_t          schema;
41     ngx_str_t          host_header;
42     ngx_str_t          port;
43     ngx_str_t          uri;
44 } ngx_http_proxy_vars_t;
45
46
47 typedef struct {
48     ngx_array_t          *flushes;
49     ngx_array_t          *lengths;
50     ngx_array_t          *values;
51     ngx_hash_t          hash;
52 } ngx_http_proxy_headers_t;
53
54
55 typedef struct {
56     ngx_http_upstream_conf_t  upstream;
57
58     ngx_array_t          *body_flushes;
59     ngx_array_t          *body_lengths;
60     ngx_array_t          *body_values;
61     ngx_str_t          body_source;

```

```

62
63     ngx\_http\_proxy\_headers\_t         headers;
64 #if (NGX_HTTP_CACHE)
65     ngx\_http\_proxy\_headers\_t         headers_cache;
66 #endif
67     ngx\_array\_t                       *headers_source;
68
69     ngx\_array\_t                       *proxy_lengths;
70     ngx\_array\_t                       *proxy_values;
71
72     ngx\_array\_t                       *redirects;
73     ngx\_array\_t                       *cookie_domains;
74     ngx\_array\_t                       *cookie_paths;
75
76     ngx\_str\_t                          method;
77     ngx\_str\_t                          location;
78     ngx\_str\_t                          url;
79
80 #if (NGX_HTTP_CACHE)
81     ngx\_http\_complex\_value\_t         cache_key;
82 #endif
83
84     ngx\_http\_proxy\_vars\_t            vars;
85
86     ngx\_flag\_t                       redirect;
87
88     ngx\_uint\_t                       http_version;
89
90     ngx\_uint\_t                       headers_hash_max_size;
91     ngx\_uint\_t                       headers_hash_bucket_size;
92
93 #if (NGX_HTTP_SSL)
94     ngx\_uint\_t                       ssl;
95     ngx\_uint\_t                       ssl_protocols;
96     ngx\_str\_t                          ssl_ciphers;
97     ngx\_uint\_t                       ssl_verify_depth;
98     ngx\_str\_t                          ssl_trusted_certificate;
99     ngx\_str\_t                          ssl_crl;
100    ngx\_str\_t                          ssl_certificate;
101    ngx\_str\_t                          ssl_certificate_key;
102    ngx\_array\_t                       *ssl_passwords;
103 #endif
104 } ngx\_http\_proxy\_loc\_conf\_t;
105
106
107 typedef struct {
108     ngx\_http\_status\_t                status;
109     ngx\_http\_chunked\_t              chunked;
110     ngx\_http\_proxy\_vars\_t          vars;
111     off_t                          internal_body_length;
112
113     ngx\_uint\_t                       head; /* unsigned head:1 */
114 } ngx\_http\_proxy\_ctx\_t;
115
116
117 static ngx\_int\_t ngx\_http\_proxy\_eval(ngx\_http\_request\_t *r,
118     ngx\_http\_proxy\_ctx\_t *ctx, ngx\_http\_proxy\_loc\_conf\_t *plcf);
119 #if (NGX_HTTP_CACHE)
120 static ngx\_int\_t ngx\_http\_proxy\_create\_key(ngx\_http\_request\_t *r);
121 #endif
122 static ngx\_int\_t ngx\_http\_proxy\_create\_request(ngx\_http\_request\_t *r);
123 static ngx\_int\_t ngx\_http\_proxy\_reinit\_request(ngx\_http\_request\_t *r);
124 static ngx\_int\_t ngx\_http\_proxy\_process\_status\_line(ngx\_http\_request\_t *r);
125 static ngx\_int\_t ngx\_http\_proxy\_process\_header(ngx\_http\_request\_t *r);
126 static ngx\_int\_t ngx\_http\_proxy\_input\_filter\_init(void *data);
127 static ngx\_int\_t ngx\_http\_proxy\_copy\_filter(ngx\_event\_pipe\_t *p,
128     ngx\_buf\_t *buf);
129 static ngx\_int\_t ngx\_http\_proxy\_chunked\_filter(ngx\_event\_pipe\_t *p,
130     ngx\_buf\_t *buf);
131 static ngx\_int\_t ngx\_http\_proxy\_non\_buffered\_copy\_filter(void *data,
132     ssize_t bytes);
133 static ngx\_int\_t ngx\_http\_proxy\_non\_buffered\_chunked\_filter(void *data,
134     ssize_t bytes);
135 static void ngx\_http\_proxy\_abort\_request(ngx\_http\_request\_t *r);
136 static void ngx\_http\_proxy\_finalize\_request(ngx\_http\_request\_t *r,
137     ngx\_int\_t rc);

```

```

138
139 static ngx_int_t ngx_http_proxy_host_variable(ngx_http_request_t *r,
140     ngx_http_variable_value_t *v, uintptr_t data);
141 static ngx_int_t ngx_http_proxy_port_variable(ngx_http_request_t *r,
142     ngx_http_variable_value_t *v, uintptr_t data);
143 static ngx_int_t
144     ngx_http_proxy_add_x_forwarded_for_variable(ngx_http_request_t *r,
145     ngx_http_variable_value_t *v, uintptr_t data);
146 static ngx_int_t
147     ngx_http_proxy_internal_body_length_variable(ngx_http_request_t *r,
148     ngx_http_variable_value_t *v, uintptr_t data);
149 static ngx_int_t ngx_http_proxy_rewrite_redirect(ngx_http_request_t *r,
150     ngx_table_elt_t *h, size_t prefix);
151 static ngx_int_t ngx_http_proxy_rewrite_cookie(ngx_http_request_t *r,
152     ngx_table_elt_t *h);
153 static ngx_int_t ngx_http_proxy_rewrite_cookie_value(ngx_http_request_t *r,
154     ngx_table_elt_t *h, u_char *value, ngx_array_t *rewrites);
155 static ngx_int_t ngx_http_proxy_rewrite(ngx_http_request_t *r,
156     ngx_table_elt_t *h, size_t prefix, size_t len, ngx_str_t *replacement);
157
158 static ngx_int_t ngx_http_proxy_add_variables(ngx_conf_t *cf);
159 static void *ngx_http_proxy_create_main_conf(ngx_conf_t *cf);
160 static void *ngx_http_proxy_create_loc_conf(ngx_conf_t *cf);
161 static char *ngx_http_proxy_merge_loc_conf(ngx_conf_t *cf,
162     void *parent, void *child);
163 static ngx_int_t ngx_http_proxy_init_headers(ngx_conf_t *cf,
164     ngx_http_proxy_loc_conf_t *conf, ngx_http_proxy_headers_t *headers,
165     ngx_keyval_t *default_headers);
166
167 static char *ngx_http_proxy_pass(ngx_conf_t *cf, ngx_command_t *cmd,
168     void *conf);
169 static char *ngx_http_proxy_redirect(ngx_conf_t *cf, ngx_command_t *cmd,
170     void *conf);
171 static char *ngx_http_proxy_cookie_domain(ngx_conf_t *cf, ngx_command_t *cmd,
172     void *conf);
173 static char *ngx_http_proxy_cookie_path(ngx_conf_t *cf, ngx_command_t *cmd,
174     void *conf);
175 static char *ngx_http_proxy_store(ngx_conf_t *cf, ngx_command_t *cmd,
176     void *conf);
177 #if (NGX_HTTP_CACHE)
178 static char *ngx_http_proxy_cache(ngx_conf_t *cf, ngx_command_t *cmd,
179     void *conf);
180 static char *ngx_http_proxy_cache_key(ngx_conf_t *cf, ngx_command_t *cmd,
181     void *conf);
182 #endif
183 #if (NGX_HTTP_SSL)
184 static char *ngx_http_proxy_ssl_password_file(ngx_conf_t *cf,
185     ngx_command_t *cmd, void *conf);
186 #endif
187
188 static char *ngx_http_proxy_lowat_check(ngx_conf_t *cf, void *post, void *data);
189
190 static ngx_int_t ngx_http_proxy_rewrite_regex(ngx_conf_t *cf,
191     ngx_http_proxy_rewrite_t *pr, ngx_str_t *regex, ngx_uint_t caseless);
192
193 #if (NGX_HTTP_SSL)
194 static ngx_int_t ngx_http_proxy_set_ssl(ngx_conf_t *cf,
195     ngx_http_proxy_loc_conf_t *plcf);
196 #endif
197 static void ngx_http_proxy_set_vars(ngx_url_t *u, ngx_http_proxy_vars_t *v);
198
199
200 static ngx_conf_post_t  ngx_http_proxy_lowat_post =
201     { ngx_http_proxy_lowat_check };
202
203
204 static ngx_conf_bitmask_t  ngx_http_proxy_next_upstream_masks[] = {
205     { ngx_string("error"), NGX_HTTP_UPSTREAM_FT_ERROR },
206     { ngx_string("timeout"), NGX_HTTP_UPSTREAM_FT_TIMEOUT },
207     { ngx_string("invalid_header"), NGX_HTTP_UPSTREAM_FT_INVALID_HEADER },
208     { ngx_string("http_500"), NGX_HTTP_UPSTREAM_FT_HTTP_500 },
209     { ngx_string("http_502"), NGX_HTTP_UPSTREAM_FT_HTTP_502 },
210     { ngx_string("http_503"), NGX_HTTP_UPSTREAM_FT_HTTP_503 },
211     { ngx_string("http_504"), NGX_HTTP_UPSTREAM_FT_HTTP_504 },
212     { ngx_string("http_403"), NGX_HTTP_UPSTREAM_FT_HTTP_403 },
213     { ngx_string("http_404"), NGX_HTTP_UPSTREAM_FT_HTTP_404 },

```

```

214     { ngx_string("updating"), NGX_HTTP_UPSTREAM_FT_UPDATING },
215     { ngx_string("off"), NGX_HTTP_UPSTREAM_FT_OFF },
216     { ngx_null_string, 0 }
217 };
218
219
220 #if (NGX_HTTP_SSL)
221
222 static ngx_conf_bitmask_t ngx_http_proxy_ssl_protocols[] = {
223     { ngx_string("SSLv2"), NGX_SSL_SSLV2 },
224     { ngx_string("SSLv3"), NGX_SSL_SSLV3 },
225     { ngx_string("TLSv1"), NGX_SSL_TLSV1 },
226     { ngx_string("TLSv1.1"), NGX_SSL_TLSV1_1 },
227     { ngx_string("TLSv1.2"), NGX_SSL_TLSV1_2 },
228     { ngx_null_string, 0 }
229 };
230
231 #endif
232
233
234 static ngx_conf_enum_t ngx_http_proxy_http_version[] = {
235     { ngx_string("1.0"), NGX_HTTP_VERSION_10 },
236     { ngx_string("1.1"), NGX_HTTP_VERSION_11 },
237     { ngx_null_string, 0 }
238 };
239
240
241 ngx_module_t ngx_http_proxy_module;
242
243
244 static ngx_command_t ngx_http_proxy_commands[] = {
245
246     { ngx_string("proxy_pass"),
247       NGX_HTTP_LOC_CONF|NGX_HTTP_LIF_CONF|NGX_HTTP_LMT_CONF|NGX_CONF TAKE1,
248       ngx_http_proxy_pass,
249       NGX_HTTP_LOC_CONF_OFFSET,
250       0,
251       NULL },
252
253     { ngx_string("proxy_redirect"),
254       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF TAKE12,
255       ngx_http_proxy_redirect,
256       NGX_HTTP_LOC_CONF_OFFSET,
257       0,
258       NULL },
259
260     { ngx_string("proxy_cookie_domain"),
261       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF TAKE12,
262       ngx_http_proxy_cookie_domain,
263       NGX_HTTP_LOC_CONF_OFFSET,
264       0,
265       NULL },
266
267     { ngx_string("proxy_cookie_path"),
268       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF TAKE12,
269       ngx_http_proxy_cookie_path,
270       NGX_HTTP_LOC_CONF_OFFSET,
271       0,
272       NULL },
273
274     { ngx_string("proxy_store"),
275       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF TAKE1,
276       ngx_http_proxy_store,
277       NGX_HTTP_LOC_CONF_OFFSET,
278       0,
279       NULL },
280
281     { ngx_string("proxy_store_access"),
282       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF TAKE123,
283       ngx_conf_set_access_slot,
284       NGX_HTTP_LOC_CONF_OFFSET,
285       offsetof(ngx_http_proxy_loc_conf_t, upstream.store_access),
286       NULL },
287
288     { ngx_string("proxy_buffering"),
289       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF FLAG,

```

```

290     ngx_conf_set_flag_slot,
291     NGX\_HTTP\_LOC\_CONF\_OFFSET,
292     offsetof(ngx\_http\_proxy\_loc\_conf\_t, upstream.buffering),
293     NULL },
294
295 { ngx\_string\("proxy\_ignore\_client\_abort"\),
296     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_FLAG,
297     ngx_conf_set_flag_slot,
298     NGX\_HTTP\_LOC\_CONF\_OFFSET,
299     offsetof(ngx\_http\_proxy\_loc\_conf\_t, upstream.ignore_client_abort),
300     NULL },
301
302 { ngx\_string\("proxy\_bind"\),
303     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
304     ngx_http_upstream_bind_set_slot,
305     NGX\_HTTP\_LOC\_CONF\_OFFSET,
306     offsetof(ngx\_http\_proxy\_loc\_conf\_t, upstream.local),
307     NULL },
308
309 { ngx\_string\("proxy\_connect\_timeout"\),
310     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
311     ngx_conf_set_msec_slot,
312     NGX\_HTTP\_LOC\_CONF\_OFFSET,
313     offsetof(ngx\_http\_proxy\_loc\_conf\_t, upstream.connect_timeout),
314     NULL },
315
316 { ngx\_string\("proxy\_send\_timeout"\),
317     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
318     ngx_conf_set_msec_slot,
319     NGX\_HTTP\_LOC\_CONF\_OFFSET,
320     offsetof(ngx\_http\_proxy\_loc\_conf\_t, upstream.send_timeout),
321     NULL },
322
323 { ngx\_string\("proxy\_send\_lowat"\),
324     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
325     ngx_conf_set_size_slot,
326     NGX\_HTTP\_LOC\_CONF\_OFFSET,
327     offsetof(ngx\_http\_proxy\_loc\_conf\_t, upstream.send_lowat),
328     &ngx\_http\_proxy\_lowat\_post },
329
330 { ngx\_string\("proxy\_intercept\_errors"\),
331     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_FLAG,
332     ngx_conf_set_flag_slot,
333     NGX\_HTTP\_LOC\_CONF\_OFFSET,
334     offsetof(ngx\_http\_proxy\_loc\_conf\_t, upstream.intercept_errors),
335     NULL },
336
337 { ngx\_string\("proxy\_set\_header"\),
338     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE2,
339     ngx_conf_set_keyval_slot,
340     NGX\_HTTP\_LOC\_CONF\_OFFSET,
341     offsetof(ngx\_http\_proxy\_loc\_conf\_t, headers_source),
342     NULL },
343
344 { ngx\_string\("proxy\_headers\_hash\_max\_size"\),
345     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
346     ngx_conf_set_num_slot,
347     NGX\_HTTP\_LOC\_CONF\_OFFSET,
348     offsetof(ngx\_http\_proxy\_loc\_conf\_t, headers_hash_max_size),
349     NULL },
350
351 { ngx\_string\("proxy\_headers\_hash\_bucket\_size"\),
352     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
353     ngx_conf_set_num_slot,
354     NGX\_HTTP\_LOC\_CONF\_OFFSET,
355     offsetof(ngx\_http\_proxy\_loc\_conf\_t, headers_hash_bucket_size),
356     NULL },
357
358 { ngx\_string\("proxy\_set\_body"\),
359     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
360     ngx_conf_set_str_slot,
361     NGX\_HTTP\_LOC\_CONF\_OFFSET,
362     offsetof(ngx\_http\_proxy\_loc\_conf\_t, body_source),
363     NULL },
364
365 { ngx\_string\("proxy\_method"\),

```

```

366 NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
367 ngx\_conf\_set\_str\_slot,
368 NGX\_HTTP\_LOC\_CONF\_OFFSET,
369 offsetof(ngx\_http\_proxy\_loc\_conf\_t, method),
370 NULL },
371
372 { ngx\_string\("proxy\_pass\_request\_headers"\),
373 NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_FLAG,
374 ngx\_conf\_set\_flag\_slot,
375 NGX\_HTTP\_LOC\_CONF\_OFFSET,
376 offsetof(ngx\_http\_proxy\_loc\_conf\_t, upstream.pass_request_headers),
377 NULL },
378
379 { ngx\_string\("proxy\_pass\_request\_body"\),
380 NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_FLAG,
381 ngx\_conf\_set\_flag\_slot,
382 NGX\_HTTP\_LOC\_CONF\_OFFSET,
383 offsetof(ngx\_http\_proxy\_loc\_conf\_t, upstream.pass_request_body),
384 NULL },
385
386 { ngx\_string\("proxy\_buffer\_size"\),
387 NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
388 ngx\_conf\_set\_size\_slot,
389 NGX\_HTTP\_LOC\_CONF\_OFFSET,
390 offsetof(ngx\_http\_proxy\_loc\_conf\_t, upstream.buffer_size),
391 NULL },
392
393 { ngx\_string\("proxy\_read\_timeout"\),
394 NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
395 ngx\_conf\_set\_msec\_slot,
396 NGX\_HTTP\_LOC\_CONF\_OFFSET,
397 offsetof(ngx\_http\_proxy\_loc\_conf\_t, upstream.read_timeout),
398 NULL },
399
400 { ngx\_string\("proxy\_buffers"\),
401 NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE2,
402 ngx\_conf\_set\_bufs\_slot,
403 NGX\_HTTP\_LOC\_CONF\_OFFSET,
404 offsetof(ngx\_http\_proxy\_loc\_conf\_t, upstream.bufs),
405 NULL },
406
407 { ngx\_string\("proxy\_busy\_buffers\_size"\),
408 NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
409 ngx\_conf\_set\_size\_slot,
410 NGX\_HTTP\_LOC\_CONF\_OFFSET,
411 offsetof(ngx\_http\_proxy\_loc\_conf\_t, upstream.busy_buffers_size_conf),
412 NULL },
413
414 { ngx\_string\("proxy\_force\_ranges"\),
415 NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_FLAG,
416 ngx\_conf\_set\_flag\_slot,
417 NGX\_HTTP\_LOC\_CONF\_OFFSET,
418 offsetof(ngx\_http\_proxy\_loc\_conf\_t, upstream.force_ranges),
419 NULL },
420
421 { ngx\_string\("proxy\_limit\_rate"\),
422 NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
423 ngx\_conf\_set\_size\_slot,
424 NGX\_HTTP\_LOC\_CONF\_OFFSET,
425 offsetof(ngx\_http\_proxy\_loc\_conf\_t, upstream.limit_rate),
426 NULL },
427
428 #if (NGX_HTTP_CACHE)
429
430 { ngx\_string\("proxy\_cache"\),
431 NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
432 ngx\_http\_proxy\_cache,
433 NGX\_HTTP\_LOC\_CONF\_OFFSET,
434 0,
435 NULL },
436
437 { ngx\_string\("proxy\_cache\_key"\),
438 NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
439 ngx\_http\_proxy\_cache\_key,
440 NGX\_HTTP\_LOC\_CONF\_OFFSET,
441 0,

```

```

442     NULL },
443
444 { ngx_string("proxy_cache_path"),
445     NGX_HTTP_MAIN_CONF|NGX_CONF_2MORE,
446     ngx_http_file_cache_set_slot,
447     NGX_HTTP_MAIN_CONF_OFFSET,
448     offsetof(ngx_http_proxy_main_conf_t, caches),
449     &ngx_http_proxy_module },
450
451 { ngx_string("proxy_cache_bypass"),
452     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_1MORE,
453     ngx_http_set_predicate_slot,
454     NGX_HTTP_LOC_CONF_OFFSET,
455     offsetof(ngx_http_proxy_loc_conf_t, upstream.cache_bypass),
456     NULL },
457
458 { ngx_string("proxy_no_cache"),
459     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_1MORE,
460     ngx_http_set_predicate_slot,
461     NGX_HTTP_LOC_CONF_OFFSET,
462     offsetof(ngx_http_proxy_loc_conf_t, upstream.no_cache),
463     NULL },
464
465 { ngx_string("proxy_cache_valid"),
466     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_1MORE,
467     ngx_http_file_cache_valid_set_slot,
468     NGX_HTTP_LOC_CONF_OFFSET,
469     offsetof(ngx_http_proxy_loc_conf_t, upstream.cache_valid),
470     NULL },
471
472 { ngx_string("proxy_cache_min_uses"),
473     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
474     ngx_conf_set_num_slot,
475     NGX_HTTP_LOC_CONF_OFFSET,
476     offsetof(ngx_http_proxy_loc_conf_t, upstream.cache_min_uses),
477     NULL },
478
479 { ngx_string("proxy_cache_use_stale"),
480     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_1MORE,
481     ngx_conf_set_bitmask_slot,
482     NGX_HTTP_LOC_CONF_OFFSET,
483     offsetof(ngx_http_proxy_loc_conf_t, upstream.cache_use_stale),
484     &ngx_http_proxy_next_upstream_masks },
485
486 { ngx_string("proxy_cache_methods"),
487     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_1MORE,
488     ngx_conf_set_bitmask_slot,
489     NGX_HTTP_LOC_CONF_OFFSET,
490     offsetof(ngx_http_proxy_loc_conf_t, upstream.cache_methods),
491     &ngx_http_upstream_cache_method_mask },
492
493 { ngx_string("proxy_cache_lock"),
494     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
495     ngx_conf_set_flag_slot,
496     NGX_HTTP_LOC_CONF_OFFSET,
497     offsetof(ngx_http_proxy_loc_conf_t, upstream.cache_lock),
498     NULL },
499
500 { ngx_string("proxy_cache_lock_timeout"),
501     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
502     ngx_conf_set_msec_slot,
503     NGX_HTTP_LOC_CONF_OFFSET,
504     offsetof(ngx_http_proxy_loc_conf_t, upstream.cache_lock_timeout),
505     NULL },
506
507 { ngx_string("proxy_cache_lock_age"),
508     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
509     ngx_conf_set_msec_slot,
510     NGX_HTTP_LOC_CONF_OFFSET,
511     offsetof(ngx_http_proxy_loc_conf_t, upstream.cache_lock_age),
512     NULL },
513
514 { ngx_string("proxy_cache_revalidate"),
515     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
516     ngx_conf_set_flag_slot,
517     NGX_HTTP_LOC_CONF_OFFSET,

```

```

518     offsetof(ngx_http_proxy_loc_conf_t, upstream.cache_revalidate),
519     NULL },
520
521 #endif
522
523 { ngx_string("proxy_temp_path"),
524   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1234,
525   ngx_conf_set_path_slot,
526   NGX_HTTP_LOC_CONF_OFFSET,
527   offsetof(ngx_http_proxy_loc_conf_t, upstream.temp_path),
528   NULL },
529
530 { ngx_string("proxy_max_temp_file_size"),
531   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
532   ngx_conf_set_size_slot,
533   NGX_HTTP_LOC_CONF_OFFSET,
534   offsetof(ngx_http_proxy_loc_conf_t, upstream.max_temp_file_size_conf),
535   NULL },
536
537 { ngx_string("proxy_temp_file_write_size"),
538   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
539   ngx_conf_set_size_slot,
540   NGX_HTTP_LOC_CONF_OFFSET,
541   offsetof(ngx_http_proxy_loc_conf_t, upstream.temp_file_write_size_conf),
542   NULL },
543
544 { ngx_string("proxy_next_upstream"),
545   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_1MORE,
546   ngx_conf_set_bitmask_slot,
547   NGX_HTTP_LOC_CONF_OFFSET,
548   offsetof(ngx_http_proxy_loc_conf_t, upstream.next_upstream),
549   &ngx_http_proxy_next_upstream_masks },
550
551 { ngx_string("proxy_next_upstream_tries"),
552   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
553   ngx_conf_set_num_slot,
554   NGX_HTTP_LOC_CONF_OFFSET,
555   offsetof(ngx_http_proxy_loc_conf_t, upstream.next_upstream_tries),
556   NULL },
557
558 { ngx_string("proxy_next_upstream_timeout"),
559   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
560   ngx_conf_set_msec_slot,
561   NGX_HTTP_LOC_CONF_OFFSET,
562   offsetof(ngx_http_proxy_loc_conf_t, upstream.next_upstream_timeout),
563   NULL },
564
565 { ngx_string("proxy_pass_header"),
566   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
567   ngx_conf_set_str_array_slot,
568   NGX_HTTP_LOC_CONF_OFFSET,
569   offsetof(ngx_http_proxy_loc_conf_t, upstream.pass_headers),
570   NULL },
571
572 { ngx_string("proxy_hide_header"),
573   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
574   ngx_conf_set_str_array_slot,
575   NGX_HTTP_LOC_CONF_OFFSET,
576   offsetof(ngx_http_proxy_loc_conf_t, upstream.hide_headers),
577   NULL },
578
579 { ngx_string("proxy_ignore_headers"),
580   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_1MORE,
581   ngx_conf_set_bitmask_slot,
582   NGX_HTTP_LOC_CONF_OFFSET,
583   offsetof(ngx_http_proxy_loc_conf_t, upstream.ignore_headers),
584   &ngx_http_upstream_ignore_headers_masks },
585
586 { ngx_string("proxy_http_version"),
587   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
588   ngx_conf_set_enum_slot,
589   NGX_HTTP_LOC_CONF_OFFSET,
590   offsetof(ngx_http_proxy_loc_conf_t, http_version),
591   &ngx_http_proxy_http_version },
592
593 #if (NGX_HTTP_SSL)

```



```

594 { ngx_string("proxy_ssl_session_reuse"),
595     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_FLAG,
596     ngx_conf_set_flag_slot,
597     NGX\_HTTP\_LOC\_CONF\_OFFSET,
598     offsetof(ngx\_http\_proxy\_loc\_conf\_t, upstream.ssl_session_reuse),
599     NULL },
600
601
602 { ngx_string("proxy_ssl_protocols"),
603     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_1MORE,
604     ngx_conf_set_bitmask_slot,
605     NGX\_HTTP\_LOC\_CONF\_OFFSET,
606     offsetof(ngx\_http\_proxy\_loc\_conf\_t, ssl_protocols),
607     &ngx\_http\_proxy\_ssl\_protocols },
608
609 { ngx_string("proxy_ssl_ciphers"),
610     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
611     ngx_conf_set_str_slot,
612     NGX\_HTTP\_LOC\_CONF\_OFFSET,
613     offsetof(ngx\_http\_proxy\_loc\_conf\_t, ssl_ciphers),
614     NULL },
615
616 { ngx_string("proxy_ssl_name"),
617     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
618     ngx_http_set_complex_value_slot,
619     NGX\_HTTP\_LOC\_CONF\_OFFSET,
620     offsetof(ngx\_http\_proxy\_loc\_conf\_t, upstream.ssl_name),
621     NULL },
622
623 { ngx_string("proxy_ssl_server_name"),
624     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_FLAG,
625     ngx_conf_set_flag_slot,
626     NGX\_HTTP\_LOC\_CONF\_OFFSET,
627     offsetof(ngx\_http\_proxy\_loc\_conf\_t, upstream.ssl_server_name),
628     NULL },
629
630 { ngx_string("proxy_ssl_verify"),
631     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_FLAG,
632     ngx_conf_set_flag_slot,
633     NGX\_HTTP\_LOC\_CONF\_OFFSET,
634     offsetof(ngx\_http\_proxy\_loc\_conf\_t, upstream.ssl_verify),
635     NULL },
636
637 { ngx_string("proxy_ssl_verify_depth"),
638     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
639     ngx_conf_set_num_slot,
640     NGX\_HTTP\_LOC\_CONF\_OFFSET,
641     offsetof(ngx\_http\_proxy\_loc\_conf\_t, ssl_verify_depth),
642     NULL },
643
644 { ngx_string("proxy_ssl_trusted_certificate"),
645     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
646     ngx_conf_set_str_slot,
647     NGX\_HTTP\_LOC\_CONF\_OFFSET,
648     offsetof(ngx\_http\_proxy\_loc\_conf\_t, ssl_trusted_certificate),
649     NULL },
650
651 { ngx_string("proxy_ssl_crl"),
652     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
653     ngx_conf_set_str_slot,
654     NGX\_HTTP\_LOC\_CONF\_OFFSET,
655     offsetof(ngx\_http\_proxy\_loc\_conf\_t, ssl_crl),
656     NULL },
657
658 { ngx_string("proxy_ssl_certificate"),
659     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
660     ngx_conf_set_str_slot,
661     NGX\_HTTP\_LOC\_CONF\_OFFSET,
662     offsetof(ngx\_http\_proxy\_loc\_conf\_t, ssl_certificate),
663     NULL },
664
665 { ngx_string("proxy_ssl_certificate_key"),
666     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
667     ngx_conf_set_str_slot,
668     NGX\_HTTP\_LOC\_CONF\_OFFSET,
669     offsetof(ngx\_http\_proxy\_loc\_conf\_t, ssl_certificate_key),

```

```

670     NULL },
671
672     { ngx_string("proxy_ssl_password_file"),
673       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
674       ngx_http_proxy_ssl_password_file,
675       NGX_HTTP_LOC_CONF_OFFSET,
676       0,
677       NULL },
678
679 #endif
680
681     ngx_null_command
682 };
683
684
685 static ngx_http_module_t  ngx_http_proxy_module_ctx = {
686     ngx_http_proxy_add_variables,      /* preconfiguration */
687     NULL,                              /* postconfiguration */
688
689     ngx_http_proxy_create_main_conf,   /* create main configuration */
690     NULL,                              /* init main configuration */
691
692     NULL,                              /* create server configuration */
693     NULL,                              /* merge server configuration */
694
695     ngx_http_proxy_create_loc_conf,    /* create location configuration */
696     ngx_http_proxy_merge_loc_conf     /* merge location configuration */
697 };
698
699
700 ngx_module_t  ngx_http_proxy_module = {
701     NGX_MODULE_V1,
702     &ngx_http_proxy_module_ctx,      /* module context */
703     ngx_http_proxy_commands,        /* module directives */
704     NGX_HTTP_MODULE,                /* module type */
705     NULL,                            /* init master */
706     NULL,                            /* init module */
707     NULL,                            /* init process */
708     NULL,                            /* init thread */
709     NULL,                            /* exit thread */
710     NULL,                            /* exit process */
711     NULL,                            /* exit master */
712     NGX_MODULE_V1_PADDING
713 };
714
715
716 static char  ngx_http_proxy_version[] = " HTTP/1.0" CRLF;
717 static char  ngx_http_proxy_version_11[] = " HTTP/1.1" CRLF;
718
719
720 static ngx_keyval_t  ngx_http_proxy_headers[] = {
721     { ngx_string("Host"), ngx_string("$proxy_host") },
722     { ngx_string("Connection"), ngx_string("close") },
723     { ngx_string("Content-Length"), ngx_string("$proxy_internal_body_length") },
724     { ngx_string("TE"), ngx_string("") },
725     { ngx_string("Transfer-Encoding"), ngx_string("") },
726     { ngx_string("Keep-Alive"), ngx_string("") },
727     { ngx_string("Expect"), ngx_string("") },
728     { ngx_string("Upgrade"), ngx_string("") },
729     { ngx_null_string, ngx_null_string }
730 };
731
732
733 static ngx_str_t  ngx_http_proxy_hide_headers[] = {
734     ngx_string("Date"),
735     ngx_string("Server"),
736     ngx_string("X-Pad"),
737     ngx_string("X-Accel-Expires"),
738     ngx_string("X-Accel-Redirect"),
739     ngx_string("X-Accel-Limit-Rate"),
740     ngx_string("X-Accel-Buffering"),
741     ngx_string("X-Accel-Charset"),
742     ngx_null_string
743 };
744
745

```

```

746 #if (NGX_HTTP_CACHE)
747
748 static ngx_keyval_t ngx_http_proxy_cache_headers[] = {
749     { ngx_string("Host"), ngx_string("$proxy_host") },
750     { ngx_string("Connection"), ngx_string("close") },
751     { ngx_string("Content-Length"), ngx_string("$proxy_internal_body_length") },
752     { ngx_string("TE"), ngx_string("") },
753     { ngx_string("Transfer-Encoding"), ngx_string("") },
754     { ngx_string("Keep-Alive"), ngx_string("") },
755     { ngx_string("Expect"), ngx_string("") },
756     { ngx_string("Upgrade"), ngx_string("") },
757     { ngx_string("If-Modified-Since"),
758       ngx_string("$upstream_cache_last_modified") },
759     { ngx_string("If-Unmodified-Since"), ngx_string("") },
760     { ngx_string("If-None-Match"), ngx_string("$upstream_cache_etag") },
761     { ngx_string("If-Match"), ngx_string("") },
762     { ngx_string("Range"), ngx_string("") },
763     { ngx_string("If-Range"), ngx_string("") },
764     { ngx_null_string, ngx_null_string }
765 };
766
767 #endif
768
769
770 static ngx_http_variable_t ngx_http_proxy_vars[] = {
771
772     { ngx_string("proxy_host"), NULL, ngx_http_proxy_host_variable, 0,
773       NGX_HTTP_VAR_CHANGEABLE|NGX_HTTP_VAR_NOCACHEABLE|NGX_HTTP_VAR_NOHASH, 0 },
774
775     { ngx_string("proxy_port"), NULL, ngx_http_proxy_port_variable, 0,
776       NGX_HTTP_VAR_CHANGEABLE|NGX_HTTP_VAR_NOCACHEABLE|NGX_HTTP_VAR_NOHASH, 0 },
777
778     { ngx_string("proxy_add_x_forwarded_for"), NULL,
779       ngx_http_proxy_add_x_forwarded_for_variable, 0, NGX_HTTP_VAR_NOHASH, 0 },
780
781     #if 0
782     { ngx_string("proxy_add_via"), NULL, NULL, 0, NGX_HTTP_VAR_NOHASH, 0 },
783     #endif
784
785     { ngx_string("proxy_internal_body_length"), NULL,
786       ngx_http_proxy_internal_body_length_variable, 0,
787       NGX_HTTP_VAR_NOCACHEABLE|NGX_HTTP_VAR_NOHASH, 0 },
788
789     { ngx_null_string, NULL, NULL, 0, 0, 0 }
790 };
791
792
793 static ngx_path_init_t ngx_http_proxy_temp_path = {
794     ngx_string(NGX_HTTP_PROXY_TEMP_PATH), { 1, 2, 0 }
795 };
796
797
798 static ngx_int_t
799 ngx_http_proxy_handler(ngx_http_request_t *r)
800 {
801     ngx_int_t          rc;
802     ngx_http_upstream_t *u;
803     ngx_http_proxy_ctx_t *ctx;
804     ngx_http_proxy_loc_conf_t *plcf;
805     #if (NGX_HTTP_CACHE)
806     ngx_http_proxy_main_conf_t *pmcf;
807     #endif
808
809     if (ngx_http_upstream_create(r) != NGX_OK) {
810         return NGX_HTTP_INTERNAL_SERVER_ERROR;
811     }
812
813     ctx = ngx_palloc(r->pool, sizeof(ngx_http_proxy_ctx_t));
814     if (ctx == NULL) {
815         return NGX_ERROR;
816     }
817
818     ngx_http_set_ctx(r, ctx, ngx_http_proxy_module);
819
820     plcf = ngx_http_get_module_loc_conf(r, ngx_http_proxy_module);
821

```

```

822     u = r->upstream;
823
824     if (plcf->proxy_lengths == NULL) {
825         ctx->vars = plcf->vars;
826         u->schema = plcf->vars.schema;
827 #if (NGX_HTTP_SSL)
828         u->ssl = (plcf->upstream.ssl != NULL);
829 #endif
830
831     } else {
832         if (ngx_http_proxy_eval(r, ctx, plcf) != NGX_OK) {
833             return NGX_HTTP_INTERNAL_SERVER_ERROR;
834         }
835     }
836
837     u->output.tag = (ngx_buf_tag_t) &ngx_http_proxy_module;
838
839     u->conf = &plcf->upstream;
840
841 #if (NGX_HTTP_CACHE)
842     pmcf = ngx_http_get_module_main_conf(r, ngx_http_proxy_module);
843
844     u->caches = &pmcf->caches;
845     u->create_key = ngx_http_proxy_create_key;
846 #endif
847
848     u->create_request = ngx_http_proxy_create_request;
849     u->reinit_request = ngx_http_proxy_reinit_request;
850     u->process_header = ngx_http_proxy_process_status_line;
851     u->abort_request = ngx_http_proxy_abort_request;
852     u->finalize_request = ngx_http_proxy_finalize_request;
853     r->state = 0;
854
855     if (plcf->redirects) {
856         u->rewrite_redirect = ngx_http_proxy_rewrite_redirect;
857     }
858
859     if (plcf->cookie_domains || plcf->cookie_paths) {
860         u->rewrite_cookie = ngx_http_proxy_rewrite_cookie;
861     }
862
863     u->buffering = plcf->upstream.buffering;
864
865     u->pipe = ngx_palloc(r->pool, sizeof(ngx_event_pipe_t));
866     if (u->pipe == NULL) {
867         return NGX_HTTP_INTERNAL_SERVER_ERROR;
868     }
869
870     u->pipe->input_filter = ngx_http_proxy_copy_filter;
871     u->pipe->input_ctx = r;
872
873     u->input_filter_init = ngx_http_proxy_input_filter_init;
874     u->input_filter = ngx_http_proxy_non_buffered_copy_filter;
875     u->input_filter_ctx = r;
876
877     u->accel = 1;
878
879     rc = ngx_http_read_client_request_body(r, ngx_http_upstream_init);
880
881     if (rc >= NGX_HTTP_SPECIAL_RESPONSE) {
882         return rc;
883     }
884
885     return NGX_DONE;
886 }
887
888
889 static ngx_int_t
890 ngx_http_proxy_eval(ngx_http_request_t *r, ngx_http_proxy_ctx_t *ctx,
891 ngx_http_proxy_loc_conf_t *plcf)
892 {
893     u_char          *p;
894     size_t          add;
895     u_short         port;
896     ngx_str_t      proxy;
897     ngx_url_t      url;

```

```

898     ngx_http_upstream_t *u;
899
900     if (ngx_http_script_run(r, &proxy, plcf->proxy_lengths->elts, 0,
901                             plcf->proxy_values->elts)
902         == NULL)
903     {
904         return NGX_ERROR;
905     }
906
907     if (proxy.len > 7
908         && ngx_strncasecmp(proxy.data, (u_char *) "http://", 7) == 0)
909     {
910         add = 7;
911         port = 80;
912
913     #if (NGX_HTTP_SSL)
914
915     } else if (proxy.len > 8
916               && ngx_strncasecmp(proxy.data, (u_char *) "https://", 8) == 0)
917     {
918         add = 8;
919         port = 443;
920         r->upstream->ssl = 1;
921     #endif
922
923     } else {
924         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
925                     "invalid URL prefix in \"%V\"", &proxy);
926         return NGX_ERROR;
927     }
928
929     u = r->upstream;
930
931     u->schema.len = add;
932     u->schema.data = proxy.data;
933
934     ngx_memzero(&url, sizeof(ngx_url_t));
935
936     url.url.len = proxy.len - add;
937     url.url.data = proxy.data + add;
938     url.default_port = port;
939     url.uri_part = 1;
940     url.no_resolve = 1;
941
942     if (ngx_parse_url(r->pool, &url) != NGX_OK) {
943         if (url.err) {
944             ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
945                         "%s in upstream \"%V\"", url.err, &url.url);
946         }
947
948         return NGX_ERROR;
949     }
950
951     if (url.uri.len) {
952         if (url.uri.data[0] == '?') {
953             p = ngx_pnalloc(r->pool, url.uri.len + 1);
954             if (p == NULL) {
955                 return NGX_ERROR;
956             }
957
958             *p++ = '/';
959             ngx_memcpy(p, url.uri.data, url.uri.len);
960
961             url.uri.len++;
962             url.uri.data = p - 1;
963         }
964     }
965
966     ctx->vars.key_start = u->schema;
967
968     ngx_http_proxy_set_vars(&url, &ctx->vars);
969
970     u->resolved = ngx_pcalloc(r->pool, sizeof(ngx_http_upstream_resolved_t));
971     if (u->resolved == NULL) {
972         return NGX_ERROR;
973     }

```

```

974 }
975
976 if (url.addrs && url.addrs[0].sockaddr) {
977     u->resolved->sockaddr = url.addrs[0].sockaddr;
978     u->resolved->socklen = url.addrs[0].socklen;
979     u->resolved->naddrs = 1;
980     u->resolved->host = url.addrs[0].name;
981
982 } else {
983     u->resolved->host = url.host;
984     u->resolved->port = (in_port_t) (url.no_port ? port : url.port);
985     u->resolved->no_port = url.no_port;
986 }
987
988 return NGX_OK;
989 }
990
991
992 #if (NGX_HTTP_CACHE)
993
994 static ngx_int_t
995 ngx_http_proxy_create_key(ngx_http_request_t *r)
996 {
997     size_t          len, loc_len;
998     u_char          *p;
999     uintptr_t       escape;
1000     ngx_str_t       *key;
1001     ngx_http_upstream_t *u;
1002     ngx_http_proxy_ctx_t *ctx;
1003     ngx_http_proxy_loc_conf_t *plcf;
1004
1005     u = r->upstream;
1006
1007     plcf = ngx_http_get_module_loc_conf(r, ngx_http_proxy_module);
1008
1009     ctx = ngx_http_get_module_ctx(r, ngx_http_proxy_module);
1010
1011     key = ngx_array_push(&r->cache->keys);
1012     if (key == NULL) {
1013         return NGX_ERROR;
1014     }
1015
1016     if (plcf->cache_key.value.data) {
1017
1018         if (ngx_http_complex_value(r, &plcf->cache_key, key) != NGX_OK) {
1019             return NGX_ERROR;
1020         }
1021
1022         return NGX_OK;
1023     }
1024
1025     *key = ctx->vars.key_start;
1026
1027     key = ngx_array_push(&r->cache->keys);
1028     if (key == NULL) {
1029         return NGX_ERROR;
1030     }
1031
1032     if (plcf->proxy_lengths && ctx->vars.uri.len) {
1033
1034         *key = ctx->vars.uri;
1035         u->uri = ctx->vars.uri;
1036
1037         return NGX_OK;
1038     } else if (ctx->vars.uri.len == 0 && r->valid_unparsed_uri && r == r->main)
1039     {
1040
1041         *key = r->unparsed_uri;
1042         u->uri = r->unparsed_uri;
1043
1044         return NGX_OK;
1045     }
1046
1047     loc_len = (r->valid_location && ctx->vars.uri.len) ? plcf->location.len : 0;
1048
1049     if (r->quoted_uri || r->internal) {

```

```

1050         escape = 2 * ngx_escape_uri(NULL, r->uri.data + loc_len,
1051                                     r->uri.len - loc_len, NGX_ESCAPE_URI);
1052     } else {
1053         escape = 0;
1054     }
1055
1056     len = ctx->vars.uri.len + r->uri.len - loc_len + escape
1057         + sizeof("?") - 1 + r->args.len;
1058
1059     p = ngx_pnalloc(r->pool, len);
1060     if (p == NULL) {
1061         return NGX_ERROR;
1062     }
1063
1064     key->data = p;
1065
1066     if (r->valid_location) {
1067         p = ngx_copy(p, ctx->vars.uri.data, ctx->vars.uri.len);
1068     }
1069
1070     if (escape) {
1071         ngx_escape_uri(p, r->uri.data + loc_len,
1072                       r->uri.len - loc_len, NGX_ESCAPE_URI);
1073         p += r->uri.len - loc_len + escape;
1074     } else {
1075         p = ngx_copy(p, r->uri.data + loc_len, r->uri.len - loc_len);
1076     }
1077
1078     if (r->args.len > 0) {
1079         *p++ = '?';
1080         p = ngx_copy(p, r->args.data, r->args.len);
1081     }
1082
1083     key->len = p - key->data;
1084     u->uri = *key;
1085
1086     return NGX_OK;
1087 }
1088
1089 #endif
1090
1091
1092
1093 static ngx_int_t
1094 ngx_http_proxy_create_request(ngx_http_request_t *r)
1095 {
1096     size_t                len, uri_len, loc_len, body_len;
1097     uintptr_t            escape;
1098     ngx_buf_t            *b;
1099     ngx_str_t            method;
1100     ngx_uint_t           i, unparsed_uri;
1101     ngx_chain_t          *cl, *body;
1102     ngx_list_part_t     *part;
1103     ngx_table_elt_t     *header;
1104     ngx_http_upstream_t *u;
1105     ngx_http_proxy_ctx_t *ctx;
1106     ngx_http_script_code_pt code;
1107     ngx_http_proxy_headers_t *headers;
1108     ngx_http_script_engine_t e, le;
1109     ngx_http_proxy_loc_conf_t *plcf;
1110     ngx_http_script_len_code_pt lcode;
1111
1112     u = r->upstream;
1113
1114     plcf = ngx_http_get_module_loc_conf(r, ngx_http_proxy_module);
1115
1116     #if (NGX_HTTP_CACHE)
1117     headers = u->cacheable ? &plcf->headers_cache : &plcf->headers;
1118     #else
1119     headers = &plcf->headers;
1120     #endif
1121
1122     if (u->method.len) {
1123         /* HEAD was changed to GET to cache response */
1124         method = u->method;
1125         method.len++;

```

```

1126 } else if (plcf->method.len) {
1127     method = plcf->method;
1128 }
1129 } else {
1130     method = r->method_name;
1131     method.len++;
1132 }
1133 }
1134
1135 ctx = ngx_http_get_module_ctx(r, ngx_http_proxy_module);
1136
1137 if (method.len == 5
1138     && ngx_strncasecmp(method.data, (u_char *) "HEAD ", 5) == 0)
1139 {
1140     ctx->head = 1;
1141 }
1142
1143 len = method.len + sizeof(ngx_http_proxy_version) - 1 + sizeof(CRLF) - 1;
1144
1145 escape = 0;
1146 loc_len = 0;
1147 unparsed_uri = 0;
1148
1149 if (plcf->proxy_lengths && ctx->vars.uri.len) {
1150     uri_len = ctx->vars.uri.len;
1151 } else if (ctx->vars.uri.len == 0 && r->valid_unparsed_uri && r == r->main)
1152 {
1153     unparsed_uri = 1;
1154     uri_len = r->unparsed_uri.len;
1155 } else {
1156     loc_len = (r->valid_location && ctx->vars.uri.len) ?
1157         plcf->location.len : 0;
1158
1159     if (r->quoted_uri || r->space_in_uri || r->internal) {
1160         escape = 2 * ngx_escape_uri(NULL, r->uri.data + loc_len,
1161             r->uri.len - loc_len, NGX_ESCAPE_URI);
1162     }
1163
1164     uri_len = ctx->vars.uri.len + r->uri.len - loc_len + escape
1165         + sizeof("?") - 1 + r->args.len;
1166 }
1167
1168 if (uri_len == 0) {
1169     ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1170         "zero length URI to proxy");
1171     return NGX_ERROR;
1172 }
1173
1174 len += uri_len;
1175
1176 ngx_memzero(&le, sizeof(ngx_http_script_engine_t));
1177
1178 ngx_http_script_flush_no_cacheable_variables(r, plcf->body_flushes);
1179 ngx_http_script_flush_no_cacheable_variables(r, headers->flushes);
1180
1181 if (plcf->body_lengths) {
1182     le.ip = plcf->body_lengths->elts;
1183     le.request = r;
1184     le.flushed = 1;
1185     body_len = 0;
1186
1187     while (*(uintptr_t *) le.ip) {
1188         lcode = *(ngx_http_script_len_code_pt *) le.ip;
1189         body_len += lcode(&le);
1190     }
1191
1192     ctx->internal_body_length = body_len;
1193     len += body_len;
1194 } else {
1195     ctx->internal_body_length = r->headers_in.content_length_n;
1196 }
1197
1198 le.ip = headers->lengths->elts;

```



```

1202     le.request = r;
1203     le.flushed = 1;
1204
1205     while (*(uintptr_t *) le.ip) {
1206         while (*(uintptr_t *) le.ip) {
1207             lcode = *(ngx\_http\_script\_len\_code\_pt *) le.ip;
1208             len += lcode(&le);
1209         }
1210         le.ip += sizeof(uintptr_t);
1211     }
1212
1213
1214     if (plcf->upstream.pass_request_headers) {
1215         part = &r->headers_in.headers.part;
1216         header = part->elts;
1217
1218         for (i = 0; /* void */; i++) {
1219
1220             if (i >= part->nelts) {
1221                 if (part->next == NULL) {
1222                     break;
1223                 }
1224
1225                 part = part->next;
1226                 header = part->elts;
1227                 i = 0;
1228             }
1229
1230             if (ngx\_hash\_find(&headers->hash, header[i].hash,
1231                             header[i].lowercase_key, header[i].key.len))
1232             {
1233                 continue;
1234             }
1235
1236             len += header[i].key.len + sizeof(": ") - 1
1237                  + header[i].value.len + sizeof(CRLF) - 1;
1238         }
1239     }
1240
1241
1242     b = ngx\_create\_temp\_buf(r->pool, len);
1243     if (b == NULL) {
1244         return NGX\_ERROR;
1245     }
1246
1247     cl = ngx\_alloc\_chain\_link(r->pool);
1248     if (cl == NULL) {
1249         return NGX\_ERROR;
1250     }
1251
1252     cl->buf = b;
1253
1254
1255     /* the request line */
1256
1257     b->last = ngx\_copy(b->last, method.data, method.len);
1258
1259     u->uri.data = b->last;
1260
1261     if (plcf->proxy_lengths && ctx->vars.uri.len) {
1262         b->last = ngx\_copy(b->last, ctx->vars.uri.data, ctx->vars.uri.len);
1263     } else if (unparsed_uri) {
1264         b->last = ngx\_copy(b->last, r->unparsed_uri.data, r->unparsed_uri.len);
1265     } else {
1266         if (r->valid_location) {
1267             b->last = ngx\_copy(b->last, ctx->vars.uri.data, ctx->vars.uri.len);
1268         }
1269
1270         if (escape) {
1271             ngx\_escape\_uri(b->last, r->uri.data + loc_len,
1272                             r->uri.len - loc_len, NGX\_ESCAPE\_URI);
1273             b->last += r->uri.len - loc_len + escape;
1274         } else {

```

```

1278         b->last = ngx_copy(b->last, r->uri.data + loc_len,
1279                             r->uri.len - loc_len);
1280     }
1281
1282     if (r->args.len > 0) {
1283         *b->last++ = '?';
1284         b->last = ngx_copy(b->last, r->args.data, r->args.len);
1285     }
1286 }
1287
1288 u->uri.len = b->last - u->uri.data;
1289
1290 if (plcf->http_version == NGX_HTTP_VERSION_11) {
1291     b->last = ngx_cpymem(b->last, ngx_http_proxy_version_11,
1292                         sizeof(ngx_http_proxy_version_11) - 1);
1293 } else {
1294     b->last = ngx_cpymem(b->last, ngx_http_proxy_version,
1295                         sizeof(ngx_http_proxy_version) - 1);
1296 }
1297
1298 ngx_memzero(&e, sizeof(ngx_http_script_engine_t));
1299
1300 e.ip = headers->values->elts;
1301 e.pos = b->last;
1302 e.request = r;
1303 e.flushed = 1;
1304
1305 le.ip = headers->lengths->elts;
1306
1307 while (*(uintptr_t *) le.ip) {
1308     lcode = *(ngx_http_script_len_code_pt *) le.ip;
1309
1310     /* skip the header line name length */
1311     (void) lcode(&le);
1312
1313     if (*(ngx_http_script_len_code_pt *) le.ip) {
1314         for (len = 0; *(uintptr_t *) le.ip; len += lcode(&le)) {
1315             lcode = *(ngx_http_script_len_code_pt *) le.ip;
1316         }
1317
1318         e.skip = (len == sizeof(CRLF) - 1) ? 1 : 0;
1319     } else {
1320         e.skip = 0;
1321     }
1322
1323     le.ip += sizeof(uintptr_t);
1324
1325     while (*(uintptr_t *) e.ip) {
1326         code = *(ngx_http_script_code_pt *) e.ip;
1327         code((ngx_http_script_engine_t *) &e);
1328     }
1329     e.ip += sizeof(uintptr_t);
1330 }
1331
1332 b->last = e.pos;
1333
1334 if (plcf->upstream.pass_request_headers) {
1335     part = &r->headers_in.headers.part;
1336     header = part->elts;
1337
1338     for (i = 0; /* void */; i++) {
1339         if (i >= part->nelts) {
1340             if (part->next == NULL) {
1341                 break;
1342             }
1343
1344             part = part->next;
1345             header = part->elts;
1346             i = 0;
1347         }
1348     }
1349 }
1350
1351 }
1352
1353

```

```

1354     if (ngx\_hash\_find(&headers->hash, header[i].hash,
1355                     header[i].lowercase_key, header[i].key.len))
1356     {
1357         continue;
1358     }
1359
1360     b->last = ngx\_copy(b->last, header[i].key.data, header[i].key.len);
1361
1362     *b->last++ = ':'; *b->last++ = ' ';
1363
1364     b->last = ngx\_copy(b->last, header[i].value.data,
1365                     header[i].value.len);
1366
1367     *b->last++ = CR; *b->last++ = LF;
1368
1369     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
1370                 "http proxy header: \"%V: %V\"",
1371                 &header[i].key, &header[i].value);
1372 }
1373 }
1374
1375 /* add "\r\n" at the header end */
1376 *b->last++ = CR; *b->last++ = LF;
1377
1378 if (plcf->body_values) {
1379     e.ip = plcf->body_values->elts;
1380     e.pos = b->last;
1381
1382     while (*(uintptr\_t *) e.ip) {
1383         code = *(ngx\_http\_script\_code\_pt *) e.ip;
1384         code((ngx\_http\_script\_engine\_t *) &e);
1385     }
1386
1387     b->last = e.pos;
1388 }
1389
1390 ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
1391                 "http proxy header:%N\"%*s\"",
1392                 (size_t) (b->last - b->pos), b->pos);
1393
1394 if (plcf->body_values == NULL && plcf->upstream.pass_request_body) {
1395
1396     body = u->request_bufs;
1397     u->request_bufs = cl;
1398
1399     while (body) {
1400         b = ngx\_alloc\_buf(r->pool);
1401         if (b == NULL) {
1402             return NGX\_ERROR;
1403         }
1404
1405         ngx\_memcpy(b, body->buf, sizeof(ngx\_buf\_t));
1406
1407         cl->next = ngx\_alloc\_chain\_link(r->pool);
1408         if (cl->next == NULL) {
1409             return NGX\_ERROR;
1410         }
1411
1412         cl = cl->next;
1413         cl->buf = b;
1414
1415         body = body->next;
1416     }
1417
1418 } else {
1419     u->request_bufs = cl;
1420 }
1421
1422 b->flush = 1;
1423 cl->next = NULL;
1424
1425 return NGX\_OK;
1426 }
1427 }
1428
1429

```

```

1430 static ngx_int_t
1431 ngx_http_proxy_reinit_request(ngx_http_request_t *r)
1432 {
1433     ngx_http_proxy_ctx_t *ctx;
1434
1435     ctx = ngx_http_get_module_ctx(r, ngx_http_proxy_module);
1436
1437     if (ctx == NULL) {
1438         return NGX_OK;
1439     }
1440
1441     ctx->status.code = 0;
1442     ctx->status.count = 0;
1443     ctx->status.start = NULL;
1444     ctx->status.end = NULL;
1445     ctx->chunked.state = 0;
1446
1447     r->upstream->process_header = ngx_http_proxy_process_status_line;
1448     r->upstream->pipe->input_filter = ngx_http_proxy_copy_filter;
1449     r->upstream->input_filter = ngx_http_proxy_non_buffered_copy_filter;
1450     r->state = 0;
1451
1452     return NGX_OK;
1453 }
1454
1455
1456 static ngx_int_t
1457 ngx_http_proxy_process_status_line(ngx_http_request_t *r)
1458 {
1459     size_t len;
1460     ngx_int_t rc;
1461     ngx_http_upstream_t *u;
1462     ngx_http_proxy_ctx_t *ctx;
1463
1464     ctx = ngx_http_get_module_ctx(r, ngx_http_proxy_module);
1465
1466     if (ctx == NULL) {
1467         return NGX_ERROR;
1468     }
1469
1470     u = r->upstream;
1471
1472     rc = ngx_http_parse_status_line(r, &u->buffer, &ctx->status);
1473
1474     if (rc == NGX_AGAIN) {
1475         return rc;
1476     }
1477
1478     if (rc == NGX_ERROR) {
1479
1480 #if (NGX_HTTP_CACHE)
1481
1482         if (r->cache) {
1483             r->http_version = NGX_HTTP_VERSION_9;
1484             return NGX_OK;
1485         }
1486
1487 #endif
1488
1489         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1490             "upstream sent no valid HTTP/1.0 header");
1491
1492 #if 0
1493         if (u->accel) {
1494             return NGX_HTTP_UPSTREAM_INVALID_HEADER;
1495         }
1496 #endif
1497
1498         r->http_version = NGX_HTTP_VERSION_9;
1499         u->state->status = NGX_HTTP_OK;
1500         u->headers_in.connection_close = 1;
1501
1502         return NGX_OK;
1503     }
1504
1505     if (u->state && u->state->status == 0) {

```

```

1506     u->state->status = ctx->status.code;
1507 }
1508
1509 u->headers_in.status_n = ctx->status.code;
1510
1511 len = ctx->status.end - ctx->status.start;
1512 u->headers_in.status_line.len = len;
1513
1514 u->headers_in.status_line.data = ngx_pnalloc(r->pool, len);
1515 if (u->headers_in.status_line.data == NULL) {
1516     return NGX_ERROR;
1517 }
1518
1519 ngx_memcpy(u->headers_in.status_line.data, ctx->status.start, len);
1520
1521 ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1522     "http proxy status %ui \"%V\"",
1523     u->headers_in.status_n, &u->headers_in.status_line);
1524
1525 if (ctx->status.http_version < NGX_HTTP_VERSION_11) {
1526     u->headers_in.connection_close = 1;
1527 }
1528
1529 u->process_header = ngx_http_proxy_process_header;
1530
1531 return ngx_http_proxy_process_header(r);
1532 }
1533
1534
1535 static ngx_int_t
1536 ngx_http_proxy_process_header(ngx_http_request_t *r)
1537 {
1538     ngx_int_t          rc;
1539     ngx_table_elt_t   *h;
1540     ngx_http_upstream_t *u;
1541     ngx_http_proxy_ctx_t *ctx;
1542     ngx_http_upstream_header_t *hh;
1543     ngx_http_upstream_main_conf_t *umcf;
1544
1545     umcf = ngx_http_get_module_main_conf(r, ngx_http_upstream_module);
1546
1547     for ( ;; ) {
1548
1549         rc = ngx_http_parse_header_line(r, &r->upstream->buffer, 1);
1550
1551         if (rc == NGX_OK) {
1552
1553             /* a header line has been parsed successfully */
1554
1555             h = ngx_list_push(&r->upstream->headers_in.headers);
1556             if (h == NULL) {
1557                 return NGX_ERROR;
1558             }
1559
1560             h->hash = r->header_hash;
1561
1562             h->key.len = r->header_name_end - r->header_name_start;
1563             h->value.len = r->header_end - r->header_start;
1564
1565             h->key.data = ngx_pnalloc(r->pool,
1566                 h->key.len + 1 + h->value.len + 1 + h->key.len);
1567             if (h->key.data == NULL) {
1568                 return NGX_ERROR;
1569             }
1570
1571             h->value.data = h->key.data + h->key.len + 1;
1572             h->lowercase_key = h->key.data + h->key.len + 1 + h->value.len + 1;
1573
1574             ngx_memcpy(h->key.data, r->header_name_start, h->key.len);
1575             h->key.data[h->key.len] = '\0';
1576             ngx_memcpy(h->value.data, r->header_start, h->value.len);
1577             h->value.data[h->value.len] = '\0';
1578
1579             if (h->key.len == r->lowercase_index) {
1580                 ngx_memcpy(h->lowercase_key, r->lowercase_header, h->key.len);

```

```

1582     } else {
1583         ngx_strlow(h->lowercase_key, h->key.data, h->key.len);
1584     }
1585
1586     hh = ngx_hash_find(&umcf->headers_in_hash, h->hash,
1587                     h->lowercase_key, h->key.len);
1588
1589     if (hh && hh->handler(r, h, hh->offset) != NGX_OK) {
1590         return NGX_ERROR;
1591     }
1592
1593     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1594                 "http proxy header: \"%V: %V\"",
1595                 &h->key, &h->value);
1596
1597     continue;
1598 }
1599
1600 if (rc == NGX_HTTP_PARSE_HEADER_DONE) {
1601     /* a whole header has been parsed successfully */
1602
1603     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1604                 "http proxy header done");
1605
1606     /*
1607     * if no "Server" and "Date" in header line,
1608     * then add the special empty headers
1609     */
1610
1611     if (r->upstream->headers_in.server == NULL) {
1612         h = ngx_list_push(&r->upstream->headers_in.headers);
1613         if (h == NULL) {
1614             return NGX_ERROR;
1615         }
1616
1617         h->hash = ngx_hash(ngx_hash(ngx_hash(ngx_hash(
1618             ngx_hash('s', 'e'), 'r'), 'v'), 'e'), 'r');
1619
1620         ngx_str_set(&h->key, "Server");
1621         ngx_str_null(&h->value);
1622         h->lowercase_key = (u_char *) "server";
1623     }
1624
1625     if (r->upstream->headers_in.date == NULL) {
1626         h = ngx_list_push(&r->upstream->headers_in.headers);
1627         if (h == NULL) {
1628             return NGX_ERROR;
1629         }
1630
1631         h->hash = ngx_hash(ngx_hash(ngx_hash('d', 'a'), 't'), 'e');
1632
1633         ngx_str_set(&h->key, "Date");
1634         ngx_str_null(&h->value);
1635         h->lowercase_key = (u_char *) "date";
1636     }
1637
1638     /* clear content length if response is chunked */
1639
1640     u = r->upstream;
1641
1642     if (u->headers_in.chunked) {
1643         u->headers_in.content_length_n = -1;
1644     }
1645
1646     /*
1647     * set u->keepalive if response has no body; this allows to keep
1648     * connections alive in case of r->header_only or X-Accel-Redirect
1649     */
1650
1651     ctx = ngx_http_get_module_ctx(r, ngx_http_proxy_module);
1652
1653     if (u->headers_in.status_n == NGX_HTTP_NO_CONTENT
1654         || u->headers_in.status_n == NGX_HTTP_NOT_MODIFIED
1655         || ctx->head
1656         || (!u->headers_in.chunked

```

```

1658         && u->headers_in.content_length_n == 0))
1659     {
1660         u->keepalive = !u->headers_in.connection_close;
1661     }
1662
1663     if (u->headers_in.status_n == NGX\_HTTP\_SWITCHING\_PROTOCOLS) {
1664         u->keepalive = 0;
1665
1666         if (r->headers_in.upgrade) {
1667             u->upgrade = 1;
1668         }
1669     }
1670
1671     return NGX\_OK;
1672 }
1673
1674 if (rc == NGX\_AGAIN) {
1675     return NGX\_AGAIN;
1676 }
1677
1678 /* there was error while a header line parsing */
1679
1680 ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, 0,
1681             "upstream sent invalid header");
1682
1683     return NGX\_HTTP\_UPSTREAM\_INVALID\_HEADER;
1684 }
1685 }
1686
1687
1688 static ngx\_int\_t
1689 ngx\_http\_proxy\_input\_filter\_init(void *data)
1690 {
1691     ngx\_http\_request\_t *r = data;
1692     ngx\_http\_upstream\_t *u;
1693     ngx\_http\_proxy\_ctx\_t *ctx;
1694
1695     u = r->upstream;
1696     ctx = ngx\_http\_get\_module\_ctx(r, ngx\_http\_proxy\_module);
1697
1698     if (ctx == NULL) {
1699         return NGX\_ERROR;
1700     }
1701
1702     ngx\_log\_debug4(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
1703                 "http proxy filter init s:%d h:%d c:%d l:%0",
1704                 u->headers_in.status_n, ctx->head, u->headers_in.chunked,
1705                 u->headers_in.content_length_n);
1706
1707     /* as per RFC2616, 4.4 Message Length */
1708
1709     if (u->headers_in.status_n == NGX\_HTTP\_NO\_CONTENT
1710         || u->headers_in.status_n == NGX\_HTTP\_NOT\_MODIFIED
1711         || ctx->head)
1712     {
1713         /* 1xx, 204, and 304 and replies to HEAD requests */
1714         /* no 1xx since we don't send Expect and Upgrade */
1715
1716         u->pipe->length = 0;
1717         u->length = 0;
1718         u->keepalive = !u->headers_in.connection_close;
1719
1720     } else if (u->headers_in.chunked) {
1721         /* chunked */
1722
1723         u->pipe->input_filter = ngx\_http\_proxy\_chunked\_filter;
1724         u->pipe->length = 3; /* "0" LF LF */
1725
1726         u->input_filter = ngx\_http\_proxy\_non\_buffered\_chunked\_filter;
1727         u->length = 1;
1728
1729     } else if (u->headers_in.content_length_n == 0) {
1730         /* empty body: special case as filter won't be called */
1731
1732         u->pipe->length = 0;
1733         u->length = 0;

```

```

1734         u->keepalive = !u->headers_in.connection_close;
1735
1736     } else {
1737         /* content length or connection close */
1738
1739         u->pipe->length = u->headers_in.content_length_n;
1740         u->length = u->headers_in.content_length_n;
1741     }
1742
1743     return NGX_OK;
1744 }
1745
1746
1747 static ngx_int_t
1748 ngx_http_proxy_copy_filter(ngx_event_pipe_t *p, ngx_buf_t *buf)
1749 {
1750     ngx_buf_t      *b;
1751     ngx_chain_t     *cl;
1752     ngx_http_request_t *r;
1753
1754     if (buf->pos == buf->last) {
1755         return NGX_OK;
1756     }
1757
1758     cl = ngx_chain_get_free_buf(p->pool, &p->free);
1759     if (cl == NULL) {
1760         return NGX_ERROR;
1761     }
1762
1763     b = cl->buf;
1764
1765     ngx_memcpy(b, buf, sizeof(ngx_buf_t));
1766     b->shadow = buf;
1767     b->tag = p->tag;
1768     b->last_shadow = 1;
1769     b->recycled = 1;
1770     buf->shadow = b;
1771
1772     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, p->log, 0, "input buf #%d", b->num);
1773
1774     if (p->in) {
1775         *p->last_in = cl;
1776     } else {
1777         p->in = cl;
1778     }
1779     p->last_in = &cl->next;
1780
1781     if (p->length == -1) {
1782         return NGX_OK;
1783     }
1784
1785     p->length -= b->last - b->pos;
1786
1787     if (p->length == 0) {
1788         r = p->input_ctx;
1789         p->upstream_done = 1;
1790         r->upstream->keepalive = !r->upstream->headers_in.connection_close;
1791     }
1792     else if (p->length < 0) {
1793         r = p->input_ctx;
1794         p->upstream_done = 1;
1795
1796         ngx_log_error(NGX_LOG_WARN, r->connection->log, 0,
1797             "upstream sent more data than specified in "
1798             "\"Content-Length\" header");
1799     }
1800
1801     return NGX_OK;
1802 }
1803
1804
1805 static ngx_int_t
1806 ngx_http_proxy_chunked_filter(ngx_event_pipe_t *p, ngx_buf_t *buf)
1807 {
1808     ngx_int_t      rc;
1809     ngx_buf_t      *b, **prev;

```



```

1810     ngx_chain_t      *cl;
1811     ngx_http_request_t *r;
1812     ngx_http_proxy_ctx_t *ctx;
1813
1814     if (buf->pos == buf->last) {
1815         return NGX_OK;
1816     }
1817
1818     r = p->input_ctx;
1819     ctx = ngx_http_get_module_ctx(r, ngx_http_proxy_module);
1820
1821     if (ctx == NULL) {
1822         return NGX_ERROR;
1823     }
1824
1825     b = NULL;
1826     prev = &buf->shadow;
1827
1828     for ( ;; ) {
1829
1830         rc = ngx_http_parse_chunked(r, buf, &ctx->chunked);
1831
1832         if (rc == NGX_OK) {
1833
1834             /* a chunk has been parsed successfully */
1835
1836             cl = ngx_chain_get_free_buf(p->pool, &p->free);
1837             if (cl == NULL) {
1838                 return NGX_ERROR;
1839             }
1840
1841             b = cl->buf;
1842
1843             ngx_memzero(b, sizeof(ngx_buf_t));
1844
1845             b->pos = buf->pos;
1846             b->start = buf->start;
1847             b->end = buf->end;
1848             b->tag = p->tag;
1849             b->temporary = 1;
1850             b->recycled = 1;
1851
1852             *prev = b;
1853             prev = &b->shadow;
1854
1855             if (p->in) {
1856                 *p->last_in = cl;
1857             } else {
1858                 p->in = cl;
1859             }
1860             p->last_in = &cl->next;
1861
1862             /* STUB */ b->num = buf->num;
1863
1864             ngx_log_debug2(NGX_LOG_DEBUG_EVENT, p->log, 0,
1865                 "input buf #%d %p", b->num, b->pos);
1866
1867             if (buf->last - buf->pos >= ctx->chunked.size) {
1868
1869                 buf->pos += (size_t) ctx->chunked.size;
1870                 b->last = buf->pos;
1871                 ctx->chunked.size = 0;
1872
1873                 continue;
1874             }
1875
1876             ctx->chunked.size -= buf->last - buf->pos;
1877             buf->pos = buf->last;
1878             b->last = buf->last;
1879
1880             continue;
1881         }
1882
1883         if (rc == NGX_DONE) {
1884
1885             /* a whole response has been parsed successfully */

```

```

1886         p->upstream_done = 1;
1887         r->upstream->keepalive = !r->upstream->headers_in.connection_close;
1888
1889     }
1890     break;
1891 }
1892
1893 if (rc == NGX\_AGAIN) {
1894     /* set p->length, minimal amount of data we want to see */
1895     p->length = ctx->chunked.length;
1896     break;
1897 }
1898
1899 /* invalid response */
1900
1901 ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, 0,
1902     "upstream sent invalid chunked response");
1903
1904 return NGX\_ERROR;
1905 }
1906
1907 ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
1908     "http proxy chunked state %d, length %d",
1909     ctx->chunked.state, p->length);
1910
1911 if (b) {
1912     b->shadow = buf;
1913     b->last_shadow = 1;
1914
1915     ngx\_log\_debug2(NGX\_LOG\_DEBUG\_EVENT, p->log, 0,
1916         "input buf %p %z", b->pos, b->last - b->pos);
1917
1918     return NGX\_OK;
1919 }
1920
1921 /* there is no data record in the buf, add it to free chain */
1922
1923 if (ngx\_event\_pipe\_add\_free\_buf(p, buf) != NGX\_OK) {
1924     return NGX\_ERROR;
1925 }
1926
1927 return NGX\_OK;
1928 }
1929
1930 static ngx\_int\_t
1931 ngx\_http\_proxy\_non\_buffered\_copy\_filter(void *data, ssize\_t bytes)
1932 {
1933     ngx\_http\_request\_t *r = data;
1934
1935     ngx\_buf\_t *b;
1936     ngx\_chain\_t *cl, **ll;
1937     ngx\_http\_upstream\_t *u;
1938
1939     u = r->upstream;
1940
1941     for (cl = u->out_bufs, ll = &u->out_bufs; cl; cl = cl->next) {
1942         ll = &cl->next;
1943     }
1944
1945     cl = ngx\_chain\_get\_free\_buf(r->pool, &u->free_bufs);
1946     if (cl == NULL) {
1947         return NGX\_ERROR;
1948     }
1949
1950     *ll = cl;
1951
1952     cl->buf->flush = 1;
1953     cl->buf->memory = 1;
1954
1955     b = &u->buffer;
1956
1957     cl->buf->pos = b->last;

```

```

1962     b->last += bytes;
1963     cl->buf->last = b->last;
1964     cl->buf->tag = u->output.tag;
1965
1966     if (u->length == -1) {
1967         return NGX\_OK;
1968     }
1969
1970     u->length -= bytes;
1971
1972     if (u->length == 0) {
1973         u->keepalive = !u->headers_in.connection_close;
1974     }
1975
1976     return NGX\_OK;
1977 }
1978
1979
1980 static ngx\_int\_t
1981 ngx\_http\_proxy\_non\_buffered\_chunked\_filter(void *data, ssize\_t bytes)
1982 {
1983     ngx\_http\_request\_t *r = data;
1984
1985     ngx\_int\_t rc;
1986     ngx\_buf\_t *b, *buf;
1987     ngx\_chain\_t *cl, **ll;
1988     ngx\_http\_upstream\_t *u;
1989     ngx\_http\_proxy\_ctx\_t *ctx;
1990
1991     ctx = ngx\_http\_get\_module\_ctx(r, ngx\_http\_proxy\_module);
1992
1993     if (ctx == NULL) {
1994         return NGX\_ERROR;
1995     }
1996
1997     u = r->upstream;
1998     buf = &u->buffer;
1999
2000     buf->pos = buf->last;
2001     buf->last += bytes;
2002
2003     for (cl = u->out_bufs, ll = &u->out_bufs; cl; cl = cl->next) {
2004         ll = &cl->next;
2005     }
2006
2007     for ( ;; ) {
2008
2009         rc = ngx\_http\_parse\_chunked(r, buf, &ctx->chunked);
2010
2011         if (rc == NGX\_OK) {
2012
2013             /* a chunk has been parsed successfully */
2014
2015             cl = ngx\_chain\_get\_free\_buf(r->pool, &u->free_bufs);
2016             if (cl == NULL) {
2017                 return NGX\_ERROR;
2018             }
2019
2020             *ll = cl;
2021             ll = &cl->next;
2022
2023             b = cl->buf;
2024
2025             b->flush = 1;
2026             b->memory = 1;
2027
2028             b->pos = buf->pos;
2029             b->tag = u->output.tag;
2030
2031             if (buf->last - buf->pos >= ctx->chunked.size) {
2032                 buf->pos += (size\_t) ctx->chunked.size;
2033                 b->last = buf->pos;
2034                 ctx->chunked.size = 0;
2035
2036             } else {
2037                 ctx->chunked.size -= buf->last - buf->pos;

```

```

2038         buf->pos = buf->last;
2039         b->last = buf->last;
2040     }
2041
2042     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
2043                 "http proxy out buf %p %z",
2044                 b->pos, b->last - b->pos);
2045
2046     continue;
2047 }
2048
2049 if (rc == NGX_DONE) {
2050
2051     /* a whole response has been parsed successfully */
2052
2053     u->keepalive = !u->headers_in.connection_close;
2054     u->length = 0;
2055
2056     break;
2057 }
2058
2059 if (rc == NGX_AGAIN) {
2060     break;
2061 }
2062
2063 /* invalid response */
2064
2065 ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
2066             "upstream sent invalid chunked response");
2067
2068 return NGX_ERROR;
2069 }
2070
2071 /* provide continuous buffer for subrequests in memory */
2072
2073 if (r->subrequest_in_memory) {
2074
2075     cl = u->out_bufs;
2076
2077     if (cl) {
2078         buf->pos = cl->buf->pos;
2079     }
2080
2081     buf->last = buf->pos;
2082
2083     for (cl = u->out_bufs; cl; cl = cl->next) {
2084         ngx_log_debug3(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
2085                     "http proxy in memory %p-%p %uz",
2086                     cl->buf->pos, cl->buf->last, ngx_buf_size(cl->buf));
2087
2088         if (buf->last == cl->buf->pos) {
2089             buf->last = cl->buf->last;
2090             continue;
2091         }
2092
2093         buf->last = ngx_movemem(buf->last, cl->buf->pos,
2094                               cl->buf->last - cl->buf->pos);
2095
2096         cl->buf->pos = buf->last - (cl->buf->last - cl->buf->pos);
2097         cl->buf->last = buf->last;
2098     }
2099 }
2100
2101 return NGX_OK;
2102 }
2103
2104
2105 static void
2106 ngx_http_proxy_abort_request(ngx_http_request_t *r)
2107 {
2108     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
2109                 "abort http proxy request");
2110
2111     return;
2112 }
2113

```

```

2114 static void
2115 ngx_http_proxy_finalize_request(ngx_http_request_t *r, ngx_int_t rc)
2116 {
2117     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
2118                 "finalize http proxy request");
2119
2120     return;
2121 }
2122
2123
2124 static ngx_int_t
2125 ngx_http_proxy_host_variable(ngx_http_request_t *r,
2126     ngx_http_variable_value_t *v, uintptr_t data)
2127 {
2128     ngx_http_proxy_ctx_t *ctx;
2129
2130     ctx = ngx_http_get_module_ctx(r, ngx_http_proxy_module);
2131
2132     if (ctx == NULL) {
2133         v->not_found = 1;
2134         return NGX_OK;
2135     }
2136
2137     v->len = ctx->vars.host_header.len;
2138     v->valid = 1;
2139     v->no_cacheable = 0;
2140     v->not_found = 0;
2141     v->data = ctx->vars.host_header.data;
2142
2143     return NGX_OK;
2144 }
2145
2146
2147 static ngx_int_t
2148 ngx_http_proxy_port_variable(ngx_http_request_t *r,
2149     ngx_http_variable_value_t *v, uintptr_t data)
2150 {
2151     ngx_http_proxy_ctx_t *ctx;
2152
2153     ctx = ngx_http_get_module_ctx(r, ngx_http_proxy_module);
2154
2155     if (ctx == NULL) {
2156         v->not_found = 1;
2157         return NGX_OK;
2158     }
2159
2160     v->len = ctx->vars.port.len;
2161     v->valid = 1;
2162     v->no_cacheable = 0;
2163     v->not_found = 0;
2164     v->data = ctx->vars.port.data;
2165
2166     return NGX_OK;
2167 }
2168
2169
2170 static ngx_int_t
2171 ngx_http_proxy_add_x_forwarded_for_variable(ngx_http_request_t *r,
2172     ngx_http_variable_value_t *v, uintptr_t data)
2173 {
2174     size_t len;
2175     u_char *p;
2176     ngx_uint_t i, n;
2177     ngx_table_elt_t **h;
2178
2179     v->valid = 1;
2180     v->no_cacheable = 0;
2181     v->not_found = 0;
2182
2183     n = r->headers_in.x_forwarded_for.nelts;
2184     h = r->headers_in.x_forwarded_for.elts;
2185
2186     len = 0;
2187
2188     for (i = 0; i < n; i++) {

```

```

2190     len += h[i]->value.len + sizeof(", ") - 1;
2191 }
2192
2193 if (len == 0) {
2194     v->len = r->connection->addr_text.len;
2195     v->data = r->connection->addr_text.data;
2196     return NGX\_OK;
2197 }
2198
2199 len += r->connection->addr_text.len;
2200
2201 p = ngx\_pnalloc(r->pool, len);
2202 if (p == NULL) {
2203     return NGX\_ERROR;
2204 }
2205
2206 v->len = len;
2207 v->data = p;
2208
2209 for (i = 0; i < n; i++) {
2210     p = ngx\_copy(p, h[i]->value.data, h[i]->value.len);
2211     *p++ = ','; *p++ = ' ';
2212 }
2213
2214 ngx\_memcpy(p, r->connection->addr_text.data, r->connection->addr_text.len);
2215
2216 return NGX\_OK;
2217 }
2218
2219
2220 static ngx\_int\_t
2221 ngx\_http\_proxy\_internal\_body\_length\_variable(ngx\_http\_request\_t *r,
2222 ngx\_http\_variable\_value\_t *v, uintptr\_t data)
2223 {
2224     ngx\_http\_proxy\_ctx\_t *ctx;
2225
2226     ctx = ngx\_http\_get\_module\_ctx(r, ngx\_http\_proxy\_module);
2227
2228     if (ctx == NULL || ctx->internal_body_length < 0) {
2229         v->not_found = 1;
2230         return NGX\_OK;
2231     }
2232
2233     v->valid = 1;
2234     v->no_cacheable = 0;
2235     v->not_found = 0;
2236
2237     v->data = ngx\_pnalloc(r->pool, NGX\_OFF\_T\_LEN);
2238
2239     if (v->data == NULL) {
2240         return NGX\_ERROR;
2241     }
2242
2243     v->len = ngx\_sprintf(v->data, "%O", ctx->internal_body_length) - v->data;
2244
2245     return NGX\_OK;
2246 }
2247
2248
2249 static ngx\_int\_t
2250 ngx\_http\_proxy\_rewrite\_redirect(ngx\_http\_request\_t *r, ngx\_table\_elt\_t *h,
2251 size\_t prefix)
2252 {
2253     size\_t len;
2254     ngx\_int\_t rc;
2255     ngx\_uint\_t i;
2256     ngx\_http\_proxy\_rewrite\_t *pr;
2257     ngx\_http\_proxy\_loc\_conf\_t *plcf;
2258
2259     plcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_proxy\_module);
2260
2261     pr = plcf->redirects->elts;
2262
2263     if (pr == NULL) {
2264         return NGX\_DECLINED;
2265     }

```

```

2266     len = h->value.len - prefix;
2267
2268
2269     for (i = 0; i < plcf->redirects->nelts; i++) {
2270         rc = pr[i].handler(r, h, prefix, len, &pr[i]);
2271
2272         if (rc != NGX\_DECLINED) {
2273             return rc;
2274         }
2275     }
2276
2277     return NGX\_DECLINED;
2278 }
2279
2280
2281 static ngx\_int\_t
2282 ngx\_http\_proxy\_rewrite\_cookie(ngx\_http\_request\_t *r, ngx\_table\_elt\_t *h)
2283 {
2284     size\_t                prefix;
2285     u\_char                *p;
2286     ngx\_int\_t            rc, rv;
2287     ngx\_http\_proxy\_loc\_conf\_t *plcf;
2288
2289     p = (u\_char *) ngx\_strchr(h->value.data, ';');
2290     if (p == NULL) {
2291         return NGX\_DECLINED;
2292     }
2293
2294     prefix = p + 1 - h->value.data;
2295
2296     rv = NGX\_DECLINED;
2297
2298     plcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_proxy\_module);
2299
2300     if (plcf->cookie_domains) {
2301         p = ngx\_strcasestrn(h->value.data + prefix, "domain=", 7 - 1);
2302
2303         if (p) {
2304             rc = ngx\_http\_proxy\_rewrite\_cookie\_value(r, h, p + 7,
2305                                                       plcf->cookie_domains);
2306             if (rc == NGX\_ERROR) {
2307                 return NGX\_ERROR;
2308             }
2309
2310             if (rc != NGX\_DECLINED) {
2311                 rv = rc;
2312             }
2313         }
2314     }
2315
2316     if (plcf->cookie_paths) {
2317         p = ngx\_strcasestrn(h->value.data + prefix, "path=", 5 - 1);
2318
2319         if (p) {
2320             rc = ngx\_http\_proxy\_rewrite\_cookie\_value(r, h, p + 5,
2321                                                       plcf->cookie_paths);
2322             if (rc == NGX\_ERROR) {
2323                 return NGX\_ERROR;
2324             }
2325
2326             if (rc != NGX\_DECLINED) {
2327                 rv = rc;
2328             }
2329         }
2330     }
2331
2332     return rv;
2333 }
2334
2335
2336 static ngx\_int\_t
2337 ngx\_http\_proxy\_rewrite\_cookie\_value(ngx\_http\_request\_t *r, ngx\_table\_elt\_t *h,
2338 u\_char *value, ngx\_array\_t *rewrites)
2339 {
2340     size\_t                len, prefix;
2341     u\_char                *p;

```

```

2342     ngx_int_t         rc;
2343     ngx_uint_t        i;
2344     ngx_http_proxy_rewrite_t *pr;
2345
2346     prefix = value - h->value.data;
2347
2348     p = (u_char *) ngx_strchr(value, ';');
2349
2350     len = p ? (size_t) (p - value) : (h->value.len - prefix);
2351
2352     pr = rewrites->elts;
2353
2354     for (i = 0; i < rewrites->nelts; i++) {
2355         rc = pr[i].handler(r, h, prefix, len, &pr[i]);
2356
2357         if (rc != NGX_DECLINED) {
2358             return rc;
2359         }
2360     }
2361
2362     return NGX_DECLINED;
2363 }
2364
2365
2366 static ngx_int_t
2367 ngx_http_proxy_rewrite_complex_handler(ngx_http_request_t *r,
2368     ngx_table_elt_t *h, size_t prefix, size_t len, ngx_http_proxy_rewrite_t *pr)
2369 {
2370     ngx_str_t  pattern, replacement;
2371
2372     if (ngx_http_complex_value(r, &pr->pattern.complex, &pattern) != NGX_OK) {
2373         return NGX_ERROR;
2374     }
2375
2376     if (pattern.len > len
2377         || ngx_rstrncmp(h->value.data + prefix, pattern.data,
2378             pattern.len) != 0)
2379     {
2380         return NGX_DECLINED;
2381     }
2382
2383     if (ngx_http_complex_value(r, &pr->replacement, &replacement) != NGX_OK) {
2384         return NGX_ERROR;
2385     }
2386
2387     return ngx_http_proxy_rewrite(r, h, prefix, pattern.len, &replacement);
2388 }
2389
2390
2391 #if (NGX_PCRE)
2392
2393 static ngx_int_t
2394 ngx_http_proxy_rewrite_regex_handler(ngx_http_request_t *r, ngx_table_elt_t *h,
2395     size_t prefix, size_t len, ngx_http_proxy_rewrite_t *pr)
2396 {
2397     ngx_str_t  pattern, replacement;
2398
2399     pattern.len = len;
2400     pattern.data = h->value.data + prefix;
2401
2402     if (ngx_http_regex_exec(r, pr->pattern.regex, &pattern) != NGX_OK) {
2403         return NGX_DECLINED;
2404     }
2405
2406     if (ngx_http_complex_value(r, &pr->replacement, &replacement) != NGX_OK) {
2407         return NGX_ERROR;
2408     }
2409
2410     if (prefix == 0 && h->value.len == len) {
2411         h->value = replacement;
2412         return NGX_OK;
2413     }
2414
2415     return ngx_http_proxy_rewrite(r, h, prefix, len, &replacement);
2416 }
2417

```



```

2418 #endif
2419
2420
2421 static ngx_int_t
2422 ngx_http_proxy_rewrite_domain_handler(ngx_http_request_t *r,
2423     ngx_table_elt_t *h, size_t prefix, size_t len, ngx_http_proxy_rewrite_t *pr)
2424 {
2425     u_char      *p;
2426     ngx_str_t    pattern, replacement;
2427
2428     if (ngx_http_complex_value(r, &pr->pattern.complex, &pattern) != NGX_OK) {
2429         return NGX_ERROR;
2430     }
2431
2432     p = h->value.data + prefix;
2433
2434     if (p[0] == '.') {
2435         p++;
2436         prefix++;
2437         len--;
2438     }
2439
2440     if (pattern.len != len || ngx_rstrncasecmp(pattern.data, p, len) != 0) {
2441         return NGX_DECLINED;
2442     }
2443
2444     if (ngx_http_complex_value(r, &pr->replacement, &replacement) != NGX_OK) {
2445         return NGX_ERROR;
2446     }
2447
2448     return ngx_http_proxy_rewrite(r, h, prefix, len, &replacement);
2449 }
2450
2451
2452 static ngx_int_t
2453 ngx_http_proxy_rewrite(ngx_http_request_t *r, ngx_table_elt_t *h, size_t prefix,
2454     size_t len, ngx_str_t *replacement)
2455 {
2456     u_char      *p, *data;
2457     size_t      new_len;
2458
2459     new_len = replacement->len + h->value.len - len;
2460
2461     if (replacement->len > len) {
2462
2463         data = ngx_pnalloc(r->pool, new_len + 1);
2464         if (data == NULL) {
2465             return NGX_ERROR;
2466         }
2467
2468         p = ngx_copy(data, h->value.data, prefix);
2469         p = ngx_copy(p, replacement->data, replacement->len);
2470
2471         ngx_memcpy(p, h->value.data + prefix + len,
2472             h->value.len - len - prefix + 1);
2473
2474         h->value.data = data;
2475
2476     } else {
2477         p = ngx_copy(h->value.data + prefix, replacement->data,
2478             replacement->len);
2479
2480         ngx_memmove(p, h->value.data + prefix + len,
2481             h->value.len - len - prefix + 1);
2482     }
2483
2484     h->value.len = new_len;
2485
2486     return NGX_OK;
2487 }
2488
2489
2490 static ngx_int_t
2491 ngx_http_proxy_add_variables(ngx_conf_t *cf)
2492 {
2493     ngx_http_variable_t *var, *v;

```

```

2494     for (v = ngx_http_proxy_vars; v->name.len; v++) {
2495         var = ngx_http_add_variable(cf, &v->name, v->flags);
2496         if (var == NULL) {
2497             return NGX_ERROR;
2498         }
2499     }
2500
2501     var->get_handler = v->get_handler;
2502     var->data = v->data;
2503 }
2504
2505 return NGX_OK;
2506 }
2507
2508
2509 static void *
2510 ngx_http_proxy_create_main_conf(ngx_conf_t *cf)
2511 {
2512     ngx_http_proxy_main_conf_t *conf;
2513
2514     conf = ngx_palloc(cf->pool, sizeof(ngx_http_proxy_main_conf_t));
2515     if (conf == NULL) {
2516         return NULL;
2517     }
2518
2519 #if (NGX_HTTP_CACHE)
2520     if (ngx_array_init(&conf->caches, cf->pool, 4,
2521         sizeof(ngx_http_file_cache_t *))
2522         != NGX_OK)
2523     {
2524         return NULL;
2525     }
2526 #endif
2527
2528     return conf;
2529 }
2530
2531
2532 static void *
2533 ngx_http_proxy_create_loc_conf(ngx_conf_t *cf)
2534 {
2535     ngx_http_proxy_loc_conf_t *conf;
2536
2537     conf = ngx_palloc(cf->pool, sizeof(ngx_http_proxy_loc_conf_t));
2538     if (conf == NULL) {
2539         return NULL;
2540     }
2541
2542     /*
2543      * set by ngx_palloc():
2544      *
2545      *     conf->upstream.bufs.num = 0;
2546      *     conf->upstream.ignore_headers = 0;
2547      *     conf->upstream.next_upstream = 0;
2548      *     conf->upstream.cache_zone = NULL;
2549      *     conf->upstream.cache_use_stale = 0;
2550      *     conf->upstream.cache_methods = 0;
2551      *     conf->upstream.temp_path = NULL;
2552      *     conf->upstream.hide_headers_hash = { NULL, 0 };
2553      *     conf->upstream.uri = { 0, NULL };
2554      *     conf->upstream.location = NULL;
2555      *     conf->upstream.store_lengths = NULL;
2556      *     conf->upstream.store_values = NULL;
2557      *     conf->upstream.ssl_name = NULL;
2558      *
2559      *     conf->method = { 0, NULL };
2560      *     conf->headers_source = NULL;
2561      *     conf->headers.lengths = NULL;
2562      *     conf->headers.values = NULL;
2563      *     conf->headers.hash = { NULL, 0 };
2564      *     conf->headers_cache.lengths = NULL;
2565      *     conf->headers_cache.values = NULL;
2566      *     conf->headers_cache.hash = { NULL, 0 };
2567      *     conf->body_lengths = NULL;
2568      *     conf->body_values = NULL;
2569      *     conf->body_source = { 0, NULL };

```

```

2570 *     conf->redirects = NULL;
2571 *     conf->ssl = 0;
2572 *     conf->ssl_protocols = 0;
2573 *     conf->ssl_ciphers = { 0, NULL };
2574 *     conf->ssl_trusted_certificate = { 0, NULL };
2575 *     conf->ssl_crl = { 0, NULL };
2576 *     conf->ssl_certificate = { 0, NULL };
2577 *     conf->ssl_certificate_key = { 0, NULL };
2578 */
2579
2580 conf->upstream.store = NGX_CONF_UNSET;
2581 conf->upstream.store_access = NGX_CONF_UNSET_UINT;
2582 conf->upstream.next_upstream_tries = NGX_CONF_UNSET_UINT;
2583 conf->upstream.buffering = NGX_CONF_UNSET;
2584 conf->upstream.ignore_client_abort = NGX_CONF_UNSET;
2585 conf->upstream.force_ranges = NGX_CONF_UNSET;
2586
2587 conf->upstream.local = NGX_CONF_UNSET_PTR;
2588
2589 conf->upstream.connect_timeout = NGX_CONF_UNSET_MSEC;
2590 conf->upstream.send_timeout = NGX_CONF_UNSET_MSEC;
2591 conf->upstream.read_timeout = NGX_CONF_UNSET_MSEC;
2592 conf->upstream.next_upstream_timeout = NGX_CONF_UNSET_MSEC;
2593
2594 conf->upstream.send_lowat = NGX_CONF_UNSET_SIZE;
2595 conf->upstream.buffer_size = NGX_CONF_UNSET_SIZE;
2596 conf->upstream.limit_rate = NGX_CONF_UNSET_SIZE;
2597
2598 conf->upstream.busy_buffers_size_conf = NGX_CONF_UNSET_SIZE;
2599 conf->upstream.max_temp_file_size_conf = NGX_CONF_UNSET_SIZE;
2600 conf->upstream.temp_file_write_size_conf = NGX_CONF_UNSET_SIZE;
2601
2602 conf->upstream.pass_request_headers = NGX_CONF_UNSET;
2603 conf->upstream.pass_request_body = NGX_CONF_UNSET;
2604
2605 #if (NGX_HTTP_CACHE)
2606 conf->upstream.cache = NGX_CONF_UNSET;
2607 conf->upstream.cache_min_uses = NGX_CONF_UNSET_UINT;
2608 conf->upstream.cache_bypass = NGX_CONF_UNSET_PTR;
2609 conf->upstream.no_cache = NGX_CONF_UNSET_PTR;
2610 conf->upstream.cache_valid = NGX_CONF_UNSET_PTR;
2611 conf->upstream.cache_lock = NGX_CONF_UNSET;
2612 conf->upstream.cache_lock_timeout = NGX_CONF_UNSET_MSEC;
2613 conf->upstream.cache_lock_age = NGX_CONF_UNSET_MSEC;
2614 conf->upstream.cache_revalidate = NGX_CONF_UNSET;
2615 #endif
2616
2617 conf->upstream.hide_headers = NGX_CONF_UNSET_PTR;
2618 conf->upstream.pass_headers = NGX_CONF_UNSET_PTR;
2619
2620 conf->upstream.intercept_errors = NGX_CONF_UNSET;
2621
2622 #if (NGX_HTTP_SSL)
2623 conf->upstream.ssl_session_reuse = NGX_CONF_UNSET;
2624 conf->upstream.ssl_server_name = NGX_CONF_UNSET;
2625 conf->upstream.ssl_verify = NGX_CONF_UNSET;
2626 conf->ssl_verify_depth = NGX_CONF_UNSET_UINT;
2627 conf->ssl_passwords = NGX_CONF_UNSET_PTR;
2628 #endif
2629
2630 /* "proxy_cyclic_temp_file" is disabled */
2631 conf->upstream.cyclic_temp_file = 0;
2632
2633 conf->redirect = NGX_CONF_UNSET;
2634 conf->upstream.change_buffering = 1;
2635
2636 conf->cookie_domains = NGX_CONF_UNSET_PTR;
2637 conf->cookie_paths = NGX_CONF_UNSET_PTR;
2638
2639 conf->http_version = NGX_CONF_UNSET_UINT;
2640
2641 conf->headers_hash_max_size = NGX_CONF_UNSET_UINT;
2642 conf->headers_hash_bucket_size = NGX_CONF_UNSET_UINT;
2643
2644 ngx_str_set(&conf->upstream.module, "proxy");
2645

```

```

2646     return conf;
2647 }
2648
2649
2650 static char *
2651 ngx_http_proxy_merge_loc_conf(ngx_conf_t *cf, void *parent, void *child)
2652 {
2653     ngx_http_proxy_loc_conf_t *prev = parent;
2654     ngx_http_proxy_loc_conf_t *conf = child;
2655
2656     u_char                *p;
2657     size_t                size;
2658     ngx_int_t             rc;
2659     ngx_hash_init_t       hash;
2660     ngx_http_core_loc_conf_t *clcf;
2661     ngx_http_proxy_rewrite_t *pr;
2662     ngx_http_script_compile_t sc;
2663
2664     #if (NGX_HTTP_CACHE)
2665
2666     if (conf->upstream.store > 0) {
2667         conf->upstream.cache = 0;
2668     }
2669
2670     if (conf->upstream.cache > 0) {
2671         conf->upstream.store = 0;
2672     }
2673
2674     #endif
2675
2676     if (conf->upstream.store == NGX_CONF_UNSET) {
2677         ngx_conf_merge_value(conf->upstream.store,
2678             prev->upstream.store, 0);
2679
2680         conf->upstream.store_lengths = prev->upstream.store_lengths;
2681         conf->upstream.store_values = prev->upstream.store_values;
2682     }
2683
2684     ngx_conf_merge_uint_value(conf->upstream.store_access,
2685         prev->upstream.store_access, 0600);
2686
2687     ngx_conf_merge_uint_value(conf->upstream.next_upstream_tries,
2688         prev->upstream.next_upstream_tries, 0);
2689
2690     ngx_conf_merge_value(conf->upstream.buffering,
2691         prev->upstream.buffering, 1);
2692
2693     ngx_conf_merge_value(conf->upstream.ignore_client_abort,
2694         prev->upstream.ignore_client_abort, 0);
2695
2696     ngx_conf_merge_value(conf->upstream.force_ranges,
2697         prev->upstream.force_ranges, 0);
2698
2699     ngx_conf_merge_ptr_value(conf->upstream.local,
2700         prev->upstream.local, NULL);
2701
2702     ngx_conf_merge_msec_value(conf->upstream.connect_timeout,
2703         prev->upstream.connect_timeout, 60000);
2704
2705     ngx_conf_merge_msec_value(conf->upstream.send_timeout,
2706         prev->upstream.send_timeout, 60000);
2707
2708     ngx_conf_merge_msec_value(conf->upstream.read_timeout,
2709         prev->upstream.read_timeout, 60000);
2710
2711     ngx_conf_merge_msec_value(conf->upstream.next_upstream_timeout,
2712         prev->upstream.next_upstream_timeout, 0);
2713
2714     ngx_conf_merge_size_value(conf->upstream.send_lowat,
2715         prev->upstream.send_lowat, 0);
2716
2717     ngx_conf_merge_size_value(conf->upstream.buffer_size,
2718         prev->upstream.buffer_size,
2719         (size_t) ngx_pagesize);
2720
2721     ngx_conf_merge_size_value(conf->upstream.limit_rate,

```

```

2722         prev->upstream.limit_rate, 0);
2723
2724     ngx_conf_merge_bufs_value(conf->upstream.bufts, prev->upstream.bufts,
2725                               8, ngx_pagesize);
2726
2727     if (conf->upstream.bufts.num < 2) {
2728         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
2729                             "there must be at least 2 \"proxy_buffers\");
2730         return NGX_CONF_ERROR;
2731     }
2732
2733
2734     size = conf->upstream.buffer_size;
2735     if (size < conf->upstream.bufts.size) {
2736         size = conf->upstream.bufts.size;
2737     }
2738
2739
2740     ngx_conf_merge_size_value(conf->upstream.busy_buffers_size_conf,
2741                               prev->upstream.busy_buffers_size_conf,
2742                               NGX_CONF_UNSET_SIZE);
2743
2744     if (conf->upstream.busy_buffers_size_conf == NGX_CONF_UNSET_SIZE) {
2745         conf->upstream.busy_buffers_size = 2 * size;
2746     } else {
2747         conf->upstream.busy_buffers_size =
2748             conf->upstream.busy_buffers_size_conf;
2749     }
2750
2751     if (conf->upstream.busy_buffers_size < size) {
2752         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
2753                             "\"proxy_busy_buffers_size\" must be equal to or greater than "
2754                             "the maximum of the value of \"proxy_buffer_size\" and "
2755                             "one of the \"proxy_buffers\");
2756
2757         return NGX_CONF_ERROR;
2758     }
2759
2760     if (conf->upstream.busy_buffers_size
2761         > (conf->upstream.bufts.num - 1) * conf->upstream.bufts.size)
2762     {
2763         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
2764                             "\"proxy_busy_buffers_size\" must be less than "
2765                             "the size of all \"proxy_buffers\" minus one buffer");
2766
2767         return NGX_CONF_ERROR;
2768     }
2769
2770
2771     ngx_conf_merge_size_value(conf->upstream.temp_file_write_size_conf,
2772                               prev->upstream.temp_file_write_size_conf,
2773                               NGX_CONF_UNSET_SIZE);
2774
2775     if (conf->upstream.temp_file_write_size_conf == NGX_CONF_UNSET_SIZE) {
2776         conf->upstream.temp_file_write_size = 2 * size;
2777     } else {
2778         conf->upstream.temp_file_write_size =
2779             conf->upstream.temp_file_write_size_conf;
2780     }
2781
2782     if (conf->upstream.temp_file_write_size < size) {
2783         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
2784                             "\"proxy_temp_file_write_size\" must be equal to or greater "
2785                             "than the maximum of the value of \"proxy_buffer_size\" and "
2786                             "one of the \"proxy_buffers\");
2787
2788         return NGX_CONF_ERROR;
2789     }
2790
2791     ngx_conf_merge_size_value(conf->upstream.max_temp_file_size_conf,
2792                               prev->upstream.max_temp_file_size_conf,
2793                               NGX_CONF_UNSET_SIZE);
2794
2795     if (conf->upstream.max_temp_file_size_conf == NGX_CONF_UNSET_SIZE) {
2796         conf->upstream.max_temp_file_size = 1024 * 1024 * 1024;
2797     } else {

```

```

2798     conf->upstream.max_temp_file_size =
2799         conf->upstream.max_temp_file_size_conf;
2800 }
2801
2802 if (conf->upstream.max_temp_file_size != 0
2803     && conf->upstream.max_temp_file_size < size)
2804 {
2805     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
2806         "\"proxy_max_temp_file_size\" must be equal to zero to disable "
2807         "temporary files usage or must be equal to or greater than "
2808         "the maximum of the value of \"proxy_buffer_size\" and "
2809         "one of the \"proxy_buffers\"");
2810
2811     return NGX_CONF_ERROR;
2812 }
2813
2814
2815 ngx_conf_merge_bitmask_value(conf->upstream.ignore_headers,
2816     prev->upstream.ignore_headers,
2817     NGX_CONF_BITMASK_SET);
2818
2819
2820 ngx_conf_merge_bitmask_value(conf->upstream.next_upstream,
2821     prev->upstream.next_upstream,
2822     (NGX_CONF_BITMASK_SET
2823      |NGX_HTTP_UPSTREAM_FT_ERROR
2824      |NGX_HTTP_UPSTREAM_FT_TIMEOUT));
2825
2826 if (conf->upstream.next_upstream & NGX_HTTP_UPSTREAM_FT_OFF) {
2827     conf->upstream.next_upstream = NGX_CONF_BITMASK_SET
2828         |NGX_HTTP_UPSTREAM_FT_OFF;
2829 }
2830
2831 if (ngx_conf_merge_path_value(cf, &conf->upstream.temp_path,
2832     prev->upstream.temp_path,
2833     &ngx_http_proxy_temp_path)
2834     != NGX_OK)
2835 {
2836     return NGX_CONF_ERROR;
2837 }
2838
2839
2840 #if (NGX_HTTP_CACHE)
2841
2842 if (conf->upstream.cache == NGX_CONF_UNSET) {
2843     ngx_conf_merge_value(conf->upstream.cache,
2844         prev->upstream.cache, 0);
2845
2846     conf->upstream.cache_zone = prev->upstream.cache_zone;
2847     conf->upstream.cache_value = prev->upstream.cache_value;
2848 }
2849
2850 if (conf->upstream.cache_zone && conf->upstream.cache_zone->data == NULL) {
2851     ngx_shm_zone_t *shm_zone;
2852
2853     shm_zone = conf->upstream.cache_zone;
2854
2855     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
2856         "\"proxy_cache\" zone \"%V\" is unknown",
2857         &shm_zone->shm.name);
2858
2859     return NGX_CONF_ERROR;
2860 }
2861
2862 ngx_conf_merge_uint_value(conf->upstream.cache_min_uses,
2863     prev->upstream.cache_min_uses, 1);
2864
2865 ngx_conf_merge_bitmask_value(conf->upstream.cache_use_stale,
2866     prev->upstream.cache_use_stale,
2867     (NGX_CONF_BITMASK_SET
2868      |NGX_HTTP_UPSTREAM_FT_OFF));
2869
2870 if (conf->upstream.cache_use_stale & NGX_HTTP_UPSTREAM_FT_OFF) {
2871     conf->upstream.cache_use_stale = NGX_CONF_BITMASK_SET
2872         |NGX_HTTP_UPSTREAM_FT_OFF;
2873 }

```

```

2874
2875 if (conf->upstream.cache_use_stale & NGX\_HTTP\_UPSTREAM\_FT\_ERROR) {
2876     conf->upstream.cache_use_stale |= NGX\_HTTP\_UPSTREAM\_FT\_NOLIVE;
2877 }
2878
2879 if (conf->upstream.cache_methods == 0) {
2880     conf->upstream.cache_methods = prev->upstream.cache_methods;
2881 }
2882
2883 conf->upstream.cache_methods |= NGX\_HTTP\_GET|NGX\_HTTP\_HEAD;
2884
2885 ngx\_conf\_merge\_ptr\_value(conf->upstream.cache_bypass,
2886     prev->upstream.cache_bypass, NULL);
2887
2888 ngx\_conf\_merge\_ptr\_value(conf->upstream.no_cache,
2889     prev->upstream.no_cache, NULL);
2890
2891 ngx\_conf\_merge\_ptr\_value(conf->upstream.cache_valid,
2892     prev->upstream.cache_valid, NULL);
2893
2894 if (conf->cache_key.value.data == NULL) {
2895     conf->cache_key = prev->cache_key;
2896 }
2897
2898 ngx\_conf\_merge\_value(conf->upstream.cache_lock,
2899     prev->upstream.cache_lock, 0);
2900
2901 ngx\_conf\_merge\_msec\_value(conf->upstream.cache_lock_timeout,
2902     prev->upstream.cache_lock_timeout, 5000);
2903
2904 ngx\_conf\_merge\_msec\_value(conf->upstream.cache_lock_age,
2905     prev->upstream.cache_lock_age, 5000);
2906
2907 ngx\_conf\_merge\_value(conf->upstream.cache_revalidate,
2908     prev->upstream.cache_revalidate, 0);
2909
2910 #endif
2911
2912 ngx\_conf\_merge\_str\_value(conf->method, prev->method, "");
2913
2914 if (conf->method.len
2915     && conf->method.data[conf->method.len - 1] != ' ')
2916 {
2917     conf->method.data[conf->method.len] = ' ';
2918     conf->method.len++;
2919 }
2920
2921 ngx\_conf\_merge\_value(conf->upstream.pass_request_headers,
2922     prev->upstream.pass_request_headers, 1);
2923 ngx\_conf\_merge\_value(conf->upstream.pass_request_body,
2924     prev->upstream.pass_request_body, 1);
2925
2926 ngx\_conf\_merge\_value(conf->upstream.intercept_errors,
2927     prev->upstream.intercept_errors, 0);
2928
2929 #if (NGX\_HTTP\_SSL)
2930
2931 ngx\_conf\_merge\_value(conf->upstream.ssl_session_reuse,
2932     prev->upstream.ssl_session_reuse, 1);
2933
2934 ngx\_conf\_merge\_bitmask\_value(conf->ssl_protocols, prev->ssl_protocols,
2935     (NGX\_CONF\_BITMASK\_SET|NGX\_SSL\_SSLv3
2936     |NGX\_SSL\_TLSv1|NGX\_SSL\_TLSv1\_1
2937     |NGX\_SSL\_TLSv1\_2));
2938
2939 ngx\_conf\_merge\_str\_value(conf->ssl_ciphers, prev->ssl_ciphers,
2940     "DEFAULT");
2941
2942 if (conf->upstream.ssl_name == NULL) {
2943     conf->upstream.ssl_name = prev->upstream.ssl_name;
2944 }
2945
2946 ngx\_conf\_merge\_value(conf->upstream.ssl_server_name,
2947     prev->upstream.ssl_server_name, 0);
2948 ngx\_conf\_merge\_value(conf->upstream.ssl_verify,
2949     prev->upstream.ssl_verify, 0);

```

```

2950 ngx\_conf\_merge\_uint\_value(conf->ssl_verify_depth,
2951     prev->ssl_verify_depth, 1);
2952 ngx\_conf\_merge\_str\_value(conf->ssl_trusted_certificate,
2953     prev->ssl_trusted_certificate, "");
2954 ngx\_conf\_merge\_str\_value(conf->ssl_crl, prev->ssl_crl, "");
2955
2956 ngx\_conf\_merge\_str\_value(conf->ssl_certificate,
2957     prev->ssl_certificate, "");
2958 ngx\_conf\_merge\_str\_value(conf->ssl_certificate_key,
2959     prev->ssl_certificate_key, "");
2960 ngx\_conf\_merge\_ptr\_value(conf->ssl_passwords, prev->ssl_passwords, NULL);
2961
2962 if (conf->ssl && ngx\_http\_proxy\_set\_ssl(cf, conf) != NGX\_OK) {
2963     return NGX\_CONF\_ERROR;
2964 }
2965
2966 #endif
2967
2968 ngx\_conf\_merge\_value(conf->redirect, prev->redirect, 1);
2969
2970 if (conf->redirect) {
2971
2972     if (conf->redirects == NULL) {
2973         conf->redirects = prev->redirects;
2974     }
2975
2976     if (conf->redirects == NULL && conf->url.data) {
2977
2978         conf->redirects = ngx\_array\_create(cf->pool, 1,
2979             sizeof(ngx\_http\_proxy\_rewrite\_t));
2980         if (conf->redirects == NULL) {
2981             return NGX\_CONF\_ERROR;
2982         }
2983
2984         pr = ngx\_array\_push(conf->redirects);
2985         if (pr == NULL) {
2986             return NGX\_CONF\_ERROR;
2987         }
2988
2989         ngx\_memzero(&pr->pattern.complex,
2990             sizeof(ngx\_http\_complex\_value\_t));
2991
2992         ngx\_memzero(&pr->replacement, sizeof(ngx\_http\_complex\_value\_t));
2993
2994         pr->handler = ngx\_http\_proxy\_rewrite\_complex\_handler;
2995
2996         if (conf->vars.uri.len) {
2997             pr->pattern.complex.value = conf->url;
2998             pr->replacement.value = conf->location;
2999
3000         } else {
3001             pr->pattern.complex.value.len = conf->url.len
3002                 + sizeof("/") - 1;
3003
3004             p = ngx\_pnalloc(cf->pool, pr->pattern.complex.value.len);
3005             if (p == NULL) {
3006                 return NGX\_CONF\_ERROR;
3007             }
3008
3009             pr->pattern.complex.value.data = p;
3010
3011             p = ngx\_cpymem(p, conf->url.data, conf->url.len);
3012             *p = '/';
3013
3014             ngx\_str\_set(&pr->replacement.value, "/");
3015         }
3016     }
3017 }
3018
3019 ngx\_conf\_merge\_ptr\_value(conf->cookie_domains, prev->cookie_domains, NULL);
3020
3021 ngx\_conf\_merge\_ptr\_value(conf->cookie_paths, prev->cookie_paths, NULL);
3022
3023 ngx\_conf\_merge\_uint\_value(conf->http_version, prev->http_version,
3024     NGX\_HTTP\_VERSION\_10);
3025

```



```

3026 ngx\_conf\_merge\_uint\_value(conf->headers_hash_max_size,
3027     prev->headers_hash_max_size, 512);
3028
3029 ngx\_conf\_merge\_uint\_value(conf->headers_hash_bucket_size,
3030     prev->headers_hash_bucket_size, 64);
3031
3032 conf->headers_hash_bucket_size = ngx\_align(conf->headers_hash_bucket_size,
3033     ngx\_cacheline\_size);
3034
3035 hash.max_size = conf->headers_hash_max_size;
3036 hash.bucket_size = conf->headers_hash_bucket_size;
3037 hash.name = "proxy_headers_hash";
3038
3039 if (ngx\_http\_upstream\_hide\_headers\_hash(cf, &conf->upstream,
3040     &prev->upstream, ngx\_http\_proxy\_hide\_headers, &hash)
3041     != NGX\_OK)
3042 {
3043     return NGX\_CONF\_ERROR;
3044 }
3045
3046 clcf = ngx\_http\_conf\_get\_module\_loc\_conf(cf, ngx\_http\_core\_module);
3047
3048 if (clcf->noname
3049     && conf->upstream.upstream == NULL && conf->proxy_lengths == NULL)
3050 {
3051     conf->upstream.upstream = prev->upstream.upstream;
3052     conf->location = prev->location;
3053     conf->vars = prev->vars;
3054
3055     conf->proxy_lengths = prev->proxy_lengths;
3056     conf->proxy_values = prev->proxy_values;
3057
3058 #if (NGX\_HTTP\_SSL)
3059     conf->upstream.ssl = prev->upstream.ssl;
3060 #endif
3061 }
3062
3063 if (clcf->lmt_excpt && clcf->handler == NULL
3064     && (conf->upstream.upstream || conf->proxy_lengths))
3065 {
3066     clcf->handler = ngx\_http\_proxy\_handler;
3067 }
3068
3069 if (conf->body_source.data == NULL) {
3070     conf->body_flushes = prev->body_flushes;
3071     conf->body_source = prev->body_source;
3072     conf->body_lengths = prev->body_lengths;
3073     conf->body_values = prev->body_values;
3074 }
3075
3076 if (conf->body_source.data && conf->body_lengths == NULL) {
3077
3078     ngx\_memzero(&sc, sizeof(ngx\_http\_script\_compile\_t));
3079
3080     sc.cf = cf;
3081     sc.source = &conf->body_source;
3082     sc.flushes = &conf->body_flushes;
3083     sc.lengths = &conf->body_lengths;
3084     sc.values = &conf->body_values;
3085     sc.complete_lengths = 1;
3086     sc.complete_values = 1;
3087
3088     if (ngx\_http\_script\_compile(&sc) != NGX\_OK) {
3089         return NGX\_CONF\_ERROR;
3090     }
3091 }
3092
3093 if (conf->headers_source == NULL) {
3094     conf->headers = prev->headers;
3095 #if (NGX\_HTTP\_CACHE)
3096     conf->headers_cache = prev->headers_cache;
3097 #endif
3098     conf->headers_source = prev->headers_source;
3099 }
3100
3101 rc = ngx\_http\_proxy\_init\_headers(cf, conf, &conf->headers,

```

```

3102                                     ngx_http_proxy_headers);
3103     if (rc != NGX_OK) {
3104         return NGX_CONF_ERROR;
3105     }
3106
3107     #if (NGX_HTTP_CACHE)
3108
3109     if (conf->upstream.cache) {
3110         rc = ngx_http_proxy_init_headers(cf, conf, &conf->headers_cache,
3111                                         ngx_http_proxy_cache_headers);
3112         if (rc != NGX_OK) {
3113             return NGX_CONF_ERROR;
3114         }
3115     }
3116
3117     #endif
3118
3119     return NGX_CONF_OK;
3120 }
3121
3122
3123 static ngx_int_t
3124 ngx_http_proxy_init_headers(ngx_conf_t *cf, ngx_http_proxy_loc_conf_t *conf,
3125                             ngx_http_proxy_headers_t *headers, ngx_keyval_t *default_headers)
3126 {
3127     u_char                *p;
3128     size_t                size;
3129     uintptr_t             *code;
3130     ngx_uint_t            i;
3131     ngx_array_t           headers_names, headers_merged;
3132     ngx_keyval_t          *src, *s, *h;
3133     ngx_hash_key_t        *hk;
3134     ngx_hash_init_t        hash;
3135     ngx_http_script_compile_t sc;
3136     ngx_http_script_copy_code_t *copy;
3137
3138     if (headers->hash.buckets) {
3139         return NGX_OK;
3140     }
3141
3142     if (ngx_array_init(&headers_names, cf->temp_pool, 4, sizeof(ngx_hash_key_t))
3143         != NGX_OK)
3144     {
3145         return NGX_ERROR;
3146     }
3147
3148     if (ngx_array_init(&headers_merged, cf->temp_pool, 4, sizeof(ngx_keyval_t))
3149         != NGX_OK)
3150     {
3151         return NGX_ERROR;
3152     }
3153
3154     if (conf->headers_source == NULL) {
3155         conf->headers_source = ngx_array_create(cf->pool, 4,
3156                                                sizeof(ngx_keyval_t));
3157         if (conf->headers_source == NULL) {
3158             return NGX_ERROR;
3159         }
3160     }
3161
3162     headers->lengths = ngx_array_create(cf->pool, 64, 1);
3163     if (headers->lengths == NULL) {
3164         return NGX_ERROR;
3165     }
3166
3167     headers->values = ngx_array_create(cf->pool, 512, 1);
3168     if (headers->values == NULL) {
3169         return NGX_ERROR;
3170     }
3171
3172     src = conf->headers_source->elts;
3173     for (i = 0; i < conf->headers_source->nelts; i++) {
3174
3175         s = ngx_array_push(&headers_merged);
3176         if (s == NULL) {
3177             return NGX_ERROR;

```

```

3178     }
3179
3180     *s = src[i];
3181 }
3182
3183 h = default_headers;
3184
3185 while (h->key.len) {
3186
3187     src = headers_merged.elts;
3188     for (i = 0; i < headers_merged.nelts; i++) {
3189         if (ngx_strcasecmp(h->key.data, src[i].key.data) == 0) {
3190             goto next;
3191         }
3192     }
3193
3194     s = ngx_array_push(&headers_merged);
3195     if (s == NULL) {
3196         return NGX_ERROR;
3197     }
3198
3199     *s = *h;
3200
3201 next:
3202
3203     h++;
3204 }
3205
3206
3207 src = headers_merged.elts;
3208 for (i = 0; i < headers_merged.nelts; i++) {
3209
3210     hk = ngx_array_push(&headers_names);
3211     if (hk == NULL) {
3212         return NGX_ERROR;
3213     }
3214
3215     hk->key = src[i].key;
3216     hk->key_hash = ngx_hash_key_lc(src[i].key.data, src[i].key.len);
3217     hk->value = (void *) 1;
3218
3219     if (src[i].value.len == 0) {
3220         continue;
3221     }
3222
3223     if (ngx_http_script_variables_count(&src[i].value) == 0) {
3224         copy = ngx_array_push_n(headers->lengths,
3225                                 sizeof(ngx_http_script_copy_code_t));
3226         if (copy == NULL) {
3227             return NGX_ERROR;
3228         }
3229
3230         copy->code = (ngx_http_script_code_pt)
3231                     ngx_http_script_copy_len_code;
3232         copy->len = src[i].key.len + sizeof(": ") - 1
3233                   + src[i].value.len + sizeof(CRLF) - 1;
3234
3235         size = (sizeof(ngx_http_script_copy_code_t)
3236                + src[i].key.len + sizeof(": ") - 1
3237                + src[i].value.len + sizeof(CRLF) - 1
3238                + sizeof(uintptr_t) - 1)
3239               & ~(sizeof(uintptr_t) - 1);
3240
3241         copy = ngx_array_push_n(headers->values, size);
3242         if (copy == NULL) {
3243             return NGX_ERROR;
3244         }
3245     }
3246
3247     copy->code = ngx_http_script_copy_code;
3248     copy->len = src[i].key.len + sizeof(": ") - 1
3249               + src[i].value.len + sizeof(CRLF) - 1;
3250
3251     p = (u_char *) copy + sizeof(ngx_http_script_copy_code_t);
3252
3253     p = ngx_cpymem(p, src[i].key.data, src[i].key.len);

```

```

3254     *p++ = ':'; *p++ = ' ';
3255     p = ngx_cpymem(p, src[i].value.data, src[i].value.len);
3256     *p++ = CR; *p = LF;
3257
3258 } else {
3259     copy = ngx_array_push_n(headers->lengths,
3260                             sizeof(ngx_http_script_copy_code_t));
3261     if (copy == NULL) {
3262         return NGX_ERROR;
3263     }
3264
3265     copy->code = (ngx_http_script_code_pt)
3266                 ngx_http_script_copy_len_code;
3267     copy->len = src[i].key.len + sizeof(": ") - 1;
3268
3269
3270     size = (sizeof(ngx_http_script_copy_code_t)
3271            + src[i].key.len + sizeof(": ") - 1 + sizeof(uintptr_t) - 1)
3272           & ~(sizeof(uintptr_t) - 1);
3273
3274     copy = ngx_array_push_n(headers->values, size);
3275     if (copy == NULL) {
3276         return NGX_ERROR;
3277     }
3278
3279     copy->code = ngx_http_script_copy_code;
3280     copy->len = src[i].key.len + sizeof(": ") - 1;
3281
3282     p = (u_char *) copy + sizeof(ngx_http_script_copy_code_t);
3283     p = ngx_cpymem(p, src[i].key.data, src[i].key.len);
3284     *p++ = ':'; *p = ' ';
3285
3286
3287     ngx_memzero(&sc, sizeof(ngx_http_script_compile_t));
3288
3289     sc.cf = cf;
3290     sc.source = &src[i].value;
3291     sc.flushes = &headers->flushes;
3292     sc.lengths = &headers->lengths;
3293     sc.values = &headers->values;
3294
3295     if (ngx_http_script_compile(&sc) != NGX_OK) {
3296         return NGX_ERROR;
3297     }
3298
3299
3300     copy = ngx_array_push_n(headers->lengths,
3301                             sizeof(ngx_http_script_copy_code_t));
3302     if (copy == NULL) {
3303         return NGX_ERROR;
3304     }
3305
3306     copy->code = (ngx_http_script_code_pt)
3307                 ngx_http_script_copy_len_code;
3308     copy->len = sizeof(CRLF) - 1;
3309
3310
3311     size = (sizeof(ngx_http_script_copy_code_t)
3312            + sizeof(CRLF) - 1 + sizeof(uintptr_t) - 1)
3313           & ~(sizeof(uintptr_t) - 1);
3314
3315     copy = ngx_array_push_n(headers->values, size);
3316     if (copy == NULL) {
3317         return NGX_ERROR;
3318     }
3319
3320     copy->code = ngx_http_script_copy_code;
3321     copy->len = sizeof(CRLF) - 1;
3322
3323     p = (u_char *) copy + sizeof(ngx_http_script_copy_code_t);
3324     *p++ = CR; *p = LF;
3325 }
3326
3327 code = ngx_array_push_n(headers->lengths, sizeof(uintptr_t));
3328 if (code == NULL) {
3329     return NGX_ERROR;

```

```

3330     }
3331
3332     *code = (uintptr_t) NULL;
3333
3334     code = ngx_array_push_n(headers->values, sizeof(uintptr_t));
3335     if (code == NULL) {
3336         return NGX_ERROR;
3337     }
3338
3339     *code = (uintptr_t) NULL;
3340 }
3341
3342 code = ngx_array_push_n(headers->lengths, sizeof(uintptr_t));
3343 if (code == NULL) {
3344     return NGX_ERROR;
3345 }
3346
3347 *code = (uintptr_t) NULL;
3348
3349
3350 hash.hash = &headers->hash;
3351 hash.key = ngx_hash_key_lc;
3352 hash.max_size = conf->headers_hash_max_size;
3353 hash.bucket_size = conf->headers_hash_bucket_size;
3354 hash.name = "proxy_headers_hash";
3355 hash.pool = cf->pool;
3356 hash.temp_pool = NULL;
3357
3358 return ngx_hash_init(&hash, headers_names.elts, headers_names.nelts);
3359 }
3360
3361
3362 static char *
3363 ngx_http_proxy_pass(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
3364 {
3365     ngx_http_proxy_loc_conf_t *plcf = conf;
3366
3367     size_t      add;
3368     u_short     port;
3369     ngx_str_t   *value, *url;
3370     ngx_url_t   u;
3371     ngx_uint_t   n;
3372     ngx_http_core_loc_conf_t *clcf;
3373     ngx_http_script_compile_t sc;
3374
3375     if (plcf->upstream.upstream || plcf->proxy_lengths) {
3376         return "is duplicate";
3377     }
3378
3379     clcf = ngx_http_conf_get_module_loc_conf(cf, ngx_http_core_module);
3380
3381     clcf->handler = ngx_http_proxy_handler;
3382
3383     if (clcf->name.data[clcf->name.len - 1] == '/') {
3384         clcf->auto_redirect = 1;
3385     }
3386
3387     value = cf->args->elts;
3388
3389     url = &value[1];
3390
3391     n = ngx_http_script_variables_count(url);
3392
3393     if (n) {
3394         ngx_memzero(&sc, sizeof(ngx_http_script_compile_t));
3395
3396         sc.cf = cf;
3397         sc.source = url;
3398         sc.lengths = &plcf->proxy_lengths;
3399         sc.values = &plcf->proxy_values;
3400         sc.variables = n;
3401         sc.complete_lengths = 1;
3402         sc.complete_values = 1;
3403
3404         if (ngx_http_script_compile(&sc) != NGX_OK) {

```

```

3406         return NGX\_CONF\_ERROR;
3407     }
3408
3409     #if (NGX_HTTP_SSL)
3410         plcf->ssl = 1;
3411     #endif
3412
3413     return NGX\_CONF\_OK;
3414 }
3415
3416 if (ngx\_strncasecmp(url->data, (u_char *) "http://", 7) == 0) {
3417     add = 7;
3418     port = 80;
3419
3420 } else if (ngx\_strncasecmp(url->data, (u_char *) "https://", 8) == 0) {
3421
3422     #if (NGX_HTTP_SSL)
3423         plcf->ssl = 1;
3424
3425         add = 8;
3426         port = 443;
3427     #else
3428         ngx\_conf\_log\_error(NGX_LOG_EMERG, cf, 0,
3429             "https protocol requires SSL support");
3430         return NGX\_CONF\_ERROR;
3431     #endif
3432
3433     } else {
3434         ngx\_conf\_log\_error(NGX_LOG_EMERG, cf, 0, "invalid URL prefix");
3435         return NGX\_CONF\_ERROR;
3436     }
3437
3438     ngx\_memzero(&u, sizeof(ngx\_url\_t));
3439
3440     u.url.len = url->len - add;
3441     u.url.data = url->data + add;
3442     u.default_port = port;
3443     u.uri_part = 1;
3444     u.no_resolve = 1;
3445
3446     plcf->upstream.upstream = ngx\_http\_upstream\_add(cf, &u, 0);
3447     if (plcf->upstream.upstream == NULL) {
3448         return NGX\_CONF\_ERROR;
3449     }
3450
3451     plcf->vars.schema.len = add;
3452     plcf->vars.schema.data = url->data;
3453     plcf->vars.key_start = plcf->vars.schema;
3454
3455     ngx\_http\_proxy\_set\_vars(&u, &plcf->vars);
3456
3457     plcf->location = clcf->name;
3458
3459     if (clcf->named
3460 #if (NGX_PCRE)
3461         || clcf->regex
3462 #endif
3463         || clcf->noname)
3464     {
3465         if (plcf->vars.uri.len) {
3466             ngx\_conf\_log\_error(NGX_LOG_EMERG, cf, 0,
3467                 "\"proxy_pass\" cannot have URI part in "
3468                 "location given by regular expression, "
3469                 "or inside named location, "
3470                 "or inside \"if\" statement, "
3471                 "or inside \"limit_except\" block");
3472             return NGX\_CONF\_ERROR;
3473         }
3474
3475         plcf->location.len = 0;
3476     }
3477
3478     plcf->url = *url;
3479
3480     return NGX\_CONF\_OK;
3481 }

```

```

3482
3483
3484 static char *
3485 ngx_http_proxy_redirect(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
3486 {
3487     ngx_http_proxy_loc_conf_t *plcf = conf;
3488
3489     u_char                *p;
3490     ngx_str_t             *value;
3491     ngx_http_proxy_rewrite_t *pr;
3492     ngx_http_complex_value_t ccv;
3493
3494     if (plcf->redirect == 0) {
3495         return NGX_CONF_OK;
3496     }
3497
3498     plcf->redirect = 1;
3499
3500     value = cf->args->elts;
3501
3502     if (cf->args->nelts == 2) {
3503         if (ngx_strcmp(value[1].data, "off") == 0) {
3504             plcf->redirect = 0;
3505             plcf->redirects = NULL;
3506             return NGX_CONF_OK;
3507         }
3508
3509         if (ngx_strcmp(value[1].data, "false") == 0) {
3510             ngx_conf_log_error(NGX_LOG_ERR, cf, 0,
3511                 "invalid parameter \"false\", use \"off\" instead");
3512             plcf->redirect = 0;
3513             plcf->redirects = NULL;
3514             return NGX_CONF_OK;
3515         }
3516
3517         if (ngx_strcmp(value[1].data, "default") != 0) {
3518             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
3519                 "invalid parameter \"%V\"", &value[1]);
3520             return NGX_CONF_ERROR;
3521         }
3522     }
3523
3524     if (plcf->redirects == NULL) {
3525         plcf->redirects = ngx_array_create(cf->pool, 1,
3526             sizeof(ngx_http_proxy_rewrite_t));
3527         if (plcf->redirects == NULL) {
3528             return NGX_CONF_ERROR;
3529         }
3530     }
3531
3532     pr = ngx_array_push(plcf->redirects);
3533     if (pr == NULL) {
3534         return NGX_CONF_ERROR;
3535     }
3536
3537     if (ngx_strcmp(value[1].data, "default") == 0) {
3538         if (plcf->proxy_lengths) {
3539             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
3540                 "\"proxy_redirect default\" cannot be used "
3541                 "with \"proxy_pass\" directive with variables");
3542             return NGX_CONF_ERROR;
3543         }
3544
3545         if (plcf->url.data == NULL) {
3546             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
3547                 "\"proxy_redirect default\" should be placed "
3548                 "after the \"proxy_pass\" directive");
3549             return NGX_CONF_ERROR;
3550         }
3551
3552         pr->handler = ngx_http_proxy_rewrite_complex_handler;
3553
3554         ngx_memzero(&pr->pattern.complex, sizeof(ngx_http_complex_value_t));
3555
3556         ngx_memzero(&pr->replacement, sizeof(ngx_http_complex_value_t));
3557

```

```

3558     if (plcf->vars.uri.len) {
3559         pr->pattern.complex.value = plcf->url;
3560         pr->replacement.value = plcf->location;
3561
3562     } else {
3563         pr->pattern.complex.value.len = plcf->url.len + sizeof("/") - 1;
3564
3565         p = ngx_pnalloc(cf->pool, pr->pattern.complex.value.len);
3566         if (p == NULL) {
3567             return NGX_CONF_ERROR;
3568         }
3569
3570         pr->pattern.complex.value.data = p;
3571
3572         p = ngx_cpymem(p, plcf->url.data, plcf->url.len);
3573         *p = '/';
3574
3575         ngx_str_set(&pr->replacement.value, "/");
3576     }
3577
3578     return NGX_CONF_OK;
3579 }
3580
3581
3582 if (value[1].data[0] == '~') {
3583     value[1].len--;
3584     value[1].data++;
3585
3586     if (value[1].data[0] == '*') {
3587         value[1].len--;
3588         value[1].data++;
3589
3590         if (ngx_http_proxy_rewrite_regex(cf, pr, &value[1], 1) != NGX_OK) {
3591             return NGX_CONF_ERROR;
3592         }
3593     } else {
3594         if (ngx_http_proxy_rewrite_regex(cf, pr, &value[1], 0) != NGX_OK) {
3595             return NGX_CONF_ERROR;
3596         }
3597     }
3598 }
3599
3600 } else {
3601
3602     ngx_memzero(&ccv, sizeof(ngx_http_compile_complex_value_t));
3603
3604     ccv.cf = cf;
3605     ccv.value = &value[1];
3606     ccv.complex_value = &pr->pattern.complex;
3607
3608     if (ngx_http_compile_complex_value(&ccv) != NGX_OK) {
3609         return NGX_CONF_ERROR;
3610     }
3611
3612     pr->handler = ngx_http_proxy_rewrite_complex_handler;
3613 }
3614
3615
3616 ngx_memzero(&ccv, sizeof(ngx_http_compile_complex_value_t));
3617
3618 ccv.cf = cf;
3619 ccv.value = &value[2];
3620 ccv.complex_value = &pr->replacement;
3621
3622 if (ngx_http_compile_complex_value(&ccv) != NGX_OK) {
3623     return NGX_CONF_ERROR;
3624 }
3625
3626 return NGX_CONF_OK;
3627 }
3628
3629
3630 static char *
3631 ngx_http_proxy_cookie_domain(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
3632 {
3633     ngx_http_proxy_loc_conf_t *plcf = conf;

```



```

3634
3635 ngx\_str\_t *value;
3636 ngx\_http\_proxy\_rewrite\_t *pr;
3637 ngx\_http\_compile\_complex\_value\_t ccv;
3638
3639 if (plcf->cookie_domains == NULL) {
3640     return NGX\_CONF\_OK;
3641 }
3642
3643 value = cf->args->elts;
3644
3645 if (cf->args->nelts == 2) {
3646
3647     if (ngx\_strcmp(value[1].data, "off") == 0) {
3648         plcf->cookie_domains = NULL;
3649         return NGX\_CONF\_OK;
3650     }
3651
3652     ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
3653         "invalid parameter \"%v\"", &value[1]);
3654     return NGX\_CONF\_ERROR;
3655 }
3656
3657 if (plcf->cookie_domains == NGX\_CONF\_UNSET\_PTR) {
3658     plcf->cookie_domains = ngx\_array\_create(cf->pool, 1,
3659         sizeof(ngx\_http\_proxy\_rewrite\_t));
3660     if (plcf->cookie_domains == NULL) {
3661         return NGX\_CONF\_ERROR;
3662     }
3663 }
3664
3665 pr = ngx\_array\_push(plcf->cookie_domains);
3666 if (pr == NULL) {
3667     return NGX\_CONF\_ERROR;
3668 }
3669
3670 if (value[1].data[0] == '~') {
3671     value[1].len--;
3672     value[1].data++;
3673
3674     if (ngx\_http\_proxy\_rewrite\_regex(cf, pr, &value[1], 1) != NGX\_OK) {
3675         return NGX\_CONF\_ERROR;
3676     }
3677
3678 } else {
3679
3680     if (value[1].data[0] == '.') {
3681         value[1].len--;
3682         value[1].data++;
3683     }
3684
3685     ngx\_memzero(&ccv, sizeof(ngx\_http\_compile\_complex\_value\_t));
3686
3687     ccv.cf = cf;
3688     ccv.value = &value[1];
3689     ccv.complex_value = &pr->pattern.complex;
3690
3691     if (ngx\_http\_compile\_complex\_value(&ccv) != NGX\_OK) {
3692         return NGX\_CONF\_ERROR;
3693     }
3694
3695     pr->handler = ngx\_http\_proxy\_rewrite\_domain\_handler;
3696
3697     if (value[2].data[0] == '.') {
3698         value[2].len--;
3699         value[2].data++;
3700     }
3701 }
3702
3703 ngx\_memzero(&ccv, sizeof(ngx\_http\_compile\_complex\_value\_t));
3704
3705 ccv.cf = cf;
3706 ccv.value = &value[2];
3707 ccv.complex_value = &pr->replacement;
3708
3709 if (ngx\_http\_compile\_complex\_value(&ccv) != NGX\_OK) {

```

```

3710     return NGX\_CONF\_ERROR;
3711 }
3712
3713     return NGX\_CONF\_OK;
3714 }
3715
3716
3717 static char *
3718 ngx\_http\_proxy\_cookie\_path(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
3719 {
3720     ngx\_http\_proxy\_loc\_conf\_t *plcf = conf;
3721
3722     ngx\_str\_t                *value;
3723     ngx\_http\_proxy\_rewrite\_t *pr;
3724     ngx\_http\_compile\_complex\_value\_t ccv;
3725
3726     if (plcf->cookie_paths == NULL) {
3727         return NGX\_CONF\_OK;
3728     }
3729
3730     value = cf->args->elts;
3731
3732     if (cf->args->nelts == 2) {
3733
3734         if (ngx\_strcmp(value[1].data, "off") == 0) {
3735             plcf->cookie_paths = NULL;
3736             return NGX\_CONF\_OK;
3737         }
3738
3739         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
3740             "invalid parameter \"%V\"", &value[1]);
3741         return NGX\_CONF\_ERROR;
3742     }
3743
3744     if (plcf->cookie_paths == NGX\_CONF\_UNSET\_PTR) {
3745         plcf->cookie_paths = ngx\_array\_create(cf->pool, 1,
3746             sizeof(ngx\_http\_proxy\_rewrite\_t));
3747         if (plcf->cookie_paths == NULL) {
3748             return NGX\_CONF\_ERROR;
3749         }
3750     }
3751
3752     pr = ngx\_array\_push(plcf->cookie_paths);
3753     if (pr == NULL) {
3754         return NGX\_CONF\_ERROR;
3755     }
3756
3757     if (value[1].data[0] == '~') {
3758         value[1].len--;
3759         value[1].data++;
3760
3761         if (value[1].data[0] == '*') {
3762             value[1].len--;
3763             value[1].data++;
3764
3765             if (ngx\_http\_proxy\_rewrite\_regex(cf, pr, &value[1], 1) != NGX\_OK) {
3766                 return NGX\_CONF\_ERROR;
3767             }
3768
3769         } else {
3770             if (ngx\_http\_proxy\_rewrite\_regex(cf, pr, &value[1], 0) != NGX\_OK) {
3771                 return NGX\_CONF\_ERROR;
3772             }
3773         }
3774     } else {
3775
3776         ngx\_memzero(&ccv, sizeof(ngx\_http\_compile\_complex\_value\_t));
3777
3778         ccv.cf = cf;
3779         ccv.value = &value[1];
3780         ccv.complex_value = &pr->pattern.complex;
3781
3782         if (ngx\_http\_compile\_complex\_value(&ccv) != NGX\_OK) {
3783             return NGX\_CONF\_ERROR;
3784         }
3785     }

```

```

3786     pr->handler = ngx\_http\_proxy\_rewrite\_complex\_handler;
3787 }
3788
3789 ngx\_memzero(&ccv, sizeof(ngx\_http\_compile\_complex\_value\_t));
3790
3791 ccv.cf = cf;
3792 ccv.value = &value[2];
3793 ccv.complex_value = &pr->replacement;
3794
3795 if (ngx\_http\_compile\_complex\_value(&ccv) != NGX\_OK) {
3796     return NGX\_CONF\_ERROR;
3797 }
3798
3799 return NGX\_CONF\_OK;
3800 }
3801
3802
3803
3804 static ngx\_int\_t
3805 ngx\_http\_proxy\_rewrite\_regex(ngx\_conf\_t *cf, ngx\_http\_proxy\_rewrite\_t *pr,
3806 ngx\_str\_t *regex, ngx\_uint\_t caseless)
3807 {
3808     #if (NGX\_PCRE)
3809         u\_char          errstr[NGX\_MAX\_CONF\_ERRSTR];
3810         ngx\_regex\_compile\_t rc;
3811
3812         ngx\_memzero(&rc, sizeof(ngx\_regex\_compile\_t));
3813
3814         rc.pattern = *regex;
3815         rc.err.len = NGX\_MAX\_CONF\_ERRSTR;
3816         rc.err.data = errstr;
3817
3818         if (caseless) {
3819             rc.options = NGX\_REGEX\_CASELESS;
3820         }
3821
3822         pr->pattern.regex = ngx\_http\_regex\_compile(cf, &rc);
3823         if (pr->pattern.regex == NULL) {
3824             return NGX\_ERROR;
3825         }
3826
3827         pr->handler = ngx\_http\_proxy\_rewrite\_regex\_handler;
3828
3829         return NGX\_OK;
3830
3831     #else
3832
3833         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
3834             "using regex \"%V\" requires PCRE library", regex);
3835         return NGX\_ERROR;
3836
3837     #endif
3838 }
3839
3840
3841 static char *
3842 ngx\_http\_proxy\_store(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
3843 {
3844     ngx\_http\_proxy\_loc\_conf\_t *plcf = conf;
3845
3846     ngx\_str\_t          *value;
3847     ngx\_http\_script\_compile\_t sc;
3848
3849     if (plcf->upstream.store != NGX\_CONF\_UNSET) {
3850         return "is duplicate";
3851     }
3852
3853     value = cf->args->elts;
3854
3855     if (ngx\_strcmp(value[1].data, "off") == 0) {
3856         plcf->upstream.store = 0;
3857         return NGX\_CONF\_OK;
3858     }
3859
3860     #if (NGX\_HTTP\_CACHE)
3861     if (plcf->upstream.cache > 0) {

```

```

3862     return "is incompatible with \"proxy_cache\"";
3863 }
3864 #endif
3865
3866     plcf->upstream.store = 1;
3867
3868     if (ngx_strcmp(value[1].data, "on") == 0) {
3869         return NGX_CONF_OK;
3870     }
3871
3872     /* include the terminating '\0' into script */
3873     value[1].len++;
3874
3875     ngx_memzero(&sc, sizeof(ngx_http_script_compile_t));
3876
3877     sc.cf = cf;
3878     sc.source = &value[1];
3879     sc.lengths = &plcf->upstream.store_lengths;
3880     sc.values = &plcf->upstream.store_values;
3881     sc.variables = ngx_http_script_variables_count(&value[1]);
3882     sc.complete_lengths = 1;
3883     sc.complete_values = 1;
3884
3885     if (ngx_http_script_compile(&sc) != NGX_OK) {
3886         return NGX_CONF_ERROR;
3887     }
3888
3889     return NGX_CONF_OK;
3890 }
3891
3892
3893 #if (NGX_HTTP_CACHE)
3894
3895 static char *
3896 ngx_http_proxy_cache(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
3897 {
3898     ngx_http_proxy_loc_conf_t *plcf = conf;
3899
3900     ngx_str_t          *value;
3901     ngx_http_complex_value_t  cv;
3902     ngx_http_compile_complex_value_t  ccv;
3903
3904     value = cf->args->elts;
3905
3906     if (plcf->upstream.cache != NGX_CONF_UNSET) {
3907         return "is duplicate";
3908     }
3909
3910     if (ngx_strcmp(value[1].data, "off") == 0) {
3911         plcf->upstream.cache = 0;
3912         return NGX_CONF_OK;
3913     }
3914
3915     if (plcf->upstream.store > 0) {
3916         return "is incompatible with \"proxy_store\"";
3917     }
3918
3919     plcf->upstream.cache = 1;
3920
3921     ngx_memzero(&ccv, sizeof(ngx_http_compile_complex_value_t));
3922
3923     ccv.cf = cf;
3924     ccv.value = &value[1];
3925     ccv.complex_value = &cv;
3926
3927     if (ngx_http_compile_complex_value(&ccv) != NGX_OK) {
3928         return NGX_CONF_ERROR;
3929     }
3930
3931     if (cv.lengths != NULL) {
3932
3933         plcf->upstream.cache_value = ngx_palloc(cf->pool,
3934                                                 sizeof(ngx_http_complex_value_t));
3935         if (plcf->upstream.cache_value == NULL) {
3936             return NGX_CONF_ERROR;
3937         }

```

```

3938     *plcf->upstream.cache_value = cv;
3939
3940     return NGX\_CONF\_OK;
3941 }
3942
3943 plcf->upstream.cache_zone = ngx\_shared\_memory\_add(cf, &value[1], 0,
3944                                             &ngx\_http\_proxy\_module);
3945
3946 if (plcf->upstream.cache_zone == NULL) {
3947     return NGX\_CONF\_ERROR;
3948 }
3949
3950 return NGX\_CONF\_OK;
3951 }
3952
3953
3954 static char *
3955 ngx\_http\_proxy\_cache\_key(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
3956 {
3957     ngx\_http\_proxy\_loc\_conf\_t *plcf = conf;
3958
3959     ngx\_str\_t *value;
3960     ngx\_http\_compile\_complex\_value\_t ccv;
3961
3962     value = cf->args->elts;
3963
3964     if (plcf->cache_key.value.data) {
3965         return "is duplicate";
3966     }
3967
3968     ngx\_memzero(&ccv, sizeof(ngx\_http\_compile\_complex\_value\_t));
3969
3970     ccv.cf = cf;
3971     ccv.value = &value[1];
3972     ccv.complex_value = &plcf->cache_key;
3973
3974     if (ngx\_http\_compile\_complex\_value(&ccv) != NGX\_OK) {
3975         return NGX\_CONF\_ERROR;
3976     }
3977
3978     return NGX\_CONF\_OK;
3979 }
3980
3981 #endif
3982
3983
3984 #if (NGX\_HTTP\_SSL)
3985
3986 static char *
3987 ngx\_http\_proxy\_ssl\_password\_file(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
3988 {
3989     ngx\_http\_proxy\_loc\_conf\_t *plcf = conf;
3990
3991     ngx\_str\_t *value;
3992
3993     if (plcf->ssl_passwords != NGX\_CONF\_UNSET\_PTR) {
3994         return "is duplicate";
3995     }
3996
3997     value = cf->args->elts;
3998
3999     plcf->ssl_passwords = ngx\_ssl\_read\_password\_file(cf, &value[1]);
4000
4001     if (plcf->ssl_passwords == NULL) {
4002         return NGX\_CONF\_ERROR;
4003     }
4004
4005     return NGX\_CONF\_OK;
4006 }
4007
4008 #endif
4009
4010
4011 static char *
4012 ngx\_http\_proxy\_lowat\_check(ngx\_conf\_t *cf, void *post, void *data)
4013 {

```

```

4014 #if (NGX_FREEBSD)
4015     ssize_t *np = data;
4016
4017     if ((u_long) *np >= ngx_freebsd_net_inet_tcp_sendspace) {
4018         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
4019             "\"proxy_send_lowat\" must be less than %d "
4020             "(sysctl net.inet.tcp.sendspace)",
4021             ngx_freebsd_net_inet_tcp_sendspace);
4022
4023         return NGX_CONF_ERROR;
4024     }
4025
4026 #elif !(NGX_HAVE_SO_SNDLOWAT)
4027     ssize_t *np = data;
4028
4029     ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
4030         "\"proxy_send_lowat\" is not supported, ignored");
4031
4032     *np = 0;
4033
4034 #endif
4035
4036     return NGX_CONF_OK;
4037 }
4038
4039 #if (NGX_HTTP_SSL)
4040
4041 static ngx_int_t
4042 ngx_http_proxy_set_ssl(ngx_conf_t *cf, ngx_http_proxy_loc_conf_t *plcf)
4043 {
4044     ngx_pool_cleanup_t *cfn;
4045
4046     plcf->upstream.ssl = ngx_palloc(cf->pool, sizeof(ngx_ssl_t));
4047     if (plcf->upstream.ssl == NULL) {
4048         return NGX_ERROR;
4049     }
4050
4051     plcf->upstream.ssl->log = cf->log;
4052
4053     if (ngx_ssl_create(plcf->upstream.ssl, plcf->ssl_protocols, NULL)
4054         != NGX_OK)
4055     {
4056         return NGX_ERROR;
4057     }
4058
4059     cfn = ngx_pool_cleanup_add(cf->pool, 0);
4060     if (cfn == NULL) {
4061         return NGX_ERROR;
4062     }
4063
4064     cfn->handler = ngx_ssl_cleanup_ctx;
4065     cfn->data = plcf->upstream.ssl;
4066
4067     if (plcf->ssl_certificate.len) {
4068         if (plcf->ssl_certificate_key.len == 0) {
4069             ngx_log_error(NGX_LOG_EMERG, cf->log, 0,
4070                 "no \"proxy_ssl_certificate_key\" is defined "
4071                 "for certificate \"%V\"", &plcf->ssl_certificate);
4072             return NGX_ERROR;
4073         }
4074
4075         if (ngx_ssl_certificate(cf, plcf->upstream.ssl, &plcf->ssl_certificate,
4076             &plcf->ssl_certificate_key, plcf->ssl_passwords)
4077             != NGX_OK)
4078         {
4079             return NGX_ERROR;
4080         }
4081     }
4082
4083     if (SSL_CTX_set_cipher_list(plcf->upstream.ssl->ctx,
4084         (const char *) plcf->ssl_ciphers.data)
4085         == 0)
4086     {
4087         ngx_ssl_error(NGX_LOG_EMERG, cf->log, 0,

```

```

4090         "SSL_CTX_set_cipher_list(\"%V\") failed",
4091         &plcf->ssl_ciphers);
4092     return NGX_ERROR;
4093 }
4094
4095 if (plcf->upstream.ssl_verify) {
4096     if (plcf->ssl_trusted_certificate.len == 0) {
4097         ngx_log_error(NGX_LOG_EMERG, cf->log, 0,
4098             "no proxy_ssl_trusted_certificate for proxy_ssl_verify");
4099         return NGX_ERROR;
4100     }
4101
4102     if (ngx_ssl_trusted_certificate(cf, plcf->upstream.ssl,
4103         &plcf->ssl_trusted_certificate,
4104         plcf->ssl_verify_depth)
4105         != NGX_OK)
4106     {
4107         return NGX_ERROR;
4108     }
4109
4110     if (ngx_ssl_crl(cf, plcf->upstream.ssl, &plcf->ssl_crl) != NGX_OK) {
4111         return NGX_ERROR;
4112     }
4113 }
4114
4115 return NGX_OK;
4116 }
4117
4118 #endif
4119
4120
4121 static void
4122 ngx_http_proxy_set_vars(ngx_url_t *u, ngx_http_proxy_vars_t *v)
4123 {
4124     if (u->family != AF_UNIX) {
4125
4126         if (u->no_port || u->port == u->default_port) {
4127
4128             v->host_header = u->host;
4129
4130             if (u->default_port == 80) {
4131                 ngx_str_set(&v->port, "80");
4132             } else {
4133                 ngx_str_set(&v->port, "443");
4134             }
4135         } else {
4136
4137             v->host_header.len = u->host.len + 1 + u->port_text.len;
4138             v->host_header.data = u->host.data;
4139             v->port = u->port_text;
4140         }
4141
4142         v->key_start.len += v->host_header.len;
4143     } else {
4144         ngx_str_set(&v->host_header, "localhost");
4145         ngx_str_null(&v->port);
4146         v->key_start.len += sizeof("unix:") - 1 + u->host.len + 1;
4147     }
4148
4149     v->uri = u->uri;
4150 }
4151
4152 }

```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_random\_index\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_random\\_index\\_commands](#)
- [ngx\\_http\\_random\\_index\\_module](#)
- [ngx\\_http\\_random\\_index\\_module\\_ctx](#)

## Data types defined

- [ngx\\_http\\_random\\_index\\_loc\\_conf\\_t](#)

## Functions defined

- [ngx\\_http\\_random\\_index\\_create\\_loc\\_conf](#)
- [ngx\\_http\\_random\\_index\\_error](#)
- [ngx\\_http\\_random\\_index\\_handler](#)
- [ngx\\_http\\_random\\_index\\_init](#)
- [ngx\\_http\\_random\\_index\\_merge\\_loc\\_conf](#)

## Macros defined

- [NGX\\_HTTP\\_RANDOM\\_INDEX\\_PREALLOCATE](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 typedef struct {
14     ngx_flag_t enable;
15 } ngx_http_random_index_loc_conf_t;
16
17
18 #define NGX_HTTP_RANDOM_INDEX_PREALLOCATE 50
19
20
21 static ngx_int_t ngx_http_random_index_error(ngx_http_request_t *r,
22     ngx_dir_t *dir, ngx_str_t *name);
23 static ngx_int_t ngx_http_random_index_init(ngx_conf_t *cf);
24 static void *ngx_http_random_index_create_loc_conf(ngx_conf_t *cf);
25 static char *ngx_http_random_index_merge_loc_conf(ngx_conf_t *cf,
26     void *parent, void *child);
27
28
29 static ngx_command_t ngx_http_random_index_commands[] = {
30
31     { ngx_string("random_index"),
```



```

32     NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_FLAG,
33     ngx\_conf\_set\_flag\_slot,
34     NGX\_HTTP\_LOC\_CONF\_OFFSET,
35     offsetof\(ngx\_http\_random\_index\_loc\_conf\_t, enable\),
36     NULL },
37
38     ngx\_null\_command
39 };
40
41
42 static ngx\_http\_module\_t ngx\_http\_random\_index\_module\_ctx = {
43     NULL, /* preconfiguration */
44     ngx\_http\_random\_index\_init, /* postconfiguration */
45
46     NULL, /* create main configuration */
47     NULL, /* init main configuration */
48
49     NULL, /* create server configuration */
50     NULL, /* merge server configuration */
51
52     ngx\_http\_random\_index\_create\_loc\_conf, /* create location configuration */
53     ngx\_http\_random\_index\_merge\_loc\_conf /* merge location configuration */
54 };
55
56
57 ngx\_module\_t ngx\_http\_random\_index\_module = {
58     NGX\_MODULE\_V1,
59     &ngx\_http\_random\_index\_module\_ctx, /* module context */
60     ngx\_http\_random\_index\_commands, /* module directives */
61     NGX\_HTTP\_MODULE, /* module type */
62     NULL, /* init master */
63     NULL, /* init module */
64     NULL, /* init process */
65     NULL, /* init thread */
66     NULL, /* exit thread */
67     NULL, /* exit process */
68     NULL, /* exit master */
69     NGX\_MODULE\_V1\_PADDING
70 };
71
72
73 static ngx\_int\_t
74 ngx\_http\_random\_index\_handler(ngx\_http\_request\_t *r)
75 {
76     u\_char *last, *filename;
77     size\_t len, allocated, root;
78     ngx\_err\_t err;
79     ngx\_int\_t rc;
80     ngx\_str\_t path, uri, *name;
81     ngx\_dir\_t dir;
82     ngx\_uint\_t n, level;
83     ngx\_array\_t names;
84     ngx\_http\_random\_index\_loc\_conf\_t *rlcf;
85
86     if (r->uri.data[r->uri.len - 1] != '/') {
87         return NGX\_DECLINED;
88     }
89
90     if (!(r->method & (NGX\_HTTP\_GET|NGX\_HTTP\_HEAD|NGX\_HTTP\_POST))) {
91         return NGX\_DECLINED;
92     }
93
94     rlcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_random\_index\_module);
95
96     if (!rlcf->enable) {
97         return NGX\_DECLINED;
98     }
99
100 #if (NGX\_HAVE\_D\_TYPE)
101     len = NGX\_DIR\_MASK\_LEN;
102 #else
103     len = NGX\_HTTP\_RANDOM\_INDEX\_PREALLOCATE;
104 #endif
105
106     last = ngx\_http\_map\_uri\_to\_path(r, &path, &root, len);
107     if (last == NULL) {

```

```

108     return NGX_HTTP_INTERNAL_SERVER_ERROR;
109 }
110
111 allocated = path.len;
112
113 path.len = last - path.data - 1;
114 path.data[path.len] = '\0';
115
116 ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
117     "http random index: \"%s\"", path.data);
118
119 if (ngx_open_dir(&path, &dir) == NGX_ERROR) {
120     err = ngx_errno;
121
122     if (err == NGX_ENOENT
123         || err == NGX_ENOTDIR
124         || err == NGX_ENAMETOOLONG)
125     {
126         level = NGX_LOG_ERR;
127         rc = NGX_HTTP_NOT_FOUND;
128
129     } else if (err == NGX_EACCES) {
130         level = NGX_LOG_ERR;
131         rc = NGX_HTTP_FORBIDDEN;
132
133     } else {
134         level = NGX_LOG_CRIT;
135         rc = NGX_HTTP_INTERNAL_SERVER_ERROR;
136     }
137
138     ngx_log_error(level, r->connection->log, err,
139         ngx_open_dir_n " \"%s\" failed", path.data);
140
141     return rc;
142 }
143
144 if (ngx_array_init(&names, r->pool, 32, sizeof(ngx_str_t)) != NGX_OK) {
145     return ngx_http_random_index_error(r, &dir, &path);
146 }
147
148 filename = path.data;
149 filename[path.len] = '/';
150
151 for ( ;; ) {
152     ngx_set_errno(0);
153
154     if (ngx_read_dir(&dir) == NGX_ERROR) {
155         err = ngx_errno;
156
157         if (err != NGX_ENOMOREFILES) {
158             ngx_log_error(NGX_LOG_CRIT, r->connection->log, err,
159                 ngx_read_dir_n " \"%s\" failed", &path);
160             return ngx_http_random_index_error(r, &dir, &path);
161         }
162
163         break;
164     }
165
166     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
167         "http random index file: \"%s\"", ngx_de_name(&dir));
168
169     if (ngx_de_name(&dir)[0] == '.') {
170         continue;
171     }
172
173     len = ngx_de_namelen(&dir);
174
175     if (dir.type == 0 || ngx_de_is_link(&dir)) {
176
177         /* 1 byte for '/' and 1 byte for terminating '\0' */
178
179         if (path.len + 1 + len + 1 > allocated) {
180             allocated = path.len + 1 + len + 1
181                 + NGX_HTTP_RANDOM_INDEX_PREALLOCATE;
182
183             filename = ngx_pnalloc(r->pool, allocated);

```

```

184     if (filename == NULL) {
185         return ngx_http_random_index_error(r, &dir, &path);
186     }
187
188     last = ngx_cpystrn(filename, path.data, path.len + 1);
189     *last++ = '/';
190 }
191
192 ngx_cpystrn(last, ngx_de_name(&dir), len + 1);
193
194 if (ngx_de_info(filename, &dir) == NGX_FILE_ERROR) {
195     err = ngx_errno;
196
197     if (err != NGX_ENOENT) {
198         ngx_log_error(NGX_LOG_CRIT, r->connection->log, err,
199             ngx_de_info_n " \"%s\" failed", filename);
200         return ngx_http_random_index_error(r, &dir, &path);
201     }
202
203     if (ngx_de_link_info(filename, &dir) == NGX_FILE_ERROR) {
204         ngx_log_error(NGX_LOG_CRIT, r->connection->log, ngx_errno,
205             ngx_de_link_info_n " \"%s\" failed",
206             filename);
207         return ngx_http_random_index_error(r, &dir, &path);
208     }
209 }
210 }
211
212 if (!ngx_de_is_file(&dir)) {
213     continue;
214 }
215
216 name = ngx_array_push(&names);
217 if (name == NULL) {
218     return ngx_http_random_index_error(r, &dir, &path);
219 }
220
221 name->len = len;
222
223 name->data = ngx_pnalloc(r->pool, len);
224 if (name->data == NULL) {
225     return ngx_http_random_index_error(r, &dir, &path);
226 }
227
228 ngx_memcpy(name->data, ngx_de_name(&dir), len);
229 }
230
231 if (ngx_close_dir(&dir) == NGX_ERROR) {
232     ngx_log_error(NGX_LOG_ALERT, r->connection->log, ngx_errno,
233         ngx_close_dir_n " \"%s\" failed", &path);
234 }
235
236 n = names.nelts;
237
238 if (n == 0) {
239     return NGX_DECLINED;
240 }
241
242 name = names.elts;
243
244 n = (ngx_uint_t) (((uint64_t) ngx_random() * n) / 0x80000000);
245
246 uri.len = r->uri.len + name[n].len;
247
248 uri.data = ngx_pnalloc(r->pool, uri.len);
249 if (uri.data == NULL) {
250     return NGX_HTTP_INTERNAL_SERVER_ERROR;
251 }
252
253 last = ngx_copy(uri.data, r->uri.data, r->uri.len);
254 ngx_memcpy(last, name[n].data, name[n].len);
255
256 return ngx_http_internal_redirect(r, &uri, &r->args);
257 }
258
259

```

```

260 static ngx_int_t
261 ngx_http_random_index_error(ngx_http_request_t *r, ngx_dir_t *dir,
262     ngx_str_t *name)
263 {
264     if (ngx_close_dir(dir) == NGX_ERROR) {
265         ngx_log_error(NGX_LOG_ALERT, r->connection->log, ngx_errno,
266             ngx_close_dir_n "%V" failed", name);
267     }
268
269     return NGX_HTTP_INTERNAL_SERVER_ERROR;
270 }
271
272
273 static void *
274 ngx_http_random_index_create_loc_conf(ngx_conf_t *cf)
275 {
276     ngx_http_random_index_loc_conf_t *conf;
277
278     conf = ngx_palloc(cf->pool, sizeof(ngx_http_random_index_loc_conf_t));
279     if (conf == NULL) {
280         return NULL;
281     }
282
283     conf->enable = NGX_CONF_UNSET;
284
285     return conf;
286 }
287
288
289 static char *
290 ngx_http_random_index_merge_loc_conf(ngx_conf_t *cf, void *parent, void *child)
291 {
292     ngx_http_random_index_loc_conf_t *prev = parent;
293     ngx_http_random_index_loc_conf_t *conf = child;
294
295     ngx_conf_merge_value(conf->enable, prev->enable, 0);
296
297     return NGX_CONF_OK;
298 }
299
300
301 static ngx_int_t
302 ngx_http_random_index_init(ngx_conf_t *cf)
303 {
304     ngx_http_handler_pt *h;
305     ngx_http_core_main_conf_t *cmcf;
306
307     cmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_core_module);
308
309     h = ngx_array_push(&cmcf->phases[NGX_HTTP_CONTENT_PHASE].handlers);
310     if (h == NULL) {
311         return NGX_ERROR;
312     }
313
314     *h = ngx_http_random_index_handler;
315
316     return NGX_OK;
317 }

```

[One Level Up](#)

[Top Level](#)

## src/http/modules/nginx\_http\_range\_filter\_module.c - nginx-1.7.10

### Global variables defined

- [ngx\\_http\\_next\\_body\\_filter](#)
- [ngx\\_http\\_next\\_header\\_filter](#)
- [ngx\\_http\\_range\\_body\\_filter\\_module](#)
- [ngx\\_http\\_range\\_body\\_filter\\_module\\_ctx](#)
- [ngx\\_http\\_range\\_header\\_filter\\_module](#)
- [ngx\\_http\\_range\\_header\\_filter\\_module\\_ctx](#)

### Data types defined

- [ngx\\_http\\_range\\_filter\\_ctx\\_t](#)
- [ngx\\_http\\_range\\_t](#)

### Functions defined

- [ngx\\_http\\_range\\_body\\_filter](#)
- [ngx\\_http\\_range\\_body\\_filter\\_init](#)
- [ngx\\_http\\_range\\_header\\_filter](#)
- [ngx\\_http\\_range\\_header\\_filter\\_init](#)
- [ngx\\_http\\_range\\_multipart\\_body](#)
- [ngx\\_http\\_range\\_multipart\\_header](#)
- [ngx\\_http\\_range\\_not\\_satisfiable](#)
- [ngx\\_http\\_range\\_parse](#)
- [ngx\\_http\\_range\\_singlepart\\_body](#)
- [ngx\\_http\\_range\\_singlepart\\_header](#)
- [ngx\\_http\\_range\\_test\\_overlapped](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 /*
14  * the single part format:
```

```

15 *
16 * "HTTP/1.0 206 Partial Content" CRLF
17 * ... header ...
18 * "Content-Type: image/jpeg" CRLF
19 * "Content-Length: SIZE" CRLF
20 * "Content-Range: bytes START-END/SIZE" CRLF
21 * CRLF
22 * ... data ...
23 *
24 *
25 * the multipart format:
26 *
27 * "HTTP/1.0 206 Partial Content" CRLF
28 * ... header ...
29 * "Content-Type: multipart/byteranges; boundary=0123456789" CRLF
30 * CRLF
31 * CRLF
32 * "--0123456789" CRLF
33 * "Content-Type: image/jpeg" CRLF
34 * "Content-Range: bytes START0-END0/SIZE" CRLF
35 * CRLF
36 * ... data ...
37 * CRLF
38 * "--0123456789" CRLF
39 * "Content-Type: image/jpeg" CRLF
40 * "Content-Range: bytes START1-END1/SIZE" CRLF
41 * CRLF
42 * ... data ...
43 * CRLF
44 * "--0123456789--" CRLF
45 */
46
47
48 typedef struct {
49     off_t      start;
50     off_t      end;
51     ngx_str_t  content_range;
52 } ngx_http_range_t;
53
54
55 typedef struct {
56     off_t      offset;
57     ngx_str_t  boundary_header;
58     ngx_array_t ranges;
59 } ngx_http_range_filter_ctx_t;
60
61
62 static ngx_int_t ngx_http_range_parse(ngx_http_request_t *r,
63     ngx_http_range_filter_ctx_t *ctx, ngx_uint_t ranges);
64 static ngx_int_t ngx_http_range_singlepart_header(ngx_http_request_t *r,
65     ngx_http_range_filter_ctx_t *ctx);
66 static ngx_int_t ngx_http_range_multipart_header(ngx_http_request_t *r,
67     ngx_http_range_filter_ctx_t *ctx);
68 static ngx_int_t ngx_http_range_not_satisfiable(ngx_http_request_t *r);
69 static ngx_int_t ngx_http_range_test_overlapped(ngx_http_request_t *r,
70     ngx_http_range_filter_ctx_t *ctx, ngx_chain_t *in);
71 static ngx_int_t ngx_http_range_singlepart_body(ngx_http_request_t *r,
72     ngx_http_range_filter_ctx_t *ctx, ngx_chain_t *in);
73 static ngx_int_t ngx_http_range_multipart_body(ngx_http_request_t *r,
74     ngx_http_range_filter_ctx_t *ctx, ngx_chain_t *in);
75
76 static ngx_int_t ngx_http_range_header_filter_init(ngx_conf_t *cf);
77 static ngx_int_t ngx_http_range_body_filter_init(ngx_conf_t *cf);
78
79
80 static ngx_http_module_t ngx_http_range_header_filter_module_ctx = {
81     NULL, /* preconfiguration */
82     ngx_http_range_header_filter_init, /* postconfiguration */
83
84     NULL, /* create main configuration */
85     NULL, /* init main configuration */
86
87     NULL, /* create server configuration */
88     NULL, /* merge server configuration */
89
90     NULL, /* create location configuration */

```

```

91     NULL,                                /* merge location configuration */
92 };
93
94
95 ngx_module_t ngx_http_range_header_filter_module = {
96     NGX_MODULE_V1,
97     &ngx_http_range_header_filter_module_ctx, /* module context */
98     NULL,                                    /* module directives */
99     NGX_HTTP_MODULE,                        /* module type */
100    NULL,                                    /* init master */
101    NULL,                                    /* init module */
102    NULL,                                    /* init process */
103    NULL,                                    /* init thread */
104    NULL,                                    /* exit thread */
105    NULL,                                    /* exit process */
106    NULL,                                    /* exit master */
107     NGX_MODULE_V1_PADDING
108 };
109
110
111 static ngx_http_module_t ngx_http_range_body_filter_module_ctx = {
112     NULL,                                    /* preconfiguration */
113     ngx_http_range_body_filter_init,         /* postconfiguration */
114
115     NULL,                                    /* create main configuration */
116     NULL,                                    /* init main configuration */
117
118     NULL,                                    /* create server configuration */
119     NULL,                                    /* merge server configuration */
120
121     NULL,                                    /* create location configuration */
122     NULL,                                    /* merge location configuration */
123 };
124
125
126 ngx_module_t ngx_http_range_body_filter_module = {
127     NGX_MODULE_V1,
128     &ngx_http_range_body_filter_module_ctx, /* module context */
129     NULL,                                    /* module directives */
130     NGX_HTTP_MODULE,                        /* module type */
131     NULL,                                    /* init master */
132     NULL,                                    /* init module */
133     NULL,                                    /* init process */
134     NULL,                                    /* init thread */
135     NULL,                                    /* exit thread */
136     NULL,                                    /* exit process */
137     NULL,                                    /* exit master */
138     NGX_MODULE_V1_PADDING
139 };
140
141
142 static ngx_http_output_header_filter_pt ngx_http_next_header_filter;
143 static ngx_http_output_body_filter_pt  ngx_http_next_body_filter;
144
145
146 static ngx_int_t
147 ngx_http_range_header_filter(ngx_http_request_t *r)
148 {
149     time_t          if_range_time;
150     ngx_str_t       *if_range, *etag;
151     ngx_uint_t      ranges;
152     ngx_http_core_loc_conf_t *clcf;
153     ngx_http_range_filter_ctx_t *ctx;
154
155     if (r->http_version < NGX_HTTP_VERSION_10
156         || r->headers_out.status != NGX_HTTP_OK
157         || r != r->main
158         || r->headers_out.content_length_n == -1
159         || !r->allow_ranges)
160     {
161         return ngx_http_next_header_filter(r);
162     }
163
164     clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
165
166     if (clcf->max_ranges == 0) {

```

```

167     return ngx_http_next_header_filter(r);
168 }
169
170 if (r->headers_in.range == NULL
171     || r->headers_in.range->value.len < 7
172     || ngx_strncasecmp(r->headers_in.range->value.data,
173                       (u_char *) "bytes=", 6)
174     != 0)
175 {
176     goto next_filter;
177 }
178
179 if (r->headers_in.if_range) {
180
181     if_range = &r->headers_in.if_range->value;
182
183     if (if_range->len >= 2 && if_range->data[if_range->len - 1] == '"') {
184
185         if (r->headers_out.etag == NULL) {
186             goto next_filter;
187         }
188
189         etag = &r->headers_out.etag->value;
190
191         ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
192                       "http ir:%V etag:%V", if_range, etag);
193
194         if (if_range->len != etag->len
195             || ngx_strncmp(if_range->data, etag->data, etag->len) != 0)
196         {
197             goto next_filter;
198         }
199
200         goto parse;
201     }
202
203     if (r->headers_out.last_modified_time == (time_t) -1) {
204         goto next_filter;
205     }
206
207     if_range_time = ngx_http_parse_time(if_range->data, if_range->len);
208
209     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
210                   "http ir:%d lm:%d",
211                   if_range_time, r->headers_out.last_modified_time);
212
213     if (if_range_time != r->headers_out.last_modified_time) {
214         goto next_filter;
215     }
216 }
217
218 parse:
219
220 ctx = ngx_palloc(r->pool, sizeof(ngx_http_range_filter_ctx_t));
221 if (ctx == NULL) {
222     return NGX_ERROR;
223 }
224
225 if (ngx_array_init(&ctx->ranges, r->pool, 1, sizeof(ngx_http_range_t))
226     != NGX_OK)
227 {
228     return NGX_ERROR;
229 }
230
231 ranges = r->single_range ? 1 : clcf->max_ranges;
232
233 switch (ngx_http_range_parse(r, ctx, ranges)) {
234
235 case NGX_OK:
236     ngx_http_set_ctx(r, ctx, ngx_http_range_body_filter_module);
237
238     r->headers_out.status = NGX_HTTP_PARTIAL_CONTENT;
239     r->headers_out.status_line.len = 0;
240
241     if (ctx->ranges.nelts == 1) {
242         return ngx_http_range_singlepart_header(r, ctx);

```



```

243     }
244
245     return ngx_http_range_multipart_header(r, ctx);
246
247     case NGX_HTTP_RANGE_NOT_SATISFIABLE:
248         return ngx_http_range_not_satisfiable(r);
249
250     case NGX_ERROR:
251         return NGX_ERROR;
252
253     default: /* NGX_DECLINED */
254         break;
255     }
256
257 next_filter:
258
259     r->headers_out.accept_ranges = ngx_list_push(&r->headers_out.headers);
260     if (r->headers_out.accept_ranges == NULL) {
261         return NGX_ERROR;
262     }
263
264     r->headers_out.accept_ranges->hash = 1;
265     ngx_str_set(&r->headers_out.accept_ranges->key, "Accept-Ranges");
266     ngx_str_set(&r->headers_out.accept_ranges->value, "bytes");
267
268     return ngx_http_next_header_filter(r);
269 }
270
271
272 static ngx_int_t
273 ngx_http_range_parse(ngx_http_request_t *r, ngx_http_range_filter_ctx_t *ctx,
274 ngx_uint_t ranges)
275 {
276     u_char          *p;
277     off_t           start, end, size, content_length;
278     ngx_uint_t      suffix;
279     ngx_http_range_t *range;
280
281     p = r->headers_in.range->value.data + 6;
282     size = 0;
283     content_length = r->headers_out.content_length_n;
284
285     for ( ;; ) {
286         start = 0;
287         end = 0;
288         suffix = 0;
289
290         while (*p == ' ') { p++; }
291
292         if (*p != '-') {
293             if (*p < '0' || *p > '9') {
294                 return NGX_HTTP_RANGE_NOT_SATISFIABLE;
295             }
296
297             while (*p >= '0' && *p <= '9') {
298                 start = start * 10 + *p++ - '0';
299             }
300
301             while (*p == ' ') { p++; }
302
303             if (*p++ != '-') {
304                 return NGX_HTTP_RANGE_NOT_SATISFIABLE;
305             }
306
307             while (*p == ' ') { p++; }
308
309             if (*p == ',' || *p == '\\0') {
310                 end = content_length;
311                 goto found;
312             }
313
314         } else {
315             suffix = 1;
316             p++;
317         }
318     }

```

```

319     if (*p < '0' || *p > '9') {
320         return NGX\_HTTP\_RANGE\_NOT\_SATISFIABLE;
321     }
322
323     while (*p >= '0' && *p <= '9') {
324         end = end * 10 + *p++ - '0';
325     }
326
327     while (*p == ' ') { p++; }
328
329     if (*p != ',' && *p != '\\0') {
330         return NGX\_HTTP\_RANGE\_NOT\_SATISFIABLE;
331     }
332
333     if (suffix) {
334         start = content_length - end;
335         end = content_length - 1;
336     }
337
338     if (end >= content_length) {
339         end = content_length;
340
341     } else {
342         end++;
343     }
344
345     found:
346
347     if (start < end) {
348         range = ngx\_array\_push(&ctx->ranges);
349         if (range == NULL) {
350             return NGX\_ERROR;
351         }
352
353         range->start = start;
354         range->end = end;
355
356         size += end - start;
357
358         if (ranges-- == 0) {
359             return NGX\_DECLINED;
360         }
361     }
362
363     if (*p++ != ',') {
364         break;
365     }
366 }
367
368 if (ctx->ranges.nelts == 0) {
369     return NGX\_HTTP\_RANGE\_NOT\_SATISFIABLE;
370 }
371
372 if (size > content_length) {
373     return NGX\_DECLINED;
374 }
375
376 return NGX\_OK;
377 }
378
379
380 static ngx\_int\_t
381 ngx\_http\_range\_singlepart\_header(ngx\_http\_request\_t *r,
382     ngx\_http\_range\_filter\_ctx\_t *ctx)
383 {
384     ngx\_table\_elt\_t *content_range;
385     ngx\_http\_range\_t *range;
386
387     content_range = ngx\_list\_push(&r->headers_out.headers);
388     if (content_range == NULL) {
389         return NGX\_ERROR;
390     }
391
392     r->headers_out.content_range = content_range;
393
394     content_range->hash = 1;

```



```

471 } else if (r->headers_out.content_type.len) {
472     ctx->boundary_header.len = ngx_sprintf(ctx->boundary_header.data,
473         CRLF "--%0muA" CRLF
474         "Content-Type: %V" CRLF
475         "Content-Range: bytes ",
476         boundary,
477         &r->headers_out.content_type)
478     - ctx->boundary_header.data;
479
480 } else {
481     ctx->boundary_header.len = ngx_sprintf(ctx->boundary_header.data,
482         CRLF "--%0muA" CRLF
483         "Content-Range: bytes ",
484         boundary)
485     - ctx->boundary_header.data;
486 }
487
488 r->headers_out.content_type.data =
489     ngx_pnalloc(r->pool,
490         sizeof("Content-Type: multipart/byteranges; boundary=") - 1
491         + NGX_ATOMIC_T_LEN);
492
493 if (r->headers_out.content_type.data == NULL) {
494     return NGX_ERROR;
495 }
496
497 r->headers_out.content_type_lowercase = NULL;
498
499 /* "Content-Type: multipart/byteranges; boundary=0123456789" */
500
501 r->headers_out.content_type.len =
502     ngx_sprintf(r->headers_out.content_type.data,
503         "multipart/byteranges; boundary=%0muA",
504         boundary)
505     - r->headers_out.content_type.data;
506
507 r->headers_out.content_type_len = r->headers_out.content_type.len;
508
509 r->headers_out.charset.len = 0;
510
511 /* the size of the last boundary CRLF "--0123456789--" CRLF */
512
513 len = sizeof(CRLF "--") - 1 + NGX_ATOMIC_T_LEN + sizeof("--" CRLF) - 1;
514
515 range = ctx->ranges.elts;
516 for (i = 0; i < ctx->ranges.nelts; i++) {
517
518     /* the size of the range: "SSSS-EEEE/TTTT" CRLF CRLF */
519
520     range[i].content_range.data =
521         ngx_pnalloc(r->pool, 3 * NGX_OFF_T_LEN + 2 + 4);
522
523     if (range[i].content_range.data == NULL) {
524         return NGX_ERROR;
525     }
526
527     range[i].content_range.len = ngx_sprintf(range[i].content_range.data,
528         "%0-%0/%0" CRLF CRLF,
529         range[i].start, range[i].end - 1,
530         r->headers_out.content_length_n)
531         - range[i].content_range.data;
532
533     len += ctx->boundary_header.len + range[i].content_range.len
534         + (size_t) (range[i].end - range[i].start);
535 }
536
537 r->headers_out.content_length_n = len;
538
539 if (r->headers_out.content_length) {
540     r->headers_out.content_length->hash = 0;
541     r->headers_out.content_length = NULL;
542 }
543
544 return ngx_http_next_header_filter(r);
545 }
546

```

```

547 static ngx_int_t
548 ngx_http_range_not_satisfiable(ngx_http_request_t *r)
549 {
550     ngx_table_elt_t *content_range;
551
552     r->headers_out.status = NGX_HTTP_RANGE_NOT_SATISFIABLE;
553
554     content_range = ngx_list_push(&r->headers_out.headers);
555     if (content_range == NULL) {
556         return NGX_ERROR;
557     }
558
559     r->headers_out.content_range = content_range;
560
561     content_range->hash = 1;
562     ngx_str_set(&content_range->key, "Content-Range");
563
564     content_range->value.data = ngx_pnalloc(r->pool,
565                                             sizeof("bytes */") - 1 + NGX_OFF_T_LEN);
566     if (content_range->value.data == NULL) {
567         return NGX_ERROR;
568     }
569
570     content_range->value.len = ngx_sprintf(content_range->value.data,
571                                           "bytes */%0",
572                                           r->headers_out.content_length_n)
573                               - content_range->value.data;
574
575     ngx_http_clear_content_length(r);
576
577     return NGX_HTTP_RANGE_NOT_SATISFIABLE;
578 }
579
580
581 static ngx_int_t
582 ngx_http_range_body_filter(ngx_http_request_t *r, ngx_chain_t *in)
583 {
584     ngx_http_range_filter_ctx_t *ctx;
585
586     if (in == NULL) {
587         return ngx_http_next_body_filter(r, in);
588     }
589
590     ctx = ngx_http_get_module_ctx(r, ngx_http_range_body_filter_module);
591
592     if (ctx == NULL) {
593         return ngx_http_next_body_filter(r, in);
594     }
595
596     if (ctx->ranges.nelts == 1) {
597         return ngx_http_range_singlepart_body(r, ctx, in);
598     }
599
600     /*
601      * multipart ranges are supported only if whole body is in a single buffer
602      */
603
604     if (ngx_buf_special(in->buf)) {
605         return ngx_http_next_body_filter(r, in);
606     }
607
608     if (ngx_http_range_test_overlapped(r, ctx, in) != NGX_OK) {
609         return NGX_ERROR;
610     }
611
612     return ngx_http_range_multipart_body(r, ctx, in);
613 }
614
615
616 static ngx_int_t
617 ngx_http_range_test_overlapped(ngx_http_request_t *r,
618                                ngx_http_range_filter_ctx_t *ctx, ngx_chain_t *in)
619 {
620     off_t start, last;
621     ngx_buf_t *buf;

```

```

623     ngx_uint_t      i;
624     ngx_http_range_t *range;
625
626     if (ctx->offset) {
627         goto overlapped;
628     }
629
630     buf = in->buf;
631
632     if (!buf->last_buf) {
633         start = ctx->offset;
634         last = ctx->offset + ngx_buf_size(buf);
635
636         range = ctx->ranges.elts;
637         for (i = 0; i < ctx->ranges.nelts; i++) {
638             if (start > range[i].start || last < range[i].end) {
639                 goto overlapped;
640             }
641         }
642     }
643
644     ctx->offset = ngx_buf_size(buf);
645
646     return NGX_OK;
647
648 overlapped:
649
650     ngx_log_error(NGX_LOG_ALERT, r->connection->log, 0,
651                 "range in overlapped buffers");
652
653     return NGX_ERROR;
654 }
655
656
657 static ngx_int_t
658 ngx_http_range_singlepart_body(ngx_http_request_t *r,
659                               ngx_http_range_filter_ctx_t *ctx, ngx_chain_t *in)
660 {
661     off_t      start, last;
662     ngx_buf_t  *buf;
663     ngx_chain_t *out, *cl, **ll;
664     ngx_http_range_t *range;
665
666     out = NULL;
667     ll = &out;
668     range = ctx->ranges.elts;
669
670     for (cl = in; cl; cl = cl->next) {
671         buf = cl->buf;
672
673         start = ctx->offset;
674         last = ctx->offset + ngx_buf_size(buf);
675
676         ctx->offset = last;
677
678         ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
679                     "http range body buf: %0-%0", start, last);
680
681         if (ngx_buf_special(buf)) {
682             *ll = cl;
683             ll = &cl->next;
684             continue;
685         }
686     }
687
688     if (range->end <= start || range->start >= last) {
689
690         ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
691                     "http range body skip");
692
693         if (buf->in_file) {
694             buf->file_pos = buf->file_last;
695         }
696
697         buf->pos = buf->last;
698         buf->sync = 1;

```

```

699     continue;
700 }
701
702
703 if (range->start > start) {
704     if (buf->in_file) {
705         buf->file_pos += range->start - start;
706     }
707
708     if (ngx_buf_in_memory(buf)) {
709         buf->pos += (size_t) (range->start - start);
710     }
711 }
712
713
714 if (range->end <= last) {
715     if (buf->in_file) {
716         buf->file_last -= last - range->end;
717     }
718
719     if (ngx_buf_in_memory(buf)) {
720         buf->last -= (size_t) (last - range->end);
721     }
722
723     buf->last_buf = 1;
724     *ll = cl;
725     cl->next = NULL;
726
727     break;
728 }
729
730 *ll = cl;
731 ll = &cl->next;
732 }
733
734 if (out == NULL) {
735     return NGX_OK;
736 }
737
738 return ngx_http_next_body_filter(r, out);
739 }
740
741
742
743 static ngx_int_t
744 ngx_http_range_multipart_body(ngx_http_request_t *r,
745     ngx_http_range_filter_ctx_t *ctx, ngx_chain_t *in)
746 {
747     ngx_buf_t      *b, *buf;
748     ngx_uint_t      i;
749     ngx_chain_t     *out, *hcl, *rcl, *dcl, **ll;
750     ngx_http_range_t *range;
751
752     ll = &out;
753     buf = in->buf;
754     range = ctx->ranges.elts;
755
756     for (i = 0; i < ctx->ranges.nelts; i++) {
757
758         /*
759          * The boundary header of the range:
760          * CRLF
761          * "--0123456789" CRLF
762          * "Content-Type: image/jpeg" CRLF
763          * "Content-Range: bytes "
764          */
765
766         b = ngx_calloc_buf(r->pool);
767         if (b == NULL) {
768             return NGX_ERROR;
769         }
770
771         b->memory = 1;
772         b->pos = ctx->boundary_header.data;
773         b->last = ctx->boundary_header.data + ctx->boundary_header.len;
774

```

```

775     hcl = ngx_alloc_chain_link(r->pool);
776     if (hcl == NULL) {
777         return NGX_ERROR;
778     }
779
780     hcl->buf = b;
781
782
783     /* "SSSS-EEEE/TTTT" CRLF CRLF */
784
785     b = ngx_calloc_buf(r->pool);
786     if (b == NULL) {
787         return NGX_ERROR;
788     }
789
790     b->temporary = 1;
791     b->pos = range[i].content_range.data;
792     b->last = range[i].content_range.data + range[i].content_range.len;
793
794     rcl = ngx_alloc_chain_link(r->pool);
795     if (rcl == NULL) {
796         return NGX_ERROR;
797     }
798
799     rcl->buf = b;
800
801
802     /* the range data */
803
804     b = ngx_calloc_buf(r->pool);
805     if (b == NULL) {
806         return NGX_ERROR;
807     }
808
809     b->in_file = buf->in_file;
810     b->temporary = buf->temporary;
811     b->memory = buf->memory;
812     b->mmap = buf->mmap;
813     b->file = buf->file;
814
815     if (buf->in_file) {
816         b->file_pos = buf->file_pos + range[i].start;
817         b->file_last = buf->file_pos + range[i].end;
818     }
819
820     if (ngx_buf_in_memory(buf)) {
821         b->pos = buf->pos + (size_t) range[i].start;
822         b->last = buf->pos + (size_t) range[i].end;
823     }
824
825     dcl = ngx_alloc_chain_link(r->pool);
826     if (dcl == NULL) {
827         return NGX_ERROR;
828     }
829
830     dcl->buf = b;
831
832     *ll = hcl;
833     hcl->next = rcl;
834     rcl->next = dcl;
835     ll = &dcl->next;
836 }
837
838 /* the last boundary CRLF "--0123456789--" CRLF */
839
840 b = ngx_calloc_buf(r->pool);
841 if (b == NULL) {
842     return NGX_ERROR;
843 }
844
845 b->temporary = 1;
846 b->last_buf = 1;
847
848 b->pos = ngx_pnalloc(r->pool, sizeof(CRLF "--") - 1 + NGX_ATOMIC_T_LEN
849     + sizeof("--" CRLF) - 1);
850 if (b->pos == NULL) {

```



```

851     return NGX\_ERROR;
852 }
853
854 b->last = ngx\_cpymem(b->pos, ctx->boundary_header.data,
855     sizeof(CRLF "--") - 1 + NGX\_ATOMIC\_T\_LEN);
856 *b->last++ = '-'; *b->last++ = '-';
857 *b->last++ = CR; *b->last++ = LF;
858
859 hc1 = ngx\_alloc\_chain\_link(r->pool);
860 if (hc1 == NULL) {
861     return NGX\_ERROR;
862 }
863
864 hc1->buf = b;
865 hc1->next = NULL;
866
867 *ll = hc1;
868
869 return ngx\_http\_next\_body\_filter(r, out);
870 }
871
872
873 static ngx\_int\_t
874 ngx\_http\_range\_header\_filter\_init(ngx\_conf\_t *cf)
875 {
876     ngx\_http\_next\_header\_filter = ngx\_http\_top\_header\_filter;
877     ngx\_http\_top\_header\_filter = ngx\_http\_range\_header\_filter;
878
879     return NGX\_OK;
880 }
881
882
883 static ngx\_int\_t
884 ngx\_http\_range\_body\_filter\_init(ngx\_conf\_t *cf)
885 {
886     ngx\_http\_next\_body\_filter = ngx\_http\_top\_body\_filter;
887     ngx\_http\_top\_body\_filter = ngx\_http\_range\_body\_filter;
888
889     return NGX\_OK;
890 }

```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_realip\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_realip\\_commands](#)
- [ngx\\_http\\_realip\\_module](#)
- [ngx\\_http\\_realip\\_module\\_ctx](#)

## Data types defined

- [ngx\\_http\\_realip\\_ctx\\_t](#)
- [ngx\\_http\\_realip\\_loc\\_conf\\_t](#)

## Functions defined

- [ngx\\_http\\_realip](#)
- [ngx\\_http\\_realip\\_cleanup](#)
- [ngx\\_http\\_realip\\_create\\_loc\\_conf](#)
- [ngx\\_http\\_realip\\_from](#)
- [ngx\\_http\\_realip\\_handler](#)
- [ngx\\_http\\_realip\\_init](#)
- [ngx\\_http\\_realip\\_merge\\_loc\\_conf](#)
- [ngx\\_http\\_realip\\_set\\_addr](#)

## Macros defined

- [NGX\\_HTTP\\_REALIP\\_HEADER](#)
- [NGX\\_HTTP\\_REALIP\\_PROXY](#)
- [NGX\\_HTTP\\_REALIP\\_XFWD](#)
- [NGX\\_HTTP\\_REALIP\\_XREALIP](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 #define NGX_HTTP_REALIP_XREALIP 0
14 #define NGX_HTTP_REALIP_XFWD 1
15 #define NGX_HTTP_REALIP_HEADER 2
```

```

16 #define NGX_HTTP_REALIP_PROXY 3
17
18
19 typedef struct {
20     ngx_array_t    *from;    /* array of ngx_cidr_t */
21     ngx_uint_t     type;
22     ngx_uint_t     hash;
23     ngx_str_t      header;
24     ngx_flag_t     recursive;
25 } ngx_http_realip_loc_conf_t;
26
27
28 typedef struct {
29     ngx_connection_t *connection;
30     struct sockaddr *sockaddr;
31     socklen_t       socklen;
32     ngx_str_t       addr_text;
33 } ngx_http_realip_ctx_t;
34
35
36 static ngx_int_t ngx_http_realip_handler(ngx_http_request_t *r);
37 static ngx_int_t ngx_http_realip_set_addr(ngx_http_request_t *r,
38     ngx_addr_t *addr);
39 static void ngx_http_realip_cleanup(void *data);
40 static char *ngx_http_realip_from(ngx_conf_t *cf, ngx_command_t *cmd,
41     void *conf);
42 static char *ngx_http_realip(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
43 static void *ngx_http_realip_create_loc_conf(ngx_conf_t *cf);
44 static char *ngx_http_realip_merge_loc_conf(ngx_conf_t *cf,
45     void *parent, void *child);
46 static ngx_int_t ngx_http_realip_init(ngx_conf_t *cf);
47
48
49 static ngx_command_t ngx_http_realip_commands[] = {
50
51     { ngx_string("set_real_ip_from"),
52       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
53       ngx_http_realip_from,
54       NGX_HTTP_LOC_CONF_OFFSET,
55       0,
56       NULL },
57
58     { ngx_string("real_ip_header"),
59       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
60       ngx_http_realip,
61       NGX_HTTP_LOC_CONF_OFFSET,
62       0,
63       NULL },
64
65     { ngx_string("real_ip_recursive"),
66       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
67       ngx_conf_set_flag_slot,
68       NGX_HTTP_LOC_CONF_OFFSET,
69       offsetof(ngx_http_realip_loc_conf_t, recursive),
70       NULL },
71
72     ngx_null_command
73 };
74
75
76
77 static ngx_http_module_t ngx_http_realip_module_ctx = {
78     NULL,                                /* preconfiguration */
79     ngx_http_realip_init,                 /* postconfiguration */
80
81     NULL,                                  /* create main configuration */
82     NULL,                                  /* init main configuration */
83
84     NULL,                                  /* create server configuration */
85     NULL,                                  /* merge server configuration */
86
87     ngx_http_realip_create_loc_conf,      /* create location configuration */
88     ngx_http_realip_merge_loc_conf       /* merge location configuration */
89 };
90
91

```

```

92 ngx_module_t ngx_http_realip_module = {
93     NGX_MODULE_V1,
94     &ngx_http_realip_module_ctx,          /* module context */
95     ngx_http_realip_commands,           /* module directives */
96     NGX_HTTP_MODULE,                   /* module type */
97     NULL,                               /* init master */
98     NULL,                               /* init module */
99     NULL,                               /* init process */
100    NULL,                               /* init thread */
101    NULL,                               /* exit thread */
102    NULL,                               /* exit process */
103    NULL,                               /* exit master */
104    NGX_MODULE_V1_PADDING
105 };
106
107
108 static ngx_int_t
109 ngx_http_realip_handler(ngx_http_request_t *r)
110 {
111     u_char                *p;
112     size_t                len;
113     ngx_str_t             *value;
114     ngx_uint_t            i, hash;
115     ngx_addr_t            addr;
116     ngx_array_t           *xfwd;
117     ngx_list_part_t       *part;
118     ngx_table_elt_t       *header;
119     ngx_connection_t      *c;
120     ngx_http_realip_ctx_t *ctx;
121     ngx_http_realip_loc_conf_t *rlcf;
122
123     ctx = ngx_http_get_module_ctx(r, ngx_http_realip_module);
124
125     if (ctx) {
126         return NGX_DECLINED;
127     }
128
129     rlcf = ngx_http_get_module_loc_conf(r, ngx_http_realip_module);
130
131     if (rlcf->from == NULL) {
132         return NGX_DECLINED;
133     }
134
135     switch (rlcf->type) {
136
137     case NGX_HTTP_REALIP_XREALIP:
138
139         if (r->headers_in.x_real_ip == NULL) {
140             return NGX_DECLINED;
141         }
142
143         value = &r->headers_in.x_real_ip->value;
144         xfwd = NULL;
145
146         break;
147
148     case NGX_HTTP_REALIP_XFWD:
149
150         xfwd = &r->headers_in.x_forwarded_for;
151
152         if (xfwd->elts == NULL) {
153             return NGX_DECLINED;
154         }
155
156         value = NULL;
157
158         break;
159
160     case NGX_HTTP_REALIP_PROXY:
161
162         value = &r->connection->proxy_protocol_addr;
163
164         if (value->len == 0) {
165             return NGX_DECLINED;
166         }
167

```

```

168     xfwd = NULL;
169
170     break;
171
172     default: /* NGX\_HTTP\_REALIP\_HEADER */
173
174     part = &r->headers_in.headers.part;
175     header = part->elts;
176
177     hash = rlcfc->hash;
178     len = rlcfc->header.len;
179     p = rlcfc->header.data;
180
181     for (i = 0; /* void */ ; i++) {
182
183         if (i >= part->nelts) {
184             if (part->next == NULL) {
185                 break;
186             }
187
188             part = part->next;
189             header = part->elts;
190             i = 0;
191         }
192
193         if (hash == header[i].hash
194             && len == header[i].key.len
195             && ngx\_strncmp(p, header[i].lowcase_key, len) == 0)
196         {
197             value = &header[i].value;
198             xfwd = NULL;
199
200             goto found;
201         }
202     }
203
204     return NGX\_DECLINED;
205 }
206
207 found:
208
209     c = r->connection;
210
211     addr.sockaddr = c->sockaddr;
212     addr.socklen = c->socklen;
213     /* addr.name = c->addr_text; */
214
215     if (ngx\_http\_get\_forwarded\_addr(r, &addr, xfwd, value, rlcfc->from,
216                                     rlcfc->recursive)
217         != NGX\_DECLINED)
218     {
219         return ngx\_http\_realip\_set\_addr(r, &addr);
220     }
221
222     return NGX\_DECLINED;
223 }
224
225
226 static ngx\_int\_t
227 ngx\_http\_realip\_set\_addr(ngx\_http\_request\_t *r, ngx\_addr\_t *addr)
228 {
229     size\_t          len;
230     u\_char         *p;
231     u\_char         text[NGX\_SOCKADDR\_STRLEN];
232     ngx\_connection\_t *c;
233     ngx\_pool\_cleanup\_t *cfn;
234     ngx\_http\_realip\_ctx\_t *ctx;
235
236     cfn = ngx\_pool\_cleanup\_add(r->pool, sizeof(ngx\_http\_realip\_ctx\_t));
237     if (cfn == NULL) {
238         return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
239     }
240
241     ctx = cfn->data;
242     ngx\_http\_set\_ctx(r, ctx, ngx\_http\_realip\_module);
243

```

```

244     c = r->connection;
245
246     len = ngx_sock_ntop(addr->sockaddr, addr->socklen, text,
247                        NGX_SOCKADDR_STRLEN, 0);
248     if (len == 0) {
249         return NGX_HTTP_INTERNAL_SERVER_ERROR;
250     }
251
252     p = ngx_pnalloc(c->pool, len);
253     if (p == NULL) {
254         return NGX_HTTP_INTERNAL_SERVER_ERROR;
255     }
256
257     ngx_memcpy(p, text, len);
258
259     cln->handler = ngx_http_realip_cleanup;
260
261     ctx->connection = c;
262     ctx->sockaddr = c->sockaddr;
263     ctx->socklen = c->socklen;
264     ctx->addr_text = c->addr_text;
265
266     c->sockaddr = addr->sockaddr;
267     c->socklen = addr->socklen;
268     c->addr_text.len = len;
269     c->addr_text.data = p;
270
271     return NGX_DECLINED;
272 }
273
274
275 static void
276 ngx_http_realip_cleanup(void *data)
277 {
278     ngx_http_realip_ctx_t *ctx = data;
279
280     ngx_connection_t *c;
281
282     c = ctx->connection;
283
284     c->sockaddr = ctx->sockaddr;
285     c->socklen = ctx->socklen;
286     c->addr_text = ctx->addr_text;
287 }
288
289
290 static char *
291 ngx_http_realip_from(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
292 {
293     ngx_http_realip_loc_conf_t *rlcf = conf;
294
295     ngx_int_t          rc;
296     ngx_str_t          *value;
297     ngx_cidr_t          *cidr;
298
299     value = cf->args->elts;
300
301     if (rlcf->from == NULL) {
302         rlcf->from = ngx_array_create(cf->pool, 2,
303                                     sizeof(ngx_cidr_t));
304         if (rlcf->from == NULL) {
305             return NGX_CONF_ERROR;
306         }
307     }
308
309     cidr = ngx_array_push(rlcf->from);
310     if (cidr == NULL) {
311         return NGX_CONF_ERROR;
312     }
313
314     #if (NGX_HAVE_UNIX_DOMAIN)
315
316     if (ngx_strcmp(value[1].data, "unix:") == 0) {
317         cidr->family = AF_UNIX;
318         return NGX_CONF_OK;
319     }

```

```

320
321 #endif
322
323     rc = ngx_ptocidr(&value[1], cidr);
324
325     if (rc == NGX_ERROR) {
326         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0, "invalid parameter \"%V\"",
327             &value[1]);
328         return NGX_CONF_ERROR;
329     }
330
331     if (rc == NGX_DONE) {
332         ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
333             "low address bits of %V are meaningless", &value[1]);
334     }
335
336     return NGX_CONF_OK;
337 }
338
339 static char *
340 ngx_http_realip(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
341 {
342     ngx_http_realip_loc_conf_t *rlcf = conf;
343
344     ngx_str_t *value;
345
346     value = cf->args->elts;
347
348     if (ngx_strcmp(value[1].data, "X-Real-IP") == 0) {
349         rlcf->type = NGX_HTTP_REALIP_XREALIP;
350         return NGX_CONF_OK;
351     }
352
353     if (ngx_strcmp(value[1].data, "X-Forwarded-For") == 0) {
354         rlcf->type = NGX_HTTP_REALIP_XFWD;
355         return NGX_CONF_OK;
356     }
357
358     if (ngx_strcmp(value[1].data, "proxy_protocol") == 0) {
359         rlcf->type = NGX_HTTP_REALIP_PROXY;
360         return NGX_CONF_OK;
361     }
362
363     rlcf->type = NGX_HTTP_REALIP_HEADER;
364     rlcf->hash = ngx_hash_strlow(value[1].data, value[1].data, value[1].len);
365     rlcf->header = value[1];
366
367     return NGX_CONF_OK;
368 }
369
370
371 static void *
372 ngx_http_realip_create_loc_conf(ngx_conf_t *cf)
373 {
374     ngx_http_realip_loc_conf_t *conf;
375
376     conf = ngx_palloc(cf->pool, sizeof(ngx_http_realip_loc_conf_t));
377     if (conf == NULL) {
378         return NULL;
379     }
380
381     /*
382     * set by ngx_palloc():
383     *
384     *     conf->from = NULL;
385     *     conf->hash = 0;
386     *     conf->header = { 0, NULL };
387     */
388
389     conf->type = NGX_CONF_UNSET_UINT;
390     conf->recursive = NGX_CONF_UNSET;
391
392     return conf;
393 }
394 }
395

```

```

396 static char *
397 ngx_http_realip_merge_loc_conf(ngx_conf_t *cf, void *parent, void *child)
398 {
399     ngx_http_realip_loc_conf_t *prev = parent;
400     ngx_http_realip_loc_conf_t *conf = child;
401
402     if (conf->from == NULL) {
403         conf->from = prev->from;
404     }
405
406     ngx_conf_merge_uint_value(conf->type, prev->type, NGX_HTTP_REALIP_XREALIP);
407     ngx_conf_merge_value(conf->recursive, prev->recursive, 0);
408
409     if (conf->header.len == 0) {
410         conf->hash = prev->hash;
411         conf->header = prev->header;
412     }
413
414     return NGX_CONF_OK;
415 }
416
417
418
419 static ngx_int_t
420 ngx_http_realip_init(ngx_conf_t *cf)
421 {
422     ngx_http_handler_pt *h;
423     ngx_http_core_main_conf_t *cmcf;
424
425     cmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_core_module);
426
427     h = ngx_array_push(&cmcf->phases[NGX_HTTP_POST_READ_PHASE].handlers);
428     if (h == NULL) {
429         return NGX_ERROR;
430     }
431
432     *h = ngx_http_realip_handler;
433
434     h = ngx_array_push(&cmcf->phases[NGX_HTTP_PREACCESS_PHASE].handlers);
435     if (h == NULL) {
436         return NGX_ERROR;
437     }
438
439     *h = ngx_http_realip_handler;
440
441     return NGX_OK;
442 }

```

[One Level Up](#)

[Top Level](#)



# src/http/modules/nginx\_http\_referer\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_referer\\_commands](#)
- [ngx\\_http\\_referer\\_module](#)
- [ngx\\_http\\_referer\\_module\\_ctx](#)

## Data types defined

- [ngx\\_http\\_referer\\_conf\\_t](#)

## Functions defined

- [ngx\\_http\\_add\\_referer](#)
- [ngx\\_http\\_add\\_regex\\_referer](#)
- [ngx\\_http\\_add\\_regex\\_server\\_name](#)
- [ngx\\_http\\_cmp\\_referer\\_wildcards](#)
- [ngx\\_http\\_referer\\_create\\_conf](#)
- [ngx\\_http\\_referer\\_merge\\_conf](#)
- [ngx\\_http\\_referer\\_variable](#)
- [ngx\\_http\\_valid\\_referers](#)

## Macros defined

- [NGX\\_HTTP\\_REFERER\\_NO\\_URI\\_PART](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 #define NGX_HTTP_REFERER_NO_URI_PART ((void *) 4)
14
15
16 typedef struct {
17     ngx_hash_combined_t    hash;
18
19     #if (NGX_PCRE)
20     ngx_array_t            *regex;
21     ngx_array_t            *server_name_regex;
22     #endif
23
24     ngx_flag_t             no_referer;
```

```

25     ngx_flag_t          blocked_referer;
26     ngx_flag_t          server_names;
27
28     ngx_hash_keys_arrays_t *keys;
29
30     ngx_uint_t          referer_hash_max_size;
31     ngx_uint_t          referer_hash_bucket_size;
32 } ngx_http_referer_conf_t;
33
34
35 static void * ngx_http_referer_create_conf(ngx_conf_t *cf);
36 static char * ngx_http_referer_merge_conf(ngx_conf_t *cf, void *parent,
37     void *child);
38 static char * ngx_http_valid_referers(ngx_conf_t *cf, ngx_command_t *cmd,
39     void *conf);
40 static ngx_int_t ngx_http_add_referer(ngx_conf_t *cf,
41     ngx_hash_keys_arrays_t *keys, ngx_str_t *value, ngx_str_t *uri);
42 static ngx_int_t ngx_http_add_regex_referer(ngx_conf_t *cf,
43     ngx_http_referer_conf_t *rlcf, ngx_str_t *name);
44 #if (NGX_PCRE)
45 static ngx_int_t ngx_http_add_regex_server_name(ngx_conf_t *cf,
46     ngx_http_referer_conf_t *rlcf, ngx_http_regex_t *regex);
47 #endif
48 static int ngx_libc_cdecl ngx_http_cmp_referer_wildcards(const void *one,
49     const void *two);
50
51
52 static ngx_command_t ngx_http_referer_commands[] = {
53
54     { ngx_string("valid_referers"),
55       NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_1MORE,
56       ngx_http_valid_referers,
57       NGX_HTTP_LOC_CONF_OFFSET,
58       0,
59       NULL },
60
61     { ngx_string("referer_hash_max_size"),
62       NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
63       ngx_conf_set_num_slot,
64       NGX_HTTP_LOC_CONF_OFFSET,
65       offsetof(ngx_http_referer_conf_t, referer_hash_max_size),
66       NULL },
67
68     { ngx_string("referer_hash_bucket_size"),
69       NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
70       ngx_conf_set_num_slot,
71       NGX_HTTP_LOC_CONF_OFFSET,
72       offsetof(ngx_http_referer_conf_t, referer_hash_bucket_size),
73       NULL },
74
75     ngx_null_command
76 };
77
78
79 static ngx_http_module_t ngx_http_referer_module_ctx = {
80     NULL, /* preconfiguration */
81     NULL, /* postconfiguration */
82
83     NULL, /* create main configuration */
84     NULL, /* init main configuration */
85
86     NULL, /* create server configuration */
87     NULL, /* merge server configuration */
88
89     ngx_http_referer_create_conf, /* create location configuration */
90     ngx_http_referer_merge_conf /* merge location configuration */
91 };
92
93
94 ngx_module_t ngx_http_referer_module = {
95     NGX_MODULE_V1,
96     &ngx_http_referer_module_ctx, /* module context */
97     ngx_http_referer_commands, /* module directives */
98     NGX_HTTP_MODULE, /* module type */
99     NULL, /* init master */
100    NULL, /* init module */

```

```

101     NULL,                                /* init process */
102     NULL,                                /* init thread */
103     NULL,                                /* exit thread */
104     NULL,                                /* exit process */
105     NULL,                                /* exit master */
106     NGX_MODULE_V1_PADDING
107 };
108
109
110 static ngx_int_t
111 ngx_http_referer_variable(ngx_http_request_t *r, ngx_http_variable_value_t *v,
112     uintptr_t data)
113 {
114     u_char                *p, *ref, *last;
115     size_t                len;
116     ngx_str_t            *uri;
117     ngx_uint_t           i, key;
118     ngx_http_referer_conf_t *rlcf;
119     u_char                buf[256];
120 #if (NGX_PCRE)
121     ngx_int_t            rc;
122     ngx_str_t            referer;
123 #endif
124
125     rlcf = ngx_http_get_module_loc_conf(r, ngx_http_referer_module);
126
127     if (rlcf->hash.hash.buckets == NULL
128         && rlcf->hash.wc_head == NULL
129         && rlcf->hash.wc_tail == NULL
130 #if (NGX_PCRE)
131         && rlcf->regex == NULL
132         && rlcf->server_name_regex == NULL
133 #endif
134     )
135     {
136         goto valid;
137     }
138
139     if (r->headers_in.referer == NULL) {
140         if (rlcf->no_referer) {
141             goto valid;
142         }
143
144         goto invalid;
145     }
146
147     len = r->headers_in.referer->value.len;
148     ref = r->headers_in.referer->value.data;
149
150     if (len >= sizeof("http://i.ru") - 1) {
151         last = ref + len;
152
153         if (ngx_strncasecmp(ref, (u_char *) "http://", 7) == 0) {
154             ref += 7;
155             len -= 7;
156             goto valid_scheme;
157
158         } else if (ngx_strncasecmp(ref, (u_char *) "https://", 8) == 0) {
159             ref += 8;
160             len -= 8;
161             goto valid_scheme;
162         }
163     }
164
165     if (rlcf->blocked_referer) {
166         goto valid;
167     }
168
169     goto invalid;
170
171 valid_scheme:
172
173     i = 0;
174     key = 0;
175
176     for (p = ref; p < last; p++) {

```

```

177     if (*p == '/' || *p == ':') {
178         break;
179     }
180
181     if (i == 256) {
182         goto invalid;
183     }
184
185     buf[i] = ngx_tolower(*p);
186     key = ngx_hash(key, buf[i++]);
187 }
188
189 uri = ngx_hash_find_combined(&rllcf->hash, key, buf, p - ref);
190
191 if (uri) {
192     goto uri;
193 }
194
195 #if (NGX_PCRE)
196
197 if (rllcf->server_name_regex) {
198     referer.len = p - ref;
199     referer.data = buf;
200
201     rc = ngx_regex_exec_array(rllcf->server_name_regex, &referer,
202                               r->connection->log);
203
204     if (rc == NGX_OK) {
205         goto valid;
206     }
207
208     if (rc == NGX_ERROR) {
209         return rc;
210     }
211
212     /* NGX_DECLINED */
213 }
214
215 if (rllcf->regex) {
216     referer.len = len;
217     referer.data = ref;
218
219     rc = ngx_regex_exec_array(rllcf->regex, &referer, r->connection->log);
220
221     if (rc == NGX_OK) {
222         goto valid;
223     }
224
225     if (rc == NGX_ERROR) {
226         return rc;
227     }
228
229     /* NGX_DECLINED */
230 }
231
232 #endif
233
234 invalid:
235
236     *v = ngx_http_variable_true_value;
237
238     return NGX_OK;
239
240 uri:
241
242 for ( /* void */ ; p < last; p++) {
243     if (*p == '/') {
244         break;
245     }
246 }
247
248 len = last - p;
249
250 if (uri == NGX_HTTP_REFERER_NO_URI_PART) {
251     goto valid;
252 }

```

```

253     if (len < uri->len || ngx_strncmp(uri->data, p, uri->len) != 0) {
254         goto invalid;
255     }
256 }
257
258 valid:
259
260     *v = ngx_http_variable_null_value;
261
262     return NGX_OK;
263 }
264
265
266 static void *
267 ngx_http_referer_create_conf(ngx_conf_t *cf)
268 {
269     ngx_http_referer_conf_t *conf;
270
271     conf = ngx_palloc(cf->pool, sizeof(ngx_http_referer_conf_t));
272     if (conf == NULL) {
273         return NULL;
274     }
275
276     /*
277      * set by ngx_palloc():
278      *
279      *     conf->hash = { NULL };
280      *     conf->server_names = 0;
281      *     conf->keys = NULL;
282      */
283
284     #if (NGX_PCRE)
285     conf->regex = NGX_CONF_UNSET_PTR;
286     conf->server_name_regex = NGX_CONF_UNSET_PTR;
287     #endif
288
289     conf->no_referer = NGX_CONF_UNSET;
290     conf->blocked_referer = NGX_CONF_UNSET;
291     conf->referer_hash_max_size = NGX_CONF_UNSET_UINT;
292     conf->referer_hash_bucket_size = NGX_CONF_UNSET_UINT;
293
294     return conf;
295 }
296
297
298 static char *
299 ngx_http_referer_merge_conf(ngx_conf_t *cf, void *parent, void *child)
300 {
301     ngx_http_referer_conf_t *prev = parent;
302     ngx_http_referer_conf_t *conf = child;
303
304     ngx_uint_t                n;
305     ngx_hash_init_t           hash;
306     ngx_http_server_name_t    *sn;
307     ngx_http_core_srv_conf_t  *cscf;
308
309     if (conf->keys == NULL) {
310         conf->hash = prev->hash;
311     }
312
313     #if (NGX_PCRE)
314     ngx_conf_merge_ptr_value(conf->regex, prev->regex, NULL);
315     ngx_conf_merge_ptr_value(conf->server_name_regex,
316                               prev->server_name_regex, NULL);
317     #endif
318
319     ngx_conf_merge_value(conf->no_referer, prev->no_referer, 0);
320     ngx_conf_merge_value(conf->blocked_referer, prev->blocked_referer, 0);
321     ngx_conf_merge_uint_value(conf->referer_hash_max_size,
322                               prev->referer_hash_max_size, 2048);
323     ngx_conf_merge_uint_value(conf->referer_hash_bucket_size,
324                               prev->referer_hash_bucket_size, 64);
325
326     return NGX_CONF_OK;
327 }
328
329 if (conf->server_names == 1) {
330     cscf = ngx_http_conf_get_module_srv_conf(cf, ngx_http_core_module);

```

```

329     sn = cscf->server_names.elts;
330     for (n = 0; n < cscf->server_names.nelts; n++) {
331
332
333     #if (NGX_PCRE)
334         if (sn[n].regex) {
335
336             if (ngx\_http\_add\_regex\_server\_name(cf, conf, sn[n].regex)
337                 != NGX\_OK)
338             {
339                 return NGX\_CONF\_ERROR;
340             }
341
342             continue;
343         }
344     #endif
345
346     if (ngx\_http\_add\_referer(cf, conf->keys, &sn[n].name, NULL)
347         != NGX\_OK)
348     {
349         return NGX\_CONF\_ERROR;
350     }
351 }
352
353
354 if ((conf->no_referer == 1 || conf->blocked_referer == 1)
355     && conf->keys->keys.nelts == 0
356     && conf->keys->dns_wc_head.nelts == 0
357     && conf->keys->dns_wc_tail.nelts == 0)
358 {
359     ngx\_log\_error(NGX\_LOG\_EMERG, cf->log, 0,
360                 "the \"none\" or \"blocked\" referers are specified "
361                 "in the \"valid_referers\" directive "
362                 "without any valid referer");
363     return NGX\_CONF\_ERROR;
364 }
365
366 ngx\_conf\_merge\_uint\_value(conf->referer_hash_max_size,
367                             prev->referer_hash_max_size, 2048);
368 ngx\_conf\_merge\_uint\_value(conf->referer_hash_bucket_size,
369                             prev->referer_hash_bucket_size, 64);
370 conf->referer_hash_bucket_size = ngx\_align(conf->referer_hash_bucket_size,
371                                         ngx\_cacheline\_size);
372
373 hash.key = ngx\_hash\_key\_lc;
374 hash.max_size = conf->referer_hash_max_size;
375 hash.bucket_size = conf->referer_hash_bucket_size;
376 hash.name = "referer_hash";
377 hash.pool = cf->pool;
378
379 if (conf->keys->keys.nelts) {
380     hash.hash = &conf->hash.hash;
381     hash.temp_pool = NULL;
382
383     if (ngx\_hash\_init(&hash, conf->keys->keys.elts, conf->keys->keys.nelts)
384         != NGX\_OK)
385     {
386         return NGX\_CONF\_ERROR;
387     }
388 }
389
390 if (conf->keys->dns_wc_head.nelts) {
391
392     ngx\_qsort(conf->keys->dns_wc_head.elts,
393              (size_t) conf->keys->dns_wc_head.nelts,
394              sizeof(ngx\_hash\_key\_t),
395              ngx\_http\_cmp\_referer\_wildcards);
396
397     hash.hash = NULL;
398     hash.temp_pool = cf->temp_pool;
399
400     if (ngx\_hash\_wildcard\_init(&hash, conf->keys->dns_wc_head.elts,
401                               conf->keys->dns_wc_head.nelts)
402         != NGX\_OK)
403     {
404         return NGX\_CONF\_ERROR;

```

```

405     }
406     conf->hash.wc_head = (ngx\_hash\_wildcard\_t *) hash.hash;
407 }
408
409
410 if (conf->keys->dns_wc_tail.nelts) {
411     ngx\_qsort(conf->keys->dns_wc_tail.elts,
412             (size_t) conf->keys->dns_wc_tail.nelts,
413             sizeof(ngx\_hash\_key\_t),
414             ngx\_http\_cmp\_referer\_wildcards);
415
416     hash.hash = NULL;
417     hash.temp_pool = cf->temp_pool;
418
419     if (ngx\_hash\_wildcard\_init(&hash, conf->keys->dns_wc_tail.elts,
420                             conf->keys->dns_wc_tail.nelts)
421         != NGX\_OK)
422     {
423         return NGX\_CONF\_ERROR;
424     }
425
426     conf->hash.wc_tail = (ngx\_hash\_wildcard\_t *) hash.hash;
427 }
428
429
430 #if (NGX\_PCRE)
431     ngx\_conf\_merge\_ptr\_value(conf->regex, prev->regex, NULL);
432     ngx\_conf\_merge\_ptr\_value(conf->server_name_regex, prev->server_name_regex,
433                             NULL);
434 #endif
435
436 if (conf->no_referer == NGX\_CONF\_UNSET) {
437     conf->no_referer = 0;
438 }
439
440 if (conf->blocked_referer == NGX\_CONF\_UNSET) {
441     conf->blocked_referer = 0;
442 }
443
444 conf->keys = NULL;
445
446 return NGX\_CONF\_OK;
447 }
448
449
450 static char *
451 ngx\_http\_valid\_referers(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
452 {
453     ngx\_http\_referer\_conf\_t *rlcf = conf;
454
455     u_char          *p;
456     ngx\_str\_t      *value, uri, name;
457     ngx\_uint\_t     i;
458     ngx\_http\_variable\_t *var;
459
460     ngx\_str\_set(&name, "invalid_referer");
461
462     var = ngx\_http\_add\_variable(cf, &name, NGX\_HTTP\_VAR\_CHANGEABLE);
463     if (var == NULL) {
464         return NGX\_CONF\_ERROR;
465     }
466
467     var->get_handler = ngx\_http\_referer\_variable;
468
469     if (rlcf->keys == NULL) {
470         rlcf->keys = ngx\_palloc(cf->temp_pool, sizeof(ngx\_hash\_keys\_arrays\_t));
471         if (rlcf->keys == NULL) {
472             return NGX\_CONF\_ERROR;
473         }
474
475         rlcf->keys->pool = cf->pool;
476         rlcf->keys->temp_pool = cf->temp_pool;
477
478         if (ngx\_hash\_keys\_array\_init(rlcf->keys, NGX\_HASH\_SMALL) != NGX\_OK) {
479             return NGX\_CONF\_ERROR;
480         }
481     }

```

```

481 }
482
483 value = cf->args->elts;
484
485 for (i = 1; i < cf->args->nelts; i++) {
486     if (value[i].len == 0) {
487         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
488             "invalid referer \"%V\"", &value[i]);
489         return NGX_CONF_ERROR;
490     }
491
492     if (ngx_strcmp(value[i].data, "none") == 0) {
493         rlcfc->no_referer = 1;
494         continue;
495     }
496
497     if (ngx_strcmp(value[i].data, "blocked") == 0) {
498         rlcfc->blocked_referer = 1;
499         continue;
500     }
501
502     if (ngx_strcmp(value[i].data, "server_names") == 0) {
503         rlcfc->server_names = 1;
504         continue;
505     }
506
507     if (value[i].data[0] == '~') {
508         if (ngx_http_add_regex_referer(cf, rlcfc, &value[i]) != NGX_OK) {
509             return NGX_CONF_ERROR;
510         }
511
512         continue;
513     }
514
515     ngx_str_null(&uri);
516
517     p = (u_char *) ngx_strchr(value[i].data, '/');
518
519     if (p) {
520         uri.len = (value[i].data + value[i].len) - p;
521         uri.data = p;
522         value[i].len = p - value[i].data;
523     }
524
525     if (ngx_http_add_referer(cf, rlcfc->keys, &value[i], &uri) != NGX_OK) {
526         return NGX_CONF_ERROR;
527     }
528 }
529
530 return NGX_CONF_OK;
531 }
532
533
534 static ngx_int_t
535 ngx_http_add_referer(ngx_conf_t *cf, ngx_hash_keys_arrays_t *keys,
536     ngx_str_t *value, ngx_str_t *uri)
537 {
538     ngx_int_t rc;
539     ngx_str_t *u;
540
541     if (uri == NULL || uri->len == 0) {
542         u = NGX_HTTP_REFERER_NO_URI_PART;
543     } else {
544         u = ngx_palloc(cf->pool, sizeof(ngx_str_t));
545         if (u == NULL) {
546             return NGX_ERROR;
547         }
548
549         *u = *uri;
550     }
551
552     rc = ngx_hash_add_key(keys, value, u, NGX_HASH_WILDCARD_KEY);
553
554     if (rc == NGX_OK) {
555         return NGX_OK;
556     }

```



```

557     }
558
559     if (rc == NGX_DECLINED) {
560         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
561             "invalid hostname or wildcard \"%V\"", value);
562     }
563
564     if (rc == NGX_BUSY) {
565         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
566             "conflicting parameter \"%V\"", value);
567     }
568
569     return NGX_ERROR;
570 }
571
572
573 static ngx_int_t
574 ngx_http_add_regex_referer(ngx_conf_t *cf, ngx_http_referer_conf_t *rlcf,
575     ngx_str_t *name)
576 {
577     #if (NGX_PCRE)
578         ngx_regex_elt_t    *re;
579         ngx_regex_compile_t  rc;
580         u_char              errstr[NGX_MAX_CONF_ERRSTR];
581
582         if (name->len == 1) {
583             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0, "empty regex in \"%V\"", name);
584             return NGX_ERROR;
585         }
586
587         if (rlcf->regex == NGX_CONF_UNSET_PTR) {
588             rlcf->regex = ngx_array_create(cf->pool, 2, sizeof(ngx_regex_elt_t));
589             if (rlcf->regex == NULL) {
590                 return NGX_ERROR;
591             }
592         }
593
594         re = ngx_array_push(rlcf->regex);
595         if (re == NULL) {
596             return NGX_ERROR;
597         }
598
599         name->len--;
600         name->data++;
601
602         ngx_memzero(&rc, sizeof(ngx_regex_compile_t));
603
604         rc.pattern = *name;
605         rc.pool = cf->pool;
606         rc.options = NGX_REGEX_CASELESS;
607         rc.err.len = NGX_MAX_CONF_ERRSTR;
608         rc.err.data = errstr;
609
610         if (ngx_regex_compile(&rc) != NGX_OK) {
611             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0, "%V", &rc.err);
612             return NGX_ERROR;
613         }
614
615         re->regex = rc.regex;
616         re->name = name->data;
617
618         return NGX_OK;
619     #else
620     #endif
621
622     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
623         "the using of the regex \"%V\" requires PCRE library",
624         name);
625
626     return NGX_ERROR;
627
628 #endif
629 }
630
631
632 #if (NGX_PCRE)

```

```

633 static ngx_int_t
634 ngx_http_add_regex_server_name(ngx_conf_t *cf, ngx_http_referer_conf_t *rlcf,
635 ngx_http_regex_t *regex)
636 {
637     ngx_regex_elt_t *re;
638
639     if (rlcf->server_name_regex == NGX_CONF_UNSET_PTR) {
640         rlcf->server_name_regex = ngx_array_create(cf->pool, 2,
641             sizeof(ngx_regex_elt_t));
642         if (rlcf->server_name_regex == NULL) {
643             return NGX_ERROR;
644         }
645     }
646
647     re = ngx_array_push(rlcf->server_name_regex);
648     if (re == NULL) {
649         return NGX_ERROR;
650     }
651
652     re->regex = regex->regex;
653     re->name = regex->name.data;
654
655     return NGX_OK;
656 }
657
658 #endif
659
660
661 static int ngx_libc_cdecl
662 ngx_http_cmp_referer_wildcards(const void *one, const void *two)
663 {
664     ngx_hash_key_t *first, *second;
665
666     first = (ngx_hash_key_t *) one;
667     second = (ngx_hash_key_t *) two;
668
669     return ngx_dns_strcmp(first->key.data, second->key.data);
670 }
671

```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_rewrite\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_rewrite\\_commands](#)
- [ngx\\_http\\_rewrite\\_module](#)
- [ngx\\_http\\_rewrite\\_module\\_ctx](#)

## Data types defined

- [ngx\\_http\\_rewrite\\_loc\\_conf\\_t](#)

## Functions defined

- [ngx\\_http\\_rewrite](#)
- [ngx\\_http\\_rewrite\\_break](#)
- [ngx\\_http\\_rewrite\\_create\\_loc\\_conf](#)
- [ngx\\_http\\_rewrite\\_handler](#)
- [ngx\\_http\\_rewrite\\_if](#)
- [ngx\\_http\\_rewrite\\_if\\_condition](#)
- [ngx\\_http\\_rewrite\\_init](#)
- [ngx\\_http\\_rewrite\\_merge\\_loc\\_conf](#)
- [ngx\\_http\\_rewrite\\_return](#)
- [ngx\\_http\\_rewrite\\_set](#)
- [ngx\\_http\\_rewrite\\_value](#)
- [ngx\\_http\\_rewrite\\_var](#)
- [ngx\\_http\\_rewrite\\_variable](#)

## Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 typedef struct {
14     ngx\_array\_t *codes;          /* uintptr\_t */
15
16     ngx\_uint\_t    stack_size;
17
18     ngx\_flag\_t    log;
19     ngx\_flag\_t    uninitialized_variable_warn;
```

```

20 } ngx_http_rewrite_loc_conf_t;
21
22
23 static void *ngx_http_rewrite_create_loc_conf(ngx_conf_t *cf);
24 static char *ngx_http_rewrite_merge_loc_conf(ngx_conf_t *cf,
25     void *parent, void *child);
26 static ngx_int_t ngx_http_rewrite_init(ngx_conf_t *cf);
27 static char *ngx_http_rewrite(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
28 static char *ngx_http_rewrite_return(ngx_conf_t *cf, ngx_command_t *cmd,
29     void *conf);
30 static char *ngx_http_rewrite_break(ngx_conf_t *cf, ngx_command_t *cmd,
31     void *conf);
32 static char *ngx_http_rewrite_if(ngx_conf_t *cf, ngx_command_t *cmd,
33     void *conf);
34 static char * ngx_http_rewrite_if_condition(ngx_conf_t *cf,
35     ngx_http_rewrite_loc_conf_t *lcf);
36 static char *ngx_http_rewrite_variable(ngx_conf_t *cf,
37     ngx_http_rewrite_loc_conf_t *lcf, ngx_str_t *value);
38 static char *ngx_http_rewrite_set(ngx_conf_t *cf, ngx_command_t *cmd,
39     void *conf);
40 static char * ngx_http_rewrite_value(ngx_conf_t *cf,
41     ngx_http_rewrite_loc_conf_t *lcf, ngx_str_t *value);
42
43
44 static ngx_command_t  ngx_http_rewrite_commands[] = {
45
46     { ngx_string("rewrite"),
47       NGX_HTTP_SRV_CONF|NGX_HTTP_SIF_CONF|NGX_HTTP_LOC_CONF|NGX_HTTP_LIF_CONF
48         |NGX_CONF_TAKE23,
49       ngx_http_rewrite,
50       NGX_HTTP_LOC_CONF_OFFSET,
51       0,
52       NULL },
53
54     { ngx_string("return"),
55       NGX_HTTP_SRV_CONF|NGX_HTTP_SIF_CONF|NGX_HTTP_LOC_CONF|NGX_HTTP_LIF_CONF
56         |NGX_CONF_TAKE12,
57       ngx_http_rewrite_return,
58       NGX_HTTP_LOC_CONF_OFFSET,
59       0,
60       NULL },
61
62     { ngx_string("break"),
63       NGX_HTTP_SRV_CONF|NGX_HTTP_SIF_CONF|NGX_HTTP_LOC_CONF|NGX_HTTP_LIF_CONF
64         |NGX_CONF_NOARGS,
65       ngx_http_rewrite_break,
66       NGX_HTTP_LOC_CONF_OFFSET,
67       0,
68       NULL },
69
70     { ngx_string("if"),
71       NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_BLOCK|NGX_CONF_1MORE,
72       ngx_http_rewrite_if,
73       NGX_HTTP_LOC_CONF_OFFSET,
74       0,
75       NULL },
76
77     { ngx_string("set"),
78       NGX_HTTP_SRV_CONF|NGX_HTTP_SIF_CONF|NGX_HTTP_LOC_CONF|NGX_HTTP_LIF_CONF
79         |NGX_CONF_TAKE2,
80       ngx_http_rewrite_set,
81       NGX_HTTP_LOC_CONF_OFFSET,
82       0,
83       NULL },
84
85     { ngx_string("rewrite_log"),
86       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_SIF_CONF|NGX_HTTP_LOC_CONF
87         |NGX_HTTP_LIF_CONF|NGX_CONF_FLAG,
88       ngx_conf_set_flag_slot,
89       NGX_HTTP_LOC_CONF_OFFSET,
90       offsetof(ngx_http_rewrite_loc_conf_t, log),
91       NULL },
92
93     { ngx_string("uninitialized_variable_warn"),
94       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_SIF_CONF|NGX_HTTP_LOC_CONF
95         |NGX_HTTP_LIF_CONF|NGX_CONF_FLAG,

```

```

96     ngx_conf_set_flag_slot,
97     NGX_HTTP_LOC_CONF_OFFSET,
98     offsetof(ngx_http_rewrite_loc_conf_t, uninitialized_variable_warn),
99     NULL },
100
101     ngx_null_command
102 };
103
104
105 static ngx_http_module_t ngx_http_rewrite_module_ctx = {
106     NULL,                                     /* preconfiguration */
107     ngx_http_rewrite_init,                   /* postconfiguration */
108
109     NULL,                                     /* create main configuration */
110     NULL,                                     /* init main configuration */
111
112     NULL,                                     /* create server configuration */
113     NULL,                                     /* merge server configuration */
114
115     ngx_http_rewrite_create_loc_conf,        /* create location configuration */
116     ngx_http_rewrite_merge_loc_conf         /* merge location configuration */
117 };
118
119
120 ngx_module_t ngx_http_rewrite_module = {
121     NGX_MODULE_V1,
122     &ngx_http_rewrite_module_ctx,           /* module context */
123     ngx_http_rewrite_commands,             /* module directives */
124     NGX_HTTP_MODULE,                       /* module type */
125     NULL,                                   /* init master */
126     NULL,                                   /* init module */
127     NULL,                                   /* init process */
128     NULL,                                   /* init thread */
129     NULL,                                   /* exit thread */
130     NULL,                                   /* exit process */
131     NULL,                                   /* exit master */
132     NGX_MODULE_V1_PADDING
133 };
134
135
136 static ngx_int_t
137 ngx_http_rewrite_handler(ngx_http_request_t *r)
138 {
139     ngx_int_t          index;
140     ngx_http_script_code_pt  code;
141     ngx_http_script_engine_t *e;
142     ngx_http_core_srv_conf_t *cscf;
143     ngx_http_core_main_conf_t *cmcf;
144     ngx_http_rewrite_loc_conf_t *rlcf;
145
146     cmcf = ngx_http_get_module_main_conf(r, ngx_http_core_module);
147     cscf = ngx_http_get_module_srv_conf(r, ngx_http_core_module);
148     index = cmcf->phase_engine.location_rewrite_index;
149
150     if (r->phase_handler == index && r->loc_conf == cscf->ctx->loc_conf) {
151         /* skipping location rewrite phase for server null location */
152         return NGX_DECLINED;
153     }
154
155     rlcf = ngx_http_get_module_loc_conf(r, ngx_http_rewrite_module);
156
157     if (rlcf->codes == NULL) {
158         return NGX_DECLINED;
159     }
160
161     e = ngx_palloc(r->pool, sizeof(ngx_http_script_engine_t));
162     if (e == NULL) {
163         return NGX_HTTP_INTERNAL_SERVER_ERROR;
164     }
165
166     e->sp = ngx_palloc(r->pool,
167         rlcf->stack_size * sizeof(ngx_http_variable_value_t));
168     if (e->sp == NULL) {
169         return NGX_HTTP_INTERNAL_SERVER_ERROR;
170     }
171

```

```

172 e->ip = rlcfc->codes->elts;
173 e->request = r;
174 e->quote = 1;
175 e->log = rlcfc->log;
176 e->status = NGX_DECLINED;
177
178 while (*(uintptr_t *) e->ip) {
179     code = *(ngx_http_script_code_pt *) e->ip;
180     code(e);
181 }
182
183 if (e->status < NGX_HTTP_BAD_REQUEST) {
184     return e->status;
185 }
186
187 if (r->err_status == 0) {
188     return e->status;
189 }
190
191 return r->err_status;
192 }
193
194
195 static ngx_int_t
196 ngx_http_rewrite_var(ngx_http_request_t *r, ngx_http_variable_value_t *v,
197     uintptr_t data)
198 {
199     ngx_http_variable_t *var;
200     ngx_http_core_main_conf_t *cmcf;
201     ngx_http_rewrite_loc_conf_t *rlcf;
202
203     rlcf = ngx_http_get_module_loc_conf(r, ngx_http_rewrite_module);
204
205     if (rlcf->uninitialized_variable_warn == 0) {
206         *v = ngx_http_variable_null_value;
207         return NGX_OK;
208     }
209
210     cmcf = ngx_http_get_module_main_conf(r, ngx_http_core_module);
211
212     var = cmcf->variables.elts;
213
214     /*
215      * the ngx_http_rewrite_module sets variables directly in r->variables,
216      * and they should be handled by ngx_http_get_indexed_variable(),
217      * so the handler is called only if the variable is not initialized
218      */
219
220     ngx_log_error(NGX_LOG_WARN, r->connection->log, 0,
221         "using uninitialized \"%V\" variable", &var[data].name);
222
223     *v = ngx_http_variable_null_value;
224
225     return NGX_OK;
226 }
227
228
229 static void *
230 ngx_http_rewrite_create_loc_conf(ngx_conf_t *cf)
231 {
232     ngx_http_rewrite_loc_conf_t *conf;
233
234     conf = ngx_palloc(cf->pool, sizeof(ngx_http_rewrite_loc_conf_t));
235     if (conf == NULL) {
236         return NULL;
237     }
238
239     conf->stack_size = NGX_CONF_UNSET_UINT;
240     conf->log = NGX_CONF_UNSET;
241     conf->uninitialized_variable_warn = NGX_CONF_UNSET;
242
243     return conf;
244 }
245
246
247 static char *

```

```

248 ngx_http_rewrite_merge_loc_conf(ngx_conf_t *cf, void *parent, void *child)
249 {
250     ngx_http_rewrite_loc_conf_t *prev = parent;
251     ngx_http_rewrite_loc_conf_t *conf = child;
252
253     uintptr_t *code;
254
255     ngx_conf_merge_value(conf->log, prev->log, 0);
256     ngx_conf_merge_value(conf->uninitialized_variable_warn,
257         prev->uninitialized_variable_warn, 1);
258     ngx_conf_merge_uint_value(conf->stack_size, prev->stack_size, 10);
259
260     if (conf->codes == NULL) {
261         return NGX_CONF_OK;
262     }
263
264     if (conf->codes == prev->codes) {
265         return NGX_CONF_OK;
266     }
267
268     code = ngx_array_push_n(conf->codes, sizeof(uintptr_t));
269     if (code == NULL) {
270         return NGX_CONF_ERROR;
271     }
272
273     *code = (uintptr_t) NULL;
274
275     return NGX_CONF_OK;
276 }
277
278
279 static ngx_int_t
280 ngx_http_rewrite_init(ngx_conf_t *cf)
281 {
282     ngx_http_handler_pt *h;
283     ngx_http_core_main_conf_t *cmcf;
284
285     cmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_core_module);
286
287     h = ngx_array_push(&cmcf->phases[NGX_HTTP_SERVER_REWRITE_PHASE].handlers);
288     if (h == NULL) {
289         return NGX_ERROR;
290     }
291
292     *h = ngx_http_rewrite_handler;
293
294     h = ngx_array_push(&cmcf->phases[NGX_HTTP_REWRITE_PHASE].handlers);
295     if (h == NULL) {
296         return NGX_ERROR;
297     }
298
299     *h = ngx_http_rewrite_handler;
300
301     return NGX_OK;
302 }
303
304
305 static char *
306 ngx_http_rewrite(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
307 {
308     ngx_http_rewrite_loc_conf_t *lcf = conf;
309
310     ngx_str_t *value;
311     ngx_uint_t last;
312     ngx_regex_compile_t rc;
313     ngx_http_script_code_pt *code;
314     ngx_http_script_compile_t sc;
315     ngx_http_script_regex_code_t *regex;
316     ngx_http_script_regex_end_code_t *regex_end;
317     u_char errstr[NGX_MAX_CONF_ERRSTR];
318
319     regex = ngx_http_script_start_code(cf->pool, &lcf->codes,
320         sizeof(ngx_http_script_regex_code_t));
321     if (regex == NULL) {
322         return NGX_CONF_ERROR;
323     }

```

```

324 ngx\_memzero(regex, sizeof(ngx\_http\_script\_regex\_code\_t));
325
326
327 value = cf->args->elts;
328
329 ngx\_memzero(&rc, sizeof(ngx\_regex\_compile\_t));
330
331 rc.pattern = value[1];
332 rc.err.len = NGX\_MAX\_CONF\_ERRSTR;
333 rc.err.data = errstr;
334
335 /* TODO: NGX\_REGEX\_CASELESS */
336
337 regex->regex = ngx\_http\_regex\_compile(cf, &rc);
338 if (regex->regex == NULL) {
339     return NGX\_CONF\_ERROR;
340 }
341
342 regex->code = ngx\_http\_script\_regex\_start\_code;
343 regex->uri = 1;
344 regex->name = value[1];
345
346 if (value[2].data[value[2].len - 1] == '?') {
347     /* the last "?" drops the original arguments */
348     value[2].len--;
349
350 } else {
351     regex->add_args = 1;
352 }
353
354 last = 0;
355
356 if (ngx\_strncmp(value[2].data, "http://", sizeof("http://") - 1) == 0
357     || ngx\_strncmp(value[2].data, "https://", sizeof("https://") - 1) == 0
358     || ngx\_strncmp(value[2].data, "$scheme", sizeof("$scheme") - 1) == 0)
359 {
360     regex->status = NGX\_HTTP\_MOVED\_TEMPORARILY;
361     regex->redirect = 1;
362     last = 1;
363 }
364
365 if (cf->args->nelts == 4) {
366     if (ngx\_strncmp(value[3].data, "last") == 0) {
367         last = 1;
368
369     } else if (ngx\_strncmp(value[3].data, "break") == 0) {
370         regex->break_cycle = 1;
371         last = 1;
372
373     } else if (ngx\_strncmp(value[3].data, "redirect") == 0) {
374         regex->status = NGX\_HTTP\_MOVED\_TEMPORARILY;
375         regex->redirect = 1;
376         last = 1;
377
378     } else if (ngx\_strncmp(value[3].data, "permanent") == 0) {
379         regex->status = NGX\_HTTP\_MOVED\_PERMANENTLY;
380         regex->redirect = 1;
381         last = 1;
382
383     } else {
384         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
385             "invalid parameter \"%V\\\"", &value[3]);
386         return NGX\_CONF\_ERROR;
387     }
388 }
389
390 ngx\_memzero(&sc, sizeof(ngx\_http\_script\_compile\_t));
391
392 sc.cf = cf;
393 sc.source = &value[2];
394 sc.lengths = &regex->lengths;
395 sc.values = &lcf->codes;
396 sc.variables = ngx\_http\_script\_variables\_count(&value[2]);
397 sc.main = regex;
398 sc.complete_lengths = 1;
399

```



```

400     sc.compile_args = !regex->redirect;
401
402     if (ngx_http_script_compile(&sc) != NGX_OK) {
403         return NGX_CONF_ERROR;
404     }
405
406     regex = sc.main;
407
408     regex->size = sc.size;
409     regex->args = sc.args;
410
411     if (sc.variables == 0 && !sc.dup_capture) {
412         regex->lengths = NULL;
413     }
414
415     regex_end = ngx_http_script_add_code(lcf->codes,
416                                         sizeof(ngx_http_script_regex_end_code_t),
417                                         &regex);
418
419     if (regex_end == NULL) {
420         return NGX_CONF_ERROR;
421     }
422
423     regex_end->code = ngx_http_script_regex_end_code;
424     regex_end->uri = regex->uri;
425     regex_end->args = regex->args;
426     regex_end->add_args = regex->add_args;
427     regex_end->redirect = regex->redirect;
428
429     if (last) {
430         code = ngx_http_script_add_code(lcf->codes, sizeof(uintptr_t), &regex);
431         if (code == NULL) {
432             return NGX_CONF_ERROR;
433         }
434
435         *code = NULL;
436     }
437
438     regex->next = (u_char *) lcf->codes->elts + lcf->codes->nelts
439                 - (u_char *) regex;
440
441     return NGX_CONF_OK;
442 }
443
444 static char *
445 ngx_http_rewrite_return(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
446 {
447     ngx_http_rewrite_loc_conf_t *lcf = conf;
448
449     u_char *p;
450     ngx_str_t value, *v;
451     ngx_http_script_return_code_t *ret;
452     ngx_http_compile_complex_value_t ccv;
453
454     ret = ngx_http_script_start_code(cf->pool, &lcf->codes,
455                                     sizeof(ngx_http_script_return_code_t));
456     if (ret == NULL) {
457         return NGX_CONF_ERROR;
458     }
459
460     value = cf->args->elts;
461
462     ngx_memzero(ret, sizeof(ngx_http_script_return_code_t));
463
464     ret->code = ngx_http_script_return_code;
465
466     p = value[1].data;
467
468     ret->status = ngx_atoi(p, value[1].len);
469
470     if (ret->status == (uintptr_t) NGX_ERROR) {
471
472         if (cf->args->nelts == 2
473             && (ngx_strncmp(p, "http://", sizeof("http://") - 1) == 0
474                 || ngx_strncmp(p, "https://", sizeof("https://") - 1) == 0
475                 || ngx_strncmp(p, "$scheme", sizeof("$scheme") - 1) == 0))

```

```

476     {
477         ret->status = NGX\_HTTP\_MOVED\_TEMPORARILY;
478         v = &value[1];
479
480     } else {
481         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
482             "invalid return code \"%V\"", &value[1]);
483         return NGX\_CONF\_ERROR;
484     }
485
486 } else {
487
488     if (ret->status > 999) {
489         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
490             "invalid return code \"%V\"", &value[1]);
491         return NGX\_CONF\_ERROR;
492     }
493
494     if (cf->args->nelts == 2) {
495         return NGX\_CONF\_OK;
496     }
497
498     v = &value[2];
499 }
500
501 ngx\_memzero(&ccv, sizeof(ngx\_http\_compile\_complex\_value\_t));
502
503 ccv.cf = cf;
504 ccv.value = v;
505 ccv.complex_value = &ret->text;
506
507 if (ngx\_http\_compile\_complex\_value(&ccv) != NGX\_OK) {
508     return NGX\_CONF\_ERROR;
509 }
510
511 return NGX\_CONF\_OK;
512 }
513
514
515 static char *
516 ngx\_http\_rewrite\_break(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
517 {
518     ngx\_http\_rewrite\_loc\_conf\_t *lcf = conf;
519
520     ngx\_http\_script\_code\_pt *code;
521
522     code = ngx\_http\_script\_start\_code(cf->pool, &lcf->codes, sizeof(uintptr\_t));
523     if (code == NULL) {
524         return NGX\_CONF\_ERROR;
525     }
526
527     *code = ngx\_http\_script\_break\_code;
528
529     return NGX\_CONF\_OK;
530 }
531
532
533 static char *
534 ngx\_http\_rewrite\_if(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
535 {
536     ngx\_http\_rewrite\_loc\_conf\_t *lcf = conf;
537
538     void *mconf;
539     char *rv;
540     u_char *elts;
541     ngx\_uint\_t i;
542     ngx\_conf\_t save;
543     ngx\_http\_module\_t *module;
544     ngx\_http\_conf\_ctx\_t *ctx, *pctx;
545     ngx\_http\_core\_loc\_conf\_t *clcf, *pclcf;
546     ngx\_http\_script\_if\_code\_t *if_code;
547     ngx\_http\_rewrite\_loc\_conf\_t *nlcf;
548
549     ctx = ngx\_palloc(cf->pool, sizeof(ngx\_http\_conf\_ctx\_t));
550     if (ctx == NULL) {
551         return NGX\_CONF\_ERROR;

```

```

552 }
553
554 pctx = cf->ctx;
555 ctx->main_conf = pctx->main_conf;
556 ctx->srv_conf = pctx->srv_conf;
557
558 ctx->loc_conf = ngx_palloc(cf->pool, sizeof(void *) * ngx_http_max_module);
559 if (ctx->loc_conf == NULL) {
560     return NGX_CONF_ERROR;
561 }
562
563 for (i = 0; ngx_modules[i]; i++) {
564     if (ngx_modules[i]->type != NGX_HTTP_MODULE) {
565         continue;
566     }
567
568     module = ngx_modules[i]->ctx;
569
570     if (module->create_loc_conf) {
571
572         mconf = module->create_loc_conf(cf);
573         if (mconf == NULL) {
574             return NGX_CONF_ERROR;
575         }
576
577         ctx->loc_conf[ngx_modules[i]->ctx_index] = mconf;
578     }
579 }
580
581 pclcf = pctx->loc_conf[ngx_http_core_module.ctx_index];
582
583 clcf = ctx->loc_conf[ngx_http_core_module.ctx_index];
584 clcf->loc_conf = ctx->loc_conf;
585 clcf->name = pclcf->name;
586 clcf->noname = 1;
587
588 if (ngx_http_add_location(cf, &pclcf->locations, clcf) != NGX_OK) {
589     return NGX_CONF_ERROR;
590 }
591
592 if (ngx_http_rewrite_if_condition(cf, lcf) != NGX_CONF_OK) {
593     return NGX_CONF_ERROR;
594 }
595
596 if_code = ngx_array_push_n(lcf->codes, sizeof(ngx_http_script_if_code_t));
597 if (if_code == NULL) {
598     return NGX_CONF_ERROR;
599 }
600
601 if_code->code = ngx_http_script_if_code;
602
603 elts = lcf->codes->elts;
604
605
606 /* the inner directives must be compiled to the same code array */
607
608 nlcf = ctx->loc_conf[ngx_http_rewrite_module.ctx_index];
609 nlcf->codes = lcf->codes;
610
611
612 save = *cf;
613 cf->ctx = ctx;
614
615 if (pclcf->name.len == 0) {
616     if_code->loc_conf = NULL;
617     cf->cmd_type = NGX_HTTP_SIF_CONF;
618 } else {
619     if_code->loc_conf = ctx->loc_conf;
620     cf->cmd_type = NGX_HTTP_LIF_CONF;
621 }
622
623
624 rv = ngx_conf_parse(cf, NULL);
625
626 *cf = save;
627

```

```

628     if (rv != NGX_CONF_OK) {
629         return rv;
630     }
631
632
633     if (elts != lcf->codes->elts) {
634         if_code = (ngx_http_script_if_code_t *)
635             ((u_char *) if_code + ((u_char *) lcf->codes->elts - elts));
636     }
637
638     if_code->next = (u_char *) lcf->codes->elts + lcf->codes->nelts
639                 - (u_char *) if_code;
640
641     /* the code array belong to parent block */
642
643     nlcf->codes = NULL;
644
645     return NGX_CONF_OK;
646 }
647
648
649 static char *
650 ngx_http_rewrite_if_condition(ngx_conf_t *cf, ngx_http_rewrite_loc_conf_t *lcf)
651 {
652     u_char          *p;
653     size_t          len;
654     ngx_str_t       *value;
655     ngx_uint_t      cur, last;
656     ngx_regex_compile_t rc;
657     ngx_http_script_code_pt *code;
658     ngx_http_script_file_code_t *fop;
659     ngx_http_script_regex_code_t *regex;
660     u_char          errstr[NGX_MAX_CONF_ERRSTR];
661
662     value = cf->args->elts;
663     last = cf->args->nelts - 1;
664
665     if (value[1].len < 1 || value[1].data[0] != '(') {
666         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
667             "invalid condition \"%V\"", &value[1]);
668         return NGX_CONF_ERROR;
669     }
670
671     if (value[1].len == 1) {
672         cur = 2;
673     }
674     else {
675         cur = 1;
676         value[1].len--;
677         value[1].data++;
678     }
679
680     if (value[last].len < 1 || value[last].data[value[last].len - 1] != ')') {
681         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
682             "invalid condition \"%V\"", &value[last]);
683         return NGX_CONF_ERROR;
684     }
685
686     if (value[last].len == 1) {
687         last--;
688     }
689     else {
690         value[last].len--;
691         value[last].data[value[last].len] = '\\0';
692     }
693
694     len = value[cur].len;
695     p = value[cur].data;
696
697     if (len > 1 && p[0] == '$') {
698
699         if (cur != last && cur + 2 != last) {
700             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
701                 "invalid condition \"%V\"", &value[cur]);
702             return NGX_CONF_ERROR;
703         }

```

```

704
705 if (ngx\_http\_rewrite\_variable(cf, lcf, &value[cur]) != NGX\_CONF\_OK) {
706     return NGX\_CONF\_ERROR;
707 }
708
709 if (cur == last) {
710     return NGX\_CONF\_OK;
711 }
712
713 cur++;
714
715 len = value[cur].len;
716 p = value[cur].data;
717
718 if (len == 1 && p[0] == '=') {
719
720     if (ngx\_http\_rewrite\_value(cf, lcf, &value[last]) != NGX\_CONF\_OK) {
721         return NGX\_CONF\_ERROR;
722     }
723
724     code = ngx\_http\_script\_start\_code(cf->pool, &lcf->codes,
725                                         sizeof(uintptr\_t));
726
727     if (code == NULL) {
728         return NGX\_CONF\_ERROR;
729     }
730
731     *code = ngx\_http\_script\_equal\_code;
732
733     return NGX\_CONF\_OK;
734 }
735
736 if (len == 2 && p[0] == '!' && p[1] == '=') {
737
738     if (ngx\_http\_rewrite\_value(cf, lcf, &value[last]) != NGX\_CONF\_OK) {
739         return NGX\_CONF\_ERROR;
740     }
741
742     code = ngx\_http\_script\_start\_code(cf->pool, &lcf->codes,
743                                         sizeof(uintptr\_t));
744
745     if (code == NULL) {
746         return NGX\_CONF\_ERROR;
747     }
748
749     *code = ngx\_http\_script\_not\_equal\_code;
750     return NGX\_CONF\_OK;
751 }
752
753 if ((len == 1 && p[0] == '~')
754     || (len == 2 && p[0] == '~' && p[1] == '*')
755     || (len == 2 && p[0] == '!' && p[1] == '~')
756     || (len == 3 && p[0] == '!' && p[1] == '~' && p[2] == '*'))
757 {
758     regex = ngx\_http\_script\_start\_code(cf->pool, &lcf->codes,
759                                         sizeof(ngx\_http\_script\_regex\_code\_t));
760
761     if (regex == NULL) {
762         return NGX\_CONF\_ERROR;
763     }
764
765     ngx\_memzero(regex, sizeof(ngx\_http\_script\_regex\_code\_t));
766
767     ngx\_memzero(&rc, sizeof(ngx\_regex\_compile\_t));
768
769     rc.pattern = value[last];
770     rc.options = (p[len - 1] == '*') ? NGX\_REGEX\_CASELESS : 0;
771     rc.err.len = NGX\_MAX\_CONF\_ERRSTR;
772     rc.err.data = errstr;
773
774     regex->regex = ngx\_http\_regex\_compile(cf, &rc);
775
776     if (regex->regex == NULL) {
777         return NGX\_CONF\_ERROR;
778     }
779
780     regex->code = ngx\_http\_script\_regex\_start\_code;
781     regex->next = sizeof(ngx\_http\_script\_regex\_code\_t);
782     regex->test = 1;
783     if (p[0] == '!') {

```

```

780         regex->negative_test = 1;
781     }
782     regex->name = value[last];
783
784     return NGX\_CONF\_OK;
785 }
786
787 ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
788     "unexpected \"%V\" in condition", &value[cur]);
789 return NGX\_CONF\_ERROR;
790
791 } else if ((len == 2 && p[0] == '-')
792     || (len == 3 && p[0] == '!' && p[1] == '-'))
793 {
794     if (cur + 1 != last) {
795         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
796             "invalid condition \"%V\"", &value[cur]);
797         return NGX\_CONF\_ERROR;
798     }
799
800     value[last].data[value[last].len] = '\\0';
801     value[last].len++;
802
803     if (ngx\_http\_rewrite\_value(cf, lcf, &value[last]) != NGX\_CONF\_OK) {
804         return NGX\_CONF\_ERROR;
805     }
806
807     fop = ngx\_http\_script\_start\_code(cf->pool, &lcf->codes,
808         sizeof(ngx\_http\_script\_file\_code\_t));
809     if (fop == NULL) {
810         return NGX\_CONF\_ERROR;
811     }
812
813     fop->code = ngx\_http\_script\_file\_code;
814
815     if (p[1] == 'f') {
816         fop->op = ngx\_http\_script\_file\_plain;
817         return NGX\_CONF\_OK;
818     }
819
820     if (p[1] == 'd') {
821         fop->op = ngx\_http\_script\_file\_dir;
822         return NGX\_CONF\_OK;
823     }
824
825     if (p[1] == 'e') {
826         fop->op = ngx\_http\_script\_file\_exists;
827         return NGX\_CONF\_OK;
828     }
829
830     if (p[1] == 'x') {
831         fop->op = ngx\_http\_script\_file\_exec;
832         return NGX\_CONF\_OK;
833     }
834
835     if (p[0] == '!') {
836         if (p[2] == 'f') {
837             fop->op = ngx\_http\_script\_file\_not\_plain;
838             return NGX\_CONF\_OK;
839         }
840
841         if (p[2] == 'd') {
842             fop->op = ngx\_http\_script\_file\_not\_dir;
843             return NGX\_CONF\_OK;
844         }
845
846         if (p[2] == 'e') {
847             fop->op = ngx\_http\_script\_file\_not\_exists;
848             return NGX\_CONF\_OK;
849         }
850
851         if (p[2] == 'x') {
852             fop->op = ngx\_http\_script\_file\_not\_exec;
853             return NGX\_CONF\_OK;
854         }
855     }

```

```

856         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
857             "invalid condition \"%V\"", &value[cur]);
858         return NGX_CONF_ERROR;
859     }
860 }
861
862 ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
863     "invalid condition \"%V\"", &value[cur]);
864
865 return NGX_CONF_ERROR;
866 }
867
868
869 static char *
870 ngx_http_rewrite_variable(ngx_conf_t *cf, ngx_http_rewrite_loc_conf_t *lcf,
871     ngx_str_t *value)
872 {
873     ngx_int_t          index;
874     ngx_http_script_var_code_t *var_code;
875
876     value->len--;
877     value->data++;
878
879     index = ngx_http_get_variable_index(cf, value);
880
881     if (index == NGX_ERROR) {
882         return NGX_CONF_ERROR;
883     }
884
885     var_code = ngx_http_script_start_code(cf->pool, &lcf->codes,
886         sizeof(ngx_http_script_var_code_t));
887     if (var_code == NULL) {
888         return NGX_CONF_ERROR;
889     }
890
891     var_code->code = ngx_http_script_var_code;
892     var_code->index = index;
893
894     return NGX_CONF_OK;
895 }
896
897
898 static char *
899 ngx_http_rewrite_set(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
900 {
901     ngx_http_rewrite_loc_conf_t *lcf = conf;
902
903     ngx_int_t          index;
904     ngx_str_t          *value;
905     ngx_http_variable_t *v;
906     ngx_http_script_var_code_t *vcode;
907     ngx_http_script_var_handler_code_t *vhcode;
908
909     value = cf->args->elts;
910
911     if (value[1].data[0] != '$') {
912         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
913             "invalid variable name \"%V\"", &value[1]);
914         return NGX_CONF_ERROR;
915     }
916
917     value[1].len--;
918     value[1].data++;
919
920     v = ngx_http_add_variable(cf, &value[1], NGX_HTTP_VAR_CHANGEABLE);
921     if (v == NULL) {
922         return NGX_CONF_ERROR;
923     }
924
925     index = ngx_http_get_variable_index(cf, &value[1]);
926     if (index == NGX_ERROR) {
927         return NGX_CONF_ERROR;
928     }
929
930     if (v->get_handler == NULL
931         && ngx_strncasecmp(value[1].data, (u_char *) "http_", 5) != 0

```

```

932     && ngx_strncasecmp(value[1].data, (u_char *) "sent_http_", 10) != 0
933     && ngx_strncasecmp(value[1].data, (u_char *) "upstream_http_", 14) != 0
934     && ngx_strncasecmp(value[1].data, (u_char *) "cookie_", 7) != 0
935     && ngx_strncasecmp(value[1].data, (u_char *) "upstream_cookie_", 16)
936         != 0
937     && ngx_strncasecmp(value[1].data, (u_char *) "arg_", 4) != 0
938 {
939     v->get_handler = ngx_http_rewrite_var;
940     v->data = index;
941 }
942
943 if (ngx_http_rewrite_value(cf, lcf, &value[2]) != NGX_CONF_OK) {
944     return NGX_CONF_ERROR;
945 }
946
947 if (v->set_handler) {
948     vhcode = ngx_http_script_start_code(cf->pool, &lcf->codes,
949                                         sizeof(ngx_http_script_var_handler_code_t));
950     if (vhcode == NULL) {
951         return NGX_CONF_ERROR;
952     }
953
954     vhcode->code = ngx_http_script_var_set_handler_code;
955     vhcode->handler = v->set_handler;
956     vhcode->data = v->data;
957
958     return NGX_CONF_OK;
959 }
960
961 vcode = ngx_http_script_start_code(cf->pool, &lcf->codes,
962                                     sizeof(ngx_http_script_var_code_t));
963 if (vcode == NULL) {
964     return NGX_CONF_ERROR;
965 }
966
967 vcode->code = ngx_http_script_set_var_code;
968 vcode->index = (uintptr_t) index;
969
970 return NGX_CONF_OK;
971 }
972
973
974 static char *
975 ngx_http_rewrite_value(ngx_conf_t *cf, ngx_http_rewrite_loc_conf_t *lcf,
976                       ngx_str_t *value)
977 {
978     ngx_int_t          n;
979     ngx_http_script_compile_t  sc;
980     ngx_http_script_value_code_t *val;
981     ngx_http_script_complex_value_code_t *complex;
982
983     n = ngx_http_script_variables_count(value);
984
985     if (n == 0) {
986         val = ngx_http_script_start_code(cf->pool, &lcf->codes,
987                                         sizeof(ngx_http_script_value_code_t));
988         if (val == NULL) {
989             return NGX_CONF_ERROR;
990         }
991
992         n = ngx_atoi(value->data, value->len);
993
994         if (n == NGX_ERROR) {
995             n = 0;
996         }
997
998         val->code = ngx_http_script_value_code;
999         val->value = (uintptr_t) n;
1000         val->text_len = (uintptr_t) value->len;
1001         val->text_data = (uintptr_t) value->data;
1002
1003         return NGX_CONF_OK;
1004     }
1005
1006     complex = ngx_http_script_start_code(cf->pool, &lcf->codes,
1007                                         sizeof(ngx_http_script_complex_value_code_t));

```



```
1008     if (complex == NULL) {
1009         return NGX\_CONF\_ERROR;
1010     }
1011
1012     complex->code = ngx\_http\_script\_complex\_value\_code;
1013     complex->lengths = NULL;
1014
1015     ngx\_memzero(&sc, sizeof(ngx\_http\_script\_compile\_t));
1016
1017     sc.cf = cf;
1018     sc.source = value;
1019     sc.lengths = &complex->lengths;
1020     sc.values = &lcf->codes;
1021     sc.variables = n;
1022     sc.complete_lengths = 1;
1023
1024     if (ngx\_http\_script\_compile(&sc) != NGX\_OK) {
1025         return NGX\_CONF\_ERROR;
1026     }
1027
1028     return NGX\_CONF\_OK;
1029 }
```

[One Level Up](#)

[Top Level](#)

## src/http/modules/nginx\_http\_scgi\_module.c - nginx-1.7.10

### Global variables defined

- [ngx\\_http\\_scgi\\_cache\\_headers](#)
- [ngx\\_http\\_scgi\\_commands](#)
- [ngx\\_http\\_scgi\\_hide\\_headers](#)
- [ngx\\_http\\_scgi\\_module](#)
- [ngx\\_http\\_scgi\\_module](#)
- [ngx\\_http\\_scgi\\_module\\_ctx](#)
- [ngx\\_http\\_scgi\\_next\\_upstream\\_masks](#)
- [ngx\\_http\\_scgi\\_temp\\_path](#)

### Data types defined

- [ngx\\_http\\_scgi\\_loc\\_conf\\_t](#)
- [ngx\\_http\\_scgi\\_main\\_conf\\_t](#)
- [ngx\\_http\\_scgi\\_params\\_t](#)

### Functions defined

- [ngx\\_http\\_scgi\\_abort\\_request](#)
- [ngx\\_http\\_scgi\\_cache](#)
- [ngx\\_http\\_scgi\\_cache\\_key](#)
- [ngx\\_http\\_scgi\\_create\\_key](#)
- [ngx\\_http\\_scgi\\_create\\_loc\\_conf](#)
- [ngx\\_http\\_scgi\\_create\\_main\\_conf](#)
- [ngx\\_http\\_scgi\\_create\\_request](#)
- [ngx\\_http\\_scgi\\_eval](#)
- [ngx\\_http\\_scgi\\_finalize\\_request](#)
- [ngx\\_http\\_scgi\\_handler](#)
- [ngx\\_http\\_scgi\\_init\\_params](#)
- [ngx\\_http\\_scgi\\_merge\\_loc\\_conf](#)
- [ngx\\_http\\_scgi\\_pass](#)
- [ngx\\_http\\_scgi\\_process\\_header](#)
- [ngx\\_http\\_scgi\\_process\\_status\\_line](#)
- [ngx\\_http\\_scgi\\_reinit\\_request](#)

- [ngx\\_http\\_scgi\\_store](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  * Copyright (C) Manlio Perillo (manlio.perillo@gmail.com)
6  */
7
8
9 #include <ngx_config.h>
10 #include <ngx_core.h>
11 #include <ngx_http.h>
12
13
14 typedef struct {
15     ngx_array_t          caches; /* ngx_http_file_cache_t * */
16 } ngx_http_scgi_main_conf_t;
17
18
19 typedef struct {
20     ngx_array_t          *flushes;
21     ngx_array_t          *lengths;
22     ngx_array_t          *values;
23     ngx_uint_t           number;
24     ngx_hash_t           hash;
25 } ngx_http_scgi_params_t;
26
27
28 typedef struct {
29     ngx_http_upstream_conf_t  upstream;
30
31     ngx_http_scgi_params_t    params;
32 #if (NGX_HTTP_CACHE)
33     ngx_http_scgi_params_t    params_cache;
34 #endif
35     ngx_array_t               *params_source;
36
37     ngx_array_t               *scgi_lengths;
38     ngx_array_t               *scgi_values;
39
40 #if (NGX_HTTP_CACHE)
41     ngx_http_complex_value_t   cache_key;
42 #endif
43 } ngx_http_scgi_loc_conf_t;
44
45
46 static ngx_int_t ngx_http_scgi_eval(ngx_http_request_t *r,
47     ngx_http_scgi_loc_conf_t *scf);
48 static ngx_int_t ngx_http_scgi_create_request(ngx_http_request_t *r);
49 static ngx_int_t ngx_http_scgi_reinit_request(ngx_http_request_t *r);
50 static ngx_int_t ngx_http_scgi_process_status_line(ngx_http_request_t *r);
51 static ngx_int_t ngx_http_scgi_process_header(ngx_http_request_t *r);
52 static void ngx_http_scgi_abort_request(ngx_http_request_t *r);
53 static void ngx_http_scgi_finalize_request(ngx_http_request_t *r, ngx_int_t rc);
54
55 static void *ngx_http_scgi_create_main_conf(ngx_conf_t *cf);
56 static void *ngx_http_scgi_create_loc_conf(ngx_conf_t *cf);
57 static char *ngx_http_scgi_merge_loc_conf(ngx_conf_t *cf, void *parent,
58     void *child);
59 static ngx_int_t ngx_http_scgi_init_params(ngx_conf_t *cf,
60     ngx_http_scgi_loc_conf_t *conf, ngx_http_scgi_params_t *params,
61     ngx_keyval_t *default_params);
62
63 static char *ngx_http_scgi_pass(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
64 static char *ngx_http_scgi_store(ngx_conf_t *cf, ngx_command_t *cmd,
65     void *conf);
66
67 #if (NGX_HTTP_CACHE)
68 static ngx_int_t ngx_http_scgi_create_key(ngx_http_request_t *r);
69 static char *ngx_http_scgi_cache(ngx_conf_t *cf, ngx_command_t *cmd,
70     void *conf);
```

```

71 static char *ngx_http_scgi_cache_key(ngx_conf_t *cf, ngx_command_t *cmd,
72     void *conf);
73 #endif
74
75
76 static ngx_conf_bitmask_t ngx_http_scgi_next_upstream_masks[] = {
77     { ngx_string("error"), NGX_HTTP_UPSTREAM_FT_ERROR },
78     { ngx_string("timeout"), NGX_HTTP_UPSTREAM_FT_TIMEOUT },
79     { ngx_string("invalid_header"), NGX_HTTP_UPSTREAM_FT_INVALID_HEADER },
80     { ngx_string("http_500"), NGX_HTTP_UPSTREAM_FT_HTTP_500 },
81     { ngx_string("http_503"), NGX_HTTP_UPSTREAM_FT_HTTP_503 },
82     { ngx_string("http_403"), NGX_HTTP_UPSTREAM_FT_HTTP_403 },
83     { ngx_string("http_404"), NGX_HTTP_UPSTREAM_FT_HTTP_404 },
84     { ngx_string("updating"), NGX_HTTP_UPSTREAM_FT_UPDATING },
85     { ngx_string("off"), NGX_HTTP_UPSTREAM_FT_OFF },
86     { ngx_null_string, 0 }
87 };
88
89
90 ngx_module_t ngx_http_scgi_module;
91
92
93 static ngx_command_t ngx_http_scgi_commands[] = {
94
95     { ngx_string("scgi_pass"),
96       NGX_HTTP_LOC_CONF|NGX_HTTP_LIF_CONF|NGX_CONF TAKE1,
97       ngx_http_scgi_pass,
98       NGX_HTTP_LOC_CONF_OFFSET,
99       0,
100      NULL },
101
102     { ngx_string("scgi_store"),
103       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF TAKE1,
104       ngx_http_scgi_store,
105       NGX_HTTP_LOC_CONF_OFFSET,
106       0,
107       NULL },
108
109     { ngx_string("scgi_store_access"),
110       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF TAKE123,
111       ngx_conf_set_access_slot,
112       NGX_HTTP_LOC_CONF_OFFSET,
113       offsetof(ngx_http_scgi_loc_conf_t, upstream.store_access),
114       NULL },
115
116     { ngx_string("scgi_buffering"),
117       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF FLAG,
118       ngx_conf_set_flag_slot,
119       NGX_HTTP_LOC_CONF_OFFSET,
120       offsetof(ngx_http_scgi_loc_conf_t, upstream.buffering),
121       NULL },
122
123     { ngx_string("scgi_ignore_client_abort"),
124       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF FLAG,
125       ngx_conf_set_flag_slot,
126       NGX_HTTP_LOC_CONF_OFFSET,
127       offsetof(ngx_http_scgi_loc_conf_t, upstream.ignore_client_abort),
128       NULL },
129
130     { ngx_string("scgi_bind"),
131       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF TAKE1,
132       ngx_http_upstream_bind_set_slot,
133       NGX_HTTP_LOC_CONF_OFFSET,
134       offsetof(ngx_http_scgi_loc_conf_t, upstream.local),
135       NULL },
136
137     { ngx_string("scgi_connect_timeout"),
138       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF TAKE1,
139       ngx_conf_set_msec_slot,
140       NGX_HTTP_LOC_CONF_OFFSET,
141       offsetof(ngx_http_scgi_loc_conf_t, upstream.connect_timeout),
142       NULL },
143
144     { ngx_string("scgi_send_timeout"),
145       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF TAKE1,
146       ngx_conf_set_msec_slot,

```

```

147     NGX_HTTP_LOC_CONF_OFFSET,
148     offsetof(ngx_http_scgi_loc_conf_t, upstream.send_timeout),
149     NULL },
150
151 { ngx_string("scgi_buffer_size"),
152   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
153   ngx_conf_set_size_slot,
154   NGX_HTTP_LOC_CONF_OFFSET,
155   offsetof(ngx_http_scgi_loc_conf_t, upstream.buffer_size),
156   NULL },
157
158 { ngx_string("scgi_pass_request_headers"),
159   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
160   ngx_conf_set_flag_slot,
161   NGX_HTTP_LOC_CONF_OFFSET,
162   offsetof(ngx_http_scgi_loc_conf_t, upstream.pass_request_headers),
163   NULL },
164
165 { ngx_string("scgi_pass_request_body"),
166   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
167   ngx_conf_set_flag_slot,
168   NGX_HTTP_LOC_CONF_OFFSET,
169   offsetof(ngx_http_scgi_loc_conf_t, upstream.pass_request_body),
170   NULL },
171
172 { ngx_string("scgi_intercept_errors"),
173   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
174   ngx_conf_set_flag_slot,
175   NGX_HTTP_LOC_CONF_OFFSET,
176   offsetof(ngx_http_scgi_loc_conf_t, upstream.intercept_errors),
177   NULL },
178
179 { ngx_string("scgi_read_timeout"),
180   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
181   ngx_conf_set_msec_slot,
182   NGX_HTTP_LOC_CONF_OFFSET,
183   offsetof(ngx_http_scgi_loc_conf_t, upstream.read_timeout),
184   NULL },
185
186 { ngx_string("scgi_buffers"),
187   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE2,
188   ngx_conf_set_bufs_slot,
189   NGX_HTTP_LOC_CONF_OFFSET,
190   offsetof(ngx_http_scgi_loc_conf_t, upstream.bufs),
191   NULL },
192
193 { ngx_string("scgi_busy_buffers_size"),
194   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
195   ngx_conf_set_size_slot,
196   NGX_HTTP_LOC_CONF_OFFSET,
197   offsetof(ngx_http_scgi_loc_conf_t, upstream.busy_buffers_size_conf),
198   NULL },
199
200 { ngx_string("scgi_force_ranges"),
201   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
202   ngx_conf_set_flag_slot,
203   NGX_HTTP_LOC_CONF_OFFSET,
204   offsetof(ngx_http_scgi_loc_conf_t, upstream.force_ranges),
205   NULL },
206
207 { ngx_string("scgi_limit_rate"),
208   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
209   ngx_conf_set_size_slot,
210   NGX_HTTP_LOC_CONF_OFFSET,
211   offsetof(ngx_http_scgi_loc_conf_t, upstream.limit_rate),
212   NULL },
213
214 #if (NGX_HTTP_CACHE)
215
216 { ngx_string("scgi_cache"),
217   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
218   ngx_http_scgi_cache,
219   NGX_HTTP_LOC_CONF_OFFSET,
220   0,
221   NULL },
222

```

```

223 { ngx_string("scgi_cache_key"),
224     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
225     ngx_http_scgi_cache_key,
226     NGX\_HTTP\_LOC\_CONF\_OFFSET,
227     0,
228     NULL },
229
230 { ngx_string("scgi_cache_path"),
231     NGX\_HTTP\_MAIN\_CONF | NGX\_CONF\_2MORE,
232     ngx_http_file_cache_set_slot,
233     NGX\_HTTP\_MAIN\_CONF\_OFFSET,
234     offsetof(ngx\_http\_scgi\_main\_conf\_t, caches),
235     &ngx_http_scgi_module },
236
237 { ngx_string("scgi_cache_bypass"),
238     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_1MORE,
239     ngx_http_set_predicate_slot,
240     NGX\_HTTP\_LOC\_CONF\_OFFSET,
241     offsetof(ngx\_http\_scgi\_loc\_conf\_t, upstream.cache_bypass),
242     NULL },
243
244 { ngx_string("scgi_no_cache"),
245     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_1MORE,
246     ngx_http_set_predicate_slot,
247     NGX\_HTTP\_LOC\_CONF\_OFFSET,
248     offsetof(ngx\_http\_scgi\_loc\_conf\_t, upstream.no_cache),
249     NULL },
250
251 { ngx_string("scgi_cache_valid"),
252     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_1MORE,
253     ngx_http_file_cache_valid_set_slot,
254     NGX\_HTTP\_LOC\_CONF\_OFFSET,
255     offsetof(ngx\_http\_scgi\_loc\_conf\_t, upstream.cache_valid),
256     NULL },
257
258 { ngx_string("scgi_cache_min_uses"),
259     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
260     ngx_conf_set_num_slot,
261     NGX\_HTTP\_LOC\_CONF\_OFFSET,
262     offsetof(ngx\_http\_scgi\_loc\_conf\_t, upstream.cache_min_uses),
263     NULL },
264
265 { ngx_string("scgi_cache_use_stale"),
266     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_1MORE,
267     ngx_conf_set_bitmask_slot,
268     NGX\_HTTP\_LOC\_CONF\_OFFSET,
269     offsetof(ngx\_http\_scgi\_loc\_conf\_t, upstream.cache_use_stale),
270     &ngx_http_scgi_next_upstream_masks },
271
272 { ngx_string("scgi_cache_methods"),
273     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_1MORE,
274     ngx_conf_set_bitmask_slot,
275     NGX\_HTTP\_LOC\_CONF\_OFFSET,
276     offsetof(ngx\_http\_scgi\_loc\_conf\_t, upstream.cache_methods),
277     &ngx_http_upstream_cache_method_mask },
278
279 { ngx_string("scgi_cache_lock"),
280     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_FLAG,
281     ngx_conf_set_flag_slot,
282     NGX\_HTTP\_LOC\_CONF\_OFFSET,
283     offsetof(ngx\_http\_scgi\_loc\_conf\_t, upstream.cache_lock),
284     NULL },
285
286 { ngx_string("scgi_cache_lock_timeout"),
287     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
288     ngx_conf_set_msec_slot,
289     NGX\_HTTP\_LOC\_CONF\_OFFSET,
290     offsetof(ngx\_http\_scgi\_loc\_conf\_t, upstream.cache_lock_timeout),
291     NULL },
292
293 { ngx_string("scgi_cache_lock_age"),
294     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
295     ngx_conf_set_msec_slot,
296     NGX\_HTTP\_LOC\_CONF\_OFFSET,
297     offsetof(ngx\_http\_scgi\_loc\_conf\_t, upstream.cache_lock_age),
298     NULL },

```

```

299
300 { ngx_string("scgi_cache_revalidate"),
301     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_FLAG,
302     ngx_conf_set_flag_slot,
303     NGX\_HTTP\_LOC\_CONF\_OFFSET,
304     offsetof(ngx\_http\_scgi\_loc\_conf\_t, upstream.cache_revalidate),
305     NULL },
306
307 #endif
308
309 { ngx_string("scgi_temp_path"),
310     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1234,
311     ngx_conf_set_path_slot,
312     NGX\_HTTP\_LOC\_CONF\_OFFSET,
313     offsetof(ngx\_http\_scgi\_loc\_conf\_t, upstream.temp_path),
314     NULL },
315
316 { ngx_string("scgi_max_temp_file_size"),
317     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
318     ngx_conf_set_size_slot,
319     NGX\_HTTP\_LOC\_CONF\_OFFSET,
320     offsetof(ngx\_http\_scgi\_loc\_conf\_t, upstream.max_temp_file_size_conf),
321     NULL },
322
323 { ngx_string("scgi_temp_file_write_size"),
324     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
325     ngx_conf_set_size_slot,
326     NGX\_HTTP\_LOC\_CONF\_OFFSET,
327     offsetof(ngx\_http\_scgi\_loc\_conf\_t, upstream.temp_file_write_size_conf),
328     NULL },
329
330 { ngx_string("scgi_next_upstream"),
331     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_1MORE,
332     ngx_conf_set_bitmask_slot,
333     NGX\_HTTP\_LOC\_CONF\_OFFSET,
334     offsetof(ngx\_http\_scgi\_loc\_conf\_t, upstream.next_upstream),
335     &ngx\_http\_scgi\_next\_upstream\_masks },
336
337 { ngx_string("scgi_next_upstream_tries"),
338     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
339     ngx_conf_set_num_slot,
340     NGX\_HTTP\_LOC\_CONF\_OFFSET,
341     offsetof(ngx\_http\_scgi\_loc\_conf\_t, upstream.next_upstream_tries),
342     NULL },
343
344 { ngx_string("scgi_next_upstream_timeout"),
345     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
346     ngx_conf_set_msec_slot,
347     NGX\_HTTP\_LOC\_CONF\_OFFSET,
348     offsetof(ngx\_http\_scgi\_loc\_conf\_t, upstream.next_upstream_timeout),
349     NULL },
350
351 { ngx_string("scgi_param"),
352     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE23,
353     ngx_http_upstream_param_set_slot,
354     NGX\_HTTP\_LOC\_CONF\_OFFSET,
355     offsetof(ngx\_http\_scgi\_loc\_conf\_t, params_source),
356     NULL },
357
358 { ngx_string("scgi_pass_header"),
359     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
360     ngx_conf_set_str_array_slot,
361     NGX\_HTTP\_LOC\_CONF\_OFFSET,
362     offsetof(ngx\_http\_scgi\_loc\_conf\_t, upstream.pass_headers),
363     NULL },
364
365 { ngx_string("scgi_hide_header"),
366     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE1,
367     ngx_conf_set_str_array_slot,
368     NGX\_HTTP\_LOC\_CONF\_OFFSET,
369     offsetof(ngx\_http\_scgi\_loc\_conf\_t, upstream.hide_headers),
370     NULL },
371
372 { ngx_string("scgi_ignore_headers"),
373     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_1MORE,
374     ngx_conf_set_bitmask_slot,

```

```

375     NGX_HTTP_LOC_CONF_OFFSET,
376     offsetof(ngx_http_scgi_loc_conf_t, upstream.ignore_headers),
377     &ngx_http_upstream_ignore_headers_masks },
378
379     ngx_null_command
380 };
381
382
383 static ngx_http_module_t ngx_http_scgi_module_ctx = {
384     NULL,                                /* preconfiguration */
385     NULL,                                /* postconfiguration */
386
387     ngx_http_scgi_create_main_conf,    /* create main configuration */
388     NULL,                                /* init main configuration */
389
390     NULL,                                /* create server configuration */
391     NULL,                                /* merge server configuration */
392
393     ngx_http_scgi_create_loc_conf,    /* create location configuration */
394     ngx_http_scgi_merge_loc_conf    /* merge location configuration */
395 };
396
397
398 ngx_module_t ngx_http_scgi_module = {
399     NGX_MODULE_V1,
400     &ngx_http_scgi_module_ctx,        /* module context */
401     ngx_http_scgi_commands,        /* module directives */
402     NGX_HTTP_MODULE,                /* module type */
403     NULL,                              /* init master */
404     NULL,                              /* init module */
405     NULL,                              /* init process */
406     NULL,                              /* init thread */
407     NULL,                              /* exit thread */
408     NULL,                              /* exit process */
409     NULL,                              /* exit master */
410     NGX_MODULE_V1_PADDING
411 };
412
413
414 static ngx_str_t ngx_http_scgi_hide_headers[] = {
415     ngx_string("Status"),
416     ngx_string("X-Accel-Expires"),
417     ngx_string("X-Accel-Redirect"),
418     ngx_string("X-Accel-Limit-Rate"),
419     ngx_string("X-Accel-Buffering"),
420     ngx_string("X-Accel-Charset"),
421     ngx_null_string
422 };
423
424
425 #if (NGX_HTTP_CACHE)
426
427 static ngx_keyval_t ngx_http_scgi_cache_headers[] = {
428     { ngx_string("HTTP_IF_MODIFIED_SINCE"),
429       ngx_string("$upstream_cache_last_modified") },
430     { ngx_string("HTTP_IF_UNMODIFIED_SINCE"), ngx_string("") },
431     { ngx_string("HTTP_IF_NONE_MATCH"), ngx_string("$upstream_cache_etag") },
432     { ngx_string("HTTP_IF_MATCH"), ngx_string("") },
433     { ngx_string("HTTP_RANGE"), ngx_string("") },
434     { ngx_string("HTTP_IF_RANGE"), ngx_string("") },
435     { ngx_null_string, ngx_null_string }
436 };
437
438 #endif
439
440
441 static ngx_path_init_t ngx_http_scgi_temp_path = {
442     ngx_string(NGX_HTTP_SCGI_TEMP_PATH), { 1, 2, 0 }
443 };
444
445
446 static ngx_int_t
447 ngx_http_scgi_handler(ngx_http_request_t *r)
448 {
449     ngx_int_t          rc;
450     ngx_http_status_t *status;

```



```

451     ngx_http_upstream_t      *u;
452     ngx_http_scgi_loc_conf_t *scf;
453 #if (NGX_HTTP_CACHE)
454     ngx_http_scgi_main_conf_t *smcf;
455 #endif
456
457     if (ngx_http_upstream_create(r) != NGX_OK) {
458         return NGX_HTTP_INTERNAL_SERVER_ERROR;
459     }
460
461     status = ngx_palloc(r->pool, sizeof(ngx_http_status_t));
462     if (status == NULL) {
463         return NGX_HTTP_INTERNAL_SERVER_ERROR;
464     }
465
466     ngx_http_set_ctx(r, status, ngx_http_scgi_module);
467
468     scf = ngx_http_get_module_loc_conf(r, ngx_http_scgi_module);
469
470     if (scf->scgi_lengths) {
471         if (ngx_http_scgi_eval(r, scf) != NGX_OK) {
472             return NGX_HTTP_INTERNAL_SERVER_ERROR;
473         }
474     }
475
476     u = r->upstream;
477
478     ngx_str_set(&u->schema, "scgi://");
479     u->output.tag = (ngx_buf_tag_t) &ngx_http_scgi_module;
480
481     u->conf = &scf->upstream;
482
483 #if (NGX_HTTP_CACHE)
484     smcf = ngx_http_get_module_main_conf(r, ngx_http_scgi_module);
485
486     u->caches = &smcf->caches;
487     u->create_key = ngx_http_scgi_create_key;
488 #endif
489
490     u->create_request = ngx_http_scgi_create_request;
491     u->reinit_request = ngx_http_scgi_reinit_request;
492     u->process_header = ngx_http_scgi_process_status_line;
493     u->abort_request = ngx_http_scgi_abort_request;
494     u->finalize_request = ngx_http_scgi_finalize_request;
495     r->state = 0;
496
497     u->buffering = scf->upstream.buffering;
498
499     u->pipe = ngx_palloc(r->pool, sizeof(ngx_event_pipe_t));
500     if (u->pipe == NULL) {
501         return NGX_HTTP_INTERNAL_SERVER_ERROR;
502     }
503
504     u->pipe->input_filter = ngx_event_pipe_copy_input_filter;
505     u->pipe->input_ctx = r;
506
507     rc = ngx_http_read_client_request_body(r, ngx_http_upstream_init);
508
509     if (rc >= NGX_HTTP_SPECIAL_RESPONSE) {
510         return rc;
511     }
512
513     return NGX_DONE;
514 }
515
516
517 static ngx_int_t
518 ngx_http_scgi_eval(ngx_http_request_t *r, ngx_http_scgi_loc_conf_t * scf)
519 {
520     ngx_url_t      url;
521     ngx_http_upstream_t *u;
522
523     ngx_memzero(&url, sizeof(ngx_url_t));
524
525     if (ngx_http_script_run(r, &url.url, scf->scgi_lengths->elts, 0,
526         scf->scgi_values->elts)

```

```

527     == NULL)
528     {
529         return NGX\_ERROR;
530     }
531
532     url.no_resolve = 1;
533
534     if (ngx\_parse\_url(r->pool, &url) != NGX\_OK) {
535         if (url.err) {
536             ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, 0,
537                 "%s in upstream \"%V\"", url.err, &url.url);
538         }
539
540         return NGX\_ERROR;
541     }
542
543     u = r->upstream;
544
545     u->resolved = ngx\_palloc(r->pool, sizeof(ngx\_http\_upstream\_resolved\_t));
546     if (u->resolved == NULL) {
547         return NGX\_ERROR;
548     }
549
550     if (url.addrs && url.addrs[0].sockaddr) {
551         u->resolved->sockaddr = url.addrs[0].sockaddr;
552         u->resolved->socklen = url.addrs[0].socklen;
553         u->resolved->naddrs = 1;
554         u->resolved->host = url.addrs[0].name;
555     } else {
556         u->resolved->host = url.host;
557         u->resolved->port = url.port;
558         u->resolved->no_port = url.no_port;
559     }
560
561     return NGX\_OK;
562 }
563
564
565
566 #if (NGX\_HTTP\_CACHE)
567
568 static ngx\_int\_t
569 ngx\_http\_scgi\_create\_key(ngx\_http\_request\_t *r)
570 {
571     ngx\_str\_t          *key;
572     ngx\_http\_scgi\_loc\_conf\_t *scf;
573
574     key = ngx\_array\_push(&r->cache->keys);
575     if (key == NULL) {
576         return NGX\_ERROR;
577     }
578
579     scf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_scgi\_module);
580
581     if (ngx\_http\_complex\_value(r, &scf->cache_key, key) != NGX\_OK) {
582         return NGX\_ERROR;
583     }
584
585     return NGX\_OK;
586 }
587
588 #endif
589
590
591 static ngx\_int\_t
592 ngx\_http\_scgi\_create\_request(ngx\_http\_request\_t *r)
593 {
594     off\_t                content_length_n;
595     u\_char              ch, *key, *val, *lowercase_key;
596     size\_t              len, key_len, val_len, allocated;
597     ngx\_buf\_t           *b;
598     ngx\_str\_t           content_length;
599     ngx\_uint\_t          i, n, hash, skip_empty, header_params;
600     ngx\_chain\_t         *cl, *body;
601     ngx\_list\_part\_t     *part;
602     ngx\_table\_elt\_t     *header, **ignored;

```

```

603     ngx_http_scgi_params_t      *params;
604     ngx_http_script_code_pt     code;
605     ngx_http_script_engine_t    e, le;
606     ngx_http_scgi_loc_conf_t    *scf;
607     ngx_http_script_len_code_pt lcode;
608     u_char                      buffer[NGX_OFF_T_LEN];
609
610     content_length_n = 0;
611     body = r->upstream->request_bufs;
612
613     while (body) {
614         content_length_n += ngx_buf_size(body->buf);
615         body = body->next;
616     }
617
618     content_length.data = buffer;
619     content_length.len = ngx_sprintf(buffer, "%0", content_length_n) - buffer;
620
621     len = sizeof("CONTENT_LENGTH") + content_length.len + 1;
622
623     header_params = 0;
624     ignored = NULL;
625
626     scf = ngx_http_get_module_loc_conf(r, ngx_http_scgi_module);
627
628     #if (NGX_HTTP_CACHE)
629     params = r->upstream->cacheable ? &scf->params_cache : &scf->params;
630 #else
631     params = &scf->params;
632 #endif
633
634     if (params->lengths) {
635         ngx_memzero(&le, sizeof(ngx_http_script_engine_t));
636
637         ngx_http_script_flush_no_cacheable_variables(r, params->flushes);
638         le.flushed = 1;
639
640         le.ip = params->lengths->elts;
641         le.request = r;
642
643         while (*(uintptr_t *) le.ip) {
644
645             lcode = *(ngx_http_script_len_code_pt *) le.ip;
646             key_len = lcode(&le);
647
648             lcode = *(ngx_http_script_len_code_pt *) le.ip;
649             skip_empty = lcode(&le);
650
651             for (val_len = 0; *(uintptr_t *) le.ip; val_len += lcode(&le)) {
652                 lcode = *(ngx_http_script_len_code_pt *) le.ip;
653             }
654             le.ip += sizeof(uintptr_t);
655
656             if (skip_empty && val_len == 0) {
657                 continue;
658             }
659
660             len += key_len + val_len + 1;
661         }
662     }
663
664     if (scf->upstream.pass_request_headers) {
665
666         allocated = 0;
667         lowercase_key = NULL;
668
669         if (params->number) {
670             n = 0;
671             part = &r->headers_in.headers.part;
672
673             while (part) {
674                 n += part->nelts;
675                 part = part->next;
676             }
677
678             ignored = ngx_palloc(r->pool, n * sizeof(void *));

```

```

679     if (ignored == NULL) {
680         return NGX\_ERROR;
681     }
682 }
683
684 part = &r->headers_in.headers.part;
685 header = part->elts;
686
687 for (i = 0; /* void */; i++) {
688
689     if (i >= part->nelts) {
690         if (part->next == NULL) {
691             break;
692         }
693
694         part = part->next;
695         header = part->elts;
696         i = 0;
697     }
698
699     if (params->number) {
700         if (allocated < header[i].key.len) {
701             allocated = header[i].key.len + 16;
702             lowercase_key = ngx\_pnalloc(r->pool, allocated);
703             if (lowercase_key == NULL) {
704                 return NGX\_ERROR;
705             }
706         }
707
708         hash = 0;
709
710         for (n = 0; n < header[i].key.len; n++) {
711             ch = header[i].key.data[n];
712
713             if (ch >= 'A' && ch <= 'Z') {
714                 ch |= 0x20;
715
716             } else if (ch == '-') {
717                 ch = '_';
718             }
719
720             hash = ngx\_hash(hash, ch);
721             lowercase_key[n] = ch;
722         }
723
724         if (ngx\_hash\_find(&params->hash, hash, lowercase_key, n)) {
725             ignored[header_params++] = &header[i];
726             continue;
727         }
728     }
729
730     len += sizeof("HTTP_") - 1 + header[i].key.len + 1
731         + header[i].value.len + 1;
732 }
733 }
734
735 /* netstring: "length:" + packet + "," */
736
737 b = ngx\_create\_temp\_buf(r->pool, NGX_SIZE_T_LEN + 1 + len + 1);
738 if (b == NULL) {
739     return NGX\_ERROR;
740 }
741
742 cl = ngx\_alloc\_chain\_link(r->pool);
743 if (cl == NULL) {
744     return NGX\_ERROR;
745 }
746
747 cl->buf = b;
748
749 b->last = ngx\_sprintf(b->last, "%ui:CONTENT_LENGTH%Z%V%Z",
750     len, &content_length);
751
752 if (params->lengths) {
753     ngx\_memzero(&e, sizeof(ngx\_http\_script\_engine\_t));
754

```

```

755     e.ip = params->values->elts;
756     e.pos = b->last;
757     e.request = r;
758     e.flushed = 1;
759
760     le.ip = params->lengths->elts;
761
762     while (*(uintptr_t *) le.ip) {
763
764         lcode = *(ngx_http_script_len_code_pt *) le.ip;
765         lcode(&le); /* key length */
766
767         lcode = *(ngx_http_script_len_code_pt *) le.ip;
768         skip_empty = lcode(&le);
769
770         for (val_len = 0; *(uintptr_t *) le.ip; val_len += lcode(&le)) {
771             lcode = *(ngx_http_script_len_code_pt *) le.ip;
772         }
773         le.ip += sizeof(uintptr_t);
774
775         if (skip_empty && val_len == 0) {
776             e.skip = 1;
777
778             while (*(uintptr_t *) e.ip) {
779                 code = *(ngx_http_script_code_pt *) e.ip;
780                 code((ngx_http_script_engine_t *) &e);
781             }
782             e.ip += sizeof(uintptr_t);
783
784             e.skip = 0;
785
786             continue;
787         }
788
789         #if (NGX_DEBUG)
790             key = e.pos;
791         #endif
792
793         code = *(ngx_http_script_code_pt *) e.ip;
794         code((ngx_http_script_engine_t *) &e);
795
796         #if (NGX_DEBUG)
797             val = e.pos;
798         #endif
799
800         while (*(uintptr_t *) e.ip) {
801             code = *(ngx_http_script_code_pt *) e.ip;
802             code((ngx_http_script_engine_t *) &e);
803         }
804         *e.pos++ = '\0';
805         e.ip += sizeof(uintptr_t);
806
807         ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
808             "scgi param: \"%s: %s\"", key, val);
809     }
810     b->last = e.pos;
811 }
812
813 if (scf->upstream.pass_request_headers) {
814     part = &r->headers_in.headers.part;
815     header = part->elts;
816
817     for (i = 0; /* void */; i++) {
818
819         if (i >= part->nelts) {
820             if (part->next == NULL) {
821                 break;
822             }
823
824             part = part->next;
825             header = part->elts;
826             i = 0;
827         }
828
829         for (n = 0; n < header_params; n++) {
830             if (&header[i] == ignored[n]) {

```

```

831         goto next;
832     }
833 }
834
835 key = b->last;
836 b->last = ngx_cpymem(key, "HTTP_", sizeof("HTTP_") - 1);
837
838 for (n = 0; n < header[i].key.len; n++) {
839     ch = header[i].key.data[n];
840
841     if (ch >= 'a' && ch <= 'z') {
842         ch &= ~0x20;
843
844     } else if (ch == '-') {
845         ch = '_';
846     }
847
848     *b->last++ = ch;
849 }
850
851 *b->last++ = (u_char) 0;
852
853 val = b->last;
854 b->last = ngx_copy(val, header[i].value.data, header[i].value.len);
855 *b->last++ = (u_char) 0;
856
857 ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
858               "scgi param: \"%s: %s\"", key, val);
859
860 next:
861
862     continue;
863 }
864 }
865
866 *b->last++ = (u_char) ',';
867
868 if (scf->upstream.pass_request_body) {
869     body = r->upstream->request_bufs;
870     r->upstream->request_bufs = c1;
871
872     while (body) {
873         b = ngx_alloc_buf(r->pool);
874         if (b == NULL) {
875             return NGX_ERROR;
876         }
877
878         ngx_memcpy(b, body->buf, sizeof(ngx_buf_t));
879
880         c1->next = ngx_alloc_chain_link(r->pool);
881         if (c1->next == NULL) {
882             return NGX_ERROR;
883         }
884
885         c1 = c1->next;
886         c1->buf = b;
887
888         body = body->next;
889     }
890 } else {
891     r->upstream->request_bufs = c1;
892 }
893
894 c1->next = NULL;
895
896 return NGX_OK;
897 }
898 }
899
900 static ngx_int_t
901 ngx_http_scgi_reinit_request(ngx_http_request_t *r)
902 {
903     ngx_http_status_t *status;
904
905     status = ngx_http_get_module_ctx(r, ngx_http_scgi_module);

```

```

907     if (status == NULL) {
908         return NGX_OK;
909     }
910
911     status->code = 0;
912     status->count = 0;
913     status->start = NULL;
914     status->end = NULL;
915
916     r->upstream->process_header = ngx_http_scgi_process_status_line;
917     r->state = 0;
918
919     return NGX_OK;
920 }
921
922
923
924 static ngx_int_t
925 ngx_http_scgi_process_status_line(ngx_http_request_t *r)
926 {
927     size_t          len;
928     ngx_int_t      rc;
929     ngx_http_status_t *status;
930     ngx_http_upstream_t *u;
931
932     status = ngx_http_get_module_ctx(r, ngx_http_scgi_module);
933
934     if (status == NULL) {
935         return NGX_ERROR;
936     }
937
938     u = r->upstream;
939
940     rc = ngx_http_parse_status_line(r, &u->buffer, status);
941
942     if (rc == NGX_AGAIN) {
943         return rc;
944     }
945
946     if (rc == NGX_ERROR) {
947         u->process_header = ngx_http_scgi_process_header;
948         return ngx_http_scgi_process_header(r);
949     }
950
951     if (u->state && u->state->status == 0) {
952         u->state->status = status->code;
953     }
954
955     u->headers_in.status_n = status->code;
956
957     len = status->end - status->start;
958     u->headers_in.status_line.len = len;
959
960     u->headers_in.status_line.data = ngx_pnalloc(r->pool, len);
961     if (u->headers_in.status_line.data == NULL) {
962         return NGX_ERROR;
963     }
964
965     ngx_memcpy(u->headers_in.status_line.data, status->start, len);
966
967     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
968                 "http scgi status %ui \"%V\"",
969                 u->headers_in.status_n, &u->headers_in.status_line);
970
971     u->process_header = ngx_http_scgi_process_header;
972
973     return ngx_http_scgi_process_header(r);
974 }
975
976
977 static ngx_int_t
978 ngx_http_scgi_process_header(ngx_http_request_t *r)
979 {
980     ngx_str_t          *status_line;
981     ngx_int_t          rc, status;
982     ngx_table_elt_t    *h;

```

```

983     ngx_http_upstream_t      *u;
984     ngx_http_upstream_header_t *hh;
985     ngx_http_upstream_main_conf_t *umcf;
986
987     umcf = ngx_http_get_module_main_conf(r, ngx_http_upstream_module);
988
989     for ( ;; ) {
990
991         rc = ngx_http_parse_header_line(r, &r->upstream->buffer, 1);
992
993         if (rc == NGX_OK) {
994
995             /* a header line has been parsed successfully */
996
997             h = ngx_list_push(&r->upstream->headers_in.headers);
998             if (h == NULL) {
999                 return NGX_ERROR;
1000             }
1001
1002             h->hash = r->header_hash;
1003
1004             h->key.len = r->header_name_end - r->header_name_start;
1005             h->value.len = r->header_end - r->header_start;
1006
1007             h->key.data = ngx_pnalloc(r->pool,
1008                                     h->key.len + 1 + h->value.len + 1
1009                                     + h->key.len);
1010             if (h->key.data == NULL) {
1011                 return NGX_ERROR;
1012             }
1013
1014             h->value.data = h->key.data + h->key.len + 1;
1015             h->lowercase_key = h->key.data + h->key.len + 1 + h->value.len + 1;
1016
1017             ngx_memcpy(h->key.data, r->header_name_start, h->key.len);
1018             h->key.data[h->key.len] = '\0';
1019             ngx_memcpy(h->value.data, r->header_start, h->value.len);
1020             h->value.data[h->value.len] = '\0';
1021
1022             if (h->key.len == r->lowercase_index) {
1023                 ngx_memcpy(h->lowercase_key, r->lowercase_header, h->key.len);
1024             } else {
1025                 ngx_strlow(h->lowercase_key, h->key.data, h->key.len);
1026             }
1027
1028             hh = ngx_hash_find(&umcf->headers_in_hash, h->hash,
1029                              h->lowercase_key, h->key.len);
1030
1031             if (hh && hh->handler(r, h, hh->offset) != NGX_OK) {
1032                 return NGX_ERROR;
1033             }
1034
1035             ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1036                          "http scgi header: \"%V: %V\"", &h->key, &h->value);
1037
1038             continue;
1039         }
1040
1041         if (rc == NGX_HTTP_PARSE_HEADER_DONE) {
1042
1043             /* a whole header has been parsed successfully */
1044
1045             ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1046                          "http scgi header done");
1047
1048             u = r->upstream;
1049
1050             if (u->headers_in.status_n) {
1051                 goto done;
1052             }
1053
1054             if (u->headers_in.status) {
1055                 status_line = &u->headers_in.status->value;
1056
1057                 status = ngx_atoi(status_line->data, 3);

```



```

1059     if (status == NGX\_ERROR) {
1060         ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, 0,
1061             "upstream sent invalid status \"%V\"",
1062             status_line);
1063         return NGX\_HTTP\_UPSTREAM\_INVALID\_HEADER;
1064     }
1065
1066     u->headers_in.status_n = status;
1067     u->headers_in.status_line = *status_line;
1068
1069     } else if (u->headers_in.location) {
1070         u->headers_in.status_n = 302;
1071         ngx\_str\_set(&u->headers_in.status_line,
1072             "302 Moved Temporarily");
1073
1074     } else {
1075         u->headers_in.status_n = 200;
1076         ngx\_str\_set(&u->headers_in.status_line, "200 OK");
1077     }
1078
1079     if (u->state && u->state->status == 0) {
1080         u->state->status = u->headers_in.status_n;
1081     }
1082
1083     done:
1084
1085     if (u->headers_in.status_n == NGX\_HTTP\_SWITCHING\_PROTOCOLS
1086         && r->headers_in.upgrade)
1087     {
1088         u->upgrade = 1;
1089     }
1090
1091     return NGX\_OK;
1092 }
1093
1094 if (rc == NGX\_AGAIN) {
1095     return NGX\_AGAIN;
1096 }
1097
1098 /* there was error while a header line parsing */
1099
1100 ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, 0,
1101     "upstream sent invalid header");
1102
1103     return NGX\_HTTP\_UPSTREAM\_INVALID\_HEADER;
1104 }
1105 }
1106
1107
1108 static void
1109 ngx\_http\_scgi\_abort\_request(ngx\_http\_request\_t *r)
1110 {
1111     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
1112         "abort http scgi request");
1113
1114     return;
1115 }
1116
1117
1118 static void
1119 ngx\_http\_scgi\_finalize\_request(ngx\_http\_request\_t *r, ngx\_int\_t rc)
1120 {
1121     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
1122         "finalize http scgi request");
1123
1124     return;
1125 }
1126
1127
1128 static void *
1129 ngx\_http\_scgi\_create\_main\_conf(ngx\_conf\_t *cf)
1130 {
1131     ngx\_http\_scgi\_main\_conf\_t *conf;
1132
1133     conf = ngx\_palloc(cf->pool, sizeof(ngx\_http\_scgi\_main\_conf\_t));
1134     if (conf == NULL) {

```

```

1135     return NULL;
1136 }
1137
1138 #if (NGX_HTTP_CACHE)
1139     if (ngx_array_init(&conf->caches, cf->pool, 4,
1140         sizeof(ngx_http_file_cache_t *)))
1141         != NGX_OK)
1142     {
1143         return NULL;
1144     }
1145 #endif
1146
1147     return conf;
1148 }
1149
1150
1151 static void *
1152 ngx_http_scgi_create_loc_conf(ngx_conf_t *cf)
1153 {
1154     ngx_http_scgi_loc_conf_t *conf;
1155
1156     conf = ngx_palloc(cf->pool, sizeof(ngx_http_scgi_loc_conf_t));
1157     if (conf == NULL) {
1158         return NULL;
1159     }
1160
1161     conf->upstream.store = NGX_CONF_UNSET;
1162     conf->upstream.store_access = NGX_CONF_UNSET_UINT;
1163     conf->upstream.next_upstream_tries = NGX_CONF_UNSET_UINT;
1164     conf->upstream.buffering = NGX_CONF_UNSET;
1165     conf->upstream.ignore_client_abort = NGX_CONF_UNSET;
1166     conf->upstream.force_ranges = NGX_CONF_UNSET;
1167
1168     conf->upstream.local = NGX_CONF_UNSET_PTR;
1169
1170     conf->upstream.connect_timeout = NGX_CONF_UNSET_MSEC;
1171     conf->upstream.send_timeout = NGX_CONF_UNSET_MSEC;
1172     conf->upstream.read_timeout = NGX_CONF_UNSET_MSEC;
1173     conf->upstream.next_upstream_timeout = NGX_CONF_UNSET_MSEC;
1174
1175     conf->upstream.send_lowat = NGX_CONF_UNSET_SIZE;
1176     conf->upstream.buffer_size = NGX_CONF_UNSET_SIZE;
1177     conf->upstream.limit_rate = NGX_CONF_UNSET_SIZE;
1178
1179     conf->upstream.busy_buffers_size_conf = NGX_CONF_UNSET_SIZE;
1180     conf->upstream.max_temp_file_size_conf = NGX_CONF_UNSET_SIZE;
1181     conf->upstream.temp_file_write_size_conf = NGX_CONF_UNSET_SIZE;
1182
1183     conf->upstream.pass_request_headers = NGX_CONF_UNSET;
1184     conf->upstream.pass_request_body = NGX_CONF_UNSET;
1185
1186 #if (NGX_HTTP_CACHE)
1187     conf->upstream.cache = NGX_CONF_UNSET;
1188     conf->upstream.cache_min_uses = NGX_CONF_UNSET_UINT;
1189     conf->upstream.cache_bypass = NGX_CONF_UNSET_PTR;
1190     conf->upstream.no_cache = NGX_CONF_UNSET_PTR;
1191     conf->upstream.cache_valid = NGX_CONF_UNSET_PTR;
1192     conf->upstream.cache_lock = NGX_CONF_UNSET;
1193     conf->upstream.cache_lock_timeout = NGX_CONF_UNSET_MSEC;
1194     conf->upstream.cache_lock_age = NGX_CONF_UNSET_MSEC;
1195     conf->upstream.cache_revalidate = NGX_CONF_UNSET;
1196 #endif
1197
1198     conf->upstream.hide_headers = NGX_CONF_UNSET_PTR;
1199     conf->upstream.pass_headers = NGX_CONF_UNSET_PTR;
1200
1201     conf->upstream.intercept_errors = NGX_CONF_UNSET;
1202
1203     /* "scgi_cyclic_temp_file" is disabled */
1204     conf->upstream.cyclic_temp_file = 0;
1205
1206     conf->upstream.change_buffering = 1;
1207
1208     ngx_str_set(&conf->upstream.module, "scgi");
1209
1210     return conf;

```

```

1211 }
1212
1213
1214 static char *
1215 ngx_http_scgi_merge_loc_conf(ngx_conf_t *cf, void *parent, void *child)
1216 {
1217     ngx_http_scgi_loc_conf_t *prev = parent;
1218     ngx_http_scgi_loc_conf_t *conf = child;
1219
1220     size_t                size;
1221     ngx_int_t             rc;
1222     ngx_hash_init_t       hash;
1223     ngx_http_core_loc_conf_t *clcf;
1224
1225     #if (NGX_HTTP_CACHE)
1226
1227     if (conf->upstream.store > 0) {
1228         conf->upstream.cache = 0;
1229     }
1230
1231     if (conf->upstream.cache > 0) {
1232         conf->upstream.store = 0;
1233     }
1234
1235     #endif
1236
1237     if (conf->upstream.store == NGX_CONF_UNSET) {
1238         ngx_conf_merge_value(conf->upstream.store, prev->upstream.store, 0);
1239
1240         conf->upstream.store_lengths = prev->upstream.store_lengths;
1241         conf->upstream.store_values = prev->upstream.store_values;
1242     }
1243
1244     ngx_conf_merge_uint_value(conf->upstream.store_access,
1245                               prev->upstream.store_access, 0600);
1246
1247     ngx_conf_merge_uint_value(conf->upstream.next_upstream_tries,
1248                               prev->upstream.next_upstream_tries, 0);
1249
1250     ngx_conf_merge_value(conf->upstream.buffering,
1251                          prev->upstream.buffering, 1);
1252
1253     ngx_conf_merge_value(conf->upstream.ignore_client_abort,
1254                          prev->upstream.ignore_client_abort, 0);
1255
1256     ngx_conf_merge_value(conf->upstream.force_ranges,
1257                          prev->upstream.force_ranges, 0);
1258
1259     ngx_conf_merge_ptr_value(conf->upstream.local,
1260                              prev->upstream.local, NULL);
1261
1262     ngx_conf_merge_msec_value(conf->upstream.connect_timeout,
1263                               prev->upstream.connect_timeout, 60000);
1264
1265     ngx_conf_merge_msec_value(conf->upstream.send_timeout,
1266                               prev->upstream.send_timeout, 60000);
1267
1268     ngx_conf_merge_msec_value(conf->upstream.read_timeout,
1269                               prev->upstream.read_timeout, 60000);
1270
1271     ngx_conf_merge_msec_value(conf->upstream.next_upstream_timeout,
1272                               prev->upstream.next_upstream_timeout, 0);
1273
1274     ngx_conf_merge_size_value(conf->upstream.send_lowat,
1275                               prev->upstream.send_lowat, 0);
1276
1277     ngx_conf_merge_size_value(conf->upstream.buffer_size,
1278                               prev->upstream.buffer_size,
1279                               (size_t) ngx_pagesize);
1280
1281     ngx_conf_merge_size_value(conf->upstream.limit_rate,
1282                               prev->upstream.limit_rate, 0);
1283
1284
1285     ngx_conf_merge_bufs_value(conf->upstream.bufs, prev->upstream.bufs,
1286                               8, ngx_pagesize);

```

```

1287
1288 if (conf->upstream.bufs.num < 2) {
1289     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1290         "there must be at least 2 \"scgi_buffers\"");
1291     return NGX_CONF_ERROR;
1292 }
1293
1294
1295 size = conf->upstream.buffer_size;
1296 if (size < conf->upstream.bufs.size) {
1297     size = conf->upstream.bufs.size;
1298 }
1299
1300
1301 ngx_conf_merge_size_value(conf->upstream.busy_buffers_size_conf,
1302     prev->upstream.busy_buffers_size_conf,
1303     NGX_CONF_UNSET_SIZE);
1304
1305 if (conf->upstream.busy_buffers_size_conf == NGX_CONF_UNSET_SIZE) {
1306     conf->upstream.busy_buffers_size = 2 * size;
1307 } else {
1308     conf->upstream.busy_buffers_size =
1309         conf->upstream.busy_buffers_size_conf;
1310 }
1311
1312 if (conf->upstream.busy_buffers_size < size) {
1313     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1314         "\"scgi_busy_buffers_size\" must be equal to or greater "
1315         "than the maximum of the value of \"scgi_buffer_size\" and "
1316         "one of the \"scgi_buffers\"");
1317
1318     return NGX_CONF_ERROR;
1319 }
1320
1321 if (conf->upstream.busy_buffers_size
1322     > (conf->upstream.bufs.num - 1) * conf->upstream.bufs.size)
1323 {
1324     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1325         "\"scgi_busy_buffers_size\" must be less than "
1326         "the size of all \"scgi_buffers\" minus one buffer");
1327
1328     return NGX_CONF_ERROR;
1329 }
1330
1331
1332 ngx_conf_merge_size_value(conf->upstream.temp_file_write_size_conf,
1333     prev->upstream.temp_file_write_size_conf,
1334     NGX_CONF_UNSET_SIZE);
1335
1336 if (conf->upstream.temp_file_write_size_conf == NGX_CONF_UNSET_SIZE) {
1337     conf->upstream.temp_file_write_size = 2 * size;
1338 } else {
1339     conf->upstream.temp_file_write_size =
1340         conf->upstream.temp_file_write_size_conf;
1341 }
1342
1343 if (conf->upstream.temp_file_write_size < size) {
1344     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1345         "\"scgi_temp_file_write_size\" must be equal to or greater than "
1346         "the maximum of the value of \"scgi_buffer_size\" and "
1347         "one of the \"scgi_buffers\"");
1348
1349     return NGX_CONF_ERROR;
1350 }
1351
1352
1353 ngx_conf_merge_size_value(conf->upstream.max_temp_file_size_conf,
1354     prev->upstream.max_temp_file_size_conf,
1355     NGX_CONF_UNSET_SIZE);
1356
1357 if (conf->upstream.max_temp_file_size_conf == NGX_CONF_UNSET_SIZE) {
1358     conf->upstream.max_temp_file_size = 1024 * 1024 * 1024;
1359 } else {
1360     conf->upstream.max_temp_file_size =
1361         conf->upstream.max_temp_file_size_conf;
1362 }

```

```

1363 if (conf->upstream.max_temp_file_size != 0
1364     && conf->upstream.max_temp_file_size < size)
1365 {
1366     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1367         "\"scgi_max_temp_file_size\" must be equal to zero to disable \"
1368         \"temporary files usage or must be equal to or greater than \"
1369         \"the maximum of the value of \"scgi_buffer_size\" and \"
1370         \"one of the \"scgi_buffers\"");
1371
1372     return NGX_CONF_ERROR;
1373 }
1374
1375
1376 ngx_conf_merge_bitmask_value(conf->upstream.ignore_headers,
1377     prev->upstream.ignore_headers,
1378     NGX_CONF_BITMASK_SET);
1379
1380
1381 ngx_conf_merge_bitmask_value(conf->upstream.next_upstream,
1382     prev->upstream.next_upstream,
1383     (NGX_CONF_BITMASK_SET
1384      |NGX_HTTP_UPSTREAM_FT_ERROR
1385      |NGX_HTTP_UPSTREAM_FT_TIMEOUT));
1386
1387
1388 if (conf->upstream.next_upstream & NGX_HTTP_UPSTREAM_FT_OFF) {
1389     conf->upstream.next_upstream = NGX_CONF_BITMASK_SET
1390         |NGX_HTTP_UPSTREAM_FT_OFF;
1391 }
1392
1393 if (ngx_conf_merge_path_value(cf, &conf->upstream.temp_path,
1394     prev->upstream.temp_path,
1395     &ngx_http_scgi_temp_path)
1396     != NGX_OK)
1397 {
1398     return NGX_CONF_ERROR;
1399 }
1400
1401 #if (NGX_HTTP_CACHE)
1402
1403 if (conf->upstream.cache == NGX_CONF_UNSET) {
1404     ngx_conf_merge_value(conf->upstream.cache,
1405         prev->upstream.cache, 0);
1406
1407     conf->upstream.cache_zone = prev->upstream.cache_zone;
1408     conf->upstream.cache_value = prev->upstream.cache_value;
1409 }
1410
1411 if (conf->upstream.cache_zone && conf->upstream.cache_zone->data == NULL) {
1412     ngx_shm_zone_t *shm_zone;
1413
1414     shm_zone = conf->upstream.cache_zone;
1415
1416     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1417         "\"scgi_cache\" zone \"%V\" is unknown",
1418         &shm_zone->shm.name);
1419
1420     return NGX_CONF_ERROR;
1421 }
1422
1423 ngx_conf_merge_uint_value(conf->upstream.cache_min_uses,
1424     prev->upstream.cache_min_uses, 1);
1425
1426 ngx_conf_merge_bitmask_value(conf->upstream.cache_use_stale,
1427     prev->upstream.cache_use_stale,
1428     (NGX_CONF_BITMASK_SET
1429      |NGX_HTTP_UPSTREAM_FT_OFF));
1430
1431 if (conf->upstream.cache_use_stale & NGX_HTTP_UPSTREAM_FT_OFF) {
1432     conf->upstream.cache_use_stale = NGX_CONF_BITMASK_SET
1433         |NGX_HTTP_UPSTREAM_FT_OFF;
1434 }
1435
1436 if (conf->upstream.cache_use_stale & NGX_HTTP_UPSTREAM_FT_ERROR) {
1437     conf->upstream.cache_use_stale |= NGX_HTTP_UPSTREAM_FT_NOLIVE;
1438 }

```

```

1439
1440 if (conf->upstream.cache_methods == 0) {
1441     conf->upstream.cache_methods = prev->upstream.cache_methods;
1442 }
1443
1444 conf->upstream.cache_methods |= NGX\_HTTP\_GET|NGX\_HTTP\_HEAD;
1445
1446 ngx\_conf\_merge\_ptr\_value(conf->upstream.cache_bypass,
1447     prev->upstream.cache_bypass, NULL);
1448
1449 ngx\_conf\_merge\_ptr\_value(conf->upstream.no_cache,
1450     prev->upstream.no_cache, NULL);
1451
1452 ngx\_conf\_merge\_ptr\_value(conf->upstream.cache_valid,
1453     prev->upstream.cache_valid, NULL);
1454
1455 if (conf->cache_key.value.data == NULL) {
1456     conf->cache_key = prev->cache_key;
1457 }
1458
1459 if (conf->upstream.cache && conf->cache_key.value.data == NULL) {
1460     ngx\_conf\_log\_error(NGX\_LOG\_WARN, cf, 0,
1461         "no \"scgi_cache_key\" for \"scgi_cache\"");
1462 }
1463
1464 ngx\_conf\_merge\_value(conf->upstream.cache_lock,
1465     prev->upstream.cache_lock, 0);
1466
1467 ngx\_conf\_merge\_msec\_value(conf->upstream.cache_lock_timeout,
1468     prev->upstream.cache_lock_timeout, 5000);
1469
1470 ngx\_conf\_merge\_msec\_value(conf->upstream.cache_lock_age,
1471     prev->upstream.cache_lock_age, 5000);
1472
1473 ngx\_conf\_merge\_value(conf->upstream.cache_revalidate,
1474     prev->upstream.cache_revalidate, 0);
1475
1476 #endif
1477
1478 ngx\_conf\_merge\_value(conf->upstream.pass_request_headers,
1479     prev->upstream.pass_request_headers, 1);
1480 ngx\_conf\_merge\_value(conf->upstream.pass_request_body,
1481     prev->upstream.pass_request_body, 1);
1482
1483 ngx\_conf\_merge\_value(conf->upstream.intercept_errors,
1484     prev->upstream.intercept_errors, 0);
1485
1486 hash.max_size = 512;
1487 hash.bucket_size = ngx\_align(64, ngx\_cacheline\_size);
1488 hash.name = "scgi_hide_headers_hash";
1489
1490 if (ngx\_http\_upstream\_hide\_headers\_hash(cf, &conf->upstream,
1491     &prev->upstream, ngx\_http\_scgi\_hide\_headers, &hash)
1492     != NGX\_OK)
1493 {
1494     return NGX\_CONF\_ERROR;
1495 }
1496
1497 clcf = ngx\_http\_conf\_get\_module\_loc\_conf(cf, ngx\_http\_core\_module);
1498
1499 if (clcf->noname
1500     && conf->upstream.upstream == NULL && conf->scgi_lengths == NULL)
1501 {
1502     conf->upstream.upstream = prev->upstream.upstream;
1503     conf->scgi_lengths = prev->scgi_lengths;
1504     conf->scgi_values = prev->scgi_values;
1505 }
1506
1507 if (clcf->lmt_excpt && clcf->handler == NULL
1508     && (conf->upstream.upstream || conf->scgi_lengths))
1509 {
1510     clcf->handler = ngx\_http\_scgi\_handler;
1511 }
1512
1513 if (conf->params_source == NULL) {
1514     conf->params = prev->params;

```

```

1515 #if (NGX_HTTP_CACHE)
1516     conf->params_cache = prev->params_cache;
1517 #endif
1518     conf->params_source = prev->params_source;
1519 }
1520
1521 rc = ngx_http_scgi_init_params(cf, conf, &conf->params, NULL);
1522 if (rc != NGX_OK) {
1523     return NGX_CONF_ERROR;
1524 }
1525
1526 #if (NGX_HTTP_CACHE)
1527
1528     if (conf->upstream.cache) {
1529         rc = ngx_http_scgi_init_params(cf, conf, &conf->params_cache,
1530             ngx_http_scgi_cache_headers);
1531         if (rc != NGX_OK) {
1532             return NGX_CONF_ERROR;
1533         }
1534     }
1535
1536 #endif
1537
1538     return NGX_CONF_OK;
1539 }
1540
1541
1542 static ngx_int_t
1543 ngx_http_scgi_init_params(ngx_conf_t *cf, ngx_http_scgi_loc_conf_t *conf,
1544     ngx_http_scgi_params_t *params, ngx_keyval_t *default_params)
1545 {
1546     u_char                *p;
1547     size_t                size;
1548     uintptr_t             *code;
1549     ngx_uint_t            i, nsrc;
1550     ngx_array_t           headers_names, params_merged;
1551     ngx_keyval_t          *h;
1552     ngx_hash_key_t        *hk;
1553     ngx_hash_init_t       hash;
1554     ngx_http_upstream_param_t *src, *s;
1555     ngx_http_script_compile_t sc;
1556     ngx_http_script_copy_code_t *copy;
1557
1558     if (params->hash.buckets) {
1559         return NGX_OK;
1560     }
1561
1562     if (conf->params_source == NULL && default_params == NULL) {
1563         params->hash.buckets = (void *) 1;
1564         return NGX_OK;
1565     }
1566
1567     params->lengths = ngx_array_create(cf->pool, 64, 1);
1568     if (params->lengths == NULL) {
1569         return NGX_ERROR;
1570     }
1571
1572     params->values = ngx_array_create(cf->pool, 512, 1);
1573     if (params->values == NULL) {
1574         return NGX_ERROR;
1575     }
1576
1577     if (ngx_array_init(&headers_names, cf->temp_pool, 4, sizeof(ngx_hash_key_t))
1578         != NGX_OK)
1579     {
1580         return NGX_ERROR;
1581     }
1582
1583     if (conf->params_source) {
1584         src = conf->params_source->elts;
1585         nsrc = conf->params_source->nelts;
1586
1587     } else {
1588         src = NULL;
1589         nsrc = 0;
1590     }

```

```

1591
1592 if (default_params) {
1593     if (ngx\_array\_init(&params_merged, cf->temp_pool, 4,
1594         sizeof(ngx\_http\_upstream\_param\_t))
1595         != NGX\_OK)
1596     {
1597         return NGX\_ERROR;
1598     }
1599
1600     for (i = 0; i < nsrc; i++) {
1601
1602         s = ngx\_array\_push(&params_merged);
1603         if (s == NULL) {
1604             return NGX\_ERROR;
1605         }
1606
1607         *s = src[i];
1608     }
1609
1610     h = default_params;
1611
1612     while (h->key.len) {
1613
1614         src = params_merged.elts;
1615         nsrc = params_merged.nelts;
1616
1617         for (i = 0; i < nsrc; i++) {
1618             if (ngx\_strcasecmp(h->key.data, src[i].key.data) == 0) {
1619                 goto next;
1620             }
1621         }
1622
1623         s = ngx\_array\_push(&params_merged);
1624         if (s == NULL) {
1625             return NGX\_ERROR;
1626         }
1627
1628         s->key = h->key;
1629         s->value = h->value;
1630         s->skip_empty = 1;
1631
1632     next:
1633
1634         h++;
1635     }
1636
1637     src = params_merged.elts;
1638     nsrc = params_merged.nelts;
1639 }
1640
1641 for (i = 0; i < nsrc; i++) {
1642
1643     if (src[i].key.len > sizeof("HTTP_") - 1
1644         && ngx\_strncmp(src[i].key.data, "HTTP_", sizeof("HTTP_") - 1) == 0)
1645     {
1646         hk = ngx\_array\_push(&headers_names);
1647         if (hk == NULL) {
1648             return NGX\_ERROR;
1649         }
1650
1651         hk->key.len = src[i].key.len - 5;
1652         hk->key.data = src[i].key.data + 5;
1653         hk->key_hash = ngx\_hash\_key\_lc(hk->key.data, hk->key.len);
1654         hk->value = (void *) 1;
1655
1656         if (src[i].value.len == 0) {
1657             continue;
1658         }
1659     }
1660
1661     copy = ngx\_array\_push\_n(params->lengths,
1662         sizeof(ngx\_http\_script\_copy\_code\_t));
1663     if (copy == NULL) {
1664         return NGX\_ERROR;
1665     }
1666

```



```

1667     copy->code = (ngx\_http\_script\_code\_pt) ngx\_http\_script\_copy\_len\_code;
1668     copy->len = src[i].key.len + 1;
1669
1670     copy = ngx\_array\_push\_n(params->lengths,
1671                               sizeof(ngx\_http\_script\_copy\_code\_t));
1672     if (copy == NULL) {
1673         return NGX\_ERROR;
1674     }
1675
1676     copy->code = (ngx\_http\_script\_code\_pt) ngx\_http\_script\_copy\_len\_code;
1677     copy->len = src[i].skip_empty;
1678
1679
1680     size = (sizeof(ngx\_http\_script\_copy\_code\_t)
1681            + src[i].key.len + 1 + sizeof(uintptr\_t) - 1)
1682            & ~(sizeof(uintptr\_t) - 1);
1683
1684     copy = ngx\_array\_push\_n(params->values, size);
1685     if (copy == NULL) {
1686         return NGX\_ERROR;
1687     }
1688
1689     copy->code = ngx\_http\_script\_copy\_code;
1690     copy->len = src[i].key.len + 1;
1691
1692     p = (u\_char *) copy + sizeof(ngx\_http\_script\_copy\_code\_t);
1693     (void) ngx\_cpystrn(p, src[i].key.data, src[i].key.len + 1);
1694
1695
1696     ngx\_memzero(&sc, sizeof(ngx\_http\_script\_compile\_t));
1697
1698     sc.cf = cf;
1699     sc.source = &src[i].value;
1700     sc.flushes = &params->flushes;
1701     sc.lengths = &params->lengths;
1702     sc.values = &params->values;
1703
1704     if (ngx\_http\_script\_compile(&sc) != NGX\_OK) {
1705         return NGX\_ERROR;
1706     }
1707
1708     code = ngx\_array\_push\_n(params->lengths, sizeof(uintptr\_t));
1709     if (code == NULL) {
1710         return NGX\_ERROR;
1711     }
1712
1713     *code = (uintptr\_t) NULL;
1714
1715
1716     code = ngx\_array\_push\_n(params->values, sizeof(uintptr\_t));
1717     if (code == NULL) {
1718         return NGX\_ERROR;
1719     }
1720
1721     *code = (uintptr\_t) NULL;
1722 }
1723
1724 code = ngx\_array\_push\_n(params->lengths, sizeof(uintptr\_t));
1725 if (code == NULL) {
1726     return NGX\_ERROR;
1727 }
1728
1729 *code = (uintptr\_t) NULL;
1730
1731 params->number = headers_names.nelts;
1732
1733 hash.hash = &params->hash;
1734 hash.key = ngx\_hash\_key\_lc;
1735 hash.max_size = 512;
1736 hash.bucket_size = 64;
1737 hash.name = "scgi\_params\_hash";
1738 hash.pool = cf->pool;
1739 hash.temp_pool = NULL;
1740
1741 return ngx\_hash\_init(&hash, headers_names.elts, headers_names.nelts);
1742 }

```

```

1743
1744
1745 static char *
1746 ngx_http_scgi_pass(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
1747 {
1748     ngx_http_scgi_loc_conf_t *scf = conf;
1749
1750     ngx_url_t          u;
1751     ngx_str_t          *value, *url;
1752     ngx_uint_t         n;
1753     ngx_http_core_loc_conf_t *clcf;
1754     ngx_http_script_compile_t sc;
1755
1756     if (scf->upstream.upstream || scf->scgi_lengths) {
1757         return "is duplicate";
1758     }
1759
1760     clcf = ngx_http_conf_get_module_loc_conf(cf, ngx_http_core_module);
1761     clcf->handler = ngx_http_scgi_handler;
1762
1763     value = cf->args->elts;
1764
1765     url = &value[1];
1766
1767     n = ngx_http_script_variables_count(url);
1768
1769     if (n) {
1770
1771         ngx_memzero(&sc, sizeof(ngx_http_script_compile_t));
1772
1773         sc.cf = cf;
1774         sc.source = url;
1775         sc.lengths = &scf->scgi_lengths;
1776         sc.values = &scf->scgi_values;
1777         sc.variables = n;
1778         sc.complete_lengths = 1;
1779         sc.complete_values = 1;
1780
1781         if (ngx_http_script_compile(&sc) != NGX_OK) {
1782             return NGX_CONF_ERROR;
1783         }
1784
1785         return NGX_CONF_OK;
1786     }
1787
1788     ngx_memzero(&u, sizeof(ngx_url_t));
1789
1790     u.url = value[1];
1791     u.no_resolve = 1;
1792
1793     scf->upstream.upstream = ngx_http_upstream_add(cf, &u, 0);
1794     if (scf->upstream.upstream == NULL) {
1795         return NGX_CONF_ERROR;
1796     }
1797
1798     if (clcf->name.data[clcf->name.len - 1] == '/') {
1799         clcf->auto_redirect = 1;
1800     }
1801
1802     return NGX_CONF_OK;
1803 }
1804
1805
1806 static char *
1807 ngx_http_scgi_store(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
1808 {
1809     ngx_http_scgi_loc_conf_t *scf = conf;
1810
1811     ngx_str_t          *value;
1812     ngx_http_script_compile_t sc;
1813
1814     if (scf->upstream.store != NGX_CONF_UNSET) {
1815         return "is duplicate";
1816     }
1817
1818     value = cf->args->elts;

```

```

1819     if (ngx_strcmp(value[1].data, "off") == 0) {
1820         scf->upstream.store = 0;
1821         return NGX_CONF_OK;
1822     }
1823
1824     #if (NGX_HTTP_CACHE)
1825     if (scf->upstream.cache > 0) {
1826         return "is incompatible with \"scgi_cache\"";
1827     }
1828     #endif
1829
1830     scf->upstream.store = 1;
1831
1832     if (ngx_strcmp(value[1].data, "on") == 0) {
1833         return NGX_CONF_OK;
1834     }
1835
1836     /* include the terminating '\0' into script */
1837     value[1].len++;
1838
1839     ngx_memzero(&sc, sizeof(ngx_http_script_compile_t));
1840
1841     sc.cf = cf;
1842     sc.source = &value[1];
1843     sc.lengths = &scf->upstream.store_lengths;
1844     sc.values = &scf->upstream.store_values;
1845     sc.variables = ngx_http_script_variables_count(&value[1]);
1846     sc.complete_lengths = 1;
1847     sc.complete_values = 1;
1848
1849     if (ngx_http_script_compile(&sc) != NGX_OK) {
1850         return NGX_CONF_ERROR;
1851     }
1852
1853     return NGX_CONF_OK;
1854 }
1855
1856
1857
1858 #if (NGX_HTTP_CACHE)
1859
1860 static char *
1861 ngx_http_scgi_cache(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
1862 {
1863     ngx_http_scgi_loc_conf_t *scf = conf;
1864
1865     ngx_str_t          *value;
1866     ngx_http_complex_value_t  cv;
1867     ngx_http_compile_complex_value_t  ccv;
1868
1869     value = cf->args->elts;
1870
1871     if (scf->upstream.cache != NGX_CONF_UNSET) {
1872         return "is duplicate";
1873     }
1874
1875     if (ngx_strcmp(value[1].data, "off") == 0) {
1876         scf->upstream.cache = 0;
1877         return NGX_CONF_OK;
1878     }
1879
1880     if (scf->upstream.store > 0) {
1881         return "is incompatible with \"scgi_store\"";
1882     }
1883
1884     scf->upstream.cache = 1;
1885
1886     ngx_memzero(&ccv, sizeof(ngx_http_compile_complex_value_t));
1887
1888     ccv.cf = cf;
1889     ccv.value = &value[1];
1890     ccv.complex_value = &cv;
1891
1892     if (ngx_http_compile_complex_value(&ccv) != NGX_OK) {
1893         return NGX_CONF_ERROR;
1894     }

```

```

1895
1896     if (cv.lengths != NULL) {
1897
1898         scf->upstream.cache_value = ngx_palloc(cf->pool,
1899                                             sizeof(ngx_http_complex_value_t));
1900         if (scf->upstream.cache_value == NULL) {
1901             return NGX_CONF_ERROR;
1902         }
1903
1904         *scf->upstream.cache_value = cv;
1905
1906         return NGX_CONF_OK;
1907     }
1908
1909     scf->upstream.cache_zone = ngx_shared_memory_add(cf, &value[1], 0,
1910                                                    &ngx_http_scgi_module);
1911     if (scf->upstream.cache_zone == NULL) {
1912         return NGX_CONF_ERROR;
1913     }
1914
1915     return NGX_CONF_OK;
1916 }
1917
1918
1919 static char *
1920 ngx_http_scgi_cache_key(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
1921 {
1922     ngx_http_scgi_loc_conf_t *scf = conf;
1923
1924     ngx_str_t          *value;
1925     ngx_http_compile_complex_value_t ccv;
1926
1927     value = cf->args->elts;
1928
1929     if (scf->cache_key.value.data) {
1930         return "is duplicate";
1931     }
1932
1933     ngx_memzero(&ccv, sizeof(ngx_http_compile_complex_value_t));
1934
1935     ccv.cf = cf;
1936     ccv.value = &value[1];
1937     ccv.complex_value = &scf->cache_key;
1938
1939     if (ngx_http_compile_complex_value(&ccv) != NGX_OK) {
1940         return NGX_CONF_ERROR;
1941     }
1942
1943     return NGX_CONF_OK;
1944 }
1945
1946 #endif

```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_secure\_link\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_secure\\_link\\_commands](#)
- [ngx\\_http\\_secure\\_link\\_expires\\_name](#)
- [ngx\\_http\\_secure\\_link\\_module](#)
- [ngx\\_http\\_secure\\_link\\_module\\_ctx](#)
- [ngx\\_http\\_secure\\_link\\_name](#)

## Data types defined

- [ngx\\_http\\_secure\\_link\\_conf\\_t](#)
- [ngx\\_http\\_secure\\_link\\_ctx\\_t](#)

## Functions defined

- [ngx\\_http\\_secure\\_link\\_add\\_variables](#)
- [ngx\\_http\\_secure\\_link\\_create\\_conf](#)
- [ngx\\_http\\_secure\\_link\\_expires\\_variable](#)
- [ngx\\_http\\_secure\\_link\\_merge\\_conf](#)
- [ngx\\_http\\_secure\\_link\\_old\\_variable](#)
- [ngx\\_http\\_secure\\_link\\_variable](#)

## Source code

```
1
2  /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8  #include <ngx_config.h>
9  #include <ngx_core.h>
10 #include <ngx_http.h>
11 #include <ngx_md5.h>
12
13
14 typedef struct {
15     ngx_http_complex_value_t *variable;
16     ngx_http_complex_value_t *md5;
17     ngx_str_t secret;
18 } ngx_http_secure_link_conf_t;
19
20
21 typedef struct {
22     ngx_str_t expires;
23 } ngx_http_secure_link_ctx_t;
24
25
26 static ngx_int_t ngx_http_secure_link_old_variable(ngx_http_request_t *r,
27     ngx_http_secure_link_conf_t *conf, ngx_http_variable_value_t *v,
28     uintptr_t data);
```

```

29 static ngx_int_t ngx_http_secure_link_expires_variable(ngx_http_request_t *r,
30 ngx_http_variable_value_t *v, uintptr_t data);
31 static void *ngx_http_secure_link_create_conf(ngx_conf_t *cf);
32 static char *ngx_http_secure_link_merge_conf(ngx_conf_t *cf, void *parent,
33 void *child);
34 static ngx_int_t ngx_http_secure_link_add_variables(ngx_conf_t *cf);
35
36
37 static ngx_command_t ngx_http_secure_link_commands[] = {
38
39     { ngx_string("secure_link"),
40       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
41       ngx_http_set_complex_value_slot,
42       NGX_HTTP_LOC_CONF_OFFSET,
43       offsetof(ngx_http_secure_link_conf_t, variable),
44       NULL },
45
46     { ngx_string("secure_link_md5"),
47       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
48       ngx_http_set_complex_value_slot,
49       NGX_HTTP_LOC_CONF_OFFSET,
50       offsetof(ngx_http_secure_link_conf_t, md5),
51       NULL },
52
53     { ngx_string("secure_link_secret"),
54       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
55       ngx_conf_set_str_slot,
56       NGX_HTTP_LOC_CONF_OFFSET,
57       offsetof(ngx_http_secure_link_conf_t, secret),
58       NULL },
59
60     ngx_null_command
61 };
62
63
64 static ngx_http_module_t ngx_http_secure_link_module_ctx = {
65     ngx_http_secure_link_add_variables, /* preconfiguration */
66     NULL, /* postconfiguration */
67
68     NULL, /* create main configuration */
69     NULL, /* init main configuration */
70
71     NULL, /* create server configuration */
72     NULL, /* merge server configuration */
73
74     ngx_http_secure_link_create_conf, /* create location configuration */
75     ngx_http_secure_link_merge_conf /* merge location configuration */
76 };
77
78
79 ngx_module_t ngx_http_secure_link_module = {
80     NGX_MODULE_V1,
81     &ngx_http_secure_link_module_ctx, /* module context */
82     ngx_http_secure_link_commands, /* module directives */
83     NGX_HTTP_MODULE, /* module type */
84     NULL, /* init master */
85     NULL, /* init module */
86     NULL, /* init process */
87     NULL, /* init thread */
88     NULL, /* exit thread */
89     NULL, /* exit process */
90     NULL, /* exit master */
91     NGX_MODULE_V1_PADDING
92 };
93
94
95 static ngx_str_t ngx_http_secure_link_name = ngx_string("secure_link");
96 static ngx_str_t ngx_http_secure_link_expires_name =
97     ngx_string("secure_link_expires");
98
99
100 static ngx_int_t
101 ngx_http_secure_link_variable(ngx_http_request_t *r,
102 ngx_http_variable_value_t *v, uintptr_t data)
103 {
104     u_char *p, *last;

```

```

105     ngx_str_t          val, hash;
106     time_t            expires;
107     ngx_md5_t         md5;
108     ngx_http_secure_link_ctx_t *ctx;
109     ngx_http_secure_link_conf_t *conf;
110     u_char            hash_buf[16], md5_buf[16];
111
112     conf = ngx_http_get_module_loc_conf(r, ngx_http_secure_link_module);
113
114     if (conf->secret.data) {
115         return ngx_http_secure_link_old_variable(r, conf, v, data);
116     }
117
118     if (conf->variable == NULL || conf->md5 == NULL) {
119         goto not_found;
120     }
121
122     if (ngx_http_complex_value(r, conf->variable, &val) != NGX_OK) {
123         return NGX_ERROR;
124     }
125
126     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
127                  "secure link: \"%V\"", &val);
128
129     last = val.data + val.len;
130
131     p = ngx_strlchr(val.data, last, ',');
132     expires = 0;
133
134     if (p) {
135         val.len = p++ - val.data;
136
137         expires = ngx_atotm(p, last - p);
138         if (expires <= 0) {
139             goto not_found;
140         }
141
142         ctx = ngx_palloc(r->pool, sizeof(ngx_http_secure_link_ctx_t));
143         if (ctx == NULL) {
144             return NGX_ERROR;
145         }
146
147         ngx_http_set_ctx(r, ctx, ngx_http_secure_link_module);
148
149         ctx->expires.len = last - p;
150         ctx->expires.data = p;
151     }
152
153     if (val.len > 24) {
154         goto not_found;
155     }
156
157     hash.len = 16;
158     hash.data = hash_buf;
159
160     if (ngx_decode_base64url(&hash, &val) != NGX_OK) {
161         goto not_found;
162     }
163
164     if (hash.len != 16) {
165         goto not_found;
166     }
167
168     if (ngx_http_complex_value(r, conf->md5, &val) != NGX_OK) {
169         return NGX_ERROR;
170     }
171
172     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
173                  "secure link md5: \"%V\"", &val);
174
175     ngx_md5_init(&md5);
176     ngx_md5_update(&md5, val.data, val.len);
177     ngx_md5_final(md5_buf, &md5);
178
179     if (ngx_memcmp(hash_buf, md5_buf, 16) != 0) {
180         goto not_found;

```

```

181     }
182
183     v->data = (u_char *) ((expires && expires < ngx\_time()) ? "0" : "1");
184     v->len = 1;
185     v->valid = 1;
186     v->no_cacheable = 0;
187     v->not_found = 0;
188
189     return NGX\_OK;
190
191 not_found:
192
193     v->not_found = 1;
194
195     return NGX\_OK;
196 }
197
198
199 static ngx\_int\_t
200 ngx\_http\_secure\_link\_old\_variable(ngx\_http\_request\_t *r,
201 ngx\_http\_secure\_link\_conf\_t *conf, ngx\_http\_variable\_value\_t *v,
202 u\_intptr\_t data)
203 {
204     u\_char      *p, *start, *end, *last;
205     size\_t      len;
206     ngx\_int\_t   n;
207     ngx\_uint\_t  i;
208     ngx\_md5\_t   md5;
209     u\_char      hash[16];
210
211     p = &r->unparsed_uri.data[1];
212     last = r->unparsed_uri.data + r->unparsed_uri.len;
213
214     while (p < last) {
215         if (*p++ == '/') {
216             start = p;
217             goto md5_start;
218         }
219     }
220
221     goto not_found;
222
223 md5_start:
224
225     while (p < last) {
226         if (*p++ == '/') {
227             end = p - 1;
228             goto url_start;
229         }
230     }
231
232     goto not_found;
233
234 url_start:
235
236     len = last - p;
237
238     if (end - start != 32 || len == 0) {
239         goto not_found;
240     }
241
242     ngx\_md5\_init(&md5);
243     ngx\_md5\_update(&md5, p, len);
244     ngx\_md5\_update(&md5, conf->secret.data, conf->secret.len);
245     ngx\_md5\_final(hash, &md5);
246
247     for (i = 0; i < 16; i++) {
248         n = ngx\_hextoi(&start[2 * i], 2);
249         if (n == NGX\_ERROR || n != hash[i]) {
250             goto not_found;
251         }
252     }
253
254     v->len = len;
255     v->valid = 1;
256     v->no_cacheable = 0;

```



```

257     v->not_found = 0;
258     v->data = p;
259
260     return NGX_OK;
261
262 not_found:
263
264     v->not_found = 1;
265
266     return NGX_OK;
267 }
268
269
270 static ngx_int_t
271 ngx_http_secure_link_expires_variable(ngx_http_request_t *r,
272     ngx_http_variable_value_t *v, uintptr_t data)
273 {
274     ngx_http_secure_link_ctx_t *ctx;
275
276     ctx = ngx_http_get_module_ctx(r, ngx_http_secure_link_module);
277
278     if (ctx) {
279         v->len = ctx->expires.len;
280         v->valid = 1;
281         v->no_cacheable = 0;
282         v->not_found = 0;
283         v->data = ctx->expires.data;
284
285     } else {
286         v->not_found = 1;
287     }
288
289     return NGX_OK;
290 }
291
292
293 static void *
294 ngx_http_secure_link_create_conf(ngx_conf_t *cf)
295 {
296     ngx_http_secure_link_conf_t *conf;
297
298     conf = ngx_palloc(cf->pool, sizeof(ngx_http_secure_link_conf_t));
299     if (conf == NULL) {
300         return NULL;
301     }
302
303     /*
304      * set by ngx_palloc():
305      *
306      *     conf->variable = NULL;
307      *     conf->md5 = NULL;
308      *     conf->secret = { 0, NULL };
309      */
310
311     return conf;
312 }
313
314
315 static char *
316 ngx_http_secure_link_merge_conf(ngx_conf_t *cf, void *parent, void *child)
317 {
318     ngx_http_secure_link_conf_t *prev = parent;
319     ngx_http_secure_link_conf_t *conf = child;
320
321     if (conf->secret.data) {
322         if (conf->variable || conf->md5) {
323             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
324                 "\"secure_link_secret\" cannot be mixed with "
325                 "\"secure_link\" and \"secure_link_md5\"");
326             return NGX_CONF_ERROR;
327         }
328
329         return NGX_CONF_OK;
330     }
331
332     if (conf->variable == NULL) {

```

```
333     conf->variable = prev->variable;
334 }
335
336 if (conf->md5 == NULL) {
337     conf->md5 = prev->md5;
338 }
339
340 if (conf->variable == NULL && conf->md5 == NULL) {
341     conf->secret = prev->secret;
342 }
343
344 return NGX\_CONF\_OK;
345 }
346
347
348 static ngx\_int\_t
349 ngx\_http\_secure\_link\_add\_variables(ngx\_conf\_t *cf)
350 {
351     ngx\_http\_variable\_t *var;
352
353     var = ngx\_http\_add\_variable(cf, &ngx\_http\_secure\_link\_name, 0);
354     if (var == NULL) {
355         return NGX\_ERROR;
356     }
357
358     var->get_handler = ngx\_http\_secure\_link\_variable;
359
360     var = ngx\_http\_add\_variable(cf, &ngx\_http\_secure\_link\_expires\_name, 0);
361     if (var == NULL) {
362         return NGX\_ERROR;
363     }
364
365     var->get_handler = ngx\_http\_secure\_link\_expires\_variable;
366
367     return NGX\_OK;
368 }
```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_split\_clients\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_split\\_clients\\_commands](#)
- [ngx\\_http\\_split\\_clients\\_module](#)
- [ngx\\_http\\_split\\_clients\\_module\\_ctx](#)

## Data types defined

- [ngx\\_http\\_split\\_clients\\_ctx\\_t](#)
- [ngx\\_http\\_split\\_clients\\_part\\_t](#)

## Functions defined

- [ngx\\_conf\\_split\\_clients\\_block](#)
- [ngx\\_http\\_split\\_clients](#)
- [ngx\\_http\\_split\\_clients\\_variable](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 typedef struct {
14     uint32_t                percent;
15     ngx_http_variable_value_t value;
16 } ngx_http_split_clients_part_t;
17
18
19 typedef struct {
20     ngx_http_complex_value_t value;
21     ngx_array_t              parts;
22 } ngx_http_split_clients_ctx_t;
23
24
25 static char *ngx_conf_split_clients_block(ngx_conf_t *cf, ngx_command_t *cmd,
26     void *conf);
27 static char *ngx_http_split_clients(ngx_conf_t *cf, ngx_command_t *dummy,
28     void *conf);
29
30 static ngx_command_t ngx_http_split_clients_commands[] = {
31
32     { ngx_string("split_clients"),
33       NGX_HTTP_MAIN_CONF|NGX_CONF_BLOCK|NGX_CONF_TAKE2,
34       ngx_conf_split_clients_block,
35       NGX_HTTP_MAIN_CONF_OFFSET,
36       0,
37       NULL },
38
39     ngx_null_command
```

```

40 };
41
42
43 static ngx_http_module_t ngx_http_split_clients_module_ctx = {
44     NULL, /* preconfiguration */
45     NULL, /* postconfiguration */
46
47     NULL, /* create main configuration */
48     NULL, /* init main configuration */
49
50     NULL, /* create server configuration */
51     NULL, /* merge server configuration */
52
53     NULL, /* create location configuration */
54     NULL, /* merge location configuration */
55 };
56
57
58 ngx_module_t ngx_http_split_clients_module = {
59     NGX_MODULE_V1,
60     &ngx_http_split_clients_module_ctx, /* module context */
61     ngx_http_split_clients_commands, /* module directives */
62     NGX_HTTP_MODULE, /* module type */
63     NULL, /* init master */
64     NULL, /* init module */
65     NULL, /* init process */
66     NULL, /* init thread */
67     NULL, /* exit thread */
68     NULL, /* exit process */
69     NULL, /* exit master */
70     NGX_MODULE_V1_PADDING
71 };
72
73
74 static ngx_int_t
75 ngx_http_split_clients_variable(ngx_http_request_t *r,
76     ngx_http_variable_value_t *v, uintptr_t data)
77 {
78     ngx_http_split_clients_ctx_t *ctx = (ngx_http_split_clients_ctx_t *) data;
79
80     uint32_t hash;
81     ngx_str_t val;
82     ngx_uint_t i;
83     ngx_http_split_clients_part_t *part;
84
85     *v = ngx_http_variable_null_value;
86
87     if (ngx_http_complex_value(r, &ctx->value, &val) != NGX_OK) {
88         return NGX_OK;
89     }
90
91     hash = ngx_murmur_hash2(val.data, val.len);
92
93     part = ctx->parts.elts;
94
95     for (i = 0; i < ctx->parts.nelts; i++) {
96
97         ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
98             "http split: %uD %uD", hash, part[i].percent);
99
100         if (hash < part[i].percent || part[i].percent == 0) {
101             *v = part[i].value;
102             return NGX_OK;
103         }
104     }
105
106     return NGX_OK;
107 }
108
109
110 static char *
111 ngx_conf_split_clients_block(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
112 {
113     char *rv;
114     uint32_t sum, last;
115     ngx_str_t *value, name;

```

```

116     ngx_uint_t           i;
117     ngx_conf_t          save;
118     ngx_http_variable_t *var;
119     ngx_http_split_clients_ctx_t *ctx;
120     ngx_http_split_clients_part_t *part;
121     ngx_http_compile_complex_value_t ccv;
122
123     ctx = ngx_palloc(cf->pool, sizeof(ngx_http_split_clients_ctx_t));
124     if (ctx == NULL) {
125         return NGX_CONF_ERROR;
126     }
127
128     value = cf->args->elts;
129
130     ngx_memzero(&ccv, sizeof(ngx_http_compile_complex_value_t));
131
132     ccv.cf = cf;
133     ccv.value = &value[1];
134     ccv.complex_value = &ctx->value;
135
136     if (ngx_http_compile_complex_value(&ccv) != NGX_OK) {
137         return NGX_CONF_ERROR;
138     }
139
140     name = value[2];
141
142     if (name.data[0] != '$') {
143         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
144             "invalid variable name \"%V\"", &name);
145         return NGX_CONF_ERROR;
146     }
147
148     name.len--;
149     name.data++;
150
151     var = ngx_http_add_variable(cf, &name, NGX_HTTP_VAR_CHANGEABLE);
152     if (var == NULL) {
153         return NGX_CONF_ERROR;
154     }
155
156     var->get_handler = ngx_http_split_clients_variable;
157     var->data = (uintptr_t) ctx;
158
159     if (ngx_array_init(&ctx->parts, cf->pool, 2,
160         sizeof(ngx_http_split_clients_part_t))
161         != NGX_OK)
162     {
163         return NGX_CONF_ERROR;
164     }
165
166     save = *cf;
167     cf->ctx = ctx;
168     cf->handler = ngx_http_split_clients;
169     cf->handler_conf = conf;
170
171     rv = ngx_conf_parse(cf, NULL);
172
173     *cf = save;
174
175     if (rv != NGX_CONF_OK) {
176         return rv;
177     }
178
179     sum = 0;
180     last = 0;
181     part = ctx->parts.elts;
182
183     for (i = 0; i < ctx->parts.nelts; i++) {
184         sum = part[i].percent ? sum + part[i].percent : 10000;
185         if (sum > 10000) {
186             ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
187                 "percent total is greater than 100%");
188             return NGX_CONF_ERROR;
189         }
190
191         if (part[i].percent) {

```

```

192         last += part[i].percent * (uint64_t) 0xffffffff / 10000;
193         part[i].percent = last;
194     }
195 }
196
197 return rv;
198 }
199
200
201 static char *
202 ngx_http_split_clients(ngx_conf_t *cf, ngx_command_t *dummy, void *conf)
203 {
204     ngx_int_t          n;
205     ngx_str_t          *value;
206     ngx_http_split_clients_ctx_t *ctx;
207     ngx_http_split_clients_part_t *part;
208
209     ctx = cf->ctx;
210     value = cf->args->elts;
211
212     part = ngx_array_push(&ctx->parts);
213     if (part == NULL) {
214         return NGX_CONF_ERROR;
215     }
216
217     if (value[0].len == 1 && value[0].data[0] == '*') {
218         part->percent = 0;
219
220     } else {
221         if (value[0].len == 0 || value[0].data[value[0].len - 1] != '%') {
222             goto invalid;
223         }
224
225         n = ngx_atofp(value[0].data, value[0].len - 1, 2);
226         if (n == NGX_ERROR || n == 0) {
227             goto invalid;
228         }
229
230         part->percent = (uint32_t) n;
231     }
232
233     part->value.len = value[1].len;
234     part->value.valid = 1;
235     part->value.no_cacheable = 0;
236     part->value.not_found = 0;
237     part->value.data = value[1].data;
238
239     return NGX_CONF_OK;
240
241 invalid:
242
243     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
244         "invalid percent value \"%V\"", &value[0]);
245     return NGX_CONF_ERROR;
246 }

```

[One Level Up](#)

[Top Level](#)

# src/core/nginx\_murmurhash.c - nginx-1.7.10

## Functions defined

- [ngx\\_murmur\\_hash2](#)

## Source code

```
1
2  /*
3  * Copyright (C) Austin Appleby
4  */
5
6
7  #include <ngx_config.h>
8  #include <ngx_core.h>
9
10
11  uint32_t
12  ngx_murmur_hash2(u_char *data, size_t len)
13  {
14      uint32_t  h, k;
15
16      h = 0 ^ len;
17
18      while (len >= 4) {
19          k = data[0];
20          k |= data[1] << 8;
21          k |= data[2] << 16;
22          k |= data[3] << 24;
23
24          k *= 0x5bd1e995;
25          k ^= k >> 24;
26          k *= 0x5bd1e995;
27
28          h *= 0x5bd1e995;
29          h ^= k;
30
31          data += 4;
32          len -= 4;
33      }
34
35      switch (len) {
36      case 3:
37          h ^= data[2] << 16;
38      case 2:
39          h ^= data[1] << 8;
40      case 1:
41          h ^= data[0];
42          h *= 0x5bd1e995;
43      }
44
45      h ^= h >> 13;
46      h *= 0x5bd1e995;
47      h ^= h >> 15;
48
49      return h;
50 }
```

# src/http/modules/nginx\_http\_static\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_static\\_module](#)
- [ngx\\_http\\_static\\_module\\_ctx](#)

## Functions defined

- [ngx\\_http\\_static\\_handler](#)
- [ngx\\_http\\_static\\_init](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 static ngx_int_t ngx_http_static_handler(ngx_http_request_t *r);
14 static ngx_int_t ngx_http_static_init(ngx_conf_t *cf);
15
16
17 ngx_http_module_t ngx_http_static_module_ctx = {
18     NULL,                               /* preconfiguration */
19     ngx_http_static_init,               /* postconfiguration */
20
21     NULL,                               /* create main configuration */
22     NULL,                               /* init main configuration */
23
24     NULL,                               /* create server configuration */
25     NULL,                               /* merge server configuration */
26
27     NULL,                               /* create location configuration */
28     NULL,                               /* merge location configuration */
29 };
30
31
32 ngx_module_t ngx_http_static_module = {
33     NGX_MODULE_V1,
34     &ngx_http_static_module_ctx,       /* module context */
35     NULL,                               /* module directives */
36     NGX_HTTP_MODULE,                   /* module type */
37     NULL,                               /* init master */
38     NULL,                               /* init module */
39     NULL,                               /* init process */
40     NULL,                               /* init thread */
41     NULL,                               /* exit thread */
42     NULL,                               /* exit process */
43     NULL,                               /* exit master */
44     NGX_MODULE_V1_PADDING
45 };
46
47
48 static ngx_int_t
49 ngx_http_static_handler(ngx_http_request_t *r)
50 {
51     u_char          *last, *location;
52     size_t          root, len;
```



```

53     ngx_str_t      path;
54     ngx_int_t      rc;
55     ngx_uint_t     level;
56     ngx_log_t      *log;
57     ngx_buf_t      *b;
58     ngx_chain_t    out;
59     ngx_open_file_info_t of;
60     ngx_http_core_loc_conf_t *clcf;
61
62     if (!(r->method & (NGX_HTTP_GET|NGX_HTTP_HEAD|NGX_HTTP_POST))) {
63         return NGX_HTTP_NOT_ALLOWED;
64     }
65
66     if (r->uri.data[r->uri.len - 1] == '/') {
67         return NGX_DECLINED;
68     }
69
70     log = r->connection->log;
71
72     /*
73      * ngx_http_map_uri_to_path() allocates memory for terminating '\0'
74      * so we do not need to reserve memory for '/' for possible redirect
75      */
76
77     last = ngx_http_map_uri_to_path(r, &path, &root, 0);
78     if (last == NULL) {
79         return NGX_HTTP_INTERNAL_SERVER_ERROR;
80     }
81
82     path.len = last - path.data;
83
84     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, log, 0,
85                  "http filename: \"%s\"", path.data);
86
87     clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
88
89     ngx_memzero(&of, sizeof(ngx_open_file_info_t));
90
91     of.read_ahead = clcf->read_ahead;
92     of.directio = clcf->directio;
93     of.valid = clcf->open_file_cache_valid;
94     of.min_uses = clcf->open_file_cache_min_uses;
95     of.errors = clcf->open_file_cache_errors;
96     of.events = clcf->open_file_cache_events;
97
98     if (ngx_http_set_disable_symlinks(r, clcf, &path, &of) != NGX_OK) {
99         return NGX_HTTP_INTERNAL_SERVER_ERROR;
100    }
101
102    if (ngx_open_cached_file(clcf->open_file_cache, &path, &of, r->pool)
103        != NGX_OK)
104    {
105        switch (of.err) {
106
107            case 0:
108                return NGX_HTTP_INTERNAL_SERVER_ERROR;
109
110            case NGX_ENOENT:
111            case NGX_ENOTDIR:
112            case NGX_ENAMETOOLONG:
113
114                level = NGX_LOG_ERR;
115                rc = NGX_HTTP_NOT_FOUND;
116                break;
117
118            case NGX_EACCES:
119                #if (NGX_HAVE_OPENAT)
120                case NGX_EMLINK:
121                case NGX_ELOOP:
122                #endif
123
124                level = NGX_LOG_ERR;
125                rc = NGX_HTTP_FORBIDDEN;
126                break;
127
128            default:

```

```

129         level = NGX\_LOG\_CRIT;
130         rc = NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
131         break;
132     }
133
134     if (rc != NGX\_HTTP\_NOT\_FOUND || clcf->log_not_found) {
135         ngx\_log\_error(level, log, of.err,
136             "%s \"%s\" failed", of.failed, path.data);
137     }
138
139     return rc;
140 }
141
142 r->root_tested = !r->error_page;
143
144 ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, log, 0, "http static fd: %d", of.fd);
145
146 if (of.is_dir) {
147
148     ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, log, 0, "http dir");
149
150     ngx\_http\_clear\_location(r);
151
152     r->headers_out.location = ngx\_palloc(r->pool, sizeof(ngx\_table\_elt\_t));
153     if (r->headers_out.location == NULL) {
154         return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
155     }
156
157     len = r->uri.len + 1;
158
159     if (!clcf->alias && clcf->root_lengths == NULL && r->args.len == 0) {
160         location = path.data + clcf->root.len;
161
162         *last = '/';
163
164     } else {
165         if (r->args.len) {
166             len += r->args.len + 1;
167         }
168
169         location = ngx\_pnalloc(r->pool, len);
170         if (location == NULL) {
171             return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
172         }
173
174         last = ngx\_copy(location, r->uri.data, r->uri.len);
175
176         *last = '/';
177
178         if (r->args.len) {
179             *++last = '?';
180             ngx\_memcpy(++last, r->args.data, r->args.len);
181         }
182     }
183
184     /*
185     * we do not need to set the r->headers_out.location->hash and
186     * r->headers_out.location->key fields
187     */
188
189     r->headers_out.location->value.len = len;
190     r->headers_out.location->value.data = location;
191
192     return NGX\_HTTP\_MOVED\_PERMANENTLY;
193 }
194
195 #if !(NGX_WIN32) /* the not regular files are probably Unix specific */
196
197 if (!of.is_file) {
198     ngx\_log\_error(NGX\_LOG\_CRIT, log, 0,
199         "\"%s\" is not a regular file", path.data);
200
201     return NGX\_HTTP\_NOT\_FOUND;
202 }
203
204

```

```

205 #endif
206
207     if (r->method & NGX\_HTTP\_POST) {
208         return NGX\_HTTP\_NOT\_ALLOWED;
209     }
210
211     rc = ngx\_http\_discard\_request\_body(r);
212
213     if (rc != NGX\_OK) {
214         return rc;
215     }
216
217     log->action = "sending response to client";
218
219     r->headers_out.status = NGX\_HTTP\_OK;
220     r->headers_out.content_length_n = of.size;
221     r->headers_out.last_modified_time = of.mtime;
222
223     if (ngx\_http\_set\_etag(r) != NGX\_OK) {
224         return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
225     }
226
227     if (ngx\_http\_set\_content\_type(r) != NGX\_OK) {
228         return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
229     }
230
231     if (r != r->main && of.size == 0) {
232         return ngx\_http\_send\_header(r);
233     }
234
235     r->allow_ranges = 1;
236
237     /* we need to allocate all before the header would be sent */
238
239     b = ngx\_palloc(r->pool, sizeof(ngx\_buf\_t));
240     if (b == NULL) {
241         return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
242     }
243
244     b->file = ngx\_palloc(r->pool, sizeof(ngx\_file\_t));
245     if (b->file == NULL) {
246         return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;
247     }
248
249     rc = ngx\_http\_send\_header(r);
250
251     if (rc == NGX\_ERROR || rc > NGX\_OK || r->header_only) {
252         return rc;
253     }
254
255     b->file_pos = 0;
256     b->file_last = of.size;
257
258     b->in_file = b->file_last ? 1: 0;
259     b->last_buf = (r == r->main) ? 1: 0;
260     b->last_in_chain = 1;
261
262     b->file->fd = of.fd;
263     b->file->name = path;
264     b->file->log = log;
265     b->file->directio = of.is_directio;
266
267     out.buf = b;
268     out.next = NULL;
269
270     return ngx\_http\_output\_filter(r, &out);
271 }
272
273
274 static ngx\_int\_t
275 ngx\_http\_static\_init(ngx\_conf\_t *cf)
276 {
277     ngx\_http\_handler\_pt *h;
278     ngx\_http\_core\_main\_conf\_t *cmcf;
279
280     cmcf = ngx\_http\_conf\_get\_module\_main\_conf(cf, ngx\_http\_core\_module);

```

```
281 h = ngx_array_push(&cmcf->phases[NGX_HTTP_CONTENT_PHASE].handlers);
282 if (h == NULL) {
283     return NGX_ERROR;
284 }
285 *h = ngx_http_static_handler;
286 return NGX_OK;
287 }
288
289
290 }
```

[One Level Up](#)

[Top Level](#)



```

44     NULL,                                     /* create location configuration */
45     NULL,                                     /* merge location configuration */
46 };
47
48
49 ngx_module_t ngx_http_stub_status_module = {
50     NGX_MODULE_V1,
51     &ngx_http_stub_status_module_ctx,        /* module context */
52     ngx_http_status_commands,                /* module directives */
53     NGX_HTTP_MODULE,                         /* module type */
54     NULL,                                    /* init master */
55     NULL,                                    /* init module */
56     NULL,                                    /* init process */
57     NULL,                                    /* init thread */
58     NULL,                                    /* exit thread */
59     NULL,                                    /* exit process */
60     NULL,                                    /* exit master */
61     NGX_MODULE_V1_PADDING
62 };
63
64
65 static ngx_http_variable_t ngx_http_stub_status_vars[] = {
66
67     { ngx_string("connections_active"), NULL, ngx_http_stub_status_variable,
68       0, NGX_HTTP_VAR_NOCACHEABLE, 0 },
69
70     { ngx_string("connections_reading"), NULL, ngx_http_stub_status_variable,
71       1, NGX_HTTP_VAR_NOCACHEABLE, 0 },
72
73     { ngx_string("connections_writing"), NULL, ngx_http_stub_status_variable,
74       2, NGX_HTTP_VAR_NOCACHEABLE, 0 },
75
76     { ngx_string("connections_waiting"), NULL, ngx_http_stub_status_variable,
77       3, NGX_HTTP_VAR_NOCACHEABLE, 0 },
78
79     { ngx_null_string, NULL, NULL, 0, 0, 0 }
80 };
81
82
83 static ngx_int_t
84 ngx_http_stub_status_handler(ngx_http_request_t *r)
85 {
86     size_t      size;
87     ngx_int_t   rc;
88     ngx_buf_t   *b;
89     ngx_chain_t out;
90     ngx_atomic_int_t  ap, hn, ac, rq, rd, wr, wa;
91
92     if (r->method != NGX_HTTP_GET && r->method != NGX_HTTP_HEAD) {
93         return NGX_HTTP_NOT_ALLOWED;
94     }
95
96     rc = ngx_http_discard_request_body(r);
97
98     if (rc != NGX_OK) {
99         return rc;
100    }
101
102     r->headers_out.content_type_len = sizeof("text/plain") - 1;
103     ngx_str_set(&r->headers_out.content_type, "text/plain");
104     r->headers_out.content_type_lowercase = NULL;
105
106     if (r->method == NGX_HTTP_HEAD) {
107         r->headers_out.status = NGX_HTTP_OK;
108
109         rc = ngx_http_send_header(r);
110
111         if (rc == NGX_ERROR || rc > NGX_OK || r->header_only) {
112             return rc;
113         }
114     }
115
116     size = sizeof("Active connections: \n") + NGX_ATOMIC_T_LEN
117           + sizeof("server accepts handled requests\n") - 1
118           + 6 + 3 * NGX_ATOMIC_T_LEN
119           + sizeof("Reading: Writing: Waiting: \n") + 3 * NGX_ATOMIC_T_LEN;

```

```

120     b = ngx_create_temp_buf(r->pool, size);
121     if (b == NULL) {
122         return NGX_HTTP_INTERNAL_SERVER_ERROR;
123     }
124
125     out.buf = b;
126     out.next = NULL;
127
128     ap = *ngx_stat_accepted;
129     hn = *ngx_stat_handled;
130     ac = *ngx_stat_active;
131     rq = *ngx_stat_requests;
132     rd = *ngx_stat_reading;
133     wr = *ngx_stat_writing;
134     wa = *ngx_stat_waiting;
135
136     b->last = ngx_sprintf(b->last, "Active connections: %uA \n", ac);
137
138     b->last = ngx_cpymem(b->last, "server accepts handled requests\n",
139         sizeof("server accepts handled requests\n") - 1);
140
141     b->last = ngx_sprintf(b->last, " %uA %uA %uA \n", ap, hn, rq);
142
143     b->last = ngx_sprintf(b->last, "Reading: %uA Writing: %uA Waiting: %uA \n",
144         rd, wr, wa);
145
146     r->headers_out.status = NGX_HTTP_OK;
147     r->headers_out.content_length_n = b->last - b->pos;
148
149     b->last_buf = (r == r->main) ? 1 : 0;
150     b->last_in_chain = 1;
151
152     rc = ngx_http_send_header(r);
153
154     if (rc == NGX_ERROR || rc > NGX_OK || r->header_only) {
155         return rc;
156     }
157
158     return ngx_http_output_filter(r, &out);
159 }
160
161
162
163 static ngx_int_t
164 ngx_http_stub_status_variable(ngx_http_request_t *r,
165     ngx_http_variable_value_t *v, uintptr_t data)
166 {
167     u_char          *p;
168     ngx_atomic_int_t value;
169
170     p = ngx_pnalloc(r->pool, NGX_ATOMIC_T_LEN);
171     if (p == NULL) {
172         return NGX_ERROR;
173     }
174
175     switch (data) {
176     case 0:
177         value = *ngx_stat_active;
178         break;
179
180     case 1:
181         value = *ngx_stat_reading;
182         break;
183
184     case 2:
185         value = *ngx_stat_writing;
186         break;
187
188     case 3:
189         value = *ngx_stat_waiting;
190         break;
191
192     /* suppress warning */
193     default:
194         value = 0;
195         break;

```

```

196     }
197
198     v->len = ngx_sprintf(p, "%uA", value) - p;
199     v->valid = 1;
200     v->no_cacheable = 0;
201     v->not_found = 0;
202     v->data = p;
203
204     return NGX_OK;
205 }
206
207
208 static ngx_int_t
209 ngx_http_stub_status_add_variables(ngx_conf_t *cf)
210 {
211     ngx_http_variable_t *var, *v;
212
213     for (v = ngx_http_stub_status_vars; v->name.len; v++) {
214         var = ngx_http_add_variable(cf, &v->name, v->flags);
215         if (var == NULL) {
216             return NGX_ERROR;
217         }
218
219         var->get_handler = v->get_handler;
220         var->data = v->data;
221     }
222
223     return NGX_OK;
224 }
225
226
227 static char *
228 ngx_http_set_stub_status(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
229 {
230     ngx_http_core_loc_conf_t *clcf;
231
232     clcf = ngx_http_conf_get_module_loc_conf(cf, ngx_http_core_module);
233     clcf->handler = ngx_http_stub_status_handler;
234
235     return NGX_CONF_OK;
236 }

```

[One Level Up](#)

[Top Level](#)



# src/http/modules/nginx\_http\_sub\_filter\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_next\\_body\\_filter](#)
- [ngx\\_http\\_next\\_header\\_filter](#)
- [ngx\\_http\\_sub\\_filter\\_commands](#)
- [ngx\\_http\\_sub\\_filter\\_module](#)
- [ngx\\_http\\_sub\\_filter\\_module\\_ctx](#)

## Data types defined

- [ngx\\_http\\_sub\\_ctx\\_t](#)
- [ngx\\_http\\_sub\\_loc\\_conf\\_t](#)
- [ngx\\_http\\_sub\\_state\\_e](#)

## Functions defined

- [ngx\\_http\\_sub\\_body\\_filter](#)
- [ngx\\_http\\_sub\\_create\\_conf](#)
- [ngx\\_http\\_sub\\_filter](#)
- [ngx\\_http\\_sub\\_filter\\_init](#)
- [ngx\\_http\\_sub\\_header\\_filter](#)
- [ngx\\_http\\_sub\\_merge\\_conf](#)
- [ngx\\_http\\_sub\\_output](#)
- [ngx\\_http\\_sub\\_parse](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 typedef struct {
14     ngx\_str\_t match;
15     ngx\_http\_complex\_value\_t value;
16
17     ngx\_hash\_t types;
18
19     ngx\_flag\_t once;
20     ngx\_flag\_t last_modified;
21
```

```

22     ngx\_array\_t                *types_keys;
23 } ngx\_http\_sub\_loc\_conf\_t;
24
25
26 typedef enum {
27     sub_start_state = 0,
28     sub_match_state,
29 } ngx\_http\_sub\_state\_e;
30
31
32 typedef struct {
33     ngx\_str\_t                match;
34     ngx\_str\_t                saved;
35     ngx\_str\_t                looked;
36
37     ngx\_uint\_t              once;    /* unsigned once:1 */
38
39     ngx\_buf\_t               *buf;
40
41     u_char                 *pos;
42     u_char                 *copy_start;
43     u_char                 *copy_end;
44
45     ngx\_chain\_t             *in;
46     ngx\_chain\_t             *out;
47     ngx\_chain\_t             **last_out;
48     ngx\_chain\_t             *busy;
49     ngx\_chain\_t             *free;
50
51     ngx\_str\_t                sub;
52
53     ngx\_uint\_t              state;
54 } ngx\_http\_sub\_ctx\_t;
55
56
57 static ngx\_int\_t ngx\_http\_sub\_output(ngx\_http\_request\_t *r,
58     ngx\_http\_sub\_ctx\_t *ctx);
59 static ngx\_int\_t ngx\_http\_sub\_parse(ngx\_http\_request\_t *r,
60     ngx\_http\_sub\_ctx\_t *ctx);
61
62 static char * ngx\_http\_sub\_filter(ngx\_conf\_t *cf, ngx\_command\_t *cmd,
63     void *conf);
64 static void *ngx\_http\_sub\_create\_conf(ngx\_conf\_t *cf);
65 static char *ngx\_http\_sub\_merge\_conf(ngx\_conf\_t *cf,
66     void *parent, void *child);
67 static ngx\_int\_t ngx\_http\_sub\_filter\_init(ngx\_conf\_t *cf);
68
69
70 static ngx\_command\_t ngx\_http\_sub\_filter\_commands[] = {
71
72     { ngx\_string("sub_filter"),
73       NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE2,
74       ngx\_http\_sub\_filter,
75       NGX\_HTTP\_LOC\_CONF\_OFFSET,
76       0,
77       NULL },
78
79     { ngx\_string("sub_filter_types"),
80       NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_1MORE,
81       ngx\_http\_types\_slot,
82       NGX\_HTTP\_LOC\_CONF\_OFFSET,
83       offsetof(ngx\_http\_sub\_loc\_conf\_t, types_keys),
84       &ngx\_http\_html\_default\_types[0] },
85
86     { ngx\_string("sub_filter_once"),
87       NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_FLAG,
88       ngx\_conf\_set\_flag\_slot,
89       NGX\_HTTP\_LOC\_CONF\_OFFSET,
90       offsetof(ngx\_http\_sub\_loc\_conf\_t, once),
91       NULL },
92
93     { ngx\_string("sub_filter_last_modified"),
94       NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_FLAG,
95       ngx\_conf\_set\_flag\_slot,
96       NGX\_HTTP\_LOC\_CONF\_OFFSET,
97       offsetof(ngx\_http\_sub\_loc\_conf\_t, last_modified),

```

```

98     NULL },
99
100     ngx_null_command
101 };
102
103
104 static ngx_http_module_t ngx_http_sub_filter_module_ctx = {
105     NULL,                                     /* preconfiguration */
106     ngx_http_sub_filter_init,              /* postconfiguration */
107
108     NULL,                                     /* create main configuration */
109     NULL,                                     /* init main configuration */
110
111     NULL,                                     /* create server configuration */
112     NULL,                                     /* merge server configuration */
113
114     ngx_http_sub_create_conf,              /* create location configuration */
115     ngx_http_sub_merge_conf,              /* merge location configuration */
116 };
117
118
119 ngx_module_t ngx_http_sub_filter_module = {
120     NGX_MODULE_V1,
121     &ngx_http_sub_filter_module_ctx,        /* module context */
122     ngx_http_sub_filter_commands,        /* module directives */
123     NGX_HTTP_MODULE,                     /* module type */
124     NULL,                                    /* init master */
125     NULL,                                    /* init module */
126     NULL,                                    /* init process */
127     NULL,                                    /* init thread */
128     NULL,                                    /* exit thread */
129     NULL,                                    /* exit process */
130     NULL,                                    /* exit master */
131     NGX_MODULE_V1_PADDING
132 };
133
134
135 static ngx_http_output_header_filter_pt ngx_http_next_header_filter;
136 static ngx_http_output_body_filter_pt ngx_http_next_body_filter;
137
138
139 static ngx_int_t
140 ngx_http_sub_header_filter(ngx_http_request_t *r)
141 {
142     ngx_http_sub_ctx_t      *ctx;
143     ngx_http_sub_loc_conf_t *slcf;
144
145     slcf = ngx_http_get_module_loc_conf(r, ngx_http_sub_filter_module);
146
147     if (slcf->match.len == 0
148         || r->headers_out.content_length_n == 0
149         || ngx_http_test_content_type(r, &slcf->types) == NULL)
150     {
151         return ngx_http_next_header_filter(r);
152     }
153
154     ctx = ngx_palloc(r->pool, sizeof(ngx_http_sub_ctx_t));
155     if (ctx == NULL) {
156         return NGX_ERROR;
157     }
158
159     ctx->saved.data = ngx_pnalloc(r->pool, slcf->match.len);
160     if (ctx->saved.data == NULL) {
161         return NGX_ERROR;
162     }
163
164     ctx->looked.data = ngx_pnalloc(r->pool, slcf->match.len);
165     if (ctx->looked.data == NULL) {
166         return NGX_ERROR;
167     }
168
169     ngx_http_set_ctx(r, ctx, ngx_http_sub_filter_module);
170
171     ctx->match = slcf->match;
172     ctx->last_out = &ctx->out;
173

```

```

174     r->filter_need_in_memory = 1;
175
176     if (r == r->main) {
177         ngx_http_clear_content_length(r);
178
179         if (!slcf->last_modified) {
180             ngx_http_clear_last_modified(r);
181             ngx_http_clear_etag(r);
182
183         } else {
184             ngx_http_weak_etag(r);
185         }
186     }
187
188     return ngx_http_next_header_filter(r);
189 }
190
191
192 static ngx_int_t
193 ngx_http_sub_body_filter(ngx_http_request_t *r, ngx_chain_t *in)
194 {
195     ngx_int_t          rc;
196     ngx_buf_t         *b;
197     ngx_chain_t       *cl;
198     ngx_http_sub_ctx_t *ctx;
199     ngx_http_sub_loc_conf_t *slcf;
200
201     ctx = ngx_http_get_module_ctx(r, ngx_http_sub_filter_module);
202
203     if (ctx == NULL) {
204         return ngx_http_next_body_filter(r, in);
205     }
206
207     if ((in == NULL
208         && ctx->buf == NULL
209         && ctx->in == NULL
210         && ctx->busy == NULL))
211     {
212         return ngx_http_next_body_filter(r, in);
213     }
214
215     if (ctx->once && (ctx->buf == NULL || ctx->in == NULL)) {
216
217         if (ctx->busy) {
218             if (ngx_http_sub_output(r, ctx) == NGX_ERROR) {
219                 return NGX_ERROR;
220             }
221         }
222
223         return ngx_http_next_body_filter(r, in);
224     }
225
226     /* add the incoming chain to the chain ctx->in */
227
228     if (in) {
229         if (ngx_chain_add_copy(r->pool, &ctx->in, in) != NGX_OK) {
230             return NGX_ERROR;
231         }
232     }
233
234     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
235                  "http sub filter \"%V\"", &r->uri);
236
237     while (ctx->in || ctx->buf) {
238
239         if (ctx->buf == NULL) {
240             ctx->buf = ctx->in->buf;
241             ctx->in = ctx->in->next;
242             ctx->pos = ctx->buf->pos;
243         }
244
245         if (ctx->state == sub_start_state) {
246             ctx->copy_start = ctx->pos;
247             ctx->copy_end = ctx->pos;
248         }
249

```

```

250     b = NULL;
251
252     while (ctx->pos < ctx->buf->last) {
253
254         ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
255             "saved: \"%V\" state: %d", &ctx->saved, ctx->state);
256
257         rc = ngx_http_sub_parse(r, ctx);
258
259         ngx_log_debug4(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
260             "parse: %d, looked: \"%V\" %p-%p",
261             rc, &ctx->looked, ctx->copy_start, ctx->copy_end);
262
263         if (rc == NGX_ERROR) {
264             return rc;
265         }
266
267         if (ctx->saved.len) {
268
269             ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
270                 "saved: \"%V\"", &ctx->saved);
271
272             cl = ngx_chain_get_free_buf(r->pool, &ctx->free);
273             if (cl == NULL) {
274                 return NGX_ERROR;
275             }
276
277             b = cl->buf;
278
279             ngx_memzero(b, sizeof(ngx_buf_t));
280
281             b->pos = ngx_pnalloc(r->pool, ctx->saved.len);
282             if (b->pos == NULL) {
283                 return NGX_ERROR;
284             }
285
286             ngx_memcpy(b->pos, ctx->saved.data, ctx->saved.len);
287             b->last = b->pos + ctx->saved.len;
288             b->memory = 1;
289
290             *ctx->last_out = cl;
291             ctx->last_out = &cl->next;
292
293             ctx->saved.len = 0;
294         }
295
296         if (ctx->copy_start != ctx->copy_end) {
297
298             cl = ngx_chain_get_free_buf(r->pool, &ctx->free);
299             if (cl == NULL) {
300                 return NGX_ERROR;
301             }
302
303             b = cl->buf;
304
305             ngx_memcpy(b, ctx->buf, sizeof(ngx_buf_t));
306
307             b->pos = ctx->copy_start;
308             b->last = ctx->copy_end;
309             b->shadow = NULL;
310             b->last_buf = 0;
311             b->last_in_chain = 0;
312             b->recycled = 0;
313
314             if (b->in_file) {
315                 b->file_last = b->file_pos + (b->last - ctx->buf->pos);
316                 b->file_pos += b->pos - ctx->buf->pos;
317             }
318
319             *ctx->last_out = cl;
320             ctx->last_out = &cl->next;
321         }
322
323         if (ctx->state == sub_start_state) {
324             ctx->copy_start = ctx->pos;
325             ctx->copy_end = ctx->pos;

```

```

326 } else {
327     ctx->copy_start = NULL;
328     ctx->copy_end = NULL;
329 }
330
331
332 if (ctx->looked.len > (size_t) (ctx->pos - ctx->buf->pos)) {
333     ctx->saved.len = ctx->looked.len - (ctx->pos - ctx->buf->pos);
334     ngx_memcpy(ctx->saved.data, ctx->looked.data, ctx->saved.len);
335 }
336
337 if (rc == NGX_AGAIN) {
338     continue;
339 }
340
341
342 /* rc == NGX_OK */
343
344 cl = ngx_chain_get_free_buf(r->pool, &ctx->free);
345 if (cl == NULL) {
346     return NGX_ERROR;
347 }
348
349 b = cl->buf;
350
351 ngx_memzero(b, sizeof(ngx_buf_t));
352
353 slcf = ngx_http_get_module_loc_conf(r, ngx_http_sub_filter_module);
354
355 if (ctx->sub.data == NULL) {
356     if (ngx_http_complex_value(r, &slcf->value, &ctx->sub)
357         != NGX_OK)
358     {
359         return NGX_ERROR;
360     }
361 }
362
363 if (ctx->sub.len) {
364     b->memory = 1;
365     b->pos = ctx->sub.data;
366     b->last = ctx->sub.data + ctx->sub.len;
367 } else {
368     b->sync = 1;
369 }
370
371 *ctx->last_out = cl;
372 ctx->last_out = &cl->next;
373
374 ctx->once = slcf->once;
375
376 continue;
377 }
378
379
380
381 if (ctx->looked.len
382     && (ctx->buf->last_buf || ctx->buf->last_in_chain))
383 {
384     cl = ngx_chain_get_free_buf(r->pool, &ctx->free);
385     if (cl == NULL) {
386         return NGX_ERROR;
387     }
388
389     b = cl->buf;
390
391     ngx_memzero(b, sizeof(ngx_buf_t));
392
393     b->pos = ctx->looked.data;
394     b->last = b->pos + ctx->looked.len;
395     b->memory = 1;
396
397     *ctx->last_out = cl;
398     ctx->last_out = &cl->next;
399
400     ctx->looked.len = 0;
401 }

```

```

402     if (ctx->buf->last_buf || ctx->buf->flush || ctx->buf->sync
403         || ngx_buf_in_memory(ctx->buf))
404     {
405         if (b == NULL) {
406             cl = ngx_chain_get_free_buf(r->pool, &ctx->free);
407             if (cl == NULL) {
408                 return NGX_ERROR;
409             }
410
411             b = cl->buf;
412
413             ngx_memzero(b, sizeof(ngx_buf_t));
414
415             b->sync = 1;
416
417             *ctx->last_out = cl;
418             ctx->last_out = &cl->next;
419         }
420
421         b->last_buf = ctx->buf->last_buf;
422         b->last_in_chain = ctx->buf->last_in_chain;
423         b->flush = ctx->buf->flush;
424         b->shadow = ctx->buf;
425
426         b->recycled = ctx->buf->recycled;
427     }
428
429     ctx->buf = NULL;
430
431     ctx->saved.len = ctx->looked.len;
432     ngx_memcpy(ctx->saved.data, ctx->looked.data, ctx->looked.len);
433 }
434
435 if (ctx->out == NULL && ctx->busy == NULL) {
436     return NGX_OK;
437 }
438
439 return ngx_http_sub_output(r, ctx);
440 }
441
442
443 static ngx_int_t
444 ngx_http_sub_output(ngx_http_request_t *r, ngx_http_sub_ctx_t *ctx)
445 {
446     ngx_int_t    rc;
447     ngx_buf_t    *b;
448     ngx_chain_t  *cl;
449
450 #if 1
451     b = NULL;
452     for (cl = ctx->out; cl; cl = cl->next) {
453         ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
454             "sub out: %p %p", cl->buf, cl->buf->pos);
455         if (cl->buf == b) {
456             ngx_log_error(NGX_LOG_ALERT, r->connection->log, 0,
457                 "the same buf was used in sub");
458             ngx_debug_point();
459             return NGX_ERROR;
460         }
461         b = cl->buf;
462     }
463 #endif
464
465     rc = ngx_http_next_body_filter(r, ctx->out);
466
467     if (ctx->busy == NULL) {
468         ctx->busy = ctx->out;
469     }
470
471     } else {
472         for (cl = ctx->busy; cl->next; cl = cl->next) { /* void */ }
473         cl->next = ctx->out;
474     }
475
476     ctx->out = NULL;
477     ctx->last_out = &ctx->out;

```

```

478
479 while (ctx->busy) {
480
481     cl = ctx->busy;
482     b = cl->buf;
483
484     if (ngx\_buf\_size(b) != 0) {
485         break;
486     }
487
488     if (b->shadow) {
489         b->shadow->pos = b->shadow->last;
490     }
491
492     ctx->busy = cl->next;
493
494     if (ngx\_buf\_in\_memory(b) || b->in_file) {
495         /* add data bufs only to the free buf chain */
496
497         cl->next = ctx->free;
498         ctx->free = cl;
499     }
500 }
501
502 if (ctx->in || ctx->buf) {
503     r->buffered |= NGX\_HTTP\_SUB\_BUFFERED;
504 } else {
505     r->buffered &= ~NGX\_HTTP\_SUB\_BUFFERED;
506 }
507
508 return rc;
509 }
510
511
512
513 static ngx\_int\_t
514 ngx\_http\_sub\_parse(ngx\_http\_request\_t *r, ngx\_http\_sub\_ctx\_t *ctx)
515 {
516     u_char          *p, *last, *copy_end, ch, match;
517     size_t          looked, i;
518     ngx\_http\_sub\_state\_e state;
519
520     if (ctx->once) {
521         ctx->copy_start = ctx->pos;
522         ctx->copy_end = ctx->buf->last;
523         ctx->pos = ctx->buf->last;
524         ctx->looked.len = 0;
525
526         ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0, "once");
527
528         return NGX\_AGAIN;
529     }
530
531     state = ctx->state;
532     looked = ctx->looked.len;
533     last = ctx->buf->last;
534     copy_end = ctx->copy_end;
535
536     for (p = ctx->pos; p < last; p++) {
537
538         ch = *p;
539         ch = ngx\_tolower(ch);
540
541         if (state == sub_start_state) {
542
543             /* the tight loop */
544
545             match = ctx->match.data[0];
546
547             for ( ;; ) {
548                 if (ch == match) {
549
550                     if (ctx->match.len == 1) {
551                         ctx->pos = p + 1;
552                         ctx->copy_end = p;
553

```



```

554         return NGX_OK;
555     }
556
557     copy_end = p;
558     ctx->looked.data[0] = *p;
559     looked = 1;
560     state = sub_match_state;
561
562     goto match_started;
563 }
564
565 if (++p == last) {
566     break;
567 }
568
569 ch = *p;
570 ch = ngx_tolower(ch);
571 }
572
573 ctx->state = state;
574 ctx->pos = p;
575 ctx->looked.len = looked;
576 ctx->copy_end = p;
577
578 if (ctx->copy_start == NULL) {
579     ctx->copy_start = ctx->buf->pos;
580 }
581
582 return NGX_AGAIN;
583
584 match_started:
585
586     continue;
587 }
588
589 /* state == sub_match_state */
590
591 if (ch == ctx->match.data[looked]) {
592     ctx->looked.data[looked] = *p;
593     looked++;
594
595     if (looked == ctx->match.len) {
596
597         ctx->state = sub_start_state;
598         ctx->pos = p + 1;
599         ctx->looked.len = 0;
600         ctx->saved.len = 0;
601         ctx->copy_end = copy_end;
602
603         if (ctx->copy_start == NULL && copy_end) {
604             ctx->copy_start = ctx->buf->pos;
605         }
606
607         return NGX_OK;
608     }
609 } else {
610     /*
611      * check if there is another partial match in previously
612      * matched substring to catch cases like "aab" in "aabb"
613      */
614
615     ctx->looked.data[looked] = *p;
616     looked++;
617
618     for (i = 1; i < looked; i++) {
619         if (ngx_strncasecmp(ctx->looked.data + i,
620                             ctx->match.data, looked - i)
621             == 0)
622             {
623                 break;
624             }
625     }
626
627     if (i < looked) {
628         if (ctx->saved.len > i) {

```

```

630         ctx->saved.len = i;
631     }
632
633     if ((size_t) (p + 1 - ctx->buf->pos) >= looked - i) {
634         copy_end = p + 1 - (looked - i);
635     }
636
637     ngx_memmove(ctx->looked.data, ctx->looked.data + i, looked - i);
638     looked = looked - i;
639
640 } else {
641     copy_end = p;
642     looked = 0;
643     state = sub_start_state;
644 }
645
646 if (ctx->saved.len) {
647     p++;
648     goto out;
649 }
650 }
651 }
652
653 ctx->saved.len = 0;
654
655 out:
656
657 ctx->state = state;
658 ctx->pos = p;
659 ctx->looked.len = looked;
660
661 ctx->copy_end = (state == sub_start_state) ? p : copy_end;
662
663 if (ctx->copy_start == NULL && ctx->copy_end) {
664     ctx->copy_start = ctx->buf->pos;
665 }
666
667 return NGX_AGAIN;
668 }
669
670
671 static char *
672 ngx_http_sub_filter(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
673 {
674     ngx_http_sub_loc_conf_t *slcf = conf;
675
676     ngx_str_t          *value;
677     ngx_http_compile_complex_value_t ccv;
678
679     if (slcf->match.data) {
680         return "is duplicate";
681     }
682
683     value = cf->args->elts;
684
685     ngx_strlow(value[1].data, value[1].data, value[1].len);
686
687     slcf->match = value[1];
688
689     ngx_memzero(&ccv, sizeof(ngx_http_compile_complex_value_t));
690
691     ccv.cf = cf;
692     ccv.value = &value[2];
693     ccv.complex_value = &slcf->value;
694
695     if (ngx_http_compile_complex_value(&ccv) != NGX_OK) {
696         return NGX_CONF_ERROR;
697     }
698
699     return NGX_CONF_OK;
700 }
701
702
703 static void *
704 ngx_http_sub_create_conf(ngx_conf_t *cf)
705 {

```

```

706 ngx\_http\_sub\_loc\_conf\_t *slcf;
707
708 slcf = ngx\_palloc(cf->pool, sizeof(ngx\_http\_sub\_loc\_conf\_t));
709 if (slcf == NULL) {
710     return NULL;
711 }
712
713 /*
714  * set by ngx\_palloc\(\):
715  *
716  *     conf->match = { 0, NULL };
717  *     conf->types = { NULL };
718  *     conf->types_keys = NULL;
719  */
720
721 slcf->once = NGX\_CONF\_UNSET;
722 slcf->last_modified = NGX\_CONF\_UNSET;
723
724 return slcf;
725 }
726
727
728 static char *
729 ngx\_http\_sub\_merge\_conf(ngx\_conf\_t *cf, void *parent, void *child)
730 {
731     ngx\_http\_sub\_loc\_conf\_t *prev = parent;
732     ngx\_http\_sub\_loc\_conf\_t *conf = child;
733
734     ngx\_conf\_merge\_value(conf->once, prev->once, 1);
735     ngx\_conf\_merge\_str\_value(conf->match, prev->match, "");
736     ngx\_conf\_merge\_value(conf->last_modified, prev->last_modified, 0);
737
738     if (conf->value.value.data == NULL) {
739         conf->value = prev->value;
740     }
741
742     if (ngx\_http\_merge\_types(cf, &conf->types_keys, &conf->types,
743                             &prev->types_keys, &prev->types,
744                             ngx\_http\_html\_default\_types)
745         != NGX\_OK)
746     {
747         return NGX\_CONF\_ERROR;
748     }
749
750     return NGX\_CONF\_OK;
751 }
752
753
754 static ngx\_int\_t
755 ngx\_http\_sub\_filter\_init(ngx\_conf\_t *cf)
756 {
757     ngx\_http\_next\_header\_filter = ngx\_http\_top\_header\_filter;
758     ngx\_http\_top\_header\_filter = ngx\_http\_sub\_header\_filter;
759
760     ngx\_http\_next\_body\_filter = ngx\_http\_top\_body\_filter;
761     ngx\_http\_top\_body\_filter = ngx\_http\_sub\_body\_filter;
762
763     return NGX\_OK;
764 }

```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_upstream\_hash\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_upstream\\_hash\\_commands](#)
- [ngx\\_http\\_upstream\\_hash\\_module](#)
- [ngx\\_http\\_upstream\\_hash\\_module\\_ctx](#)

## Data types defined

- [ngx\\_http\\_upstream\\_chash\\_point\\_t](#)
- [ngx\\_http\\_upstream\\_chash\\_points\\_t](#)
- [ngx\\_http\\_upstream\\_hash\\_peer\\_data\\_t](#)
- [ngx\\_http\\_upstream\\_hash\\_srv\\_conf\\_t](#)

## Functions defined

- [ngx\\_http\\_upstream\\_add\\_chash\\_point](#)
- [ngx\\_http\\_upstream\\_find\\_chash\\_point](#)
- [ngx\\_http\\_upstream\\_get\\_chash\\_peer](#)
- [ngx\\_http\\_upstream\\_get\\_hash\\_peer](#)
- [ngx\\_http\\_upstream\\_hash](#)
- [ngx\\_http\\_upstream\\_hash\\_create\\_conf](#)
- [ngx\\_http\\_upstream\\_init\\_chash](#)
- [ngx\\_http\\_upstream\\_init\\_chash\\_peer](#)
- [ngx\\_http\\_upstream\\_init\\_hash](#)
- [ngx\\_http\\_upstream\\_init\\_hash\\_peer](#)

## Source code

```
1
2 /*
3  * Copyright (C) Roman Arutyunyan
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 typedef struct {
14     uint32_t                hash;
15     ngx_str_t               *server;
16 } ngx_http_upstream_chash_point_t;
17
18
19 typedef struct {
```

```

20     ngx_uint_t          number;
21     ngx_http_upstream_chash_point_t point[1];
22 } ngx_http_upstream_chash_points_t;
23
24
25 typedef struct {
26     ngx_http_complex_value_t key;
27     ngx_http_upstream_chash_points_t *points;
28 } ngx_http_upstream_hash_srv_conf_t;
29
30
31 typedef struct {
32     /* the round robin data must be first */
33     ngx_http_upstream_rr_peer_data_t rrp;
34     ngx_http_upstream_hash_srv_conf_t *conf;
35     ngx_str_t key;
36     ngx_uint_t tries;
37     ngx_uint_t rehash;
38     uint32_t hash;
39     ngx_event_get_peer_pt get_rr_peer;
40 } ngx_http_upstream_hash_peer_data_t;
41
42
43 static ngx_int_t ngx_http_upstream_init_hash(ngx_conf_t *cf,
44     ngx_http_upstream_srv_conf_t *us);
45 static ngx_int_t ngx_http_upstream_init_hash_peer(ngx_http_request_t *r,
46     ngx_http_upstream_srv_conf_t *us);
47 static ngx_int_t ngx_http_upstream_get_hash_peer(ngx_peer_connection_t *pc,
48     void *data);
49
50 static ngx_int_t ngx_http_upstream_init_chash(ngx_conf_t *cf,
51     ngx_http_upstream_srv_conf_t *us);
52 static void ngx_http_upstream_add_chash_point(
53     ngx_http_upstream_chash_points_t *points, uint32_t hash, ngx_str_t *server);
54 static ngx_uint_t ngx_http_upstream_find_chash_point(
55     ngx_http_upstream_chash_points_t *points, uint32_t hash);
56 static ngx_int_t ngx_http_upstream_init_chash_peer(ngx_http_request_t *r,
57     ngx_http_upstream_srv_conf_t *us);
58 static ngx_int_t ngx_http_upstream_get_chash_peer(ngx_peer_connection_t *pc,
59     void *data);
60
61 static void *ngx_http_upstream_hash_create_conf(ngx_conf_t *cf);
62 static char *ngx_http_upstream_hash(ngx_conf_t *cf, ngx_command_t *cmd,
63     void *conf);
64
65
66 static ngx_command_t ngx_http_upstream_hash_commands[] = {
67
68     { ngx_string("hash"),
69       NGX_HTTP_UPS_CONF|NGX_CONF_TAKE12,
70       ngx_http_upstream_hash,
71       NGX_HTTP_SRV_CONF_OFFSET,
72       0,
73       NULL },
74
75     ngx_null_command
76 };
77
78
79 static ngx_http_module_t ngx_http_upstream_hash_module_ctx = {
80     NULL, /* preconfiguration */
81     NULL, /* postconfiguration */
82
83     NULL, /* create main configuration */
84     NULL, /* init main configuration */
85
86     ngx_http_upstream_hash_create_conf, /* create server configuration */
87     NULL, /* merge server configuration */
88
89     NULL, /* create location configuration */
90     NULL, /* merge location configuration */
91 };
92
93
94 ngx_module_t ngx_http_upstream_hash_module = {
95     NGX_MODULE_V1,

```

```

96     &ngx_http_upstream_hash_module_ctx,      /* module context */
97     ngx_http_upstream_hash_commands,        /* module directives */
98     NGX_HTTP_MODULE,                        /* module type */
99     NULL,                                    /* init master */
100    NULL,                                    /* init module */
101    NULL,                                    /* init process */
102    NULL,                                    /* init thread */
103    NULL,                                    /* exit thread */
104    NULL,                                    /* exit process */
105    NULL,                                    /* exit master */
106    NGX_MODULE_V1_PADDING
107 };
108
109
110 static ngx_int_t
111 ngx_http_upstream_init_hash(ngx_conf_t *cf, ngx_http_upstream_srv_conf_t *us)
112 {
113     if (ngx_http_upstream_init_round_robin(cf, us) != NGX_OK) {
114         return NGX_ERROR;
115     }
116
117     us->peer.init = ngx_http_upstream_init_hash_peer;
118
119     return NGX_OK;
120 }
121
122
123 static ngx_int_t
124 ngx_http_upstream_init_hash_peer(ngx_http_request_t *r,
125     ngx_http_upstream_srv_conf_t *us)
126 {
127     ngx_http_upstream_hash_srv_conf_t *hcf;
128     ngx_http_upstream_hash_peer_data_t *hp;
129
130     hp = ngx_palloc(r->pool, sizeof(ngx_http_upstream_hash_peer_data_t));
131     if (hp == NULL) {
132         return NGX_ERROR;
133     }
134
135     r->upstream->peer.data = &hp->rrp;
136
137     if (ngx_http_upstream_init_round_robin_peer(r, us) != NGX_OK) {
138         return NGX_ERROR;
139     }
140
141     r->upstream->peer.get = ngx_http_upstream_get_hash_peer;
142
143     hcf = ngx_http_conf_upstream_srv_conf(us, ngx_http_upstream_hash_module);
144
145     if (ngx_http_complex_value(r, &hcf->key, &hp->key) != NGX_OK) {
146         return NGX_ERROR;
147     }
148
149     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
150         "upstream hash key: \"%V\"", &hp->key);
151
152     hp->conf = hcf;
153     hp->tries = 0;
154     hp->rehash = 0;
155     hp->hash = 0;
156     hp->get_rr_peer = ngx_http_upstream_get_round_robin_peer;
157
158     return NGX_OK;
159 }
160
161
162 static ngx_int_t
163 ngx_http_upstream_get_hash_peer(ngx_peer_connection_t *pc, void *data)
164 {
165     ngx_http_upstream_hash_peer_data_t *hp = data;
166
167     time_t          now;
168     u_char          buf[NGX_INT_T_LEN];
169     size_t          size;
170     uint32_t        hash;
171     ngx_int_t       w;

```

```

172     uintptr_t             m;
173     ngx_uint_t           i, n, p;
174     ngx_http_upstream_rr_peer_t *peer;
175
176     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, pc->log, 0,
177                  "get hash peer, try: %ui", pc->tries);
178
179     if (hp->tries > 20 || hp->rrp.peers->single) {
180         return hp->get_rr_peer(pc, &hp->rrp);
181     }
182
183     now = ngx_time();
184
185     pc->cached = 0;
186     pc->connection = NULL;
187
188     for ( ;; ) {
189
190         /*
191          * Hash expression is compatible with Cache::Memcached:
192          * ((crc32([REHASH] KEY) >> 16) & 0x7fff) + PREV_HASH
193          * with REHASH omitted at the first iteration.
194          */
195
196         ngx_crc32_init(hash);
197
198         if (hp->rehash > 0) {
199             size = ngx_sprintf(buf, "%ui", hp->rehash) - buf;
200             ngx_crc32_update(&hash, buf, size);
201         }
202
203         ngx_crc32_update(&hash, hp->key.data, hp->key.len);
204         ngx_crc32_final(hash);
205
206         hash = (hash >> 16) & 0x7fff;
207
208         hp->hash += hash;
209         hp->rehash++;
210
211         if (!hp->rrp.peers->weighted) {
212             p = hp->hash % hp->rrp.peers->number;
213
214         } else {
215             w = hp->hash % hp->rrp.peers->total_weight;
216
217             for (i = 0; i < hp->rrp.peers->number; i++) {
218                 w -= hp->rrp.peers->peer[i].weight;
219                 if (w < 0) {
220                     break;
221                 }
222             }
223
224             p = i;
225         }
226
227         n = p / (8 * sizeof(uintptr_t));
228         m = (uintptr_t) 1 << p % (8 * sizeof(uintptr_t));
229
230         if (hp->rrp.tried[n] & m) {
231             goto next;
232         }
233
234         ngx_log_debug2(NGX_LOG_DEBUG_HTTP, pc->log, 0,
235                      "get hash peer, value:%uD, peer:%ui", hp->hash, p);
236
237         peer = &hp->rrp.peers->peer[p];
238
239         if (peer->down) {
240             goto next;
241         }
242
243         if (peer->max_fails
244             && peer->fails >= peer->max_fails
245             && now - peer->checked <= peer->fail_timeout)
246         {
247             goto next;

```

```

248     }
249
250     break;
251
252 next:
253
254     if (++hp->tries > 20) {
255         return hp->get_rr_peer(pc, &hp->rrp);
256     }
257 }
258
259 hp->rrp.current = p;
260
261 pc->sockaddr = peer->sockaddr;
262 pc->socklen = peer->socklen;
263 pc->name = &peer->name;
264
265 if (now - peer->checked > peer->fail_timeout) {
266     peer->checked = now;
267 }
268
269 hp->rrp.tried[n] |= m;
270
271 return NGX_OK;
272 }
273
274
275 static ngx_int_t
276 ngx_http_upstream_init_chash(ngx_conf_t *cf, ngx_http_upstream_srv_conf_t *us)
277 {
278     u_char          *host, *port, c;
279     size_t          host_len, port_len, size;
280     uint32_t        hash, base_hash, prev_hash;
281     ngx_str_t       *server;
282     ngx_uint_t      npoints, i, j;
283     ngx_http_upstream_rr_peer_t *peer;
284     ngx_http_upstream_rr_peers_t *peers;
285     ngx_http_upstream_chash_points_t *points;
286     ngx_http_upstream_hash_srv_conf_t *hcf;
287
288     if (ngx_http_upstream_init_round_robin(cf, us) != NGX_OK) {
289         return NGX_ERROR;
290     }
291
292     us->peer.init = ngx_http_upstream_init_chash_peer;
293
294     peers = us->peer.data;
295     npoints = peers->total_weight * 160;
296
297     size = sizeof(ngx_http_upstream_chash_points_t)
298         + sizeof(ngx_http_upstream_chash_point_t) * (npoints - 1);
299
300     points = ngx_palloc(cf->pool, size);
301     if (points == NULL) {
302         return NGX_ERROR;
303     }
304
305     points->number = 0;
306
307     for (i = 0; i < peers->number; i++) {
308         peer = &peers->peer[i];
309         server = &peer->server;
310
311         /*
312          * Hash expression is compatible with Cache::Memcached::Fast:
313          * crc32(HOST \0 PORT PREV_HASH).
314          */
315
316         if (server->len >= 5
317             && ngx_strncasecmp(server->data, (u_char *) "unix:", 5) == 0)
318         {
319             host = server->data + 5;
320             host_len = server->len - 5;
321             port = NULL;
322             port_len = 0;
323             goto done;

```



```

324     }
325
326     for (j = 0; j < server->len; j++) {
327         c = server->data[server->len - j - 1];
328
329         if (c == ':') {
330             host = server->data;
331             host_len = server->len - j - 1;
332             port = server->data + server->len - j;
333             port_len = j;
334             goto done;
335         }
336
337         if (c < '0' || c > '9') {
338             break;
339         }
340     }
341
342     host = server->data;
343     host_len = server->len;
344     port = NULL;
345     port_len = 0;
346
347     done:
348
349     ngx_crc32_init(base_hash);
350     ngx_crc32_update(&base_hash, host, host_len);
351     ngx_crc32_update(&base_hash, (u_char *) "", 1);
352     ngx_crc32_update(&base_hash, port, port_len);
353
354     prev_hash = 0;
355     npoints = peer->weight * 160;
356
357     for (j = 0; j < npoints; j++) {
358         hash = base_hash;
359
360         ngx_crc32_update(&hash, (u_char *) &prev_hash, sizeof(uint32_t));
361         ngx_crc32_final(hash);
362
363         ngx_http_upstream_add_chash_point(points, hash, &peer->server);
364
365         prev_hash = hash;
366     }
367 }
368
369 hcf = ngx_http_conf_upstream_srv_conf(us, ngx_http_upstream_hash_module);
370 hcf->points = points;
371
372 return NGX_OK;
373 }
374
375
376 static void
377 ngx_http_upstream_add_chash_point(ngx_http_upstream_chash_points_t *points,
378     uint32_t hash, ngx_str_t *server)
379 {
380     size_t          size;
381     ngx_uint_t      i;
382     ngx_http_upstream_chash_point_t *point;
383
384     i = ngx_http_upstream_find_chash_point(points, hash);
385     point = &points->point[i];
386
387     if (point->hash == hash) {
388         return;
389     }
390
391     size = (points->number - i) * sizeof(ngx_http_upstream_chash_point_t);
392
393     ngx_memmove(point + 1, point, size);
394
395     points->number++;
396     point->hash = hash;
397     point->server = server;
398 }
399

```

```

400 static ngx_uint_t
401 ngx_http_upstream_find_chash_point(ngx_http_upstream_chash_points_t *points,
402     uint32_t hash)
403 {
404     ngx_uint_t          i, j, k;
405     ngx_http_upstream_chash_point_t *point;
406
407     /* find first point >= hash */
408
409     point = &points->point[0];
410
411     i = 0;
412     j = points->number;
413
414     while (i < j) {
415         k = (i + j) / 2;
416
417         if (hash > point[k].hash) {
418             i = k + 1;
419
420         } else if (hash < point[k].hash) {
421             j = k;
422
423         } else {
424             return k;
425         }
426     }
427
428     return i;
429 }
430
431
432
433 static ngx_int_t
434 ngx_http_upstream_init_chash_peer(ngx_http_request_t *r,
435     ngx_http_upstream_srv_conf_t *us)
436 {
437     uint32_t          hash;
438     ngx_http_upstream_hash_srv_conf_t *hcf;
439     ngx_http_upstream_hash_peer_data_t *hp;
440
441     if (ngx_http_upstream_init_hash_peer(r, us) != NGX_OK) {
442         return NGX_ERROR;
443     }
444
445     r->upstream->peer.get = ngx_http_upstream_get_chash_peer;
446
447     hp = r->upstream->peer.data;
448     hcf = ngx_http_conf_upstream_srv_conf(us, ngx_http_upstream_hash_module);
449
450     hash = ngx_crc32_long(hp->key.data, hp->key.len);
451     hp->hash = ngx_http_upstream_find_chash_point(hcf->points, hash);
452
453     return NGX_OK;
454 }
455
456
457 static ngx_int_t
458 ngx_http_upstream_get_chash_peer(ngx_peer_connection_t *pc, void *data)
459 {
460     ngx_http_upstream_hash_peer_data_t *hp = data;
461
462     time_t          now;
463     intptr_t        m;
464     ngx_str_t       *server;
465     ngx_int_t       total;
466     ngx_uint_t      i, n;
467     ngx_http_upstream_rr_peer_t *peer, *best;
468     ngx_http_upstream_chash_point_t *point;
469     ngx_http_upstream_chash_points_t *points;
470     ngx_http_upstream_hash_srv_conf_t *hcf;
471
472     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, pc->log, 0,
473         "get consistent hash peer, try: %ui", pc->tries);
474
475     pc->cached = 0;

```

```

476 pc->connection = NULL;
477
478 now = ngx_time();
479 hcf = hp->conf;
480
481 points = hcf->points;
482 point = &points->point[0];
483
484 for ( ;; ) {
485     server = point[hp->hash % points->number].server;
486
487     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, pc->log, 0,
488                 "consistent hash peer:%uD, server:\"%V\\\"",
489                 hp->hash, server);
490
491     best = NULL;
492     total = 0;
493
494     for (i = 0; i < hp->rrp.peers->number; i++) {
495
496         n = i / (8 * sizeof(uintptr_t));
497         m = (uintptr_t) 1 << i % (8 * sizeof(uintptr_t));
498
499         if (hp->rrp.tried[n] & m) {
500             continue;
501         }
502
503         peer = &hp->rrp.peers->peer[i];
504
505         if (peer->down) {
506             continue;
507         }
508
509         if (peer->server.len != server->len
510             || ngx_strncmp(peer->server.data, server->data, server->len)
511                 != 0)
512         {
513             continue;
514         }
515
516         if (peer->max_fails
517             && peer->fails >= peer->max_fails
518             && now - peer->checked <= peer->fail_timeout)
519         {
520             continue;
521         }
522
523         peer->current_weight += peer->effective_weight;
524         total += peer->effective_weight;
525
526         if (peer->effective_weight < peer->weight) {
527             peer->effective_weight++;
528         }
529
530         if (best == NULL || peer->current_weight > best->current_weight) {
531             best = peer;
532         }
533     }
534
535     if (best) {
536         best->current_weight -= total;
537
538         i = best - &hp->rrp.peers->peer[0];
539
540         hp->rrp.current = i;
541
542         n = i / (8 * sizeof(uintptr_t));
543         m = (uintptr_t) 1 << i % (8 * sizeof(uintptr_t));
544
545         hp->rrp.tried[n] |= m;
546
547         if (now - best->checked > best->fail_timeout) {
548             best->checked = now;
549         }
550
551         pc->sockaddr = best->sockaddr;

```

```

552         pc->socklen = best->socklen;
553         pc->name = &best->name;
554
555         return NGX_OK;
556     }
557
558     hp->hash++;
559     hp->tries++;
560
561     if (hp->tries >= points->number) {
562         return NGX_BUSY;
563     }
564 }
565 }
566
567
568 static void *
569 ngx_http_upstream_hash_create_conf(ngx_conf_t *cf)
570 {
571     ngx_http_upstream_hash_srv_conf_t *conf;
572
573     conf = ngx_palloc(cf->pool, sizeof(ngx_http_upstream_hash_srv_conf_t));
574     if (conf == NULL) {
575         return NULL;
576     }
577
578     conf->points = NULL;
579
580     return conf;
581 }
582
583
584 static char *
585 ngx_http_upstream_hash(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
586 {
587     ngx_http_upstream_hash_srv_conf_t *hcf = conf;
588
589     ngx_str_t *value;
590     ngx_http_upstream_srv_conf_t *uscf;
591     ngx_http_compile_complex_value_t ccv;
592
593     value = cf->args->elts;
594
595     ngx_memzero(&ccv, sizeof(ngx_http_compile_complex_value_t));
596
597     ccv.cf = cf;
598     ccv.value = &value[1];
599     ccv.complex_value = &hcf->key;
600
601     if (ngx_http_compile_complex_value(&ccv) != NGX_OK) {
602         return NGX_CONF_ERROR;
603     }
604
605     uscf = ngx_http_conf_get_module_srv_conf(cf, ngx_http_upstream_module);
606
607     if (uscf->peer.init_upstream) {
608         ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
609             "load balancing method redefined");
610     }
611
612     uscf->flags = NGX_HTTP_UPSTREAM_CREATE
613         | NGX_HTTP_UPSTREAM_WEIGHT
614         | NGX_HTTP_UPSTREAM_MAX_FAILS
615         | NGX_HTTP_UPSTREAM_FAIL_TIMEOUT
616         | NGX_HTTP_UPSTREAM_DOWN;
617
618     if (cf->args->nelts == 2) {
619         uscf->peer.init_upstream = ngx_http_upstream_init_hash;
620     }
621     else if (ngx_strcmp(value[2].data, "consistent") == 0) {
622         uscf->peer.init_upstream = ngx_http_upstream_init_chash;
623     }
624     else {
625         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
626             "invalid parameter \"%V\"", &value[2]);
627         return NGX_CONF_ERROR;

```

```
628     }  
629  
630     return NGX\_CONF\_OK;  
631 }
```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_upstream\_ip\_hash\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_upstream\\_ip\\_hash\\_commands](#)
- [ngx\\_http\\_upstream\\_ip\\_hash\\_module](#)
- [ngx\\_http\\_upstream\\_ip\\_hash\\_module\\_ctx](#)
- [ngx\\_http\\_upstream\\_ip\\_hash\\_pseudo\\_addr](#)

## Data types defined

- [ngx\\_http\\_upstream\\_ip\\_hash\\_peer\\_data\\_t](#)

## Functions defined

- [ngx\\_http\\_upstream\\_get\\_ip\\_hash\\_peer](#)
- [ngx\\_http\\_upstream\\_init\\_ip\\_hash](#)
- [ngx\\_http\\_upstream\\_init\\_ip\\_hash\\_peer](#)
- [ngx\\_http\\_upstream\\_ip\\_hash](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 typedef struct {
14     /* the round robin data must be first */
15     ngx_http_upstream_rr_peer_data_t  rrp;
16
17     ngx_uint_t                        hash;
18
19     u_char                             addrlen;
20     u_char                             *addr;
21
22     u_char                             tries;
23
24     ngx_event_get_peer_pt              get_rr_peer;
25 } ngx_http_upstream_ip_hash_peer_data_t;
26
27
28 static ngx_int_t ngx_http_upstream_init_ip_hash_peer(ngx_http_request_t *r,
29     ngx_http_upstream_srv_conf_t *us);
30 static ngx_int_t ngx_http_upstream_get_ip_hash_peer(ngx_peer_connection_t *pc,
31     void *data);
32 static char *ngx_http_upstream_ip_hash(ngx_conf_t *cf, ngx_command_t *cmd,
33     void *conf);
34
35
36 static ngx_command_t  ngx_http_upstream_ip_hash_commands[] = {
37
```

```

38     { ngx_string("ip_hash"),
39       NGX_HTTP_UPS_CONF|NGX_CONF_NOARGS,
40       ngx_http_upstream_ip_hash,
41       0,
42       0,
43       NULL },
44
45     ngx_null_command
46 };
47
48
49 static ngx_http_module_t  ngx_http_upstream_ip_hash_module_ctx = {
50     NULL,                               /* preconfiguration */
51     NULL,                               /* postconfiguration */
52
53     NULL,                               /* create main configuration */
54     NULL,                               /* init main configuration */
55
56     NULL,                               /* create server configuration */
57     NULL,                               /* merge server configuration */
58
59     NULL,                               /* create location configuration */
60     NULL,                               /* merge location configuration */
61 };
62
63
64 ngx_module_t  ngx_http_upstream_ip_hash_module = {
65     NGX_MODULE_V1,
66     &ngx_http_upstream_ip_hash_module_ctx, /* module context */
67     ngx_http_upstream_ip_hash_commands, /* module directives */
68     NGX_HTTP_MODULE,                   /* module type */
69     NULL,                               /* init master */
70     NULL,                               /* init module */
71     NULL,                               /* init process */
72     NULL,                               /* init thread */
73     NULL,                               /* exit thread */
74     NULL,                               /* exit process */
75     NULL,                               /* exit master */
76     NGX_MODULE_V1_PADDING
77 };
78
79
80 static u_char  ngx_http_upstream_ip_hash_pseudo_addr[3];
81
82
83 static ngx_int_t
84 ngx_http_upstream_init_ip_hash(ngx_conf_t *cf, ngx_http_upstream_srv_conf_t *us)
85 {
86     if (ngx_http_upstream_init_round_robin(cf, us) != NGX_OK) {
87         return NGX_ERROR;
88     }
89
90     us->peer.init = ngx_http_upstream_init_ip_hash_peer;
91
92     return NGX_OK;
93 }
94
95
96 static ngx_int_t
97 ngx_http_upstream_init_ip_hash_peer(ngx_http_request_t *r,
98     ngx_http_upstream_srv_conf_t *us)
99 {
100     struct sockaddr_in          *sin;
101     #if (NGX_HAVE_INET6)
102     struct sockaddr_in6        *sin6;
103     #endif
104     ngx_http_upstream_ip_hash_peer_data_t  *iphp;
105
106     iphp = ngx_palloc(r->pool, sizeof(ngx_http_upstream_ip_hash_peer_data_t));
107     if (iphp == NULL) {
108         return NGX_ERROR;
109     }
110
111     r->upstream->peer.data = &iphp->rrp;
112
113     if (ngx_http_upstream_init_round_robin_peer(r, us) != NGX_OK) {

```

```

114     return NGX\_ERROR;
115 }
116
117 r->upstream->peer.get = ngx\_http\_upstream\_get\_ip\_hash\_peer;
118
119 switch (r->connection->sockaddr->sa_family) {
120
121     case AF_INET:
122         sin = (struct sockaddr_in *) r->connection->sockaddr;
123         iphp->addr = (u_char *) &sin->sin_addr.s_addr;
124         iphp->addrlen = 3;
125         break;
126
127     #if (NGX_HAVE_INET6)
128     case AF_INET6:
129         sin6 = (struct sockaddr_in6 *) r->connection->sockaddr;
130         iphp->addr = (u_char *) &sin6->sin6_addr.s6_addr;
131         iphp->addrlen = 16;
132         break;
133     #endif
134
135     default:
136         iphp->addr = ngx\_http\_upstream\_ip\_hash\_pseudo\_addr;
137         iphp->addrlen = 3;
138     }
139
140     iphp->hash = 89;
141     iphp->tries = 0;
142     iphp->get_rr_peer = ngx\_http\_upstream\_get\_round\_robin\_peer;
143
144     return NGX\_OK;
145 }
146
147
148 static ngx\_int\_t
149 ngx\_http\_upstream\_get\_ip\_hash\_peer(ngx\_peer\_connection\_t *pc, void *data)
150 {
151     ngx\_http\_upstream\_ip\_hash\_peer\_data\_t *iphp = data;
152
153     time_t                now;
154     ngx\_int\_t             w;
155     uintptr_t             m;
156     ngx\_uint\_t           i, n, p, hash;
157     ngx\_http\_upstream\_rr\_peer\_t *peer;
158
159     ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, pc->log, 0,
160                 "get ip hash peer, try: %ui", pc->tries);
161
162     /* TODO: cached */
163
164     if (iphp->tries > 20 || iphp->rrp.peers->single) {
165         return iphp->get_rr_peer(pc, &iphp->rrp);
166     }
167
168     now = ngx\_time();
169
170     pc->cached = 0;
171     pc->connection = NULL;
172
173     hash = iphp->hash;
174
175     for ( ;; ) {
176
177         for (i = 0; i < (ngx\_uint\_t) iphp->addrlen; i++) {
178             hash = (hash * 113 + iphp->addr[i]) % 6271;
179         }
180
181         if (!iphp->rrp.peers->weighted) {
182             p = hash % iphp->rrp.peers->number;
183
184         } else {
185             w = hash % iphp->rrp.peers->total_weight;
186
187             for (i = 0; i < iphp->rrp.peers->number; i++) {
188                 w -= iphp->rrp.peers->peer[i].weight;
189                 if (w < 0) {

```



```

190         break;
191     }
192 }
193
194     p = i;
195 }
196
197     n = p / (8 * sizeof(uintptr_t));
198     m = (uintptr_t) 1 << p % (8 * sizeof(uintptr_t));
199
200     if (iphp->rrp.tried[n] & m) {
201         goto next;
202     }
203
204     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, pc->log, 0,
205                  "get ip hash peer, hash: %ui %04XA", p, m);
206
207     peer = &iphp->rrp.peers->peer[p];
208
209     /* ngx_lock_mutex(iphp->rrp.peers->mutex); */
210
211     if (peer->down) {
212         goto next_try;
213     }
214
215     if (peer->max_fails
216         && peer->fails >= peer->max_fails
217         && now - peer->checked <= peer->fail_timeout)
218     {
219         goto next_try;
220     }
221
222     break;
223
224 next_try:
225
226     iphp->rrp.tried[n] |= m;
227
228     /* ngx_unlock_mutex(iphp->rrp.peers->mutex); */
229
230     pc->tries--;
231
232 next:
233
234     if (++iphp->tries > 20) {
235         return iphp->get_rr_peer(pc, &iphp->rrp);
236     }
237 }
238
239 iphp->rrp.current = p;
240
241 pc->sockaddr = peer->sockaddr;
242 pc->socklen = peer->socklen;
243 pc->name = &peer->name;
244
245 if (now - peer->checked > peer->fail_timeout) {
246     peer->checked = now;
247 }
248
249 /* ngx_unlock_mutex(iphp->rrp.peers->mutex); */
250
251 iphp->rrp.tried[n] |= m;
252 iphp->hash = hash;
253
254 return NGX_OK;
255 }
256
257
258 static char *
259 ngx_http_upstream_ip_hash(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
260 {
261     ngx_http_upstream_srv_conf_t *uscf;
262
263     uscf = ngx_http_conf_get_module_srv_conf(cf, ngx_http_upstream_module);
264
265     if (uscf->peer.init_upstream) {

```

```
266     ngx_conf_log_error(NGX_LOG_WARN, cf, 0,  
267         "load balancing method redefined");  
268 }  
269  
270 uscf->peer.init_upstream = ngx_http_upstream_init_ip_hash;  
271  
272 uscf->flags = NGX_HTTP_UPSTREAM_CREATE  
273     |NGX_HTTP_UPSTREAM_WEIGHT  
274     |NGX_HTTP_UPSTREAM_MAX_FAILS  
275     |NGX_HTTP_UPSTREAM_FAIL_TIMEOUT  
276     |NGX_HTTP_UPSTREAM_DOWN;  
277  
278 return NGX_CONF_OK;  
279 }
```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_upstream\_keepalive\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_upstream\\_keepalive\\_commands](#)
- [ngx\\_http\\_upstream\\_keepalive\\_module](#)
- [ngx\\_http\\_upstream\\_keepalive\\_module\\_ctx](#)

## Data types defined

- [ngx\\_http\\_upstream\\_keepalive\\_cache\\_t](#)
- [ngx\\_http\\_upstream\\_keepalive\\_peer\\_data\\_t](#)
- [ngx\\_http\\_upstream\\_keepalive\\_srv\\_conf\\_t](#)

## Functions defined

- [ngx\\_http\\_upstream\\_free\\_keepalive\\_peer](#)
- [ngx\\_http\\_upstream\\_get\\_keepalive\\_peer](#)
- [ngx\\_http\\_upstream\\_init\\_keepalive](#)
- [ngx\\_http\\_upstream\\_init\\_keepalive\\_peer](#)
- [ngx\\_http\\_upstream\\_keepalive](#)
- [ngx\\_http\\_upstream\\_keepalive\\_close](#)
- [ngx\\_http\\_upstream\\_keepalive\\_close\\_handler](#)
- [ngx\\_http\\_upstream\\_keepalive\\_create\\_conf](#)
- [ngx\\_http\\_upstream\\_keepalive\\_dummy\\_handler](#)
- [ngx\\_http\\_upstream\\_keepalive\\_save\\_session](#)
- [ngx\\_http\\_upstream\\_keepalive\\_set\\_session](#)

## Source code

```
1
2 /*
3  * Copyright (C) Maxim Dounin
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 typedef struct {
14     ngx_uint_t                max_cached;
15
16     ngx_queue_t               cache;
17     ngx_queue_t               free;
18
19     ngx_http_upstream_init_pt original_init_upstream;
```

```

20     ngx_http_upstream_init_peer_pt    original_init_peer;
21
22 } ngx_http_upstream_keepalive_srv_conf_t;
23
24
25 typedef struct {
26     ngx_http_upstream_keepalive_srv_conf_t  *conf;
27
28     ngx_http_upstream_t                    *upstream;
29
30     void                                    *data;
31
32     ngx_event_get_peer_pt                  original_get_peer;
33     ngx_event_free_peer_pt                original_free_peer;
34
35 #if (NGX_HTTP_SSL)
36     ngx_event_set_peer_session_pt         original_set_session;
37     ngx_event_save_peer_session_pt       original_save_session;
38 #endif
39
40 } ngx_http_upstream_keepalive_peer_data_t;
41
42
43 typedef struct {
44     ngx_http_upstream_keepalive_srv_conf_t  *conf;
45
46     ngx_queue_t                               queue;
47     ngx_connection_t                         *connection;
48
49     socklen_t                                 socklen;
50     u_char                                    sockaddr[NGX_SOCKADDRLEN];
51
52 } ngx_http_upstream_keepalive_cache_t;
53
54
55 static ngx_int_t ngx_http_upstream_init_keepalive_peer(ngx_http_request_t *r,
56     ngx_http_upstream_srv_conf_t *us);
57 static ngx_int_t ngx_http_upstream_get_keepalive_peer(ngx_peer_connection_t *pc,
58     void *data);
59 static void ngx_http_upstream_free_keepalive_peer(ngx_peer_connection_t *pc,
60     void *data, ngx_uint_t state);
61
62 static void ngx_http_upstream_keepalive_dummy_handler(ngx_event_t *ev);
63 static void ngx_http_upstream_keepalive_close_handler(ngx_event_t *ev);
64 static void ngx_http_upstream_keepalive_close(ngx_connection_t *c);
65
66
67 #if (NGX_HTTP_SSL)
68 static ngx_int_t ngx_http_upstream_keepalive_set_session(
69     ngx_peer_connection_t *pc, void *data);
70 static void ngx_http_upstream_keepalive_save_session(ngx_peer_connection_t *pc,
71     void *data);
72 #endif
73
74 static void *ngx_http_upstream_keepalive_create_conf(ngx_conf_t *cf);
75 static char *ngx_http_upstream_keepalive(ngx_conf_t *cf, ngx_command_t *cmd,
76     void *conf);
77
78
79 static ngx_command_t  ngx_http_upstream_keepalive_commands[] = {
80
81     { ngx_string("keepalive"),
82       NGX_HTTP_UPS_CONF|NGX_CONF_TAKE1,
83       ngx_http_upstream_keepalive,
84       NGX_HTTP_SRV_CONF_OFFSET,
85       0,
86       NULL },
87
88     ngx_null_command
89 };
90
91
92 static ngx_http_module_t  ngx_http_upstream_keepalive_module_ctx = {
93     NULL,
94     NULL,
95     /* preconfiguration */
96     /* postconfiguration */

```

```

96     NULL,                                /* create main configuration */
97     NULL,                                /* init main configuration */
98
99     ngx_http_upstream_keepalive_create_conf, /* create server configuration */
100    NULL,                                /* merge server configuration */
101
102    NULL,                                /* create location configuration */
103    NULL,                                /* merge location configuration */
104 };
105
106
107 ngx_module_t ngx_http_upstream_keepalive_module = {
108     NGX_MODULE_V1,
109     &ngx_http_upstream_keepalive_module_ctx, /* module context */
110     ngx_http_upstream_keepalive_commands, /* module directives */
111     NGX_HTTP_MODULE, /* module type */
112     NULL, /* init master */
113     NULL, /* init module */
114     NULL, /* init process */
115     NULL, /* init thread */
116     NULL, /* exit thread */
117     NULL, /* exit process */
118     NULL, /* exit master */
119     NGX_MODULE_V1_PADDING
120 };
121
122
123 static ngx_int_t
124 ngx_http_upstream_init_keepalive(ngx_conf_t *cf,
125     ngx_http_upstream_srv_conf_t *us)
126 {
127     ngx_uint_t i;
128     ngx_http_upstream_keepalive_srv_conf_t *kcf;
129     ngx_http_upstream_keepalive_cache_t *cached;
130
131     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, cf->log, 0,
132         "init keepalive");
133
134     kcf = ngx_http_conf_upstream_srv_conf(us,
135         ngx_http_upstream_keepalive_module);
136
137     if (kcf->original_init_upstream(cf, us) != NGX_OK) {
138         return NGX_ERROR;
139     }
140
141     kcf->original_init_peer = us->peer.init;
142
143     us->peer.init = ngx_http_upstream_init_keepalive_peer;
144
145     /* allocate cache items and add to free queue */
146
147     cached = ngx_palloc(cf->pool,
148         sizeof(ngx_http_upstream_keepalive_cache_t) * kcf->max_cached);
149     if (cached == NULL) {
150         return NGX_ERROR;
151     }
152
153     ngx_queue_init(&kcf->cache);
154     ngx_queue_init(&kcf->free);
155
156     for (i = 0; i < kcf->max_cached; i++) {
157         ngx_queue_insert_head(&kcf->free, &cached[i].queue);
158         cached[i].conf = kcf;
159     }
160
161     return NGX_OK;
162 }
163
164
165 static ngx_int_t
166 ngx_http_upstream_init_keepalive_peer(ngx_http_request_t *r,
167     ngx_http_upstream_srv_conf_t *us)
168 {
169     ngx_http_upstream_keepalive_peer_data_t *kp;
170     ngx_http_upstream_keepalive_srv_conf_t *kcf;
171

```

```

172 ngx\_log\_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
173     "init keepalive peer");
174
175 kcf = ngx\_http\_conf\_upstream\_srv\_conf(us,
176     ngx\_http\_upstream\_keepalive\_module);
177
178 kp = ngx\_palloc(r->pool, sizeof(ngx\_http\_upstream\_keepalive\_peer\_data\_t));
179 if (kp == NULL) {
180     return NGX\_ERROR;
181 }
182
183 if (kcf->original_init_peer(r, us) != NGX\_OK) {
184     return NGX\_ERROR;
185 }
186
187 kp->conf = kcf;
188 kp->upstream = r->upstream;
189 kp->data = r->upstream->peer.data;
190 kp->original_get_peer = r->upstream->peer.get;
191 kp->original_free_peer = r->upstream->peer.free;
192
193 r->upstream->peer.data = kp;
194 r->upstream->peer.get = ngx\_http\_upstream\_get\_keepalive\_peer;
195 r->upstream->peer.free = ngx\_http\_upstream\_free\_keepalive\_peer;
196
197 #if (NGX\_HTTP\_SSL)
198 kp->original_set_session = r->upstream->peer.set_session;
199 kp->original_save_session = r->upstream->peer.save_session;
200 r->upstream->peer.set_session = ngx\_http\_upstream\_keepalive\_set\_session;
201 r->upstream->peer.save_session = ngx\_http\_upstream\_keepalive\_save\_session;
202 #endif
203
204     return NGX\_OK;
205 }
206
207
208 static ngx\_int\_t
209 ngx\_http\_upstream\_get\_keepalive\_peer(ngx\_peer\_connection\_t *pc, void *data)
210 {
211     ngx\_http\_upstream\_keepalive\_peer\_data\_t *kp = data;
212     ngx\_http\_upstream\_keepalive\_cache\_t *item;
213
214     ngx\_int\_t rc;
215     ngx\_queue\_t *q, *cache;
216     ngx\_connection\_t *c;
217
218     ngx\_log\_debug0(NGX_LOG_DEBUG_HTTP, pc->log, 0,
219         "get keepalive peer");
220
221     /* ask balancer */
222
223     rc = kp->original_get_peer(pc, kp->data);
224
225     if (rc != NGX\_OK) {
226         return rc;
227     }
228
229     /* search cache for suitable connection */
230
231     cache = &kp->conf->cache;
232
233     for (q = ngx\_queue\_head(cache);
234         q != ngx\_queue\_sentinel(cache);
235         q = ngx\_queue\_next(q))
236     {
237         item = ngx\_queue\_data(q, ngx\_http\_upstream\_keepalive\_cache\_t, queue);
238         c = item->connection;
239
240         if (ngx\_memn2cmp((u_char *) &item->sockaddr, (u_char *) pc->sockaddr,
241             item->socklen, pc->socklen)
242             == 0)
243         {
244             ngx\_queue\_remove(q);
245             ngx\_queue\_insert\_head(&kp->conf->free, q);
246
247             ngx\_log\_debug1(NGX_LOG_DEBUG_HTTP, pc->log, 0,

```

```

248         "get keepalive peer: using connection %p", c);
249
250     c->idle = 0;
251     c->sent = 0;
252     c->log = pc->log;
253     c->read->log = pc->log;
254     c->write->log = pc->log;
255     c->pool->log = pc->log;
256
257     pc->connection = c;
258     pc->cached = 1;
259
260     return NGX_DONE;
261 }
262 }
263
264 return NGX_OK;
265 }
266
267 static void
268 ngx_http_upstream_free_keepalive_peer(ngx_peer_connection_t *pc, void *data,
269 ngx_uint_t state)
270 {
271     ngx_http_upstream_keepalive_peer_data_t *kp = data;
272     ngx_http_upstream_keepalive_cache_t *item;
273
274     ngx_queue_t *q;
275     ngx_connection_t *c;
276     ngx_http_upstream_t *u;
277
278     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, pc->log, 0,
279         "free keepalive peer");
280
281     /* cache valid connections */
282
283     u = kp->upstream;
284     c = pc->connection;
285
286     if (state & NGX_PEER_FAILED
287         || c == NULL
288         || c->read->eof
289         || c->read->error
290         || c->read->timedout
291         || c->write->error
292         || c->write->timedout)
293     {
294         goto invalid;
295     }
296
297     if (!u->keepalive) {
298         goto invalid;
299     }
300
301     if (ngx_handle_read_event(c->read, 0) != NGX_OK) {
302         goto invalid;
303     }
304
305     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, pc->log, 0,
306         "free keepalive peer: saving connection %p", c);
307
308     if (ngx_queue_empty(&kp->conf->free)) {
309
310         q = ngx_queue_last(&kp->conf->cache);
311         ngx_queue_remove(q);
312
313         item = ngx_queue_data(q, ngx_http_upstream_keepalive_cache_t, queue);
314
315         ngx_http_upstream_keepalive_close(item->connection);
316
317     } else {
318         q = ngx_queue_head(&kp->conf->free);
319         ngx_queue_remove(q);
320
321         item = ngx_queue_data(q, ngx_http_upstream_keepalive_cache_t, queue);
322     }
323 }

```

```

324     item->connection = c;
325     ngx_queue_insert_head(&kp->conf->cache, q);
326
327
328     pc->connection = NULL;
329
330     if (c->read->timer_set) {
331         ngx_del_timer(c->read);
332     }
333     if (c->write->timer_set) {
334         ngx_del_timer(c->write);
335     }
336
337     c->write->handler = ngx_http_upstream_keepalive_dummy_handler;
338     c->read->handler = ngx_http_upstream_keepalive_close_handler;
339
340     c->data = item;
341     c->idle = 1;
342     c->log = ngx_cycle->log;
343     c->read->log = ngx_cycle->log;
344     c->write->log = ngx_cycle->log;
345     c->pool->log = ngx_cycle->log;
346
347     item->socklen = pc->socklen;
348     ngx_memcpy(&item->sockaddr, pc->sockaddr, pc->socklen);
349
350     if (c->read->ready) {
351         ngx_http_upstream_keepalive_close_handler(c->read);
352     }
353
354 invalid:
355
356     kp->original_free_peer(pc, kp->data, state);
357 }
358
359
360 static void
361 ngx_http_upstream_keepalive_dummy_handler(ngx_event_t *ev)
362 {
363     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, ev->log, 0,
364                 "keepalive dummy handler");
365 }
366
367
368 static void
369 ngx_http_upstream_keepalive_close_handler(ngx_event_t *ev)
370 {
371     ngx_http_upstream_keepalive_srv_conf_t *conf;
372     ngx_http_upstream_keepalive_cache_t *item;
373
374     int n;
375     char buf[1];
376     ngx_connection_t *c;
377
378     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, ev->log, 0,
379                 "keepalive close handler");
380
381     c = ev->data;
382
383     if (c->close) {
384         goto close;
385     }
386
387     n = recv(c->fd, buf, 1, MSG_PEEK);
388
389     if (n == -1 && ngx_socket_errno == NGX_EAGAIN) {
390         /* stale event */
391
392         if (ngx_handle_read_event(c->read, 0) != NGX_OK) {
393             goto close;
394         }
395
396         return;
397     }
398
399 close:

```



```

400     item = c->data;
401     conf = item->conf;
402
403     ngx\_http\_upstream\_keepalive\_close(c);
404
405     ngx\_queue\_remove(&item->queue);
406     ngx\_queue\_insert\_head(&conf->free, &item->queue);
407 }
408
409
410 static void
411 ngx\_http\_upstream\_keepalive\_close(ngx\_connection\_t *c)
412 {
413     #if (NGX_HTTP_SSL)
414
415     if (c->ssl) {
416         c->ssl->no_wait_shutdown = 1;
417         c->ssl->no_send_shutdown = 1;
418
419         if (ngx\_ssl\_shutdown(c) == NGX\_AGAIN) {
420             c->ssl->handler = ngx\_http\_upstream\_keepalive\_close;
421             return;
422         }
423     }
424 }
425 #endif
426
427     ngx\_destroy\_pool(c->pool);
428     ngx\_close\_connection(c);
429 }
430
431 #if (NGX_HTTP_SSL)
432
433 static ngx\_int\_t
434 ngx\_http\_upstream\_keepalive\_set\_session(ngx\_peer\_connection\_t *pc, void *data)
435 {
436     ngx\_http\_upstream\_keepalive\_peer\_data\_t *kp = data;
437
438     return kp->original_set_session(pc, kp->data);
439 }
440
441 static void
442 ngx\_http\_upstream\_keepalive\_save\_session(ngx\_peer\_connection\_t *pc, void *data)
443 {
444     ngx\_http\_upstream\_keepalive\_peer\_data\_t *kp = data;
445
446     kp->original_save_session(pc, kp->data);
447     return;
448 }
449 #endif
450
451 static void *
452 ngx\_http\_upstream\_keepalive\_create\_conf(ngx\_conf\_t *cf)
453 {
454     ngx\_http\_upstream\_keepalive\_srv\_conf\_t *conf;
455
456     conf = ngx\_palloc(cf->pool,
457                     sizeof(ngx\_http\_upstream\_keepalive\_srv\_conf\_t));
458     if (conf == NULL) {
459         return NULL;
460     }
461
462     /*
463     * set by ngx\_palloc():
464     *
465     *     conf->original_init_upstream = NULL;
466     *     conf->original_init_peer = NULL;
467     */
468     conf->max_cached = 1;

```

```

476     return conf;
477 }
478 }
479
480
481 static char *
482 ngx_http_upstream_keepalive(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
483 {
484     ngx_http_upstream_srv_conf_t      *uscf;
485     ngx_http_upstream_keepalive_srv_conf_t *kcf = conf;
486
487     ngx_int_t      n;
488     ngx_str_t      *value;
489
490     uscf = ngx_http_conf_get_module_srv_conf(cf, ngx_http_upstream_module);
491
492     if (kcf->original_init_upstream) {
493         return "is duplicate";
494     }
495
496     kcf->original_init_upstream = uscf->peer.init_upstream
497         ? uscf->peer.init_upstream
498         : ngx_http_upstream_init_round_robin;
499
500     uscf->peer.init_upstream = ngx_http_upstream_init_keepalive;
501
502     /* read options */
503
504     value = cf->args->elts;
505
506     n = ngx_atoi(value[1].data, value[1].len);
507
508     if (n == NGX_ERROR || n == 0) {
509         ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
510             "invalid value \"%V\" in \"%V\" directive",
511             &value[1], &cmd->name);
512         return NGX_CONF_ERROR;
513     }
514
515     kcf->max_cached = n;
516
517     return NGX_CONF_OK;
518 }

```

[One Level Up](#)

[Top Level](#)

# src/http/modules/nginx\_http\_upstream\_least\_conn\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_http\\_upstream\\_least\\_conn\\_commands](#)
- [ngx\\_http\\_upstream\\_least\\_conn\\_module](#)
- [ngx\\_http\\_upstream\\_least\\_conn\\_module\\_ctx](#)

## Data types defined

- [ngx\\_http\\_upstream\\_lc\\_peer\\_data\\_t](#)
- [ngx\\_http\\_upstream\\_least\\_conn\\_conf\\_t](#)

## Functions defined

- [ngx\\_http\\_upstream\\_free\\_least\\_conn\\_peer](#)
- [ngx\\_http\\_upstream\\_get\\_least\\_conn\\_peer](#)
- [ngx\\_http\\_upstream\\_init\\_least\\_conn](#)
- [ngx\\_http\\_upstream\\_init\\_least\\_conn\\_peer](#)
- [ngx\\_http\\_upstream\\_least\\_conn](#)
- [ngx\\_http\\_upstream\\_least\\_conn\\_create\\_conf](#)

## Source code

```
1
2 /*
3  * Copyright (C) Maxim Dounin
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12
13 typedef struct {
14     ngx\_uint\_t *conns;
15 } ngx\_http\_upstream\_least\_conn\_conf\_t;
16
17
18 typedef struct {
19     /* the round robin data must be first */
20     ngx\_http\_upstream\_rr\_peer\_data\_t rrp;
21
22     ngx\_uint\_t *conns;
23
24     ngx\_event\_get\_peer\_pt get_rr_peer;
25     ngx\_event\_free\_peer\_pt free_rr_peer;
26 } ngx\_http\_upstream\_lc\_peer\_data\_t;
27
28
29 static ngx\_int\_t ngx\_http\_upstream\_init\_least\_conn\_peer(ngx\_http\_request\_t *r,
30     ngx\_http\_upstream\_srv\_conf\_t *us);
31 static ngx\_int\_t ngx\_http\_upstream\_get\_least\_conn\_peer(
32     ngx\_peer\_connection\_t *pc, void *data);
33 static void ngx\_http\_upstream\_free\_least\_conn\_peer(ngx\_peer\_connection\_t *pc,
```

```

34 void *data, ngx_uint_t state);
35 static void *ngx_http_upstream_least_conn_create_conf(ngx_conf_t *cf);
36 static char *ngx_http_upstream_least_conn(ngx_conf_t *cf, ngx_command_t *cmd,
37 void *conf);
38
39
40 static ngx_command_t ngx_http_upstream_least_conn_commands[] = {
41
42     { ngx_string("least_conn"),
43       NGX_HTTP_UPS_CONF|NGX_CONF_NOARGS,
44       ngx_http_upstream_least_conn,
45       0,
46       0,
47       NULL },
48
49     ngx_null_command
50 };
51
52
53 static ngx_http_module_t ngx_http_upstream_least_conn_module_ctx = {
54     NULL, /* preconfiguration */
55     NULL, /* postconfiguration */
56
57     NULL, /* create main configuration */
58     NULL, /* init main configuration */
59
60     ngx_http_upstream_least_conn_create_conf, /* create server configuration */
61     NULL, /* merge server configuration */
62
63     NULL, /* create location configuration */
64     NULL, /* merge location configuration */
65 };
66
67
68 ngx_module_t ngx_http_upstream_least_conn_module = {
69     NGX_MODULE_V1,
70     &ngx_http_upstream_least_conn_module_ctx, /* module context */
71     ngx_http_upstream_least_conn_commands, /* module directives */
72     NGX_HTTP_MODULE, /* module type */
73     NULL, /* init master */
74     NULL, /* init module */
75     NULL, /* init process */
76     NULL, /* init thread */
77     NULL, /* exit thread */
78     NULL, /* exit process */
79     NULL, /* exit master */
80     NGX_MODULE_V1_PADDING
81 };
82
83
84 static ngx_int_t
85 ngx_http_upstream_init_least_conn(ngx_conf_t *cf,
86 ngx_http_upstream_srv_conf_t *us)
87 {
88     ngx_uint_t n;
89     ngx_http_upstream_rr_peers_t *peers;
90     ngx_http_upstream_least_conn_conf_t *lcf;
91
92     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, cf->log, 0,
93 "init least conn");
94
95     if (ngx_http_upstream_init_round_robin(cf, us) != NGX_OK) {
96         return NGX_ERROR;
97     }
98
99     peers = us->peer.data;
100
101     n = peers->number;
102
103     if (peers->next) {
104         n += peers->next->number;
105     }
106
107     lcf = ngx_http_conf_upstream_srv_conf(us,
108 ngx_http_upstream_least_conn_module);
109

```

```

110     lcf->conns = ngx_palloc(cf->pool, sizeof(ngx_uint_t) * n);
111     if (lcf->conns == NULL) {
112         return NGX_ERROR;
113     }
114
115     us->peer.init = ngx_http_upstream_init_least_conn_peer;
116
117     return NGX_OK;
118 }
119
120
121 static ngx_int_t
122 ngx_http_upstream_init_least_conn_peer(ngx_http_request_t *r,
123     ngx_http_upstream_srv_conf_t *us)
124 {
125     ngx_http_upstream_lc_peer_data_t *lcp;
126     ngx_http_upstream_least_conn_conf_t *lcf;
127
128     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
129         "init least conn peer");
130
131     lcf = ngx_http_conf_upstream_srv_conf(us,
132         ngx_http_upstream_least_conn_module);
133
134     lcp = ngx_palloc(r->pool, sizeof(ngx_http_upstream_lc_peer_data_t));
135     if (lcp == NULL) {
136         return NGX_ERROR;
137     }
138
139     lcp->conns = lcf->conns;
140
141     r->upstream->peer.data = &lcp->rrp;
142
143     if (ngx_http_upstream_init_round_robin_peer(r, us) != NGX_OK) {
144         return NGX_ERROR;
145     }
146
147     r->upstream->peer.get = ngx_http_upstream_get_least_conn_peer;
148     r->upstream->peer.free = ngx_http_upstream_free_least_conn_peer;
149
150     lcp->get_rr_peer = ngx_http_upstream_get_round_robin_peer;
151     lcp->free_rr_peer = ngx_http_upstream_free_round_robin_peer;
152
153     return NGX_OK;
154 }
155
156
157 static ngx_int_t
158 ngx_http_upstream_get_least_conn_peer(ngx_peer_connection_t *pc, void *data)
159 {
160     ngx_http_upstream_lc_peer_data_t *lcp = data;
161
162     time_t                now;
163     uintptr_t             m;
164     ngx_int_t             rc, total;
165     ngx_uint_t            i, n, p, many;
166     ngx_http_upstream_rr_peer_t *peer, *best;
167     ngx_http_upstream_rr_peers_t *peers;
168
169     ngx_log_debug1(NGX_LOG_DEBUG_HTTP, pc->log, 0,
170         "get least conn peer, try: %ui", pc->tries);
171
172     if (lcp->rrp.peers->single) {
173         return lcp->get_rr_peer(pc, &lcp->rrp);
174     }
175
176     pc->cached = 0;
177     pc->connection = NULL;
178
179     now = ngx_time();
180
181     peers = lcp->rrp.peers;
182
183     best = NULL;
184     total = 0;
185

```

```

186 #if (NGX_SUPPRESS_WARN)
187     many = 0;
188     p = 0;
189 #endif
190
191     for (i = 0; i < peers->number; i++) {
192
193         n = i / (8 * sizeof(uintptr_t));
194         m = (uintptr_t) 1 << i % (8 * sizeof(uintptr_t));
195
196         if (lcp->rrp.tried[n] & m) {
197             continue;
198         }
199
200         peer = &peers->peer[i];
201
202         if (peer->down) {
203             continue;
204         }
205
206         if (peer->max_fails
207             && peer->fails >= peer->max_fails
208             && now - peer->checked <= peer->fail_timeout)
209         {
210             continue;
211         }
212
213         /*
214          * select peer with least number of connections; if there are
215          * multiple peers with the same number of connections, select
216          * based on round-robin
217          */
218
219         if (best == NULL
220             || lcp->conns[i] * best->weight < lcp->conns[p] * peer->weight)
221         {
222             best = peer;
223             many = 0;
224             p = i;
225
226         } else if (lcp->conns[i] * best->weight
227                 == lcp->conns[p] * peer->weight)
228         {
229             many = 1;
230         }
231     }
232
233     if (best == NULL) {
234         ngx_log_debug0(NGX_LOG_DEBUG_HTTP, pc->log, 0,
235                       "get least conn peer, no peer found");
236
237         goto failed;
238     }
239
240     if (many) {
241         ngx_log_debug0(NGX_LOG_DEBUG_HTTP, pc->log, 0,
242                       "get least conn peer, many");
243
244         for (i = p; i < peers->number; i++) {
245
246             n = i / (8 * sizeof(uintptr_t));
247             m = (uintptr_t) 1 << i % (8 * sizeof(uintptr_t));
248
249             if (lcp->rrp.tried[n] & m) {
250                 continue;
251             }
252
253             peer = &peers->peer[i];
254
255             if (peer->down) {
256                 continue;
257             }
258
259             if (lcp->conns[i] * best->weight != lcp->conns[p] * peer->weight) {
260                 continue;
261             }

```

```

262         if (peer->max_fails
263             && peer->fails >= peer->max_fails
264             && now - peer->checked <= peer->fail_timeout)
265         {
266             continue;
267         }
268
269         peer->current_weight += peer->effective_weight;
270         total += peer->effective_weight;
271
272         if (peer->effective_weight < peer->weight) {
273             peer->effective_weight++;
274         }
275
276         if (peer->current_weight > best->current_weight) {
277             best = peer;
278             p = i;
279         }
280     }
281 }
282 }
283
284 best->current_weight -= total;
285
286 if (now - best->checked > best->fail_timeout) {
287     best->checked = now;
288 }
289
290 pc->sockaddr = best->sockaddr;
291 pc->socklen = best->socklen;
292 pc->name = &best->name;
293
294 lcp->rrp.current = p;
295
296 n = p / (8 * sizeof(uintptr_t));
297 m = (uintptr_t) 1 << p % (8 * sizeof(uintptr_t));
298
299 lcp->rrp.tried[n] |= m;
300 lcp->conns[p]++;
301
302 return NGX_OK;
303
304 failed:
305
306 if (peers->next) {
307     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, pc->log, 0,
308                  "get least conn peer, backup servers");
309
310     lcp->conns += peers->number;
311
312     lcp->rrp.peers = peers->next;
313
314     n = (lcp->rrp.peers->number + (8 * sizeof(uintptr_t) - 1))
315         / (8 * sizeof(uintptr_t));
316
317     for (i = 0; i < n; i++) {
318         lcp->rrp.tried[i] = 0;
319     }
320
321     rc = ngx_http_upstream_get_least_conn_peer(pc, lcp);
322
323     if (rc != NGX_BUSY) {
324         return rc;
325     }
326 }
327
328 /* all peers failed, mark them as live for quick recovery */
329
330 for (i = 0; i < peers->number; i++) {
331     peers->peer[i].fails = 0;
332 }
333
334 pc->name = peers->name;
335
336 return NGX_BUSY;
337 }

```

```

338
339
340 static void
341 ngx_http_upstream_free_least_conn_peer(ngx_peer_connection_t *pc,
342 void *data, ngx_uint_t state)
343 {
344     ngx_http_upstream_lc_peer_data_t *lcp = data;
345
346     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, pc->log, 0,
347         "free least conn peer %ui %ui", pc->tries, state);
348
349     if (lcp->rrp.peers->single) {
350         lcp->free_rr_peer(pc, &lcp->rrp, state);
351         return;
352     }
353
354     lcp->conns[lcp->rrp.current]--;
355
356     lcp->free_rr_peer(pc, &lcp->rrp, state);
357 }
358
359
360 static void *
361 ngx_http_upstream_least_conn_create_conf(ngx_conf_t *cf)
362 {
363     ngx_http_upstream_least_conn_conf_t *conf;
364
365     conf = ngx_palloc(cf->pool,
366         sizeof(ngx_http_upstream_least_conn_conf_t));
367     if (conf == NULL) {
368         return NULL;
369     }
370
371     /*
372      * set by ngx_palloc():
373      *
374      *     conf->conns = NULL;
375      */
376
377     return conf;
378 }
379
380
381 static char *
382 ngx_http_upstream_least_conn(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
383 {
384     ngx_http_upstream_srv_conf_t *uscf;
385
386     uscf = ngx_http_conf_get_module_srv_conf(cf, ngx_http_upstream_module);
387
388     if (uscf->peer.init_upstream) {
389         ngx_conf_log_error(NGX_LOG_WARN, cf, 0,
390             "load balancing method redefined");
391     }
392
393     uscf->peer.init_upstream = ngx_http_upstream_init_least_conn;
394
395     uscf->flags = NGX_HTTP_UPSTREAM_CREATE
396         |NGX_HTTP_UPSTREAM_WEIGHT
397         |NGX_HTTP_UPSTREAM_MAX_FAILS
398         |NGX_HTTP_UPSTREAM_FAIL_TIMEOUT
399         |NGX_HTTP_UPSTREAM_DOWN
400         |NGX_HTTP_UPSTREAM_BACKUP;
401
402     return NGX_CONF_OK;
403 }

```

[One Level Up](#)

[Top Level](#)



## src/http/modules/nginx\_http\_uwsgi\_module.c - nginx-1.7.10

### Global variables defined

- [ngx\\_http\\_uwsgi\\_cache\\_headers](#)
- [ngx\\_http\\_uwsgi\\_commands](#)
- [ngx\\_http\\_uwsgi\\_hide\\_headers](#)
- [ngx\\_http\\_uwsgi\\_modifier\\_bounds](#)
- [ngx\\_http\\_uwsgi\\_module](#)
- [ngx\\_http\\_uwsgi\\_module](#)
- [ngx\\_http\\_uwsgi\\_module\\_ctx](#)
- [ngx\\_http\\_uwsgi\\_next\\_upstream\\_masks](#)
- [ngx\\_http\\_uwsgi\\_ssl\\_protocols](#)
- [ngx\\_http\\_uwsgi\\_temp\\_path](#)

### Data types defined

- [ngx\\_http\\_uwsgi\\_loc\\_conf\\_t](#)
- [ngx\\_http\\_uwsgi\\_main\\_conf\\_t](#)
- [ngx\\_http\\_uwsgi\\_params\\_t](#)

### Functions defined

- [ngx\\_http\\_uwsgi\\_abort\\_request](#)
- [ngx\\_http\\_uwsgi\\_cache](#)
- [ngx\\_http\\_uwsgi\\_cache\\_key](#)
- [ngx\\_http\\_uwsgi\\_create\\_key](#)
- [ngx\\_http\\_uwsgi\\_create\\_loc\\_conf](#)
- [ngx\\_http\\_uwsgi\\_create\\_main\\_conf](#)
- [ngx\\_http\\_uwsgi\\_create\\_request](#)
- [ngx\\_http\\_uwsgi\\_eval](#)
- [ngx\\_http\\_uwsgi\\_finalize\\_request](#)
- [ngx\\_http\\_uwsgi\\_handler](#)
- [ngx\\_http\\_uwsgi\\_init\\_params](#)
- [ngx\\_http\\_uwsgi\\_merge\\_loc\\_conf](#)
- [ngx\\_http\\_uwsgi\\_pass](#)
- [ngx\\_http\\_uwsgi\\_process\\_header](#)

- [ngx http uwsgi process status line](#)
- [ngx http uwsgi reinit request](#)
- [ngx http uwsgi set ssl](#)
- [ngx http uwsgi ssl password file](#)
- [ngx http uwsgi store](#)

## Source code

```

1
2 /*
3  * Copyright (C) Unbit S.a.s. 2009-2010
4  * Copyright (C) 2008 Manlio Perillo (manlio.perillo@gmail.com)
5  * Copyright (C) Igor Sysoev
6  * Copyright (C) Nginx, Inc.
7  */
8
9
10 #include <ngx_config.h>
11 #include <ngx_core.h>
12 #include <ngx_http.h>
13
14
15 typedef struct {
16     ngx_array_t          caches; /* ngx_http_file_cache_t * */
17 } ngx_http_uwsgi_main_conf_t;
18
19
20 typedef struct {
21     ngx_array_t          *flushes;
22     ngx_array_t          *lengths;
23     ngx_array_t          *values;
24     ngx_uint_t           number;
25     ngx_hash_t           hash;
26 } ngx_http_uwsgi_params_t;
27
28
29 typedef struct {
30     ngx_http_upstream_conf_t  upstream;
31
32     ngx_http_uwsgi_params_t  params;
33     #if (NGX_HTTP_CACHE)
34     ngx_http_uwsgi_params_t  params_cache;
35     #endif
36     ngx_array_t              *params_source;
37
38     ngx_array_t              *uwsgi_lengths;
39     ngx_array_t              *uwsgi_values;
40
41     #if (NGX_HTTP_CACHE)
42     ngx_http_complex_value_t  cache_key;
43     #endif
44
45     ngx_str_t                uwsgi_string;
46
47     ngx_uint_t               modifier1;
48     ngx_uint_t               modifier2;
49
50     #if (NGX_HTTP_SSL)
51     ngx_uint_t               ssl;
52     ngx_uint_t               ssl_protocols;
53     ngx_str_t                ssl_ciphers;
54     ngx_uint_t               ssl_verify_depth;
55     ngx_str_t                ssl_trusted_certificate;
56     ngx_str_t                ssl_crl;
57     ngx_str_t                ssl_certificate;
58     ngx_str_t                ssl_certificate_key;
59     ngx_array_t              *ssl_passwords;
60     #endif
61 } ngx_http_uwsgi_loc_conf_t;

```

```

62
63
64 static ngx_int_t ngx_http_uwsgi_eval(ngx_http_request_t *r,
65     ngx_http_uwsgi_loc_conf_t *uwcf);
66 static ngx_int_t ngx_http_uwsgi_create_request(ngx_http_request_t *r);
67 static ngx_int_t ngx_http_uwsgi_reinit_request(ngx_http_request_t *r);
68 static ngx_int_t ngx_http_uwsgi_process_status_line(ngx_http_request_t *r);
69 static ngx_int_t ngx_http_uwsgi_process_header(ngx_http_request_t *r);
70 static void ngx_http_uwsgi_abort_request(ngx_http_request_t *r);
71 static void ngx_http_uwsgi_finalize_request(ngx_http_request_t *r,
72     ngx_int_t rc);
73
74 static void *ngx_http_uwsgi_create_main_conf(ngx_conf_t *cf);
75 static void *ngx_http_uwsgi_create_loc_conf(ngx_conf_t *cf);
76 static char *ngx_http_uwsgi_merge_loc_conf(ngx_conf_t *cf, void *parent,
77     void *child);
78 static ngx_int_t ngx_http_uwsgi_init_params(ngx_conf_t *cf,
79     ngx_http_uwsgi_loc_conf_t *conf, ngx_http_uwsgi_params_t *params,
80     ngx_keyval_t *default_params);
81
82 static char *ngx_http_uwsgi_pass(ngx_conf_t *cf, ngx_command_t *cmd,
83     void *conf);
84 static char *ngx_http_uwsgi_store(ngx_conf_t *cf, ngx_command_t *cmd,
85     void *conf);
86
87 #if (NGX_HTTP_CACHE)
88 static ngx_int_t ngx_http_uwsgi_create_key(ngx_http_request_t *r);
89 static char *ngx_http_uwsgi_cache(ngx_conf_t *cf, ngx_command_t *cmd,
90     void *conf);
91 static char *ngx_http_uwsgi_cache_key(ngx_conf_t *cf, ngx_command_t *cmd,
92     void *conf);
93 #endif
94
95 #if (NGX_HTTP_SSL)
96 static char *ngx_http_uwsgi_ssl_password_file(ngx_conf_t *cf,
97     ngx_command_t *cmd, void *conf);
98 static ngx_int_t ngx_http_uwsgi_set_ssl(ngx_conf_t *cf,
99     ngx_http_uwsgi_loc_conf_t *uwcf);
100 #endif
101
102
103 static ngx_conf_num_bounds_t ngx_http_uwsgi_modifier_bounds = {
104     ngx_conf_check_num_bounds, 0, 255
105 };
106
107
108 static ngx_conf_bitmask_t ngx_http_uwsgi_next_upstream_masks[] = {
109     { ngx_string("error"), NGX_HTTP_UPSTREAM_FT_ERROR },
110     { ngx_string("timeout"), NGX_HTTP_UPSTREAM_FT_TIMEOUT },
111     { ngx_string("invalid_header"), NGX_HTTP_UPSTREAM_FT_INVALID_HEADER },
112     { ngx_string("http_500"), NGX_HTTP_UPSTREAM_FT_HTTP_500 },
113     { ngx_string("http_503"), NGX_HTTP_UPSTREAM_FT_HTTP_503 },
114     { ngx_string("http_403"), NGX_HTTP_UPSTREAM_FT_HTTP_403 },
115     { ngx_string("http_404"), NGX_HTTP_UPSTREAM_FT_HTTP_404 },
116     { ngx_string("updating"), NGX_HTTP_UPSTREAM_FT_UPDATING },
117     { ngx_string("off"), NGX_HTTP_UPSTREAM_FT_OFF },
118     { ngx_null_string, 0 }
119 };
120
121
122 #if (NGX_HTTP_SSL)
123
124 static ngx_conf_bitmask_t ngx_http_uwsgi_ssl_protocols[] = {
125     { ngx_string("SSLv2"), NGX_SSL_SSLV2 },
126     { ngx_string("SSLv3"), NGX_SSL_SSLV3 },
127     { ngx_string("TLSv1"), NGX_SSL_TLSV1 },
128     { ngx_string("TLSv1.1"), NGX_SSL_TLSV1_1 },
129     { ngx_string("TLSv1.2"), NGX_SSL_TLSV1_2 },
130     { ngx_null_string, 0 }
131 };
132
133 #endif
134
135
136 ngx_module_t ngx_http_uwsgi_module;
137

```

```

138
139 static ngx_command_t ngx_http_uwsgi_commands[] = {
140
141     { ngx_string("uwsgi_pass"),
142       NGX_HTTP_LOC_CONF|NGX_HTTP_LIF_CONF|NGX_CONF_TAKE1,
143       ngx_http_uwsgi_pass,
144       NGX_HTTP_LOC_CONF_OFFSET,
145       0,
146       NULL },
147
148     { ngx_string("uwsgi_modifier1"),
149       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
150       ngx_conf_set_num_slot,
151       NGX_HTTP_LOC_CONF_OFFSET,
152       offsetof(ngx_http_uwsgi_loc_conf_t, modifier1),
153       &ngx_http_uwsgi_modifier_bounds },
154
155     { ngx_string("uwsgi_modifier2"),
156       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
157       ngx_conf_set_num_slot,
158       NGX_HTTP_LOC_CONF_OFFSET,
159       offsetof(ngx_http_uwsgi_loc_conf_t, modifier2),
160       &ngx_http_uwsgi_modifier_bounds },
161
162     { ngx_string("uwsgi_store"),
163       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
164       ngx_http_uwsgi_store,
165       NGX_HTTP_LOC_CONF_OFFSET,
166       0,
167       NULL },
168
169     { ngx_string("uwsgi_store_access"),
170       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE123,
171       ngx_conf_set_access_slot,
172       NGX_HTTP_LOC_CONF_OFFSET,
173       offsetof(ngx_http_uwsgi_loc_conf_t, upstream.store_access),
174       NULL },
175
176     { ngx_string("uwsgi_buffering"),
177       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
178       ngx_conf_set_flag_slot,
179       NGX_HTTP_LOC_CONF_OFFSET,
180       offsetof(ngx_http_uwsgi_loc_conf_t, upstream.buffering),
181       NULL },
182
183     { ngx_string("uwsgi_ignore_client_abort"),
184       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
185       ngx_conf_set_flag_slot,
186       NGX_HTTP_LOC_CONF_OFFSET,
187       offsetof(ngx_http_uwsgi_loc_conf_t, upstream.ignore_client_abort),
188       NULL },
189
190     { ngx_string("uwsgi_bind"),
191       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
192       ngx_http_upstream_bind_set_slot,
193       NGX_HTTP_LOC_CONF_OFFSET,
194       offsetof(ngx_http_uwsgi_loc_conf_t, upstream.local),
195       NULL },
196
197     { ngx_string("uwsgi_connect_timeout"),
198       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
199       ngx_conf_set_msec_slot,
200       NGX_HTTP_LOC_CONF_OFFSET,
201       offsetof(ngx_http_uwsgi_loc_conf_t, upstream.connect_timeout),
202       NULL },
203
204     { ngx_string("uwsgi_send_timeout"),
205       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
206       ngx_conf_set_msec_slot,
207       NGX_HTTP_LOC_CONF_OFFSET,
208       offsetof(ngx_http_uwsgi_loc_conf_t, upstream.send_timeout),
209       NULL },
210
211     { ngx_string("uwsgi_buffer_size"),
212       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
213       ngx_conf_set_size_slot,

```

```

214     NGX_HTTP_LOC_CONF_OFFSET,
215     offsetof(ngx_http_uwsgi_loc_conf_t, upstream.buffer_size),
216     NULL },
217
218 { ngx_string("uwsgi_pass_request_headers"),
219   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
220   ngx_conf_set_flag_slot,
221   NGX_HTTP_LOC_CONF_OFFSET,
222   offsetof(ngx_http_uwsgi_loc_conf_t, upstream.pass_request_headers),
223   NULL },
224
225 { ngx_string("uwsgi_pass_request_body"),
226   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
227   ngx_conf_set_flag_slot,
228   NGX_HTTP_LOC_CONF_OFFSET,
229   offsetof(ngx_http_uwsgi_loc_conf_t, upstream.pass_request_body),
230   NULL },
231
232 { ngx_string("uwsgi_intercept_errors"),
233   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
234   ngx_conf_set_flag_slot,
235   NGX_HTTP_LOC_CONF_OFFSET,
236   offsetof(ngx_http_uwsgi_loc_conf_t, upstream.intercept_errors),
237   NULL },
238
239 { ngx_string("uwsgi_read_timeout"),
240   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
241   ngx_conf_set_msec_slot,
242   NGX_HTTP_LOC_CONF_OFFSET,
243   offsetof(ngx_http_uwsgi_loc_conf_t, upstream.read_timeout),
244   NULL },
245
246 { ngx_string("uwsgi_buffers"),
247   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE2,
248   ngx_conf_set_bufs_slot,
249   NGX_HTTP_LOC_CONF_OFFSET,
250   offsetof(ngx_http_uwsgi_loc_conf_t, upstream.bufs),
251   NULL },
252
253 { ngx_string("uwsgi_busy_buffers_size"),
254   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
255   ngx_conf_set_size_slot,
256   NGX_HTTP_LOC_CONF_OFFSET,
257   offsetof(ngx_http_uwsgi_loc_conf_t, upstream.busy_buffers_size_conf),
258   NULL },
259
260 { ngx_string("uwsgi_force_ranges"),
261   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_FLAG,
262   ngx_conf_set_flag_slot,
263   NGX_HTTP_LOC_CONF_OFFSET,
264   offsetof(ngx_http_uwsgi_loc_conf_t, upstream.force_ranges),
265   NULL },
266
267 { ngx_string("uwsgi_limit_rate"),
268   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
269   ngx_conf_set_size_slot,
270   NGX_HTTP_LOC_CONF_OFFSET,
271   offsetof(ngx_http_uwsgi_loc_conf_t, upstream.limit_rate),
272   NULL },
273
274 #if (NGX_HTTP_CACHE)
275
276 { ngx_string("uwsgi_cache"),
277   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
278   ngx_http_uwsgi_cache,
279   NGX_HTTP_LOC_CONF_OFFSET,
280   0,
281   NULL },
282
283 { ngx_string("uwsgi_cache_key"),
284   NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
285   ngx_http_uwsgi_cache_key,
286   NGX_HTTP_LOC_CONF_OFFSET,
287   0,
288   NULL },
289

```

```

290 { ngx_string("uwsgi_cache_path"),
291     NGX\_HTTP\_MAIN\_CONF|NGX\_CONF\_2MORE,
292     ngx_http_file_cache_set_slot,
293     NGX\_HTTP\_MAIN\_CONF\_OFFSET,
294     offsetof(ngx\_http\_uwsgi\_main\_conf\_t, caches),
295     &ngx_http_uwsgi_module },
296
297 { ngx_string("uwsgi_cache_bypass"),
298     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_1MORE,
299     ngx_http_set_predicate_slot,
300     NGX\_HTTP\_LOC\_CONF\_OFFSET,
301     offsetof(ngx\_http\_uwsgi\_loc\_conf\_t, upstream.cache_bypass),
302     NULL },
303
304 { ngx_string("uwsgi_no_cache"),
305     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_1MORE,
306     ngx_http_set_predicate_slot,
307     NGX\_HTTP\_LOC\_CONF\_OFFSET,
308     offsetof(ngx\_http\_uwsgi\_loc\_conf\_t, upstream.no_cache),
309     NULL },
310
311 { ngx_string("uwsgi_cache_valid"),
312     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_1MORE,
313     ngx_http_file_cache_valid_set_slot,
314     NGX\_HTTP\_LOC\_CONF\_OFFSET,
315     offsetof(ngx\_http\_uwsgi\_loc\_conf\_t, upstream.cache_valid),
316     NULL },
317
318 { ngx_string("uwsgi_cache_min_uses"),
319     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
320     ngx_conf_set_num_slot,
321     NGX\_HTTP\_LOC\_CONF\_OFFSET,
322     offsetof(ngx\_http\_uwsgi\_loc\_conf\_t, upstream.cache_min_uses),
323     NULL },
324
325 { ngx_string("uwsgi_cache_use_stale"),
326     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_1MORE,
327     ngx_conf_set_bitmask_slot,
328     NGX\_HTTP\_LOC\_CONF\_OFFSET,
329     offsetof(ngx\_http\_uwsgi\_loc\_conf\_t, upstream.cache_use_stale),
330     &ngx_http_uwsgi_next_upstream_masks },
331
332 { ngx_string("uwsgi_cache_methods"),
333     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_1MORE,
334     ngx_conf_set_bitmask_slot,
335     NGX\_HTTP\_LOC\_CONF\_OFFSET,
336     offsetof(ngx\_http\_uwsgi\_loc\_conf\_t, upstream.cache_methods),
337     &ngx_http_upstream_cache_method_mask },
338
339 { ngx_string("uwsgi_cache_lock"),
340     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_FLAG,
341     ngx_conf_set_flag_slot,
342     NGX\_HTTP\_LOC\_CONF\_OFFSET,
343     offsetof(ngx\_http\_uwsgi\_loc\_conf\_t, upstream.cache_lock),
344     NULL },
345
346 { ngx_string("uwsgi_cache_lock_timeout"),
347     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
348     ngx_conf_set_msec_slot,
349     NGX\_HTTP\_LOC\_CONF\_OFFSET,
350     offsetof(ngx\_http\_uwsgi\_loc\_conf\_t, upstream.cache_lock_timeout),
351     NULL },
352
353 { ngx_string("uwsgi_cache_lock_age"),
354     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
355     ngx_conf_set_msec_slot,
356     NGX\_HTTP\_LOC\_CONF\_OFFSET,
357     offsetof(ngx\_http\_uwsgi\_loc\_conf\_t, upstream.cache_lock_age),
358     NULL },
359
360 { ngx_string("uwsgi_cache_revalidate"),
361     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_FLAG,
362     ngx_conf_set_flag_slot,
363     NGX\_HTTP\_LOC\_CONF\_OFFSET,
364     offsetof(ngx\_http\_uwsgi\_loc\_conf\_t, upstream.cache_revalidate),
365     NULL },

```

```

366
367 #endif
368
369 { ngx_string("uwsgi_temp_path"),
370     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1234,
371     ngx\_conf\_set\_path\_slot,
372     NGX\_HTTP\_LOC\_CONF\_OFFSET,
373     offsetof(ngx\_http\_uwsgi\_loc\_conf\_t, upstream.temp_path),
374     NULL },
375
376 { ngx_string("uwsgi_max_temp_file_size"),
377     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
378     ngx\_conf\_set\_size\_slot,
379     NGX\_HTTP\_LOC\_CONF\_OFFSET,
380     offsetof(ngx\_http\_uwsgi\_loc\_conf\_t, upstream.max_temp_file_size_conf),
381     NULL },
382
383 { ngx_string("uwsgi_temp_file_write_size"),
384     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
385     ngx\_conf\_set\_size\_slot,
386     NGX\_HTTP\_LOC\_CONF\_OFFSET,
387     offsetof(ngx\_http\_uwsgi\_loc\_conf\_t, upstream.temp_file_write_size_conf),
388     NULL },
389
390 { ngx_string("uwsgi_next_upstream"),
391     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_1MORE,
392     ngx\_conf\_set\_bitmask\_slot,
393     NGX\_HTTP\_LOC\_CONF\_OFFSET,
394     offsetof(ngx\_http\_uwsgi\_loc\_conf\_t, upstream.next_upstream),
395     &ngx\_http\_uwsgi\_next\_upstream\_masks },
396
397 { ngx_string("uwsgi_next_upstream_tries"),
398     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
399     ngx\_conf\_set\_num\_slot,
400     NGX\_HTTP\_LOC\_CONF\_OFFSET,
401     offsetof(ngx\_http\_uwsgi\_loc\_conf\_t, upstream.next_upstream_tries),
402     NULL },
403
404 { ngx_string("uwsgi_next_upstream_timeout"),
405     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
406     ngx\_conf\_set\_msec\_slot,
407     NGX\_HTTP\_LOC\_CONF\_OFFSET,
408     offsetof(ngx\_http\_uwsgi\_loc\_conf\_t, upstream.next_upstream_timeout),
409     NULL },
410
411 { ngx_string("uwsgi_param"),
412     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE23,
413     ngx\_http\_upstream\_param\_set\_slot,
414     NGX\_HTTP\_LOC\_CONF\_OFFSET,
415     offsetof(ngx\_http\_uwsgi\_loc\_conf\_t, params_source),
416     NULL },
417
418 { ngx_string("uwsgi_string"),
419     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
420     ngx\_conf\_set\_str\_slot,
421     NGX\_HTTP\_LOC\_CONF\_OFFSET,
422     offsetof(ngx\_http\_uwsgi\_loc\_conf\_t, uwsgi_string),
423     NULL },
424
425 { ngx_string("uwsgi_pass_header"),
426     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
427     ngx\_conf\_set\_str\_array\_slot,
428     NGX\_HTTP\_LOC\_CONF\_OFFSET,
429     offsetof(ngx\_http\_uwsgi\_loc\_conf\_t, upstream.pass_headers),
430     NULL },
431
432 { ngx_string("uwsgi_hide_header"),
433     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
434     ngx\_conf\_set\_str\_array\_slot,
435     NGX\_HTTP\_LOC\_CONF\_OFFSET,
436     offsetof(ngx\_http\_uwsgi\_loc\_conf\_t, upstream.hide_headers),
437     NULL },
438
439 { ngx_string("uwsgi_ignore_headers"),
440     NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_1MORE,
441     ngx\_conf\_set\_bitmask\_slot,

```

```

442     NGX\_HTTP\_LOC\_CONF\_OFFSET,
443     offsetof\(ngx\\_http\\_uwsgi\\_loc\\_conf\\_t, upstream.ignore\\_headers\),
444     &ngx\\_http\\_upstream\\_ignore\\_headers\\_masks },
445
446 #if \(NGX\\_HTTP\\_SSL\)
447
448     { ngx\_string\("uwsgi\_ssl\_session\_reuse"\),
449       NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_FLAG,
450       ngx\_conf\_set\_flag\_slot,
451       NGX\_HTTP\_LOC\_CONF\_OFFSET,
452       offsetof\(ngx\\_http\\_uwsgi\\_loc\\_conf\\_t, upstream.ssl\\_session\\_reuse\),
453       NULL },
454
455     { ngx\_string\("uwsgi\_ssl\_protocols"\),
456       NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_1MORE,
457       ngx\_conf\_set\_bitmask\_slot,
458       NGX\_HTTP\_LOC\_CONF\_OFFSET,
459       offsetof\(ngx\\_http\\_uwsgi\\_loc\\_conf\\_t, ssl\\_protocols\),
460       &ngx\\_http\\_uwsgi\\_ssl\\_protocols },
461
462     { ngx\_string\("uwsgi\_ssl\_ciphers"\),
463       NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
464       ngx\_conf\_set\_str\_slot,
465       NGX\_HTTP\_LOC\_CONF\_OFFSET,
466       offsetof\(ngx\\_http\\_uwsgi\\_loc\\_conf\\_t, ssl\\_ciphers\),
467       NULL },
468
469     { ngx\_string\("uwsgi\_ssl\_name"\),
470       NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
471       ngx\_http\_set\_complex\_value\_slot,
472       NGX\_HTTP\_LOC\_CONF\_OFFSET,
473       offsetof\(ngx\\_http\\_uwsgi\\_loc\\_conf\\_t, upstream.ssl\\_name\),
474       NULL },
475
476     { ngx\_string\("uwsgi\_ssl\_server\_name"\),
477       NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_FLAG,
478       ngx\_conf\_set\_flag\_slot,
479       NGX\_HTTP\_LOC\_CONF\_OFFSET,
480       offsetof\(ngx\\_http\\_uwsgi\\_loc\\_conf\\_t, upstream.ssl\\_server\\_name\),
481       NULL },
482
483     { ngx\_string\("uwsgi\_ssl\_verify"\),
484       NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_FLAG,
485       ngx\_conf\_set\_flag\_slot,
486       NGX\_HTTP\_LOC\_CONF\_OFFSET,
487       offsetof\(ngx\\_http\\_uwsgi\\_loc\\_conf\\_t, upstream.ssl\\_verify\),
488       NULL },
489
490     { ngx\_string\("uwsgi\_ssl\_verify\_depth"\),
491       NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
492       ngx\_conf\_set\_num\_slot,
493       NGX\_HTTP\_LOC\_CONF\_OFFSET,
494       offsetof\(ngx\\_http\\_uwsgi\\_loc\\_conf\\_t, ssl\\_verify\\_depth\),
495       NULL },
496
497     { ngx\_string\("uwsgi\_ssl\_trusted\_certificate"\),
498       NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
499       ngx\_conf\_set\_str\_slot,
500       NGX\_HTTP\_LOC\_CONF\_OFFSET,
501       offsetof\(ngx\\_http\\_uwsgi\\_loc\\_conf\\_t, ssl\\_trusted\\_certificate\),
502       NULL },
503
504     { ngx\_string\("uwsgi\_ssl\_crl"\),
505       NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
506       ngx\_conf\_set\_str\_slot,
507       NGX\_HTTP\_LOC\_CONF\_OFFSET,
508       offsetof\(ngx\\_http\\_uwsgi\\_loc\\_conf\\_t, ssl\\_crl\),
509       NULL },
510
511     { ngx\_string\("uwsgi\_ssl\_certificate"\),
512       NGX\_HTTP\_MAIN\_CONF|NGX\_HTTP\_SRV\_CONF|NGX\_HTTP\_LOC\_CONF|NGX\_CONF\_TAKE1,
513       ngx\_conf\_set\_str\_slot,
514       NGX\_HTTP\_LOC\_CONF\_OFFSET,
515       offsetof\(ngx\\_http\\_uwsgi\\_loc\\_conf\\_t, ssl\\_certificate\),
516       NULL },
517

```



```

518 { ngx_string("uwsgi_ssl_certificate_key"),
519     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
520     ngx_conf_set_str_slot,
521     NGX_HTTP_LOC_CONF_OFFSET,
522     offsetof(ngx_http_uwsgi_loc_conf_t, ssl_certificate_key),
523     NULL },
524
525 { ngx_string("uwsgi_ssl_password_file"),
526     NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
527     ngx_http_uwsgi_ssl_password_file,
528     NGX_HTTP_LOC_CONF_OFFSET,
529     0,
530     NULL },
531
532 #endif
533
534     ngx_null_command
535 };
536
537
538 static ngx_http_module_t ngx_http_uwsgi_module_ctx = {
539     NULL,                          /* preconfiguration */
540     NULL,                          /* postconfiguration */
541
542     ngx_http_uwsgi_create_main_conf, /* create main configuration */
543     NULL,                          /* init main configuration */
544
545     NULL,                          /* create server configuration */
546     NULL,                          /* merge server configuration */
547
548     ngx_http_uwsgi_create_loc_conf, /* create location configuration */
549     ngx_http_uwsgi_merge_loc_conf, /* merge location configuration */
550 };
551
552
553 ngx_module_t ngx_http_uwsgi_module = {
554     NGX_MODULE_V1,
555     &ngx_http_uwsgi_module_ctx,    /* module context */
556     ngx_http_uwsgi_commands,      /* module directives */
557     NGX_HTTP_MODULE,              /* module type */
558     NULL,                        /* init master */
559     NULL,                        /* init module */
560     NULL,                        /* init process */
561     NULL,                        /* init thread */
562     NULL,                        /* exit thread */
563     NULL,                        /* exit process */
564     NULL,                        /* exit master */
565     NGX_MODULE_V1_PADDING
566 };
567
568
569 static ngx_str_t ngx_http_uwsgi_hide_headers[] = {
570     ngx_string("X-Accel-Expires"),
571     ngx_string("X-Accel-Redirect"),
572     ngx_string("X-Accel-Limit-Rate"),
573     ngx_string("X-Accel-Buffering"),
574     ngx_string("X-Accel-Charset"),
575     ngx_null_string
576 };
577
578
579 #if (NGX_HTTP_CACHE)
580
581 static ngx_keyval_t ngx_http_uwsgi_cache_headers[] = {
582     { ngx_string("HTTP_IF_MODIFIED_SINCE"),
583       ngx_string("$upstream_cache_last_modified") },
584     { ngx_string("HTTP_IF_UNMODIFIED_SINCE"), ngx_string("") },
585     { ngx_string("HTTP_IF_NONE_MATCH"), ngx_string("$upstream_cache_etag") },
586     { ngx_string("HTTP_IF_MATCH"), ngx_string("") },
587     { ngx_string("HTTP_RANGE"), ngx_string("") },
588     { ngx_string("HTTP_IF_RANGE"), ngx_string("") },
589     { ngx_null_string, ngx_null_string }
590 };
591
592 #endif
593

```

```

594 static ngx_path_init_t ngx_http_uwsgi_temp_path = {
595     ngx_string(NGX_HTTP_UWSGI_TEMP_PATH), { 1, 2, 0 }
596 };
597
598
599
600 static ngx_int_t
601 ngx_http_uwsgi_handler(ngx_http_request_t *r)
602 {
603     ngx_int_t          rc;
604     ngx_http_status_t *status;
605     ngx_http_upstream_t *u;
606     ngx_http_uwsgi_loc_conf_t *uwcf;
607     #if (NGX_HTTP_CACHE)
608     ngx_http_uwsgi_main_conf_t *uwmcf;
609     #endif
610
611     if (ngx_http_upstream_create(r) != NGX_OK) {
612         return NGX_HTTP_INTERNAL_SERVER_ERROR;
613     }
614
615     status = ngx_palloc(r->pool, sizeof(ngx_http_status_t));
616     if (status == NULL) {
617         return NGX_HTTP_INTERNAL_SERVER_ERROR;
618     }
619
620     ngx_http_set_ctx(r, status, ngx_http_uwsgi_module);
621
622     uwcf = ngx_http_get_module_loc_conf(r, ngx_http_uwsgi_module);
623
624     u = r->upstream;
625
626     if (uwcf->uwsgi_lengths == NULL) {
627
628     #if (NGX_HTTP_SSL)
629         u->ssl = (uwcf->upstream.ssl != NULL);
630
631         if (u->ssl) {
632             ngx_str_set(&u->schema, "suwsgi://");
633
634         } else {
635             ngx_str_set(&u->schema, "uwsgi://");
636         }
637     #else
638         ngx_str_set(&u->schema, "uwsgi://");
639     #endif
640
641     } else {
642         if (ngx_http_uwsgi_eval(r, uwcf) != NGX_OK) {
643             return NGX_HTTP_INTERNAL_SERVER_ERROR;
644         }
645     }
646
647     u->output.tag = (ngx_buf_tag_t) &ngx_http_uwsgi_module;
648
649     u->conf = &uwcf->upstream;
650
651     #if (NGX_HTTP_CACHE)
652     uwmcf = ngx_http_get_module_main_conf(r, ngx_http_uwsgi_module);
653
654     u->caches = &uwmcf->caches;
655     u->create_key = ngx_http_uwsgi_create_key;
656     #endif
657
658     u->create_request = ngx_http_uwsgi_create_request;
659     u->reinit_request = ngx_http_uwsgi_reinit_request;
660     u->process_header = ngx_http_uwsgi_process_status_line;
661     u->abort_request = ngx_http_uwsgi_abort_request;
662     u->finalize_request = ngx_http_uwsgi_finalize_request;
663     r->state = 0;
664
665     u->buffering = uwcf->upstream.buffering;
666
667     u->pipe = ngx_palloc(r->pool, sizeof(ngx_event_pipe_t));
668     if (u->pipe == NULL) {
669         return NGX_HTTP_INTERNAL_SERVER_ERROR;

```

```

670     }
671
672     u->pipe->input_filter = ngx\_event\_pipe\_copy\_input\_filter;
673     u->pipe->input_ctx = r;
674
675     rc = ngx\_http\_read\_client\_request\_body(r, ngx\_http\_upstream\_init);
676
677     if (rc >= NGX\_HTTP\_SPECIAL\_RESPONSE) {
678         return rc;
679     }
680
681     return NGX\_DONE;
682 }
683
684
685 static ngx\_int\_t
686 ngx\_http\_uwsgi\_eval(ngx\_http\_request\_t *r, ngx\_http\_uwsgi\_loc\_conf\_t *uwcf)
687 {
688     size\_t          add;
689     ngx\_url\_t      url;
690     ngx\_http\_upstream\_t *u;
691
692     ngx\_memzero(&url, sizeof(ngx\_url\_t));
693
694     if (ngx\_http\_script\_run(r, &url.url, uwcf->uwsgi_lengths->elts, 0,
695                             uwcf->uwsgi_values->elts)
696         == NULL)
697     {
698         return NGX\_ERROR;
699     }
700
701     if (url.url.len > 8
702         && ngx\_strncasecmp(url.url.data, (u_char *) "uwsgi://", 8) == 0)
703     {
704         add = 8;
705     }
706     else if (url.url.len > 9
707             && ngx\_strncasecmp(url.url.data, (u_char *) "suwsgi://", 9) == 0)
708     {
709
710     #if (NGX\_HTTP\_SSL)
711         add = 9;
712         r->upstream->ssl = 1;
713     #else
714         ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, 0,
715                     "suwsgi protocol requires SSL support");
716         return NGX\_ERROR;
717     #endif
718
719     } else {
720         add = 0;
721     }
722
723     u = r->upstream;
724
725     if (add) {
726         u->schema.len = add;
727         u->schema.data = url.url.data;
728
729         url.url.data += add;
730         url.url.len -= add;
731     }
732     else {
733         ngx\_str\_set(&u->schema, "uwsgi://");
734     }
735
736     url.no_resolve = 1;
737
738     if (ngx\_parse\_url(r->pool, &url) != NGX\_OK) {
739         if (url.err) {
740             ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, 0,
741                         "%s in upstream \"%V\"", url.err, &url.url);
742         }
743
744         return NGX\_ERROR;
745     }

```

```

746 u->resolved = ngx_palloc(r->pool, sizeof(ngx_http_upstream_resolved_t));
747 if (u->resolved == NULL) {
748     return NGX_ERROR;
749 }
750
751 if (url.addrs && url.addrs[0].sockaddr) {
752     u->resolved->sockaddr = url.addrs[0].sockaddr;
753     u->resolved->socklen = url.addrs[0].socklen;
754     u->resolved->naddrs = 1;
755     u->resolved->host = url.addrs[0].name;
756 } else {
757     u->resolved->host = url.host;
758     u->resolved->port = url.port;
759     u->resolved->no_port = url.no_port;
760 }
761
762 return NGX_OK;
763 }
764
765 #if (NGX_HTTP_CACHE)
766
767 static ngx_int_t
768 ngx_http_uwsgi_create_key(ngx_http_request_t *r)
769 {
770     ngx_str_t          *key;
771     ngx_http_uwsgi_loc_conf_t *uwcf;
772
773     key = ngx_array_push(&r->cache->keys);
774     if (key == NULL) {
775         return NGX_ERROR;
776     }
777
778     uwcf = ngx_http_get_module_loc_conf(r, ngx_http_uwsgi_module);
779
780     if (ngx_http_complex_value(r, &uwcf->cache_key, key) != NGX_OK) {
781         return NGX_ERROR;
782     }
783
784     return NGX_OK;
785 }
786
787 #endif
788
789 static ngx_int_t
790 ngx_http_uwsgi_create_request(ngx_http_request_t *r)
791 {
792     u_char          ch, *lowercase_key;
793     size_t          key_len, val_len, len, allocated;
794     ngx_uint_t      i, n, hash, skip_empty, header_params;
795     ngx_buf_t       *b;
796     ngx_chain_t     *cl, *body;
797     ngx_list_part_t *part;
798     ngx_table_elt_t *header, **ignored;
799     ngx_http_uwsgi_params_t *params;
800     ngx_http_script_code_pt code;
801     ngx_http_script_engine_t e, le;
802     ngx_http_uwsgi_loc_conf_t *uwcf;
803     ngx_http_script_len_code_pt lcode;
804
805     len = 0;
806     header_params = 0;
807     ignored = NULL;
808
809     uwcf = ngx_http_get_module_loc_conf(r, ngx_http_uwsgi_module);
810
811     #if (NGX_HTTP_CACHE)
812     params = r->upstream->cacheable ? &uwcf->params_cache : &uwcf->params;
813     #else
814     params = &uwcf->params;
815     #endif
816
817     if (params->lengths) {

```

```

822     ngx_memzero(&le, sizeof(ngx_http_script_engine_t));
823
824     ngx_http_script_flush_no_cacheable_variables(r, params->flushes);
825     le.flushed = 1;
826
827     le.ip = params->lengths->elts;
828     le.request = r;
829
830     while (*(uintptr_t *) le.ip) {
831         lcode = *(ngx_http_script_len_code_pt *) le.ip;
832         key_len = lcode(&le);
833
834         lcode = *(ngx_http_script_len_code_pt *) le.ip;
835         skip_empty = lcode(&le);
836
837         for (val_len = 0; *(uintptr_t *) le.ip; val_len += lcode (&le)) {
838             lcode = *(ngx_http_script_len_code_pt *) le.ip;
839         }
840         le.ip += sizeof(uintptr_t);
841
842         if (skip_empty && val_len == 0) {
843             continue;
844         }
845
846         len += 2 + key_len + 2 + val_len;
847     }
848 }
849
850
851 if (uwcf->upstream.pass_request_headers) {
852     allocated = 0;
853     lowercase_key = NULL;
854
855     if (params->number) {
856         n = 0;
857         part = &r->headers_in.headers.part;
858
859         while (part) {
860             n += part->nelts;
861             part = part->next;
862         }
863
864         ignored = ngx_palloc(r->pool, n * sizeof(void *));
865         if (ignored == NULL) {
866             return NGX_ERROR;
867         }
868     }
869 }
870
871 part = &r->headers_in.headers.part;
872 header = part->elts;
873
874 for (i = 0; /* void */ ; i++) {
875     if (i >= part->nelts) {
876         if (part->next == NULL) {
877             break;
878         }
879     }
880
881     part = part->next;
882     header = part->elts;
883     i = 0;
884 }
885
886 if (params->number) {
887     if (allocated < header[i].key.len) {
888         allocated = header[i].key.len + 16;
889         lowercase_key = ngx_pnalloc(r->pool, allocated);
890         if (lowercase_key == NULL) {
891             return NGX_ERROR;
892         }
893     }
894
895     hash = 0;
896
897     for (n = 0; n < header[i].key.len; n++) {

```

```

898         ch = header[i].key.data[n];
899
900         if (ch >= 'A' && ch <= 'Z') {
901             ch |= 0x20;
902
903         } else if (ch == '-') {
904             ch = '_';
905         }
906
907         hash = ngx_hash(hash, ch);
908         lowercase_key[n] = ch;
909     }
910
911     if (ngx_hash_find(&params->hash, hash, lowercase_key, n)) {
912         ignored[header_params++] = &header[i];
913         continue;
914     }
915 }
916
917 len += 2 + sizeof("HTTP_") - 1 + header[i].key.len
918       + 2 + header[i].value.len;
919 }
920 }
921
922 len += uwcf->uwsgi_string.len;
923
924 #if 0
925 /* allow custom uwsgi packet */
926 if (len > 0 && len < 2) {
927     ngx_log_error(NGX_LOG_ALERT, r->connection->log, 0,
928                 "uwsgi request is too little: %uz", len);
929     return NGX_ERROR;
930 }
931 #endif
932
933 b = ngx_create_temp_buf(r->pool, len + 4);
934 if (b == NULL) {
935     return NGX_ERROR;
936 }
937
938 cl = ngx_alloc_chain_link(r->pool);
939 if (cl == NULL) {
940     return NGX_ERROR;
941 }
942
943 cl->buf = b;
944
945 *b->last++ = (u_char) uwcf->modifier1;
946 *b->last++ = (u_char) (len & 0xff);
947 *b->last++ = (u_char) ((len >> 8) & 0xff);
948 *b->last++ = (u_char) uwcf->modifier2;
949
950 if (params->lengths) {
951     ngx_memzero(&e, sizeof(ngx_http_script_engine_t));
952
953     e.ip = params->values->elts;
954     e.pos = b->last;
955     e.request = r;
956     e.flushed = 1;
957
958     le.ip = params->lengths->elts;
959
960     while (*(uintptr_t *) le.ip) {
961
962         lcode = *(ngx_http_script_len_code_pt *) le.ip;
963         key_len = (u_char) lcode (&le);
964
965         lcode = *(ngx_http_script_len_code_pt *) le.ip;
966         skip_empty = lcode(&le);
967
968         for (val_len = 0; *(uintptr_t *) le.ip; val_len += lcode(&le)) {
969             lcode = *(ngx_http_script_len_code_pt *) le.ip;
970         }
971         le.ip += sizeof(uintptr_t);
972
973         if (skip_empty && val_len == 0) {

```

```

974     e.skip = 1;
975
976     while (*(uintptr_t *) e.ip) {
977         code = *(ngx_http_script_code_pt *) e.ip;
978         code((ngx_http_script_engine_t *) &e);
979     }
980     e.ip += sizeof(uintptr_t);
981
982     e.skip = 0;
983
984     continue;
985 }
986
987 *e.pos++ = (u_char) (key_len & 0xff);
988 *e.pos++ = (u_char) ((key_len >> 8) & 0xff);
989
990 code = *(ngx_http_script_code_pt *) e.ip;
991 code((ngx_http_script_engine_t *) &e);
992
993 *e.pos++ = (u_char) (val_len & 0xff);
994 *e.pos++ = (u_char) ((val_len >> 8) & 0xff);
995
996 while (*(uintptr_t *) e.ip) {
997     code = *(ngx_http_script_code_pt *) e.ip;
998     code((ngx_http_script_engine_t *) &e);
999 }
1000
1001 e.ip += sizeof(uintptr_t);
1002
1003 ngx_log_debug4(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1004               "uwsgi param: \"%*s: %*s\"",
1005               key_len, e.pos - (key_len + 2 + val_len),
1006               val_len, e.pos - val_len);
1007 }
1008
1009 b->last = e.pos;
1010 }
1011
1012 if (uwcf->upstream.pass_request_headers) {
1013
1014     part = &r->headers_in.headers.part;
1015     header = part->elts;
1016
1017     for (i = 0; /* void */ ; i++) {
1018
1019         if (i >= part->nelts) {
1020             if (part->next == NULL) {
1021                 break;
1022             }
1023
1024             part = part->next;
1025             header = part->elts;
1026             i = 0;
1027         }
1028
1029         for (n = 0; n < header_params; n++) {
1030             if (&header[i] == ignored[n]) {
1031                 goto next;
1032             }
1033         }
1034
1035         key_len = sizeof("HTTP_") - 1 + header[i].key.len;
1036         *b->last++ = (u_char) (key_len & 0xff);
1037         *b->last++ = (u_char) ((key_len >> 8) & 0xff);
1038
1039         b->last = ngx_cpymem(b->last, "HTTP_", sizeof("HTTP_") - 1);
1040         for (n = 0; n < header[i].key.len; n++) {
1041             ch = header[i].key.data[n];
1042
1043             if (ch >= 'a' && ch <= 'z') {
1044                 ch &= ~0x20;
1045
1046             } else if (ch == '-') {
1047                 ch = '_';
1048             }
1049         }

```

```

1050         *b->last++ = ch;
1051     }
1052
1053     val_len = header[i].value.len;
1054     *b->last++ = (u_char) (val_len & 0xff);
1055     *b->last++ = (u_char) ((val_len >> 8) & 0xff);
1056     b->last = ngx_copy(b->last, header[i].value.data, val_len);
1057
1058     ngx_log_debug4(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1059                 "uwsgi param: \"%*s: %*s\"",
1060                 key_len, b->last - (key_len + 2 + val_len),
1061                 val_len, b->last - val_len);
1062     next:
1063
1064     continue;
1065 }
1066 }
1067
1068 b->last = ngx_copy(b->last, uwcf->uwsgi_string.data,
1069                 uwcf->uwsgi_string.len);
1070
1071 if (uwcf->upstream.pass_request_body) {
1072     body = r->upstream->request_bufs;
1073     r->upstream->request_bufs = c1;
1074
1075     while (body) {
1076         b = ngx_alloc_buf(r->pool);
1077         if (b == NULL) {
1078             return NGX_ERROR;
1079         }
1080
1081         ngx_memcpy(b, body->buf, sizeof(ngx_buf_t));
1082
1083         c1->next = ngx_alloc_chain_link(r->pool);
1084         if (c1->next == NULL) {
1085             return NGX_ERROR;
1086         }
1087
1088         c1 = c1->next;
1089         c1->buf = b;
1090
1091         body = body->next;
1092     }
1093
1094 } else {
1095     r->upstream->request_bufs = c1;
1096 }
1097
1098 c1->next = NULL;
1099
1100 return NGX_OK;
1101 }
1102
1103
1104 static ngx_int_t
1105 ngx_http_uwsgi_reinit_request(ngx_http_request_t *r)
1106 {
1107     ngx_http_status_t *status;
1108
1109     status = ngx_http_get_module_ctx(r, ngx_http_uwsgi_module);
1110
1111     if (status == NULL) {
1112         return NGX_OK;
1113     }
1114
1115     status->code = 0;
1116     status->count = 0;
1117     status->start = NULL;
1118     status->end = NULL;
1119
1120     r->upstream->process_header = ngx_http_uwsgi_process_status_line;
1121     r->state = 0;
1122
1123     return NGX_OK;
1124 }
1125

```



```

1126 static ngx_int_t
1127 ngx_http_uwsgi_process_status_line(ngx_http_request_t *r)
1128 {
1129     size_t          len;
1130     ngx_int_t       rc;
1131     ngx_http_status_t *status;
1132     ngx_http_upstream_t *u;
1133
1134     status = ngx_http_get_module_ctx(r, ngx_http_uwsgi_module);
1135
1136     if (status == NULL) {
1137         return NGX_ERROR;
1138     }
1139
1140     u = r->upstream;
1141
1142     rc = ngx_http_parse_status_line(r, &u->buffer, status);
1143
1144     if (rc == NGX_AGAIN) {
1145         return rc;
1146     }
1147
1148     if (rc == NGX_ERROR) {
1149         u->process_header = ngx_http_uwsgi_process_header;
1150         return ngx_http_uwsgi_process_header(r);
1151     }
1152
1153     if (u->state && u->state->status == 0) {
1154         u->state->status = status->code;
1155     }
1156
1157     u->headers_in.status_n = status->code;
1158
1159     len = status->end - status->start;
1160     u->headers_in.status_line.len = len;
1161
1162     u->headers_in.status_line.data = ngx_pnalloc(r->pool, len);
1163     if (u->headers_in.status_line.data == NULL) {
1164         return NGX_ERROR;
1165     }
1166
1167     ngx_memcpy(u->headers_in.status_line.data, status->start, len);
1168
1169     ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1170                 "http uwsgi status %ui \"%V\"",
1171                 u->headers_in.status_n, &u->headers_in.status_line);
1172
1173     u->process_header = ngx_http_uwsgi_process_header;
1174
1175     return ngx_http_uwsgi_process_header(r);
1176 }
1177
1178
1179 static ngx_int_t
1180 ngx_http_uwsgi_process_header(ngx_http_request_t *r)
1181 {
1182     ngx_str_t          *status_line;
1183     ngx_int_t          rc, status;
1184     ngx_table_elt_t    *h;
1185     ngx_http_upstream_t *u;
1186     ngx_http_upstream_header_t *hh;
1187     ngx_http_upstream_main_conf_t *umcf;
1188
1189     umcf = ngx_http_get_module_main_conf(r, ngx_http_upstream_module);
1190
1191     for ( ;; ) {
1192
1193         rc = ngx_http_parse_header_line(r, &r->upstream->buffer, 1);
1194
1195         if (rc == NGX_OK) {
1196
1197             /* a header line has been parsed successfully */
1198
1199             h = ngx_list_push(&r->upstream->headers_in.headers);
1200             if (h == NULL) {
1201

```

```

1202     return NGX_ERROR;
1203 }
1204
1205 h->hash = r->header_hash;
1206
1207 h->key.len = r->header_name_end - r->header_name_start;
1208 h->value.len = r->header_end - r->header_start;
1209
1210 h->key.data = ngx_pnalloc(r->pool,
1211                          h->key.len + 1 + h->value.len + 1
1212                          + h->key.len);
1213 if (h->key.data == NULL) {
1214     return NGX_ERROR;
1215 }
1216
1217 h->value.data = h->key.data + h->key.len + 1;
1218 h->lowercase_key = h->key.data + h->key.len + 1 + h->value.len + 1;
1219
1220 ngx_memcpy(h->key.data, r->header_name_start, h->key.len);
1221 h->key.data[h->key.len] = '\0';
1222 ngx_memcpy(h->value.data, r->header_start, h->value.len);
1223 h->value.data[h->value.len] = '\0';
1224
1225 if (h->key.len == r->lowercase_index) {
1226     ngx_memcpy(h->lowercase_key, r->lowercase_header, h->key.len);
1227 } else {
1228     ngx_strlow(h->lowercase_key, h->key.data, h->key.len);
1229 }
1230
1231 hh = ngx_hash_find(&umcf->headers_in_hash, h->hash,
1232                  h->lowercase_key, h->key.len);
1233
1234 if (hh && hh->handler(r, h, hh->offset) != NGX_OK) {
1235     return NGX_ERROR;
1236 }
1237
1238 ngx_log_debug2(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1239               "http uwsgi header: \"%V: %V\"", &h->key, &h->value);
1240
1241 continue;
1242 }
1243
1244 if (rc == NGX_HTTP_PARSE_HEADER_DONE) {
1245     /* a whole header has been parsed successfully */
1246
1247     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1248                   "http uwsgi header done");
1249
1250     u = r->upstream;
1251
1252     if (u->headers_in.status_n) {
1253         goto done;
1254     }
1255
1256     if (u->headers_in.status) {
1257         status_line = &u->headers_in.status->value;
1258
1259         status = ngx_atoi(status_line->data, 3);
1260         if (status == NGX_ERROR) {
1261             ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1262                           "upstream sent invalid status \"%V\"",
1263                           status_line);
1264             return NGX_HTTP_UPSTREAM_INVALID_HEADER;
1265         }
1266
1267         u->headers_in.status_n = status;
1268         u->headers_in.status_line = *status_line;
1269
1270     } else if (u->headers_in.location) {
1271         u->headers_in.status_n = 302;
1272         ngx_str_set(&u->headers_in.status_line,
1273                   "302 Moved Temporarily");
1274     } else {
1275

```

```

1278         u->headers_in.status_n = 200;
1279         ngx_str_set(&u->headers_in.status_line, "200 OK");
1280     }
1281
1282     if (u->state && u->state->status == 0) {
1283         u->state->status = u->headers_in.status_n;
1284     }
1285
1286     done:
1287
1288     if (u->headers_in.status_n == NGX_HTTP_SWITCHING_PROTOCOLS
1289         && r->headers_in.upgrade)
1290     {
1291         u->upgrade = 1;
1292     }
1293
1294     return NGX_OK;
1295 }
1296
1297 if (rc == NGX_AGAIN) {
1298     return NGX_AGAIN;
1299 }
1300
1301 /* there was error while a header line parsing */
1302
1303 ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
1304             "upstream sent invalid header");
1305
1306     return NGX_HTTP_UPSTREAM_INVALID_HEADER;
1307 }
1308 }
1309
1310
1311 static void
1312 ngx_http_uwsgi_abort_request(ngx_http_request_t *r)
1313 {
1314     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1315                 "abort http uwsgi request");
1316
1317     return;
1318 }
1319
1320
1321 static void
1322 ngx_http_uwsgi_finalize_request(ngx_http_request_t *r, ngx_int_t rc)
1323 {
1324     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
1325                 "finalize http uwsgi request");
1326
1327     return;
1328 }
1329
1330
1331 static void *
1332 ngx_http_uwsgi_create_main_conf(ngx_conf_t *cf)
1333 {
1334     ngx_http_uwsgi_main_conf_t *conf;
1335
1336     conf = ngx_palloc(cf->pool, sizeof(ngx_http_uwsgi_main_conf_t));
1337     if (conf == NULL) {
1338         return NULL;
1339     }
1340
1341     #if (NGX_HTTP_CACHE)
1342     if (ngx_array_init(&conf->caches, cf->pool, 4,
1343                     sizeof(ngx_http_file_cache_t *))
1344         != NGX_OK)
1345     {
1346         return NULL;
1347     }
1348     #endif
1349
1350     return conf;
1351 }
1352
1353

```

```

1354 static void *
1355 ngx_http_uwsgi_create_loc_conf(ngx_conf_t *cf)
1356 {
1357     ngx_http_uwsgi_loc_conf_t *conf;
1358
1359     conf = ngx_palloc(cf->pool, sizeof(ngx_http_uwsgi_loc_conf_t));
1360     if (conf == NULL) {
1361         return NULL;
1362     }
1363
1364     conf->modifier1 = NGX_CONF_UNSET_UINT;
1365     conf->modifier2 = NGX_CONF_UNSET_UINT;
1366
1367     conf->upstream.store = NGX_CONF_UNSET;
1368     conf->upstream.store_access = NGX_CONF_UNSET_UINT;
1369     conf->upstream.next_upstream_tries = NGX_CONF_UNSET_UINT;
1370     conf->upstream.buffering = NGX_CONF_UNSET;
1371     conf->upstream.ignore_client_abort = NGX_CONF_UNSET;
1372     conf->upstream.force_ranges = NGX_CONF_UNSET;
1373
1374     conf->upstream.local = NGX_CONF_UNSET_PTR;
1375
1376     conf->upstream.connect_timeout = NGX_CONF_UNSET_MSEC;
1377     conf->upstream.send_timeout = NGX_CONF_UNSET_MSEC;
1378     conf->upstream.read_timeout = NGX_CONF_UNSET_MSEC;
1379     conf->upstream.next_upstream_timeout = NGX_CONF_UNSET_MSEC;
1380
1381     conf->upstream.send_lowat = NGX_CONF_UNSET_SIZE;
1382     conf->upstream.buffer_size = NGX_CONF_UNSET_SIZE;
1383     conf->upstream.limit_rate = NGX_CONF_UNSET_SIZE;
1384
1385     conf->upstream.busy_buffers_size_conf = NGX_CONF_UNSET_SIZE;
1386     conf->upstream.max_temp_file_size_conf = NGX_CONF_UNSET_SIZE;
1387     conf->upstream.temp_file_write_size_conf = NGX_CONF_UNSET_SIZE;
1388
1389     conf->upstream.pass_request_headers = NGX_CONF_UNSET;
1390     conf->upstream.pass_request_body = NGX_CONF_UNSET;
1391
1392     #if (NGX_HTTP_CACHE)
1393     conf->upstream.cache = NGX_CONF_UNSET;
1394     conf->upstream.cache_min_uses = NGX_CONF_UNSET_UINT;
1395     conf->upstream.cache_bypass = NGX_CONF_UNSET_PTR;
1396     conf->upstream.no_cache = NGX_CONF_UNSET_PTR;
1397     conf->upstream.cache_valid = NGX_CONF_UNSET_PTR;
1398     conf->upstream.cache_lock = NGX_CONF_UNSET;
1399     conf->upstream.cache_lock_timeout = NGX_CONF_UNSET_MSEC;
1400     conf->upstream.cache_lock_age = NGX_CONF_UNSET_MSEC;
1401     conf->upstream.cache_revalidate = NGX_CONF_UNSET;
1402     #endif
1403
1404     conf->upstream.hide_headers = NGX_CONF_UNSET_PTR;
1405     conf->upstream.pass_headers = NGX_CONF_UNSET_PTR;
1406
1407     conf->upstream.intercept_errors = NGX_CONF_UNSET;
1408
1409     #if (NGX_HTTP_SSL)
1410     conf->upstream.ssl_session_reuse = NGX_CONF_UNSET;
1411     conf->upstream.ssl_server_name = NGX_CONF_UNSET;
1412     conf->upstream.ssl_verify = NGX_CONF_UNSET;
1413     conf->ssl_verify_depth = NGX_CONF_UNSET_UINT;
1414     conf->ssl_passwords = NGX_CONF_UNSET_PTR;
1415     #endif
1416
1417     /* "uwsgi_cyclic_temp_file" is disabled */
1418     conf->upstream.cyclic_temp_file = 0;
1419
1420     conf->upstream.change_buffering = 1;
1421
1422     ngx_str_set(&conf->upstream.module, "uwsgi");
1423
1424     return conf;
1425 }
1426
1427
1428 static char *
1429 ngx_http_uwsgi_merge_loc_conf(ngx_conf_t *cf, void *parent, void *child)

```

```

1430 {
1431     ngx\_http\_uwsgi\_loc\_conf\_t *prev = parent;
1432     ngx\_http\_uwsgi\_loc\_conf\_t *conf = child;
1433
1434     size_t                size;
1435     ngx\_int\_t              rc;
1436     ngx\_hash\_init\_t       hash;
1437     ngx\_http\_core\_loc\_conf\_t *clcf;
1438
1439     #if (NGX_HTTP_CACHE)
1440
1441     if (conf->upstream.store > 0) {
1442         conf->upstream.cache = 0;
1443     }
1444
1445     if (conf->upstream.cache > 0) {
1446         conf->upstream.store = 0;
1447     }
1448
1449     #endif
1450
1451     if (conf->upstream.store == NGX\_CONF\_UNSET) {
1452         ngx\_conf\_merge\_value(conf->upstream.store, prev->upstream.store, 0);
1453
1454         conf->upstream.store_lengths = prev->upstream.store_lengths;
1455         conf->upstream.store_values = prev->upstream.store_values;
1456     }
1457
1458     ngx\_conf\_merge\_uint\_value(conf->upstream.store_access,
1459                             prev->upstream.store_access, 0600);
1460
1461     ngx\_conf\_merge\_uint\_value(conf->upstream.next_upstream_tries,
1462                             prev->upstream.next_upstream_tries, 0);
1463
1464     ngx\_conf\_merge\_value(conf->upstream.buffering,
1465                        prev->upstream.buffering, 1);
1466
1467     ngx\_conf\_merge\_value(conf->upstream.ignore_client_abort,
1468                        prev->upstream.ignore_client_abort, 0);
1469
1470     ngx\_conf\_merge\_value(conf->upstream.force_ranges,
1471                        prev->upstream.force_ranges, 0);
1472
1473     ngx\_conf\_merge\_ptr\_value(conf->upstream.local,
1474                            prev->upstream.local, NULL);
1475
1476     ngx\_conf\_merge\_msec\_value(conf->upstream.connect_timeout,
1477                             prev->upstream.connect_timeout, 60000);
1478
1479     ngx\_conf\_merge\_msec\_value(conf->upstream.send_timeout,
1480                             prev->upstream.send_timeout, 60000);
1481
1482     ngx\_conf\_merge\_msec\_value(conf->upstream.read_timeout,
1483                             prev->upstream.read_timeout, 60000);
1484
1485     ngx\_conf\_merge\_msec\_value(conf->upstream.next_upstream_timeout,
1486                             prev->upstream.next_upstream_timeout, 0);
1487
1488     ngx\_conf\_merge\_size\_value(conf->upstream.send_lowat,
1489                             prev->upstream.send_lowat, 0);
1490
1491     ngx\_conf\_merge\_size\_value(conf->upstream.buffer_size,
1492                             prev->upstream.buffer_size,
1493                             (size_t) ngx\_pagesize);
1494
1495     ngx\_conf\_merge\_size\_value(conf->upstream.limit_rate,
1496                             prev->upstream.limit_rate, 0);
1497
1498
1499     ngx\_conf\_merge\_bufs\_value(conf->upstream.bufs, prev->upstream.bufs,
1500                             8, ngx\_pagesize);
1501
1502     if (conf->upstream.bufs.num < 2) {
1503         ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
1504                          "there must be at least 2 \"uwsgi_buffers\"");
1505         return NGX\_CONF\_ERROR;

```

```

1506 }
1507
1508
1509 size = conf->upstream.buffer_size;
1510 if (size < conf->upstream.bufs.size) {
1511     size = conf->upstream.bufs.size;
1512 }
1513
1514
1515 ngx\_conf\_merge\_size\_value(conf->upstream.busy_buffers_size_conf,
1516     prev->upstream.busy_buffers_size_conf,
1517     NGX\_CONF\_UNSET\_SIZE);
1518
1519 if (conf->upstream.busy_buffers_size_conf == NGX\_CONF\_UNSET\_SIZE) {
1520     conf->upstream.busy_buffers_size = 2 * size;
1521 } else {
1522     conf->upstream.busy_buffers_size =
1523         conf->upstream.busy_buffers_size_conf;
1524 }
1525
1526 if (conf->upstream.busy_buffers_size < size) {
1527     ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
1528         "\"uwsGI_busy_buffers_size\" must be equal to or greater "
1529         "than the maximum of the value of \"uwsGI_buffer_size\" and "
1530         "one of the \"uwsGI_buffers\"");
1531
1532     return NGX\_CONF\_ERROR;
1533 }
1534
1535 if (conf->upstream.busy_buffers_size
1536     > (conf->upstream.bufs.num - 1) * conf->upstream.bufs.size)
1537 {
1538     ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
1539         "\"uwsGI_busy_buffers_size\" must be less than "
1540         "the size of all \"uwsGI_buffers\" minus one buffer");
1541
1542     return NGX\_CONF\_ERROR;
1543 }
1544
1545
1546 ngx\_conf\_merge\_size\_value(conf->upstream.temp_file_write_size_conf,
1547     prev->upstream.temp_file_write_size_conf,
1548     NGX\_CONF\_UNSET\_SIZE);
1549
1550 if (conf->upstream.temp_file_write_size_conf == NGX\_CONF\_UNSET\_SIZE) {
1551     conf->upstream.temp_file_write_size = 2 * size;
1552 } else {
1553     conf->upstream.temp_file_write_size =
1554         conf->upstream.temp_file_write_size_conf;
1555 }
1556
1557 if (conf->upstream.temp_file_write_size < size) {
1558     ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,
1559         "\"uwsGI_temp_file_write_size\" must be equal to or greater than "
1560         "the maximum of the value of \"uwsGI_buffer_size\" and "
1561         "one of the \"uwsGI_buffers\"");
1562
1563     return NGX\_CONF\_ERROR;
1564 }
1565
1566
1567 ngx\_conf\_merge\_size\_value(conf->upstream.max_temp_file_size_conf,
1568     prev->upstream.max_temp_file_size_conf,
1569     NGX\_CONF\_UNSET\_SIZE);
1570
1571 if (conf->upstream.max_temp_file_size_conf == NGX\_CONF\_UNSET\_SIZE) {
1572     conf->upstream.max_temp_file_size = 1024 * 1024 * 1024;
1573 } else {
1574     conf->upstream.max_temp_file_size =
1575         conf->upstream.max_temp_file_size_conf;
1576 }
1577
1578 if (conf->upstream.max_temp_file_size != 0
1579     && conf->upstream.max_temp_file_size < size)
1580 {
1581     ngx\_conf\_log\_error(NGX\_LOG\_EMERG, cf, 0,

```

```

1582     "\"uwsigi_max_temp_file_size\" must be equal to zero to disable "
1583     "temporary files usage or must be equal to or greater than "
1584     "the maximum of the value of \"uwsigi_buffer_size\" and "
1585     "one of the \"uwsigi_buffers\"");
1586
1587     return NGX_CONF_ERROR;
1588 }
1589
1590
1591 ngx_conf_merge_bitmask_value(conf->upstream.ignore_headers,
1592     prev->upstream.ignore_headers,
1593     NGX_CONF_BITMASK_SET);
1594
1595
1596 ngx_conf_merge_bitmask_value(conf->upstream.next_upstream,
1597     prev->upstream.next_upstream,
1598     (NGX_CONF_BITMASK_SET
1599     |NGX_HTTP_UPSTREAM_FT_ERROR
1600     |NGX_HTTP_UPSTREAM_FT_TIMEOUT));
1601
1602 if (conf->upstream.next_upstream & NGX_HTTP_UPSTREAM_FT_OFF) {
1603     conf->upstream.next_upstream = NGX_CONF_BITMASK_SET
1604         |NGX_HTTP_UPSTREAM_FT_OFF;
1605 }
1606
1607 if (ngx_conf_merge_path_value(cf, &conf->upstream.temp_path,
1608     prev->upstream.temp_path,
1609     &ngx_http_uwsigi_temp_path)
1610     != NGX_OK)
1611 {
1612     return NGX_CONF_ERROR;
1613 }
1614
1615 #if (NGX_HTTP_CACHE)
1616
1617 if (conf->upstream.cache == NGX_CONF_UNSET) {
1618     ngx_conf_merge_value(conf->upstream.cache,
1619         prev->upstream.cache, 0);
1620
1621     conf->upstream.cache_zone = prev->upstream.cache_zone;
1622     conf->upstream.cache_value = prev->upstream.cache_value;
1623 }
1624
1625 if (conf->upstream.cache_zone && conf->upstream.cache_zone->data == NULL) {
1626     ngx_shm_zone_t *shm_zone;
1627
1628     shm_zone = conf->upstream.cache_zone;
1629
1630     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
1631         "\"uwsigi_cache\" zone \"%V\" is unknown",
1632         &shm_zone->shm.name);
1633
1634     return NGX_CONF_ERROR;
1635 }
1636
1637 ngx_conf_merge_uint_value(conf->upstream.cache_min_uses,
1638     prev->upstream.cache_min_uses, 1);
1639
1640 ngx_conf_merge_bitmask_value(conf->upstream.cache_use_stale,
1641     prev->upstream.cache_use_stale,
1642     (NGX_CONF_BITMASK_SET
1643     |NGX_HTTP_UPSTREAM_FT_OFF));
1644
1645 if (conf->upstream.cache_use_stale & NGX_HTTP_UPSTREAM_FT_OFF) {
1646     conf->upstream.cache_use_stale = NGX_CONF_BITMASK_SET
1647         |NGX_HTTP_UPSTREAM_FT_OFF;
1648 }
1649
1650 if (conf->upstream.cache_use_stale & NGX_HTTP_UPSTREAM_FT_ERROR) {
1651     conf->upstream.cache_use_stale |= NGX_HTTP_UPSTREAM_FT_NOLIVE;
1652 }
1653
1654 if (conf->upstream.cache_methods == 0) {
1655     conf->upstream.cache_methods = prev->upstream.cache_methods;
1656 }
1657

```

```

1658 conf->upstream.cache_methods |= NGX\_HTTP\_GET|NGX\_HTTP\_HEAD;
1659
1660 ngx\_conf\_merge\_ptr\_value(conf->upstream.cache_bypass,
1661     prev->upstream.cache_bypass, NULL);
1662
1663 ngx\_conf\_merge\_ptr\_value(conf->upstream.no_cache,
1664     prev->upstream.no_cache, NULL);
1665
1666 ngx\_conf\_merge\_ptr\_value(conf->upstream.cache_valid,
1667     prev->upstream.cache_valid, NULL);
1668
1669 if (conf->cache_key.value.data == NULL) {
1670     conf->cache_key = prev->cache_key;
1671 }
1672
1673 if (conf->upstream.cache && conf->cache_key.value.data == NULL) {
1674     ngx\_conf\_log\_error(NGX\_LOG\_WARN, cf, 0,
1675         "no \"uwsgi_cache_key\" for \"uwsgi_cache\"");
1676 }
1677
1678 ngx\_conf\_merge\_value(conf->upstream.cache_lock,
1679     prev->upstream.cache_lock, 0);
1680
1681 ngx\_conf\_merge\_msec\_value(conf->upstream.cache_lock_timeout,
1682     prev->upstream.cache_lock_timeout, 5000);
1683
1684 ngx\_conf\_merge\_msec\_value(conf->upstream.cache_lock_age,
1685     prev->upstream.cache_lock_age, 5000);
1686
1687 ngx\_conf\_merge\_value(conf->upstream.cache_revalidate,
1688     prev->upstream.cache_revalidate, 0);
1689
1690 #endif
1691
1692 ngx\_conf\_merge\_value(conf->upstream.pass_request_headers,
1693     prev->upstream.pass_request_headers, 1);
1694 ngx\_conf\_merge\_value(conf->upstream.pass_request_body,
1695     prev->upstream.pass_request_body, 1);
1696
1697 ngx\_conf\_merge\_value(conf->upstream.intercept_errors,
1698     prev->upstream.intercept_errors, 0);
1699
1700 #if (NGX\_HTTP\_SSL)
1701
1702 ngx\_conf\_merge\_value(conf->upstream.ssl_session_reuse,
1703     prev->upstream.ssl_session_reuse, 1);
1704
1705 ngx\_conf\_merge\_bitmask\_value(conf->ssl_protocols, prev->ssl_protocols,
1706     (NGX\_CONF\_BITMASK\_SET|NGX\_SSL\_SSLv3
1707     |NGX\_SSL\_TLSv1|NGX\_SSL\_TLSv1\_1
1708     |NGX\_SSL\_TLSv1\_2));
1709
1710 ngx\_conf\_merge\_str\_value(conf->ssl_ciphers, prev->ssl_ciphers,
1711     "DEFAULT");
1712
1713 if (conf->upstream.ssl_name == NULL) {
1714     conf->upstream.ssl_name = prev->upstream.ssl_name;
1715 }
1716
1717 ngx\_conf\_merge\_value(conf->upstream.ssl_server_name,
1718     prev->upstream.ssl_server_name, 0);
1719 ngx\_conf\_merge\_value(conf->upstream.ssl_verify,
1720     prev->upstream.ssl_verify, 0);
1721 ngx\_conf\_merge\_uint\_value(conf->ssl_verify_depth,
1722     prev->ssl_verify_depth, 1);
1723 ngx\_conf\_merge\_str\_value(conf->ssl_trusted_certificate,
1724     prev->ssl_trusted_certificate, "");
1725 ngx\_conf\_merge\_str\_value(conf->ssl_crl, prev->ssl_crl, "");
1726
1727 ngx\_conf\_merge\_str\_value(conf->ssl_certificate,
1728     prev->ssl_certificate, "");
1729 ngx\_conf\_merge\_str\_value(conf->ssl_certificate_key,
1730     prev->ssl_certificate_key, "");
1731 ngx\_conf\_merge\_ptr\_value(conf->ssl_passwords, prev->ssl_passwords, NULL);
1732
1733 if (conf->ssl && ngx\_http\_uwsgi\_set\_ssl(cf, conf) != NGX\_OK) {

```



```

1734     return NGX\_CONF\_ERROR;
1735 }
1736
1737 #endif
1738
1739     ngx\_conf\_merge\_str\_value(conf->uwsgi_string, prev->uwsgi_string, "");
1740
1741     hash.max_size = 512;
1742     hash.bucket_size = ngx\_align(64, ngx\_cacheline\_size);
1743     hash.name = "uwsgi_hide_headers_hash";
1744
1745     if (ngx\_http\_upstream\_hide\_headers\_hash(cf, &conf->upstream,
1746         &prev->upstream, ngx\_http\_uwsgi\_hide\_headers, &hash)
1747         != NGX\_OK)
1748     {
1749         return NGX\_CONF\_ERROR;
1750     }
1751
1752     clcf = ngx\_http\_conf\_get\_module\_loc\_conf(cf, ngx\_http\_core\_module);
1753
1754     if (clcf->noname
1755         && conf->upstream.upstream == NULL && conf->uwsgi_lengths == NULL)
1756     {
1757         conf->upstream.upstream = prev->upstream.upstream;
1758
1759         conf->uwsgi_lengths = prev->uwsgi_lengths;
1760         conf->uwsgi_values = prev->uwsgi_values;
1761
1762         #if (NGX\_HTTP\_SSL)
1763             conf->upstream.ssl = prev->upstream.ssl;
1764         #endif
1765     }
1766
1767     if (clcf->lmt_excpt && clcf->handler == NULL
1768         && (conf->upstream.upstream || conf->uwsgi_lengths))
1769     {
1770         clcf->handler = ngx\_http\_uwsgi\_handler;
1771     }
1772
1773     ngx\_conf\_merge\_uint\_value(conf->modifier1, prev->modifier1, 0);
1774     ngx\_conf\_merge\_uint\_value(conf->modifier2, prev->modifier2, 0);
1775
1776     if (conf->params_source == NULL) {
1777         conf->params = prev->params;
1778     #if (NGX\_HTTP\_CACHE)
1779         conf->params_cache = prev->params_cache;
1780     #endif
1781         conf->params_source = prev->params_source;
1782     }
1783
1784     rc = ngx\_http\_uwsgi\_init\_params(cf, conf, &conf->params, NULL);
1785     if (rc != NGX\_OK) {
1786         return NGX\_CONF\_ERROR;
1787     }
1788
1789     #if (NGX\_HTTP\_CACHE)
1790
1791     if (conf->upstream.cache) {
1792         rc = ngx\_http\_uwsgi\_init\_params(cf, conf, &conf->params_cache,
1793             ngx\_http\_uwsgi\_cache\_headers);
1794         if (rc != NGX\_OK) {
1795             return NGX\_CONF\_ERROR;
1796         }
1797     }
1798
1799     #endif
1800
1801     return NGX\_CONF\_OK;
1802 }
1803
1804
1805 static ngx\_int\_t
1806 ngx\_http\_uwsgi\_init\_params(ngx\_conf\_t *cf, ngx\_http\_uwsgi\_loc\_conf\_t *conf,
1807     ngx\_http\_uwsgi\_params\_t *params, ngx\_keyval\_t *default_params)
1808 {
1809     u_char                *p;

```

```

1810     size_t                size;
1811     uintptr_t            *code;
1812     ngx_uint_t          i, nsrc;
1813     ngx_array_t         headers_names, params_merged;
1814     ngx_keyval_t        *h;
1815     ngx_hash_key_t      *hk;
1816     ngx_hash_init_t     hash;
1817     ngx_http_upstream_param_t *src, *s;
1818     ngx_http_script_compile_t sc;
1819     ngx_http_script_copy_code_t *copy;
1820
1821     if (params->hash.buckets) {
1822         return NGX_OK;
1823     }
1824
1825     if (conf->params_source == NULL && default_params == NULL) {
1826         params->hash.buckets = (void *) 1;
1827         return NGX_OK;
1828     }
1829
1830     params->lengths = ngx_array_create(cf->pool, 64, 1);
1831     if (params->lengths == NULL) {
1832         return NGX_ERROR;
1833     }
1834
1835     params->values = ngx_array_create(cf->pool, 512, 1);
1836     if (params->values == NULL) {
1837         return NGX_ERROR;
1838     }
1839
1840     if (ngx_array_init(&headers_names, cf->temp_pool, 4, sizeof(ngx_hash_key_t))
1841         != NGX_OK)
1842     {
1843         return NGX_ERROR;
1844     }
1845
1846     if (conf->params_source) {
1847         src = conf->params_source->elts;
1848         nsrc = conf->params_source->nelts;
1849
1850     } else {
1851         src = NULL;
1852         nsrc = 0;
1853     }
1854
1855     if (default_params) {
1856         if (ngx_array_init(&params_merged, cf->temp_pool, 4,
1857             sizeof(ngx_http_upstream_param_t))
1858             != NGX_OK)
1859         {
1860             return NGX_ERROR;
1861         }
1862
1863         for (i = 0; i < nsrc; i++) {
1864
1865             s = ngx_array_push(&params_merged);
1866             if (s == NULL) {
1867                 return NGX_ERROR;
1868             }
1869
1870             *s = src[i];
1871         }
1872
1873         h = default_params;
1874
1875         while (h->key.len) {
1876
1877             src = params_merged.elts;
1878             nsrc = params_merged.nelts;
1879
1880             for (i = 0; i < nsrc; i++) {
1881                 if (ngx_strcasecmp(h->key.data, src[i].key.data) == 0) {
1882                     goto next;
1883                 }
1884             }
1885

```

```

1886     s = ngx_array_push(&params_merged);
1887     if (s == NULL) {
1888         return NGX_ERROR;
1889     }
1890
1891     s->key = h->key;
1892     s->value = h->value;
1893     s->skip_empty = 1;
1894
1895     next:
1896
1897     h++;
1898 }
1899
1900     src = params_merged.elts;
1901     nsrc = params_merged.nelts;
1902 }
1903
1904 for (i = 0; i < nsrc; i++) {
1905
1906     if (src[i].key.len > sizeof("HTTP_") - 1
1907         && ngx_strncmp(src[i].key.data, "HTTP_", sizeof("HTTP_") - 1) == 0)
1908     {
1909         hk = ngx_array_push(&headers_names);
1910         if (hk == NULL) {
1911             return NGX_ERROR;
1912         }
1913
1914         hk->key.len = src[i].key.len - 5;
1915         hk->key.data = src[i].key.data + 5;
1916         hk->key_hash = ngx_hash_key_lc(hk->key.data, hk->key.len);
1917         hk->value = (void *) 1;
1918
1919         if (src[i].value.len == 0) {
1920             continue;
1921         }
1922     }
1923
1924     copy = ngx_array_push_n(params->lengths,
1925                             sizeof(ngx_http_script_copy_code_t));
1926     if (copy == NULL) {
1927         return NGX_ERROR;
1928     }
1929
1930     copy->code = (ngx_http_script_code_pt) ngx_http_script_copy_len_code;
1931     copy->len = src[i].key.len;
1932
1933     copy = ngx_array_push_n(params->lengths,
1934                             sizeof(ngx_http_script_copy_code_t));
1935     if (copy == NULL) {
1936         return NGX_ERROR;
1937     }
1938
1939     copy->code = (ngx_http_script_code_pt) ngx_http_script_copy_len_code;
1940     copy->len = src[i].skip_empty;
1941
1942
1943     size = (sizeof(ngx_http_script_copy_code_t)
1944             + src[i].key.len + sizeof(uintptr_t) - 1)
1945           & ~(sizeof(uintptr_t) - 1);
1946
1947     copy = ngx_array_push_n(params->values, size);
1948     if (copy == NULL) {
1949         return NGX_ERROR;
1950     }
1951
1952     copy->code = ngx_http_script_copy_code;
1953     copy->len = src[i].key.len;
1954
1955     p = (u_char *) copy + sizeof(ngx_http_script_copy_code_t);
1956     ngx_memcpy(p, src[i].key.data, src[i].key.len);
1957
1958
1959     ngx_memzero(&sc, sizeof(ngx_http_script_compile_t));
1960
1961     sc.cf = cf;

```

```

1962     sc.source = &src[i].value;
1963     sc.flushes = &params->flushes;
1964     sc.lengths = &params->lengths;
1965     sc.values = &params->values;
1966
1967     if (ngx\_http\_script\_compile(&sc) != NGX\_OK) {
1968         return NGX\_ERROR;
1969     }
1970
1971     code = ngx\_array\_push\_n(params->lengths, sizeof(uintptr_t));
1972     if (code == NULL) {
1973         return NGX\_ERROR;
1974     }
1975
1976     *code = (uintptr_t) NULL;
1977
1978
1979     code = ngx\_array\_push\_n(params->values, sizeof(uintptr_t));
1980     if (code == NULL) {
1981         return NGX\_ERROR;
1982     }
1983
1984     *code = (uintptr_t) NULL;
1985 }
1986
1987 code = ngx\_array\_push\_n(params->lengths, sizeof(uintptr_t));
1988 if (code == NULL) {
1989     return NGX\_ERROR;
1990 }
1991
1992 *code = (uintptr_t) NULL;
1993
1994 params->number = headers_names.nelts;
1995
1996 hash.hash = &params->hash;
1997 hash.key = ngx\_hash\_key\_lc;
1998 hash.max_size = 512;
1999 hash.bucket_size = 64;
2000 hash.name = "uwsgi_params_hash";
2001 hash.pool = cf->pool;
2002 hash.temp_pool = NULL;
2003
2004 return ngx\_hash\_init(&hash, headers_names.elts, headers_names.nelts);
2005 }
2006
2007
2008 static char *
2009 ngx\_http\_uwsgi\_pass(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
2010 {
2011     ngx\_http\_uwsgi\_loc\_conf\_t *uwcf = conf;
2012
2013     size_t          add;
2014     ngx\_url\_t       u;
2015     ngx\_str\_t       *value, *url;
2016     ngx\_uint\_t      n;
2017     ngx\_http\_core\_loc\_conf\_t *clcf;
2018     ngx\_http\_script\_compile\_t sc;
2019
2020     if (uwcf->upstream.upstream || uwcf->uwsgi_lengths) {
2021         return "is duplicate";
2022     }
2023
2024     clcf = ngx\_http\_conf\_get\_module\_loc\_conf(cf, ngx\_http\_core\_module);
2025     clcf->handler = ngx\_http\_uwsgi\_handler;
2026
2027     value = cf->args->elts;
2028
2029     url = &value[1];
2030
2031     n = ngx\_http\_script\_variables\_count(url);
2032
2033     if (n) {
2034
2035         ngx\_memzero(&sc, sizeof(ngx\_http\_script\_compile\_t));
2036
2037         sc.cf = cf;

```

```

2038     sc.source = url;
2039     sc.lengths = &uwcf->uwsgi_lengths;
2040     sc.values = &uwcf->uwsgi_values;
2041     sc.variables = n;
2042     sc.complete_lengths = 1;
2043     sc.complete_values = 1;
2044
2045     if (ngx_http_script_compile(&sc) != NGX_OK) {
2046         return NGX_CONF_ERROR;
2047     }
2048
2049 #if (NGX_HTTP_SSL)
2050     uwcf->ssl = 1;
2051 #endif
2052
2053     return NGX_CONF_OK;
2054 }
2055
2056 if (ngx_strncasecmp(url->data, (u_char *) "uwsgi://", 8) == 0) {
2057     add = 8;
2058
2059 } else if (ngx_strncasecmp(url->data, (u_char *) "suwsgi://", 9) == 0) {
2060
2061 #if (NGX_HTTP_SSL)
2062     add = 9;
2063     uwcf->ssl = 1;
2064 #else
2065     ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
2066         "suwsgi protocol requires SSL support");
2067     return NGX_CONF_ERROR;
2068 #endif
2069
2070 } else {
2071     add = 0;
2072 }
2073
2074 ngx_memzero(&u, sizeof(ngx_url_t));
2075
2076 u.url.len = url->len - add;
2077 u.url.data = url->data + add;
2078 u.no_resolve = 1;
2079
2080 uwcf->upstream.upstream = ngx_http_upstream_add(cf, &u, 0);
2081 if (uwcf->upstream.upstream == NULL) {
2082     return NGX_CONF_ERROR;
2083 }
2084
2085 if (clcf->name.data[clcf->name.len - 1] == '/') {
2086     clcf->auto_redirect = 1;
2087 }
2088
2089 return NGX_CONF_OK;
2090 }
2091
2092
2093 static char *
2094 ngx_http_uwsgi_store(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
2095 {
2096     ngx_http_uwsgi_loc_conf_t *uwcf = conf;
2097
2098     ngx_str_t *value;
2099     ngx_http_script_compile_t sc;
2100
2101     if (uwcf->upstream.store != NGX_CONF_UNSET) {
2102         return "is duplicate";
2103     }
2104
2105     value = cf->args->elts;
2106
2107     if (ngx_strcmp(value[1].data, "off") == 0) {
2108         uwcf->upstream.store = 0;
2109         return NGX_CONF_OK;
2110     }
2111
2112 #if (NGX_HTTP_CACHE)
2113

```

```

2114     if (uwcf->upstream.cache > 0) {
2115         return "is incompatible with \"uwsgi_cache\"";
2116     }
2117
2118 #endif
2119
2120     uwcf->upstream.store = 1;
2121
2122     if (ngx_strcmp(value[1].data, "on") == 0) {
2123         return NGX_CONF_OK;
2124     }
2125
2126     /* include the terminating '\0' into script */
2127     value[1].len++;
2128
2129     ngx_memzero(&sc, sizeof(ngx_http_script_compile_t));
2130
2131     sc.cf = cf;
2132     sc.source = &value[1];
2133     sc.lengths = &uwcf->upstream.store_lengths;
2134     sc.values = &uwcf->upstream.store_values;
2135     sc.variables = ngx_http_script_variables_count(&value[1]);
2136     sc.complete_lengths = 1;
2137     sc.complete_values = 1;
2138
2139     if (ngx_http_script_compile(&sc) != NGX_OK) {
2140         return NGX_CONF_ERROR;
2141     }
2142
2143     return NGX_CONF_OK;
2144 }
2145
2146
2147 #if (NGX_HTTP_CACHE)
2148
2149 static char *
2150 ngx_http_uwsgi_cache(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
2151 {
2152     ngx_http_uwsgi_loc_conf_t *uwcf = conf;
2153
2154     ngx_str_t          *value;
2155     ngx_http_complex_value_t  cv;
2156     ngx_http_compile_complex_value_t  ccv;
2157
2158     value = cf->args->elts;
2159
2160     if (uwcf->upstream.cache != NGX_CONF_UNSET) {
2161         return "is duplicate";
2162     }
2163
2164     if (ngx_strcmp(value[1].data, "off") == 0) {
2165         uwcf->upstream.cache = 0;
2166         return NGX_CONF_OK;
2167     }
2168
2169     if (uwcf->upstream.store > 0) {
2170         return "is incompatible with \"uwsgi_store\"";
2171     }
2172
2173     uwcf->upstream.cache = 1;
2174
2175     ngx_memzero(&ccv, sizeof(ngx_http_compile_complex_value_t));
2176
2177     ccv.cf = cf;
2178     ccv.value = &value[1];
2179     ccv.complex_value = &cv;
2180
2181     if (ngx_http_compile_complex_value(&ccv) != NGX_OK) {
2182         return NGX_CONF_ERROR;
2183     }
2184
2185     if (cv.lengths != NULL) {
2186
2187         uwcf->upstream.cache_value = ngx_palloc(cf->pool,
2188             sizeof(ngx_http_complex_value_t));
2189         if (uwcf->upstream.cache_value == NULL) {

```

```

2190         return NGX\_CONF\_ERROR;
2191     }
2192
2193     *uwcf->upstream.cache_value = cv;
2194
2195     return NGX\_CONF\_OK;
2196 }
2197
2198 uwcf->upstream.cache_zone = ngx\_shared\_memory\_add(cf, &value[1], 0,
2199                                             &ngx\_http\_uwsgi\_module);
2200 if (uwcf->upstream.cache_zone == NULL) {
2201     return NGX\_CONF\_ERROR;
2202 }
2203
2204 return NGX\_CONF\_OK;
2205 }
2206
2207
2208 static char *
2209 ngx\_http\_uwsgi\_cache\_key(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
2210 {
2211     ngx\_http\_uwsgi\_loc\_conf\_t *uwcf = conf;
2212
2213     ngx\_str\_t *value;
2214     ngx\_http\_compile\_complex\_value\_t ccv;
2215
2216     value = cf->args->elts;
2217
2218     if (uwcf->cache_key.value.data) {
2219         return "is duplicate";
2220     }
2221
2222     ngx\_memzero(&ccv, sizeof(ngx\_http\_compile\_complex\_value\_t));
2223
2224     ccv.cf = cf;
2225     ccv.value = &value[1];
2226     ccv.complex_value = &uwcf->cache_key;
2227
2228     if (ngx\_http\_compile\_complex\_value(&ccv) != NGX\_OK) {
2229         return NGX\_CONF\_ERROR;
2230     }
2231
2232     return NGX\_CONF\_OK;
2233 }
2234
2235 #endif
2236
2237
2238 #if (NGX\_HTTP\_SSL)
2239
2240 static char *
2241 ngx\_http\_uwsgi\_ssl\_password\_file(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
2242 {
2243     ngx\_http\_uwsgi\_loc\_conf\_t *uwcf = conf;
2244
2245     ngx\_str\_t *value;
2246
2247     if (uwcf->ssl_passwords != NGX\_CONF\_UNSET\_PTR) {
2248         return "is duplicate";
2249     }
2250
2251     value = cf->args->elts;
2252
2253     uwcf->ssl_passwords = ngx\_ssl\_read\_password\_file(cf, &value[1]);
2254
2255     if (uwcf->ssl_passwords == NULL) {
2256         return NGX\_CONF\_ERROR;
2257     }
2258
2259     return NGX\_CONF\_OK;
2260 }
2261
2262
2263 static ngx\_int\_t
2264 ngx\_http\_uwsgi\_set\_ssl(ngx\_conf\_t *cf, ngx\_http\_uwsgi\_loc\_conf\_t *uwcf)
2265 {

```

```

2266     ngx_pool_cleanup_t *cln;
2267
2268     uwcf->upstream.ssl = ngx_palloc(cf->pool, sizeof(ngx_ssl_t));
2269     if (uwcf->upstream.ssl == NULL) {
2270         return NGX_ERROR;
2271     }
2272
2273     uwcf->upstream.ssl->log = cf->log;
2274
2275     if (ngx_ssl_create(uwcf->upstream.ssl, uwcf->ssl_protocols, NULL)
2276         != NGX_OK)
2277     {
2278         return NGX_ERROR;
2279     }
2280
2281     cln = ngx_pool_cleanup_add(cf->pool, 0);
2282     if (cln == NULL) {
2283         return NGX_ERROR;
2284     }
2285
2286     cln->handler = ngx_ssl_cleanup_ctx;
2287     cln->data = uwcf->upstream.ssl;
2288
2289     if (uwcf->ssl_certificate.len) {
2290
2291         if (uwcf->ssl_certificate_key.len == 0) {
2292             ngx_log_error(NGX_LOG_EMERG, cf->log, 0,
2293                 "no \"uwsgi_ssl_certificate_key\" is defined "
2294                 "for certificate \"%V\"", &uwcf->ssl_certificate);
2295             return NGX_ERROR;
2296         }
2297
2298         if (ngx_ssl_certificate(cf, uwcf->upstream.ssl, &uwcf->ssl_certificate,
2299             &uwcf->ssl_certificate_key, uwcf->ssl_passwords)
2300             != NGX_OK)
2301         {
2302             return NGX_ERROR;
2303         }
2304     }
2305
2306     if (SSL_CTX_set_cipher_list(uwcf->upstream.ssl->ctx,
2307         (const char *) uwcf->ssl_ciphers.data)
2308         == 0)
2309     {
2310         ngx_ssl_error(NGX_LOG_EMERG, cf->log, 0,
2311             "SSL_CTX_set_cipher_list(\"%V\") failed",
2312             &uwcf->ssl_ciphers);
2313         return NGX_ERROR;
2314     }
2315
2316     if (uwcf->upstream.ssl_verify) {
2317         if (uwcf->ssl_trusted_certificate.len == 0) {
2318             ngx_log_error(NGX_LOG_EMERG, cf->log, 0,
2319                 "no uwsgi_ssl_trusted_certificate for uwsgi_ssl_verify");
2320             return NGX_ERROR;
2321         }
2322
2323         if (ngx_ssl_trusted_certificate(cf, uwcf->upstream.ssl,
2324             &uwcf->ssl_trusted_certificate,
2325             uwcf->ssl_verify_depth)
2326             != NGX_OK)
2327         {
2328             return NGX_ERROR;
2329         }
2330
2331         if (ngx_ssl_cr1(cf, uwcf->upstream.ssl, &uwcf->ssl_cr1) != NGX_OK) {
2332             return NGX_ERROR;
2333         }
2334     }
2335
2336     return NGX_OK;
2337 }
2338
2339 #endif

```



## src/http/modules/nginx\_http\_xslt\_filter\_module.c - nginx-1.7.10

### Global variables defined

- [ngx\\_http\\_next\\_body\\_filter](#)
- [ngx\\_http\\_next\\_header\\_filter](#)
- [ngx\\_http\\_xslt\\_default\\_types](#)
- [ngx\\_http\\_xslt\\_filter\\_commands](#)
- [ngx\\_http\\_xslt\\_filter\\_module](#)
- [ngx\\_http\\_xslt\\_filter\\_module\\_ctx](#)

### Data types defined

- [ngx\\_http\\_xslt\\_file\\_t](#)
- [ngx\\_http\\_xslt\\_filter\\_ctx\\_t](#)
- [ngx\\_http\\_xslt\\_filter\\_loc\\_conf\\_t](#)
- [ngx\\_http\\_xslt\\_filter\\_main\\_conf\\_t](#)
- [ngx\\_http\\_xslt\\_param\\_t](#)
- [ngx\\_http\\_xslt\\_sheet\\_t](#)

### Functions defined

- [ngx\\_http\\_xslt\\_add\\_chunk](#)
- [ngx\\_http\\_xslt\\_apply\\_stylesheet](#)
- [ngx\\_http\\_xslt\\_body\\_filter](#)
- [ngx\\_http\\_xslt\\_cleanup](#)
- [ngx\\_http\\_xslt\\_cleanup\\_dtd](#)
- [ngx\\_http\\_xslt\\_cleanup\\_stylesheet](#)
- [ngx\\_http\\_xslt\\_content\\_type](#)
- [ngx\\_http\\_xslt\\_encoding](#)
- [ngx\\_http\\_xslt\\_entities](#)
- [ngx\\_http\\_xslt\\_filter\\_create\\_conf](#)
- [ngx\\_http\\_xslt\\_filter\\_create\\_main\\_conf](#)
- [ngx\\_http\\_xslt\\_filter\\_exit](#)
- [ngx\\_http\\_xslt\\_filter\\_init](#)
- [ngx\\_http\\_xslt\\_filter\\_merge\\_conf](#)
- [ngx\\_http\\_xslt\\_filter\\_preconfiguration](#)

- [ngx\\_http\\_xslt\\_header\\_filter](#)
- [ngx\\_http\\_xslt\\_param](#)
- [ngx\\_http\\_xslt\\_params](#)
- [ngx\\_http\\_xslt\\_sax\\_error](#)
- [ngx\\_http\\_xslt\\_sax\\_external\\_subset](#)
- [ngx\\_http\\_xslt\\_send](#)
- [ngx\\_http\\_xslt\\_stylesheet](#)

## Macros defined

- [NGX\\_HTTP\\_XSLT\\_REUSE\\_DTD](#)

## Source code

```

1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10 #include <ngx_http.h>
11
12 #include <libxml/parser.h>
13 #include <libxml/tree.h>
14 #include <libxslt/xslt.h>
15 #include <libxslt/xsltInternals.h>
16 #include <libxslt/transform.h>
17 #include <libxslt/variables.h>
18 #include <libxslt/xsltutils.h>
19
20 #if (NGX_HAVE_EXSLT)
21 #include <libexslt/exslt.h>
22 #endif
23
24
25 #ifndef NGX\_HTTP\_XSLT\_REUSE\_DTD
26 #define NGX\_HTTP\_XSLT\_REUSE\_DTD 1
27 #endif
28
29
30 typedef struct {
31     u_char                *name;
32     void                  *data;
33 } ngx_http_xslt_file_t;
34
35
36 typedef struct {
37     ngx\_array\_t            dtd_files;    /* ngx\_http\_xslt\_file\_t */
38     ngx\_array\_t            sheet_files; /* ngx\_http\_xslt\_file\_t */
39 } ngx_http_xslt_filter_main_conf_t;
40
41
42 typedef struct {
43     u_char                *name;
44     ngx\_http\_complex\_value\_t value;
45     ngx\_uint\_t            quote;        /* unsigned quote:1; */
46 } ngx_http_xslt_param_t;
47
48
49 typedef struct {
50     xsltStylesheetPtr    stylesheet;
51     ngx\_array\_t            params;    /* ngx\_http\_xslt\_param\_t */

```

```

52 } ngx_http_xslt_sheet_t;
53
54
55 typedef struct {
56     xmlDtdPtr          dtd;
57     ngx_array_t        sheets;      /* ngx_http_xslt_sheet_t */
58     ngx_hash_t         types;
59     ngx_array_t        *types_keys;
60     ngx_array_t        *params;     /* ngx_http_xslt_param_t */
61     ngx_flag_t         last_modified;
62 } ngx_http_xslt_filter_loc_conf_t;
63
64
65 typedef struct {
66     xmlDocPtr          doc;
67     xmlParserCtxtPtr   ctxt;
68     xsltTransformContextPtr transform;
69     ngx_http_request_t *request;
70     ngx_array_t        params;
71
72     ngx_uint_t         done;        /* unsigned done:1; */
73 } ngx_http_xslt_filter_ctx_t;
74
75
76 static ngx_int_t ngx_http_xslt_send(ngx_http_request_t *r,
77     ngx_http_xslt_filter_ctx_t *ctx, ngx_buf_t *b);
78 static ngx_int_t ngx_http_xslt_add_chunk(ngx_http_request_t *r,
79     ngx_http_xslt_filter_ctx_t *ctx, ngx_buf_t *b);
80
81
82 static void ngx_http_xslt_sax_external_subset(void *data, const xmlChar *name,
83     const xmlChar *externalId, const xmlChar *systemId);
84 static void ngx_cdecl ngx_http_xslt_sax_error(void *data, const char *msg, ...);
85
86
87 static ngx_buf_t *ngx_http_xslt_apply_stylesheet(ngx_http_request_t *r,
88     ngx_http_xslt_filter_ctx_t *ctx);
89 static ngx_int_t ngx_http_xslt_params(ngx_http_request_t *r,
90     ngx_http_xslt_filter_ctx_t *ctx, ngx_array_t *params, ngx_uint_t final);
91 static u_char *ngx_http_xslt_content_type(xsltStylesheetPtr s);
92 static u_char *ngx_http_xslt_encoding(xsltStylesheetPtr s);
93 static void ngx_http_xslt_cleanup(void *data);
94
95 static char *ngx_http_xslt_entities(ngx_conf_t *cf, ngx_command_t *cmd,
96     void *conf);
97 static char *ngx_http_xslt_stylesheet(ngx_conf_t *cf, ngx_command_t *cmd,
98     void *conf);
99 static char *ngx_http_xslt_param(ngx_conf_t *cf, ngx_command_t *cmd,
100     void *conf);
101 static void ngx_http_xslt_cleanup_dtd(void *data);
102 static void ngx_http_xslt_cleanup_stylesheet(void *data);
103 static void *ngx_http_xslt_filter_create_main_conf(ngx_conf_t *cf);
104 static void *ngx_http_xslt_filter_create_conf(ngx_conf_t *cf);
105 static char *ngx_http_xslt_filter_merge_conf(ngx_conf_t *cf, void *parent,
106     void *child);
107 static ngx_int_t ngx_http_xslt_filter_preconfiguration(ngx_conf_t *cf);
108 static ngx_int_t ngx_http_xslt_filter_init(ngx_conf_t *cf);
109 static void ngx_http_xslt_filter_exit(ngx_cycle_t *cycle);
110
111
112 ngx_str_t ngx_http_xslt_default_types[] = {
113     ngx_string("text/xml"),
114     ngx_null_string
115 };
116
117
118 static ngx_command_t ngx_http_xslt_filter_commands[] = {
119
120     { ngx_string("xml_entities"),
121       NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|NGX_CONF_TAKE1,
122       ngx_http_xslt_entities,
123       NGX_HTTP_LOC_CONF_OFFSET,
124       0,
125       NULL },
126
127     { ngx_string("xslt_stylesheet"),

```

```

128     NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_1MORE,
129     ngx\_http\_xslt\_stylesheet,
130     NGX\_HTTP\_LOC\_CONF\_OFFSET,
131     0,
132     NULL },
133
134     { ngx\_string\("xslt\_param"\),
135     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE2,
136     ngx\_http\_xslt\_param,
137     NGX\_HTTP\_LOC\_CONF\_OFFSET,
138     0,
139     NULL },
140
141     { ngx\_string\("xslt\_string\_param"\),
142     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_TAKE2,
143     ngx\_http\_xslt\_param,
144     NGX\_HTTP\_LOC\_CONF\_OFFSET,
145     0,
146     (void *) 1 },
147
148     { ngx\_string\("xslt\_types"\),
149     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_1MORE,
150     ngx\_http\_types\_slot,
151     NGX\_HTTP\_LOC\_CONF\_OFFSET,
152     offsetof\(ngx\_http\_xslt\_filter\_loc\_conf\_t, types\_keys\),
153     &ngx\_http\_xslt\_default\_types[0] },
154
155     { ngx\_string\("xslt\_last\_modified"\),
156     NGX\_HTTP\_MAIN\_CONF | NGX\_HTTP\_SRV\_CONF | NGX\_HTTP\_LOC\_CONF | NGX\_CONF\_FLAG,
157     ngx\_conf\_set\_flag\_slot,
158     NGX\_HTTP\_LOC\_CONF\_OFFSET,
159     offsetof\(ngx\_http\_xslt\_filter\_loc\_conf\_t, last\_modified\),
160     NULL },
161
162     ngx\_null\_command
163 };
164
165
166 static ngx\_http\_module\_t ngx\_http\_xslt\_filter\_module\_ctx = {
167     ngx\_http\_xslt\_filter\_preconfiguration, /* preconfiguration */
168     ngx\_http\_xslt\_filter\_init,          /* postconfiguration */
169
170     ngx\_http\_xslt\_filter\_create\_main\_conf, /* create main configuration */
171     NULL,                                /* init main configuration */
172
173     NULL,                                /* create server configuration */
174     NULL,                                /* merge server configuration */
175
176     ngx\_http\_xslt\_filter\_create\_conf,    /* create location configuration */
177     ngx\_http\_xslt\_filter\_merge\_conf,    /* merge location configuration */
178 };
179
180
181 ngx\_module\_t ngx\_http\_xslt\_filter\_module = {
182     NGX\_MODULE\_V1,
183     &ngx\_http\_xslt\_filter\_module\_ctx,    /* module context */
184     ngx\_http\_xslt\_filter\_commands,    /* module directives */
185     NGX\_HTTP\_MODULE,                 /* module type */
186     NULL,                             /* init master */
187     NULL,                             /* init module */
188     NULL,                             /* init process */
189     NULL,                             /* init thread */
190     NULL,                             /* exit thread */
191     ngx\_http\_xslt\_filter\_exit,         /* exit process */
192     ngx\_http\_xslt\_filter\_exit,         /* exit master */
193     NGX\_MODULE\_V1\_PADDING
194 };
195
196
197 static ngx\_http\_output\_header\_filter\_pt ngx\_http\_next\_header\_filter;
198 static ngx\_http\_output\_body\_filter\_pt ngx\_http\_next\_body\_filter;
199
200
201 static ngx\_int\_t
202 ngx\_http\_xslt\_header\_filter(ngx\_http\_request\_t *r)
203 {

```

```

204     ngx_http_xslt_filter_ctx_t      *ctx;
205     ngx_http_xslt_filter_loc_conf_t *conf;
206
207     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
208                  "xslt filter header");
209
210     if (r->headers_out.status == NGX_HTTP_NOT_MODIFIED) {
211         return ngx_http_next_header_filter(r);
212     }
213
214     conf = ngx_http_get_module_loc_conf(r, ngx_http_xslt_filter_module);
215
216     if (conf->sheets.nelts == 0
217         || ngx_http_test_content_type(r, &conf->types) == NULL)
218     {
219         return ngx_http_next_header_filter(r);
220     }
221
222     ctx = ngx_http_get_module_ctx(r, ngx_http_xslt_filter_module);
223
224     if (ctx) {
225         return ngx_http_next_header_filter(r);
226     }
227
228     ctx = ngx_palloc(r->pool, sizeof(ngx_http_xslt_filter_ctx_t));
229     if (ctx == NULL) {
230         return NGX_ERROR;
231     }
232
233     ngx_http_set_ctx(r, ctx, ngx_http_xslt_filter_module);
234
235     r->main_filter_need_in_memory = 1;
236
237     return NGX_OK;
238 }
239
240
241 static ngx_int_t
242 ngx_http_xslt_body_filter(ngx_http_request_t *r, ngx_chain_t *in)
243 {
244     int                wellFormed;
245     ngx_chain_t       *cl;
246     ngx_http_xslt_filter_ctx_t *ctx;
247
248     ngx_log_debug0(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
249                  "xslt filter body");
250
251     if (in == NULL) {
252         return ngx_http_next_body_filter(r, in);
253     }
254
255     ctx = ngx_http_get_module_ctx(r, ngx_http_xslt_filter_module);
256
257     if (ctx == NULL || ctx->done) {
258         return ngx_http_next_body_filter(r, in);
259     }
260
261     for (cl = in; cl; cl = cl->next) {
262
263         if (ngx_http_xslt_add_chunk(r, ctx, cl->buf) != NGX_OK) {
264
265             if (ctx->ctxt->myDoc) {
266
267 #if (NGX_HTTP_XSLT_REUSE_DTD)
268                 ctx->ctxt->myDoc->extSubset = NULL;
269 #endif
270                 xmlFreeDoc(ctx->ctxt->myDoc);
271             }
272
273             xmlFreeParserCtxt(ctx->ctxt);
274
275             return ngx_http_xslt_send(r, ctx, NULL);
276         }
277
278         if (cl->buf->last_buf || cl->buf->last_in_chain) {
279

```

```

280         ctx->doc = ctx->ctxt->myDoc;
281
282     #if (NGX_HTTP_XSLT_REUSE_DTD)
283         ctx->doc->extSubset = NULL;
284     #endif
285
286     wellFormed = ctx->ctxt->wellFormed;
287
288     xmlFreeParserCtxt(ctx->ctxt);
289
290     if (wellFormed) {
291         return ngx_http_xslt_send(r, ctx,
292                                 ngx_http_xslt_apply_stylesheet(r, ctx));
293     }
294
295     xmlFreeDoc(ctx->doc);
296
297     ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
298                 "not well formed XML document");
299
300     return ngx_http_xslt_send(r, ctx, NULL);
301 }
302 }
303
304 return NGX_OK;
305 }
306
307
308 static ngx_int_t
309 ngx_http_xslt_send(ngx_http_request_t *r, ngx_http_xslt_filter_ctx_t *ctx,
310                  ngx_buf_t *b)
311 {
312     ngx_int_t          rc;
313     ngx_chain_t        out;
314     ngx_pool_cleanup_t *cIn;
315     ngx_http_xslt_filter_loc_conf_t *conf;
316
317     ctx->done = 1;
318
319     if (b == NULL) {
320         return ngx_http_filter_finalize_request(r, &ngx_http_xslt_filter_module,
321                                               NGX_HTTP_INTERNAL_SERVER_ERROR);
322     }
323
324     cIn = ngx_pool_cleanup_add(r->pool, 0);
325
326     if (cIn == NULL) {
327         ngx_free(b->pos);
328         return ngx_http_filter_finalize_request(r, &ngx_http_xslt_filter_module,
329                                               NGX_HTTP_INTERNAL_SERVER_ERROR);
330     }
331
332     if (r == r->main) {
333         r->headers_out.content_length_n = b->last - b->pos;
334
335         if (r->headers_out.content_length) {
336             r->headers_out.content_length->hash = 0;
337             r->headers_out.content_length = NULL;
338         }
339
340         conf = ngx_http_get_module_loc_conf(r, ngx_http_xslt_filter_module);
341
342         if (!conf->last_modified) {
343             ngx_http_clear_last_modified(r);
344             ngx_http_clear_etag(r);
345         }
346         else {
347             ngx_http_weak_etag(r);
348         }
349     }
350
351     rc = ngx_http_next_header_filter(r);
352
353     if (rc == NGX_ERROR || rc > NGX_OK || r->header_only) {
354         ngx_free(b->pos);
355         return rc;

```

```

356 }
357
358 cIn->handler = ngx_http_xslt_cleanup;
359 cIn->data = b->pos;
360
361 out.buf = b;
362 out.next = NULL;
363
364 return ngx_http_next_body_filter(r, &out);
365 }
366
367
368 static ngx_int_t
369 ngx_http_xslt_add_chunk(ngx_http_request_t *r, ngx_http_xslt_filter_ctx_t *ctx,
370 ngx_buf_t *b)
371 {
372     int err;
373     xmlParserCtxtPtr ctxt;
374
375     if (ctx->ctxt == NULL) {
376
377         ctxt = xmlCreatePushParserCtxt(NULL, NULL, NULL, 0, NULL);
378         if (ctxt == NULL) {
379             ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
380                 "xmlCreatePushParserCtxt() failed");
381             return NGX_ERROR;
382         }
383         xmlCtxtUseOptions(ctxt, XML_PARSE_NOENT|XML_PARSE_DTDLOAD
384             |XML_PARSE_NOWARNING);
385
386         ctxt->sax->externalSubset = ngx_http_xslt_sax_external_subset;
387         ctxt->sax->setDocumentLocator = NULL;
388         ctxt->sax->error = ngx_http_xslt_sax_error;
389         ctxt->sax->fatalError = ngx_http_xslt_sax_error;
390         ctxt->sax->_private = ctx;
391
392         ctx->ctxt = ctxt;
393         ctx->request = r;
394     }
395
396     err = xmlParseChunk(ctxt->ctxt, (char *) b->pos, (int) (b->last - b->pos),
397         (b->last_buf) || (b->last_in_chain));
398
399     if (err == 0) {
400         b->pos = b->last;
401         return NGX_OK;
402     }
403
404     ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
405         "xmlParseChunk() failed, error:%d", err);
406
407     return NGX_ERROR;
408 }
409
410
411 static void
412 ngx_http_xslt_sax_external_subset(void *data, const xmlChar *name,
413     const xmlChar *externalId, const xmlChar *systemId)
414 {
415     xmlParserCtxtPtr ctxt = data;
416
417     xmlDocPtr doc;
418     xmlDtdPtr dtd;
419     ngx_http_request_t *r;
420     ngx_http_xslt_filter_ctx_t *ctx;
421     ngx_http_xslt_filter_loc_conf_t *conf;
422
423     ctx = ctxt->sax->_private;
424     r = ctx->request;
425
426     conf = ngx_http_get_module_loc_conf(r, ngx_http_xslt_filter_module);
427
428     ngx_log_debug3(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
429         "xslt filter extSubset: \"%s\" \"%s\" \"%s\"",
430         name ? name : (xmlChar *) "",
431         externalId ? externalId : (xmlChar *) "",

```

```

432         systemId ? systemId : (xmlChar *) "");
433
434     doc = ctxt->myDoc;
435
436     #if (NGX_HTTP_XSLT_REUSE_DTD)
437
438         dtd = conf->dtd;
439
440     #else
441
442         dtd = xmlCopyDtd(conf->dtd);
443         if (dtd == NULL) {
444             ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
445                 "xmlCopyDtd() failed");
446             return;
447         }
448
449         if (doc->children == NULL) {
450             xmlAddChild((xmlNodePtr) doc, (xmlNodePtr) dtd);
451         } else {
452             xmlAddPrevSibling(doc->children, (xmlNodePtr) dtd);
453         }
454     }
455 #endif
456
457     doc->extSubset = dtd;
458 }
459
460
461 static void ngx_cdecl
462 ngx_http_xslt_sax_error(void *data, const char *msg, ...)
463 {
464     xmlParserCtxtPtr ctxt = data;
465
466     size_t          n;
467     va_list         args;
468     ngx_http_xslt_filter_ctx_t *ctx;
469     u_char          buf[NGX_MAX_ERROR_STR];
470
471     ctx = ctxt->sax->_private;
472
473     buf[0] = '\0';
474
475     va_start(args, msg);
476     n = (size_t) vsnprintf((char *) buf, NGX_MAX_ERROR_STR, msg, args);
477     va_end(args);
478
479     while (--n && (buf[n] == CR || buf[n] == LF)) { /* void */ }
480
481     ngx_log_error(NGX_LOG_ERR, ctx->request->connection->log, 0,
482         "libxml2 error: \"%s\"", n + 1, buf);
483 }
484
485
486 static ngx_buf_t *
487 ngx_http_xslt_apply_stylesheet(ngx_http_request_t *r,
488     ngx_http_xslt_filter_ctx_t *ctx)
489 {
490     int                len, rc, doc_type;
491     u_char             *type, *encoding;
492     ngx_buf_t         *b;
493     ngx_uint_t        i;
494     xmlChar            *buf;
495     xmlDocPtr         doc, res;
496     ngx_http_xslt_sheet_t *sheet;
497     ngx_http_xslt_filter_loc_conf_t *conf;
498
499     conf = ngx_http_get_module_loc_conf(r, ngx_http_xslt_filter_module);
500     sheet = conf->sheets.elts;
501     doc = ctx->doc;
502
503     /* preallocate array for 4 params */
504
505     if (ngx_array_init(&ctx->params, r->pool, 4 * 2 + 1, sizeof(char *))
506         != NGX_OK)

```



```

508     {
509         xmlFreeDoc(doc);
510         return NULL;
511     }
512
513     for (i = 0; i < conf->sheets.nelts; i++) {
514
515         ctx->transform = xsltNewTransformContext(sheet[i].stylesheet, doc);
516         if (ctx->transform == NULL) {
517             xmlFreeDoc(doc);
518             return NULL;
519         }
520
521         if (conf->params
522             && ngx_http_xslt_params(r, ctx, conf->params, 0) != NGX_OK)
523         {
524             xsltFreeTransformContext(ctx->transform);
525             xmlFreeDoc(doc);
526             return NULL;
527         }
528
529         if (ngx_http_xslt_params(r, ctx, &sheet[i].params, 1) != NGX_OK) {
530             xsltFreeTransformContext(ctx->transform);
531             xmlFreeDoc(doc);
532             return NULL;
533         }
534
535         res = xsltApplyStylesheetUser(sheet[i].stylesheet, doc,
536                                     ctx->params.elts, NULL, NULL,
537                                     ctx->transform);
538
539         xsltFreeTransformContext(ctx->transform);
540         xmlFreeDoc(doc);
541
542         if (res == NULL) {
543             ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
544                         "xsltApplyStylesheet() failed");
545             return NULL;
546         }
547
548         doc = res;
549
550         /* reset array elements */
551         ctx->params.nelts = 0;
552     }
553
554     /* there must be at least one stylesheet */
555
556     if (r == r->main) {
557         type = ngx_http_xslt_content_type(sheet[i - 1].stylesheet);
558     } else {
559         type = NULL;
560     }
561
562     encoding = ngx_http_xslt_encoding(sheet[i - 1].stylesheet);
563     doc_type = doc->type;
564
565     ngx_log_debug3(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
566                  "xslt filter type: %d t:%s e:%s",
567                  doc_type, type ? type : (u_char *) "(null)",
568                  encoding ? encoding : (u_char *) "(null)");
569
570
571     rc = xsltSaveResultToString(&buf, &len, doc, sheet[i - 1].stylesheet);
572
573     xmlFreeDoc(doc);
574
575     if (rc != 0) {
576         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
577                     "xsltSaveResultToString() failed");
578         return NULL;
579     }
580
581     if (len == 0) {
582         ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
583                     "xsltSaveResultToString() returned zero-length result");

```

```

584     return NULL;
585 }
586
587 b = ngx_palloc(r->pool, sizeof(ngx_buf_t));
588 if (b == NULL) {
589     ngx_free(buf);
590     return NULL;
591 }
592
593 b->pos = buf;
594 b->last = buf + len;
595 b->memory = 1;
596
597 if (encoding) {
598     r->headers_out.charset.len = ngx_strlen(encoding);
599     r->headers_out.charset.data = encoding;
600 }
601
602 if (r != r->main) {
603     return b;
604 }
605
606 b->last_buf = 1;
607
608 if (type) {
609     len = ngx_strlen(type);
610
611     r->headers_out.content_type_len = len;
612     r->headers_out.content_type.len = len;
613     r->headers_out.content_type.data = type;
614 } else if (doc_type == XML_HTML_DOCUMENT_NODE) {
615
616     r->headers_out.content_type_len = sizeof("text/html") - 1;
617     ngx_str_set(&r->headers_out.content_type, "text/html");
618 }
619
620 r->headers_out.content_type_lowercase = NULL;
621
622 return b;
623 }
624 }
625
626
627 static ngx_int_t
628 ngx_http_xslt_params(ngx_http_request_t *r, ngx_http_xslt_filter_ctx_t *ctx,
629 ngx_array_t *params, ngx_uint_t final)
630 {
631     u_char      *p, *last, *value, *dst, *src, **s;
632     size_t      len;
633     ngx_uint_t   i;
634     ngx_str_t    string;
635     ngx_http_xslt_param_t *param;
636
637     param = params->elts;
638
639     for (i = 0; i < params->nelts; i++) {
640
641         if (ngx_http_complex_value(r, &param[i].value, &string) != NGX_OK) {
642             return NGX_ERROR;
643         }
644
645         ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
646             "xslt filter param: \"%s\"", string.data);
647
648         if (param[i].name) {
649
650             ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
651                 "xslt filter param name: \"%s\"", param[i].name);
652
653             if (param[i].quote) {
654                 if (xsltQuoteOneUserParam(ctx->transform, param[i].name,
655                     string.data)
656                     != 0)
657                 {
658                     ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
659                         "xsltQuoteOneUserParam(\"%s\", \"%s\") failed",

```

```

660         param[i].name, string.data);
661     return NGX\_ERROR;
662 }
663
664     continue;
665 }
666
667     s = ngx\_array\_push(&ctx->params);
668     if (s == NULL) {
669         return NGX\_ERROR;
670     }
671
672     *s = param[i].name;
673
674     s = ngx\_array\_push(&ctx->params);
675     if (s == NULL) {
676         return NGX\_ERROR;
677     }
678
679     *s = string.data;
680
681     continue;
682 }
683
684 /*
685 * parse param1=value1:param2=value2 syntax as used by parameters
686 * specified in xslt_stylesheet directives
687 */
688
689     p = string.data;
690     last = string.data + string.len;
691
692     while (p && *p) {
693
694         value = p;
695         p = (u_char *) ngx\_strchr(p, '=');
696         if (p == NULL) {
697             ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, 0,
698                 "invalid libxslt parameter \"%s\"", value);
699             return NGX\_ERROR;
700         }
701         *p++ = '\0';
702
703         ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
704             "xslt filter param name: \"%s\"", value);
705
706         s = ngx\_array\_push(&ctx->params);
707         if (s == NULL) {
708             return NGX\_ERROR;
709         }
710
711         *s = value;
712
713         value = p;
714         p = (u_char *) ngx\_strchr(p, ':');
715
716         if (p) {
717             len = p - value;
718             *p++ = '\0';
719         } else {
720             len = last - value;
721         }
722
723         ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
724             "xslt filter param value: \"%s\"", value);
725
726         dst = value;
727         src = value;
728
729         ngx\_unescape\_uri(&dst, &src, len, 0);
730
731         *dst = '\0';
732
733         ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,
734             "xslt filter param unescaped: \"%s\"", value);
735

```

```

736         s = ngx_array_push(&ctx->params);
737         if (s == NULL) {
738             return NGX_ERROR;
739         }
740     }
741     *s = value;
742 }
743 }
744 }
745
746 if (final) {
747     s = ngx_array_push(&ctx->params);
748     if (s == NULL) {
749         return NGX_ERROR;
750     }
751     *s = NULL;
752 }
753 }
754
755 return NGX_OK;
756 }
757
758
759 static u_char *
760 ngx_http_xslt_content_type(xsltStylesheetPtr s)
761 {
762     u_char *type;
763
764     if (s->mediaType) {
765         return s->mediaType;
766     }
767
768     for (s = s->imports; s; s = s->next) {
769
770         type = ngx_http_xslt_content_type(s);
771
772         if (type) {
773             return type;
774         }
775     }
776
777     return NULL;
778 }
779
780
781 static u_char *
782 ngx_http_xslt_encoding(xsltStylesheetPtr s)
783 {
784     u_char *encoding;
785
786     if (s->encoding) {
787         return s->encoding;
788     }
789
790     for (s = s->imports; s; s = s->next) {
791
792         encoding = ngx_http_xslt_encoding(s);
793
794         if (encoding) {
795             return encoding;
796         }
797     }
798
799     return NULL;
800 }
801
802
803 static void
804 ngx_http_xslt_cleanup(void *data)
805 {
806     ngx_free(data);
807 }
808
809
810 static char *
811 ngx_http_xslt_entities(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)

```

```

812 {
813     ngx\_http\_xslt\_filter\_loc\_conf\_t *xlcf = conf;
814
815     ngx\_str\_t                *value;
816     ngx\_uint\_t              i;
817     ngx\_pool\_cleanup\_t      *cln;
818     ngx\_http\_xslt\_file\_t    *file;
819     ngx\_http\_xslt\_filter\_main\_conf\_t *xmcf;
820
821     if (xlcf->dtd) {
822         return "is duplicate";
823     }
824
825     value = cf->args->elts;
826
827     xmcf = ngx\_http\_conf\_get\_module\_main\_conf(cf, ngx\_http\_xslt\_filter\_module);
828
829     file = xmcf->dtd_files.elts;
830     for (i = 0; i < xmcf->dtd_files.nelts; i++) {
831         if (ngx\_strcmp(file[i].name, value[1].data) == 0) {
832             xlcf->dtd = file[i].data;
833             return NGX\_CONF\_OK;
834         }
835     }
836
837     cln = ngx\_pool\_cleanup\_add(cf->pool, 0);
838     if (cln == NULL) {
839         return NGX\_CONF\_ERROR;
840     }
841
842     xlcf->dtd = xmlParseDTD(NULL, (xmlChar *) value[1].data);
843
844     if (xlcf->dtd == NULL) {
845         ngx\_conf\_log\_error(NGX\_LOG\_ERR, cf, 0, "xmlParseDTD() failed");
846         return NGX\_CONF\_ERROR;
847     }
848
849     cln->handler = ngx\_http\_xslt\_cleanup\_dtd;
850     cln->data = xlcf->dtd;
851
852     file = ngx\_array\_push(&xmcf->dtd_files);
853     if (file == NULL) {
854         return NGX\_CONF\_ERROR;
855     }
856
857     file->name = value[1].data;
858     file->data = xlcf->dtd;
859
860     return NGX\_CONF\_OK;
861 }
862
863
864
865 static char *
866 ngx\_http\_xslt\_stylesheet(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
867 {
868     ngx\_http\_xslt\_filter\_loc\_conf\_t *xlcf = conf;
869
870     ngx\_str\_t                *value;
871     ngx\_uint\_t              i, n;
872     ngx\_pool\_cleanup\_t      *cln;
873     ngx\_http\_xslt\_file\_t    *file;
874     ngx\_http\_xslt\_sheet\_t   *sheet;
875     ngx\_http\_xslt\_param\_t   *param;
876     ngx\_http\_compile\_complex\_value\_t ccv;
877     ngx\_http\_xslt\_filter\_main\_conf\_t *xmcf;
878
879     value = cf->args->elts;
880
881     if (xlcf->sheets.elts == NULL) {
882         if (ngx\_array\_init(&xlcf->sheets, cf->pool, 1,
883             sizeof(ngx\_http\_xslt\_sheet\_t))
884             != NGX\_OK)
885         {
886             return NGX\_CONF\_ERROR;
887         }
888     }

```

```

888 }
889
890 sheet = ngx_array_push(&xlcf->sheets);
891 if (sheet == NULL) {
892     return NGX_CONF_ERROR;
893 }
894
895 ngx_memzero(sheet, sizeof(ngx_http_xslt_sheet_t));
896
897 if (ngx_conf_full_name(cf->cycle, &value[1], 0) != NGX_OK) {
898     return NGX_CONF_ERROR;
899 }
900
901 xmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_xslt_filter_module);
902
903 file = xmcf->sheet_files.elts;
904 for (i = 0; i < xmcf->sheet_files.nelts; i++) {
905     if (ngx_strcmp(file[i].name, value[1].data) == 0) {
906         sheet->stylesheet = file[i].data;
907         goto found;
908     }
909 }
910
911 cln = ngx_pool_cleanup_add(cf->pool, 0);
912 if (cln == NULL) {
913     return NGX_CONF_ERROR;
914 }
915
916 sheet->stylesheet = xsltParseStylesheetFile(value[1].data);
917 if (sheet->stylesheet == NULL) {
918     ngx_conf_log_error(NGX_LOG_ERR, cf, 0,
919         "xsltParseStylesheetFile(\"%s\") failed",
920         value[1].data);
921     return NGX_CONF_ERROR;
922 }
923
924 cln->handler = ngx_http_xslt_cleanup_stylesheet;
925 cln->data = sheet->stylesheet;
926
927 file = ngx_array_push(&xmcf->sheet_files);
928 if (file == NULL) {
929     return NGX_CONF_ERROR;
930 }
931
932 file->name = value[1].data;
933 file->data = sheet->stylesheet;
934
935 found:
936
937 n = cf->args->nelts;
938
939 if (n == 2) {
940     return NGX_CONF_OK;
941 }
942
943 if (ngx_array_init(&sheet->params, cf->pool, n - 2,
944     sizeof(ngx_http_xslt_param_t))
945     != NGX_OK)
946 {
947     return NGX_CONF_ERROR;
948 }
949
950 for (i = 2; i < n; i++) {
951
952     param = ngx_array_push(&sheet->params);
953     if (param == NULL) {
954         return NGX_CONF_ERROR;
955     }
956
957     ngx_memzero(param, sizeof(ngx_http_xslt_param_t));
958     ngx_memzero(&ccv, sizeof(ngx_http_compile_complex_value_t));
959
960     ccv.cf = cf;
961     ccv.value = &value[i];
962     ccv.complex_value = &param->value;
963     ccv.zero = 1;

```

```

964     if (ngx\_http\_compile\_complex\_value(&ccv) != NGX\_OK) {
965         return NGX\_CONF\_ERROR;
966     }
967 }
968
969 return NGX\_CONF\_OK;
970 }
971
972
973
974 static char *
975 ngx\_http\_xslt\_param(ngx\_conf\_t *cf, ngx\_command\_t *cmd, void *conf)
976 {
977     ngx\_http\_xslt\_filter\_loc\_conf\_t *xlcf = conf;
978
979     ngx\_http\_xslt\_param\_t *param;
980     ngx\_http\_compile\_complex\_value\_t ccv;
981     ngx\_str\_t *value;
982
983     value = cf->args->elts;
984
985     if (xlcf->params == NULL) {
986         xlcf->params = ngx\_array\_create(cf->pool, 2,
987             sizeof(ngx\_http\_xslt\_param\_t));
988         if (xlcf->params == NULL) {
989             return NGX\_CONF\_ERROR;
990         }
991     }
992
993     param = ngx\_array\_push(xlcf->params);
994     if (param == NULL) {
995         return NGX\_CONF\_ERROR;
996     }
997
998     param->name = value[1].data;
999     param->quote = (cmd->post == NULL) ? 0 : 1;
1000
1001     ngx\_memzero(&ccv, sizeof(ngx\_http\_compile\_complex\_value\_t));
1002
1003     ccv.cf = cf;
1004     ccv.value = &value[2];
1005     ccv.complex_value = &param->value;
1006     ccv.zero = 1;
1007
1008     if (ngx\_http\_compile\_complex\_value(&ccv) != NGX\_OK) {
1009         return NGX\_CONF\_ERROR;
1010     }
1011
1012     return NGX\_CONF\_OK;
1013 }
1014
1015
1016 static void
1017 ngx\_http\_xslt\_cleanup\_dtd(void *data)
1018 {
1019     xmlFreeDtd(data);
1020 }
1021
1022
1023 static void
1024 ngx\_http\_xslt\_cleanup\_stylesheet(void *data)
1025 {
1026     xsltFreeStylesheet(data);
1027 }
1028
1029
1030 static void *
1031 ngx\_http\_xslt\_filter\_create\_main\_conf(ngx\_conf\_t *cf)
1032 {
1033     ngx\_http\_xslt\_filter\_main\_conf\_t *conf;
1034
1035     conf = ngx\_palloc(cf->pool, sizeof(ngx\_http\_xslt\_filter\_main\_conf\_t));
1036     if (conf == NULL) {
1037         return NULL;
1038     }
1039 }

```

```

1040     if (ngx\_array\_init(&conf->dtd_files, cf->pool, 1,
1041                          sizeof(ngx\_http\_xslt\_file\_t))
1042         != NGX\_OK)
1043     {
1044         return NULL;
1045     }
1046
1047     if (ngx\_array\_init(&conf->sheet_files, cf->pool, 1,
1048                          sizeof(ngx\_http\_xslt\_file\_t))
1049         != NGX\_OK)
1050     {
1051         return NULL;
1052     }
1053
1054     return conf;
1055 }
1056
1057
1058 static void *
1059 ngx\_http\_xslt\_filter\_create\_conf(ngx\_conf\_t *cf)
1060 {
1061     ngx\_http\_xslt\_filter\_loc\_conf\_t *conf;
1062
1063     conf = ngx\_palloc(cf->pool, sizeof(ngx\_http\_xslt\_filter\_loc\_conf\_t));
1064     if (conf == NULL) {
1065         return NULL;
1066     }
1067
1068     /*
1069      * set by ngx\_palloc\(\):
1070      *
1071      *     conf->dtd = NULL;
1072      *     conf->sheets = { NULL };
1073      *     conf->types = { NULL };
1074      *     conf->types_keys = NULL;
1075      *     conf->params = NULL;
1076      */
1077
1078     conf->last_modified = NGX\_CONF\_UNSET;
1079
1080     return conf;
1081 }
1082
1083
1084 static char *
1085 ngx\_http\_xslt\_filter\_merge\_conf(ngx\_conf\_t *cf, void *parent, void *child)
1086 {
1087     ngx\_http\_xslt\_filter\_loc\_conf\_t *prev = parent;
1088     ngx\_http\_xslt\_filter\_loc\_conf\_t *conf = child;
1089
1090     if (conf->dtd == NULL) {
1091         conf->dtd = prev->dtd;
1092     }
1093
1094     if (conf->sheets.nelts == 0) {
1095         conf->sheets = prev->sheets;
1096     }
1097
1098     if (conf->params == NULL) {
1099         conf->params = prev->params;
1100     }
1101
1102     if (ngx\_http\_merge\_types(cf, &conf->types_keys, &conf->types,
1103                             &prev->types_keys, &prev->types,
1104                             ngx\_http\_xslt\_default\_types)
1105         != NGX\_OK)
1106     {
1107         return NGX\_CONF\_ERROR;
1108     }
1109
1110     ngx\_conf\_merge\_value(conf->last_modified, prev->last_modified, 0);
1111
1112     return NGX\_CONF\_OK;
1113 }
1114
1115

```



```
1116 static ngx_int_t
1117 ngx_http_xslt_filter_preconfiguration(ngx_conf_t *cf)
1118 {
1119     xmlInitParser();
1120
1121     #if (NGX_HAVE_EXSLT)
1122     exsltRegisterAll();
1123     #endif
1124
1125     return NGX_OK;
1126 }
1127
1128
1129 static ngx_int_t
1130 ngx_http_xslt_filter_init(ngx_conf_t *cf)
1131 {
1132     ngx_http_next_header_filter = ngx_http_top_header_filter;
1133     ngx_http_top_header_filter = ngx_http_xslt_header_filter;
1134
1135     ngx_http_next_body_filter = ngx_http_top_body_filter;
1136     ngx_http_top_body_filter = ngx_http_xslt_body_filter;
1137
1138     return NGX_OK;
1139 }
1140
1141
1142 static void
1143 ngx_http_xslt_filter_exit(ngx_cycle_t *cycle)
1144 {
1145     xsltCleanupGlobals();
1146     xmlCleanupParser();
1147 }
```

[One Level Up](#)

[Top Level](#)

# src/http/modules/perl/nginx.pm - nginx-1.7.10

## Data types defined

- [rflush](#)

## Source code

```
1 package nginx;
2
3 use 5.006001;
4 use strict;
5 use warnings;
6
7 require Exporter;
8
9 our @ISA = qw(Exporter);
10
11 our @EXPORT = qw(
12     OK
13     DECLINED
14
15     HTTP_OK
16     HTTP_CREATED
17     HTTP_ACCEPTED
18     HTTP_NO_CONTENT
19     HTTP_PARTIAL_CONTENT
20
21     HTTP_MOVED_PERMANENTLY
22     HTTP_MOVED_TEMPORARILY
23     HTTP_REDIRECT
24     HTTP_SEE_OTHER
25     HTTP_NOT_MODIFIED
26     HTTP_TEMPORARY_REDIRECT
27
28     HTTP_BAD_REQUEST
29     HTTP_UNAUTHORIZED
30     HTTP_PAYMENT_REQUIRED
31     HTTP_FORBIDDEN
32     HTTP_NOT_FOUND
33     HTTP_NOT_ALLOWED
34     HTTP_NOT_ACCEPTABLE
35     HTTP_REQUEST_TIME_OUT
36     HTTP_CONFLICT
37     HTTP_GONE
38     HTTP_LENGTH_REQUIRED
39     HTTP_REQUEST_ENTITY_TOO_LARGE
40     HTTP_REQUEST_URI_TOO_LARGE
41     HTTP_UNSUPPORTED_MEDIA_TYPE
42     HTTP_RANGE_NOT_SATISFIABLE
43
44     HTTP_INTERNAL_SERVER_ERROR
45     HTTP_SERVER_ERROR
46     HTTP_NOT_IMPLEMENTED
47     HTTP_BAD_GATEWAY
48     HTTP_SERVICE_UNAVAILABLE
49     HTTP_GATEWAY_TIME_OUT
50     HTTP_INSUFFICIENT_STORAGE
51 );
52
53 our $VERSION = '%%VERSION%%';
54
55 require XSLoader;
56 XSLoader::load('nginx', $VERSION);
57
58 # Preloaded methods go here.
59
60 use constant OK => 0;
61 use constant DECLINED => -5;
62
```

```

63 use constant HTTP_OK                => 200;
64 use constant HTTP_CREATED           => 201;
65 use constant HTTP_ACCEPTED         => 202;
66 use constant HTTP_NO_CONTENT       => 204;
67 use constant HTTP_PARTIAL_CONTENT  => 206;
68
69 use constant HTTP_MOVED_PERMANENTLY => 301;
70 use constant HTTP_MOVED_TEMPORARILY => 302;
71 use constant HTTP_REDIRECT         => 302;
72 use constant HTTP_SEE_OTHER        => 303;
73 use constant HTTP_NOT_MODIFIED     => 304;
74 use constant HTTP_TEMPORARY_REDIRECT => 307;
75
76 use constant HTTP_BAD_REQUEST      => 400;
77 use constant HTTP_UNAUTHORIZED     => 401;
78 use constant HTTP_PAYMENT_REQUIRED => 402;
79 use constant HTTP_FORBIDDEN        => 403;
80 use constant HTTP_NOT_FOUND        => 404;
81 use constant HTTP_NOT_ALLOWED      => 405;
82 use constant HTTP_NOT_ACCEPTABLE   => 406;
83 use constant HTTP_REQUEST_TIME_OUT => 408;
84 use constant HTTP_CONFLICT         => 409;
85 use constant HTTP_GONE             => 410;
86 use constant HTTP_LENGTH_REQUIRED  => 411;
87 use constant HTTP_REQUEST_ENTITY_TOO_LARGE => 413;
88 use constant HTTP_REQUEST_URI_TOO_LARGE => 414;
89 use constant HTTP_UNSUPPORTED_MEDIA_TYPE => 415;
90 use constant HTTP_RANGE_NOT_SATISFIABLE => 416;
91
92 use constant HTTP_INTERNAL_SERVER_ERROR => 500;
93 use constant HTTP_SERVER_ERROR       => 500;
94 use constant HTTP_NOT_IMPLEMENTED    => 501;
95 use constant HTTP_BAD_GATEWAY        => 502;
96 use constant HTTP_SERVICE_UNAVAILABLE => 503;
97 use constant HTTP_GATEWAY_TIME_OUT   => 504;
98 use constant HTTP_INSUFFICIENT_STORAGE => 507;
99
100
101 sub rflush {
102     my $r = shift;
103
104     $r->flush;
105 }
106
107
108 1;
109 __END__
110
111 =head1 NAME
112
113 nginx - Perl interface to the nginx HTTP server API
114
115 =head1 SYNOPSIS
116
117     use nginx;
118
119 =head1 DESCRIPTION
120
121 This module provides a Perl interface to the nginx HTTP server API.
122
123
124 =head1 SEE ALSO
125
126 http://nginx.org/en/docs/http/nginx\_http\_perl\_module.html
127
128 =head1 AUTHOR
129
130 Igor Sysoev
131
132 =head1 COPYRIGHT AND LICENSE
133
134 Copyright (C) Igor Sysoev
135 Copyright (C) Nginx, Inc.
136
137
138 =cut

```

[One Level Up](#)

[Top Level](#)

## src/core/nginx\_crypt.h - nginx-1.7.10

### Macros defined

- [\\_NGX\\_CRYPT\\_H\\_INCLUDED](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_CRYPT\_H\_INCLUDED
9 #define \_NGX\_CRYPT\_H\_INCLUDED
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 ngx\_int\_t ngx_crypt(ngx\_pool\_t *pool, u_char *key, u_char *salt,
17     u_char **encrypted);
18
19
20 #endif /* \_NGX\_CRYPT\_H\_INCLUDED */
```

## src/core/nginx\_murmurhash.h - nginx-1.7.10

### Macros defined

- [\\_NGX\\_MURMURHASH\\_H\\_INCLUDED](#)

### Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_MURMURHASH\_H\_INCLUDED
9 #define \_NGX\_MURMURHASH\_H\_INCLUDED
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 uint32_t ngx\_murmur\_hash2(u_char *data, size_t len);
17
18
19 #endif /* \_NGX\_MURMURHASH\_H\_INCLUDED */
```

# src/core/nginx\_parse.h - nginx-1.7.10

## Macros defined

- [\\_NGX\\_PARSE\\_H\\_INCLUDED\\_](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #ifndef \_NGX\_PARSE\_H\_INCLUDED\_
9 #define \_NGX\_PARSE\_H\_INCLUDED\_
10
11
12 #include <ngx_config.h>
13 #include <ngx_core.h>
14
15
16 ssize_t ngx\_parse\_size(ngx_str_t *line);
17 off_t ngx\_parse\_offset(ngx_str_t *line);
18 ngx_int_t ngx\_parse\_time(ngx_str_t *line, ngx_uint_t is_sec);
19
20
21 #endif /* \_NGX\_PARSE\_H\_INCLUDED\_ */
```

[One Level Up](#)

[Top Level](#)

## src/misc/ - nginx-1.7.10

- [ngx\\_cpp\\_test\\_module.cpp](#)
- [ngx\\_google\\_perftools\\_module.c](#)

[One Level Up](#)

[Top Level](#)



## src/misc/nginx\_cpp\_test\_module.cpp - nginx-1.7.10

### Functions defined

- [ngx\\_cpp\\_test\\_handler](#)

### Source code

```
1
2 // stub module to test header files' C++ compatibility
3
4 extern "C" {
5     #include <ngx_config.h>
6     #include <ngx_core.h>
7     #include <ngx_event.h>
8     #include <ngx_event_connect.h>
9     #include <ngx_event_pipe.h>
10
11     #include <ngx_http.h>
12
13     #include <ngx_mail.h>
14     #include <ngx_mail_pop3_module.h>
15     #include <ngx_mail_imap_module.h>
16     #include <ngx_mail_smtp_module.h>
17 }
18
19 // nginx header files should go before other, because they define 64-bit off_t
20 // #include <string>
21
22
23 void ngx_cpp_test_handler(void *data);
24
25 void
26 ngx_cpp_test_handler(void *data)
27 {
28     return;
29 }
```

# src/misc/nginx\_google\_perftools\_module.c - nginx-1.7.10

## Global variables defined

- [ngx\\_google\\_perftools\\_commands](#)
- [ngx\\_google\\_perftools\\_module](#)
- [ngx\\_google\\_perftools\\_module\\_ctx](#)

## Data types defined

- [ngx\\_google\\_perftools\\_conf\\_t](#)

## Functions defined

- [ngx\\_google\\_perftools\\_create\\_conf](#)
- [ngx\\_google\\_perftools\\_worker](#)

## Source code

```
1
2 /*
3  * Copyright (C) Igor Sysoev
4  * Copyright (C) Nginx, Inc.
5  */
6
7
8 #include <ngx_config.h>
9 #include <ngx_core.h>
10
11 /*
12  * declare Profiler interface here because
13  * <google/profiler.h> is C++ header file
14  */
15
16 int ProfilerStart(u_char* fname);
17 void ProfilerStop(void);
18 void ProfilerRegisterThread(void);
19
20
21 static void *ngx_google_perftools_create_conf(ngx_cycle_t *cycle);
22 static ngx_int_t ngx_google_perftools_worker(ngx_cycle_t *cycle);
23
24
25 typedef struct {
26     ngx_str_t profiles;
27 } ngx_google_perftools_conf_t;
28
29
30 static ngx_command_t ngx_google_perftools_commands[] = {
31     { ngx_string("google_perftools_profiles"),
32       NGX_MAIN_CONF|NGX_DIRECT_CONF|NGX_CONF_TAKE1,
33       ngx_conf_set_str_slot,
34       0,
35       offsetof(ngx_google_perftools_conf_t, profiles),
36       NULL },
37     ngx_null_command
38 };
39
40
41
42
43 static ngx_core_module_t ngx_google_perftools_module_ctx = {
44     ngx_string("google_perftools"),
```

```

45     ngx\_google\_perftools\_create\_conf,
46     NULL
47 };
48
49
50 ngx\_module\_t ngx\_google\_perftools\_module = {
51     NGX\_MODULE\_V1,
52     &ngx\_google\_perftools\_module\_ctx,      /* module context */
53     ngx\_google\_perftools\_commands,      /* module directives */
54     NGX\_CORE\_MODULE,                    /* module type */
55     NULL,                                /* init master */
56     NULL,                                /* init module */
57     ngx\_google\_perftools\_worker,        /* init process */
58     NULL,                                /* init thread */
59     NULL,                                /* exit thread */
60     NULL,                                /* exit process */
61     NULL,                                /* exit master */
62     NGX\_MODULE\_V1\_PADDING
63 };
64
65
66 static void *
67 ngx\_google\_perftools\_create\_conf(ngx\_cycle\_t *cycle)
68 {
69     ngx\_google\_perftools\_conf\_t *gptcf;
70
71     gptcf = ngx\_palloc(cycle->pool, sizeof(ngx\_google\_perftools\_conf\_t));
72     if (gptcf == NULL) {
73         return NULL;
74     }
75
76     /*
77      * set by ngx\_palloc()
78      *
79      * gptcf->profiles = { 0, NULL };
80      */
81
82     return gptcf;
83 }
84
85
86 static ngx\_int\_t
87 ngx\_google\_perftools\_worker(ngx\_cycle\_t *cycle)
88 {
89     u_char *profile;
90     ngx\_google\_perftools\_conf\_t *gptcf;
91
92     gptcf = (ngx\_google\_perftools\_conf\_t *)
93         ngx\_get\_conf(cycle->conf_ctx, ngx\_google\_perftools\_module);
94
95     if (gptcf->profiles.len == 0) {
96         return NGX\_OK;
97     }
98
99     profile = ngx\_alloc(gptcf->profiles.len + NGX\_INT\_T\_LEN + 2, cycle->log);
100     if (profile == NULL) {
101         return NGX\_OK;
102     }
103
104     if (getenv("CPUPROFILE")) {
105         /* disable inherited Profiler enabled in master process */
106         ProfilerStop();
107     }
108
109     ngx\_sprintf(profile, "%V.%d%Z", &gptcf->profiles, ngx\_pid);
110
111     if (ProfilerStart(profile)) {
112         /* start ITIMER_PROF timer */
113         ProfilerRegisterThread();
114     } else {
115         ngx\_log\_error(NGX\_LOG\_CRIT, cycle->log, ngx\_errno,
116             "ProfilerStart(%s) failed", profile);
117     }
118
119     ngx\_free(profile);
120

```

```
121
122     return NGX\_OK;
123 }
124
125
126 /* ProfilerStop() is called on Profiler destruction */
```

[One Level Up](#)

[Top Level](#)