nginx cache 学习总结

1.1 proxy_cache

syntax:	proxy_cache zone| off;
default:	proxy_cache off;
context:	http, server, location

定义一块共享内存区域用来进行缓存。相同名称的缓存区域可以在多个地方使用。Off参数关闭从前一个级别配置中继承下来的缓存设置

1.2 proxy_cache_bypass

yntax:	proxy_cache_bypass string…;
default:	—
context:	http, server, location

定义一个条件，在这个条件成立后将不会从缓存中获取数据。至少有一个字符串参数不为空，并且是不等于“0”，则响应不会从缓存中获取。

proxy_cache_bypass $cookie_nocache $arg_nocache$arg_comment;

proxy_cache_bypass $http_pragma $http_authorization;

Can be used along with the proxy_no_cache directive.

1.3 proxy_cache_key

syntax:	proxy_cache_key string;
	proxy_cache_key

default:	$scheme$proxy_host$request_uri;
context:	http, server, location

给缓存数据定义一个键，例如

proxy_cache_key “$host$request_uri $cookie_user”;

默认情况下，该指令的值的字符串

proxy_cache_key $scheme$proxy_hosturiis_args$args;

1.4 proxy_cache_lock

syntax:	proxy_cache_lock on | off;
default:	proxy_cache_lock off;
context:	http, server, location

This directive appeared in version 1.1.12.

当指令被指定时，根据 proxy_cache_key指令确定的若干个或得出相同响应元素若干个请求仅仅有一个能被传递给后端的代理服务器（最终传递给应用服务器）去生成响应内容，生成后响应内容添加到cache中， 其他请求从cache中获取数据。其他请求将等待cache中有内容出现或者等待超时为止。

1.5 proxy_cache_lock_timeout

syntax:	proxy_cache_lock_timeout time;
default:	proxy_cache_lock_timeout 5s;
context:	http, server, location

This directive appeared in version 1.1.12.

为proxy_cache_lock指令设置一个超时时间

1.6 proxy_cache_min_uses

syntax:	proxy_cache_min_uses number;
default:	proxy_cache_min_uses 1;
context:	http, server, location

设置请求发生多少次后，请求（响应内容）被缓存

1.7 proxy_cache_path

syntax:	proxy_cache_path path[levels=levels] keys_zone=name:size [inactive=time] [max_size=size] [loader_files=number] [loader_sleep=time] [loader_threshold=time];
default:	—
context:	http

设置高速缓存的路径和其他参数。缓存数据存储在文件中。在高速缓存中缓存内容文件名以及对应的key都用MD5函数处理。Levels参数确定缓存文件的层级关系。例如下面的配置文件

proxy_cache_path /data/nginx/cache levels=1:2 keys_zone=one:10m;

缓存后的文件名将如下：

/ data/nginx/cache/c/29/b7f54b2df7773722d382f4809d65029c

一个缓存的响应，首先被写入到一个临时文件，然后重命名一个文件。从版本0.8.9临时文件和缓存可以放在不同的文件系统，但在这种情况下，一个文件复制在两个文件系统中发生，而不是廉价的重命名操作。因此建议，对于任何给定的位置，缓存目录和proxy_temp_path指令所设定的临时文件都放在同一个文件系统中。

此外，所有活跃的数据信息包括其key都存储在一个共享内存区域中，这个区域的名称及大小等keys_zone指令的参数进行配置。被缓存在内存中的数据在一定时间没有被访问后将变成不活跃，这些数据将被从内存中移除无论数据是否还是有效，不活跃时间由inative 参数设置。默认情况下inactive是10分钟。

“高速缓存管理器”进程将监控max_size参数设置最大缓存大小，超过此大小时，它消除了最近最少使用的数据。

一分钟后开始的指定“cache loader”进程被激活，他将加载以前缓存在文件系统中的缓存数据到内存中。一个加载包括在若干个迭代过程。在一个迭代过程中加载文件数量小于loader_files 参数指定的数量（默认值100）.除此以外迭代时间小于loader_threshold限制的值，默认值200毫秒。在两个迭代中间有个短暂的暂停（加载的暂停，nginx其他工作没有暂停），这个时间被loader_sleep参数所控制，默认值50毫秒。

1.8 proxy_cache_use_stale

syntax:	proxy_cache_use_stale error | timeout | invalid_header | updating | http_500 | http_502 | http_503 | http_504 | http_404 | off

	…;
default:	proxy_cache_use_stale off;
context:	http, server, location

如果nginx同代理服务器工作过程中发生一个错误，nginx可能使用一个陈旧的被缓存的数据。这个指令决定在那种情况下使用这个功能。这个指令的参数会同proxy_next_upstream指令的参数相匹配。

此外，如果目前正在更新，更新参数允许使用过时的缓存的响应。这可以最大限度地减少更新缓存数据时，代理服务器的访问次数。

为了尽量减少当填充一个新缓存元素时访问代理服务器次数过多的问题，可以使用proxy_cache_lock指令，来缓解。

1.9 proxy_cache_valid

syntax:	proxy_cache_valid [code…] time;
default:	—
context:	http, server, location

设置不同响应代码的缓存时间，例如如下指令：

```
proxy_cache_valid 200 302   10m;

proxy_cache_valid 404       1m;
```

设置缓存的响应代码为200和302时间为10分钟，代码为404的响应缓存时间为1分钟。

如果只指定缓存时间

proxy_cache_valid 5m;

然后只有200，301和302响应被缓存。

此外，它可以被指定缓存任何响应通过使用any参数：

proxy_cache_valid 200 302 10m;

proxy_cache_valid 301 1h;

proxy_cache_valid any 1m;

缓存的参数也能通过响应头直接设置。这个设置具有更高优先级比通过缓存指令设置的缓存时间。“X-ACCEL-Expires”头字段以秒为单位设置响应的缓存时间。0禁止缓存响应。

如果一个值以前缀@开始，将针对响应设置一个绝对过期时间（自1970年1月1日0时到达过期时间的绝对秒数）的缓存。若是响应头中没有 “X-Accel-Expires” 字段，则缓存参数可以通过“Expires” or “Cache-Control”字段进行设置。若是一个响应头包括“Set-Cookie” 字段，则这样的响应将不被缓存。这样的一个或者多个响应头字段的处理可以通过proxy_ignore_headers 指令进行禁止。

1.10 proxy_no_cache

syntax:	proxy_no_cache string…;
default:	—
context:	http, server, location

定义条件下的反应将不会被保存到高速缓存。如果至少有一个值的字符串参数不为空，不等于“0”，那么响应将不会被保存：

proxy_no_cache$ cookie_nocache$ arg_nocache$ arg_comment;

proxy_no_cache$ http_pragma HTTP_AUTHORIZATION;

可用于沿的proxy_cache_bypass指令。

定义一些条件，当响应满足这些条件时将不被缓存起来。在条件字符串中至少有一个条件不为空或者0，符合这样条件的响应才不会被缓存。

proxy_no_cache $cookie_nocache $arg_nocache$arg_comment;

proxy_no_cache $http_pragma $http_authorization;

二、nginx cache的相关结构体

2.1 ngx_http_file_cache_s结构体

ngx_http_file_cache_sh_t和ngx_http_file_cache_s是用来管理cache内容的结构体，本身并不保存cache的内容。

```
typedef struct {
    ngx_rbtree_t                  rbtree;
    ngx_rbtree_node_t             sentinel;
    ngx_queue_t                   queue; /* 执行lru的inactive的队列 */
    ngx_atomic_t                  cold; /* 表示这个cache是否已经被loader进程load过 */
    ngx_atomic_t                  loading; /* loader 正在进程加载这个cache */
    off_t                         size; /* cache文件总大小 */
    ngx_uint_t                    count; /* cache文件个数 */
    ngx_uint_t                    watermark; /* cache最多允许文件个数,
                                                cache_manager 强制清除节点时使用 */
} ngx_http_file_cache_sh_t;

typedef struct ngx_http_file_cache_s  ngx_http_file_cache_t;
struct ngx_http_file_cache_s {
    ngx_http_file_cache_sh_t     *sh;
    ngx_slab_pool_t              *shpool;

    ngx_path_t                   *path; /* cache保存的目录 */
    ngx_path_t                   *temp_path; /* 零时cache保存的目录 */

    off_t                         max_size; /* cache文件尺寸的最大值 */
    size_t                        bsize; /* 文件系统的block大小 */

    time_t                        inactive; /* cache 文件的老化时间 */

    time_t                        fail_time; /* 上一次alloc cache node 失败的时间 */

    ngx_uint_t                    files; /* 当前cache文件个数 */
    ngx_uint_t                    loader_files; /* 阈值，当file超过files时,
                                                   loader进程将休眠 */
    ngx_msec_t                    last; /* 上一次被manage或者loader访问的时间 */
    ngx_msec_t                    loader_sleep; /* loader休眠时间 */
    ngx_msec_t                    loader_threshold; /* loader遍历的休眠间隔 */

    ngx_shm_zone_t               *shm_zone;
};
```

ngx_path_t 用来描述每个存储文件的目录信息

```
typedef struct {
    ngx_str_t                     name; /* cache的名字 */
    size_t                        len;
    size_t                        level[NGX_MAX_PATH_LEVEL]; /* 目录层级*/

    ngx_path_manager_pt           manager; /* 回调函数 */
    ngx_path_loader_pt            loader; /* 回调函数 */
    void                         *data; /* 回调数据 */

    u_char                       *conf_file;
    ngx_uint_t                    line;
} ngx_path_t;
```

当在nginx.conf使用xxx_cache_path这个命令时，将初始化上述结构体，具体在函数ngx_http_file_cache_set_slot 中实现对应成员的初始化。

cache->path->manager= ngx_http_file_cache_manager;

cache->path->loader= ngx_http_file_cache_loader;

```
cache->path->data= cache;

cache->path->conf_file= cf->conf_file->file.name.data;

cache->path->line= cf->conf_file->line;

cache->loader_files= loader_files;

cache->loader_sleep= loader_sleep;

cache->loader_threshold= loader_threshold;

cache->shm_zone->init= ngx_http_file_cache_init;

cache->shm_zone->data= cache;

cache->inactive= inactive;

cache->max_size= max_size;
```

其中ngx_http_file_cache_init实现ngx_http_file_cache_sh_t，也就是共享内存的初始化。

其中所有的path都由ngx_cycle_t->paths集中管理，对应的函数是ngx_add_path，临时目录：响应数据先写入临时文件，然后将其重命名为缓存文件，因此推荐xxx_temp_path和cache目录位于同一文件系统。

2.2 ngx_http_file_cache_node_t结构体

ngx_http_file_cache_node_t结构体用于将缓存文件的内容保存在共享内存中，以便多个worker进程使用。cache_node的管理算法为lru算法，即node查找和存储使用红黑树，超时管理使用队列，具体请参看http://flychao88.iteye.com/blog/1977653。

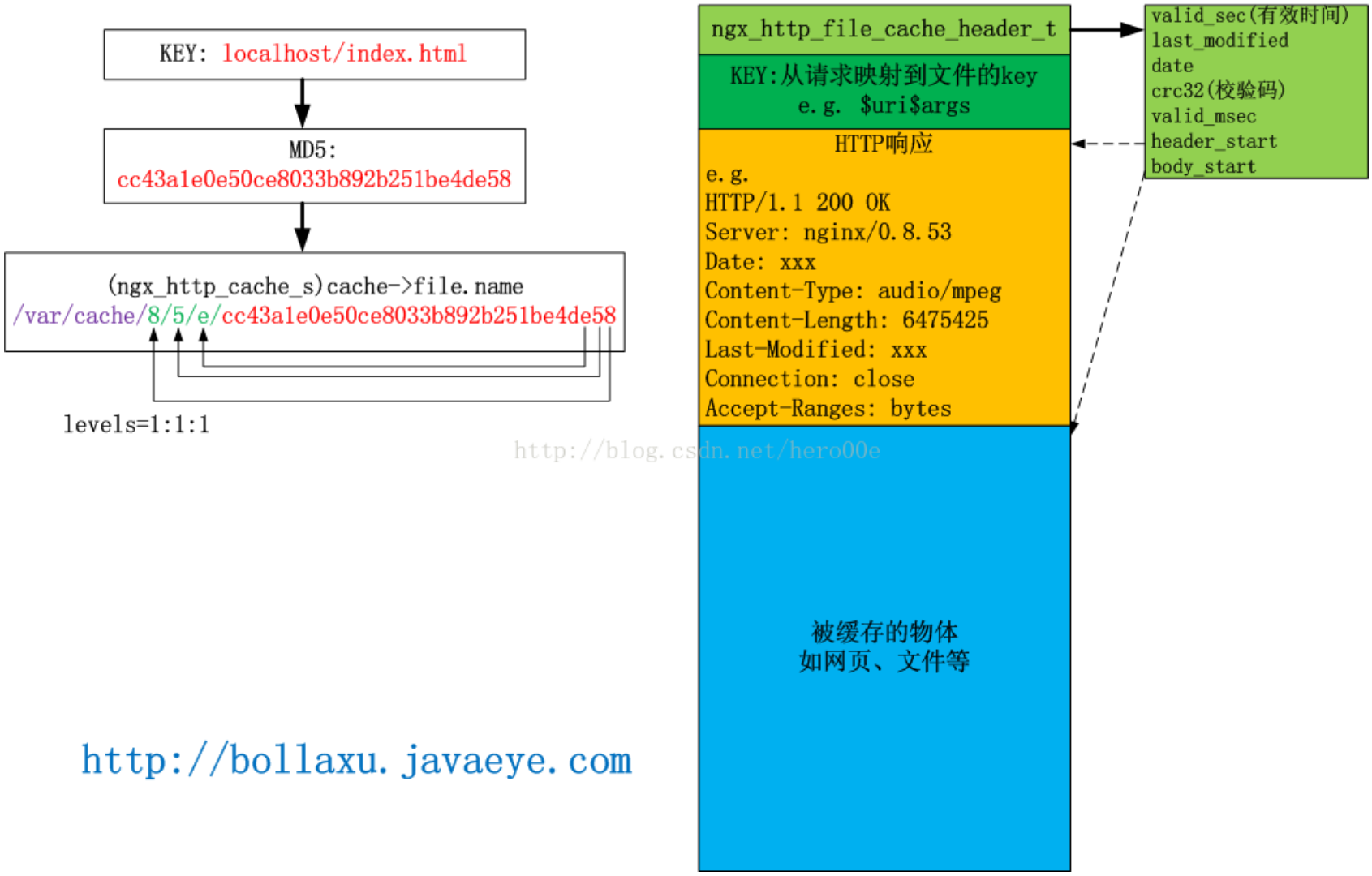
```
typedef struct {
    ngx_rbtree_node_t                   node;
    ngx_queue_t                         queue;

    u_char                              key[NGX_HTTP_CACHE_KEY_LEN
                                            - sizeof(ngx_rbtree_key_t)];

    unsigned                            count:20; /* 引用计数 */
    unsigned                            uses:10; /* 请求次数 */
    unsigned                            valid_msec:10; /* 响应码过期时间 */
    unsigned                            error:10; /* 响应码 */
    unsigned                            exists:1; /* cache文件是否存在 */
    unsigned                            updating:1; /* 正更新cache */
    unsigned                            deleting:1; /* 正删除cache */
                                        /* 11 unused bits */

    ngx_file_uniq_t                     uniq; /* 文件uniq */
    time_t                              expire; /* cache失效时间 */
    time_t                              valid_sec; /* 响应码过期时间 */
    size_t                              body_start; /* 被缓存文件的offset */
    off_t                               fs_size; /* 文件大小 */
    ngx_msec_t                          lock_time; /* cache lock的时间 */
} ngx_http_file_cache_node_t;
```


2.3 ngx_http_file_cache_header_t结构体

ngx_http_file_cache_header_t结构体为包头结构，用于表示文件系统中的缓存文件的存储格式，存储缓存文件的相关信息（修改时间、缓存 key 的 crc32 值、和用于指明HTTP 响应包头和包体在缓存文件中偏移位置的字段等）。

```
typedef struct {
    ngx_uint_t                          version;
    time_t                              valid_sec;
    time_t                              last_modified; /* 修改时间 */
    time_t                              date;
    uint32_t                            crc32; /* crc32值 */
    u_short                             valid_msec;
    u_short                             header_start; /* 包头offset */
    u_short                             body_start; /* 包体offset */
    u_char                              etag_len;
    u_char                              etag[NGX_HTTP_CACHE_ETAG_LEN];
    u_char                              vary_len;
    u_char                              vary[NGX_HTTP_CACHE_VARY_LEN];
    u_char                              variant[NGX_HTTP_CACHE_KEY_LEN];
} ngx_http_file_cache_header_t;
```


文件缓存格式信息



2.4 ngx_http_cache_t结构体

ngx_http_cache_t结构体用于表示cache的完整信息，它保存在ngx_http_request_s结构体中，即每一个http request当开启cache功能之后，将通过ngx_http_cache_t来保存对应的缓存条目信息。请求使用的缓存 file_cache 、缓存条目对应的缓存节点信息 node 、缓存文件 file 、key 值及其检验 crc32 等等，都临时保存于ngx_http_cache_t (ngx_http_request_t->cache)结构体中，这个结构体中的信息量基本上相当于 ngx_http_file_cache_header_t 和 ngx_http_file_cache_node_t 的总和。

```
typedef struct ngx_http_cache_s        ngx_http_cache_t;
struct ngx_http_cache_s {
    ngx_file_t                         file;
    ngx_array_t                        keys;
    uint32_t                           crc32;
    u_char                             key[NGX_HTTP_CACHE_KEY_LEN];
    u_char                             main[NGX_HTTP_CACHE_KEY_LEN];

    ngx_file_uniq_t                    uniq;
    time_t                             valid_sec;
    time_t                             last_modified;
    time_t                             date;

    ngx_str_t                          etag;
    ngx_str_t                          vary;
    u_char                             variant[NGX_HTTP_CACHE_KEY_LEN];

    size_t                             header_start;
    size_t                             body_start;
    off_t                              length;
    off_t                              fs_size;

    ngx_uint_t                         min_uses;
    ngx_uint_t                         error;
    ngx_uint_t                         valid_msec;

    ngx_buf_t                         *buf;

    ngx_http_file_cache_t             *file_cache;
    ngx_http_file_cache_node_t        *node;

#if (NGX_THREADS)
    ngx_thread_task_t                 *thread_task;
#endif

    ngx_msec_t                         lock_timeout;
    ngx_msec_t                         lock_age;
    ngx_msec_t                         lock_time;
    ngx_msec_t                         wait_time;

    ngx_event_t                        wait_event;

    unsigned                           lock:1;
    unsigned                           waiting:1;

    unsigned                           updated:1;
    unsigned                           updating:1;
    unsigned                           exists:1;
    unsigned                           temp_file:1;
    unsigned                           reading:1;
    unsigned                           secondary:1;
};
```


三、nginx cache的manger和loader进程工作原理

当配置解析完毕之后，就会进入进程初始化部分，在

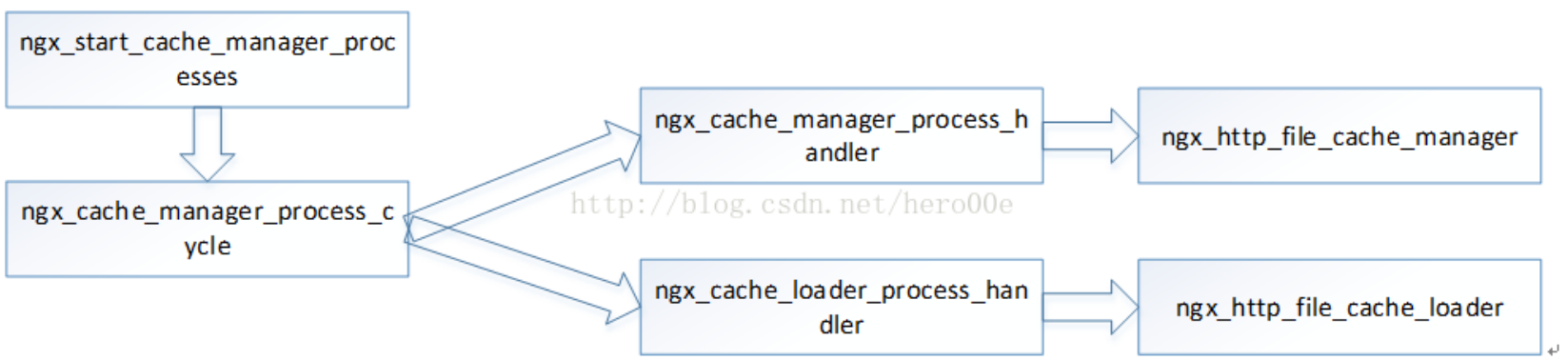
ngx_cache_manager_process_cycle函数中将启动cache manger和cache loader两个进程。

cache manger process的作用是用来定时删除无用的cache文件(引用计数为0)，一般来说只有manger会删除无用的cache(特殊情况，比如在loader中分配共享内存失败可能会强制删除一些cache， 或者说 loader的时候遇到一些特殊文件)。

cache loader process的主要作用是遍历cache目录，然后加载一些没有被加载的文件(比如nginx重启后，也就是上次遗留的文件)，或者说将cache文件重新插入(因为删除是使用LRU算法)。

3.1 cache manger和cache loader进程的启动

首先ngx_cache_manager_process_cycle 函数中调用ngx_start_cache_manager_processes函数，然后在ngx_start_cache_manager_processes中分别启动cache manger和cache loader进程。



// cache manager process，启动后立马执行

staticngx_cache_manager_ctx_t ngx_cache_manager_ctx = {

    ngx_cache_manager_process_handler,"cache manager process", 0

};

// cache loader process，启动后一分钟后执行

```
staticngx_cache_manager_ctx_t ngx_cache_loader_ctx = {
    ngx_cache_loader_process_handler,"cache loader process", 60000
};


static void
ngx_start_cache_manager_processes(ngx_cycle_t*cycle, ngx_uint_t respawn)
{
//启动cachemanager process
    ngx_spawn_process(cycle,ngx_cache_manager_process_cycle,
                      &ngx_cache_manager_ctx, "cache manager process",
                       respawn ?NGX_PROCESS_JUST_RESPAWN :
NGX_PROCESS_RESPAWN);


//启动cacheloader process
    ngx_spawn_process(cycle,ngx_cache_manager_process_cycle,
                      &ngx_cache_loader_ctx, "cache loader process",
                       respawn ?NGX_PROCESS_JUST_SPAWN :
NGX_PROCESS_NORESPAWN);


}
```

ngx_cache_manager_process_cycle主要就是设定定时器，并分别将ctx->handler传给ev。

```
static void

ngx_cache_manager_process_cycle(ngx_cycle_t*cycle, void *data)

{

    //将ctx->handler传给ev

ev.handler = ctx->handler;

    ev.data = ident;

ev.log = cycle->log;



//设定定时器

    ngx_add_timer(&ev, ctx->delay);

   ngx_process_events_and_timers(cycle);



}
```

这里的ctx->handler，分别是ngx_cache_manager_ctx的ngx_cache_manager_process_handler和ngx_cache_loader_ctx的ngx_cache_loader_process_handler。

```
static void

ngx_cache_manager_process_handler(ngx_event_t*ev)
```

```
{
//获取path结构体
    path = ngx_cycle->paths.elts;
    //遍历所有cache目录
for (i = 0; i < ngx_cycle->paths.nelts; i++) {


        if (path[i]->manager) {
            //调用path结构体中的manager函数
n =path[i]->manager(path[i]->data);
            //获取下一次定时器时间
next = (n <= next) ? n : next;
        }
    }
ngx_add_timer(ev, next * 1000);
}


static void
ngx_cache_loader_process_handler(ngx_event_t*ev)
{
    //获取path结构体
path = cycle->paths.elts;
```

```
    //遍历所有cache目录
for (i = 0; i < cycle->paths.nelts;i++) {
        if (path[i]->loader) {
                //调用path结构体中的loader函数
           path[i]->loader(path[i]->data);
        }
    }
}
```

以上path[i]->manager和path[i]->loader两个函数是在**ngx_http_file_cache_set_slot设置的，**

```
cache->path->manager= ngx_http_file_cache_manager;
cache->path->loader= ngx_http_file_cache_loader;
```

3.2 ngx_http_file_cache_manager的相关函数

```
static time_t
ngx_http_file_cache_manager(void*data)
{
//处理超时cache节点，将其删除
    next = ngx_http_file_cache_expire(cache);
```

```
    for ( ;; ) {
        //如果没有超过相关的最大限制，直接返回
        if (size < cache->max_size&& count < watermark) {
            return next;
        }
            //遍历所有cache节点，强制删除没有被引用的cache节点
        wait =ngx_http_file_cache_forced_expire(cache);
    }
}
```

ngx_http_file_cache_expire函数，这里nginx使用了LRU，也就是队列最尾端保存的是最长时间没有被使用的，并且这个函数返回的就是一个wait值。

```
static time_t
ngx_http_file_cache_expire(ngx_http_file_cache_t*cache)
{
    now = ngx_time();

   ngx_shmtx_lock(&cache->shpool->mutex);

    for ( ;; ) {
```

```
        //如果cache队列为空，则直接退出返回
if(ngx_queue_empty(&cache->sh->queue)) {
    wait = 10;
    break;
}
        //从最后一个开始
q =ngx_queue_last(&cache->sh->queue);

fcn = ngx_queue_data(q,ngx_http_file_cache_node_t, queue);

wait = fcn->expire - now;
        //如果没有超时，则退出
if (wait > 0) {
    wait = wait > 10 ? 10 : wait;
    break;
}
//如果引用计数为0，则删除这个cache节点
if (fcn->count == 0) {
    ngx_http_file_cache_delete(cache,q, name);
    continue;
}
```

```
        //如果当前节点正在删除，则退出循环
    if (fcn->deleting) {
        wait = 1;
        break;
    }
        //将当前节点放入队列最前端
    ngx_queue_remove(q);
    fcn->expire = ngx_time() +cache->inactive;
    ngx_queue_insert_head(&cache->sh->queue, &fcn->queue);
}
```

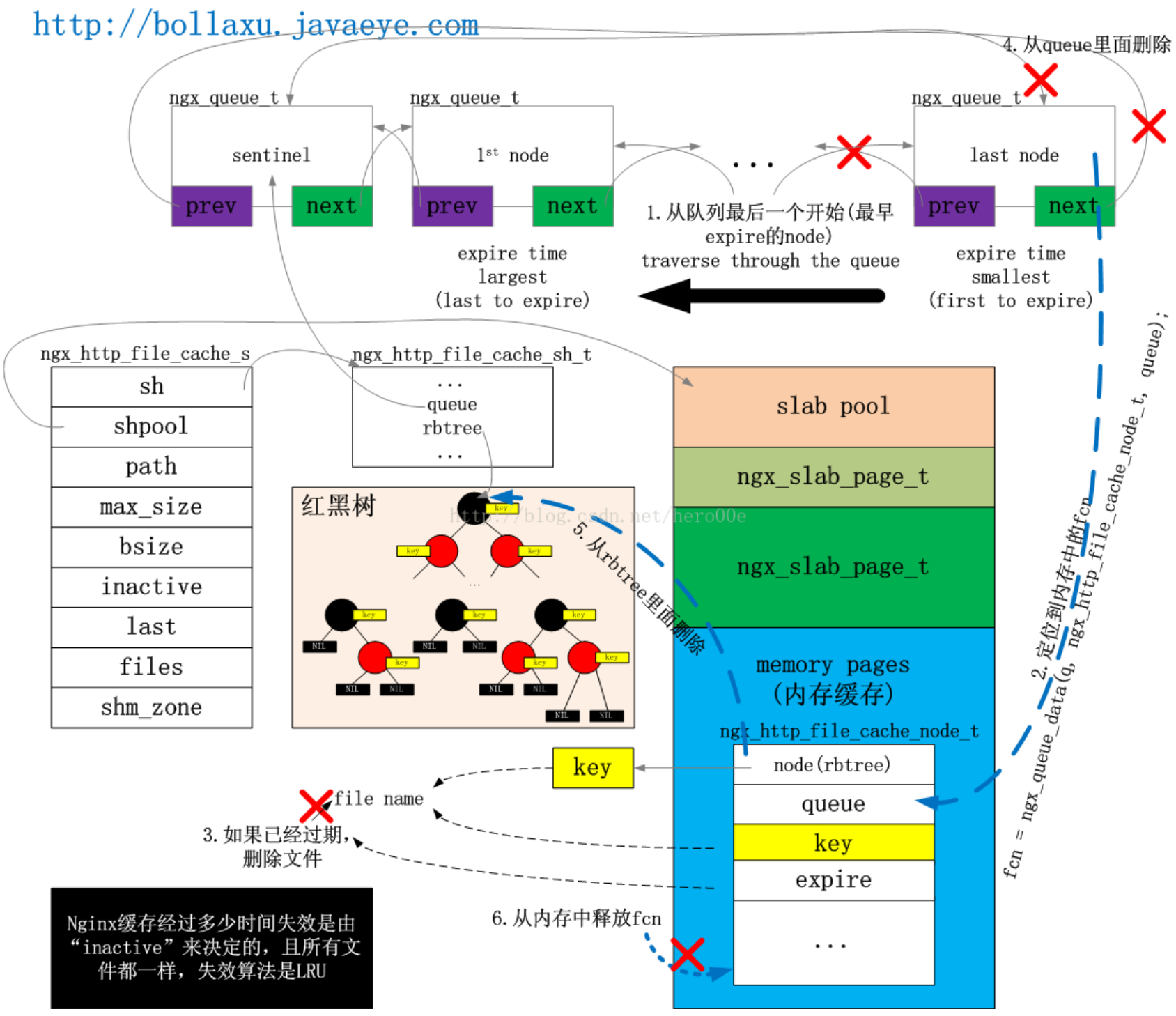
ngx_http_file_cache_forced_expire函数，就是强制删除cache 节点，它的返回值也是wait time，它的遍历也是从后到前的。

```
static time_t
ngx_http_file_cache_forced_expire(ngx_http_file_cache_t*cache)
{


    wait = 10;
    //删除节点尝试次数
tries = 20;
```

```
//遍历队列
for (q = ngx_queue_last(&cache->sh->queue);
     q !=ngx_queue_sentinel(&cache->sh->queue);
     q = ngx_queue_prev(q))
{
    fcn = ngx_queue_data(q,ngx_http_file_cache_node_t, queue);
          //如果引用计数为0则删除cache
    if (fcn->count == 0) {
        ngx_http_file_cache_delete(cache,q, name);
        wait = 0;
    } else {
                //否则尝试20次
        if (--tries) {
            continue;
        }

        wait = 1;
    }
    break;
}
```

return wait;

}



3.3 ngx_http_file_cache_loader的相关函数

static void

ngx_http_file_cache_loader(void*data)

{

ngx_tree_ctx_t tree;

```
/*
设置回调函数，其中
init_handler：初始化遍历过程中的相关数据结构，与alloc配合使用，只要alloc被赋值了，那么init_handler也必须进行赋值，即一句话，要么同时存在，要么同时为空；
file_handler：处理普通文件的回调函数；
pre_tree_handler：进入一个目录前的回调函数；
post_tree_handler：离开一个目录后的回调函数；
spec_handler：处理特殊文件的回调函数，比如socket，FIFO等；
*/
tree.init_handler = NULL;
//对于每个cache文件调用这个回调
tree.file_handler = ngx_http_file_cache_manage_file;
//主要用来遍历所有cache的目录
    tree.pre_tree_handler =ngx_http_file_cache_manage_directory;
    tree.post_tree_handler =ngx_http_file_cache_noop;
    tree.spec_handler =ngx_http_file_cache_delete_file;
    //回调数据为cache
tree.data = cache;
    tree.alloc = 0;
    tree.log = ngx_cycle->log;
```

//最后load的时间

cache->last = ngx_current_msec;

cache->files = 0;

//目录树遍历

if (ngx_walk_tree(&tree,&cache->path->name) == NGX_ABORT) {

cache->sh->loading = 0;

return;

}

cache->sh->cold = 0;

cache->sh->loading = 0;

}

ngx_walk_tree函数，nginx目录树的遍历

ngx_int_t

ngx_walk_tree(ngx_tree_ctx_t*ctx, ngx_str_t *tree)

{

//打开需要遍历的目录

if (ngx_open_dir(tree, &dir) ==NGX_ERROR) {

ngx_log_error(NGX_LOG_CRIT,ctx->log, ngx_errno,

ngx_open_dir_n "\"%s\" failed", tree->data);

```
    return NGX_ERROR;
}


    //开始遍历
for ( ;; ) {


    //读取当前目录下的目录项
    if (ngx_read_dir(&dir) ==NGX_ERROR) {
        err = ngx_errno;
        goto done;
    }


        //获取目录项的名称和长度
len = ngx_de_namelen(&dir);
name = ngx_de_name(&dir);


        //构造新目录项的完整路径
file.len = tree->len + 1 + len;
p = ngx_cpymem(buf.data, tree->data,tree->len);
*p++ = '/';
ngx_memcpy(p, name, len + 1);
```

/*如果新的目录项是普通文件，调用file_handler进行处理，如果是目录，递归调用ngx_walk_tree，如果是特殊文件，调用spec_handler进行处理*/

```
        if (ngx_de_is_file(&dir)) {

            if (ctx->file_handler(ctx,&file) == NGX_ABORT) {

                goto failed;

            }



        } else if (ngx_de_is_dir(&dir)) {



            rc = ctx->pre_tree_handler(ctx,&file);

            if (ngx_walk_tree(ctx, &file)== NGX_ABORT) {

                goto failed;

            }

            if (ctx->post_tree_handler(ctx,&file) == NGX_ABORT) {

                goto failed;

            }



    } else {

        if (ctx->spec_handler(ctx, &file)== NGX_ABORT) {
```

```
                goto failed;

            }

        }

    }


failed:


    rc = NGX_ABORT;


done:

        //关闭文件目录
    if (ngx_close_dir(&dir) == NGX_ERROR) {

        ngx_log_error(NGX_LOG_CRIT,ctx->log, ngx_errno,

                      ngx_close_dir_n "\"%s\" failed", tree->data);

    }


    return rc;

}
```

ngx_http_file_cache_manage_file函数，用来加载cache文件的句柄到内存中

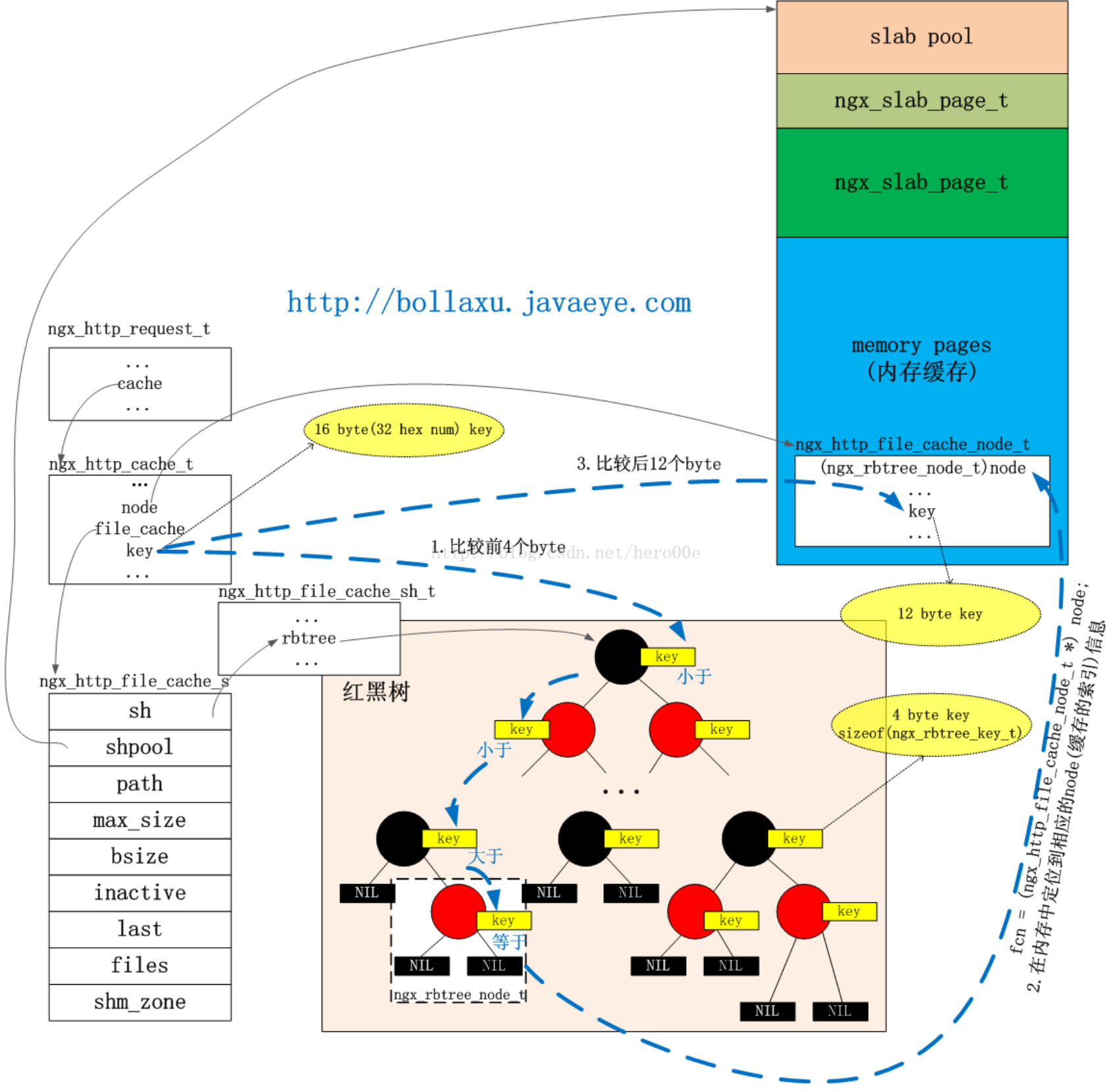
```
static ngx_int_t
```

```
ngx_http_file_cache_manage_file(ngx_tree_ctx_t*ctx, ngx_str_t *path)

{

cache = ctx->data;

        //将文件添加进cache管理结构中

    if (ngx_http_file_cache_add_file(ctx, path)!= NGX_OK) {

        (void)ngx_http_file_cache_delete_file(ctx, path);

    }

        //如果文件个数太大，则休眠并清理

    if (++cache->files >=cache->loader_files) {

      ngx_http_file_cache_loader_sleep(cache);

    } else {

        ngx_time_update();

            //否则看loader时间是不是过长，如果过长则又进入休眠

        if (elapsed >=cache->loader_threshold) {

          ngx_http_file_cache_loader_sleep(cache);

        }

    }


    return (ngx_quit || ngx_terminate) ?NGX_ABORT : NGX_OK;

}
```

ngx_http_file_cache_add_file函数，它主要是通过文件名计算hash，然后调用ngx_http_file_cache_add将这个文件加入到cache管理中。

```
static ngx_int_t

ngx_http_file_cache_add(ngx_http_file_cache_t*cache, ngx_http_cache_t *c)

{

    //查找cache node

   fcn = ngx_http_file_cache_lookup(cache,c->key);



   if (fcn == NULL) {

        //如果不在，则重新构建

    fcn =ngx_slab_calloc_locked(cache->shpool,

                        sizeof(ngx_http_file_cache_node_t));

    }



    ngx_memcpy((u_char *)&fcn->node.key, c->key, sizeof(ngx_rbtree_key_t));



    ngx_memcpy(fcn->key,&c->key[sizeof(ngx_rbtree_key_t)],

            NGX_HTTP_CACHE_KEY_LEN -sizeof(ngx_rbtree_key_t));

    //插入红黑树

ngx_rbtree_insert(&cache->sh->rbtree, &fcn->node);
```

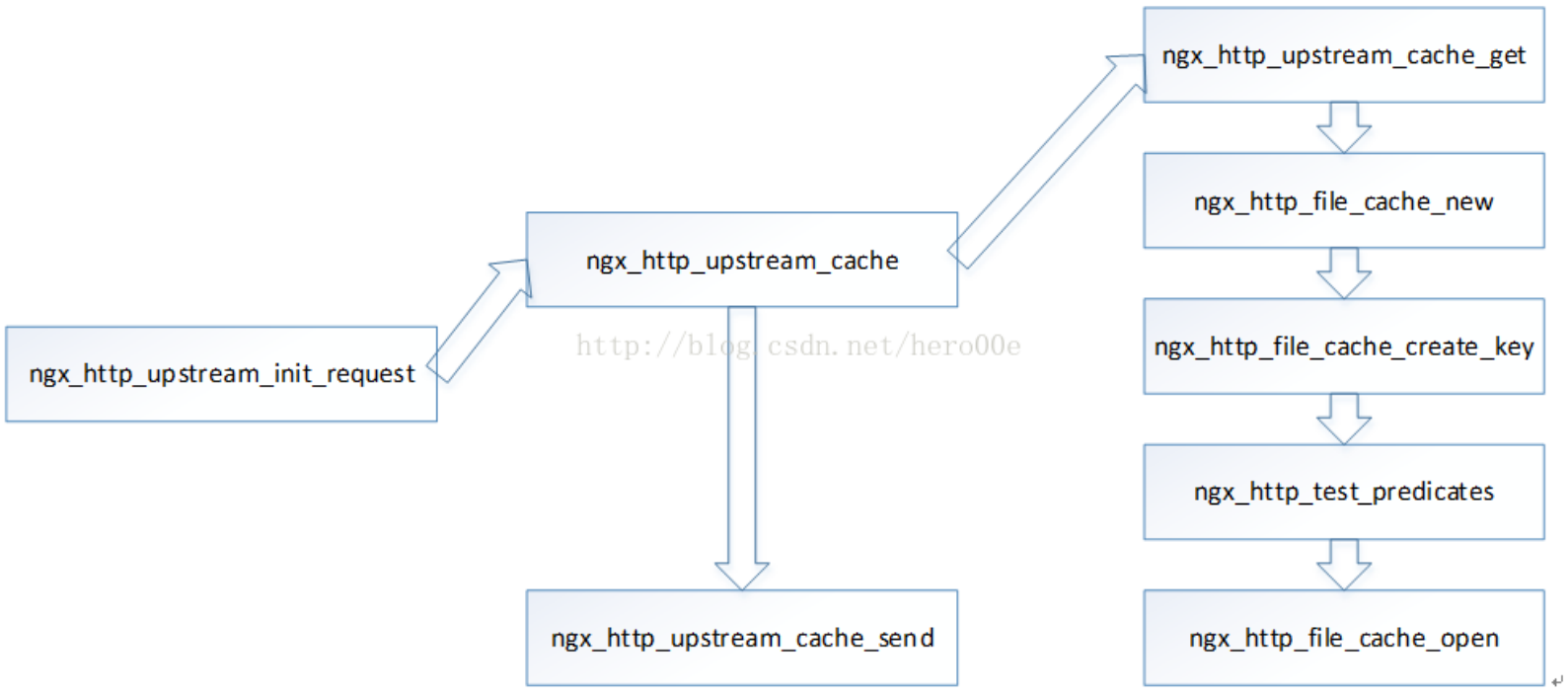
```
        fcn->uses = 1;

        fcn->exists = 1;

        fcn->fs_size = c->fs_size;



        cache->sh->size += c->fs_size;


} else {
        //否则删除queue，后续会重新插入
        ngx_queue_remove(&fcn->queue);
    }

     fcn->expire = ngx_time() +cache->inactive;

        //重新插入到lru队列的头

    ngx_queue_insert_head(&cache->sh->queue, &fcn->queue);

}
```



四、 nginx cache的应用过程

在http的请求发送给有proxy_cache和proxy_pass定义的location的时候，会调用到ngx_http_proxy_handler，nginx向上游服务器发送upstream请求时，会先检查本地是否有可用缓存。具体的位置是在ngx_http_upstream_init_request中，在向后端发起请求之前。

4.1 ngx_http_upstream_init_request函数




```
static void

ngx_http_upstream_init_request(ngx_http_request_t*r)

{

#if(NGX_HTTP_CACHE)


//检查是否有配置proxy_cache，如果有则使用缓存

    if (u->conf->cache) {

        ngx_int_t  rc;


//寻找缓存，如果返回NGX_DECLINED就是说缓存中没有，向后端发送请求

        rc = ngx_http_upstream_cache(r,u);


        if (rc == NGX_BUSY) {
```

```
        //重试一次
r->write_event_handler =ngx_http_upstream_init_request;
        return;
    }


    r->write_event_handler =ngx_http_request_empty_handler;


    if (rc == NGX_OK) {
                //缓存找到了，分析缓存文件的结构体，读取缓存的headers+body，并发送给前端
        rc = ngx_http_upstream_cache_send(r,u);
    if (rc == NGX_DONE) {
        return;
    }


    if (rc ==NGX_HTTP_UPSTREAM_INVALID_HEADER) {
        rc = NGX_DECLINED;
        r->cached = 0;
    }
}
```

```
        if (rc != NGX_DECLINED) {

            ngx_http_finalize_request(r, rc);

            return;

        }

    }


#endif


//向后端发起请求

}
```

4.2 ngx_http_upstream_cache函数

所有请求缓存使用的尝试，都是通过 ngx_http_upstream_cache 函数开始的。该函数主要完成以下几个功能。

（1）如果还未给当前请求分配缓存相关结构体（ngx_http_cache_t）时，创建此类型字段（r->cache）并初始化。

```
    //获得缓存的管理结构体（ngx_http_file_cache_t）

    ngx_http_upstream_cache_get(r, u,&cache)
```

//创建一个新的ngx_http_cache_t

ngx_http_file_cache_new(r)

//ngx_http_xxx_create_key 将 `xxx_cache_key`配置指令定义的缓存 key 根据请求信息进行取值

u->create_key(r)

//生成 md5sum(key) 和 crc32(key)并计算 `c->header_start` 值

ngx_http_file_cache_create_key(r)

//默认所有请求的响应结果都是可被缓存的

u->cacheable = 1;

c = r->cache;

c->body_start = u->conf->buffer_size;

c->min_uses = u->conf->cache_min_uses;

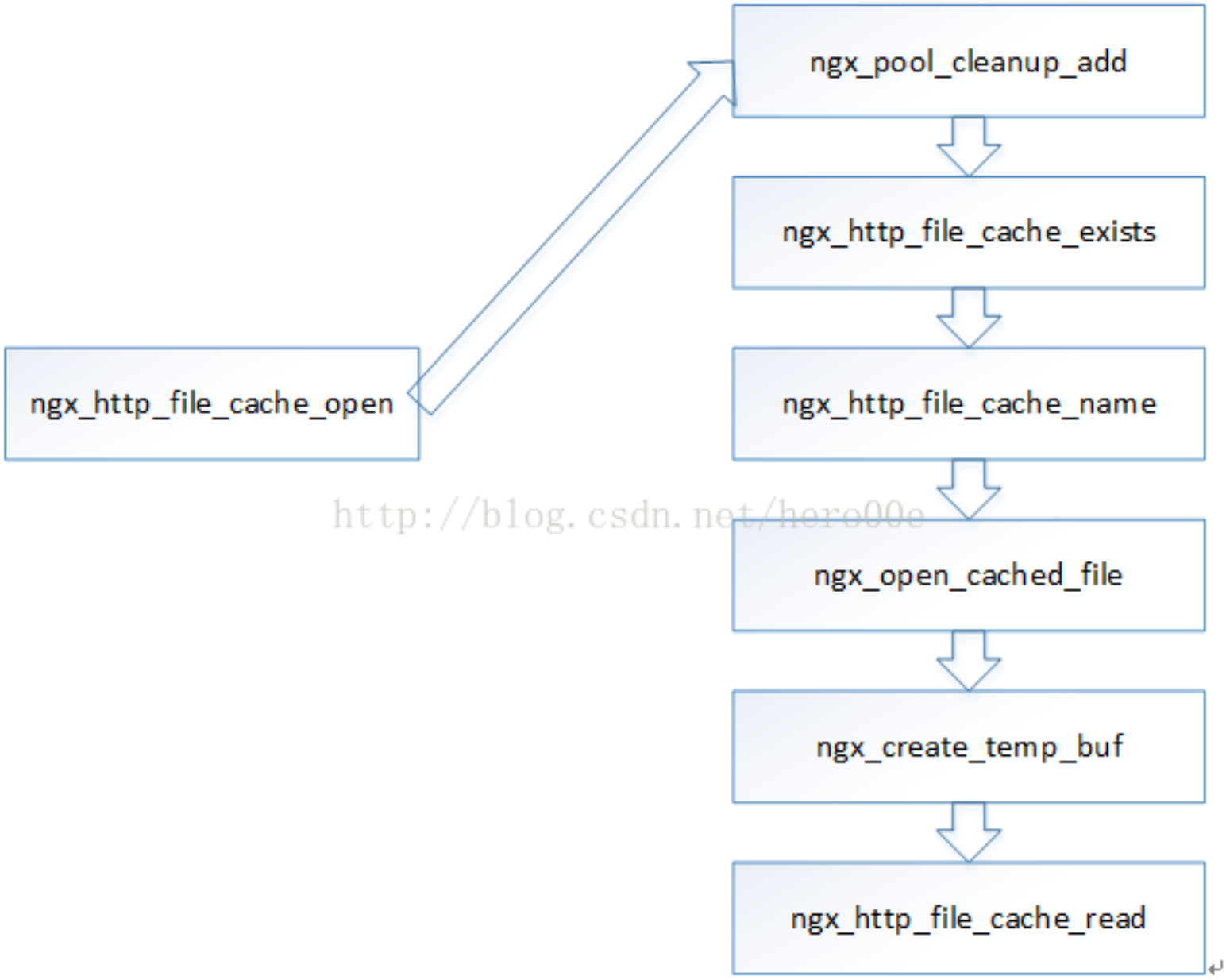
c->file_cache = cache;

/*根据配置文件中（xxx_cache_bypass）缓存绕过条件和请求信息，判断是否应该

继续尝试使用缓存数据响应该请求*/

ngx_http_test_predicates(r,u->conf->cache_bypass)

（2）调用ngx_http_file_cache_open 函数查找是否有对应的有效缓存数据，函数负责缓存文件定位、缓存文件打开和校验等操作。




```
ngx_int_t

ngx_http_file_cache_open(ngx_http_request_t*r)

{

    cache = c->file_cache;
```

//第一次根据请求信息生成的 key 查找对应缓存节点时，先注册一下请求内存池级别的清理函数

```
    if (c->node == NULL) {

        cln = ngx_pool_cleanup_add(r->pool,0);



        cln->handler =ngx_http_file_cache_cleanup;
```

```
    cln->data = c;

}

   //尝试获取缓存文件的节点信息

rc = ngx_http_file_cache_exists(cache, c);



if (rc == NGX_ERROR) {

    return rc;

}



if (rc == NGX_AGAIN) {

    return NGX_HTTP_CACHE_SCARCE;

    }



    if (rc == NGX_OK) {

        rv = NGX_DECLINED;

} else { /* rc == NGX_DECLINED */

        //根据设置的最小min_uses决定是否对cache进行缓存

         if (c->min_uses > 1) {

            rv = NGX_HTTP_CACHE_SCARCE;

        } else {

            rv = NGX_DECLINED;
```

```
    }
}
    //缓存文件完整文件名：r->cache->file.name.data,可以通过此文件名打开缓存
if (ngx_http_file_cache_name(r,cache->path) != NGX_OK) {
    return NGX_ERROR;
}

    //尝试打开并获取文件缓存信息
if(ngx_open_cached_file(clcf->open_file_cache, &c->file.name, &of,r->pool)
    != NGX_OK)
    //创建用于存储缓存文件头的临时缓冲区
    c->buf = ngx_create_temp_buf(r->pool,c->body_start);
    //读取缓存文件头并进行有效性验证
    return ngx_http_file_cache_read(r, c);

done:

    return rv;
}
```

ngx_http_file_cache_open函数负责缓存文件定位、缓存文件打开和校验等操作，其返回值及对应含义，以及其调用者 ngx_http_upstream_cache 对应的行为总结如下：

NGX_OK

- 缓存正常命中

- 设置 `cache_status` 为 `NGX_HTTP_CACHE_HIT`，然后向客户端发送缓存内容

NGX_HTTP_CACHE_STALE

- 缓存内容过期，当前请求需要向后端请求新的响应数据。

- 设置 `cache_status` 为 `NGX_HTTP_CACHE_EXPIRED`，并返回 `NGX_DECLINED`

以继续请求处理 (`r->cached = 0; c->valid_sec = 0`)。

NGX_HTTP_CACHE_UPDATING

- 缓存内容过期，同时已有同样使用该缓存节点的其它请求正在请求新的响应数据。

- 如果xxx_cache_use_stale 启用了 "updating"，设置 `cache_status` 为

`NGX_HTTP_CACHE_UPDATING`，然后向客户端发送过期缓存内容。否则，将返回

值重设为 `NGX_HTTP_CACHE_STALE`。

NGX_HTTP_CACHE_SCARCE

- 因缓存节点被查询次数还未达 `min_uses`，对此请求禁用缓存机制

- 继续请求处理，但是不再缓存其响应数据 (`u->cacheable = 0`)。

NGX_DECLINED

- 缓存内容因为不存在 (`c->exists == 0`)、缓存内容未通过校验、或者当前请

求正在更新等原因，暂时无法使用缓存。

- 继续请求处理，并尝试对其响应数据进行缓存。

NGX_AGAIN

- 缓存内容过期，并且当前缓存节点正在被其它请求更新，或者 还未能从缓存文

件中读到足够的数据 (aio 模块下)。

- 返回 `NGX_BUSY`，Nginx 会再次尝试读取缓存。

NGX_ERROR

- 内存相关、文件系统等系统错误。

- 返回 `NGX_ERROR`，Nginx 会调用 `ngx_http_finalize_request` 终止此请求。

NGX_HTTP_SPECIAL_RESPONSE

- 打开 `xxx_intercept_errors` 配置情况下，直接返回缓存的错误码。

- 设置 `cache_status` 为 `NGX_HTTP_CACHE_HIT` 并返回错误码。

（3）ngx_http_file_cache_exists函数

```
static ngx_int_t

ngx_http_file_cache_exists(ngx_http_file_cache_t*cache, ngx_http_cache_t *c)

{

    fcn = c->node;


    if (fcn == NULL) {

        //以 c->key 为查找条件从缓存中查找缓存节点

    fcn =ngx_http_file_cache_lookup(cache, c->key);

    }


//如果找到了对应 c->key 的缓存节点

if (fcn) {

    //如果该请求第一次使用此缓存节点，则增加相关引用和使用次数，继
```

续下面条件判断

```
    if (c->node == NULL) {

        fcn->uses++;

        fcn->count++;

    }
```

//如果 xxx_cache_valid 配置指令对此节点过期时间做了特殊设定

```
    if (fcn->error) {
```

//检查节点是否过期

```
        if (fcn->valid_sec <ngx_time()) {
```

//如果过期，重置节点，并返回 NGX_DECLINED

```
            goto renew;

        }
```

//如果未过期，返回NGX_OK

```
        rc = NGX_OK;

        goto done;

    }
```

//如果缓存文件存在或者缓存节点被使用次数超过 xxx_cache_min_uses配置值

```
    if (fcn->exists || fcn->uses>= c->min_uses) {
```

```
        c->exists = fcn->exists;
        if (fcn->body_start) {
            c->body_start =fcn->body_start;
        }
        rc = NGX_OK;
        goto done;
    }
    //条件2, 3 都不满足时，此次查找失败，返回NGX_AGAIN
    rc = NGX_AGAIN;
    goto done;
}
    //未找到，创建新的cache node节点，并初始化
if (fcn == NULL) {
    fcn =ngx_slab_calloc_locked(cache->shpool,
                                sizeof(ngx_http_file_cache_node_t));
}

cache->sh->count++;

ngx_memcpy((u_char *) &fcn->node.key,c->key, sizeof(ngx_rbtree_key_t));
```

```
    ngx_memcpy(fcn->key,&c->key[sizeof(ngx_rbtree_key_t)],
                NGX_HTTP_CACHE_KEY_LEN -sizeof(ngx_rbtree_key_t));


   ngx_rbtree_insert(&cache->sh->rbtree, &fcn->node);


    fcn->uses = 1;

    fcn->count = 1;


renew:

//更新节点信息

    rc = NGX_DECLINED;

    fcn->valid_msec = 0;

    fcn->error = 0;

    fcn->exists = 0;

    fcn->valid_sec = 0;

    fcn->uniq = 0;

    fcn->body_start = 0;

    fcn->fs_size = 0;

done:

    fcn->expire = ngx_time() +cache->inactive;

   ngx_queue_insert_head(&cache->sh->queue, &fcn->queue);
```

```
    c->uniq = fcn->uniq;

    c->error = fcn->error;

    c->node = fcn;



failed:

    return rc;

}
```

（4）ngx_open_cached_file函数，用于打开cache的file文件

```
ngx_int_t

ngx_open_cached_file(ngx_open_file_cache_t*cache, ngx_str_t *name,

    ngx_open_file_info_t *of, ngx_pool_t *pool)

{

  //计算cache文件名的 hash值

    hash = ngx_crc32_long(name->data,name->len);

      //根据名字查找对应的file对象

    file = ngx_open_file_lookup(cache, name,hash);



    if (file) {

//如果找到文件
```

```
        file->uses++;


        ngx_queue_remove(&file->queue);


        of->fd = file->fd;

        of->uniq = file->uniq;

            //打开文件，保存文件信息，根据uniq是否相等，判断文件是否有改变

        rc = ngx_open_and_stat_file(name, of,pool->log);

            //根据文件状态做出相应操作

        if (of->is_dir) {
    if (file->is_dir || file->err) {
            goto update;
        }
        /* file became directory */
    } else if (of->err == 0) {  /* file */
        if (file->is_dir ||file->err) {
            goto add_event;
        }
        if (of->uniq == file->uniq) {
            of->is_directio =file->is_directio;
```

```
            goto update;

        }

        /* file was changed */

    } else { /* error to cache */

        if (file->err || file->is_dir){

            goto update;

        }

        /* file was removed, etc. */

    }

ngx_rbtree_delete(&cache->rbtree, &file->node);

 cache->current--;

        file->close = 1;

        goto create;

    }


    /* not found */

        //打开文件,保存文件信息

        rc = ngx_open_and_stat_file(name, of,pool->log);


create:

//max为open_file_cache命令中定义的那个max指令，而current也就是当前
```

cache的文件个数

if (cache->current >= cache->max){

//如果大于max，则需要强制expire几个元素

ngx_expire_old_cached_files(cache,0, pool->log);

}

ngx_cpystrn(file->name, name->data,name->len + 1);

file->node.key = hash;

//插入到file的红黑树

ngx_rbtree_insert(&cache->rbtree,&file->node);

//更新current

cache->current++;

update:

//更新创建时间

file->created = now;

found:

//更新存取时间

```
 file->accessed = now;

    //将文件插入超时队列

ngx_queue_insert_head(&cache->expire_queue, &file->queue);



 if (of->err == 0) {



    if (!of->is_dir) {

                //设置清理函数

        cln->handler =ngx_open_file_cleanup;

        ofcln = cln->data;

        ofcln->cache = cache;

        ofcln->file = file;

        ofcln->min_uses =of->min_uses;

        ofcln->log = pool->log;

    }



    return NGX_OK;

}

return NGX_ERROR;

}
```

4.3 ngx_http_upstream_send_response 函数

在Nginx收到后端服务器的响应之后，会把这个响应发回给用户。而如果缓存功能启用的话，Nginx就会把响应存入磁盘里。

```
static void

ngx_http_upstream_send_response(ngx_http_request_t*r, ngx_http_upstream_t
*u)

{

#if(NGX_HTTP_CACHE)



// 是否要缓存，即proxy_no_cache指令

    switch (ngx_http_test_predicates(r,u->conf->no_cache)) {
default: /* NGX_OK */
    if (u->cache_status ==NGX_HTTP_CACHE_BYPASS) {
        /* create cache if previouslybypassed */
        if (ngx_http_file_cache_create(r)!= NGX_OK) {
           ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
            return;
        }
    }
    break;
}
```

```
    //如果需要缓存
if (u->cacheable) {
        //设置有效时间，
    if (valid) {

        r->cache->date = now;
        r->cache->body_start =(u_short) (u->buffer.pos - u->buffer.start);

        if (u->headers_in.status_n ==NGX_HTTP_OK
            || u->headers_in.status_n ==NGX_HTTP_PARTIAL_CONTENT)
        {
            r->cache->last_modified =u->headers_in.last_modified_time;

        } else {
            r->cache->last_modified =-1;
           ngx_str_null(&r->cache->etag);
        }

    if(ngx_http_file_cache_set_header(r, u->buffer.start) != NGX_OK) {
        ngx_http_upstream_finalize_request(r, u, NGX_ERROR);
```

```
            return;

        }



    } else {

        u->cacheable = 0;

    }

}



if (u->cacheable == 0 && r->cache) {

    //如果不需要保存cache，则删除cache

    ngx_http_file_cache_free(r->cache,u->pipe->temp_file);

}

#endif



    //在这儿把从后端获取的文件写到磁盘

    ngx_http_upstream_process_upstream(r,u)

}
```

在ngx_http_upstream_process_upstream中写入调用函数链为：1. ngx_event_pipe-> ngx_event_pipe_read_upstream->

ngx_event_pipe_write_chain_to_temp_file-> ngx_write_chain_to_temp_file -

>ngx_write_chain_to_file。

2. ngx_http_upstream_process_request->ngx_http_file_cache_update, 将零时文件的path进行修改，改为proxy_cache_path所要求的路径名。

4.4 ngx_open_file_cleanup 函数

ngx_open_file_cleanup函数是file的node节点的超时处理函数，每个request结束的时候，都会清理掉他所分配的pool，而nginx就是给每个有打开文件的request都绑定了对应clean handler，当request pool被释放的时候，就会来根据时间来判断是否已经超时，这里的clean handler就是ngx_open_file_cleanup。

```
static void
ngx_open_file_cleanup(void*data)
{
    c->file->count--;
      //关闭文件
    ngx_close_cached_file(c->cache,c->file, c->min_uses, c->log);
      //处理超时
    /* drop one or two expired open files */
    ngx_expire_old_cached_files(c->cache, 1,c->log);
}
```

ngx_close_cached_file函数用来尝试关闭当前的文件

```
static void

ngx_close_cached_file(ngx_open_file_cache_t*cache,

    ngx_cached_open_file_t *file, ngx_uint_tmin_uses, ngx_log_t *log)

{

//判断是否需要关闭文件

    if (!file->close) {

            //如果不需要，则更新存取时间

        file->accessed = ngx_time();



        ngx_queue_remove(&file->queue);

    ngx_queue_insert_head(&cache->expire_queue, &file->queue);

            //如果文件使用次数大于最小缓存次数，或者文件被引用，则返回

     if (file->uses >= min_uses ||file->count) {

        return;

     }

}



ngx_open_file_del_event(file);

if (file->count) {

    return;
```

```
}
   //文件需要被关闭
if (file->fd != NGX_INVALID_FILE) {
        //关闭文件
    if (ngx_close_file(file->fd) ==NGX_FILE_ERROR) {
    }

    file->fd = NGX_INVALID_FILE;
}
ngx_free(file->name);
ngx_free(file);
}
```

ngx_expire_old_cached_files函数用来执行超时，或者强制超时cache中的元素(根据参数 不同)。主要的判断就是now – file->accessed <= cache->inactive，这段主要是看这个文件多久没有被使用，是否超过了我们设置的inactive时间。

```
static void
ngx_expire_old_cached_files (ngx_open_file_cache_t *cache, ngx_uint_tn,
    ngx_log_t *log)
{
```

```
while (n < 3) {
```

//如果队列为空，则直接返回

```
if(ngx_queue_empty(&cache->expire_queue)) {
    return;
}
```

//取出最后一个文件，也就是可能需要被超时的文件

```
q= ngx_queue_last(&cache->expire_queue);
file = ngx_queue_data(q,ngx_cached_open_file_t, queue);
```

//n是控制是强制超时，还是按inactive超时，后一个判断是判断是否超时

```
if (n++ != 0 && now -file->accessed <= cache->inactive) {
    return;
}
```

//如果有超时的，或者需要强制超时，则开始从队列和红黑树中移除

```
ngx_queue_remove(q);
ngx_rbtree_delete(&cache->rbtree, &file->node);
cache->current--;

if (!file->err &&!file->is_dir) {
```

```
                //关闭文件

            file->close = 1;

            ngx_close_cached_file(cache, file,0, log);



        } else {

            ngx_free(file->name);

            ngx_free(file);

        }

    }

}
```

参考文献：

http://www.iigrowing.cn/nginx-cache-ji-ben-zhi-ling-zheng-li.html

http://blog.csdn.net/weiyuefei/article/details/35783295

http://www.tuicool.com/articles/QnMNr23

http://blog.itpub.net/15480802/viewspace-1421409/

http://blog.csdn.net/weiyuefei/article/details/34823885

http://blog.csdn.net/coloriy/article/details/47039603

http://blog.sina.com.cn/s/blog_70898f3f0100s81o.html

http://blog.csdn.net/weiyuefei/article/details/38313663

http://blog.csdn.net/weiyuefei/article/details/35782523

http://blog.csdn.net/weiyuefei/article/details/35781277