nginx源码剖析

作者: simohayha http://simohayha.javaeye.com

nginx源码剖析

目 录

1. 服务器设计

1.1 nginx的进程模型

发表时间: 2009-09-13

nginx采用的也是大部分http服务器的做法，就是master,worker模型，一个master进程管理站个或者多个worker进程，基本的事件处理都是放在woker中，master负责一些全局初始化，以及对worker的管理。

在nginx中master和worker的通信是通过socketpair来实现的，每次fork完一个子进程之后，将这个子进程的socketpaire句柄传递给前面已经存在的子进程，这样子进程之间也就可以通信了。

nginx中fork子进程是在ngx_spawn_process中进行的：

第一个参数是全局的配置，第二个参数是子进程需要执行的函数，第三个参数是proc的参数。第四个类型。

```
ngx_pid_t
ngx_spawn_process(ngx_cycle_t *cycle, ngx_spawn_proc_pt proc, void *data,
    char *name, ngx_int_t respawn)
```

这个函数主要的任务就是：

1 有一个ngx_processes全局数组，包含了所有的存货的子进程，这里会fork出来的子进程放入到相应的位置。并设置这个进程的相关属性。

2 创建socketpair，并设置相关属性。

3 在子进程中执行传递进来的函数。

在看详细代码之前，我们先来看几个主要的数据结构：

首先是进程结构，这个结构体表示了一个进程。包含了它的id状态，channel等等。

```
typedef struct {
///进程id
    ngx_pid_t           pid;
```

```
///进程的退出状态(主要在waitpid中进行处理).
    int                 status;
///进程channel(也就是通过socketpair创建的两个句柄)
    ngx_socket_t        channel[2];

///进程的执行函数（也就是每次spawn，子进程所要执行的那个函数).
    ngx_spawn_proc_pt   proc;
    void               *data;
    char               *name;
///进程的几个状态。
    unsigned            respawn:1;
    unsigned            just_respawn:1;
    unsigned            detached:1;
    unsigned            exiting:1;
    unsigned            exited:1;
} ngx_process_t;
```

下面我们来看详细的代码。
先来看第一部分：

```
//全局的进程表，保存了存活的子进程。
ngx_process_t    ngx_processes[NGX_MAX_PROCESSES];
.................................
u_long     on;
    ngx_pid_t  pid;
///表示将要fork的子进程在ngx_processes中的位置，
    ngx_int_t  s;
///首先，如果传递进来的类型大于0,则就是已经确定这个进程已经退出，我们就可以直接确定slot。
    if (respawn >= 0) {
        s = respawn;

    } else {
///遍历ngx_processess，从而找到空闲的slot，从而等会fork完毕后,将子进程信息放入全局进程信息表的相应的s
```

```
        for (s = 0; s < ngx_last_process; s++) {
            if (ngx_processes[s].pid == -1) {
                break;
            }
        }
///到达最大进程限制报错。
        if (s == NGX_MAX_PROCESSES) {
            ngx_log_error(NGX_LOG_ALERT, cycle->log, 0,
                          "no more than %d processes can be spawned",
                          NGX_MAX_PROCESSES);
            return NGX_INVALID_PID;
        }
    }
```

接下来新建一对socketpair句柄，然后初始化相关属性。

```
///如果类型为NGX_PROCESS_DETACHED，则说明是热代码替换(热代码替换也是通过这个函数进行处理的)，因此不需
if (respawn != NGX_PROCESS_DETACHED) {

        /* Solaris 9 still has no AF_LOCAL */
///建立socketpair
        if (socketpair(AF_UNIX, SOCK_STREAM, 0, ngx_processes[s].channel) == -1)
        {
            ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                          "socketpair() failed while spawning \"%s\"", name);
            return NGX_INVALID_PID;
        }
        。。。。。。。。。。。。。。。。。。。。。。。。。。。。
///设置非阻塞模式
        if (ngx_nonblocking(ngx_processes[s].channel[0]) == -1) {
.....................................................
        }
```

```
        if (ngx_nonblocking(ngx_processes[s].channel[1]) == -1) {
.....................................
        }

///打开异步模式
        on = 1;
        if (ioctl(ngx_processes[s].channel[0], FIOASYNC, &on) == -1) {
.............................................
        }
///设置异步io的所有者
        if (fcntl(ngx_processes[s].channel[0], F_SETOWN, ngx_pid) == -1) {

.........................................
        }
///当exec后关闭句柄。
        if (fcntl(ngx_processes[s].channel[0], F_SETFD, FD_CLOEXEC) == -1) {.................
        }

        if (fcntl(ngx_processes[s].channel[1], F_SETFD, FD_CLOEXEC) == -1) {
  。。。。。。。。。。。。。。。。。。。。。。。。
        }
///设置当前的子进程的句柄
        ngx_channel = ngx_processes[s].channel[1];

    } else {
        ngx_processes[s].channel[0] = -1;
        ngx_processes[s].channel[1] = -1;
    }
```

接下来就是fork子进程，并设置进程相关参数。

```
///设置进程在进程表中的slot。
ngx_process_slot = s;



    pid = fork();

    switch (pid) {

    case -1:
        ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                      "fork() failed while spawning \"%s\"", name);
        ngx_close_channel(ngx_processes[s].channel, cycle->log);
        return NGX_INVALID_PID;

    case 0
///子进程，因此执行传递进来的子进程的函数
        ngx_pid = ngx_getpid();
        proc(cycle, data);
        break;

    default:
        break;
    }

    ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "start %s %P", name, pid);

    ngx_processes[s].pid = pid;
    ngx_processes[s].exited = 0;

///如果大于0,则说明我们确定了重启的子进程，因此下面的初始化就用已死的子进程的就够了。
    if (respawn >= 0) {
        return pid;
    }
///开始初始化进程结构。
    ngx_processes[s].proc = proc;
    ngx_processes[s].data = data;
    ngx_processes[s].name = name;
```

```
    ngx_processes[s].exiting = 0;


///设置相关状态。
    switch (respawn) {

    case NGX_PROCESS_RESPAWN:
        ngx_processes[s].respawn = 1;
        ngx_processes[s].just_respawn = 0;
        ngx_processes[s].detached = 0;
        break;

    case NGX_PROCESS_JUST_RESPAWN:
        ngx_processes[s].respawn = 1;
        ngx_processes[s].just_respawn = 1;
        ngx_processes[s].detached = 0;
        break;

    case NGX_PROCESS_DETACHED:
        ngx_processes[s].respawn = 0;
        ngx_processes[s].just_respawn = 0;
        ngx_processes[s].detached = 1;
        break;
    }

    if (s == ngx_last_process) {
        ngx_last_process++;
    }

return pid;
```

这里有个问题，那就是后面fork的子进程如何来让前面已经fork的子进程得到自己的进程相关信息呢。在nginx中是每次新的子进程fork完毕后，然后父进程此时将这个子进程id，以及流管道的句柄channel[0]传递给前面的子进程。这样子进程之间也可以通信了。

先来看相关的数据结构：

```
///封装了父子进程之间传递的信息。
typedef struct {
///对端将要做得命令。
     ngx_uint_t  command;
///当前的子进程id
     ngx_pid_t   pid;
///在全局进程表中的位置
     ngx_int_t   slot;
///传递的fd
     ngx_fd_t    fd;
} ngx_channel_t;
```

接下来来看代码：

```
static void
ngx_start_worker_processes(ngx_cycle_t *cycle, ngx_int_t n, ngx_int_t type)
{
    ngx_int_t      i, s;
    ngx_channel_t  ch;

.....................................
///传递给其他子进程的命令
    ch.command = NGX_CMD_OPEN_CHANNEL;

///这里n，就是从配置文件中读取的，需要几个子进程。
    for (i = 0; i < n; i++) {

        cpu_affinity = ngx_get_cpu_affinity(i);

///这个函数刚才介绍过了。就是fork子进程。
        ngx_spawn_process(cycle, ngx_worker_process_cycle, NULL,
```

```
                        "worker process", type);
///初始化channel，ngx_process_slot这个我们在上面的spawn函数中已经赋值完毕，就是当前子进程的位置。
        ch.pid = ngx_processes[ngx_process_slot].pid;
        ch.slot = ngx_process_slot;
        ch.fd = ngx_processes[ngx_process_slot].channel[0];

///遍历整个进程表
        for (s = 0; s < ngx_last_process; s++) {
///遇到非存活的进程就跳过。
            if (s == ngx_process_slot
                || ngx_processes[s].pid == -1
                || ngx_processes[s].channel[0] == -1)
            {
                continue;
            }

            ngx_log_debug6(NGX_LOG_DEBUG_CORE, cycle->log, 0,
                           "pass channel s:%d pid:%P fd:%d to s:%i pid:%P fd:%d",
                           ch.slot, ch.pid, ch.fd,
                           s, ngx_processes[s].pid,
                           ngx_processes[s].channel[0]);

            /* TODO: NGX_AGAIN */
///然后传递这个channel给其他子进程(主要是传递句柄)。
            ngx_write_channel(ngx_processes[s].channel[0],
                              &ch, sizeof(ngx_channel_t), cycle->log);
        }
    }
}
```

而在子进程中是如何处理的呢，子进程的管道可读事件捕捉函数是ngx_channel_handler(ngx_event_t *ev)，在这个函数中，会读取mseeage，然后解析，并根据不同的命令做不同的处理，来看它的代码片断：

```
///这里ch为读取的channel。

        switch (ch.command) {

        case NGX_CMD_QUIT:
            ngx_quit = 1;
            break;

        case NGX_CMD_TERMINATE:
            ngx_terminate = 1;
            break;

        case NGX_CMD_REOPEN:
            ngx_reopen = 1;
            break;

        case NGX_CMD_OPEN_CHANNEL:
///可以看到操作很简单，就是对ngx_processes全局进程表进行赋值。
            ngx_processes[ch.slot].pid = ch.pid;
            ngx_processes[ch.slot].channel[0] = ch.fd;
            break;

        case NGX_CMD_CLOSE_CHANNEL:
.................................................

            if (close(ngx_processes[ch.slot].channel[0]) == -1) {
                ngx_log_error(NGX_LOG_ALERT, ev->log, ngx_errno,
                              "close() channel failed");
            }

            ngx_processes[ch.slot].channel[0] = -1;
            break;
        }
```

接下来详细的来看worker和master如何进行交互，以及master如何同外部进行交互(比如热代码替换，

reconfig这些操作)。

在nginx中，worker和master的交互，我们前面已经提过了，就是通过流管道以及信号，而master与外部的交互是通过信号来进行的。

在看master得主循环之前，我们先来看信号处理和函数，在nginx中，父子进程的信号处理函数是相同的，只不过有一个变量在master和worker中赋值不同，以此来区分。

在信号处理中，通过设置相应的标志变量，从而在主循环中，判断这些变量，从而做相应的操作。

```
///定义的信号值。
#define NGX_SHUTDOWN_SIGNAL      QUIT
#define NGX_TERMINATE_SIGNAL     TERM
#define NGX_NOACCEPT_SIGNAL      WINCH
#define NGX_RECONFIGURE_SIGNAL   HUP

#if (NGX_LINUXTHREADS)
#define NGX_REOPEN_SIGNAL        INFO
#define NGX_CHANGEBIN_SIGNAL     XCPU
#else
#define NGX_REOPEN_SIGNAL        USR1
#define NGX_CHANGEBIN_SIGNAL     USR2
#endif


void
ngx_signal_handler(int signo)
{
    char            *action;
    ngx_int_t        ignore;
    ngx_err_t        err;
    ngx_signal_t    *sig;

    ignore = 0;

    err = ngx_errno;
```

```
///首先得到当前的信号值
    for (sig = signals; sig->signo != 0; sig++) {
        if (sig->signo == signo) {
            break;
        }
    }

    ngx_time_update(0, 0);

    action = "";

///这里ngx_process在master和worker中赋值不同。
    switch (ngx_process) {
///master中。
    case NGX_PROCESS_MASTER:
    case NGX_PROCESS_SINGLE:
        switch (signo) {

        case ngx_signal_value(NGX_SHUTDOWN_SIGNAL):
///如果接受到quit信号，则准备退出进程。
            ngx_quit = 1;
            action = ", shutting down";
            break;

        case ngx_signal_value(NGX_TERMINATE_SIGNAL):
        case SIGINT:
///sigint信号，则
            ngx_terminate = 1;
            action = ", exiting";
            break;

        case ngx_signal_value(NGX_NOACCEPT_SIGNAL):
///winch信号，停止接受accept。
            ngx_noaccept = 1;
            action = ", stop accepting connections";
            break;
```

```
        case ngx_signal_value(NGX_RECONFIGURE_SIGNAL):
///sighup信号用来reconfig
            ngx_reconfigure = 1;
            action = ", reconfiguring";
            break;

        case ngx_signal_value(NGX_REOPEN_SIGNAL):
///用户信号，用来reopen
            ngx_reopen = 1;
            action = ", reopening logs";
            break;
///热代码替换.
        case ngx_signal_value(NGX_CHANGEBIN_SIGNAL):
            if (getppid() > 1 || ngx_new_binary > 0) {

                /*
                 * Ignore the signal in the new binary if its parent is
                 * not the init process, i.e. the old binary's process
                 * is still running.  Or ignore the signal in the old binary's
                 * process if the new binary's process is already running.
                 */
///上面注释很详细，我就不解释了。。
                action = ", ignoring";
                ignore = 1;
                break;
            }
///正常情况下，需要热代码替换。设置标志位
            ngx_change_binary = 1;
            action = ", changing binary";
            break;

        case SIGALRM:
            break;

        case SIGIO:
            ngx_sigio = 1;
            break;
```

```
        case SIGCHLD:
///子进程已退出，设置标记。
            ngx_reap = 1;
            break;
        }

        break;
///worker的信号处理。worker的比较简单。
    case NGX_PROCESS_WORKER:
        switch (signo) {

        case ngx_signal_value(NGX_NOACCEPT_SIGNAL):
            ngx_debug_quit = 1;
        case ngx_signal_value(NGX_SHUTDOWN_SIGNAL):
            ngx_quit = 1;
            action = ", shutting down";
            break;

        case ngx_signal_value(NGX_TERMINATE_SIGNAL):
        case SIGINT:
            ngx_terminate = 1;
            action = ", exiting";
            break;

        case ngx_signal_value(NGX_REOPEN_SIGNAL):
            ngx_reopen = 1;
            action = ", reopening logs";
            break;

 ..........................................
        }

        break;
    }

.........................................
```

```
///最终如果信号是sigchld，我们收割僵尸进程(用waitpid)。
    if (signo == SIGCHLD) {
        ngx_process_get_status();
    }

    ngx_set_errno(err);
}
```

先来看master的主循环，处理其实很简单，就是在循环过程中判断相应的条件，然后进入相应的处理。这里的相关标志位基本都是在上面的信号处理函数中赋值的。：

```
for ( ;; ) {
///delay用来等待子进程退出的时间，由于我们接受到SIGINT信号后，我们需要先发送信号给子进程，而子进程的退
        if (delay) {
            delay *= 2;
.........................................

            itv.it_interval.tv_sec = 0;
            itv.it_interval.tv_usec = 0;
            itv.it_value.tv_sec = delay / 1000;
            itv.it_value.tv_usec = (delay % 1000 ) * 1000;
///设置定时器。
            if (setitimer(ITIMER_REAL, &itv, NULL) == -1) {
                ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                              "setitimer() failed");
            }
        }
///延时，等待定时器。

        sigsuspend(&set);

        ngx_time_update(0, 0);
```

```
        ngx_log_debug0(NGX_LOG_DEBUG_EVENT, cycle->log, 0, "wake up");

///ngx_reap为1,说明有子进程已经退出。
        if (ngx_reap) {
            ngx_reap = 0;
            ngx_log_debug0(NGX_LOG_DEBUG_EVENT, cycle->log, 0, "reap children");
///这个里面处理退出的子进程(有的worker异常退出，这时我们就需要重启这个worker )，如果所有子进程都退出则
            live = ngx_reap_children(cycle);
        }

///如果没有存活的子进程，并且收到了ngx_terminate或者ngx_quit信号，则master退出。
        if (!live && (ngx_terminate || ngx_quit)) {
            ngx_master_process_exit(cycle);
        }
///收到了sigint信号。
        if (ngx_terminate) {
///设置延时。
            if (delay == 0) {
                delay = 50;
            }

            if (delay > 1000) {
///如果超时，则强制杀死worker
                ngx_signal_worker_processes(cycle, SIGKILL);
            } else {
///负责发送sigint给worker，让它退出。
                ngx_signal_worker_processes(cycle,
                                            ngx_signal_value(NGX_TERMINATE_SIGNAL));
            }

            continue;
        }

///收到quit信号。
        if (ngx_quit) {
///发送给worker quit信号
```

```
            ngx_signal_worker_processes(cycle,
                                        ngx_signal_value(NGX_SHUTDOWN_SIGNAL));

            ls = cycle->listening.elts;
            for (n = 0; n < cycle->listening.nelts; n++) {
                if (ngx_close_socket(ls[n].fd) == -1) {
                    ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_socket_errno,
                                  ngx_close_socket_n " %V failed",
                                  &ls[n].addr_text);
                }
            }
            cycle->listening.nelts = 0;

            continue;
        }

///收到需要reconfig的信号
        if (ngx_reconfigure) {
            ngx_reconfigure = 0;
///判断是否热代码替换后的新的代码还在运行中(也就是还没退出当前的master)。如果还在运行中，则不需要重新初
            if (ngx_new_binary) {
                ngx_start_worker_processes(cycle, ccf->worker_processes,
                                           NGX_PROCESS_RESPAWN);
                ngx_start_cache_manager_process(cycle, NGX_PROCESS_RESPAWN);
                ngx_noaccepting = 0;

                continue;
            }

            ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "reconfiguring");
///重新初始化config，并重新启动新的worker
            cycle = ngx_init_cycle(cycle);
            if (cycle == NULL) {
                cycle = (ngx_cycle_t *) ngx_cycle;
                continue;
            }
```

```
            ngx_cycle = cycle;
            ccf = (ngx_core_conf_t *) ngx_get_conf(cycle->conf_ctx,
                                                   ngx_core_module);
            ngx_start_worker_processes(cycle, ccf->worker_processes,
                                       NGX_PROCESS_JUST_RESPAWN);
            ngx_start_cache_manager_process(cycle, NGX_PROCESS_JUST_RESPAWN);
            live = 1;
            ngx_signal_worker_processes(cycle,
                                        ngx_signal_value(NGX_SHUTDOWN_SIGNAL));
        }
///这个标志没弄懂有什么意义。代码里面是当热代码替换后，如果ngx_noacceptig被设置了，则设置这个标志位(难
        if (ngx_restart) {
            ngx_restart = 0;
            ngx_start_worker_processes(cycle, ccf->worker_processes,
                                       NGX_PROCESS_RESPAWN);
            ngx_start_cache_manager_process(cycle, NGX_PROCESS_RESPAWN);
            live = 1;
        }
///重新打开log
        if (ngx_reopen) {
            ngx_reopen = 0;
            ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "reopening logs");
            ngx_reopen_files(cycle, ccf->user);
            ngx_signal_worker_processes(cycle,
                                        ngx_signal_value(NGX_REOPEN_SIGNAL));
        }

///热代码替换
        if (ngx_change_binary) {
            ngx_change_binary = 0;
            ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "changing binary");
///进行热代码替换，这里是调用execve来执行新的代码。
            ngx_new_binary = ngx_exec_new_binary(cycle, ngx_argv);
        }
///接受到停止accept连接，其实也就是worker退出(有区别的是，这里master不需要退出)。。
        if (ngx_noaccept) {
            ngx_noaccept = 0;
```

```
            ngx_noaccepting = 1;
///给worker发送信号。
            ngx_signal_worker_processes(cycle,
                                        ngx_signal_value(NGX_SHUTDOWN_SIGNAL));
        }
    }
}
```

然后来看worker的主循环,worker的比较简单。逻辑和master的很相似：

```
 for ( ;; ) {
///ngx_exiting是当收到master的quit命令后，设置为1,然后等待其他资源退出。
        if (ngx_exiting) {

            c = cycle->connections;
.........................................
///定时器超时则退出worker
            if (ngx_event_timer_rbtree.root == ngx_event_timer_rbtree.sentinel)
            {
                ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "exiting");

                ngx_worker_process_exit(cycle);
            }
        }

        ngx_log_debug0(NGX_LOG_DEBUG_EVENT, cycle->log, 0, "worker cycle");

        ngx_process_events_and_timers(cycle);

///收到shutdown命令则worker直接退出
        if (ngx_terminate) {
            ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "exiting");
```

```
            ngx_worker_process_exit(cycle);
        }

///收到quit命令
        if (ngx_quit) {
            ngx_quit = 0;
            ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0,
                          "gracefully shutting down");
            ngx_setproctitle("worker process is shutting down");

            if (!ngx_exiting) {
///关闭socket，然后设置退出标志。
                ngx_close_listening_sockets(cycle);
                ngx_exiting = 1;
            }
        }

///收到master重新打开log的命令。
        if (ngx_reopen) {
            ngx_reopen = 0;
            ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "reopening logs");
            ngx_reopen_files(cycle, -1);
        }
    }
```

1.2 nginx的内存管理

发表时间: 2009-12-10

先来看内存池的实现，nginx的内存池实现的非常简单。

这里内存池的一些图表可以看老朱同学的slides ：

http://blog.zhuzhaoyuan.com/2009/09/nginx-internals-slides-video/

当内存池初始化的时候(下面会分析到)ngx_poll_s只相当于内存池的一个头，保存了当前内存池的一些必要信息而已。

当从内存池存取数据的时候，nginx是分为两种类型来处理得，一种是小块数据，它是直接从内存池中取得数据，另一方面，当为大块数据时，它是直接malloc一块数据(也就是从内存池外部分配数据),然后保存这个指针到内存池。可以看到很多内存池，比如py的内存池实现，也基本是这个思想。这里的细节，我们将会在下面分析内存池相关函数的时候详细分析。

这里还有一个要注意就是这些子内存池和父内存池是不一样的，我们后面分析函数的时候会详细介绍。

大小块数据得分割线是你创建内存池时传递进来的size和页大小之间的最小值。

下面就是内存池的结构：

```
struct ngx_pool_s {
///数据区的指针
    ngx_pool_data_t       d;
///其实也就是内存池所能容纳的最大值。
    size_t                max;
///指向当前的内存池的头。
    ngx_pool_t           *current;
///这个主要是为了讲所有的内存池都链接起来。(他会创建多个内存池的)
    ngx_chain_t          *chain;
///这个链表表示大的数据块
    ngx_pool_large_t     *large;
```

```
///这个就是清理函数链表
    ngx_pool_cleanup_t   *cleanup;
    ngx_log_t            *log;
};
```

然后我们一个个来看上面的链表。首先是数据区的指针ngx_pool_data_t。

这个结构很简单，它就是包含了我们所需要操作这个内存池的数据的一些指针。

其中last表示当前的数据区的已经使用的数据的结尾。

end表示当前的内存池的结尾。也就是说end-last就是内存池未使用的大小。

我们要知道当我们从一个内存池请求一块内存时，如果此时内存池已经满掉，这是一般都是扩大内存池，而nginx中不是这么做的，它会直接再分配一个内存池，然后链接到data的next指针上。也就是说在nginx中，其实每个内存池都会包含一些子内存池。因此我们请求内存的时候都会遍历这些子内存池。

failed域主要是为了标记我们请求内存(小块内存)由于内存池空间不够，我们需要重新分配一个子内存池的次数。 下面分析函数的时候会再会看到这个域。

```
typedef struct {
    u_char               *last;
    u_char               *end;
//指向下一块内存池
    ngx_pool_t           *next;
///失败标记
    ngx_uint_t            failed;
} ngx_pool_data_t;
```

ngx_chain_t这里就先不介绍了，我们现在只需要知道它是与buf相关的。

然后是ngx_pool_large_s，它表示了大块的内存。可以看到这个结构非常简单，就是一个指针指向下一块

large，一个alloc指向数据。

```
struct ngx_pool_large_s {
    ngx_pool_large_t     *next;
    void                 *alloc;
};
```

接下来是ngx_pool_cleanup_s，这个结构用来表示内存池中的数据的清理handler。

其中handler表示清理函数。
data表示传递给清理函数的数据。
next表示下一个清理handler，也就是说当destroy这个pool的时候会遍历清理handler链表，然后调用handler。

```
struct ngx_pool_cleanup_s {
    ngx_pool_cleanup_pt   handler;
    void                 *data;
    ngx_pool_cleanup_t   *next;
}
```

通过ngx_create_temp_buf创建一个buff,然后通过ngx_alloc_chain_link创建一个chain,然后通过cl->buf = rb->buf;将buff链接到chain中.

下面就是pool的内存图，摘自老朱同学的nginx internal。

ok，接下来我们来通过分析具体的函数，来更好的理解pool的实现。

首先来看ngx_create_pool也就是创建一个pool。

这里我们要知道虽然我们传递进来的大小是size可是我们真正能使用的数据区大小是要减去ngx_pool_t的大小

的。

```
///内存池的数据区的最大容量。
#define NGX_MAX_ALLOC_FROM_POOL  (ngx_pagesize - 1)

ngx_pool_t *
ngx_create_pool(size_t size, ngx_log_t *log)
{
    ngx_pool_t  *p;
///可以看到直接分配size大小，也就是说我们只能使用size-sizeof(ngx_poll_t)大小
    p = ngx_alloc(size, log);
    if (p == NULL) {
        return NULL;
    }

///开始初始化数据区。

///由于一开始数据区为空，因此last指向数据区的开始。
    p->d.last = (u_char *) p + sizeof(ngx_pool_t);
///end也就是数据区的结束位置
    p->d.end = (u_char *) p + size;
    p->d.next = NULL;
    p->d.failed = 0;

///这里才是我们真正能使用的大小。
    size = size - sizeof(ngx_pool_t);

///然后设置max。内存池的最大值也就是size和最大容量之间的最小值。
    p->max = (size < NGX_MAX_ALLOC_FROM_POOL) ? size : NGX_MAX_ALLOC_FROM_POOL;

///current表示当前的内存池。
    p->current = p;

///其他的域置NULL。
    p->chain = NULL;
    p->large = NULL;
```

```
    p->cleanup = NULL;
    p->log = log;
///返回指针。
    return p;
}
```

接下来我们来看如何从内存池中分配一块内存来使用。在nginx中有3个函数可以使用，分别是ngx_palloc，ngx_calloc，ngx_pnalloc。这三个函数的区别就是第一个函数分配的内存会对齐。第二个函数用来分配一块清0的内存，第三个函数分配的内存不会对齐。

由于这三个函数差不多，因此我们就只分析一个就够了。我们就来看ngx_palloc.

```
void *
ngx_palloc(ngx_pool_t *pool, size_t size)
{
    u_char      *m;
    ngx_pool_t  *p;

///首先判断当前申请的大小是否超过max，如果超过则说明是大块，此时进入large
    if (size <= pool->max) {

///得到当前的内存池指针。
        p = pool->current;

///开始遍历内存池，
        do {
///首先对齐last指针。
            m = ngx_align_ptr(p->d.last, NGX_ALIGNMENT);

///然后得到当前内存池中的可用大小。如果大于请求大小，则直接返回当前的last，也就是数据的指针。
            if ((size_t) (p->d.end - m) >= size) {
///更新last，然后返回前面保存的last。
                p->d.last = m + size;
```

```
                return m;
            }
///否则继续遍历
            p = p->d.next;

        } while (p);
///到达这里说明内存池已经满掉，因此我们需要重新分配一个内存池然后链接到当前的data的next上。(紧接着我们会
        return ngx_palloc_block(pool, size);
    }

///申请大块。
    return ngx_palloc_large(pool, size);
}
```

接下来就来看ngx_palloc_block的实现，这个函数主要就是重新分配一块内存池，然后链接到当前内存池的数据区指针。

然后要注意这里重新new的这个内存池大小是和它的父内存池一样大的。

并且新得内存池只保存了ngx_pool_data_t这个结构，也就是说数据区直接跟在ngx_pool_data_t下面。

```
static void *
ngx_palloc_block(ngx_pool_t *pool, size_t size)
{
    u_char      *m;
    size_t       psize;
    ngx_pool_t  *p, *new, *current;

///计算当前的内存池的大小。
    psize = (size_t) (pool->d.end - (u_char *) pool);

///再分配一个同样大小的内存池
```

```
    m = ngx_alloc(psize, pool->log);
    if (m == NULL) {
        return NULL;
    }

    new = (ngx_pool_t *) m;

///接下来和我们create一个内存池做的操作一样。就是更新一些指针
    new->d.end = m + psize;
    new->d.next = NULL;
    new->d.failed = 0;

///这里要注意了，可以看到last指针是指向ngx_pool_data_t的大小再加上要分配的size大小，也就是现在的内存池
    m += sizeof(ngx_pool_data_t);
    m = ngx_align_ptr(m, NGX_ALIGNMENT);
    new->d.last = m + size;

///设置current。
    current = pool->current;

///这里遍历所有的子内存池，这里主要是通过failed来标记重新分配子内存池的次数，然后找出最后一个大于4的，标
    for (p = current; p->d.next; p = p->d.next) {
        if (p->d.failed++ > 4) {
            current = p->d.next;
        }
    }

///链接到最后一个内存池后面
    p->d.next = new;

///如果current为空，则current就为new。
    pool->current = current ? current : new;

    return m;
}
```

这里解释一下为什么这样设置current，这里的主要原因是我们在ngx_palloc中分配内存是从current开始的，而这里也就是设置current为比较新分配的内存。而当failed大于4说明我们至少请求了4次内存分配，都不能满足我们的请求，此时我们就假设老的内存都已经没有空间了，因此我们就从比较新的内存块开始。

接下来是ngx_palloc_large，这个函数也是很简单就是malloc一块ngx_poll_large_t,然后链接到主的内存池上。

```
static void *
ngx_palloc_large(ngx_pool_t *pool, size_t size)
{
    void              *p;
    ngx_uint_t         n;
    ngx_pool_large_t  *large;

///分配一块内存。
    p = ngx_alloc(size, pool->log);
    if (p == NULL) {
        return NULL;
    }

    n = 0;
///开始遍历large链表，如果有alloc(也就是内存区指针)为空，则直接指针赋值然后返回。一般第一次请求大块内存
    for (large = pool->large; large; large = large->next) {
        if (large->alloc == NULL) {
            large->alloc = p;
            return p;
        }
///这里不太懂什么意思。
        if (n++ > 3) {
            break;
        }
    }
```

```
///malloc一块ngx_pool_large_t。
    large = ngx_palloc(pool, sizeof(ngx_pool_large_t));
    if (large == NULL) {
        ngx_free(p);
        return NULL;
    }

///然后链接数据区指针p到large。这里可以看到直接插入到large链表的头的。
    large->alloc = p;
    large->next = pool->large;
    pool->large = large;

    return p;
}
```

ok，分配看完了，我们来看释放。这里要知道在nginx中，只有大块内存提供了free接口，可以供我们收工释放，而小块内存是没有提供这个接口的。也就是说小块内存只有当整个内存池被desrtoy掉时，才会被释放。

这里一些简单的函数就不分析了。
比如ngx_pfree(ngx_pool_t *pool, void *p)，这个函数就是从pool的large链表中找到p，然后free掉它。

ngx_pool_cleanup_t *
ngx_pool_cleanup_add(ngx_pool_t *p, size_t size)

这个函数也就是添加一个ngx_pool_cleanup_t到当前的pool上，然后返回，我们此时就能通过返回的结构来给对应的handler赋值。

而ngx_pool_cleanup_t这个主要是当内存池destroy的时候我们可能需要做一些清理工作，此时我们就能add这些清理handler到pool中，然后当内存池要释放的时候就会自动调用。

ok，现在来看pool 被free的实现。

这个函数主要是遍历large，遍历current，然后一一释放。

```
void
ngx_destroy_pool(ngx_pool_t *pool)
{
    ngx_pool_t          *p, *n;
    ngx_pool_large_t    *l;
    ngx_pool_cleanup_t  *c;

///先做清理工作。
    for (c = pool->cleanup; c; c = c->next) {
        if (c->handler) {
            ngx_log_debug1(NGX_LOG_DEBUG_ALLOC, pool->log, 0,
                           "run cleanup: %p", c);
            c->handler(c->data);
        }
    }

///free大块内存
    for (l = pool->large; l; l = l->next) {

        ngx_log_debug1(NGX_LOG_DEBUG_ALLOC, pool->log, 0, "free: %p", l->alloc);

        if (l->alloc) {
            ngx_free(l->alloc);
        }
    }

///遍历小块内存池。
    for (p = pool, n = pool->d.next; /* void */; p = n, n = n->d.next) {
///直接free掉。
        ngx_free(p);

        if (n == NULL) {
            break;
        }
    }
}
```

通过上面我们可以看到在nginx中内存池中的小块数据是从来不释放的，这样就简化了内存池的操作。

接下来我们来看buf的实现。

buf分为两种类型,一种是file,一种是memory.因此这里会有文件的一些操作域。

可以看到buf相对于pool多了一个pos域(file_pos).这里我们要知道我们发送往套接字异或者其他的设备，我们这里会现将数据放到buf中，然后当设备或者套接字准备好了，我们就会从buf中读取，因此这里pos指针就是放到buf中的已经被执行的数据(也就是已经送往套接字)的位置。

```
struct ngx_buf_s {
///pos表示已经执行的数据的位置。
    u_char          *pos;
///last和上面内存池中last一样，也就是使用的内存的最后一个字节的指针
    u_char          *last;
///文件指针
    off_t            file_pos;
    off_t            file_last;
///buf的开始指针
    u_char          *start;         /* start of buffer */
    u_char          *end;           /* end of buffer */

///这里表示这个buf从属于那个模块。
    ngx_buf_tag_t    tag;
    ngx_file_t      *file;
    ngx_buf_t       *shadow;

///一些标记
    /* the buf's content could be changed */
    unsigned         temporary:1;

///在内存中是不能改变的。
    unsigned         memory:1;
```

```
///是否是mmap的内存
    unsigned         mmap:1;

    unsigned         recycled:1;

///是否文件。
    unsigned         in_file:1;
    unsigned         flush:1;
    unsigned         sync:1;
    unsigned         last_buf:1;
    unsigned         last_in_chain:1;

    unsigned         last_shadow:1;
    unsigned         temp_file:1;

    /* STUB */ int   num;
};
```

ok，接下来我们来看如何创建一个buf.在nginx中一般都是调用ngx_create_temp_buf来创建一个buf。函数很简单，就是从pool中分配内存然后初始化相关域。

```
ngx_buf_t *
ngx_create_temp_buf(ngx_pool_t *pool, size_t size)
{
    ngx_buf_t *b;

///calloc一个buf，可以看到它调用的是calloc，也就是说都会清0.
    b = ngx_calloc_buf(pool);
    if (b == NULL) {
        return NULL;
    }

///然后从内存池中分配一块内存。并将这块内存链接到b->start.
```

```
    b->start = ngx_palloc(pool, size);
    if (b->start == NULL) {
        return NULL;
    }
///设置相关的域。

    b->pos = b->start;
    b->last = b->start;
///设置打消
    b->end = b->last + size;
    b->temporary = 1;

    return b;
}
```

然后我们来看chain的实现，chain其实也就是多个buf组合而成的。它主要是用来缓存一些未发出去的，或者接收的buf 以及 writev以及readv而存在的。

ok我们来看chain的实现，其实它的实现很简单，就是一个单链表。

```
struct ngx_chain_s {
///buf
    ngx_buf_t    *buf;
///下一个buf的指针。
    ngx_chain_t  *next;
};
```

然后来看如何创建一个chain。这里取得一个chain后直接返回给供其他模块使用：

```

ngx_chain_t *
```

```
ngx_alloc_chain_link(ngx_pool_t *pool)
{
    ngx_chain_t  *cl;

///取得pool的老的chain
    cl = pool->chain;
///如果chain已经存在，则直接返回这个chain，然后从pool的chain中删除这个chain。
    if (cl) {
        pool->chain = cl->next;
        return cl;
    }

///否则从内存池重新new一个chain。这里注意新建的这个chain是链接到pool的chain上的。
    cl = ngx_palloc(pool, sizeof(ngx_chain_t));
    if (cl == NULL) {
        return NULL;
    }

///然后返回
    return cl;
}
```

接下来就是两个重要的chain，它们其实就是对chain再进行了一次封装。

1 ngx_output_chain_ctx_t ， 这个chain主要是管理输出buf。

2 ngx_chain_writer_ctx_t 这个主要是用在upstream模块。

因此我们主要来看ngx_output_chain_ctx_t。

ngx_output_chain_ctx_t，它包含了三种类型的chain，分别是in，free以及busy。

现在来介绍这几个重要的域：

buf ： 这个域也就是我们拷贝数据的地方，我们一般输出的话都是从in直接copy相应的size到buf中。

in ： 这个就是我们保存那些需要发送数据的地方。

free ： 这个保存了一些空的buf，也就是说如果free存在，我们都会直接从free中取buf到前面的buf域。

busy ：这个保存了已经发送完毕的buf，也就是每次我们从in中将buf读取完毕后，确定数据已经取完，此时就会将这个chain拷贝到busy中。然后将比较老的busy buf拷贝到free中。

output_filter是一个回调函数，用来过滤输出。

剩下的就是一些标记域。

```
typedef struct {
    ngx_buf_t                    *buf;
    ngx_chain_t                  *in;
    ngx_chain_t                  *free;
    ngx_chain_t                  *busy;

///相关的标记，是否使用sendfile，是否使用directio等等。。
    unsigned                      sendfile:1;
    unsigned                      directio:1;
#if (NGX_HAVE_ALIGNED_DIRECTIO)
    unsigned                      unaligned:1;
#endif
    unsigned                      need_in_memory:1;
    unsigned                      need_in_temp:1;

///内存池。
    ngx_pool_t                   *pool;
///每次从pool中重新alloc一个buf这个值都会相应加一。
    ngx_int_t                     allocated;
    ngx_bufs_t                    bufs;
///这个用来标记当前那个模块使用这个chain
    ngx_buf_tag_t                 tag;

    ngx_output_chain_filter_pt    output_filter;
```

```
    void                        *filter_ctx;
} ngx_output_chain_ctx_t;
```

它对应的主要是ngx_output_chain函数。这个函数主要功能就是拷贝in chain的数据到buf域中。这个函数很复杂，我们这里简要分析一下：

```
ngx_int_t
ngx_output_chain(ngx_output_chain_ctx_t *ctx, ngx_chain_t *in)
{
    off_t         bsize;
    ngx_int_t     rc, last;
    ngx_chain_t  *cl, *out, **last_out;

.....................................

    /* add the incoming buf to the chain ctx->in */

///拷贝in 到ctx的in chain中。
    if (in) {
        if (ngx_output_chain_add_copy(ctx->pool, &ctx->in, in) == NGX_ERROR) {
            return NGX_ERROR;
        }
    }

    out = NULL;
    last_out = &out;
    last = NGX_NONE;

///开始循环处理ctx-in chain.这里有两层循环。
    for ( ;; ) {

///开始遍历in
        while (ctx->in) {
```

```
  ///计算当前in的buf长度。这个长度也就是还没处理的数据长度。

            bsize = ngx_buf_size(ctx->in->buf);
.....................................................
///如果buf为空，则我们需要给buf分配空间。

            if (ctx->buf == NULL) {
///这个函数很简单，主要是处理file buf，如果是file buf则会create一个buf链接到ctx
                rc = ngx_output_chain_align_file_buf(ctx, bsize);

                if (rc == NGX_ERROR) {
                    return NGX_ERROR;
                }

///如果是memory buf，都会到这里
                if (rc != NGX_OK) {
///如果free不为空，则我们从free chain中取得buf。
                    if (ctx->free) {

                        /* get the free buf */

                        cl = ctx->free;
                        ctx->buf = cl->buf;
                        ctx->free = cl->next;

                        ngx_free_chain(ctx->pool, cl);

                    } else if (out || ctx->allocated == ctx->bufs.num) {

                        break;

                    }

///否则我们要重新create一个buf，然后链接到ctx，这里主要buf的大小和in chain的没有处理的数据一样大。
else if (ngx_output_chain_get_buf(ctx, bsize) != NGX_OK) {
                        return NGX_ERROR;
```

```
                }
            }
        }

///从in chain拷贝数据到buf，并更新相关域。
        rc = ngx_output_chain_copy_buf(ctx);

        if (rc == NGX_ERROR) {
            return rc;
        }

        if (rc == NGX_AGAIN) {
            if (out) {
                break;
            }

            return rc;
        }

    ///如果size为0,说明in chain中的第一个chain的数据已经被拷贝完了，此时删除这个chain。

        if (ngx_buf_size(ctx->in->buf) == 0) {
            ctx->in = ctx->in->next;
        }

///重新分配一个 chain
        cl = ngx_alloc_chain_link(ctx->pool);
        if (cl == NULL) {
            return NGX_ERROR;
        }

///链接buf到cl
        cl->buf = ctx->buf;
        cl->next = NULL;
        *last_out = cl;
        last_out = &cl->next;
        ctx->buf = NULL;
```

```
        }

        if (out == NULL && last != NGX_NONE) {

            if (ctx->in) {
                return NGX_AGAIN;
            }

            return last;
        }

///调用回调函数
        last = ctx->output_filter(ctx->filter_ctx, out);

        if (last == NGX_ERROR || last == NGX_DONE) {
            return last;
        }

///update 相关的chain，主要是将刚才copy完的buf 加入到busy chain，然后从busy chain中取出buf放到free
        ngx_chain_update_chains(&ctx->free, &ctx->busy, &out, ctx->tag);
        last_out = &out;
    }
}
```

这里我只是简要的分析了下，详细的还需要接合其他模块来看。

最后来看ngx_chain_writer_ctx_t，这个主要用于ustream(由于没看这个模块，因此不理解这里为什么要多出来个writer).大概看了，觉得应该是ustream模块发送的数据量比较大，因此这里通过这个chain来直接调用writev来将数据发送出去。

```
typedef struct {
///保存了所要输出的chain。
    ngx_chain_t                 *out;
```

```
///这个保存了这次新加入的所需要输出的chain。
    ngx_chain_t                **last;
///这个表示当前连接
    ngx_connection_t            *connection;
    ngx_pool_t                  *pool;
    off_t                        limit;
} ngx_chain_writer_ctx_t;
```

这里我们要知道out是会变化的。每次输出后，这个都会指向下一个需要发送的chain。

```
ngx_int_t
ngx_chain_writer(void *data, ngx_chain_t *in)
{
    ngx_chain_writer_ctx_t *ctx = data;

    off_t              size;
    ngx_chain_t       *cl;
    ngx_connection_t  *c;

    c = ctx->connection;
///这里将in中的也就是新加如的chain ，全部复制到last中。也就是它保存了最后的数据。
    for (size = 0; in; in = in->next) {
.....................................

///计算大小。
        size += ngx_buf_size(in->buf);


        cl = ngx_alloc_chain_link(ctx->pool);
        if (cl == NULL) {
            return NGX_ERROR;
        }
```

```
///加入last
        cl->buf = in->buf;
        cl->next = NULL;
        *ctx->last = cl;
        ctx->last = &cl->next;
    }

    ngx_log_debug1(NGX_LOG_DEBUG_CORE, c->log, 0,
                   "chain writer in: %p", ctx->out);

///遍历out chain
    for (cl = ctx->out; cl; cl = cl->next) {

///计算所需要输出的大小
        size += ngx_buf_size(cl->buf);
    }

    if (size == 0 && !c->buffered) {
        return NGX_OK;
    }

///调用send_chain(一般是writev)来输出out中的数据。
    ctx->out = c->send_chain(c, ctx->out, ctx->limit);

.......................

    return NGX_AGAIN;
}
```

1.3 nginx中request请求的解析

发表时间: 2010-04-04

ngx_http_init_request 中初始化event 的handler 为ngx_http_process_request_line，然后首先调用ngx_http_read_request_header来读取头部，然后就是开始调用函数ngx_http_parse_request_line对request line进行解析。随后如果解析的url是complex的话，就进入complex的解析，最后进入headers的解析。

```
static void
ngx_http_process_request_line(ngx_event_t *rev)
{
 ..............................................................

    for ( ;; ) {

        if (rc == NGX_AGAIN) {
//读取request头部
            n = ngx_http_read_request_header(r);

            if (n == NGX_AGAIN || n == NGX_ERROR) {
                return;
            }
        }

//开始解析request请求头部。
        rc = ngx_http_parse_request_line(r, r->header_in);
.....................................................................................


if (r->complex_uri || r->quoted_uri) {

  ..............................
//解析complex uri。

                rc = ngx_http_parse_complex_uri(r, cscf->merge_slashes);
```

```
.....................................

            }
.................................................

//执行request header并且解析headers。
ngx_http_process_request_headers(rev);

}
```

这里nginx将request的解析分为三个部分，第一个是request-line部分，第二个是complex ui部分，第三个是request header部分。

在看nginx的处理之前，我们先来熟悉一下http的request-line的格式。

首先来看request 的格式：

引用

```
<method> <request-URL> <version>
<headers>

<entity-body>
```

其中第一行就是request-line,我们先来看
然后我们一个个来看，首先是method：

引用

```
Method = "OPTIONS"
| "GET"
| "HEAD"
| "POST"
| "PUT"
| "DELETE"
| "TRACE"
| "CONNECT"
| extension-method
extension-method = token
```

然后是URL,这里要注意这个是URL的完整格式，而我们如果和server直接通信的话，一般只需要path就够了。

引用

```
<scheme>://<user>:<password>@<host>:<port>/<path>;<params>?<query>#<frag>
```

然后是version的格式：

引用

```
HTTP/<major>.<minor>
```

整个request-line解析的一个状态，在ngx_http_parse.c中定义,这个状态保存在ngx_http_request_t 中的state中，它表示当前的request line解析到那一步了，其实也就是个状态机。

```
enum {
        sw_start = 0,
        sw_method,
```

```
        sw_spaces_before_uri,
        sw_schema,
        sw_schema_slash,
        sw_schema_slash_slash,
        sw_host,
        sw_port,
        sw_after_slash_in_uri,
        sw_check_uri,
        sw_uri,
        sw_http_09,
        sw_http_H,
        sw_http_HT,
        sw_http_HTT,
        sw_http_HTTP,
        sw_first_major_digit,
        sw_major_digit,
        sw_first_minor_digit,
        sw_minor_digit,
        sw_spaces_after_digit,
        sw_almost_done
    } state;
```

而这里nginx这些状态就是为了解析这些东西。

接下来来通过代码片断看这些状态的含义。不过在看之前一定要

首先是start状态，也就是初始状态，我们刚开始解析Request的时候，就是这个状态。

```
case sw_start:
            r->request_start = p;

//如果是回车换行则跳出switch，然后继续解析
            if (ch == CR || ch == LF) {
                break;
```

```
            }
//不是A到Z的字母(大小写敏感的),并且不是_则返回错误
            if ((ch < 'A' || ch > 'Z') && ch != '_') {
                return NGX_HTTP_PARSE_INVALID_METHOD;
            }
//到达这里说明下一步改解析方法了。因此下一个状态就是method
            state = sw_method;
            break;
```

然后是method状态，这个状态表示我们正在解析请求的method。
下面就是http的请求方法：

引用

```
Method = "OPTIONS"
| "GET"
| "HEAD"
| "POST"
| "PUT"
| "DELETE"
| "TRACE"
| "CONNECT"
| extension-method
extension-method = token
```

由于METHOD比较多，而且代码都比较重复，因此这里就看看几个代码片断.
由于

```
case sw_method:

//如果再次读到空格则说明我们已经准备解析request-URL，此时我们就能得到请求方法了。
            if (ch == ' ') {
//先得到method的结束位置
```

```
                r->method_end = p - 1;
//开始位置前面已经保存。
                m = r->request_start;

//得到方法的长度，通过长度来得到具体不同的方法，然后给request的method赋值。
                switch (p - m) {
                case 3:
                    if (ngx_str3_cmp(m, 'G', 'E', 'T', ' ')) {
                        r->method = NGX_HTTP_GET;
                        break;
                    }

                    if (ngx_str3_cmp(m, 'P', 'U', 'T', ' ')) {
                        r->method = NGX_HTTP_PUT;
                        break;
                    }

                    break;
.........................................
             }
//下一个状态准备开始解析URI
         state = sw_spaces_before_uri;
break;
.....................................................................
```

然后是sw_spaces_before_uri状态，这里由于uri会有两种情况，一种是带schema的，一种是直接相对路径的(可以看前面的uri格式).

```
case sw_spaces_before_uri:

//如果是以/开始，则进入sw_after_slash_in_uri
            if (ch == '/' ){
                r->uri_start = p;
                state = sw_after_slash_in_uri;
```

```
                break;
            }

            c = (u_char) (ch | 0x20);
//如果是字母，则进入sw_schema处理
            if (c >= 'a' && c <= 'z') {
                r->schema_start = p;
                state = sw_schema;
                break;
            }

            switch (ch) {
//空格的话继续这个状态。
            case ' ':
                break;
            default:
                return NGX_HTTP_PARSE_INVALID_REQUEST;
            }
            break;
```

sw_schema状态主要是用来解析协议类型。等到协议类型解析完毕则进入sw_schema_slash状态.

```
case sw_schema:

            c = (u_char) (ch | 0x20);
//如果是字母则break，然后继续这个状态的处理。
            if (c >= 'a' && c <= 'z') {
                break;
            }
//到这里说明schema已经结束。
            switch (ch) {
//这里必须是:，如果不是冒号则直接返回错误。
            case ':':
//设置schema_end,而start我们在上面已经设置过了
```

```
                r->schema_end = p;
//设置下一个状态。
                state = sw_schema_slash;
                break;
            default:
                return NGX_HTTP_PARSE_INVALID_REQUEST;
            }
            break;
```

sw_schema_slash和sw_schema_slash_slash是两个很简单的状态，第一个是得到schema的第一个/，然后进入sw_schema_slash_slash，而sw_schema_slash_slash则是得到了第二个/.然后进入sw_host。

```
case sw_schema_slash:
            switch (ch) {
            case '/':
//进入slash_slash
                state = sw_schema_slash_slash;
                break;
            default:
                return NGX_HTTP_PARSE_INVALID_REQUEST;
            }
            break;
case sw_schema_slash_slash:
            switch (ch) {
            case '/':
//设置host的开始指针
                r->host_start = p + 1;
//设置下一个状态为sw_host.
                state = sw_host;
                break;
            default:
                return NGX_HTTP_PARSE_INVALID_REQUEST;
```

```
            }
            break;
```

然后是sw_host状态，这个状态用来解析host。

```
case sw_host:
            c = (u_char) (ch | 0x20);
//这里很奇怪，不知道为什么不把判断写在一起。
            if (c >= 'a' && c <= 'z') {
                break;
            }

            if ((ch >= '0' && ch <= '9') || ch == '.' || ch == '-') {
                break;
            }
//到达这里说明host已经得到，因此设置end指针。
            r->host_end = p;

            switch (ch) {
//冒号说明host有跟端口的，因此进入port状态。
            case ':':
                state = sw_port;
                break;
//这个说明要开始解析path了。因此设置uri的start，然后进入slash_in_uri
            case '/':
                r->uri_start = p;
                state = sw_after_slash_in_uri;
                break;
//如果是空格，则设置uri的start和end然后进入http_09
            case ' ':
                r->uri_start = r->schema_end + 1;
                r->uri_end = r->schema_end + 2;
                state = sw_http_09;
                break;
```

```
            default:
                return NGX_HTTP_PARSE_INVALID_REQUEST;
            }
            break;
```

接下来是sw_port，这个状态用来解析协议端口。

```
case sw_port:
            if (ch >= '0' && ch <= '9') {
                break;
            }
//如果到达这里说明端口解析完毕， 然后就来判断下一步需要的状态。
            switch (ch) {
//如果紧跟着/，则说明后面是uri，因此进入uri解析，并设置port_end
            case '/':
                r->port_end = p;
                r->uri_start = p;
                state = sw_after_slash_in_uri;
                break;
//如果是空格则设置port end，并进入http_09状态。
            case ' ':
                r->port_end = p;
                r->uri_start = r->schema_end + 1;
                r->uri_end = r->schema_end + 2;
                state = sw_http_09;
                break;
            default:
                return NGX_HTTP_PARSE_INVALID_REQUEST;
            }
            break;
```

接下来是sw_after_slash_in_uri，sw_check_uri 这两个状态都是解析uri之前的状态，主要用于检测uri，比如

complex uri等。

这里要对uri的格式比较熟悉，这里可以去看rfc3986，里面对uri的格式有比较清楚的描述。

因此我们主要来看sw_uri状态，这个状态就是开始解析uri。这里可以看到对http 0.9是特殊处理的，如果直接是回车或者换行的话，就进入http 0.9的处理。

```
case sw_uri:

            if (usual[ch >> 5] & (1 << (ch & 0x1f))) {
                break;
            }

            switch (ch) {
//下面三种情况都说明是http 0.9
            case ' ':
                r->uri_end = p;
                state = sw_http_09;
                break;
            case CR:
                r->uri_end = p;
                r->http_minor = 9;
                state = sw_almost_done;
                break;
            case LF:
                r->uri_end = p;
                r->http_minor = 9;
                goto done;
//要对段进行解析。因此设置complex uri
            case '#':
                r->complex_uri = 1;
                break;
            case '\0':
                r->zero_in_uri = 1;
                break;
            }
            break;
```

接下来的sw_http_09,sw_http_H,sw_http_HT,sw_http_HTT,sw_http_HTTP,
sw_first_major_digit,sw_major_digit,sw_first_minor_digit,sw_minor_digit,这几个状态主要是用来解析http的版本号的，都比较简单，这里就不仔细分析了。

然后来看最后两个状态sw_spaces_after_digit和sw_almost_done。

第一个状态表示已经解析完http状态了，然后发现有空格。

```
case sw_spaces_after_digit:
            switch (ch) {
            case ' ':
                break;
//如果是回车，则进入almost_done,然后等待最后一个换行。
            case CR:
                state = sw_almost_done;
                break;
//如果是换行则说明request-line解析完毕
            case LF:
                goto done;
            default:
                return NGX_HTTP_PARSE_INVALID_REQUEST;
            }
            break;
```

最后是almost_done状态，也就是等待最后的换行。

```
case sw_almost_done:
            r->request_end = p - 1;
            switch (ch) {
            case LF:
                goto done;
            default:
                return NGX_HTTP_PARSE_INVALID_REQUEST;
```

```
            }
        }
```

接下来是complex_uri的解析，这里比如段(#)，比如win下的\\，等等，这里这个函数就不分析了，方法和request-line的差不多。

这里主要来看一下header的实现。

headers的格式我们知道一个name紧跟着一个冒号:,然后紧跟着一个可选的空格，然后是一个value，最后以一个CRLF结束，而headers的结束是一个CRLF。

下面是解析header时的状态列表：

```
    enum {
        sw_start = 0,
        sw_name,
        sw_space_before_value,
        sw_value,
        sw_space_after_value,
        sw_ignore_line,
        sw_almost_done,
        sw_header_almost_done
    } state;
```

解析是ngx_http_parse_header_line来做的，这个函数一次只能解析一个header，如果传递进来的buf会有多个header，则是它只处理一个header，然后设置好对应的buf的域，等待下次再进行解析。

而这个函数能够返回三个值，第一个是NGX_OK,这个表示一个header解析完毕，第二个是NGX_AGAIN，表示header没有解析完毕，也就是说buf只有一部分的数据。这个时候，下次进来的数据会继续没有完成的解析。第三个是NGX_HTTP_PARSE_HEADER_DONE;，表示整个header已经解析完毕。

而对应的goto DONE表示NGX_OK,goto HEADER_DONE表示NGX_HTTP_PARSE_HEADER_DONE，而默认

则是NGX_AFAIN.

这里有几个request的值需要先说明一下。

先来看request的域：

header_name_start 这个表示header_name的起始位置。
header_name_end 这个表示当前header_name的结束位置
header_start 这个是value的起始位置
header_end 这个是value的结束位置

header_hash 这个是header name的hash值。这个主要用来保存name和value到hash中。
lowcase_index 这个是索引值。

和上面一样，我们跟着代码来看这些状态的意义。

首先来看sw_start状态，这个是起始状态：

```
case sw_start:
// 设置header开始指针。
            r->header_name_start = p;
            r->invalid_header = 0;

//通过第一个字符的值来判断下一个状态。
            switch (ch) {
//回车的话，说明没有header，因此设置状态为almost_done，然后期待最后的换行
            case CR:
                r->header_end = p;
                state = sw_header_almost_done;
                break;
            case LF:
//如果换行则直接进入header_done,也就是整个header解析完毕
                r->header_end = p;
                goto header_done;
            default:
```

```
//默认进入sw_name状态，进行name解析
                state = sw_name;
//这里做了一个表，来进行大小写转换
                c = lowcase[ch];

                if (c) {
//得到hash值，然后设置lowcase_header,后面我会解释这两个操作的原因。
                    hash = ngx_hash(0, c);
                    r->lowcase_header[0] = c;
                    i = 1;
                    break;
                }

                r->invalid_header = 1;

                break;

            }
            break;
```

然后是sw_name状态，这个状态进行解析name。

```
case sw_name:
//小写。
            c = lowcase[ch];
//开始计算hash，然后保存header name
            if (c) {
                hash = ngx_hash(hash, c);
                r->lowcase_header[i++] = c;
                i &= (NGX_HTTP_LC_HEADER_LEN - 1);
                break;
            }
```

```
//如果存在下划线，则通过传递进来的参数来判断是否允许下划线，比如fastcgi就允许。
            if (ch == '_') {
                if (allow_underscores) {
                    hash = ngx_hash(hash, ch);
                    r->lowcase_header[i++] = ch;
                    i &= (NGX_HTTP_LC_HEADER_LEN - 1);

                } else {
                    r->invalid_header = 1;
                }

                break;
            }

//如果是冒号，则进入value的处理，由于value有可能前面有空格，因此先处理这个。
            if (ch == ':') {
//设置header name的end。
                r->header_name_end = p;
                state = sw_space_before_value;
                break;
            }
//如果是回车换行则说明当前header解析已经结束，因此进入最终结束处理。
            if (ch == CR) {
//设置对应的值。
                r->header_name_end = p;
                r->header_start = p;
                r->header_end = p;
                state = sw_almost_done;
                break;
            }

            if (ch == LF) {
//设置对应的值，然后进入done
                r->header_name_end = p;
                r->header_start = p;
                r->header_end = p;
                goto done;
```

```
            }
.....................................................

            r->invalid_header = 1;

            break;
```

sw_space_before_value状态就不分析了，这里它主要是解析value有空格的情况，并且保存value的指针。

```
case sw_space_before_value:
            switch (ch) {
//跳过空格
            case ' ':
                break;
            case CR:
                r->header_start = p;
                r->header_end = p;
                state = sw_almost_done;
                break;
            case LF:
                r->header_start = p;
                r->header_end = p;
                goto done;
            default:
//设置header_start也就是value的开始指针。
                r->header_start = p;
                state = sw_value;
                break;
            }
            break;
```

我们主要来看sw_value状态，也就是解析value的状态。

```
case sw_value:
            switch (ch) {
//如果是空格则进入sw_space_after_value处理
            case ' ':
                r->header_end = p;
                state = sw_space_after_value;
                break;
//会车换行的话，说明header解析完毕进入done或者almost_done.也就是最终会返回NGX_OK
            case CR:
                r->header_end = p;
                state = sw_almost_done;
                break;
            case LF:
                r->header_end = p;
                goto done;
            }
            break;
```

最后来看两个结束状态

```
//当前的header解析完毕
case sw_almost_done:
            switch (ch) {
            case LF:
                goto done;
            case CR:
                break;
            default:
                return NGX_HTTP_PARSE_INVALID_HEADER;
            }
```

```
            break;
//整个header解析完毕
case sw_header_almost_done:
            switch (ch) {
            case LF:
                goto header_done;
            default:
                return NGX_HTTP_PARSE_INVALID_HEADER;
            }
        }
```

最后来看这几个标记：

首先是默认，也就是当遍历完buf后，header仍然没有结束的情况：

此时设置对应的hash值，以及保存当前状态，以及buf的位置

```
b->pos = p;
    r->state = state;
    r->header_hash = hash;
    r->lowcase_index = i;

    return NGX_AGAIN;
```

然后是done，也就是当前的header已经解析完毕，此时设置状态为start，以及buf位置为p+1.

```
done:

    b->pos = p + 1;
    r->state = sw_start;
    r->header_hash = hash;
    r->lowcase_index = i;
```

```
    return NGX_OK;
```

最后是header全部解析完毕，此时是得到了最后的回车换行，因此不需要hash值。

```
header_done:

    b->pos = p + 1;
    r->state = sw_start;

    return NGX_HTTP_PARSE_HEADER_DONE;
```

1.4 nginx的filter的处理

发表时间: 2010-04-13

随笔拿一个nginx的filter模块来看，gzip模块，来看它的初始化。

```
static ngx_http_output_header_filter_pt  ngx_http_next_header_filter;
static ngx_http_output_body_filter_pt    ngx_http_next_body_filter;

static ngx_int_t
ngx_http_gzip_filter_init(ngx_conf_t *cf)
{
    ngx_http_next_header_filter = ngx_http_top_header_filter;
    ngx_http_top_header_filter = ngx_http_gzip_header_filter;

    ngx_http_next_body_filter = ngx_http_top_body_filter;
    ngx_http_top_body_filter = ngx_http_gzip_body_filter;

    return NGX_OK;
}
```

这里nginx处理filter将所有的过滤器做成一个类似链表的东东，每次声明一个ngx_http_next_header_filter以及ngx_http_next_body_filter来保存当前的最前面的filter，然后再将自己的filter处理函数赋值给ngx_http_top_header_filter以及ngx_http_top_body_filter ，这样也就是说最后面初始化的filter反而是最早处理。

而在模块本身的filter处理函数中会调用ngx_http_next_header_filter，也就是当前filter插入前的那个最top上的filter处理函数。

然后我们来看nginx如何启动filter的调用。

先来看head_filter的调用：

```
ngx_int_t
ngx_http_send_header(ngx_http_request_t *r)
{
    if (r->err_status) {
        r->headers_out.status = r->err_status;
        r->headers_out.status_line.len = 0;
    }

    return ngx_http_top_header_filter(r);
}
```

可以看到当发送header的时候就是调用ngx_http_top_header_filter,nginx这里把status这些也作为一个filter模块来处理的。当启动ngx_http_top_header_filter之后所有的filter处理函数就会象链表一样被一个个的调用。

然后是body filter的调用，这个和header的类似，因此就不解释了。

```
ngx_int_t
ngx_http_output_filter(ngx_http_request_t *r, ngx_chain_t *in)
{
    ngx_int_t          rc;
    ngx_connection_t  *c;

    c = r->connection;

    ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
                   "http output filter \"%V?%V\"", &r->uri, &r->args);

//启动body filter。
    rc = ngx_http_top_body_filter(r, in);
.....................................................

    return rc;
}
```

这里还有一个问题，那就是最后一个ngx_http_top_header_filter和ngx_http_top_body_filter是什么呢？也就是第一个被插入的filter。

先来看filter被初始化的地方。这里filter的初始化是在ngx_http_block函数中：

```
for (m = 0; ngx_modules[m]; m++) {
        if (ngx_modules[m]->type != NGX_HTTP_MODULE) {
            continue;
        }

        module = ngx_modules[m]->ctx;

//如果存在postconfiguratio则调用初始化。
        if (module->postconfiguration) {
            if (module->postconfiguration(cf) != NGX_OK) {
                return NGX_CONF_ERROR;
            }
        }
    }
```

代码很简单就是遍历ngx_modules然后调用初始化函数，而我们这里要找第一个filter，也就是ngx_modules中的第一个bodyfilter和header filter。

来看objs/ngx_modules.c中的ngx_module的定义：

```
ngx_module_t *ngx_modules[] = {
 ..........................................................
    &ngx_http_write_filter_module,
    &ngx_http_header_filter_module,
.................................................................
    NULL
};
```

可以看到ngx_http_write_filter_module和ngx_http_header_filter_module分别是body filter和header filter的第一个初始化模块，也就是filter链中的最后一个模块。

接下来我们就来详细分析这两个模块，首先是ngx_http_write_filter_module模块。

这个模块的功能起始很简单，就是遍历chain，然后输出所有的数据，如果有设置flush的话刷新chain。

这里要注意ngx_http_request_t中有一个out的chain，这个chain保存的是上一次还没有被发完的buf，这样每次我们接收到新的chain的话，就需要将新的chain连接到老的out chain上，然后再发出去。

来看代码。

```
ngx_int_t
ngx_http_write_filter(ngx_http_request_t *r, ngx_chain_t *in)
```

第一个是request请求，第二个参数是输入的chain。

先来看初始化部分：

```
 off_t                      size, sent, nsent, limit;
    ngx_uint_t                 last, flush;
    ngx_msec_t                 delay;
    ngx_chain_t               *cl, *ln, **ll, *chain;
    ngx_connection_t          *c;
    ngx_http_core_loc_conf_t  *clcf;

//得到当前所属的连接
    c = r->connection;


    if (c->error) {
```

```
        return NGX_ERROR;
    }

    size = 0;
    flush = 0;
    last = 0;
//得到上次没有发送完毕的chain
    ll = &r->out;
```

然后接下来这部分是校验并统计out chain，也就是上次没有完成的chain buf。

```
 for (cl = r->out; cl; cl = cl->next) {
        ll = &cl->next;

#if 1
//如果有0长度的buf则返回错误。
        if (ngx_buf_size(cl->buf) == 0 && !ngx_buf_special(cl->buf)) {
 ..................................................................

            ngx_debug_point();
            return NGX_ERROR;
        }
#endif

//得到buf的大小
        size += ngx_buf_size(cl->buf);
//看当传输完毕后是否要刷新buf。
        if (cl->buf->flush || cl->buf->recycled) {
            flush = 1;
        }
//看是否是最后一个buf
        if (cl->buf->last_buf) {
            last = 1;
```

```
        }
    }
```

接下来这部分是用来链接新的chain到上面的out chain后面：

```
for (ln = in; ln; ln = ln->next) {
//
        cl = ngx_alloc_chain_link(r->pool);
        if (cl == NULL) {
            return NGX_ERROR;
        }

        cl->buf = ln->buf;
//前面的代码我们知道ll已经指向out chain的最后一个位置了，因此这里就是将新的chain链接到out chain的后面
        *ll = cl;
        ll = &cl->next;
#if 1
//校验buf
        if (ngx_buf_size(cl->buf) == 0 && !ngx_buf_special(cl->buf)) {

            ngx_debug_point();
            return NGX_ERROR;
        }
#endif

//计算大小
        size += ngx_buf_size(cl->buf);

//判断是否需要flush
        if (cl->buf->flush || cl->buf->recycled) {
            flush = 1;
        }

//判断是否是最后一个buf
```

```
        if (cl->buf->last_buf) {
            last = 1;
        }
    }
```

然后接下来的这段代码主要是对进行发送前buf的一些标记的处理。

在看代码之前先来解释下几个比较重要的标记。

第一个是ngx_http_core_module的conf的一个标记postpone_output(conf里面可以配置的)，这个表示延迟输出的阀，也就是说将要发送的字节数如果小于这个的话，并且还有另外几个条件的话(下面会解释),就会直接返回不发送当前的chain。

第二个是c->write->delayed,这个表示当前的连接的写必须要被delay了，也就是说现在不能发送了(原因下面会解释)，得等另外的地方取消了delayed才能发送，此时我们修改连接的buffered的标记，然后返回NGX_AGAIN.

第三个是c->buffered，因为有时buf并没有发完，因此我们有时就会设置buffed标记，而我们可能会在多个filter模块中被buffered，因此下面就是buffered的类型。

```
//这个并没有用到
#define NGX_HTTP_LOWLEVEL_BUFFERED         0xf0
//主要是这个，这个表示在最终的write filter中被buffered
#define NGX_HTTP_WRITE_BUFFERED            0x10
//判断是否有被设置
#define NGX_LOWLEVEL_BUFFERED  0x0f

//下面几个filter中被buffered
#define NGX_HTTP_GZIP_BUFFERED             0x20
#define NGX_HTTP_SSI_BUFFERED              0x01
#define NGX_HTTP_SUB_BUFFERED              0x02
#define NGX_HTTP_COPY_BUFFERED             0x04
```

然后我们来看第二个的意思，这个表示当前的chain已经被buffered了，

第四个是r->limit_rate，这个表示当前的request的发送限制速率，这个也是在nginx.conf中配置的，而一般就是通过这个值来设置c->write->delayed的。也就是说如果发送速率大于这个limit了的话，就设置delayed，然后这边的request就会延迟发送，下面我们的代码会看到nginx如何处理。

```
 clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);

//也就是说将要发送的字节数小于postpone_output并且不是最后一个buf，并且不需要刷新chain的话，就直接返回。
    if (!last && !flush && in && size < (off_t) clcf->postpone_output) {
        return NGX_OK;
    }

///如果设置了write的delayed,则设置标记。
    if (c->write->delayed) {
        c->buffered |= NGX_HTTP_WRITE_BUFFERED;
        return NGX_AGAIN;
    }

//如果size为0,并且没有设置buffered标记，则进入清理工作。
    if (size == 0 && !(c->buffered & NGX_LOWLEVEL_BUFFERED)) {
//如果是最后一个buf，则清理buffered标记然后清理out chain
        if (last) {
            r->out = NULL;
            c->buffered &= ~NGX_HTTP_WRITE_BUFFERED;

            return NGX_OK;
        }

//如果有设置flush的话，则会强行传输当前buf之前的所有buf，因此这里就需要清理out chain。
        if (flush) {
            do {
```

```
                r->out = r->out->next;
            } while (r->out);

//清理buf 标记
            c->buffered &= ~NGX_HTTP_WRITE_BUFFERED;

            return NGX_OK;
        }

        ngx_log_error(NGX_LOG_ALERT, c->log, 0,
                      "the http output chain is empty");

        ngx_debug_point();

        return NGX_ERROR;
    }

//如果有发送速率限制。
    if (r->limit_rate) {
//计算是否有超过速率限制
        limit = r->limit_rate * (ngx_time() - r->start_sec + 1)
                - (c->sent - clcf->limit_rate_after);
//如果有
        if (limit <= 0) {
//设置delayed标记
            c->write->delayed = 1;
//设置定时器
            ngx_add_timer(c->write,
                          (ngx_msec_t) (- limit * 1000 / r->limit_rate + 1));

//设置buffered。
            c->buffered |= NGX_HTTP_WRITE_BUFFERED;

            return NGX_AGAIN;
        }

    } else if (clcf->sendfile_max_chunk) {
```

```
//sendfile所用到的limit。
        limit = clcf->sendfile_max_chunk;

    } else {
        limit = 0;
    }

    sent = c->sent;
```

然后接下来这段就是发送buf，以及发送完的处理部分。这里要注意send_chain返回值为还没有发送完的chain，这个函数我后面的blog会详细的分析的。

```
//调用发送函数。
chain = c->send_chain(c, r->out, limit);

    ngx_log_debug1(NGX_LOG_DEBUG_HTTP, c->log, 0,
                   "http write filter %p", chain);

    if (chain == NGX_CHAIN_ERROR) {
        c->error = 1;
        return NGX_ERROR;
    }

//控制imit_rate,这个值一般是在nginx.conf中配置的。
    if (r->limit_rate) {

        nsent = c->sent;

        if (clcf->limit_rate_after) {

            sent -= clcf->limit_rate_after;
            if (sent < 0) {
                sent = 0;
```

```
            }

            nsent -= clcf->limit_rate_after;
            if (nsent < 0) {
                nsent = 0;
            }
        }

        delay = (ngx_msec_t) ((nsent - sent) * 1000 / r->limit_rate + 1);

        if (delay > 0) {
            c->write->delayed = 1;
            ngx_add_timer(c->write, delay);
        }

    } else if (c->write->ready
               && clcf->sendfile_max_chunk
               && (size_t) (c->sent - sent)
                      >= clcf->sendfile_max_chunk - 2 * ngx_pagesize)
    {
        c->write->delayed = 1;
        ngx_add_timer(c->write, 1);
    }

//开始遍历上一次还没有传输完毕的chain，如果这次没有传完的里面还有的话，就跳出循环，否则free这个chain
    for (cl = r->out; cl && cl != chain; /* void */) {
        ln = cl;
        cl = cl->next;
        ngx_free_chain(r->pool, ln);
    }

///out chain赋值
    r->out = chain;

//如果chain存在，则设置buffered并且返回again。
    if (chain) {
        c->buffered |= NGX_HTTP_WRITE_BUFFERED;
```

```
        return NGX_AGAIN;
    }

//否则清理buffered
    c->buffered &= ~NGX_HTTP_WRITE_BUFFERED;

//如果有其他的filter buffered并且postponed被设置了，则我们返回again，也就是还有buf要处理。
    if ((c->buffered & NGX_LOWLEVEL_BUFFERED) && r->postponed == NULL) {
        return NGX_AGAIN;
    }

//否则返回ok
    return NGX_OK;
```

然后我们来看ngx_http_header_filter_module模块，这个模块的处理函数是ngx_http_header_filter。这个函数最终还是会调用ngx_http_write_filter来将head输出。

这个函数主要就是处理http的头域，然后设置对应的reponse值，最终输出。

这里header filter比较简单，这里没有什么复杂的东西，主要就是设置一些status。然后拷贝，最后通过ngx_http_write_filter进行发送。

1.5 nginx中的output chain的处理(一)

发表时间: 2010-04-24

这里我们详细来看ngx_linux_sendfile_chain方法，这个函数也就是nginx的发送函数。

一般来说，我们最终都会调用这个函数来发送最终的数据，因此我们来着重分析这个函数，这里主要就是对buf的一些参数的理解。

来看函数原型：

```
ngx_chain_t *
ngx_linux_sendfile_chain(ngx_connection_t *c, ngx_chain_t *in, off_t limit)
```

第一个参数是当前的连接，第二个参数是所需要发送的chain，第三个参数是所能发送的最大值。

然后来看这里的几个重要的变量：

```
send 表示将要发送的buf已经已经发送的大小。
sent表示已经发送的buf的大小
prev_send 表示上一次发送的大小，也就是已经发送的buf的大小。
fprev 和prev-send类似，只不过是file类型的。

complete表示是否buf被完全发送了，也就是sent是否等于send - prev_send.

header表示需要是用writev来发送的buf。也就是only in memory的buf。

struct iovec  *iov, headers[NGX_HEADERS] 这个主要是用于sendfile和writev的参数，这里注意上面header数
```

然后我们来看初始化

```
wev = c->write;

    if (!wev->ready) {
        return in;
    }

    if (limit == 0 || limit > (off_t) (NGX_SENDFILE_LIMIT - ngx_pagesize)) {
        limit = NGX_SENDFILE_LIMIT - ngx_pagesize;
    }


    send = 0;
//设置header，也就是in memory的数组
    header.elts = headers;
    header.size = sizeof(struct iovec);
    header.nalloc = NGX_HEADERS;
    header.pool = c->pool;
```

这里nginx的处理核心思想就是合并内存连续并相邻的buf(不管是in memory还是in file)

下面这段代码就是处理in memory的部分，然后将buf放入对应的iovec数组。

```
//开始遍历
for (cl = in;
             cl && header.nelts < IOV_MAX && send < limit;
             cl = cl->next)
        {
            if (ngx_buf_special(cl->buf)) {
                continue;
            }
//如果不止是在buf中，这是因为有时in file的文件我们可能需要内存中也有拷贝，所以如果一个buf同时in memoey
            if (!ngx_buf_in_memory_only(cl->buf)) {
                break;
```

```
            }

//得到buf的大小
            size = cl->buf->last - cl->buf->pos;

//大于limit的话修改为size
            if (send + size > limit) {
                size = limit - send;
            }
//如果prev等于pos，则说明当前的buf的数据和前一个buf的数据是连续的。
            if (prev == cl->buf->pos) {
                iov->iov_len += (size_t) size;

            } else {
//否则说明是不同的buf，因此add一个iovc。
                iov = ngx_array_push(&header);
                if (iov == NULL) {
                    return NGX_CHAIN_ERROR;
                }

                iov->iov_base = (void *) cl->buf->pos;
                iov->iov_len = (size_t) size;
            }

//这里可以看到prev保存了当前buf的结尾。
            prev = cl->buf->pos + (size_t) size;
//更新发送的大小
            send += size;
        }
```

然后是in file的处理这里比较核心的一个判断就是fprev == cl->buf->file_pos，和上面的in memory类似，fprev保存的就是上一次处理的buf的尾部。这里如果这两个相等，那就说明当前的两个buf是连续的(文件连续).

ok.来看代码。

```
//可以看到如果header的大小不为0则说明前面有需要发送的buf，因此我们就跳过in file处理
if (header.nelts == 0 && cl && cl->buf->in_file && send < limit) {
//得到file
            file = cl->buf;

//开始合并。
            do {
//得到大小
                size = cl->buf->file_last - cl->buf->file_pos;

//如果太大则进行对齐处理。
                if (send + size > limit) {
                    size = limit - send;

                    aligned = (cl->buf->file_pos + size + ngx_pagesize - 1)
                              & ~((off_t) ngx_pagesize - 1);

                    if (aligned <= cl->buf->file_last) {
                        size = aligned - cl->buf->file_pos;
                    }
                }

//设置file_size.
                file_size += (size_t) size;
//设置需要发送的大小
                send += size;
//和上面的in memory处理一样就是保存这次的last
                fprev = cl->buf->file_pos + size;
                cl = cl->next;

            } while (cl
                     && cl->buf->in_file
                     && send < limit
                     && file->file->fd == cl->buf->file->fd
                     && fprev == cl->buf->file_pos);
        }
```

然后就是发送部分，这里in file使用sendfile,in memory使用writev.这里处理比较简单，就是发送然后判断发送的大小

```
if (file) {
#if 1
            if (file_size == 0) {
                ngx_debug_point();
                return NGX_CHAIN_ERROR;
            }
#endif
#if (NGX_HAVE_SENDFILE64)
            offset = file->file_pos;
#else
            offset = (int32_t) file->file_pos;
#endif

//发送数据
            rc = sendfile(c->fd, file->file->fd, &offset, file_size);
..................................................
//得到发送的字节数
            sent = rc > 0 ? rc : 0;

        } else {
            rc = writev(c->fd, header.elts, header.nelts);
.....................................................................

            sent = rc > 0 ? rc : 0;
}
```

接下来这部分就是更新标记的部分，主要是buf的标记。

这里要注意一个地方，那就是ngx_buf_size部分，这个宏很简单就是判断buf是不是在memory中，如果是的

话，就用pos和last计算，否则认为是在file中。

可是这里就有个问题了，如果一个buf本来是在file中的，我们由于某种原因，在内存中也有一份拷贝，可是我们并没有修改内存中的副本，于是如果我们还需要切割这个buf，这个时候，如果last和pos也就是buf对应的指针没有设置正确的话，这里就会出现问题了。

这里我觉得应该还有个标记，那就是如果内存中的副本我只是只读的话，发送的时候不应该算它在memory中。

```
//如果send - prev_send == sent则说明该发送的都发完了。
if (send - prev_send == sent) {
            complete = 1;
        }
//更新congnect的sent域。
        c->sent += sent;

//开始重新遍历chain，这里是为了防止没有发送完全的情况，此时我们就需要切割buf了。
        for (cl = in; cl; cl = cl->next) {

            if (ngx_buf_special(cl->buf)) {
                continue;
            }

            if (sent == 0) {
                break;
            }
//得到buf size
            size = ngx_buf_size(cl->buf);

//如果大于当前的size，则说明这个buf的数据已经被完全发送完毕了。，因此更新它的域。
            if (sent >= size){
//更新sent域
                sent -= size;
//如果在内存则更新pos
                if (ngx_buf_in_memory(cl->buf)) {
                    cl->buf->pos = cl->buf->last;
```

```
                }
//如果在file
                if (cl->buf->in_file) {
                    cl->buf->file_pos = cl->buf->file_last;
                }

                continue;
            }

//到这里说明当前的buf只有一部分被发送出去了，因此这里我们只需要修改指针。以便于下次发送。
            if (ngx_buf_in_memory(cl->buf)) {
                cl->buf->pos += (size_t) sent;
            }
//同上。
            if (cl->buf->in_file) {
                cl->buf->file_pos += sent;
            }

            break;
        }
```

最后一部分就是一些是否退出循环的操作。这里要注意，nginx中如果发送未完全的话，将会直接返回的，返回的就是没有发送完毕的chain，它的buf也已经被更新。这是因为nginx是单线程的，不能有任何意义的空跑和阻塞，因此当complete为0,nginx就认为是系统负载过大，此时直接返回，然后处理其他的事情，等待和下次的chain一起发送。

```
if (eintr) {
            continue;
        }
//如果未完成，则返回。
        if (!complete) {
            wev->ready = 0;
            return cl;
```

```
        }


        if (send >= limit || cl == NULL) {
            return cl;
        }
//更新in，也就是开始处理下一个chain
        in = cl;
```

1.6 nginx中锁的设计以及惊群的处理

发表时间: 2010-05-03

nginx中使用的锁是自己来实现的，这里锁的实现分为两种情况，一种是支持原子操作的情况，也就是由NGX_HAVE_ATOMIC_OPS这个宏来进行控制的，一种是不支持原子操作，这是是使用文件锁来实现。

首先我们要知道在用户空间进程间锁实现的原理，起始原理很简单，就是能弄一个让所有进程共享的东西，比如mmap的内存，比如文件，然后通过这个东西来控制进程的互斥。

说起来锁很简单，就是共享一个变量，然后通过设置这个变量来控制进程的行为。

我们先来看核心的数据结构，也就是说用来控制进程的互斥的东西。

这个数据结构可以看到和我上面讲得一样，那就是通过宏来分成两种。

1 如果支持原子操作，则我们可以直接使用mmap，然后lock就保存mmap的内存区域的地址

2 如果不支持原子操作，则我们使用文件锁来实现，这里fd表示进程间共享的文件句柄，name表示文件名。

```
typedef struct {
#if (NGX_HAVE_ATOMIC_OPS)
    ngx_atomic_t  *lock;
#else
    ngx_fd_t       fd;
    u_char        *name;
#endif
} ngx_shmtx_t;
```

接着来看代码，先来看支持原子操作的情况下的实现方式。这里要注意下，下面的函数基本都会有两个实现，一个是支持原子操作，一个是不支持的，我这里全部都是分开来分析的。

先来看初始化，初始化代码在ngx_event_module_init中。

下面这段代码是设置将要设置的共享区域的大小，这里cl的大小最好是要大于或者等于cache line。
通过代码可以看到这里将会有3个区域被所有进程共享，其中我们的锁将会用到的是第一个。

```
size_t                size, cl;
    cl = 128;
//可以看到三个区域。
    size = cl            /* ngx_accept_mutex */
           + cl          /* ngx_connection_counter */
           + cl;         /* ngx_temp_number */
```

下面这段代码是初始化对应的共享内存区域。然后保存对应的互斥体指针。

```
//这个是一个全局变量，保存的是共享区域的指针。
ngx_atomic_t         *ngx_accept_mutex_ptr;
//这个就是我们上面介绍的互斥体。
ngx_shmtx_t           ngx_accept_mutex;


ngx_shm_t             shm;
//开始初始化
shm.size = size;
    shm.name.len = sizeof("nginx_shared_zone");
    shm.name.data = (u_char *) "nginx_shared_zone";
    shm.log = cycle->log;
//分配对应的内存，使用mmap或者shm之类的。
    if (ngx_shm_alloc(&shm) != NGX_OK) {
        return NGX_ERROR;
    }

    shared = shm.addr;

    ngx_accept_mutex_ptr = (ngx_atomic_t *) shared;
//初始化互斥体。
    if (ngx_shmtx_create(&ngx_accept_mutex, shared, cycle->lock_file.data)
        != NGX_OK)
    {
```

```
        return NGX_ERROR;
    }
```

下面我们来看ngx_shmtx_create的实现。
可以看到如果支持原子操作的话，非常简单，就是将共享内存的地址付给loc这个域。

```
ngx_int_t
ngx_shmtx_create(ngx_shmtx_t *mtx, void *addr, u_char *name)
{
    mtx->lock = addr;

    return NGX_OK;
}
```

然后来看nginx中如何来获得锁，以及释放锁。

我们先来看获得锁。

这里nginx分为两个函数，一个是trylock，它是非阻塞的，也就是说它会尝试的获得锁，如果没有获得的话，它会直接返回错误。

而第二个是lock，它也会尝试获得锁，而当没有获得他不会立即返回，而是开始进入循环然后不停的去获得锁，知道获得。不过nginx这里还有用到一个技巧，就是每次都会让当前的进程放到cpu的运行队列的最后一位，也就是自动放弃cpu。

先来看trylock

这个很简单，首先判断lock是否为0,为0的话表示可以获得锁，因此我们就调用ngx_atomic_cmp_set去获得锁，如果获得成功就会返回1,负责为0.

```
static ngx_inline ngx_uint_t
ngx_shmtx_trylock(ngx_shmtx_t *mtx)
```

```
{
    return (*mtx->lock == 0 && ngx_atomic_cmp_set(mtx->lock, 0, ngx_pid));
}
```

接下来详细描述下ngx_atomic_cmp_set，这里这个操作是一个原子操作，这是因为由于我们要进行比较+赋值两个操作，如果不是原子操作的话，有可能在比较之后被其他进程所抢占，此时再赋值的话就会有问题了，因此这里就必须是一个原子操作。

我们来看这个函数的实现，如果系统库不支持这个指令的话，nginx自己还用汇编实现了一个，其实实现也很简单，比如x86的话有一个cmpxchgl的指令，就是做这个的。

先来看如果系统库支持的情况，此时直接调用OSAtomicCompareAndSwap32Barrier。

```
#define ngx_atomic_cmp_set(lock, old, new)                                   \
    OSAtomicCompareAndSwap32Barrier(old, new, (int32_t *) lock)
```

来看函数的原型：

```
OSAtomicCompareAndSwap32Barrier(old, new, addr)
```

然后这个函数翻译成伪码的话就是这个：

```
f (*addr == oldvalue) {
    *addr = newvalue;
    return 1;
} else {
    return 0;
}
```

这个代码就不解释了，很浅显易懂。

因此上面的trylock的代码：

```
ngx_atomic_cmp_set(mtx->lock, 0, ngx_pid)
```

的意思就是如果lock的值是0的话，就把lock的值修改为当前的进程id，否则返回失败。

然后来看这个的汇编实现，这里nginx实现了多个平台的比如x86,sparc,ppc.
我们来看x86的：

```
static ngx_inline ngx_atomic_uint_t
ngx_atomic_cmp_set(ngx_atomic_t *lock, ngx_atomic_uint_t old,
    ngx_atomic_uint_t set)
{
    u_char  res;

    __asm__ volatile (

         NGX_SMP_LOCK
    "    cmpxchgl  %3, %1;   "
    "    sete      %0;       "

    : "=a" (res) : "m" (*lock), "a" (old), "r" (set) : "cc", "memory");

    return res;
}
```

具体的这些指令和锁可以去看intel的相关手册。

接下来来看lock的实现，lock最终会调用ngx_spinlock，因此下面我要主要来分析这个函数。

```
#define ngx_shmtx_lock(mtx)   ngx_spinlock((mtx)->lock, ngx_pid, 1024)
```

我们来看spinklock，必须支持原子指令，才会有这个函数，这里nginx采用宏来控制的.

这里和trylock的处理差不多，都是利用原子指令来实现的，只不过这里如果无法获得锁，则会继续等待。

我们来看代码的实现：

```
void
ngx_spinlock(ngx_atomic_t *lock, ngx_atomic_int_t value, ngx_uint_t spin)
{

#if (NGX_HAVE_ATOMIC_OPS)

    ngx_uint_t  i, n;

    for ( ;; ) {

//如果lock为0,则说明没有进程持有锁，因此设置lock为value(为当前进程id),然后返回。
        if (*lock == 0 && ngx_atomic_cmp_set(lock, 0, value)) {
            return;
        }

//如果cpu个数大于1(也就是多核),则进入spin-wait loop阶段。
        if (ngx_ncpu > 1) {

//开始进入循环。
            for (n = 1; n < spin; n <<= 1) {

//下面这段就是纯粹的spin-loop wait。
                for (i = 0; i < n; i++) {
//这个函数其实就是执行"PAUSE"指令，接下来会解释这个指令。
                    ngx_cpu_pause();
                }
//然后重新获取锁，如果获得则直接返回。
                if (*lock == 0 && ngx_atomic_cmp_set(lock, 0, value)) {
                    return;
```

```
                }
            }
        }

//这个函数调用的是sched_yield，它会强迫当前运行的进程放弃占有处理器。
        ngx_sched_yield();
    }

#else

#if (NGX_THREADS)

#error ngx_spinlock() or ngx_atomic_cmp_set() are not defined !

#endif

#endif

}
```

通过上面的代码可以看到spin lock实现的很简单，就是一个如果无法获得锁，就进入忙等的过程，不过这里nginx还多加了一个处理，就是如果忙等太长，就放弃cpu，直到下次任务再次占有cpu。

接下来来看下PAUSE指令，这条指令主要的功能就是告诉cpu，我现在是一个spin-wait loop,然后cpu就不会因为害怕循环退出时，内存的乱序而需要处理，所引起的效率损失问题。

下面就是intel手册的解释：

引用

> Improves the performance of spin-wait loops. When executing a “spin-wait loop,” a Pentium 4 or Intel Xeon processor suffers a severe performance penalty when exiting the loop because it detects a possible memory order violation. The PAUSE instruction provides a hint to the processor that the code sequence is a spin-wait loop. The

> processor uses this hint to avoid the memory order violation in most situations,
> which greatly improves processor performance. For this reason, it is recommended
> that a PAUSE instruction be placed in all spin-wait loops.

内核的spin lock也有用到这条指令的。

接下来就是unlokck。unlock比较简单，就是和当前进程id比较，如果相等，就把lock改为0,说明放弃这个锁。

```
#define ngx_shmtx_unlock(mtx) (void) ngx_atomic_cmp_set((mtx)->lock, ngx_pid, 0)
```

然后就是不支持原子操作的情况，此时使用文件锁来实现的，这里就不介绍这种实现了，基本原来和上面的差不多，想要了解的，可以去看nginx的相关代码。

接下来我们来看nginx如何利用lock来控制子进程的负载均衡以及惊群。

先来大概解释下这两个概念。

负载均衡是为了解决有可能一个进程处理了多个连接，因此就需要让多个进程更平均的处理连接。

惊群也就是当我们多个进程阻塞在epoll这类调用的时候，当有数据可读的时候，多个进程会被同时唤醒，此时如果去accept的话，只能有一个进程accept到句柄。

在看代码之前，我们先来看ngx_use_accept_mutex这个变量，如果有这个变量，说明nginx有必要使用accept互斥体，这个变量的初始化在ngx_event_process_init中。

这里还有两个变量，一个是ngx_accept_mutex_held，一个是ngx_accept_mutex_delay，其中前一个表示当前是否已经持有锁，后一个表示，当获得锁失败后，再次去请求锁的间隔时间，这个时间可以看到可以在配置文件中设置的。

```
//如果使用了master worker，并且worker个数大于1,并且配置文件里面有设置使用accept_mutex.的话，设置ngx_
 if (ccf->master && ccf->worker_processes > 1 && ecf->accept_mutex) {
        ngx_use_accept_mutex = 1;
```

```
//下面这两个变量后面会解释。
        ngx_accept_mutex_held = 0;
        ngx_accept_mutex_delay = ecf->accept_mutex_delay;
    } else {
        ngx_use_accept_mutex = 0;
    }
```

这里还有一个变量是ngx_accept_disabled，这个变量是一个阈值，如果大于0,说明当前的进程处理的连接过多。
下面就是这个值的初始化，可以看到初始值是全部连接的7/8(注意是负值0.

```
ngx_accept_disabled = ngx_cycle->connection_n / 8
                              - ngx_cycle->free_connection_n;
```

然后来看ngx_process_events_and_timers中的处理。

```
//如果有使用mutex，则才会进行处理。
if (ngx_use_accept_mutex) {
//如果大于0,则跳过下面的锁的处理，并减一。
        if (ngx_accept_disabled > 0) {
            ngx_accept_disabled--;

        } else {
//试着获得锁，如果出错则返回。
            if (ngx_trylock_accept_mutex(cycle) == NGX_ERROR) {
                return;
            }
//如果ngx_accept_mutex_held为1,则说明已经获得锁，此时设置flag，这个flag后面会解释。
            if (ngx_accept_mutex_held) {
```

```
                flags |= NGX_POST_EVENTS;


            } else {
//否则，设置timer，也就是定时器。接下来会解释这段。
                if (timer == NGX_TIMER_INFINITE
                    || timer > ngx_accept_mutex_delay)
                {
                    timer = ngx_accept_mutex_delay;
                }
            }
        }
    }
```

然后先来看NGX_POST_EVENTS标记，设置了这个标记就说明当socket有数据被唤醒时，我们并不会马上accept或者说读取，而是将这个事件保存起来，然后当我们释放锁之后，才会进行accept或者读取这个句柄。

```
//如果ngx_posted_accept_events不为NULL，则说明有accept event需要nginx处理。
if (ngx_posted_accept_events) {
        ngx_event_process_posted(cycle, &ngx_posted_accept_events);
    }
```

而如果没有设置NGX_POST_EVENTS标记的话，nginx会立即accept或者读取句柄。

然后是定时器，这里如果nginx没有获得锁，并不会马上再去获得锁，而是设置定时器，然后在epoll休眠(如果没有其他的东西唤醒).此时如果有连接到达，当前休眠进程会被提前唤醒，然后立即accept。否则，休眠ngx_accept_mutex_delay时间，然后继续try lock.

最后是核心的一个函数，那就是ngx_trylock_accept_mutex。这个函数用来尝试获得accept mutex.

```
ngx_int_t
ngx_trylock_accept_mutex(ngx_cycle_t *cycle)
```

```
{
//尝试获得锁
    if (ngx_shmtx_trylock(&ngx_accept_mutex)) {
//如果本来已经获得锁，则直接返回Ok
        if (ngx_accept_mutex_held
            && ngx_accept_events == 0
            && !(ngx_event_flags & NGX_USE_RTSIG_EVENT))
        {
            return NGX_OK;
        }

//到达这里，说明重新获得锁成功，因此需要打开被关闭的listening句柄。
        if (ngx_enable_accept_events(cycle) == NGX_ERROR) {
            ngx_shmtx_unlock(&ngx_accept_mutex);
            return NGX_ERROR;
        }

        ngx_accept_events = 0;
//设置获得锁的标记。
        ngx_accept_mutex_held = 1;

        return NGX_OK;
    }

//如果我们前面已经获得了锁，然后这次获得锁失败，则说明当前的listen句柄已经被其他的进程锁监听，因此此时需
    if (ngx_accept_mutex_held) {
        if (ngx_disable_accept_events(cycle) == NGX_ERROR) {
            return NGX_ERROR;
        }
//设置锁的持有为0.
        ngx_accept_mutex_held = 0;
    }

    return NGX_OK;
}
```

这里可以看到大部分情况下，每次只会有一个进程在监听listen句柄，而只有当ngx_accept_disabled大于0的情况下，才会出现一定程度的惊群。

而nginx中，由于锁的控制(以及获得锁的定时器)，每个进程都能相对公平的accept句柄，也就是比较好的解决了子进程负载均衡。

1.7 nginx中的output chain的处理(二)

发表时间: 2010-05-09

接着上次的分析继续，这次我们来看filter链中最关键的一个模块，那就是ngx_http_copy_filter_module模块，这个filter主要是用来将一些需要复制的buf(文件或者内存)重新复制一份然后发送给剩余的body filter，这里有个很重要的部分，那就是在这里nginx的剩余的body filter有可能会被调用多次，这个接下来我会一一阐述的。先来看它的初始化函数：

```
static ngx_int_t
ngx_http_copy_filter_init(ngx_conf_t *cf)
{
    ngx_http_next_filter = ngx_http_top_body_filter;
    ngx_http_top_body_filter = ngx_http_copy_filter;

    return NGX_OK;
}
```

可以看到，它只有body filter，而没有header filter，也就是说只有body filter才会使用这个filter。

然后这个模块对应也有一个命令，那就是output_buffers，这个命令保存值在它的conf的bufs中：

```
typedef struct {
    ngx_bufs_t  bufs;
} ngx_http_copy_filter_conf_t;
```

这里要知道在nginx的配置文件中所有的bufs的格式都是一样，个数+每个的大小。这个值我们接下来分析filter代码的时候会再次看到。

然后来看对应的merge方法，来看这个bufs的默认值是多少。

```
static char *
ngx_http_copy_filter_merge_conf(ngx_conf_t *cf, void *parent, void *child)
{
    ngx_http_copy_filter_conf_t *prev = parent;
    ngx_http_copy_filter_conf_t *conf = child;

//默认是1个buf，大小为32768字节
    ngx_conf_merge_bufs_value(conf->bufs, prev->bufs, 1, 32768);

    return NULL;
}
```

由于copy filter没有header filter，因此它的context的初始化也是放在body filter中的，而它的ctx就是ngx_output_chain_ctx_t，为什么名字是output_chain呢，这是因为copy filter的主要逻辑的处理都是放在ngx_output_chain中的，这个模块我们可以看到它是保存在core目录下的，而不是属于http目录的。

接下来我们就来看这个context的结构。

```
typedef struct {
//保存临时的buf
    ngx_buf_t                    *buf;
//保存了将要发送的chain
    ngx_chain_t                  *in;
//保存了已经发送完毕的chain，以便于重复利用
    ngx_chain_t                  *free;
//保存了还未发送的chain
    ngx_chain_t                  *busy;

//sendfile标记
    unsigned                      sendfile:1;
//directio标记
    unsigned                      directio:1;
#if (NGX_HAVE_ALIGNED_DIRECTIO)
    unsigned                      unaligned:1;
```

```
#endif
//是否需要在内存中保存一份(使用sendfile的话，内存中没有文件的拷贝的，而我们有时需要处理文件，此时就需要
    unsigned                         need_in_memory:1;
//是否存在的buf复制一份，这里不管是存在在内存还是文件，后面会看到这两个标记的区别。
    unsigned                         need_in_temp:1;


    ngx_pool_t                      *pool;
//已经allocated的大小
    ngx_int_t                        allocated;
//对应的bufs的大小，这个值就是我们loc conf中设置的bufs
    ngx_bufs_t                       bufs;
//表示现在处于那个模块（因为upstream也会调用output_chain）
    ngx_buf_tag_t                    tag;

//这个值一般是ngx_http_next_filter,也就是继续调用filter链
    ngx_output_chain_filter_pt       output_filter;
//当前filter的上下文，这里也是由于upstream也会调用output_chain
    void                            *filter_ctx;
} ngx_output_chain_ctx_t;
```

接下来我们来看具体函数的实现，就能更好的理解context中的这些域的意思。

来看copy_filter的body filter。

```
static ngx_int_t
ngx_http_copy_filter(ngx_http_request_t *r, ngx_chain_t *in)
{
    ngx_int_t                     rc;
    ngx_connection_t             *c;
    ngx_output_chain_ctx_t       *ctx;
    ngx_http_copy_filter_conf_t  *conf;


    c = r->connection;
```

```
//获取ctx
    ctx = ngx_http_get_module_ctx(r, ngx_http_copy_filter_module);

//如果为空，则说明需要初始化ctx
    if (ctx == NULL) {
        conf = ngx_http_get_module_loc_conf(r, ngx_http_copy_filter_module);

        ctx = ngx_pcalloc(r->pool, sizeof(ngx_output_chain_ctx_t));
        if (ctx == NULL) {
            return NGX_ERROR;
        }

        ngx_http_set_ctx(r, ctx, ngx_http_copy_filter_module);

//设置对应的域
        ctx->sendfile = c->sendfile;
//可以看到如果我们给request设置filter_need_in_memory的话，ctx的这个域就会被设置
        ctx->need_in_memory = r->main_filter_need_in_memory
                              || r->filter_need_in_memory;
//和上面类似
        ctx->need_in_temp = r->filter_need_temporary;

        ctx->pool = r->pool;
        ctx->bufs = conf->bufs;
        ctx->tag = (ngx_buf_tag_t) &ngx_http_copy_filter_module;
//可以看到output_filter就是body filter的next
        ctx->output_filter = (ngx_output_chain_filter_pt) ngx_http_next_filter;
//此时filter ctx为当前的请求
        ctx->filter_ctx = r;

        r->request_output = 1;
    }
//最关键的函数，下面会详细分析。
    rc = ngx_output_chain(ctx, in);
.....................................................
```

```
    return rc;
}
```

然后就是ngx_output_chain这个函数了，这里nginx filter的主要逻辑都在这个函数里面。下面就是这个函数的原型。

```
ngx_int_t
ngx_output_chain(ngx_output_chain_ctx_t *ctx, ngx_chain_t *in)
```

然后我们来分段看它的代码，下面这段代码可以说是一个short path，也就是说我们能直接确定所有的in chain都不需要复制的时候，我们就可以直接调用output_filter来交给剩下的filter去处理。

```
 if (ctx->in == NULL && ctx->busy == NULL) {

//下面的注释解释的很详细
        /*
         * the short path for the case when the ctx->in and ctx->busy chains
         * are empty, the incoming chain is empty too or has the single buf
         * that does not require the copy
         */

        if (in == NULL) {
            return ctx->output_filter(ctx->filter_ctx, in);
        }

//这里说明只有一个chain，并且它的buf不需要复制
        if (in->next == NULL
#if (NGX_SENDFILE_LIMIT)
            && !(in->buf->in_file && in->buf->file_last > NGX_SENDFILE_LIMIT)
#endif
```

```
            && ngx_output_chain_as_is(ctx, in->buf))
        {
            return ctx->output_filter(ctx->filter_ctx, in);
        }
    }
```

上面我们看到了一个函数 ngx_output_chain_as_is，这个函数很关键，下面还会再次被调用，这个函数主要用来判断是否需要复制buf。返回1,表示不需要拷贝，否则为需要拷贝

```
static ngx_inline ngx_int_t
ngx_output_chain_as_is(ngx_output_chain_ctx_t *ctx, ngx_buf_t *buf)
{
    ngx_uint_t  sendfile;

//是否为specialbuf，是的话返回1,也就是不用拷贝
    if (ngx_buf_special(buf)) {
        return 1;
    }

//如果buf在文件中，并且使用了directio的话，需要拷贝buf
    if (buf->in_file && buf->file->directio) {
        return 0;
    }

//sendfile标记
    sendfile = ctx->sendfile;

#if (NGX_SENDFILE_LIMIT)
//如果pos大于sendfile的限制，设置标记为0
    if (buf->in_file && buf->file_pos >= NGX_SENDFILE_LIMIT) {
        sendfile = 0;
    }

#endif
```

```
    if (!sendfile) {
//此时如果buf不在内存中，则我们就需要复制到内存一份。
        if (!ngx_buf_in_memory(buf)) {
            return 0;
        }
//否则设置in_file为0.
        buf->in_file = 0;
    }

//如果需要内存中有一份拷贝，而并不在内存中，此时返回0，表示需要拷贝
    if (ctx->need_in_memory && !ngx_buf_in_memory(buf)) {
        return 0;
    }

//如果需要内存中有拷贝,并且存在于内存中或者mmap中，则返回0.
    if (ctx->need_in_temp && (buf->memory || buf->mmap)) {
        return 0;
    }

    return 1;
}
```

上面有两个标记要注意，一个是need_in_memory ，这个主要是用于当我们使用sendfile的时候，nginx并不会将请求文件拷贝到内存中，而有时我们需要操作文件的内容，此时我们就需要设置这个标记(设置方法前面初始化有介绍).然后我们在body filter就能操作内容了。

第二个是need_in_temp，这个主要是用于把本来就存在于内存中的buf复制一份拷贝出来，这里有用到的模块有charset,也就是编解码 filter.

然后接下来这段是复制in chain到ctx->in的结尾.它是通过调用ngx_output_chain_add_copy来进行add copy的，这个函数比较简单，这里就不分析了，不过只有一个要注意的，那就是如果buf是存在于文件中，并且file_pos超过了sendfile limit，此时就会切割buf为两个buf，然后保存在两个chain中，最终连接起来.

```
if (in) {
//复制到ctx->in中.
        if (ngx_output_chain_add_copy(ctx->pool, &ctx->in, in) == NGX_ERROR) {
            return NGX_ERROR;
        }
    }
```

然后就是主要的逻辑处理阶段。这里nginx做的非常巧妙也非常复杂，首先是chain的重用，然后是buf的重用。

先来看chain的重用。关键的几个结构以及域，ctx的free,busy以及ctx->pool的chain域。
其中每次发送没有发完的chain就放到busy中，而已经发送完毕的就放到free中，而最后会调用ngx_free_chain来将free的chain放入到pool->chain中,而在ngx_alloc_chain_link中，如果pool->chain中存在chain的话，就不用malloc了，而是直接返回pool->chain,我们来看相关的代码。

```
//链接cl到pool->chain中
#define ngx_free_chain(pool, cl)                                             \
    cl->next = pool->chain;                                                  \
    pool->chain = cl

ngx_chain_t *
ngx_alloc_chain_link(ngx_pool_t *pool)
{
    ngx_chain_t  *cl;

    cl = pool->chain;
//如果cl存在，则直接返回cl
    if (cl) {
        pool->chain = cl->next;
        return cl;
    }
//否则才会malloc chain
    cl = ngx_palloc(pool, sizeof(ngx_chain_t));
    if (cl == NULL) {
```

```
        return NULL;
    }

    return cl;
}
```

然后是buf的重用，严格意义上来说buf的重用是从free中的chain中取得的，当free中的buf被重用，则这个buf对应的chain就会被链接到ctx->pool中，从而这个chain就会被重用.

也就是说buf的重用是第一被考虑的，只有当这个chain的buf确定不需要被重用(或者说已经被重用)的时候，chain才会被链接到ctx->pool中被重用。

还有一个就是ctx的allocated域，这个域表示了当前的上下文中已经分配了多少个buf，blog一开始我们有提到有个output_buffer命令用来设置output的buf大小以及buf的个数。而allocated如果比output_buffer大的话，我们就需要先发送完已经存在的buf，然后才能再次重新分配buf。

来看代码，上面所说的重用以及buf的控制，代码里面都可以看的比较清晰。这里代码我们分段来看，下面这段主要是拷贝buf前所做的一些工作，比如判断是否拷贝，以及给buf分贝内存等。

```
//out为我们最终需要传输的chain，也就是交给剩下的filter处理的chain
 out = NULL;
//last_out为out的最后一个chain
    last_out = &out;
    last = NGX_NONE;

for ( ;; ) {

//开始遍历chain
        while (ctx->in) {

//取得当前chain的buf大小
            bsize = ngx_buf_size(ctx->in->buf);

//跳过bsize为0的buf
```

```
            if (bsize == 0 && !ngx_buf_special(ctx->in->buf)) {
                ngx_debug_point();

                ctx->in = ctx->in->next;

                continue;
            }

//判断是否需要复制buf
            if (ngx_output_chain_as_is(ctx, ctx->in->buf)) {

                /* move the chain link to the output chain */
//如果不需要复制，则直接链接chain到out，然后继续循环
                cl = ctx->in;
                ctx->in = cl->next;

                *last_out = cl;
                last_out = &cl->next;
                cl->next = NULL;

                continue;
            }

//到达这里，说明我们需要拷贝buf，这里buf最终都会被拷贝进ctx->buf中，因此这里先判断ctx->buf是否为空
            if (ctx->buf == NULL) {

//如果为空，则取得buf，这里要注意，一般来说如果没有开启directio的话，这个函数都会返回NGX_DECLINED的(具
                rc = ngx_output_chain_align_file_buf(ctx, bsize);

                if (rc == NGX_ERROR) {
                    return NGX_ERROR;
                }

//大部分情况下，都会落入这个分支
                if (rc != NGX_OK) {

//准备分配buf，首先在free中寻找可以重用的buf
```

```
                    if (ctx->free) {

                        /* get the free buf */
//得到free buf
                        cl = ctx->free;
                        ctx->buf = cl->buf;
                        ctx->free = cl->next;
//将要重用的chain链接到ctx->poll中，以便于chain的重用.
                        ngx_free_chain(ctx->pool, cl);

                    } else if (out || ctx->allocated == ctx->bufs.num) {
//如果已经等于buf的个数限制，则跳出循环，发送已经存在的buf.这里可以看到如果out存在的话，nginx会跳出循环
                        break;

                    } else if (ngx_output_chain_get_buf(ctx, bsize) != NGX_OK) {
//这个函数也比较关键，它用来取得buf.我们接下来会详细看这个函数
                        return NGX_ERROR;
                    }
                }
            }
.....................................................
    }
```

上面的代码分析的时候有个很关键的函数，那就是ngx_output_chain_get_buf,这个函数是当没有可重用的buf的时候，用来分配buf的。

这里只有一个要注意的，那就是如果当前的buf是位于最后一个chain的话，会有特殊处理。这里特殊处理有两个地方，一个是buf的recycled域，一个是将要分配的buf的大小。

先来说recycled域，这个域表示我们当前的buf是需要被回收的。而我们知道nginx一般情况下(比如非last buf)是会缓存一部分buf，然后再发送的(默认是1460字节)，而设置了recycled的话，我们就不会让它缓存buf，也就是尽量发送出去，然后以供我们回收使用。

因此如果是最后一个buf的话，一般来说我们是不需要设置recycled域的，否则的话，需要设置recycled域。因为不是最后一个buf的话，我们可能还会需要重用一些buf，而buf只有被发送出去的话，我们才能重用。

然后就是size的大小。这里会有两个大小，一个是我们需要复制的buf的大小，一个是nginx.conf中设置的size。如果不是最后一个buf，则我们只需要分配我们设置的buf的size大小就行了。如果是最后一个buf，则就处理不太一样，下面的代码会看到。

```
static ngx_int_t
ngx_output_chain_get_buf(ngx_output_chain_ctx_t *ctx, off_t bsize)
{
    size_t       size;
    ngx_buf_t   *b, *in;
    ngx_uint_t   recycled;

    in = ctx->in->buf;
//可以看到这里分配的buf，每个的大小都是我们在nginx.conf中设置的size
    size = ctx->bufs.size;
//默认有设置recycled域.
    recycled = 1;
//如果当前的buf是属于最后一个chain的时候。这里我们要特殊处理。
    if (in->last_in_chain) {
//这边注释很详细，我就不解释了.
        if (bsize < (off_t) size) {

            /*
             * allocate a small temp buf for a small last buf
             * or its small last part
             */
            size = (size_t) bsize;
            recycled = 0;

        } else if (!ctx->directio
                   && ctx->bufs.num == 1
                   && (bsize < (off_t) (size + size / 4)))
        {
            /*
             * allocate a temp buf that equals to a last buf,
             * if there is no directio, the last buf size is lesser
             * than 1.25 of bufs.size and the temp buf is single
```

```
             */

            size = (size_t) bsize;
            recycled = 0;
        }
    }
//开始分配buf内存.
    b = ngx_calloc_buf(ctx->pool);
    if (b == NULL) {
        return NGX_ERROR;
    }

    if (ctx->directio) {
//directio需要对齐

        b->start = ngx_pmemalign(ctx->pool, size, NGX_DIRECTIO_BLOCK);
        if (b->start == NULL) {
            return NGX_ERROR;
        }

    } else {
//大部分情况会走到这里.
        b->start = ngx_palloc(ctx->pool, size);
        if (b->start == NULL) {
            return NGX_ERROR;
        }
    }

    b->pos = b->start;
    b->last = b->start;
    b->end = b->last + size;
//设置temporary.
    b->temporary = 1;
    b->tag = ctx->tag;
    b->recycled = recycled;

    ctx->buf = b;
```

```
//更新allocated,可以看到每分配一个就加1.
    ctx->allocated++;

    return NGX_OK;
}
```

然后接下来这部分就是复制buf，然后调用filter链进行发送。

```
//复制buf.
rc = ngx_output_chain_copy_buf(ctx);

            if (rc == NGX_ERROR) {
                return rc;
            }
//如果返回AGAIn，一般来说不会返回这个值的.
            if (rc == NGX_AGAIN) {
                if (out) {
                    break;
                }

                return rc;
            }

            /* delete the completed buf from the ctx->in chain */
//如果ctx->in中处理完毕的buf则删除当前的buf
            if (ngx_buf_size(ctx->in->buf) == 0) {
                ctx->in = ctx->in->next;
            }

            cl = ngx_alloc_chain_link(ctx->pool);
            if (cl == NULL) {
                return NGX_ERROR;
            }
//链接chain到out.
```

```
            cl->buf = ctx->buf;
            cl->next = NULL;
            *last_out = cl;
            last_out = &cl->next;
            ctx->buf = NULL;
        }

        if (out == NULL && last != NGX_NONE) {

            if (ctx->in) {
                return NGX_AGAIN;
            }

            return last;
        }
//调用filter链
        last = ctx->output_filter(ctx->filter_ctx, out);

        if (last == NGX_ERROR || last == NGX_DONE) {
            return last;
        }
//update chain，这里主要是将处理完毕的chain放入到free，没有处理完毕的放到busy中.
        ngx_chain_update_chains(&ctx->free, &ctx->busy, &out, ctx->tag);
        last_out = &out;
```

ngx_chain_update_chains这个函数我以前的blog有分析过，想了解的，可以看我前面的blog .

1.8 nginx中handler的处理(一)

发表时间: 2010-05-20

nginx中的处理一个http的请求分为了8个phase,分别是下面几个阶段.

其中特别要注意就是几个rewrite阶段。

```
typedef enum {
//读取请求phase
    NGX_HTTP_POST_READ_PHASE = 0,
//接下来就是开始处理

//这个阶段主要是处理全局的(server block)的rewrite。
    NGX_HTTP_SERVER_REWRITE_PHASE,
//这个阶段主要是通过uri来查找对应的location。然后将uri和location的数据关联起来
    NGX_HTTP_FIND_CONFIG_PHASE,
//这个主要处理location的rewrite。
    NGX_HTTP_REWRITE_PHASE,
//post rewrite，这个主要是进行一些校验以及收尾工作，以便于交给后面的模块。
    NGX_HTTP_POST_REWRITE_PHASE,
//比如流控这种类型的access就放在这个phase，也就是说它主要是进行一些比较粗粒度的access。
    NGX_HTTP_PREACCESS_PHASE,

//这个比如存取控制，权限验证就放在这个phase，一般来说处理动作是交给下面的模块做的.这个主要是做一些细粒度
    NGX_HTTP_ACCESS_PHASE,
//一般来说当上面的access模块得到access_code之后就会由这个模块根据access_code来进行操作
    NGX_HTTP_POST_ACCESS_PHASE,

//try_file模块，也就是对应配置文件中的try_files指令。
    NGX_HTTP_TRY_FILES_PHASE,
//内容处理模块，我们一般的handle都是处于这个模块
    NGX_HTTP_CONTENT_PHASE,
//log模块
    NGX_HTTP_LOG_PHASE
} ngx_http_phases;
```

这里要注意的就是这几个phase的执行是严格按照顺序的，也就是NGX_HTTP_POST_READ_PHASE是第一个，而LOG_PHASE是最后一个。只有一个特殊那就是FIND_CONFIG_PHASE，这个的话，有可能会在后面的rewrite phase再来调用这个phase。

这里handler的结构是这样的，在ngx_http_core_main_conf_t中会有一个包含了ngx_http_phase_t结构的数组，而ngx_http_phase_t包含了一个动态数组，也就是说每一个phase都有一个handler数组。

```
typedef struct {
.....................................................................
    ngx_http_phase_t           phases[NGX_HTTP_LOG_PHASE + 1];
} ngx_http_core_main_conf_t;

typedef struct {
//每个phase都会有一个handler数组。
    ngx_array_t                handlers;
} ngx_http_phase_t;
```

然后每个handler数组的元素都是一个hanler函数。

```
typedef ngx_int_t (*ngx_http_handler_pt)(ngx_http_request_t *r);
```

也就是我们写handler的时候注册的handler函数。

ok，接下来我们来看phase的初始化，初始化函数是ngx_http_init_phase_handlers。

在看phase初始化之前，我们先来看一个叫做ngx_http_phase_handler_s的结构体，这个结构体是保存在ngx_http_core_main_conf_t 中的，最终我们通过上面所讲的phases注册的handler链会被转换为ngx_http_phase_handler_s，然后保存在ngx_http_core_main_conf_t的phase_engine中。而后面对handler的调用处理都是使用ngx_http_phase_handler_s。

这个结构体是每个handler都会有一个的，也就是说所有的phase handler最终都会链接到一个大的数组中，这个大数组就是ngx_http_phase_engine_t的handlers域。

```
typedef struct {
//所有的hanler都会在这个数组中.
    ngx_http_phase_handler_t  *handlers;
    ngx_uint_t                 server_rewrite_index;
    ngx_uint_t                 location_rewrite_index;
} ngx_http_phase_engine_t;
```

然后我们来看它的每个域的含义。

checker 所有处于相同phase的handler的check都是相同的，每个phase的handler的调用都是在check中的，也就是check进行一些校验，结果判断等等操作。

handler就是对应的handler处理函数

ngxt 表示了下一个要执行的handler(也就是ngx_http_phase_handler_s)的位置,由于是数组，所以这个也就表示数组索引。而这个默认就是下一个将要执行的phase

```
struct ngx_http_phase_handler_s {
    ngx_http_phase_handler_pt  checker;
    ngx_http_handler_pt        handler;
    ngx_uint_t                 next;
};
```

来看函数的实现，其实功能很简单，就是初始化ngx_http_phase_handler_s，将我们注册的handler都链接到这个数组中，然后还有一些校验等。

这里要注意有些phase的话只会有一个handler，比如CONFIG_PHASE，下面的代码中我们会详细看到。

```
static ngx_int_t
ngx_http_init_phase_handlers(ngx_conf_t *cf, ngx_http_core_main_conf_t *cmcf)
{
    ngx_int_t                   j;
    ngx_uint_t                  i, n;
    ngx_uint_t                  find_config_index, use_rewrite, use_access;
    ngx_http_handler_pt        *h;
//最终的handler数组
    ngx_http_phase_handler_t   *ph;
    ngx_http_phase_handler_pt   checker;

    cmcf->phase_engine.server_rewrite_index = (ngx_uint_t) -1;
    cmcf->phase_engine.location_rewrite_index = (ngx_uint_t) -1;
    find_config_index = 0;
//是否有使用rewrite以及access。
    use_rewrite = cmcf->phases[NGX_HTTP_REWRITE_PHASE].handlers.nelts ? 1 : 0;
    use_access = cmcf->phases[NGX_HTTP_ACCESS_PHASE].handlers.nelts ? 1 : 0;

//开始计算handler 数组的大小
    n = use_rewrite + use_access + cmcf->try_files + 1 /* find config phase */;
    for (i = 0; i < NGX_HTTP_LOG_PHASE; i++) {
        n += cmcf->phases[i].handlers.nelts;
    }

//数组分配内存
    ph = ngx_pcalloc(cf->pool,
                     n * sizeof(ngx_http_phase_handler_t) + sizeof(void *));
    if (ph == NULL) {
        return NGX_ERROR;
    }

//handler数组放到handlers里面。
    cmcf->phase_engine.handlers = ph;
//n表示下一个phase的索引。
    n = 0;
```

```
//开始遍历phase handler.这里是一个phase一个phase的遍历。
    for (i = 0; i < NGX_HTTP_LOG_PHASE; i++) {
//取出对应的handler处理函数
        h = cmcf->phases[i].handlers.elts;
//根据不同的phase来处理
        switch (i) {
//server重写phase ( 也就是内部重定向phase)
        case NGX_HTTP_SERVER_REWRITE_PHASE:
//如果有定义重写规则则设置重写handler的索引n.
            if (cmcf->phase_engine.server_rewrite_index == (ngx_uint_t) -1) {
                cmcf->phase_engine.server_rewrite_index = n;
            }
//赋值checker
            checker = ngx_http_core_generic_phase;

            break;
//config phase只有一个.这里设置 find_config_index , 是因为当我们rewrite之后的url就必须重新挂载locati
        case NGX_HTTP_FIND_CONFIG_PHASE:
            find_config_index = n;
//自己的checker
            ph->checker = ngx_http_core_find_config_phase;
            n++;
            ph++;

            continue;

//rewrite phase
        case NGX_HTTP_REWRITE_PHASE:
            if (cmcf->phase_engine.location_rewrite_index == (ngx_uint_t) -1) {
                cmcf->phase_engine.location_rewrite_index = n;
            }
//共用的checker
            checker = ngx_http_core_generic_phase;

            break;
```

```
        case NGX_HTTP_POST_REWRITE_PHASE:
//如果有使用rewrite则给它的checker赋值
            if (use_rewrite) {
                ph->checker = ngx_http_core_post_rewrite_phase;
//注意它的next就是find_config phase,也就是说需要重新挂载location的数据。
                ph->next = find_config_index;
                n++;
                ph++;
            }

            continue;

        case NGX_HTTP_ACCESS_PHASE:
            checker = ngx_http_core_access_phase;
            n++;
            break;

        case NGX_HTTP_POST_ACCESS_PHASE:
            if (use_access) {
                ph->checker = ngx_http_core_post_access_phase;
                ph->next = n;
                ph++;
            }
            continue;

        case NGX_HTTP_TRY_FILES_PHASE:
            if (cmcf->try_files) {
                ph->checker = ngx_http_core_try_files_phase;
                n++;
                ph++;
            }

            continue;

        case NGX_HTTP_CONTENT_PHASE:
            checker = ngx_http_core_content_phase;
            break;
```

```
        default:
            checker = ngx_http_core_generic_phase;
        }

//这里n刚好就是下一个phase的其实索引
        n += cmcf->phases[i].handlers.nelts;

//开始遍历当前的phase的handler。
        for (j = cmcf->phases[i].handlers.nelts - 1; j >=0; j--) {
            ph->checker = checker;
//每个的handler就是注册的时候的回掉函数
            ph->handler = h[j];
//next为下一个phase的索引
            ph->next = n;
//下一个handler
            ph++;
        }
    }

    return NGX_OK;
}
```

这里需要注意就是只有下面这几个phase会有多个handler，剩余的都是只有一个handler的。

```
NGX_HTTP_POST_READ_PHASE
    NGX_HTTP_SERVER_REWRITE_PHASE,
    NGX_HTTP_REWRITE_PHASE,
    NGX_HTTP_PREACCESS_PHASE,
    NGX_HTTP_ACCESS_PHASE,
    NGX_HTTP_CONTENT_PHASE,
    NGX_HTTP_LOG_PHASE
```

接下来我们来看phase的启动。

phase的启动是在ngx_http_core_run_phases这个函数中的，这个函数会遍历所有phase然后调用他们的checker来进行处理，也就是说错误，返回代码的控制什么的都是由各自的checker做的。而所有的checker的返回值都是一样的。

```
void
ngx_http_core_run_phases(ngx_http_request_t *r)
{
    ngx_int_t                   rc;
    ngx_http_phase_handler_t   *ph;
    ngx_http_core_main_conf_t  *cmcf;

    cmcf = ngx_http_get_module_main_conf(r, ngx_http_core_module);

    ph = cmcf->phase_engine.handlers;

    while (ph[r->phase_handler].checker) {

//调用checker
        rc = ph[r->phase_handler].checker(r, &ph[r->phase_handler]);
//如果有一个checker返回OK，则后面的phase就不会处理的。
        if (rc == NGX_OK) {
            return;
        }
    }
}
```

1.9 nginx中handler的处理(二)

发表时间: 2010-05-30

这次我们来看各个phase的checker的处理。

首先我们要弄明白一个事情，那就是在nginx中，一般来说，都是在NGX_HTTP_CONTENT_PHASE中调用outputfilter的，也就是说filter是在handler中调用的，这样看来只能有一个handler能够执行outputfiler.所以说在写nginx的handler模块的话，要注意不同的phase的返回值代表的不同意思。

而当ngx_http_core_run_phases返回，也就是某个phase的checker返回了NGX_OK的话，那么也就代表当前的请求已经处理结束。

按照顺序来。

首先是ngx_http_core_generic_phase，他主要是处理下面几个phase：

引用

> post read, server rewrite, rewrite, and pre-access phases

这些phase的含义就不介绍了，前一篇blog已经有详细说明了。

在这几个phase的checker中，它将所要执行的handler的返回值分为4种类型。

1 NGX_OK 此时返回NGX_AGAIN,这里我们知道如果checker返回ok的话，整个handler的处理就会直接返回，也就是这次处理结束。并且这里phase_handler被赋值为ph->next,也就是下一个phase的索引。也就是说下次将会调用它的下一个phase的checker。

2 NGX_DECLINED 此时也返回NGX_AGAIN,而这个和上面有所不同，那就是phase_handler的赋值，这里这个值只是简单的++,也就是说会紧接着处理当前phase的下一个phase，只有当前的phase的handelr处理完毕了，才可能会处理下一个phase的handler

3 NGX_AGAIN 或者NGX_DONE，这个的话直接返回OK,也就是会结束handler的处理。

4 剩余的情况，主要是处理NGX_ERROR,以及NGX_HTTP_(也就是返回一些http的状态码)的处理。

```
ngx_int_t
ngx_http_core_generic_phase(ngx_http_request_t *r, ngx_http_phase_handler_t *ph)
{
    ngx_int_t  rc;

//调用handler
    rc = ph->handler(r);

//处理NGX_OK
    if (rc == NGX_OK) {
//下一个phase的索引
        r->phase_handler = ph->next;
        return NGX_AGAIN;
    }

//处理NGX_DECLINED
    if (rc == NGX_DECLINED) {
//处理本phase的下一个handler
        r->phase_handler++;
        return NGX_AGAIN;
    }
//直接返回OK
    if (rc == NGX_AGAIN || rc == NGX_DONE) {
        return NGX_OK;
    }

    /* rc == NGX_ERROR || rc == NGX_HTTP_...  */
//剩余的情况。
    ngx_http_finalize_request(r, rc);

    return NGX_OK;
}
```

这里我们会发现有一个ngx_http_finalize_request函数，顾名思义，这个函数就是用来释放request的，比如request相关的内存池，比如需要返回的一些状态码，比如需要断开连接等等操作，这个函数我们会在后面的专

门分析nginx关闭request(包括handler以及filter)时会详细分析。

接着来看NGX_HTTP_FIND_CONFIG_PHASE这个phase的checker。
这个phase对应的checker是ngx_http_core_find_config_phase.

由于nginx中location的处理还是一个很复杂的过程，而这里我们主要来看phase的处理，因此这里就介绍下location这个phase的处理，后面会有更详细的分析location。

这个checker主要作用是讲url和对应得location关联起来，其实也就是讲对应的location的命令关联起来，nginx这里location的实现是用一个tree来做得，这里就不详细分析location的实现了。主要来看这个phase的处理。

这个checker有可能会被调用多次的。因为每次url的改变都会改变对应的location，因此在前一篇里面，我们能看到专门有个find_config_index的索引，来供其他的phase调用。

注意这个phase只会有一个handler也就是ngx_http_core_find_location，checker中调用这个handler来挂载对应的location。

然后通过find_location的返回值来进行不同的操作，比如当返回error的时候，就需要调用ngx_http_finalize_request来回收请求。

```
ngx_int_t
ngx_http_core_find_config_phase(ngx_http_request_t *r,
    ngx_http_phase_handler_t *ph)
{
    u_char                    *p;
    size_t                     len;
    ngx_int_t                  rc;
    ngx_http_core_loc_conf_t  *clcf;


    r->content_handler = NULL;
    r->uri_changed = 0;
//find location
    rc = ngx_http_core_find_location(r);


    if (rc == NGX_ERROR) {
//回收request，然后返回OK，也就是停止handler的处理。
```

```
        ngx_http_finalize_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
        return NGX_OK;
    }

    clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);

//如果当前的请求不是internal，可是location却是内部请求，此时和error处理一样。
    if (!r->internal && clcf->internal) {
        ngx_http_finalize_request(r, NGX_HTTP_NOT_FOUND);
        return NGX_OK;
    }
.................................................
}
```

接下来我们来看NGX_HTTP_POST_REWRITE_PHASE这个phase的checker，ngx_http_core_post_rewrite_phase。

这个rewrite phase主要是用来进行一些校验，比如rewrite的最大次数，如果大于这个次数，则会直接finalize request。

这里有几个变量需要注意：

uri_changed:表示当前的uri是否有改变，也就是是否有被重定向。
uri_changes:这个的初始值是11,它的意思就是最多的rewrite次数是10次。

```
ngx_int_t
ngx_http_core_post_rewrite_phase(ngx_http_request_t *r,
    ngx_http_phase_handler_t *ph)
{
    ngx_http_core_srv_conf_t  *cscf;

    ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
                   "post rewrite phase: %ui", r->phase_handler);
//如果没有rewrite的话，直接返回again，继续接下来的handler处理。
```

```
    if (!r->uri_changed) {
        r->phase_handler++;
        return NGX_AGAIN;
    }

    ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
                   "uri changes: %d", r->uri_changes);
//changes减减。
    r->uri_changes--;

//如果为0,则说明rewrite太多次数，此时就直接finalize request
    if (r->uri_changes == 0) {
        ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
                      "rewrite or internal redirection cycle "
                      "while processing \"%V\"", &r->uri);

        ngx_http_finalize_request(r, NGX_HTTP_INTERNAL_SERVER_ERROR);
        return NGX_OK;
    }
//否则进入下一个phase的处理，这里我们知道如果有use_rewrite,它的下一个phase是NGX_HTTP_FIND_CONFIG_PHA

    r->phase_handler = ph->next;

    cscf = ngx_http_get_module_srv_conf(r, ngx_http_core_module);
//重新给loc conf赋值
    r->loc_conf = cscf->ctx->loc_conf;
//然后返回again，继续下面的handler处理。
    return NGX_AGAIN;
}
```

然后是NGX_HTTP_ACCESS_PHASE这个phase的checker，ngx_http_core_access_phase。

这个chekcer的主要逻辑也是比较简单，就是通过handler的返回值来进行不同的操作。不过这里她还有一些自己特殊处理的部分，先来看相关的两个域：

1 satisfy 这个域对应于命令satisfy_any（这个已经被deprecated）和satisfy(建议使用这个)。其中它的默认值就是NGX_HTTP_SATISFY_ALL.这个命令的意思是如果值为all，则必须满足所有的验证，为any，则只满足一个就可以通过。

2 access_code 这个主要是传递给下一个phase，post_access phase进行处理的。

然后来看它的返回值。

1 NGX_DECLINED 表示由于一些原因，access没有进行，此时则直接返回again，然后继续下面的access。

2 NGX_AGAIN 和NGX_DONE 找了下貌似access模块没有返回这两个的情况，不过我们只要知道如果要handler直接终止的话，返回这两个就ok了。

3 NGX_OK 表示通过access验证。

4 其他，比如error 以及一些http的错误吗等， 就交给ngx_http_finalize_request处理。

access phase主要是用来验证当前的请求是否允许通过。来看代码

```
ngx_int_t
ngx_http_core_access_phase(ngx_http_request_t *r, ngx_http_phase_handler_t *ph)
{
    ngx_int_t                  rc;
    ngx_http_core_loc_conf_t  *clcf;

//r->main不等于r则表示是子请求。
    if (r != r->main) {
        r->phase_handler = ph->next;
        return NGX_AGAIN;
    }

    ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
                   "access phase: %ui", r->phase_handler);
//调用handler
    rc = ph->handler(r);
//如果返回declined，则会接下来的handler处理。
    if (rc == NGX_DECLINED) {
```

```
        r->phase_handler++;
        return NGX_AGAIN;
    }
//返回ok，终止handler的处理。
    if (rc == NGX_AGAIN || rc == NGX_DONE) {
        return NGX_OK;
    }

    clcf = ngx_http_get_module_loc_conf(r, ngx_http_core_module);
//如果为all，则说明必须全部的access满足。
    if (clcf->satisfy == NGX_HTTP_SATISFY_ALL) {
//如果通过一个则继续本phase的下一个handler验证。
        if (rc == NGX_OK) {
            r->phase_handler++;
            return NGX_AGAIN;
        }

    } else {
        if (rc == NGX_OK) {
//清零access_code码。
            r->access_code = 0;

            if (r->headers_out.www_authenticate) {
                r->headers_out.www_authenticate->hash = 0;
            }
//否则满足一个access就可以通过，所以下一个handler为下一个phase的handler。
            r->phase_handler = ph->next;
            return NGX_AGAIN;
        }
//如果没有通过
        if (rc == NGX_HTTP_FORBIDDEN || rc == NGX_HTTP_UNAUTHORIZED) {
//则设置access_code，这里可以看到如果多个handler都没通过，则access_code为最后一个
            r->access_code = rc;
//本phase的下一个handler
            r->phase_handler++;
            return NGX_AGAIN;
        }
```

```
    }

    /* rc == NGX_ERROR || rc == NGX_HTTP_...  */
//最后清理request
    ngx_http_finalize_request(r, rc);
    return NGX_OK;
}
```

然后是NGX_HTTP_ACCESS_PHASE这个phase的checker，ngx_http_core_post_access_phase。这里要注意，只有use_access才会调用这个checker的，不然的话，checker链中就没有这个phase的。

这个phase比较简单，主要是处理access_code,也就是access phase中设置的access_code.

```
ngx_int_t
ngx_http_core_post_access_phase(ngx_http_request_t *r,
    ngx_http_phase_handler_t *ph)
{
    ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
                   "post access phase: %ui", r->phase_handler);
//如果有access_code
    if (r->access_code) {
//打印error
        if (r->access_code == NGX_HTTP_FORBIDDEN) {
            ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
                          "access forbidden by rule");
        }
//回收request
        ngx_http_finalize_request(r, r->access_code);
        return NGX_OK;
    }
//否则进入下一个handler
    r->phase_handler++;
    return NGX_AGAIN;
}
```

然后就是NGX_HTTP_TRY_FILES_PHASE这个phase的checker，ngx_http_core_try_files_phase。

这个phase的代码有点小多，不过逻辑还是比较简单的，那就是如果请求文件不在的话，则会打开try_file定义的location然后调用对应的文件，最后正常返回，也就是返回AGAIN.

最后就是NGX_HTTP_CONTENT_PHASE的checker，ngx_http_core_content_phase。

这个phase一般来说，我们编写的大部分handler都是属于这个phase的，他就是用来生成内容。

不过这里要注意一个域content_handler，如果这个handler存在的话，整个content phase就会只执行这一个handler，然后返回NGX_OK.

然后来分析handler返回值。

1 NGX_DONE 此时说明handler执行成功。

2 NGX_DECLINED 此时表示需要执行本phase的下一个handler

3 其他，此时需要finalize request。

来看代码

```
ngx_int_t
ngx_http_core_content_phase(ngx_http_request_t *r,
    ngx_http_phase_handler_t *ph)
{
    size_t     root;
    ngx_int_t  rc;
    ngx_str_t  path;

//如果有content_handler,则直接调用conteng_handler
    if (r->content_handler) {
        r->write_event_handler = ngx_http_request_empty_handler;
        ngx_http_finalize_request(r, r->content_handler(r));
//返回NGX_OK
        return NGX_OK;
```

```
    }

    ngx_log_debug1(NGX_LOG_DEBUG_HTTP, r->connection->log, 0,
                   "content phase: %ui", r->phase_handler);
//否则调用push进去的handler
    rc = ph->handler(r);

    if (rc == NGX_DONE) {
//直接返回ok，停止处理。
        return NGX_OK;
    }

    if (rc != NGX_DECLINED) {
//嗲用finalize requst
        ngx_http_finalize_request(r, rc);
        return NGX_OK;
    }

    /* rc == NGX_DECLINED */

    ph++;

//如果下一个handler的checker存在则返回again，以待下次调用
    if (ph->checker) {
        r->phase_handler++;
        return NGX_AGAIN;
    }

    /* no content handler was found */

//没有handler的情况。
    if (r->uri.data[r->uri.len - 1] == '/' && !r->zero_in_uri) {

        if (ngx_http_map_uri_to_path(r, &path, &root, 0) != NULL) {
            ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
                          "directory index of \"%s\" is forbidden", path.data);
        }
```

```
        ngx_http_finalize_request(r, NGX_HTTP_FORBIDDEN);
        return NGX_OK;
    }

    ngx_log_error(NGX_LOG_ERR, r->connection->log, 0, "no handler found");

    ngx_http_finalize_request(r, NGX_HTTP_NOT_FOUND);
    return NGX_OK;
}
```

1.10 nginx中sub_request的处理

发表时间: 2010-06-30

首先来看subrequest的处理。

什么是subrequest，顾名思义，那就是子请求，也就是在当前的一个请求中nginx再生成一个请求。比如在nginx的HttpAddition这个filter，就有用到subrequest。

这里要注意，一个subrequest是当父reuest执行完毕后才会被执行，并且它会将所有的需要进行的handler phase重新执行一遍(这个我们后面的代码会看到).

这里还涉及到一个很关键的filter，那就是postpone filter,这个filter就用来缓存住父request，这里的缓存就是将需要发送的数据保存到一个链表中。这个是因为会先执行subrequest，然后才会执行request，因此如果有subrequest的话，这个filter就会跳过后面的发送filter，直接返回ok。

sub request是通过post pone filter以及finalize_request，还有下面的三个域来配合实现的，接下来我们会一个个的分析。

因此这里有这样三个概念，一个是postpone_request,一个是post_request，一个是post_subrequest.其实这三个也就是request的三个域了：

```
//这个表示主的request，也就是当前的request链中最上面的那个request，通过这个域我们就能判断当前的reques
ngx_http_request_t               *main;
//这个表示当前的request的父request。
    ngx_http_request_t               *parent;
//最关键就是下面三个域。
    ngx_http_postponed_request_t     *postponed;
    ngx_http_post_subrequest_t       *post_subrequest;
    ngx_http_posted_request_t        *posted_requests;
```

ok，我们一个个来看，先来看postponed，这个域用来缓存父request的数据（也就是将要发送数据的request)，而缓存这个动作是在postpone filter中来做的，我们后面回来分析这个filter。下面就是它的结构：

```
struct ngx_http_postponed_request_s {
    ngx_http_request_t               *request;
    ngx_chain_t                      *out;
    ngx_http_postponed_request_t     *next;
};
```

可以看到它就是一个很简单的链表，三个域的意思分别为：
request 保存了subrequest
out保存了所需要发送的chain。
next保存了下一个postpone_request.

然后是post_subrequest,这个域保存了子请求的post request，它也就是保存了需要被发送的request. 来看它的结构：

```
typedef struct {
    ngx_http_post_subrequest_pt       handler;
    void                             *data;
} ngx_http_post_subrequest_t;
```

可以看到它的结构更加简单，一个handler，保存了到时需要执行的回掉函数，一个data，保存了传递的数据。

最后是posted_requests，这个保存了所有的需要处理的request链表，也就是说它即包含子请求也包含父请求。来看它的结构：

```
struct ngx_http_posted_request_s {
    ngx_http_request_t               *request;
    ngx_http_posted_request_t        *next;
};
```

request保存了需要处理的request，next保存了下一个需要处理的request。

然后我们来详细分析sub request的处理流程以及代码，这里代码的分析顺序是按照sub request的流程来的。

首先来看sub request的设置函数ngx_http_subrequest。

这个函数的主要功能就是新建一个request，然后设置对应的属性，其中大部分属性都是和父request相同的，还有一些特殊的sub request独有的属性我们会在下面的代码中分析到(主要是我上面介绍的4个域)。

这个函数的参数比较多，有6个参数，来看它的原型：

```
ngx_int_t
ngx_http_subrequest(ngx_http_request_t *r,
    ngx_str_t *uri, ngx_str_t *args, ngx_http_request_t **psr,
    ngx_http_post_subrequest_t *ps, ngx_uint_t flags)
```

r表示需要生成子请求的request，uri表示子请求的uri，args表示子请求的参数，psr表示最终生成的子请求，ps表示子请求的post_subrequest，flags主要控制sub request的内容是否要放到内存中。

代码比较长，我们分开来看，下面这段是相关的初始化：

```
ngx_connection_t              *c;
    ngx_http_request_t            *sr;
    ngx_http_core_srv_conf_t      *cscf;
    ngx_http_postponed_request_t  *pr, *p;

//subrequest表示了当前还可以处理的最大个数的sub request，这个值的默认值是50.表示nginx中能够处理的最多
    r->main->subrequests--;

//如果为0,则表示已达到最大的限制，因此返回error。
    if (r->main->subrequests == 0) {
        ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
                      "subrequests cycle while processing \"%V\"", uri);
        r->main->subrequests = 1;
        return NGX_ERROR;
    }
```

```
//新建一个sub request。
    sr = ngx_pcalloc(r->pool, sizeof(ngx_http_request_t));
    if (sr == NULL) {
        return NGX_ERROR;
    }

    sr->signature = NGX_HTTP_MODULE;

//设置connection。
    c = r->connection;
    sr->connection = c;

//下面的初始化大部分都是和父请求一样的。
    sr->ctx = ngx_pcalloc(r->pool, sizeof(void *) * ngx_http_max_module);
    if (sr->ctx == NULL) {
        return NGX_ERROR;
    }

    if (ngx_list_init(&sr->headers_out.headers, r->pool, 20,
                      sizeof(ngx_table_elt_t))
        != NGX_OK)
    {
        return NGX_ERROR;
    }
.....................................................
    sr->request_body = r->request_body;
//可以看到子请求只会是Get方法。
    sr->method = NGX_HTTP_GET;
    sr->http_version = r->http_version;

    sr->request_line = r->request_line;
```

接下来这段也是初始化，只不过主要是初始化一些sub request特有的属性。这里最关键的就是两个事件处理函数的赋值，read_event_handler和write_event_handler 。其中读事件的handler被赋值为一个空的函数，也就是在sub request中，不会处理读事件。而写事件的handler被赋值为ngx_http_handler，这个函数我们知道，

它就是整个nginx的handler处理的入口，因此也就是说sub request最终会把所有的phase再重新走一遍。

这里还要注意，那就是父请求可能会有多个儿子请求。

```
//子请求的uri
    sr->uri = *uri;


    if (args) {
//参数设置
        sr->args = *args;
    }


    ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
                   "http subrequest \"%V?%V\"", uri, &sr->args);
//子请求的内容是否需要放到内存中。
    sr->subrequest_in_memory = (flags & NGX_HTTP_SUBREQUEST_IN_MEMORY) != 0;
//这个貌似是ssi用到的。
    sr->waited = (flags & NGX_HTTP_SUBREQUEST_WAITED) != 0;


.....................................................................


    ngx_http_set_exten(sr);
//设置main，也就是最上层的那个request。
    sr->main = r->main;
//设置父request
    sr->parent = r;
//可以看到ost_subrequest 被设置为我们传递进来的值。
    sr->post_subrequest = ps;


//读写事件的处理函数的赋值
    sr->read_event_handler = ngx_http_request_empty_handler;
    sr->write_event_handler = ngx_http_handler;


//设置连接的request为子请求。这里的意思是如果父请求设置第二个子请求的话，这里就不需要设置连接的request
    if (c->data == r && r->postponed == NULL) {
        c->data = sr;
    }
```

```
    sr->variables = r->variables;


    sr->log_handler = r->log_handler;

//开始赋值postponed request.
    pr = ngx_palloc(r->pool, sizeof(ngx_http_postponed_request_t));
    if (pr == NULL) {
        return NGX_ERROR;
    }
//它的request设置为子请求，也就是每个子请求都会用一个postponed request包装起来。
    pr->request = sr;
    pr->out = NULL;
    pr->next = NULL;
//如果是第一次给父请求设置孩子，那么将pr放到postponed链表的结尾。
    if (r->postponed) {
        for (p = r->postponed; p->next; p = p->next) { /* void */ }
//找到尾部，然后插入。
        p->next = pr;


    } else {
//否则直接设置
        r->postponed = pr;
    }

//设置内部标记
    sr->internal = 1;


    sr->discard_body = r->discard_body;
    sr->expect_tested = 1;
    sr->main_filter_need_in_memory = r->main_filter_need_in_memory;


    sr->uri_changes = NGX_HTTP_MAX_URI_CHANGES + 1;
//subrequests加1.
    r->main->subrequests++;


//保存生成的sub request，以供外部使用。
    *psr = sr;
```

```
//设置post request。
    return ngx_http_post_request(sr);
}
```

上面的代码有个有疑问的地方，那就是subrequests，我查了下代码只有这个函数里面有对它进行操作，可是这里前面--，后面++，那不是基本没有可能这个值是0。

前面的代码我们可以看到最后会调用ngx_http_post_reques来处理，这个函数是用来讲subrequest放到post request中的。

而post request的调用我会在后面分析到。

```
ngx_int_t
ngx_http_post_request(ngx_http_request_t *r)
{
    ngx_http_posted_request_t  *pr, **p;
//新建一个post request。
    pr = ngx_palloc(r->pool, sizeof(ngx_http_posted_request_t));
    if (pr == NULL) {
        return NGX_ERROR;
    }

//设置request为sub request.
    pr->request = r;
    pr->next = NULL;
//找到post request的尾部。
    for (p = &r->main->posted_requests; *p; p = &(*p)->next) { /* void */ }
//然后赋值。
    *p = pr;

    return NGX_OK;
}
```

然后来看postpone 这个filter，这个filter就是用来缓存父request的chain， 并且控制sub request的发送。

代码分段来看，先来看第一部分，这部分主要是处理父请求进来的情况，也就是缓存父请求的chain。

```
ngx_connection_t              *c;
    ngx_http_postponed_request_t  *pr;
//取得当前的链接
    c = r->connection;

//如果r不等于c->data,前面的分析知道c->data保存的是最新的一个sub request(同级的话，是第一个),因此不等于
    if (r != c->data) {

        if (in) {
//保存数据(下面会分析这段代码)
            ngx_http_postpone_filter_add(r, in);
//这里注意不发送任何数据，直接返回OK。而最终会在finalize_request中处理。
            return NGX_OK;
        }

        return NGX_OK;
    }

//如果r->postponed为空，则说明是最后一个sub request，也就是最新的那个，因此需要将它先发送出去。
    if (r->postponed == NULL) {

//如果in存在，则发送出去
        if (in || c->buffered) {
            return ngx_http_next_filter(r->main, in);
        }

        return NGX_OK;
    }
```

然后来看ngx_http_postpone_filter_add这个方法，这个方法主要是拷贝当前需要发送的chain到postponed

的out域中。

这里要注意一个的就是由于filter有可能会进入多次，因此如果相同的request的in chain会拷贝到相同的posrponed request中。

还有这里要注意就是这里添加的postponed request的request域是NULL，也就是说明这个postponed request就是自己，也就是r==r->postponed->request.

```
static ngx_int_t
ngx_http_postpone_filter_add(ngx_http_request_t *r, ngx_chain_t *in)
{
    ngx_http_postponed_request_t  *pr, **ppr;

//如果postponed存在，则进入相关处理
    if (r->postponed) {
//找到postponed的尾部
        for (pr = r->postponed; pr->next; pr = pr->next) { /* void */ }
//如果为空，则直接添加到当前的chain
        if (pr->request == NULL) {
            goto found;
        }

        ppr = &pr->next;
    } else {
        ppr = &r->postponed;
    }

    pr = ngx_palloc(r->pool, sizeof(ngx_http_postponed_request_t));
    if (pr == NULL) {
        return NGX_ERROR;
    }

    *ppr = pr;
//可以看到request是空。
    pr->request = NULL;
    pr->out = NULL;
    pr->next = NULL;
```

```
found:
//最终复制in到pr->out,也就是保存request 需要发送的数据。
    if (ngx_chain_add_copy(r->pool, &pr->out, in) == NGX_OK) {
        return NGX_OK;
    }

    return NGX_ERROR;
}
```

然后再回到ngx_http_postpone_filter，剩下的这段代码主要就是用来发送前面保存的父请求的chain.

```
//到达这里说明需要发送父请求的数据了。
if (in) {
//如果有chain，则保存数据。
        ngx_http_postpone_filter_add(r, in);
    }

//开始遍历postponed request.
    do {
        pr = r->postponed;
//如果存在request，则说明这个postponed request是sub request，因此需要将它放到post_request中。
        if (pr->request) {

            ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
                           "http postpone filter wake \"%V?%V\"",
                           &pr->request->uri, &pr->request->args);

            r->postponed = pr->next;

            c->data = pr->request;
//放到post request中。
            return ngx_http_post_request(pr->request);
```

```
        }

        if (pr->out == NULL) {
            ngx_log_error(NGX_LOG_ALERT, c->log, 0,
                          "http postpone filter NULL output",
                          &r->uri, &r->args);

        } else {
//说明pr->out不为空，此时需要将保存的父request的数据发送。
            ngx_log_debug2(NGX_LOG_DEBUG_HTTP, c->log, 0,
                           "http postpone filter output \"%V?%V\"",
                           &r->uri, &r->args);
//发送
            if (ngx_http_next_filter(r->main, pr->out) == NGX_ERROR) {
                return NGX_ERROR;
            }
        }

        r->postponed = pr->next;

    } while (r->postponed);
```

然后我们来看ngx_http_finalize_request中sub request的处理部分，其实ngx_http_finalize_request中，一大部分都是sub request的处理，

这个处理其实主要就是修改一开始介绍的request的三个域。

这里要注意，由于如果有sub request的话，postponed filter会返回NGX_OK。

还有就是所有需要处理的request都必须放入到post request中，儿postponed request中保存的是暂时缓存的不需要发送的request。

```
//如果r不等于r->main的话，则说明当前的请求是是sub request，此时进入相关处理。
 if (r != r->main) {
```

```
//如果含有postponed的话，则说明这个request并不是最后一个sub request，因此设置write handler，并且返回
        if (r->buffered || r->postponed) {

            if (ngx_http_set_write_handler(r) != NGX_OK) {
                ngx_http_close_request(r->main, 0);
            }

            return;
        }
//取得request的父request
        pr = r->parent;

//如果r等于c->data,则说明当前是最后一个sub request，此时需要修改c->data,以便于在postponed filter中发
        if (r == c->data) {
.....................................................................

            r->done = 1;
//如果父request的postponed存在并且它的request为当前的r，则开始处理接下来的postponed。
            if (pr->postponed && pr->postponed->request == r) {
//取next
                pr->postponed = pr->postponed->next;
            }
//修改c->data,这个将会在run_post_request中使用，接下来就会分析这个函数。
            c->data = pr;

        } else {
//否则则设置write handler.
            r->write_event_handler = ngx_http_request_finalizer;

            if (r->waited) {
                r->done = 1;
            }
        }
//最终将pr也就是父request放入到post request中。
        if (ngx_http_post_request(pr) != NGX_OK) {
            ngx_http_close_request(r->main, 0);
            return;
```

```
        }
.........................................

        return;
    }
```

上面有一个函数那就是ngx_http_set_write_handler，这个用来设置write handler,这里是这是write handler为ngx_http_writer，而我们要知道sub request的处理是，不停的保存父request的chain(在postponed filter中),而每次保存完毕之后就返回ngx_ok,此时由于这个request已经经历完毕所有的handler phase，因此我们就需要修改它的write handler，以便于需要发送的时候跳过handler 阶段，因此就设置ngx_http_writer，这个函数主要就是调用out_put filter,而不经过handler phase。

最后我们来看post request的调用在那里。

在nginx中，request的执行是在ngx_http_process_request中的，来看这个函数的最后两句：

```
//处理request，每个sub request在处理之前的write handler 都是这个函数。
ngx_http_handler(r);

//开始run post request
    ngx_http_run_posted_requests(c);
```

我们来详细看ngx_http_run_posted_requests的实现。这个函数就是遍历post request,然后调用它的write handler对request进行处理。

```
void
ngx_http_run_posted_requests(ngx_connection_t *c)
{
    ngx_http_request_t         *r;
    ngx_http_log_ctx_t         *ctx;
    ngx_http_posted_request_t  *pr;
```

```
//开始遍历
    for ( ;; ) {

//如果连接已经被销毁，则直接返回。
        if (c->destroyed) {
            return;
        }
//取出c->data(可以看到在finalize request中会修改到这个域的,也就是这个域会始终保存最新的sub request (
        r = c->data;
        pr = r->main->posted_requests;

        if (pr == NULL) {
            return;
        }
//赋值为下一个。
        r->main->posted_requests = pr->next;

        r = pr->request;

        ctx = c->log->data;
        ctx->current_request = r;
//调用write handler.

        r->write_event_handler(r);
    }
}
```