

# nginx 核心讲解

## 第零章 前言

慕名对 nginx 的源码进行学习研究是早在 2009 年的事情，当时还在学校，整天呆在实验室里看动漫，时间一久就心感愧疚，觉得还是要趁有空学点东西，恰当时不知从哪里得知高性能服务器是一个很有“前途”的方向，几经搜索又机缘偶合的得识 lighttpd 与 nginx，从此开始在动漫与代码之间来回穿梭，直到毕业。

关于 lighttpd 与 nginx，无需多说，当时 lighttpd 比 nginx 要火，所以我先看的 lighttpd 源码，后看的 nginx 源码，也因此 lighttpd 的文档在我读书的时候就写完（虽然写得很矬）了，但 nginx 的文档写了一些放在电脑里，后来离开学校开始工作后，就把这件事情和这些文档都给搁在那了，直到近一年前，我建了一个个人博客站点 (<http://lenky.info/>)，为了凑文章数目，才又把它们给找了出来，并且根据最新的 nginx 源码重新整理了一下，也就是现在你看到的这篇文档。

重新整理主要是因为注意到以前写的文档过细的去逐行注释代码（网上很多 nginx 源码分析的文章也大多有这个缺点），而此次希望能从比较高一点的角度去解析 nginx，让读者尽快的把握全局，搞清楚整体实现原理而不是陷入细节。我个人认为，只要看清楚了整体的实现原理，对于一时半会没有触及到的细节，在真正遇到那个点再去理解时肯定毫不费力，最多也只不过是可能还需要去查一下 man 手册，了解一下系统 API 而已。授人以鱼不如授人以渔，这也是我此次重新整理的最主要目标，为了达到这个目标，文档里就尽量少的贴代码多画图，当然一些必要的代码是不可缺少的，所以你还是会在本文档里看到源代码。从文档名称叫做核心讲解，而不是源码分析，这也足见我的个人期望，虽然我的个人期望比较好，可惜限于水平比较差，目前写出来的文档也就这个样了。：)

最后，说一下本文档基于的相关环境，虽然列了一个表格如下，其实没那么复杂，我安装的是一个 centos 6.2 的 32 位虚拟机（Intel x86），其它开发软件包都是 centos 6.2 里所对应提供的，而 nginx 版本为 1.2.0。另外，在没有特别说明的情况下，文档讲解过程中针对的 nginx 配置环境是 http 协议、epoll 事件机制。

软件包	版本
nginx	1.2.0
os	CentOS release 6.2 (Final)/kernel-2.6.32/32bit
gcc	gcc version 4.4.6 20110731 (Red Hat 4.4.6-3) (GCC)
gdb	GNU gdb (GDB) Red Hat Enterprise Linux (7.2-50.el6)
make	GNU Make 3.81

文档版本（由于文档目前在逐步整理中，尚未仔细校订，所以难免出现错误，如发现错误请反馈，非常感谢，文档将持续更新，请关注更新地址：<http://lenky.info/ebook/>）：

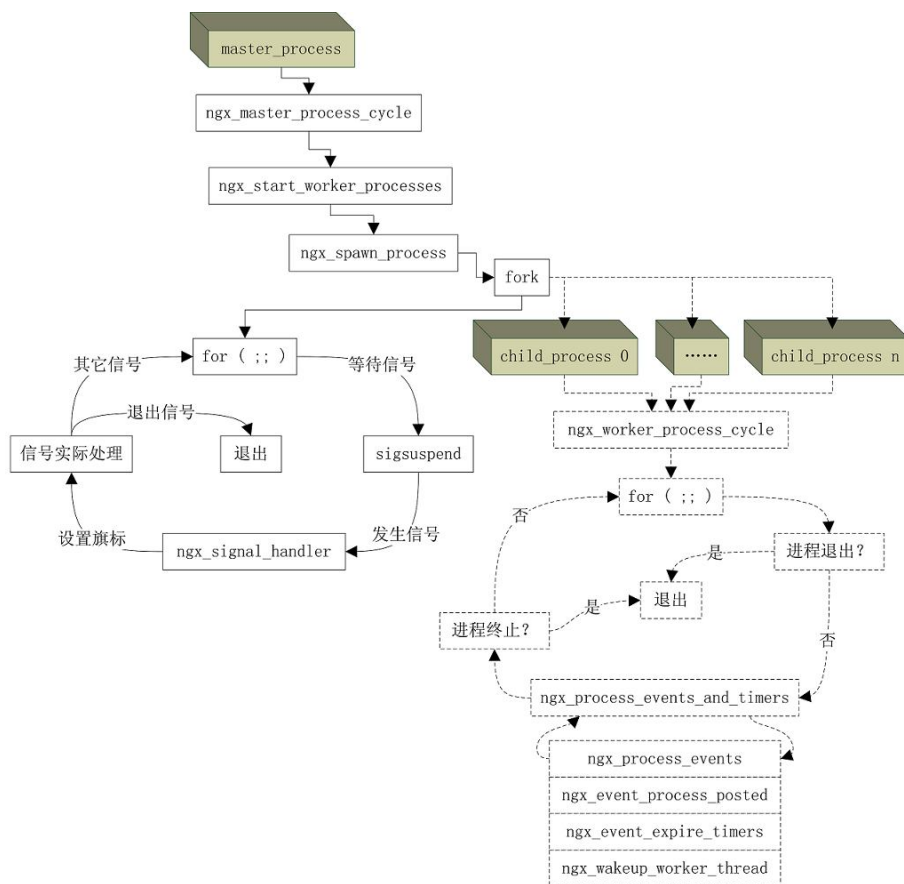
版本号	修订时间
0.1	2012-7-20
0.2	2012-7-22 新增事件机制一章
0.3	2012-7-29 新增初始变量、支撑机制、脚本引擎三节
0.31	2012-8-5 修订&完成第六章
0.32	2012-8-9 修订第三章
0.35	2012-8-11 新增第七章：创建监听套接口、创建连接套接口
0.4	2012-8-15 完成第七章

## 第一章 进程解析

## 进程模型

nginx 的进程模型和大多数后台服务程序一样，按职责将进程分成监控进程和工作进程两类，启动 nginx 的主进程充当监控进程，而由主进程 fork 出来的子进程则充当工作进程。工作进程的任务自然是完成具体的业务逻辑，而监控进程充当整个进程组的对外接口，同时对工作进程进行监护，比如如果某工作进程意外退出，监控进程将重新 fork 生成一个新的工作进程。nginx 也可以单进程模型执行，在这种进程模型下，主进程就是工作进程，此时没有监控进程，单进程模型比较简单且官方建议仅供测试使用，所以下面主要分析多进程模型。

分析 `nginx` 多进程模型的入口函数为主进程的 `ngx_master_process_cycle()` 函数，在该函数做完信号处理设置等之后就会调用一个名为 `ngx_start_worker_processes()` 的函数用于 `fork` 产生出子进程（子进程数目通过函数调用的第二个实参指定），子进程作为一个新的实体开始充当工作进程的角色执行 `ngx_worker_process_cycle()` 函数，该函数主体为一个无限 `for` 循环，持续不断的处理客户端的服务请求，而主进程继续执行 `ngx_master_process_cycle()` 函数，也就是作为监控进程执行主体 `for` 循环，这也是一个无限循环，直到进程终止才退出，服务进程基本都是这种写法，所以不用详述，下面先看看这个模型的图示：



上图中表现得很明朗，监控进程和工作进程各有一个无限 for 循环，以便进程持续的等待和处理自己负责的事务，直到进程退出。

监控进程的无限 for 循环内有一个关键的 `sigsuspend()` 函数调用，该函数的调用使得监控进程的大部分时间都处于挂起等待状态，直到监控进程接收到信号为止，当监控进程接收到信号时，信号处理函数 `ngx_signal_handler()` 就会被执行，我们知道信号处理函数一般都要求足够简单（关于信号处理函数的实现准则请 Google），所以在该函数内执行的动作主要也就是根据当前信号值对相应的旗标变量做设置，而实际的处理逻辑必须放在主体代码里来处理，所以该 for 循环接下来的代码就是判断有哪些旗标变量被设置而需要处理的，比如 `ngx_reap`（有子进程退出？）、`ngx_quit` 或 `ngx_terminate`（进程要退出或终止？注意：虽然两个旗标都是表示结束 `nginx`，不过 `ngx_quit` 的结束更优雅，它会让 `nginx` 监控进程做一些清理工作且等待子进程也完全清理并退出之后才终止，而 `ngx_terminate` 更为粗暴，不过它通过使用 `SIGKILL` 信号能保证在一段时间后必定被结束掉）、`ngx_reconfigure`（重新加载配置？）等。当所有信号都处理完时又挂起在函数 `sigsuspend()` 调用处继续等待新的信号，如此反复，构成监控进程的主要执行体。

```

82: Filename : ngx_process_cycle.c
83: void
84: ngx_master_process_cycle(ngx_cycle_t *cycle)
85: {
86: ...
146:     for (;;) {
147:         ...
170:         sigsuspend(&set);
171:         ...
177:         if (ngx_reap) {
178:             ...
184:             if (!live && (ngx_terminate || ngx_quit)) {
185:                 ...
188:                 if (ngx_terminate) {
189:                     ...
210:                     if (ngx_quit) {
211:                         ...
212:                     }
213: ...

```

工作进程的执行主体与监控进程类似，不过工作进程既名之为工作进程，那么它的主要关注点就是与客户端或后端真实服务器（此时 `nginx` 作为中间代理）之间的数据可读/可写等交互事件，而不是进程信号，所以工作进程的阻塞点是在像 `select()`、`epoll_wait()` 等这样的 I/O 多路复用函数调用处，以等待发生数据可读/可写事件，当然，也可能被新收到的进程信号中断。关于 I/O 多路复用的更多细节，请参考其他章节。

```

721: Filename : ngx_process_cycle.c

```

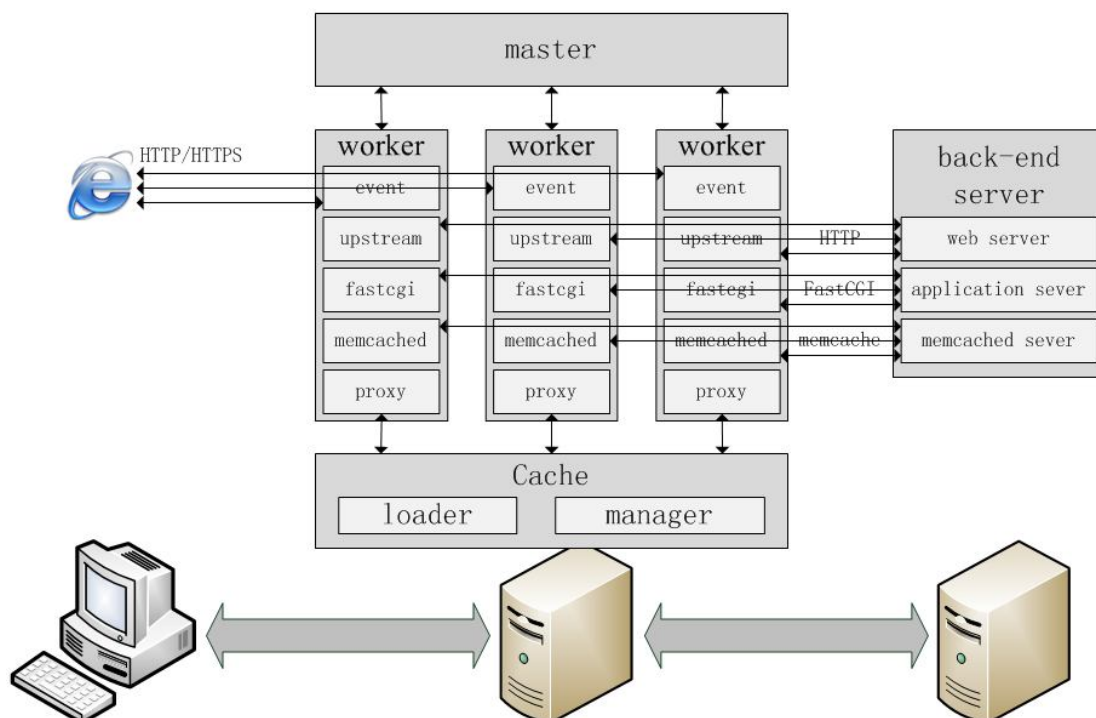
```

722: static void
723: ngx_worker_process_cycle(ngx_cycle_t *cycle, void *data)
724: {
725: ...
780:     for (;;) {
781:
782:         if(ngx_exiting) {
783:             ...
806:             ngx_process_events_and_timers(cycle);
807:
808:             if(ngx_terminate) {
809:                 ...
810:             }
811: ...

```

## 整体架构

如前面介绍的那样, 正常执行起来后的 Nginx 会有多个进程, 最基本的有 `master_process` 和 `worker_process`, 还可能会有 `cache` 相关进程 (这在后面会具体讲到)。除了自身进程之间的相互通信, Nginx 还凭借强悍的模块功能与外界四通八达, 比如通过 `upstream` 与 `web server` 通信、依靠 `fastcgi` 与 `application server` 通信等等。一个较为完整的整体架构框图如下所示:



## 进程通信

运行在多线程模型的 nginx 在正常工作时，自然就会有多个进程实例，比如下图是在配置 “worker\_processes 4;” 情况下的显示，nginx 设置的进程 title 能很好的帮助我们区分监控进程与工作进程，不过带上选项 f 的 ps 命令以树目录的形式打印各个进程信息也能帮助我们做这个区分。多线程联合工作必定要牵扯到进程之间的通信问题，下面就来看看 nginx 是如何做的。

```
[root@localhost ~]# ps auxf | grep nginx | grep -v grep
root      8706  0.0  0.2   5164   568 ?        Ss   06:36   0:00 nginx: master process ./objs/nginx
nobody    8707  0.0  0.3   5336   984 ?        S    06:36   0:00 \_ nginx: worker process
nobody    8708  0.0  0.3   5336   984 ?        S    06:36   0:00 \_ nginx: worker process
nobody    8709  0.0  0.3   5336   984 ?        S    06:36   0:00 \_ nginx: worker process
nobody    8710  0.0  0.3   5336   976 ?        S    06:36   0:00 \_ nginx: worker process
```

采用 socketpair() 函数创建一对未命名的 UNIX 域套接字来进行 Linux 下具有亲缘关系的进程之间的双向通信是一个非常不错的解决方案。nginx 就是这么做的，先看 fork 生成新工作进程的 ngx\_spawn\_process() 函数以及相关代码：

```
21: Filename : ngx_process.h
22: typedef struct {
23:     ngx_pid_t      pid;
24:     int             status;
25:     ngx_socket_t    channel[2];
26: ...
27: } ngx_process_t;
28: ...
47: #define NGX_MAX_PROCESSES          1024

35: Filename : ngx_process.c
36: ngx_process_t    ngx_processes[NGX_MAX_PROCESSES];
37:
86: ngx_pid_t
87: ngx_spawn_process(ngx_cycle_t *cycle, ngx_spawn_proc_pt proc, void *data,
88:     char *name, ngx_int_t respawn)
89: {
90: ...
117:     if (socketpair(AF_UNIX, SOCK_STREAM, 0, ngx_processes[s].channel) == -1)
118: ...
186:     pid = fork();
187: ...
```

在该函数进行 fork() 之前，先调用了 socketpair() 创建一对 socket 描述符存放在变量 ngx\_processes[s].channel 内（其中 s 标志在 ngx\_processes 数组内第一个可用元素的下标，比如最开始产生第一个工作进程时，可用元素的下标 s 为 0），而在 fork() 之后，由于子进程继承了父进程的资源，那么父子进程就都有了这一对 socket 描述符，而 nginx 将 channel[0] 给父

进程使用，`channel[1]`给子进程使用，这样分别错开的使用不同 `socket` 描述符，即可实现父子进程之间的双向通信：



除此之外，对于各个子进程之间，也可以进行双向通信。如前面所述，父子进程的通信 `channel` 设定是自然而然的事情，而子进程之间的通信 `channel` 设定就涉及到进程之间文件描述符（`socket` 描述符也属于文件描述符）的传递，因为虽然后生成的子进程通过继承的 `channel[0]`能够往前生成的子进程发送信息，但前生成的子进程无法获知后生成子进程的 `channel[0]`而不能发送信息，所以后生成的子进程必须利用已知的前生成子进程的 `channel[0]`进行主动告知，下面来看看这个具体是怎样的。

在子进程的启动初始化函数 `ngx_worker_process_init()`里，会把 `ngx_channel`（也就是 `channel[1]`）加入到读事件监听集里，对应的回调处理函数为 `ngx_channel_handler()`：

```

834: Filename : ngx_process_cycle.c
835: static void
836: ngx_worker_process_init(ngx_cycle_t *cycle, ngx_uint_t priority)
837: {
838: ...
839:     if (ngx_add_channel_event(cycle, ngx_channel, NGX_READ_EVENT,
840:                             ngx_channel_handler)
841:         == NGX_ERROR)
842:     {
843: ...
  
```

而在父进程 `fork()`生成一个新子进程后，就会立即通过 `ngx_pass_open_channel()`函数把这个子进程的相关信息告知给其前面已生成的子进程：

```

430: Filename : ngx_process_cycle.c
431: static void
432: ngx_pass_open_channel(ngx_cycle_t *cycle, ngx_channel_t *ch)
433: {
434:
435:     for (i = 0; i < ngx_last_process; i++) {
436: ...
437:         ngx_write_channel(ngx_processes[i].channel[0],
438:                         ch, sizeof(ngx_channel_t), cycle->log);
439:     }
440: }
  
```

其中参数 `ch` 里包含了刚创建的新子进程（假定为 A）的 `pid`、进程信息在全局数组里存储下标、`socket` 描述符 `channel[0]`等信息，这里通过 `for` 循环遍历所有存活的其它子进程，然后调用函数 `ngx_write_channel()`通过继承的 `channel[0]`描述符进行信息主动告知，而收到这些消息的子进程将执行设置好的回调函数 `ngx_channel_handler()`，把接收到的新子进程 A 的



相关信息存储在全局变量 `ngx_processes` 内:

```

1066:   Filename : ngx_process_cycle.c
1067:   static void
1068:   ngx_channel_handler(ngx_event_t *ev)
1069:   {
1070:   ...
1126:           case NGX_CMD_OPEN_CHANNEL:
1127:   ...
1132:           ngx_processes[ch.slot].pid = ch.pid;
1133:           ngx_processes[ch.slot].channel[0] = ch.fd;
1134:           break;
1135:   ...

```

这样，前后子进程都有了对方的相关信息，相互通信也就没有问题了，这其中还有一些没讲到的具体实现细节，请以关键字“进程之间文件描述符传递”进行 Google 搜索。直接看一下实例，就以上面显示的各个父子进程为例：

ngx_processes	父-8706	子-8707	子-8708	子-8709	子-8710
[0]-8707-channel	{3, 7}*	{-1, 7}**	{3, -1}	{3, -1}	{3, -1}
[1]-8708-channel	{8, 9}	{3, 0}	{-1, 9}	{8, -1}	{8, -1}
[2]-8709-channel	{10, 11}	{9, 0}	{7, 0}	{-1, 11}	{10, -1}
[3]-8710-channel	{12, 13}	{10, 0}	{8, 0}	{7, 0}	{-1, 13}

上表格中，{a, b}分别表示 `channel[0]`和 `channel[1]`的值，-1 表示这之前是描述符，但在其后被主动 `close()`掉了，0 表示这一直都无对应的描述符，其它数字表示对应的描述符值。比如，带\*的{3, 7}表示如果父进程 8706 向子进程 8707 发送消息，需使用 `channel[0]`，即描述符 3，它的 `channel[1]`为 7，没有被 `close()`关闭掉，但一直也都没有被使用，所以没有影响，不过按道理应该关闭才是；而带\*\*的{-1, 7}表示如果子进程 8707 向父进程 8706 发送消息，需使用 `channel[1]`，即描述符 7，它的 `channel[0]`为-1 表示已经 `close()`关闭掉了（nginx 某些地方调用 `close()`时并没有设置对应变量为-1，我这里为了好说明，对已经 `close()`掉的描述符全部标记为-1 了）；

越是后生成的子进程，其 `channel[0]`与父进程的对应 `channel[0]`值相同的越多，因为基本都是继承而来，但前面生成的子进程的 `channel[0]`是通过传递获得的，所以与父进程的对应 `channel[0]`不一定相等。比如如果子进程 8707 向子进程 8710 发送消息，需使用 `channel[0]`，即描述符 10，而对应的父进程 `channel[0]`却是 12，虽然它们在各自进程里却表现为不同的整型数字，但在内核里表示同一个描述符结构，即不管是子进程 8707 往描述符 10 写数据还是父进程 8706 往描述符 12 写数据，子进程 8710 都能通过描述符 13 正确读取到这些数据，至于子进程 8710 怎么识别它读到的数据是来之子进程 8707 还是父进程 8706，就得靠其收到的数据特征（比如 `pid` 字段）来做标记区分。

最后，就目前 nginx 代码来看，子进程并没有往父进程发送任何消息，子进程之间也没有相互通信的逻辑，也许是因为 nginx 有其它一些更好的进程通信方式，比如共享内存等，



所以这种 channel 通信目前仅做为父进程往子进程发送消息使用，但由于有这个基础在这，如果未来要使用 channel 做这样的事情，的确是可以的。

## 共享内存

共享内存是 Linux 下进程之间进行数据通信的最有效方式之一，而 nginx 就为我们提供了统一的操作接口来使用共享内存。

在 nginx 里，一块完整的共享内存以结构体 ngx\_shm\_zone\_t 来封装表示，这其中包括的字段有共享内存的名称 (shm\_zone[i].shm.name)、大小 (shm\_zone[i].shm.size)、标签 (shm\_zone[i].tag)、分配内存的起始地址 (shm\_zone[i].shm.addr) 以及初始回调函数 (shm\_zone[i].init) 等：

```
24: Filename : ngx_cycle.h
25: typedef struct ngx_shm_zone_s  ngx_shm_zone_t;
26: ...
27: struct ngx_shm_zone_s {
28:     void                *data;
29:     ngx_shm_t            shm;
30:     ngx_shm_zone_init_pt init;
31:     void                *tag;
32: };
```

这些字段大都容易理解，只有 tag 字段需要解释一下，因为看上去它和 name 字段有点重复，而事实上，name 字段主要用作共享内存的唯一标识，它能让 nginx 知道我想使用哪个共享内存，但它没法让 nginx 区分我到底是想新建一个共享内存，还是使用那个已存在的旧的共享内存。举个例子，模块 A 创建了共享内存 sa，模块 A 或另外一个模块 B 再以同样的名称 sa 去获取共享内存，那么此时 nginx 是返回模块 A 已创建的那个共享内存 sa 给模块 A/模块 B，还是直接以共享内存名重复提示模块 A/模块 B 出错呢？不管 nginx 采用哪种做法都有另外一种情况出错，所以新增一个 tag 字段做冲突标识，该字段一般也就指向当前模块的 ngx\_module\_t 变量即可。这样在上面的例子中，通过 tag 字段的帮助，如果模块 A/模块 B 再以同样的名称 sa 去获取模块 A 已创建的共享内存 sa，模块 A 将获得它之前创建的共享内存的引用（因为模块 A 前后两次请求的 tag 相同），而模块 B 则将获得共享内存已做它用的错误提示（因为模块 B 请求的 tag 与之前模块 A 请求时的 tag 不同）。

当我们要使用一个共享内存时，总会在配置文件里加上该共享内存的相关配置信息，而 nginx 在进行配置解析的过程中，根据这些配置信息就会创建对应的共享内存，不过此时的创建仅仅只是代表共享内存的结构体 ngx\_shm\_zone\_t 变量的创建，这具体实现在函数 shared\_memory\_add() 内。另外从这个函数中，我们也可以看到 nginx 使用的所有共享内存都以 list 链表的形式组织在全局变量 cf->cycle->shared\_memory 下，在创建新的共享内存之前会先

对该链表进行遍历查找以及冲突检测,对于已经存在且不存在冲突的共享内存可直接返回引用。以 ngx\_http\_limit\_req\_module 模块为例,它需要的共享内存存在配置文件里以 limit\_req\_zone 配置项出现:

```
limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;
```

nginx 在进行配置解析时,遇到 limit\_req\_zone 配置项则调用其对应的处理函数 ngx\_http\_limit\_req\_zone(),而在该函数内又将继续调用函数 shared\_memory\_add()创建对应的 ngx\_shm\_zone\_t 结构体变量并加入到全局链表内:

```
ngx_http_limit_req_zone() -> ngx_shared_memory_add() -> ngx_list_push()
```

共享内存的真正创建是在配置文件全部解析完后,所有代表共享内存的结构体 ngx\_shm\_zone\_t 变量以链表的形式挂接在全局变量 cf->cycle->shared\_memory 下,nginx 此时遍历该链表并逐个进行实际创建,即分配内存、管理机制(比如锁、slab)初始化等:

```
398: Filename : ngx_cycle.c
399:      /* create shared memory */
400:
401:      part = &cycle->shared_memory.part;
402:      shm_zone = part->elts;
403:
404:      for (i = 0; /* void */; i++) {
405: ...
467:          if (ngx_shm_alloc(&shm_zone[i].shm) != NGX_OK) {
468: ...
471:          if (ngx_init_zone_pool(cycle, &shm_zone[i]) != NGX_OK) {
472: ...
475:          if (shm_zone[i].init(&shm_zone[i], NULL) != NGX_OK) {
476: ...
477:      }
```

其中函数 ngx\_shm\_alloc()是共享内存的实际分配,针对当前系统可提供接口,可以是 mmap 或 shmget 等;而 ngx\_init\_zone\_pool()函数是共享内存管理机制的初始化,因为共享内存的使用涉及到另外两个主题,第一,既然是共享内存,那么必然是多进程共同使用,所以必须考虑互斥问题;第二,nginx 既以性能著称,那么对于共享内存自然也有其独特的使用方式,虽然我们可以不用(在马上要介绍到的 init 回调函数里做覆盖处理即可),但在这里也默认都会以这种 slab 的高效访问机制进行初始化。关于这两点,这里暂且略过,待后续再做讨论。

回调函数 shm\_zone[i].init()是各个共享内存所特定的,根据使用方的自身需求不同而不同,这也是我们在使用共享内存时需特别注意的函数。继续看实例 ngx\_http\_limit\_req\_module 模块的 init 函数 ngx\_http\_limit\_req\_init\_zone():

```
398: Filename : ngx_http_limit_req_module.c
399: static ngx_int_t
```

```

400: ngx_http_limit_req_init_zone(ngx_shm_zone_t *shm_zone, void *data)
401: {
402:     ngx_http_limit_req_ctx_t  *octx = data;
403: ...
398:     if (octx) {
399: ...
608:         ctx->shpool = octx->shpool;
609: ...
608:         return NGX_OK;
609:     }
610:
611:     ctx->shpool = (ngx_slab_pool_t *) shm_zone->shm.addr;
612: ...
608:     ctx->sh = ngx_slab_alloc(ctx->shpool, sizeof(ngx_http_limit_req_shctx_t));
609: ...

```

函数 `ngx_http_limit_req_init_zone()` 的第二个参数 `data` 表示‘旧’数据，在进行重新加载配置时（即 `nginx` 收到 `SIGHUP` 信号）该值将不为空，如果旧数据可继续使用，那么可直接返回 `NGX_OK`；否则，需根据自身模块逻辑对共享内存的使用做相关初始化，比如 `ngx_http_limit_req_module` 模块，在第 634、642 行直接使用默认已初始化好的 `slab` 机制，进行内存的分配等。当函数 `ngx_http_limit_req_init_zone()` 正确执行结束，一个完整的共享内存就已创建并初始完成，接着要做的就是共享内存的使用，这即回到前面提到的两个主题：互斥与 `slab`。

要解决互斥问题，无非就是利用锁机制，强制同一时刻只能有一个进程在访问共享内存，其基本原理就是利用共享的简单资源（比如最简单的原子变量）来代表复杂资源，一个进程在需要操作复杂资源之前先获得对简单资源的使用权限；因为简单资源足够简单，对它的使用权限的获取往往只有一步或几步，所以更容易避免冲突；这个应该是容易理解的，比如一个需要 100 步的操作肯定比一个只需要 3 步的操作更容易发生冲突（每一步需要的复杂度相同），因为前一种情况可能会一个进程在进行了 99 步后却因另外一个进程发出动作而失败，而后一种情况的进程执行完 3 步后就已经获得完全使用权限了。

要讲清楚 `nginx` 互斥锁的实现，如果不结合具体的代码恐怕是不行的，因为都是一些细节上的考量，比如根据不同的 CPU 架构选择不同的汇编指令、使用不同的共享简单资源（原子变量或文件描述符），并没有什么特别难以理解的地方，查 CPU 手册和系统 `man` 手册很容易懂，所以具体实现这里暂且不讲，还好 `nginx` 互斥锁的使用非常简单，提供的接口函数以及含义如下：

函数	含义
<code>ngx_shmtx_create()</code>	创建
<code>ngx_shmtx_destory()</code>	销毁
<code>ngx_shmtx_trylock()</code>	尝试加锁（加锁失败则直接返回，不等待）

<code>ngx_shmtx_lock()</code>	加锁（持续等待，直到加锁成功）
<code>ngx_shmtx_unlock()</code>	解锁
<code>ngx_shmtx_force_unlock()</code>	强制解锁（可对其它进程进行解锁）
<code>ngx_shmtx_wakeup()</code>	唤醒等待加锁进程（系统支持信号量的情况下才可用）

## slab 机制

nginx 的 slab 机制与 linux 的 slab 机制在基本原理上并没有什么特别大的不同（当然，相比而言，linux 的 slab 机制要复杂得多），简单来说也就是基于两点：缓存与对齐。缓存意味着预分配，即提前申请好内存并对内存做好划分形成内存池，当我们需要使用一块内存空间时，nginx 就直接从已经申请并划分好的内存池里取出一块合适大小的内存即可，而内存的释放也是把内存返还给 nginx 的内存池，而不是操作系统；对齐则意味着内存的申请与分配总是按 2 的幂次方进行，即内存大小总是为 8、16、32、64 等，比如，虽然只申请 33 个字节的内存，但也将获得实际 64 字节可用大小的内存，这的确存在一些内存浪费，但对于内存性能的提升是显著的（关于内存对齐对性能的影响，可以参考：<http://lenky.info/?p=310>），更重要的是把内部碎片也掌握在可控的范围内。

nginx 的 slab 机制主要是和共享内存一起使用，前面提到对于共享内存，nginx 在解析完配置文件，把即将使用的共享内存全部以 list 链表的形式组织在全局变量 `cf->cycle->shared_memory` 下之后，就会统一进行实际的内存分配，而 nginx 的 slab 机制要做的就是对这些共享内存进行进一步的内部划分与管理，关于这点，从函数 `ngx_slab_init()` 的逻辑即可初见端倪，不过在此之前，先看看 `ngx_init_zone_pool()` 函数对它的调用：

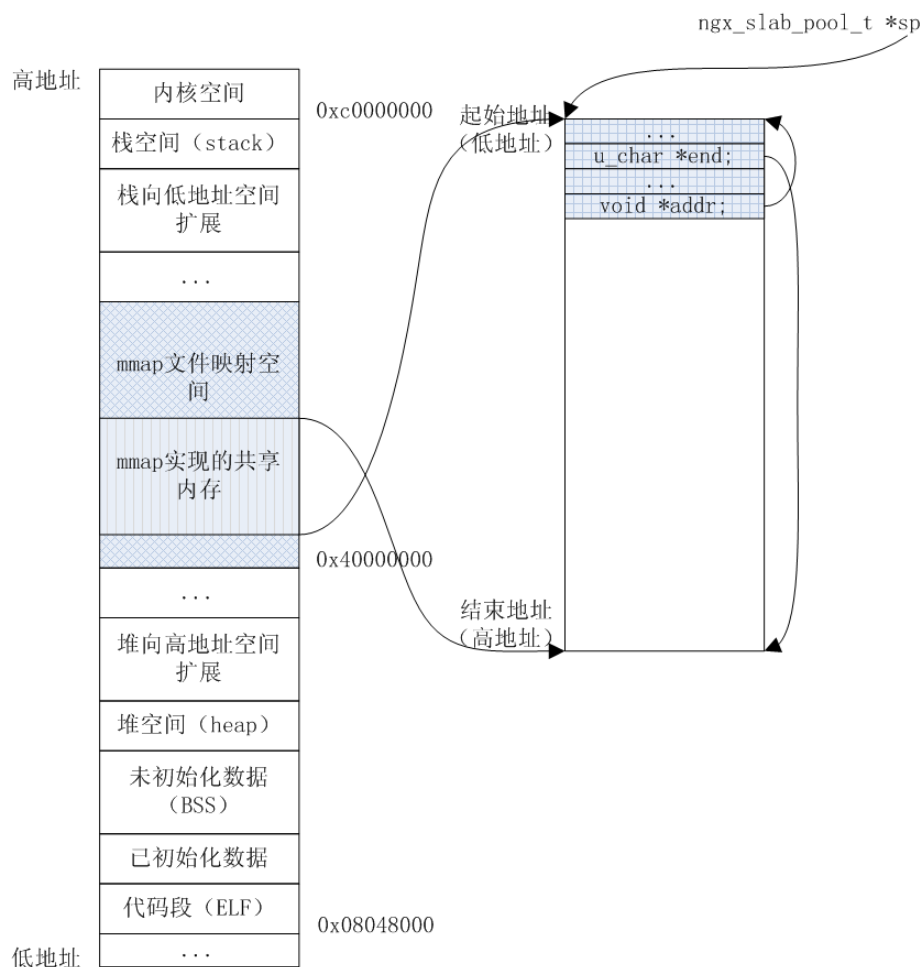
```

916: Filename : ngx_slab.c
917: static ngx_int_t
918: ngx_init_zone_pool(ngx_cycle_t *cycle, ngx_shm_zone_t *zn)
919: {
920:     u_char          *file;
921:     ngx_slab_pool_t  *sp;
922:
923:     sp = (ngx_slab_pool_t *) zn->shm.addr;
924: ...
937:     sp->end = zn->shm.addr + zn->shm.size;
938:     sp->min_shift = 3;
939:     sp->addr = zn->shm.addr;
940: ...
960:     ngx_slab_init(sp);
961: ...

```

函数 `ngx_init_zone_pool()` 是在共享内存分配好后进行的初始化调用，而该函数内又调用

了本节介绍的重点对象 slab 的初始化函数 ngx\_slab\_init();, 此时的情况图示如下:



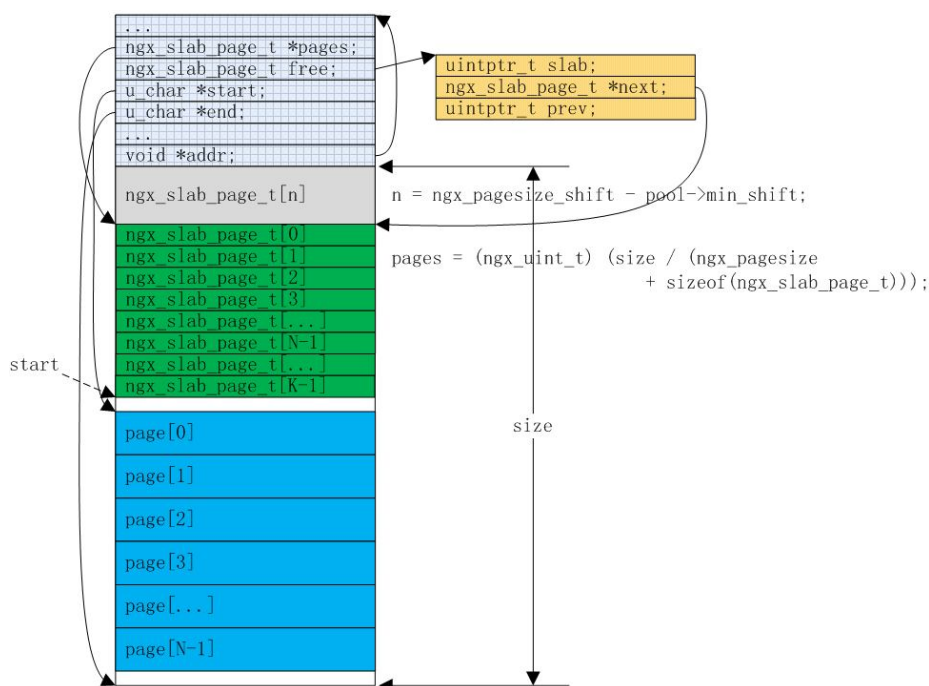
可以看到此时共享内存的开始部分内存已经被用作结构体 ngx\_slab\_pool\_t 的存储空间, 这相当于是 slab 机制的额外开销 (overhead), 后面还会看到其他额外开销, 任何一种管理机制都有自己的一些控制信息需要存储, 所以这些内存使用是无法避免的。共享内存剩下的部分才是被管理的主体, slab 机制对这部分内存进行两级管理, 首先是 page 页, 然后是 page 页内的 slab 块 (通过 slot 对相等大小的 slab 块进行管理, 为了区分 slab 机制, 下面以 slot 块来指代这些 slab 块), 也就是说 slot 块是在 page 页内存的再一次管理。

在继续对 slab 机制分析之前, 先看看下面这个表格里记录的一些变量以及其对应的值, 因为它们可以帮助我们对后面内容的理解。这些变量会根据系统环境的不同而不同, 但一旦系统环境确定, 那么这些值也就将都是一些常量值, 下面表格基于的系统环境在本书最开始有统一介绍, 这里不再累述:

变量名	值	描述
ngx_pagesize	4096	系统内存页大小, Linux 下一般情况就是 4KB。
ngx_pagesize_shift	12	对应 ngx_pagesize (4096), 即是 $4096 = 1 \ll 12$ 。
ngx_slab_max_size	2048	slots 分配和 pages 分配的分割点, 大于等于该值则需从 pages 里分配。

ngx_slab_exact_size	128	正好能用一个 <code>uintptr_t</code> 类型的位图变量表示的页划分；比如在 4KB 内存页、32 位系统环境下，一个 <code>uintptr_t</code> 类型的位图变量最多可以对应表示 32 个划分块的状态，所以要恰好完整的表示一个 4KB 内存页的每一个划分块状态，必须把这个 4KB 内存页划分为 32 块，即每一块大小为： $\text{ngx\_slab\_exact\_size} = 4096 / 32 = 128$ 。
ngx_slab_exact_shift	7	对应 <code>ngx_slab_exact_size</code> (128)，即是 $128 = 1 \ll 7$ 。
pool->min_shift	3	固定值为 3。
pool->min_size	8	固定值为 8，最小划分块大小，即是 $1 \ll \text{pool->min\_shift}$ 。

好，再来看 slab 机制对 page 页的管理，初始结构示意图如下：



slab 机制对 page 页的静态管理主要体现在 `ngx_slab_page_t[K]` 和 `page[N]` 这两个数组上，需要解释几点：

第一，虽然是一个页管理结构（即 `ngx_slab_page_t` 元素）与一个 page 内存页相对应，但因为对齐消耗以及 slot 块管理结构体的占用（图中的 `ngx_slab_page_t[n]` 数组），所以实际上页管理结构体数目比 page 页内存数目要多，即图中的 `ngx_slab_page_t[N]` 到 `ngx_slab_page_t[K-1]`，这些结构体完全被忽视，我们也不用去管它们，只是需要知道有这些东西的存在。

第二，如何根据页管理结构 page 获得对应内存页的起始地址 p？计算方法如下：

384: Filename : ngx\_slab.c

385:  $p = (\text{page} - \text{pool->pages}) \ll \text{ngx\_pagesize\_shift}$ ;

386:  $p += (\text{uintptr\_t}) \text{pool->start}$ ;

对照前面图示来看这很明显，无需过多解释；相反，根据内存页的起始地址 p 也能计算



出其对应的页管理结构 page。

第三，对齐是指实际 page 内存页按 ngx\_pagesize 大小对齐，从图中看就是原本的 start 是那个虚线箭头所指的位置，对齐后就是实线箭头所指的位置，对齐能提高对内存页的访问速度，但这有一些内存浪费，并且末尾可能因为不够一个 page 内存页而被浪费掉，所以在 ngx\_slab\_init()函数的最末尾有一次最终可用内存页的准确调整：

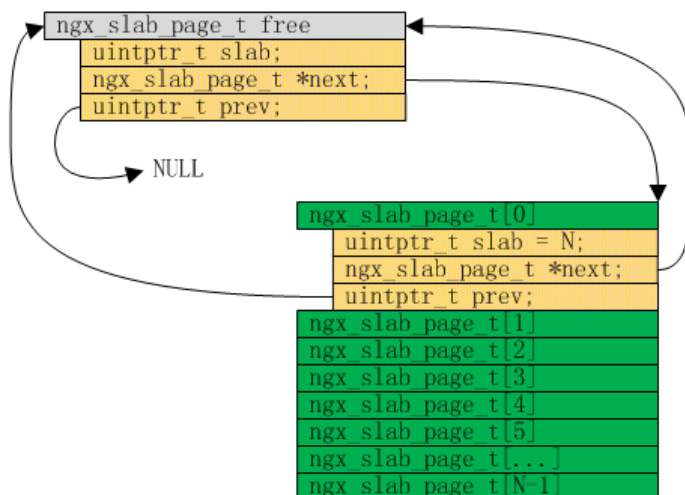
```

75: Filename : ngx_cycle.c
76: void
77: ngx_slab_init(ngx_slab_pool_t *pool)
78: {
79: ...
130:     m = pages - (pool->end - pool->start) / ngx_pagesize;
131:     if(m > 0) {
132:         pages -= m;
133:         pool->pages->slab = pages;
134:     }
135: ...

```

第 130 行计算的 m 值如果大于 0，说明对齐等操作导致实际可用内存页数减少，所以后面的 if 语句进行判断调整。

page 页的静态管理结构基本就是如此了，再来看 page 页的动态管理，即 page 页的申请与释放，这就稍微麻烦一点，因为一旦 page 页被申请或释放，那么就有了相应的状态：使用或空闲。先看空闲页的管理，nginx 对空闲 page 页进行链式管理，链表的头节点 pool->free，初始状态下的链表情况如下：



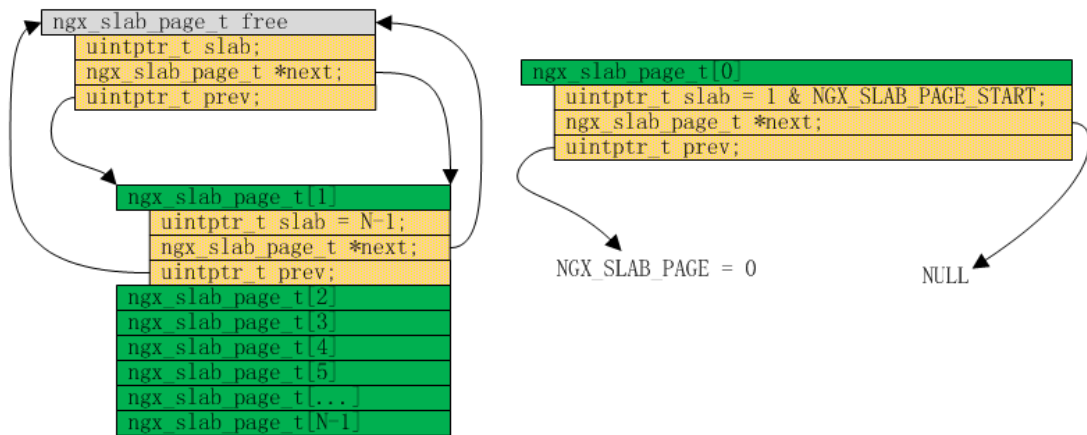
这是一个有点特别的链表，它的节点可以是一个数组，比如上图中的 ngx\_slab\_page\_t[N] 数组就是一个链表节点，这个数组通过第 0 号数组元素，即 ngx\_slab\_page\_t[0]，接入到这个空闲 page 页链表内，并且整个数组的元素个数也记录在这个第 0 号数组元素的 slab 字段内。

如果经历如下几步内存操作：子进程 1 从共享内存中申请 1 页，子进程 2 接着申请了 2

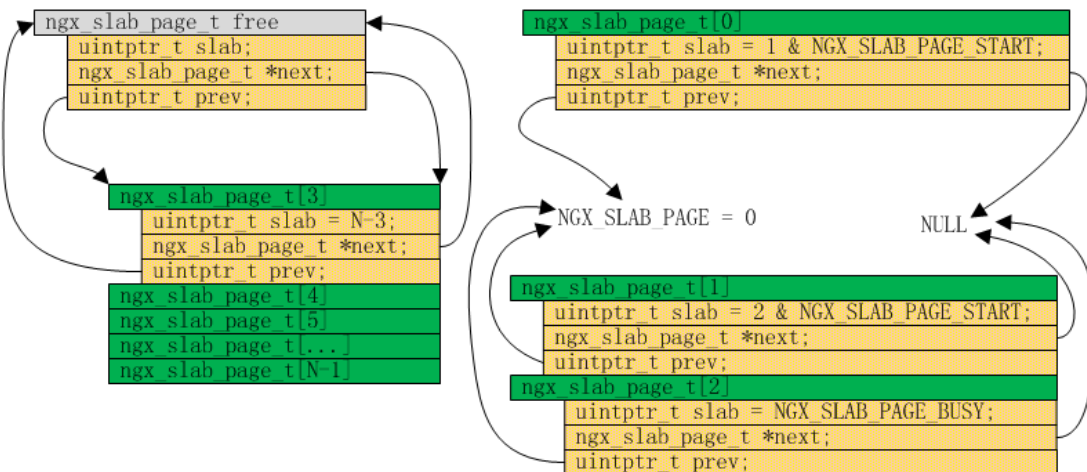


页，然后子进程 1 又释放掉刚申请的 1 页，那么空闲链表各是一个什么状态呢？逐步来看。

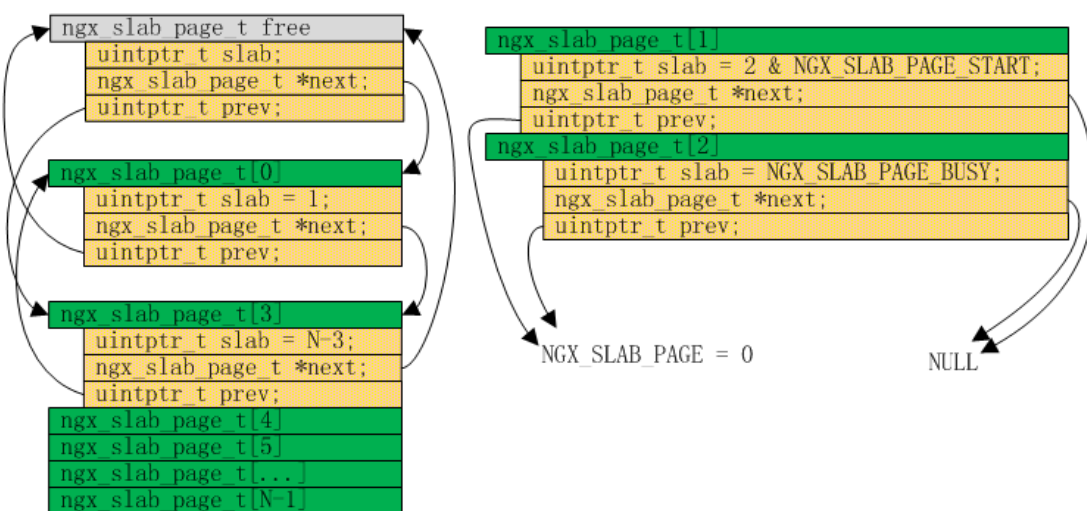
子进程 1 从共享内存中申请 1 页：



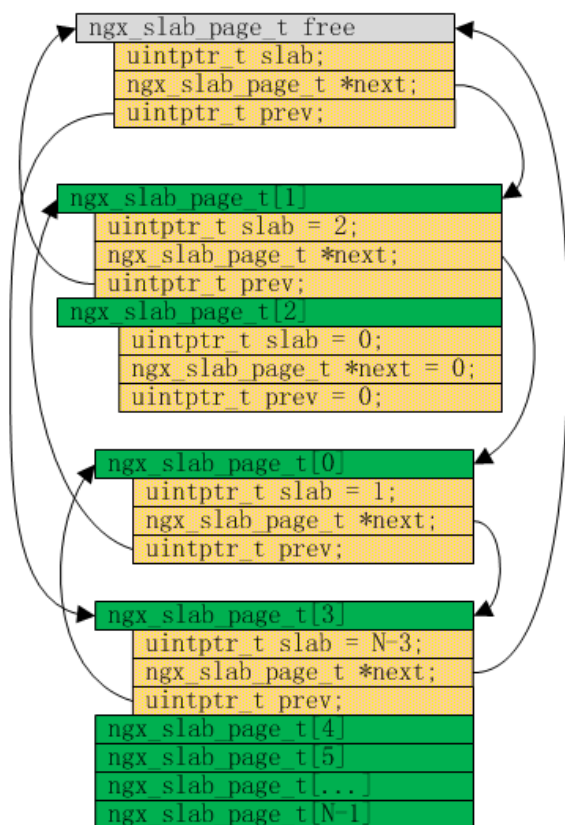
子进程 2 接着申请了 2 页：



然后子进程 1 又释放掉刚申请的 1 页：



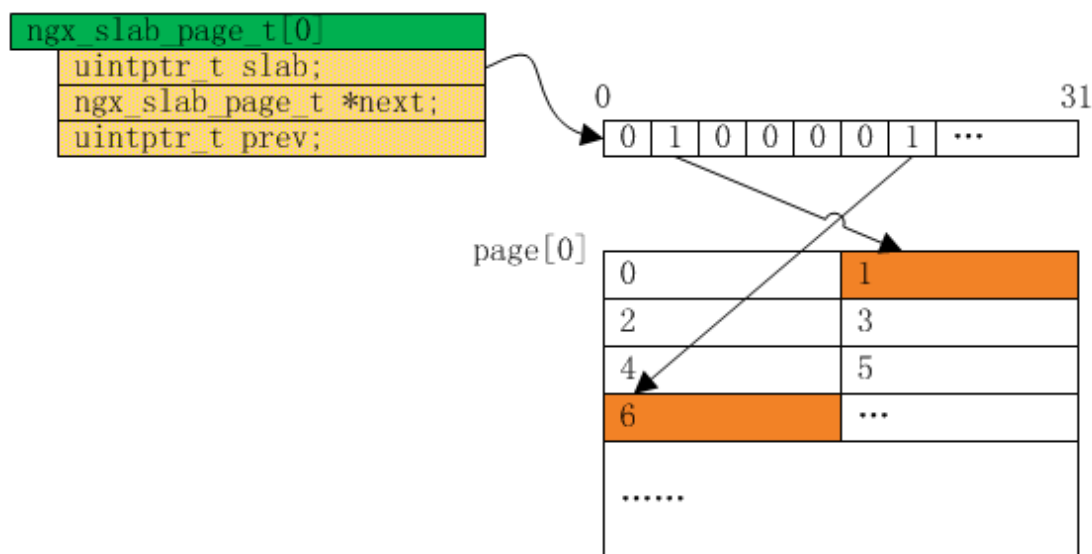
释放的 page 页被插入到链表头部，如果子进程 2 接着释放其拥有的那 2 页内存，那么空闲链表结构将如下图所示：



可以看到，nginx 对空闲 page 页的链式管理不会进行节点合并，不过关系不大，毕竟 page 页既不是 slab 机制的最小管理单元，也不是其主要分配单元。对处于使用状态中的 page 页，也是采用的链式管理，在介绍其详细之前，需先来看看 slab 机制的第二级管理机制，即 slot 块，这样便于前后的连贯理解。

slot 块是对每一页 page 内存的内部管理，它将 page 页划分成很多小块，各个 page 页的 slot 块大小可以不相等，但同一个 page 页的 slot 块大小一定相等。page 页的状态通过其所在的链表即可辨明，而 page 页内各个 slot 块的状态却需要一个额外的标记，在 nginx 的具体实现里采用的是位图方式，即一个 bit 位标记一个对应 slot 块的状态，1 为使用，0 为空闲。

根据 slot 块的大小不同，一个 page 页可划分的 slot 块数也不同，从而需要的位图大小也不一样。前面提到过，每一个 page 页对应一个名为 ngx\_slab\_page\_t 的管理结构，该结构体有一个 uintptr\_t 类型的 slab 字段。在 32 位平台上（也就是本书讨论的设定平台），uintptr\_t 类型占 4 个字节，即 slab 字段有 32 个 bit 位。如果 page 页划分的 slot 块数小于等于 32，那么 nginx 直接利用该字段充当位图，这在 nginx 内叫 exact 划分，每个 slot 块的大小保存在全局变量 ngx\_slab\_exact\_size 以及 ngx\_slab\_exact\_shift 内。比如，1 个 4KB 的 page 页，如果每个 slot 块大小为 128 字节，那么恰好可划分成 32 块。下图是这种划分下的一种可能的中间情况：



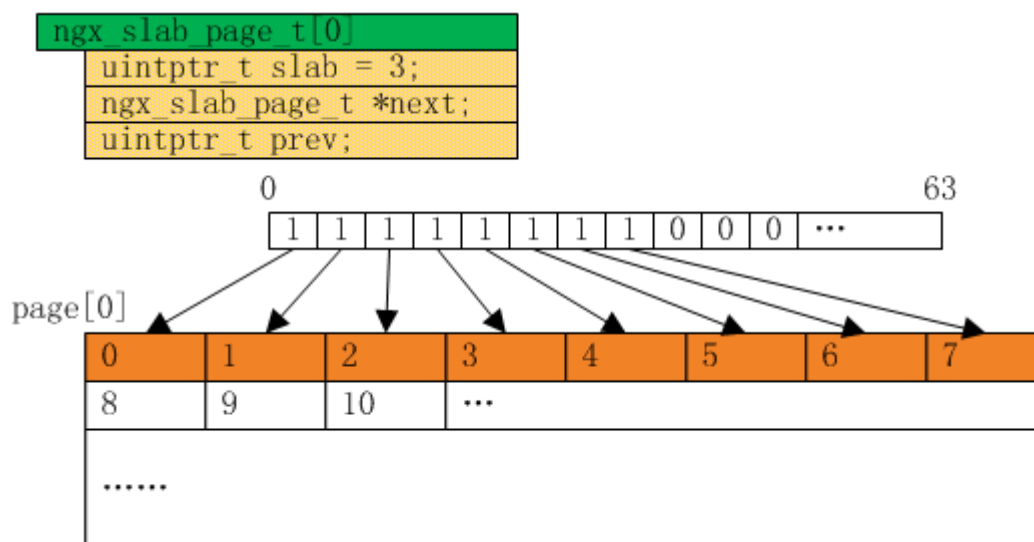
如果划分的每个 slot 块比 ngx\_slab\_exact\_size 还大，那意味着一个 page 页划分的 slot 块数更少，此时当然也是使用 ngx\_slab\_page\_t 结构体的 slab 字段作为位图。由于比 ngx\_slab\_exact\_size 大的划分可以有很多种，所以需要把其具体的大小也记录下来，这个值同样也记录在 slab 字段里。这样做是可行的，由于划分总是按 2 次幂增长，所以比 ngx\_slab\_exact\_size 还大的划分至少要减少一半的 slot 块数，因此利用 slab 字段的一半 bit 位即可完整表示所有 slot 块的状态。具体点说就是：slab 字段的高端 bit 用作位图，低端 bit 用于存储 slot 块大小（仅存其对应的移位数）。代码表现为：

378: Filename : ngx\_slab.c

379: page->slab = ((uintptr\_t) 1 << NGX\_SLAB\_MAP\_SHIFT) | shift;

如果申请的内存大于等于 ngx\_slab\_max\_size，nginx 直接返回一个 page 整页，此时已经不在 slot 块管理里，所有无需讨论。下面来看小于 ngx\_slab\_exact\_size 的情况，此时 slot 块数目已经超出了 slab 字段可表示的容量。比如假设按 8 字节划分，那么 1 个 4KB 的 page 页将被划分为 512 块，表示各个 slot 块状态的位图也就需要 512 个 bit 位，一个 slab 字段明显是不足够的，所以需要为位图另找存储空间，而 slab 字段仅用于存储 slot 块大小（仅存其对应的移位数）。

另找的位图存储空间就落在 page 页内，具体点说是其划分的前面几个 slot 块内。接着刚才说的例子，512 个 bit 位的位图，即 64 个字节，而一个 slot 块有 8 个字节，所以需要占用 page 页的前 8 个 slot 块用作位图。即，一个按 8 字节划分 slot 块的 page 页初始情况如下图所示：



由于前几个 slot 块一开始就被用作位图空间，所以必须把它们对应的 bit 位设置为 1，表示其状态为使用。

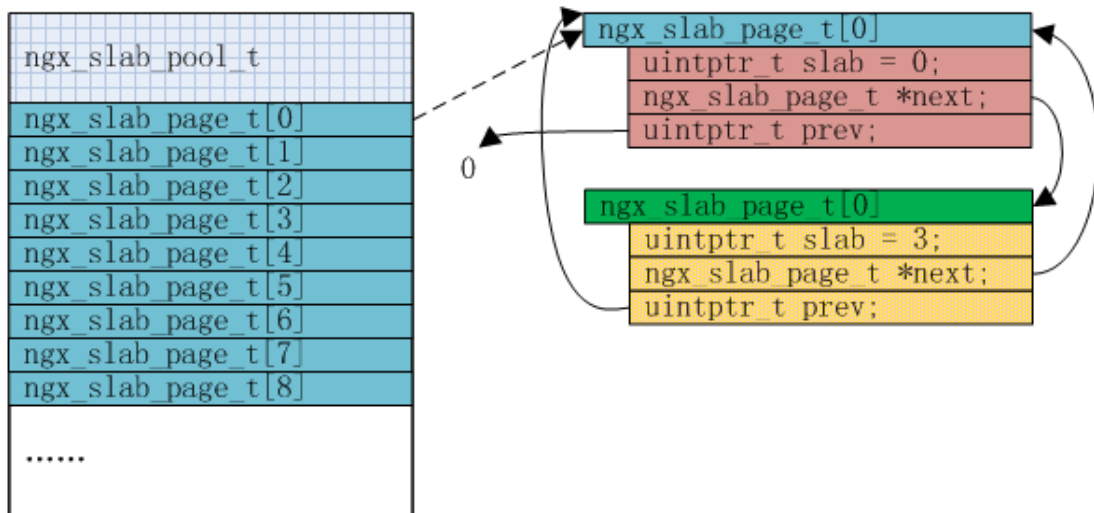
不论哪种情况，都有了 slot 块的大小以及状态，那对 slot 块的分配与释放就水到渠成了。下面回到 slab 机制的最后一个话题，即对处于使用状态中的 page 页的链式管理。其实很简单，首先，根据每页划分的 slot 块大小，将各个 page 页加入到不同的链表内。在我们这里设定的平台上，也就是按 8、16、32、64、128、256、512、1024、2048 一共 9 条链表，在 ngx\_slab\_init() 函数里有其初始化：

```
102: Filename : ngx_slab.c
103:     n = ngx_pagesize_shift - pool->min_shift;
104:
105:     for (i = 0; i < n; i++) {
106:         slots[i].slab = 0;
107:         slots[i].next = &slots[i];
108:         slots[i].prev = 0;
109:     }
```

假设申请一块 8 字节的内存，那么 slab 机制将分配一共 page 页，将它按 8 字节做 slot 划分，并且接入到链表 slots[0] 内，相关示例（表示这只是其中一处实现）代码：

```
352: Filename : ngx_slab.c
353:         page->slab = shift;
354:         page->next = &slots[slot];
355:         page->prev = (uintptr_t) &slots[slot] | NGX_SLAB_SMALL;
356:
357:         slots[slot].next = page;
```

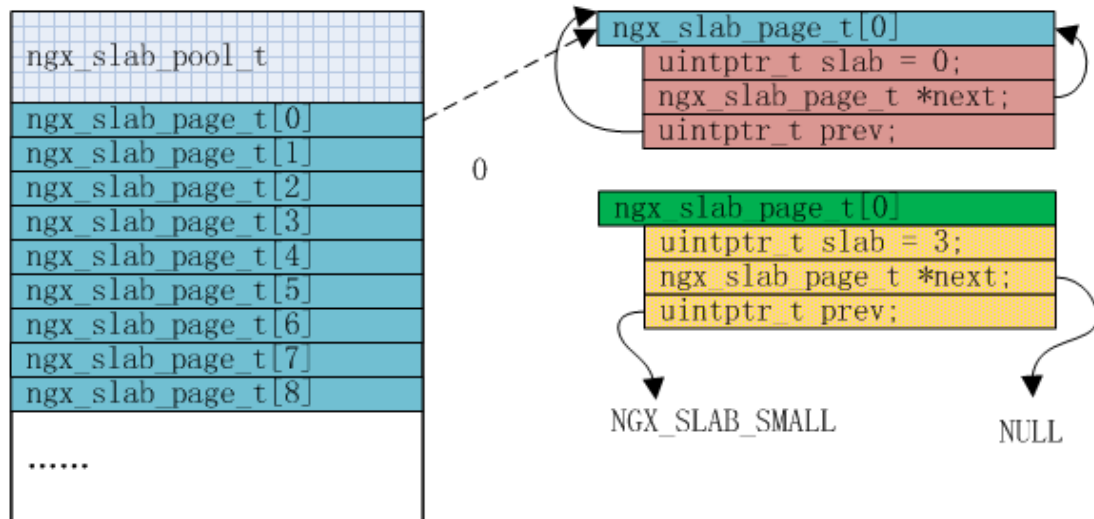
page->prev 按 4 字节对齐，所以末尾两位可以用做它用，这里用于标记当前 slot 划分类型为 NGX\_SLAB\_SMALL，图示如下：



继续申请 8 字节的内存不会分配新的 page 页，除非刚才那页 page（暂且称之为页 A）被全是用完，一旦页 A 被使用完，它会被拆除出链表，相关示例代码：

```
232: Filename : ngx_slab.c
233:         prev = (ngx_slab_page_t *)
234:             (page->prev & ~NGX_SLAB_PAGE_MASK);
235:         prev->next = page->next;
236:         page->next->prev = page->prev;
237:
238:         page->next = NULL;
239:         page->prev = NGX_SLAB_SMALL;
```

第 234 行是过滤掉末尾的标记位，以获得正确的前节点的地址，此时的图示如下：



如果仍然继续申请 8 字节的内存，那么 nginx 的 slab 机制必须分配新的 page 页（暂且称之为页 B），类似于前面介绍的那样，页 B 会被加入到链表内，此时链表中只有一个节点，但如果此时页 A 释放了某个 slot 块，它又会被加入到链表中，终于形成了具有两个节点的链表，相关示例代码（变量 page 指向页 A）以及图示如下：

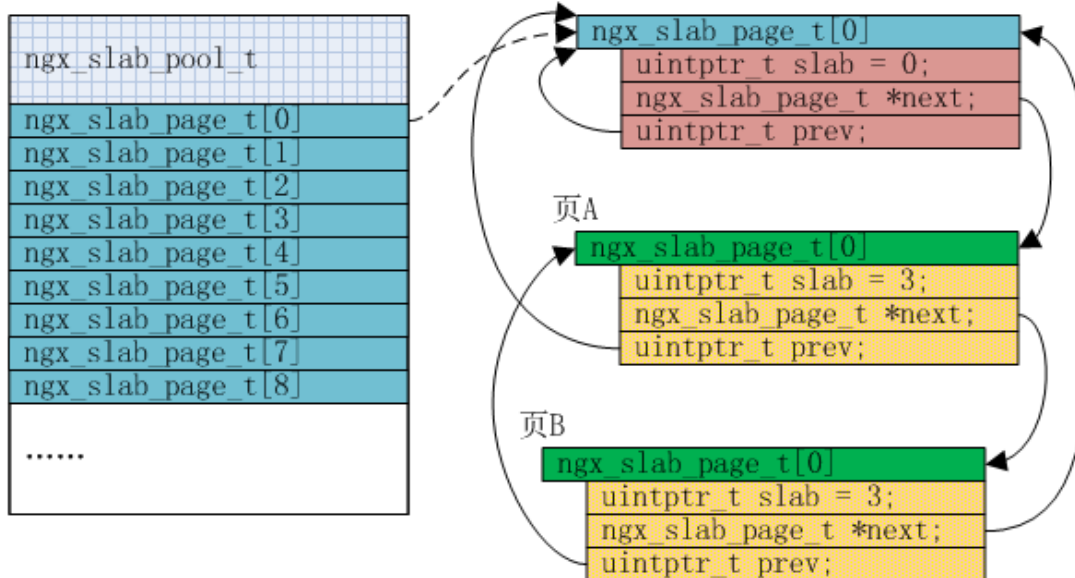
```
455: Filename : ngx_slab.c
```



```

456:     page->next = slots[slot].next;
457:     slots[slot].next = page;
458:
459:     page->prev = (uintptr_t) &slots[slot] | NGX_SLAB_SMALL;
460:     page->next->prev = (uintptr_t) page | NGX_SLAB_SMALL;

```



## 第二章 跟踪调试

### 利用日志信息跟踪

优秀的程序都会带有自己的日志输出接口，并且一般会给出不同等级的输出，以便于重复信息的过滤，比如 Linux 内核的日志输出标准接口为 `printk`，并且给出了 `KERN_EMERG`、`KERN_ALERT`、`KERN_DEBUG` 等这样的输出等级，nginx 与此类似，下面具体来看。

为了获取最丰富的日志信息，我们在编译 nginx 前进行 `configure` 配置时，需要把 `--with-debug` 选项加上，这样能生成一个名为 `NGX_DEBUG` 的宏，而在 nginx 源码内，该宏被用作控制开关，如果没有它，那么很多日志逻辑代码将在 `make` 编译时直接跳过，比如对单连接的 `debug_connection` 调试指令、分模块日志调试 `debug_http` 功能等：

```
00: Filename : ngx_auto_config.h
01: #define NGX_CONFIGURE " --with-debug"
02:
03: #ifndef NGX_DEBUG
04: #define NGX_DEBUG 1
05: #endif

620: Filename : nginx.c
621: #if (NGX_DEBUG)
622:     {
623:         char **e;
624:         for (e = env; *e; e++) {
625:             ngx_log_debug1(NGX_LOG_DEBUG_CORE, cycle->log, 0, "env: %s", *e);
626:         }
627:     }
628: #endif
```

有了上面这个编译前提条件之后，我们还想在配置文件里做恰当的设置，关于这点 nginx 提供的主要配置指令为 `error_log`，该配置项的默认情况（默认值定义在 `objs/ngx_auto_config.h` 文件内）为：

```
error_log logs/error.log error;
```

表示日志信息记录在 `logs/error.log`（如果没改变 nginx 的工作路径的话，那么默认父目录为 `/usr/local/nginx/`）文件内，而日志记录级别为 `error`。

在实际进行配置时，可以修改日志信息记录文件路径（比如修改为 `/dev/null`，此时所有日志信息将被输出到所谓的 linux 黑洞设备）或直接输出到标准终端（此时指定为 `stderr`），而 nginx 提供的日志记录级别一共有 8 级，等级从低到高分别为 `debug`、`info`、`notice`、`warn`、`error`、`crit`、`alert`、`emerg`，如果设置为 `error`，则表示 nginx 内等级为 `error`、`crit`、`alert`、`emerg` 的 4 种日志将被输出到日志文件或标准终端，另外的 `debug`、`info`、`notice`、`warn` 这 4 种日志



将被直接过滤掉而不会输出,因此如果我们只关注特别严重的信息则只需将日志等级设置为 `emerg` 即可大大减少 `nginx` 的日志输出量,这样就避免了在大量的日志信息里寻找重要信息的麻烦。

当我们利用日志跟踪 `nginx` 时,需要获取最大量的日志信息,所以此时可以把日志等级设置为最低的 `debug` 级,在这种情况下,如果觉得调试日志太多,`nginx` 提供按模块控制的更细粒等级: `debug_core`、`debug_alloc`、`debug_mutex`、`debug_event`、`debug_http`、`debug_imap`,比如如果只想看 `http` 的调试日志,则需这样设置:

```
error_log logs/error.log debug_http
```

此时 `nginx` 将输出从 `info` 到 `emerg` 所有等级的日志信息,而 `debug` 日志则将只输出与 `http` 模块相关的内容。

`error_log` 配置指令可以放在配置文件的多个上下文内,比如 `main`、`http`、`server`、`location`,但同一个上下文中只能设置一个 `error_log`,否则 `nginx` 将提示类似如下这样的错误:

```
nginx: [emerg] "error_log" directive is duplicate in /usr/local/nginx/conf/nginx.conf:9
```

但在不同的配置文件上下文里可以设置各自的 `error_log` 配置指令,通过设置不同的日志文件,这是 `nginx` 提供的又一种信息过滤手段:

```
00: Filename : example.conf
01: ...
02: error_log logs/error.log error;
03: ...
04: http {
05:     error_log logs/http.log debug;
06:     ...
07:     server {
08:         ...
09:         error_log logs/server.log debug;
10:     }
```

`nginx` 提供的另一种更有针对性的日志调试信息记录是针对特定连接的,这通过 `debug_connection` 配置指令来设置,比如如下设置调试日志仅针对 `ip` 地址 `192.168.1.1` 和 `ip` 段 `192.168.10.0/24`:

```
events {
    debug_connection 192.168.1.1;
    debug_connection 192.168.10.0/24;
}
```

`nginx` 的日志功能仍在不断的改进,如能利用得好,对于我们跟着 `nginx` 还是非常有帮助的,至少我知道有不少朋友十分习惯于 `c` 库的 `printf` 打印调试,相比如此,`nginx` 的 `ngx_log_xxx` 要强大得多。

## 利用 gdb 调试

一般来说，默认./configure 生成的 makefile 文件都将带上-g 选项，这对于利用 gdb 调试 nginx 是非常必要的，但如果在使用 gdb 调试 nginx 时提示 “No symbol table is loaded. Use the "file" command.”，则需检查 objs/Makefile 文件里的 CFLAGS 标记是否带上了-g 选项；另一个值得关注的编译选项是-O0，如果在 gdb 内打印变量提示 “<value optimized out>” 或 gdb 提示的当前正执行的代码行与源代码匹配不上而让人感觉莫名其妙，则多半是因为 gcc 优化导致，我们可以加上-O0 选项强制禁用 gcc 的编译优化。如何把 “-g -O0” 应用在 nginx 上可以有三种方法：

1， 在进行 configure 配置时，按如下方式执行：

```
[root@localhost nginx-1.2.0]# CFLAGS="-g -O0" ./configure
```

2， 直接修改文件 objs/Makefile 给其加上 “-g -O0”

3， 在执行 make 时，按如下方式执行：

```
[root@localhost nginx-1.2.0]# make CFLAGS="-g -O0"
```

第 2、3 两种方法是在我们已经执行 configure 之后进行的，如果之前已经执行过 make，那么还需刷新所有源文件的时间戳，以便重新编译 nginx：

```
[root@localhost nginx-1.2.0]# find . -name "*.c" | xargs touch
```

nginx 默认以 daemon 形式运行，并且默认包含有监控进程和多个工作进程，所以如果要直接在 gdb 内执行 nginx 并让 gdb 捕获 nginx 的监控，则需要在 nginx 的配置文件里做如下设置：

```
daemon off;
```

这样 nginx 不再是 daemon 进程，此时利用 gdb 可以从 nginx 的 main()函数开始调试，默认情况下调试的是监控进程的流，如果要调试工作进程的流需要在进入 gdb 后执行 set follow-fork-mode child（此时，上面的 daemon 配置可不改），更简单的方法是直接设置 master\_process off，将监控进程逻辑和工作进程逻辑全部合在一个进程里调试。这些设置对于调试像配置信息解析流程这一类初始逻辑是非常重要的，因为 nginx 的这些逻辑是在 nginx 启动时进行的。总之，不管怎么样做，你都必须让 gdb attach 到你想要调试的进程上，需特别注意 fork()这样的函数调用，因为一调用该函数，程序就一分为二，而 gdb 默认是继续 attach 父进程，如果父进程后续动作是直接退出（比如 ngx\_daemon()函数），那么就导致 gdb 跟丢了。

gdb 带参数运行 nginx 有很多种方法，比如：Shell 里执行 gdb --args ./objs/nginx -c /usr/local/nginx/conf/nginx.conf，进入到 gdb 后 r 即可；或者 Shell 里执行 gdb ./objs/nginx，进入到 gdb 后执行 r -c /usr/local/nginx/conf/nginx.conf，等等。

而另外的一般情况下，如果我们调试的是 nginx 的中间执行主过程，那么我们可以先执行 nginx，然后根据 nginx 进程的进程号进行 gdb 绑定来做调试。首先，需要找到对应的进

程号:

```
[root@localhost ~]# ps -efH | grep nginx
root      3971 24701  0 12:20 pts/4    00:00:00      grep nginx
root      3905      1  0 12:16 ?        00:00:00    nginx: master process ./nginx
nobody    3906   3905  0 12:16 ?        00:00:00    nginx: worker process
nobody    3907   3905  0 12:16 ?        00:00:00    nginx: worker process
[root@localhost ~]#
```

nginx 代码还给 nginx 进程加上了 title, 所以根据标题很容易区分出哪个是监控进程, 哪些个是工作进程。对工作进程 3906 的 gdb 调试, 可以利用 gdb 的 -p 命令行参数:

```
[root@localhost ~]# gdb -p 3906
```

或者是在进入 gdb 后执行:

```
(gdb) attach 3906
```

都可以。如果是要调试客户端发过来的请求处理过程, 那么要注意请求是否被交给另外一个工作进程处理而导致绑定到 gdb 的这个工作进程实际没有动作, 此时可以考虑开两个终端, 运行两个 gdb 或干脆修改配置项 worker\_processes 值为 1 而只运行一个工作进程。

将 nginx 特定进程绑定到 gdb 后, 剩余的调试操作无非就是 gdb 的使用, 这可以参考官方手册 (<http://www.gnu.org/software/gdb/documentation/>), 手册内容很多, 因为 gdb 提供的功能非常丰富, 而某些功能对于我们调试 nginx 也大有帮助, 像 Break conditions、Watchpoints 等。以 Watchpoints (监视点) 为例, 它可以监视某个变量在什么时候被修改, 这对于我们了解 nginx 的程序逻辑是非常有帮助的, 比如在理解 nginx 的共享内存逻辑时, 看到 ngx\_shared\_memory\_add() 函数内初始化的 shm\_zone->init 回调为空:

```
1256:  Filename : ngx_cycle.c
1257:  ngx_shm_zone_t *
1258:  ngx_shared_memory_add(ngx_conf_t *cf, ngx_str_t *name, size_t size, void *tag)
1259:  {
1260:  ...
1318:      shm_zone->init = NULL;
1319:  ...
```

而在 ngx\_init\_cycle() 函数里对该回调函数却是直接执行而并没有做前置判空处理:

```
41:  Filename : ngx_cycle.c
42:  ngx_cycle_t *
43:  ngx_init_cycle(ngx_cycle_t *old_cycle)
44:  {
45:  ...
475:      if (shm_zone[i].init(&shm_zone[i], NULL) != NGX_OK) {
476:          goto failed;
477:      }
478:  ...
```

这就说明这个函数指针一定是在其它某处被再次赋值, 但具体是在哪里呢? 搜索 nginx

全部源代码可能一下子没找到对应的代码行，那么此时可利用 gdb 的 Watchpoints 功能进行快速定位：

```
(gdb) b ngx_cycle.c:1318
Breakpoint 1 at 0x805d7ce: file src/core/nginx_cycle.c, line 1318.
(gdb) r
Starting          program:          /home/gqk/nginx-1.2.0/objs/nginx      -c
/usr/local/nginx/conf/nginx.conf.upstream.sharedmem
[Thread debugging using libthread_db enabled]

Breakpoint 1, ngx_shared_memory_add (cf=0xbffff39c, name=0xbffffed8, size=134217728,
tag=0x80dbd80) at src/core/nginx_cycle.c:1318
1318      shm_zone->init = NULL;
Missing separate debuginfos, use: debuginfo-install      glibc-2.12-1.47.el6.i686
nss-softokn-freebl-3.12.9-11.el6.i686      openssl-1.0.0-20.el6.i686      pcre-7.8-3.1.el6.i686
zlib-1.2.3-27.el6.i686
(gdb) p &shm_zone->init
$1 = (ngx_shm_zone_init_pt *) 0x80eba68
(gdb) watch *(ngx_shm_zone_init_pt *) 0x80eba68
Hardware watchpoint 2: *(ngx_shm_zone_init_pt *) 0x80eba68
(gdb) c
Continuing.
Hardware watchpoint 2: *(ngx_shm_zone_init_pt *) 0x80eba68

Old value = (ngx_shm_zone_init_pt) 0
New value = (ngx_shm_zone_init_pt) 0x809d9c7 <ngx_http_file_cache_init>
ngx_http_file_cache_set_slot      (cf=0xbffff39c,      cmd=0x80dc0d8,      conf=0x0)      at
src/http/nginx_http_file_cache.c:1807
1807      cache->shm_zone->data = cache;
```

在 `shm_zone->init = NULL;` 代码对应的第 1318 行先下一个 Breakpoint，执行 nginx 后将在此处停止程序，通过 p 打印获取 `shm_zone->init` 的地址值，然后直接给 `shm_zone->init` 对应的地址下 Breakpoint 进行监视，这样即便是跑出 `shm_zone->init` 变量所在的作用域也没有关系，执行 c 命令继续执行 nginx，一旦 `shm_zone->init` 被修改，那么就停止在进行修改的代码的下一行，Old value 和 New value 也被 gdb 抓取出来，可以看到修改逻辑在第 1806 行（我这里是 proxy\_cache 所用的共享内存作为示例，而在其它实例情况下，将可能与此不同）：

```
1084:      Filename : ngx_http_file_cache.c
1085:      ...
1086:      cache->shm_zone->init = ngx_http_file_cache_init;
1087:      cache->shm_zone->data = cache;
```

其实，nginx 本身对于 gdb 也有相关辅助支持，这表现在配置指令 `debug_points` 上，对于该配置项的配置值可以是 stop 或 abort。当 nginx 遇到严重错误时，比如内存超限或其他

不可意料的逻辑错误，就会调用 `ngx_debug_point()` 函数（类似于 `assert` 这样的断言，只是函数 `ngx_debug_point()` 本身不带判断），该函数根据 `debug_points` 配置指令的设置做相应的处理。

如果将 `debug_points` 设置为 `stop`，那么 `ngx_debug_point()` 函数的调用将 `nginx` 进程进入到暂停状态，以便我们通过 `gdb` 接入查看相关进程上下文信息：

```
[root@localhost ~]# ps aux | grep nginx
root      4614  0.0  0.0  24044   592 ?        Ts   12:48   0:00 ./nginx
root      4780  0.0  0.1 103152   800 pts/4    S+   13:00   0:00 grep nginx
[root@localhost ~]#
```

注意上面的 `./nginx` 状态为 `Ts`（`s` 代表 `nginx` 进程为一个会话首进程 `session leader`），其中 `T` 就代表 `nginx` 进程处在 `TASK_STOPPED` 状态，此时我们用 `gdb` 连上去即可看到问题所在（我这里只是一个测试，在 `main` 函数里主动调用 `ngx_debug_point()` 而已，所以下面看到的 `bt` 堆栈很简单，实际使用时，我们当然要把该函数放在需要观察的代码点）：

```
[root@localhost ~]# gdb -q -p 4614
Attaching to process 4614
Reading symbols from /usr/local/nginx/sbin/nginx...done.
...
openssl-1.0.0-4.el6.x86_64 pcre-7.8-3.1.el6.x86_64 zlib-1.2.3-25.el6.x86_64
(gdb) bt
#0  0x0000003a9ea0f38b in raise () from /lib64/libpthread.so.0
#1  0x0000000000431a8a in ngx_debug_point () at src/os/unix/nginx_process.c:603
#2  0x00000000004035d9 in main (argc=1, argv=0x7ffbd0a0c08) at src/core/nginx.c:406
(gdb) c
Continuing.
```

Program received signal SIGTERM, Terminated.

执行 `c` 命令，`nginx` 即自动退出。

如果将 `debug_points` 设置为 “`debug_points abort;`”，此时调用 `ngx_debug_point()` 函数将直接 `abort` 崩溃掉，如果对 `OS` 做了恰当的设置，那么将获得对应的 `core` 文件，这就非常方便我们进行事后的慢慢调试，延用上面的直接在 `main` 函数里主动调用 `ngx_debug_point()` 的例子：

```
[root@localhost nginx]# ulimit -c
0
[root@localhost nginx]# ulimit -c unlimited
[root@localhost nginx]# ulimit -c
unlimited
[root@localhost nginx]# ./sbin/nginx
[root@localhost nginx]# ls
client_body_temp  core.5242      html  proxy_temp  scgi_temp
conf              fastcgi_temp  logs  sbin        uwsgi_temp
```

```
[root@localhost nginx]#
生成了名为 core.5242 的 core 文件，利用 gdb 调试该 core 文件：
[root@localhost nginx]# gdb sbin/nginx core.5242 -q
Reading symbols from /usr/local/nginx/sbin/nginx...done.
[New Thread 5242]
...
(gdb) bt
#0  0x0000003a9de329a5 in raise () from /lib64/libc.so.6
#1  0x0000003a9de34185 in abort () from /lib64/libc.so.6
#2  0x0000000000431a92 in ngx_debug_point () at src/os/unix/ngx_process.c:607
#3  0x00000000004035d9 in main (argc=1, argv=0x7fffd5625f18) at src/core/nginx.c:406
(gdb) up 3
#3  0x00000000004035d9 in main (argc=1, argv=0x7fffd5625f18) at src/core/nginx.c:406
406     ngx_debug_point();
(gdb) list
401         }
402     }
403
404     ngx_use_stderr = 0;
405
406     ngx_debug_point();
407
408     if (ngx_process == NGX_PROCESS_SINGLE) {
409         ngx_single_process_cycle(cycle);
410
(gdb)
```

对于调试工具，我很乐意推荐另外一个封装 gdb 的开源工具 cgdb，cgdb 最大的好处是能在终端里运行并且原生具备 gdb 的强大调试功能，关于 cgdb 的相关使用可以参考官网：<http://cgdb.sourceforge.net/>或 lenky 个人网站上的粗略介绍：<http://lenky.info/?p=1409>。

cgdb 在远程 ssh 里执行的界面如下图所示，如果上面类 vi 窗口没有显示对应的代码或下面 gdb 窗口提示 No such file or directory.，那么需要利用 directory 命令把 nginx 源码增加到搜索路径即可：



```

root@localhost:/home/gqk - Xshell 4
File Edit View Tools Window Help
566     ngx_connection_t *c;
567
568     /* NGX_TIMER_INFINITE == INFTIM */
569
570     ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
571                  "epoll timer: %M", timer);
572
573->   events = epoll_wait(ep, event_list, (int) nevents, timer);
574
575   err = (events == -1) ? ngx_errno : 0;
576
577   if (flags & NGX_UPDATE_TIME || ngx_event_timer_alarm) {
578     ngx_time_update();
579   }
/home/gqk/nginx-1.2.0/src/event/modules/nginx_epoll_module.c

Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.47.el6.i686 nss-softokn-freebl-3.12.9-11
.el6.i686 openssl-1.0.0-20.el6.i686 pcre-7.8-3.1.el6.i686 zlib-1.2.3-27.el6.i686
---Type <return> to continue, or q <return> to quit---
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.47.el6.i686 nss-softokn-freebl-3.12.9-11
.el6.i686 openssl-1.0.0-20.el6.i686 pcre-7.8-3.1.el6.i686 zlib-1.2.3-27.el6.i686
(gdb) up
#1  0x00ba3858 in __epoll_wait_nocancel () from /lib/libc.so.6
(gdb) directory /home/gqk/nginx-1.2.0/src
Source directories searched: /home/gqk/nginx-1.2.0/src:$cd:$cwd
(gdb) up
#2  0x0806fd7b in ngx_epoll_process_events (cycle=0x9d936e8, timer=4294967295, flags=3) at src/event/modu
les/nginx_epoll_module.c:573
(gdb)
Connected to 192.168.164.2:22.  SSH2  xterm 105x29 29,7  1 session  CAP NUM

```

## 利用 strace/pstack 调试

Linux 下有两个命令 `strace` (<http://sourceforge.net/projects/strace/>) 和 `ltrace` (<http://www.ltrace.org/>) 可以查看一个应用程序在运行过程中所发起的系统调用，这对作为标准应用程序的 `nginx` 自然可用。由于这两个命令大同小异，所以下面仅以 `strace` 为例做下简单介绍，大致了解一些它能帮助我们获取哪些有用的调试信息，关于 `strace/ltrace` 以及后面介绍的 `pstack` 更多的用法请 Google。

从 `strace` 的 man 手册可以看到几个有用的选项：

- `-p pid`: 通过进程号来指定被跟踪的进程。
- `-o filename`: 将跟踪信息输出到指定文件。
- `-f`: 跟踪其通过 `fork` 调用产生的子进程。
- `-t`: 输出每一个系统调用的发起时间。
- `-T`: 输出每一个系统调用消耗的时间。

首先利用 `ps` 命令查看到系统当前存在的 `nginx` 进程，然后用 `strace` 命令的 `-p` 选项跟踪 `nginx` 工作进程：



```
[root@localhost nginx-1.2.0]# ps aux | grep nginx
root      4032  0.0  0.2  5164  564 ?        Ss   17:12   0:00 nginx: master process objs/nginx -c /usr/local
/nginx/conf/nginx.conf
nobody    4033  0.0  0.3  5336  884 ?        S    17:12   0:00 nginx: worker process
root      4054  0.0  0.2  4328  732 pts/0    S+   17:14   0:00 grep nginx

[root@localhost nginx-1.2.0]# strace -p 4033
Process 4033 attached - interrupt to quit
gettimeofday({1337678055, 808290}, NULL) = 0
epoll_wait(8, [EPOLLIN, {u32=161263400, u64=577736855967805224}], 512, -1) = 1
gettimeofday({1337678253, 140811}, NULL) = 0
accept4(6, {sa_family=AF_INET, sin_port=htons(41096), sin_addr=inet_addr("127.0.0.1")}, [16], SOCK_NONBLOCK) =
3
epoll_ctl(8, EPOLL_CTL_ADD, 3, {EPOLLIN|EPOLLET, {u32=161263592, u64=13813868851468480488}}) = 0
epoll_wait(8, {EPOLLIN, {u32=161263592, u64=13813868851468480488}}, 512, 60000) = 1
gettimeofday({1337678253, 148133}, NULL) = 0
recv(3, "GET / HTTP/1.0\r\nUser-Agent: Wget...", 1024, 0) = 107
stat64("/usr/local/nginx/html/index.html", {st_mode=S_IFREG|0644, st_size=151, ...}) = 0
open("/usr/local/nginx/html/index.html", O_RDONLY|O_NONBLOCK|O_LARGEFILE) = 9
fstat64(9, {st_mode=S_IFREG|0644, st_size=151, ...}) = 0
writev(3, [{"HTTP/1.1 200 OK\r\nServer: nginx/1"...}, 215], 1) = 215
sendfile64(3, 9, [0], 151) = 151
write(4, "127.0.0.1 - - [22/May/2012:17:17:...], 96) = 96
close(9) = 0
setsockopt(3, SOL_TCP, TCP_NODELAY, [1], 4) = 0
recv(3, "", 1024, 0) = 0
close(3) = 0
epoll_wait(8, [
```

为了简化操作，我这里只设定了一个工作进程，该工作进程会停顿在 `epoll_wait` 系统调用上，这是合理的，因为在没有客户端请求时，`nginx` 就阻塞于此(除非是在争用 `accept_mutex` 锁)，在另一终端执行 `wget` 命令向 `nginx` 发出 `http` 请求后，在来看 `strace` 的输出：

```
[root@localhost ~]# wget 127.0.0.1
```

```
[root@localhost nginx-1.2.0]# strace -p 4033
Process 4033 attached - interrupt to quit
gettimeofday({1337678055, 808290}, NULL) = 0
epoll_wait(8, {EPOLLIN, {u32=161263400, u64=577736855967805224}}, 512, -1) = 1
gettimeofday({1337678253, 140811}, NULL) = 0
accept4(6, {sa_family=AF_INET, sin_port=htons(41096), sin_addr=inet_addr("127.0.0.1")}, [16], SOCK_NONBLOCK) =
3
epoll_ctl(8, EPOLL_CTL_ADD, 3, {EPOLLIN|EPOLLET, {u32=161263592, u64=13813868851468480488}}) = 0
epoll_wait(8, {EPOLLIN, {u32=161263592, u64=13813868851468480488}}, 512, 60000) = 1
gettimeofday({1337678253, 148133}, NULL) = 0
recv(3, "GET / HTTP/1.0\r\nUser-Agent: Wget...", 1024, 0) = 107
stat64("/usr/local/nginx/html/index.html", {st_mode=S_IFREG|0644, st_size=151, ...}) = 0
open("/usr/local/nginx/html/index.html", O_RDONLY|O_NONBLOCK|O_LARGEFILE) = 9
fstat64(9, {st_mode=S_IFREG|0644, st_size=151, ...}) = 0
writev(3, [{"HTTP/1.1 200 OK\r\nServer: nginx/1"...}, 215], 1) = 215
sendfile64(3, 9, [0], 151) = 151
write(4, "127.0.0.1 - - [22/May/2012:17:17:...], 96) = 96
close(9) = 0
setsockopt(3, SOL_TCP, TCP_NODELAY, [1], 4) = 0
recv(3, "", 1024, 0) = 0
close(3) = 0
epoll_wait(8, [
```

通过 `strace` 的输出可以看到 `nginx` 工作进程在处理一次客户端请求过程中发起的所有系统调用。我这里测试请求的 `html` 非常简单，没有附带 `css`、`js`、`jpg` 等文件，所以看到的输出也比较简单。`strace` 输出的每一行记录一次系统调用，等号左边是系统调用名以及调用参数，等号右边是该系统调用的返回值。

1. `epoll_wait` 返回值为 1，表示有 1 个描述符存在可读/写事件，这里当然是可读事件。
2. `accept4` 接受该请求，返回的数字 3 表示 `socket` 的文件描述符。
3. `epoll_ctl` 把 `accept4` 建立的 `socket` 套接字（注意参数 3）加入到事件监听机制里。
4. `recv` 从发生可读事件的 `socket` 文件描述符内读取数据，读取的数据存在第二个参数内，读取了 107 个字节。
5. `stat64` 判断客户端请求的 `html` 文件是否存在，返回值为 0 表示存在。
6. `open/fstat64` 打开并获取文件状态信息。`open` 文件返回的文件描述符为 9，后面几个系统调用都用到这个值。
7. `writev` 把响应头通过文件描述符 3 代表的 `socket` 套接字发给客户端。
8. `sendfile64` 把文件描述符 9 代表的响应体通过文件描述符 3 代表的 `socket` 套接字发给客户端。
9. 再往文件描述符 4 代表的日志文件内 `write` 一条日志信息。

10. `recv` 看客户端是否还发了其它待处理的请求/信息。

11. 最后关闭文件描述符 3 代表的 `socket` 套接字。

由于 `strace` 能够提供 `nginx` 执行过程中的这些内部信息，所以在出现一些奇怪现象，比如 `nginx` 启动失败、响应的文件数据和预期不一致、莫名其妙的 `Segment Fault` 段错误、存在性能瓶颈（利用 `-T` 选项跟踪各个函数的消耗时间），利用 `strace` 也许能提供一些相关帮助。最后，要退出 `strace` 跟踪，按 `ctrl+c` 即可。

命令 `strace` 跟踪的是系统调用，对于 `nginx` 本身的函数调用关系无法给出更为明朗的信息，如果我们发现 `nginx` 当前运行不正常，想知道 `nginx` 当前内部到底在执行什么函数，那么命令 `pstack` 就是一个非常方便实用的工具。

`pstack` 的使用也非常简单，后面跟进程 `id` 即可，比如在无客户端请求的情况下，`nginx` 阻塞在 `epoll_wait` 系统调用处，此时利用 `pstack` 查看到的 `nginx` 函数调用堆栈关系如下：

```
[root@localhost objs]# strace -p 6966
Process 6966 attached - interrupt to quit
gettimeofday({1337695964, 474728}, NULL) = 0
epoll_wait(8, ^C <unfinished ...>
Process 6966 detached
[root@localhost objs]# pstack 6966
#0  0x00be7424 in __kernel_vsyscall ()
#1  0x001ed858 in __epoll_wait_nocancel () from /lib/libc.so.6
#2  0x08070aef in ngx_epoll_process_events ()
#3  0x08065f5a in ngx_process_events_and_timers ()
#4  0x0806f4d3 in ngx_worker_process_cycle ()
#5  0x0806ca8b in ngx_spawn_process ()
#6  0x0806e9e2 in ngx_start_worker_processes ()
#7  0x0806e286 in ngx_master_process_cycle ()
#8  0x0804a788 in main ()
```

从 `main()` 函数到 `epoll_wait()` 函数的调用关系一目了然，和在 `gdb` 内看到的堆栈信息一模一样，因为命令 `pstack` 本身就是一个利用 `gdb` 实现的 `shell` 脚本，关于这点，感兴趣的自己看看即可。

附带的说几句，我们要让 `nginx` 工作进程实际执行起来，必然要向 `nginx` 发出 `http` 请求，除了采用真实浏览器（比如 `IE`、`Opera` 等）以外，还可以使用类似于 `linux` 下的 `wget` 或 `curl` 这样的工具，可以自定义 `http` 请求头部，查看响应头部等，非常的方便。关于这两个命令的详细用法，请查 `man` 手册或 `Google`，另外可参考一下网址：<http://lenky.info/?p=1841>。

## 获得 `nginx` 程序执行流程

利用 `strace` 能帮助我们获取到 `nginx` 在运行过程中所发起的所有系统调用，但是不能看到 `nginx` 内部各个函数的调用情况；利用 `gdb` 调试 `nginx` 能让我们很清晰的获得 `nginx` 每一步的执行流程，但是单步调试毕竟是非常麻烦的，有没有更为方便的方法一次性获得 `nginx` 程序执行的整个流程呢？答案是肯定的，我们利用 `gcc` 的一个名为 “`-finstrument-functions`” 的编译选项，再加上一些我们自己的处理，就可以达到既定目的。关于 `-finstrument-function`

s 的具体介绍, 请直接参考官网手册: <http://gcc.gnu.org/onlinedocs/gcc/Code-Gen-Options.html#Code-Gen-Options>, 我就不再累述, 下面看看具体操作。

首先, 我们准备两个文件, 文件名和文件内容分别如下:

```
00: Filename : my_debug.h
01: #ifndef MY_DEBUG_LENKY_H
02: #define MY_DEBUG_LENKY_H
03: #include <stdio.h>
04:
05: void enable_my_debug( void ) __attribute__((no_instrument_function));
06: void disable_my_debug( void ) __attribute__((no_instrument_function));
07: int get_my_debug_flag( void ) __attribute__((no_instrument_function));
08: void set_my_debug_flag( int ) __attribute__((no_instrument_function));
09: void main_constructor( void ) __attribute__((no_instrument_function, constructor));
10: void main_destructor( void ) __attribute__((no_instrument_function, destructor));
11: void __cyg_profile_func_enter( void *,void *) __attribute__((no_instrument_function));
12: void __cyg_profile_func_exit( void *, void *) __attribute__((no_instrument_function));
13:
14: #ifndef MY_DEBUG_MAIN
15: extern FILE *my_debug_fd;
16: #else
17: FILE *my_debug_fd;
18: #endif
19: #endif
```

```
00: Filename : my_debug.c
01: #include "my_debug.h"
02: #define MY_DEBUG_FILE_PATH "/usr/local/nginx/sbin/mydebug.log"
03: int _flag = 0;
04:
05: #define open_my_debug_file() \
06:     (my_debug_fd = fopen(MY_DEBUG_FILE_PATH, "a"))
07:
08: #define close_my_debug_file() \
09:     do { \
10:         if(NULL != my_debug_fd) { \
11:             fclose(my_debug_fd); \
12:         } \
13:     }while(0)
14:
15: #define my_debug_print(args, fmt...) \
16:     do{ \
17:         if(0 == _flag) { \
18:             break; \
19:         } \
```

```

20:         if(NULL == my_debug_fd && NULL == open_my_debug_file()) { \
21:             printf("Err: Can not open output file.\n"); \
22:             break; \
23:         } \
24:         fprintf(my_debug_fd, args, ##fmt); \
25:         fflush(my_debug_fd); \
26:     }while(0)
27:
28: void enable_my_debug( void )
29: {
30:     _flag = 1;
31: }
32: void disable_my_debug( void )
33: {
34:     _flag = 0;
35: }
36: int get_my_debug_flag( void )
37: {
38:     return _flag;
39: }
40: void set_my_debug_flag( int flag )
41: {
42:     _flag = flag;
43: }
44: void main_constructor( void )
45: {
46:     //Do Nothing
47: }
48: void main_destructor( void )
49: {
50:     close_my_debug_file();
51: }
52: void __cyg_profile_func_enter( void *this, void *call )
53: {
54:     my_debug_print("Enter\n%p\n%p\n", call, this);
55: }
56: void __cyg_profile_func_exit( void *this, void *call )
57: {
58:     my_debug_print("Exit\n%p\n%p\n", call, this);
59: }

```

将这两个文件放到/nginx-1.2.0/src/core/目录下,然后编辑/nginx-1.2.0/objs/Makefile 文件,  
给 CFLAGS 选项增加-finstrument-functions 选项:

02: Filename : Makefile

03: CFLAGS = -pipe -O0 -W -Wall -Wpointer-arith -Wno-unused-parameter

-Wunused-function -Wunused-va                      riable -Wunused-value -Werror -g  
-finstrument-functions

接着，需要将 my\_debug.h 和 my\_debug.c 引入到 nginx 源码里一起编译，所以继续修改 /nginx-1.2.0/objs/Makefile 文件，根据 nginx 的 Makefile 文件特点，修改的地方主要有如下几处：

```
00: Filename : Makefile
01: ...
18: CORE_DEPS = src/core/nginx.h \
19:           src/core/my_debug.h \
20: ...
84: HTTP_DEPS = src/http/ngx_http.h \
85:           src/core/my_debug.h \
86: ...
102: objs/nginx:      objs/src/core/nginx.o \
103:           objs/src/core/my_debug.o \
104: ...
211:           $(LINK) -o objs/nginx \
212:           objs/src/core/my_debug.o \
213: ...
322: objs/src/core/my_debug.o: $(CORE_DEPS) \
323:           src/core/my_debug.c
324:           $(CC) -c $(CFLAGS) $(CORE_INCS) \
325:           -o objs/src/core/my_debug.o \
326:           src/core/my_debug.c
327: ...
```

为了在 nginx 源码里引入 my\_debug，这需要在 nginx 所有源文件都包含有头文件 my\_debug.h，当然没必要每个源文件都去添加对这个头文件的引入，我们只需要在头文件 ngx\_core.h 内加入对 my\_debug.h 文件的引入即可，这样其它 nginx 的源文件就间接的引入了这个文件：

```
37: Filename : ngx_core.h
38: #include "my_debug.h"
```

在源文件 nginx.c 的最前面加上对宏 MY\_DEBUG\_MAIN 的定义，以使得 nginx 程序有且仅有一个 my\_debug\_fd 变量的定义：

```
06: Filename : nginx.c
07: #define MY_DEBUG_MAIN 1
08:
09: #include <ngx_config.h>
10: #include <ngx_core.h>
11: #include <nginx.h>
```

最后就是根据我们想要截取的执行流程，在适当的位置调用函数 enable\_my\_debug();和函数 disable\_my\_debug();，这里仅作测试，直接在 main 函数入口处调用 enable\_my\_debug();，

而 `disable_my_debug();` 函数就不调用了:

```
200: Filename : nginx.c
201: main(int argc, char *const *argv)
202: {
203: ...
208: enable_my_debug();
```

至此, 代码增补工作已经完成, 重新编译 `nginx`, 如果之前已编译过 `nginx`, 那么如下的第一步源文件时间戳刷新步骤很重要:

```
[root@localhost nginx-1.2.0]# find . -name "*" | xargs touch
[root@localhost nginx-1.2.0]# make
make -f objs/Makefile
make[1]: Entering directory `/home/gqk/nginx-1.2.0'
gcc -c -pipe -O0 -W -Wall -Wpointer-arith -Wno-unused-parameter -Wun-used-function
-Wun-used-variable -Wun-used-value -Werror -g -finstrument-functions -I src/core -I src/event -I
src/event/modules -I src/os/unix -I objs \
    -o objs/src/core/nginx.o \
    src/core/nginx.c
...
-lpthread -lcrypt -lpcrc -lcrypto -lcrypto -lz
make[1]: Leaving directory `/home/gqk/nginx-1.2.0'
make -f objs/Makefile manpage
make[1]: Entering directory `/home/gqk/nginx-1.2.0'
make[1]: Nothing to be done for `manpage'.
make[1]: Leaving directory `/home/gqk/nginx-1.2.0'
[root@localhost nginx-1.2.0]#
```

以单进程模式运行 `nginx`, 并且在配置文件里将日志功能的记录级别设置低一点, 否则将有大量的日志函数调用堆栈信息, 经过这样的设置后, 我们才能获得更清晰的 `nginx` 执行流程, 即配置文件里做如下设置:

```
00: Filename : nginx.c
01: master_process off;
02: error_log logs/error.log emerg;
```

正常运行后的 `nginx` 将产生一个记录程序执行流程的文件, 这个文件会随着 `nginx` 的持续运行迅速增大, 所以在恰当的地方调用 `disable_my_debug();` 函数是非常有必要的, 不过我这里在获取到一定量的信息后就直接 `kill` 掉 `nginx` 进程了。mydebug.log 的内容类似于如下所示:

```
[root@localhost/sbin]# head -n 20 mydebug.log
Enter
0x804a5fc
0x806e2b3
Exit
0x804a5fc
0x806e2b3
```

...

这记录的是函数调用关系，不过这里的函数还只是以对应的地址显示而已，利用另外一个工具 `addr2line` 可以将这些地址转换回可读的函数名。`addr2line` 工具在大多数 linux 发行版上默认有安装，如果没有那么在官网 <http://sourceware.org/binutils/> 下载即可，其具体用法也可以参考官网手册：<http://sourceware.org/binutils/docs/binutils/addr2line.html>，这里直接使用，写个 `addr2line.sh` 脚本：

```
00: Filename : addr2line.sh
01: #!/bin/sh
02:
03: if [ $# != 3 ]; then
04:     echo 'Usage: addr2line.sh executefile addressfile functionfile'
05:     exit
06: fi;
07:
08: cat $2 | while read line
09: do
10:     if [ "$line" = 'Enter' ]; then
11:         read line1
12:         read line2
13:         # echo $line >> $3
14:         addr2line -e $1 -f $line1 -s >> $3
15:         echo "--->" >> $3
16:         addr2line -e $1 -f $line2 -s | sed 's/^/    /' >> $3
17:         echo >> $3
18:     elif [ "$line" = 'Exit' ]; then
19:         read line1
20:         read line2
21:         addr2line -e $1 -f $line2 -s | sed 's/^/    /' >> $3
22:         echo "<---" >> $3
23:         addr2line -e $1 -f $line1 -s >> $3
24:         # echo $line >> $3
25:         echo >> $3
26:     fi;
27: done
```

执行 `addr2line.sh` 进行地址与函数名的转换，这个过程挺慢的，因为从上面的 shell 脚本可以看到对于每一个函数地址都调用 `addr2line` 进行转换，执行效率完全没有考虑，不过够用就好，如果非要追求高效率，直接写个 c 程序来做这个转换工作也是可以的。

```
[root@localhost sbin]# vi addr2line.sh
[root@localhost sbin]# chmod a+x addr2line.sh
[root@localhost sbin]# ./addr2line.sh nginx mydebug.log myfun.log
[root@localhost sbin]# head -n 12 myfun.log
main
```



```
nginx.c:212
--->
    ngx_strerror_init
    ngx_errno.c:47

    ngx_strerror_init
    ngx_errno.c:47
<---
main
nginx.c:212
```

```
[root@localhost sbin]#
```

关于如获得 nginx 程序执行流程的方法大体就是上面描述的这样了，当然，这里介绍得很粗略，写的代码都也仅是作为示范以抛砖引玉，关于 gcc 以及相关工具的更深入研究以不在本书的范围之内，如感兴趣可查看上文中提供的相关链接。

## 加桩调试

如果我们对代码做过单元测试，那么肯定知道加桩的概念，简单点说就是为了让一个模块执行起来，额外添加的一些支撑代码。比如，我要简单测试一个实现某种排序算法的子函数的功能是否正常，那么我也许需要写一个 main() 函数，设置一个数组，提供一些乱序的数据，然后利用这些数据调用排序子函数（假设它提供的接口就是对数组的排序，等），然后 printf 打印排序后的结果，看是否排序正常，所有写的这些额外代码（main() 函数、数组、printf 打印）就是桩代码。

上面提到的这种用于单元测试的方法，同样也可以用来深度调试 nginx 内部逻辑，而且 nginx 很多的基础实现（比如 slab 机制、红黑树、chain 链、array 数组等）都比较独立，要调试它们只需提供少量的桩代码即可。

以 nginx 的 slab 机制为例，通过下面提供的一些桩代码来调试该功能的具体实现。nginx 的 slab 机制用于对多进程共享内存的管理，不过单进程也是一样的执行逻辑，除了加/解锁直通以外（即加锁时必定成功），所以我们采取最简单的办法，直接在 nginx 本身的 main() 函数内插入我们的桩代码。当然，必须根据具体情况把桩代码放在合适的调用位置，比如这里的 slab 机制就依赖一些全局变量（像 ngx\_pagesize 等），所以需要把桩代码的调用位置放在这些全局变量的初始化之后：

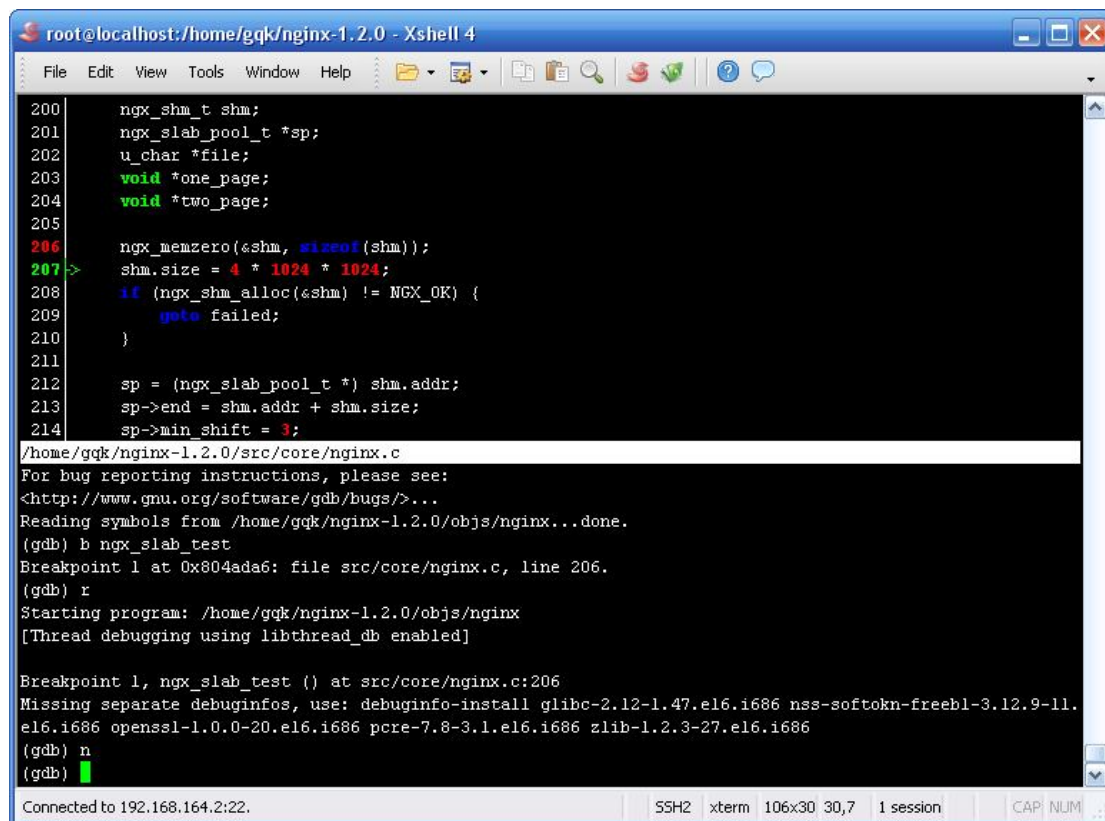
```
197: Filename : nginx.c
198: void ngx_slab_test()
199: {
200:     ngx_shm_t shm;
201:     ngx_slab_pool_t *sp;
```

```
202:    u_char *file;
203:    void *one_page;
204:    void *two_page;
205:
206:    ngx_memzero(&shm, sizeof(shm));
207:    shm.size = 4 * 1024 * 1024;
208:    if (ngx_shm_alloc(&shm) != NGX_OK) {
209:        goto failed;
210:    }
211:
212:    sp = (ngx_slab_pool_t *) shm.addr;
213:    sp->end = shm.addr + shm.size;
214:    sp->min_shift = 3;
215:    sp->addr = shm.addr;
216:
217:    #if (NGX_HAVE_ATOMIC_OPS)
218:        file = NULL;
219:    #else
220:        #error must support NGX_HAVE_ATOMIC_OPS.
221:    #endif
222:    if (ngx_shmtx_create(&sp->mutex, &sp->lock, file) != NGX_OK) {
223:        goto failed;
224:    }
225:
226:    ngx_slab_init(sp);
227:
228:    one_page = ngx_slab_alloc(sp, ngx_pagesize);
229:    two_page = ngx_slab_alloc(sp, 2 * ngx_pagesize);
230:
231:    ngx_slab_free(sp, one_page);
232:    ngx_slab_free(sp, two_page);
233:
234:    ngx_shm_free(&shm);
235:
236:    exit(0);
237: failed:
238:    printf("failed.\n");
239:    exit(-1);
240: }
241: ...
353:    if (ngx_os_init(log) != NGX_OK) {
354:        return 1;
355:    }
356:
```

```
357:     ngx_slab_test();
358: ...
```

上面是修改之后的 nginx.c 源文件，直接 make 后生成新的 nginx，不过这个可执行文件不再是一个 web server，而是一个简单的调试 slab 机制的辅助程序。可以看到，程序在进入 main() 函数后先做一些初始化工作，然后通过 ngx\_slab\_test() 函数调入到桩代码内执行调试逻辑，完成既定目标后便直接 exit() 退出整个程序。

正常运行时，nginx 本身对内存的申请与释放是不可控的，所以直接去调试 nginx 内存管理的 slab 机制的代码逻辑比较困难，利用这种加桩的办法，ngx\_slab\_alloc() 申请内存和 ngx\_slab\_free() 释放内存都能精确控制，对每一次内存的申请与释放后，slab 机制的内部结构是怎样一个变化都能进行把握，对其相关逻辑的理解起来也就没那么困难了。下面是利用 cgdb 调试这个程序的界面显示：



```

root@localhost:/home/gqk/nginx-1.2.0 - Xshell 4
File Edit View Tools Window Help
200     ngx_shm_t shm;
201     ngx_slab_pool_t *sp;
202     u_char *file;
203     void *one_page;
204     void *two_page;
205
206     ngx_memzero(&shm, sizeof(shm));
207     shm.size = 4 * 1024 * 1024;
208     if (ngx_shm_alloc(&shm) != NGX_OK) {
209         goto failed;
210     }
211
212     sp = (ngx_slab_pool_t *) shm.addr;
213     sp->end = shm.addr + shm.size;
214     sp->min_shift = 3;
/home/gqk/nginx-1.2.0/src/core/nginx.c
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/gqk/nginx-1.2.0/objs/nginx...done.
(gdb) b ngx_slab_test
Breakpoint 1 at 0x804ada6: file src/core/nginx.c, line 206.
(gdb) r
Starting program: /home/gqk/nginx-1.2.0/objs/nginx
[Thread debugging using libthread_db enabled]

Breakpoint 1, ngx_slab_test () at src/core/nginx.c:206
Missing separate debuginfos, use: debuginfo-install glibc-2.12-1.47.el6.i686 nss-softokn-freebl-3.12.9-11.
el6.i686 openssl-1.0.0-20.el6.i686 pcre-7.8-3.1.el6.i686 zlib-1.2.3-27.el6.i686
(gdb) n
(gdb)
Connected to 192.168.164.2:22.  SSH2  xterm  106x30  30,7  1 session  CAP NUM

```

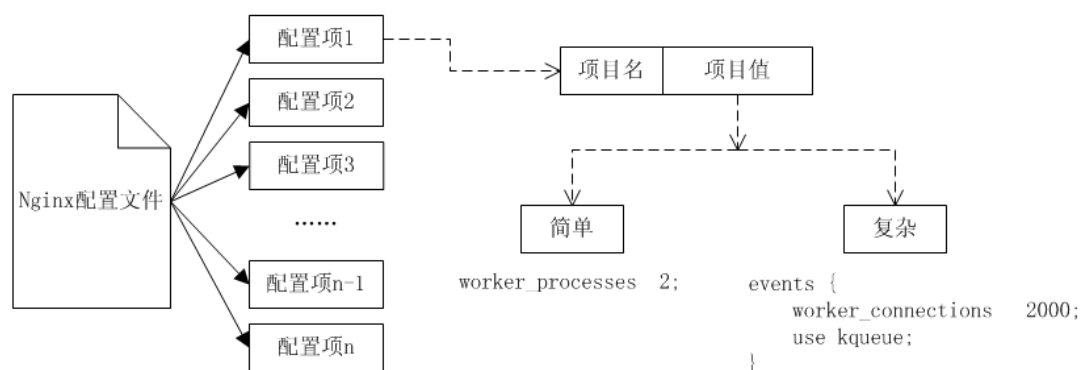
## 第三章 配置解析

### 配置文件格式

nginx 的配置文件格式是 nginx 作者自己定义的,并没有采用像语法分析生成器 LEMON 那种经典复杂的 LALR (1) 语法来描述配置信息,而是采用类似于 ini 这种普通却又简单的 name-value 对来描述配置信息,不过 nginx 对此做了扩展,以提供更为灵活的用户配置。

对于这种自定义格式的配置文件,好处就是自由、灵活,而坏处就是对于 nginx 的每一项配置信息都必须去针对性的解析和设置,因此我们很容易的看到 nginx 源码里有大量篇幅的配置信息解析与赋值代码。

类似于 ini 文件,nginx 配置文件也是由多个配置项组成的,每一个配置项都有一个项目名和对应的项目值,项目名又被称为指令 (Directive),而项目值可能是简单的字符串 (以分号结尾),也可能是由简单字符串和多个配置项组合而成配置块的复合结构 (以大括号结尾),我们可以将配置项归纳为两种:简单配置项和复杂配置项。



上图只是一个示例,而实际的简单配置项与复杂配置项会更多样化,要区分简单配置项与复杂配置项却很简单,不带大括号的就是简单配置项,反之则反,比如:

```
error_log /var/log/nginx.error_log info;
```

因为它不带大括号,所以是一个简单配置项;而

```
location ~ \.php$ {
    fastcgi_pass 127.0.0.1:1025;
}
```

带大括号,所以是一个复杂配置项。为什么要做这种看似毫无意义的区分? 因为后面会看到对于复杂配置项而言,nginx 并不做具体的解析与赋值操作,一般只是申请对应的内容空间、切换解析状态,然后递归调用 (因为复杂配置项本身含有递归的思想) 解析函数,而真正将用户配置信息转换为 nginx 内控制变量的值,还是依靠那些简单配置项所对应的处理函数来做。

不管是简单配置项还是复杂配置项,它们的项目名和项目值都是由标记 (token: 这里是指一个配置文件字符串内容中被空格、引号、分号、tab 号、括号,比如 '{'、换行符等

分割开来的字符子串)组成的,配置项目名就是一个 token,而配置项目值可以是一个、两个和多个 token 组成。

比如简单配置项:

```
daemon off;
```

其项目名 `daemon` 为一个 token,项目值 `off` 也是一个 token;简单配置项:

```
error_page 404 /404.html;
```

其项目值就包含有两个 token,分别为 `404` 和 `/404.html`。

对于复杂配置项:

```
location /gqk {
    index    index.html index.htm index.php;
    try_files $uri $uri/ @gqk;
}
```

其项目名 `location` 为一个 token,项目值是一个 token (`/gqk`) 和多条简单配置项(通过大括号)组成的复合结构(后续称之为配置块)。上面几个例子中的 token 都是被空格分割出来的,事实上下面这样的配置也是正确的:

```
"daemon" "off";
'daemon' 'off';
daemon 'off';
"daemon" off;
```

当然,一般情况下没必要画蛇添足似的去加些引号,除非我们需要在 token 内包含空格而又不想使用转义字符(`\`)的话就可以利用引号,比如:

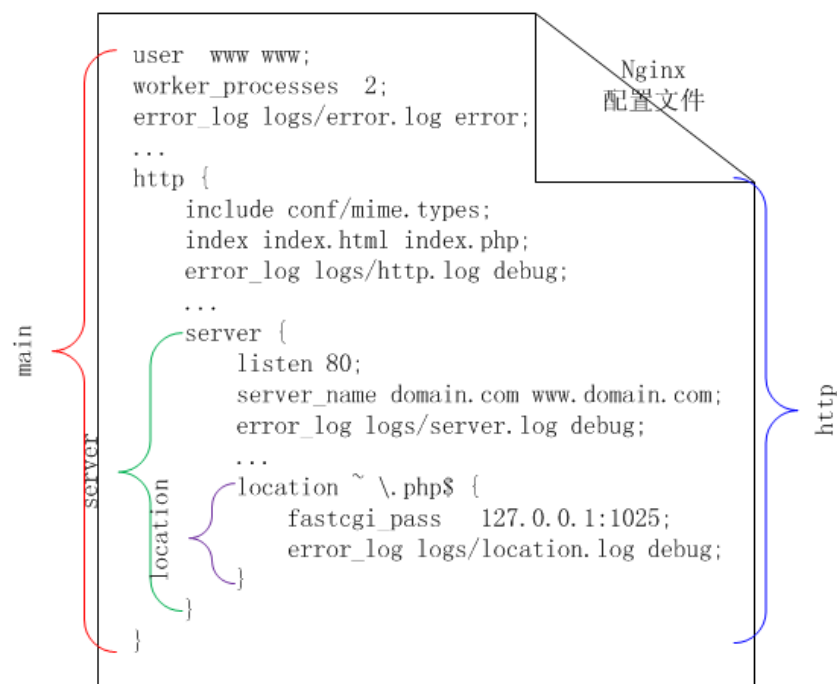
```
log_format main '$remote_addr - $remote_user [$time_local] $status '
    '"$request" $body_bytes_sent "$http_referer" '
    '"$http_user_agent" "$http_x_forwarded_for";
```

但是像下面这种格式就会有问题,这对于我们来说很容易理解,不多详叙:

```
"daemon "off";
```

最后值得提一下的是,nginx 配置文件里的注释信息以井号(`#`)作为开头标记。

直观上看到的配置文件格式大概就是上面介绍的这些,但根据 nginx 应用本身的特定,我们可以对配置文件做上下文识别和区分,或者说是配置项的作用域,因为虽然某项配置项在同一个上下文里只能设置一次,但却可以在不同的上下文里设置多次,以便达到更细粒的控制,比如配置项 `error_log` 就是如此,在不同的 `server` 上下文里可以设置不同的日志输出级别和输出文件路径。就 http 应用而言,目前 nginx 预定义的配置上下文主要包括 `main`、`http`、`server`、`location` 这四种(还有其他几种,比如 `event`、`upstream`、`if`、`mail` 等),下面是一个 http 服务器示例配置的上下文情况:



## 配置项目解析准备

前面提到对于配置文件里的每一项配置,程序都必须去针对性的解析并转化为内部控制变量的值,因此对于所有可能出现的配置项,nginx 都会提供有对应的代码去做它的解析转换工作,如果配置文件内出现了 nginx 无法解析的配置项,那么 nginx 将报错并直接退出程序。

举例来说,对于配置项 `daemon`,在模块 `ngx_core_module` 的配置项目解析数组内的第一元素就是保存的对该配置项进行解析所需要的信息,比如 `daemon` 配置项的类型,执行实际解析操作的回调函数,解析出来的配置项值所存放的地址等:

```

static ngx_command_t  ngx_core_commands[] = {
    { ngx_string("daemon"),
      NGX_MAIN_CONF|NGX_DIRECT_CONF|NGX_CONF_FLAG,
      ngx_conf_set_flag_slot,
      0,
      offsetof(ngx_core_conf_t, daemon),
      NULL },
    .....
}
    
```

而如果我在配置文件中加入如下配置内容:

```
lenky on;
```

nginx 启动后将直接返回如下提示错误,这是因为对于“lenky on”这个配置项,nginx 根本就没有对应的代码去解析它:

```
[emerg]: unknown directive "lenky" in /usr/local/nginx/conf/nginx.conf:2
```

如果你在使用 nginx 的过程中也遇到类似的错误提示,那么请立即检查配置文件是否不



小心敲错了字符。

为了统一配置项目的解析，nginx 利用 `ngx_command_s` 数据类型对所有的 nginx 对配置项进行了统一的描述：

```
struct ngx_command_s {
    ngx_str_t      name;
    ngx_uint_t     type;
    char           *(*set)(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
    ngx_uint_t     conf;
    ngx_uint_t     offset;
    void           *post;
};
```

这是一个结构体数据类型，它包含多个字段，其中几个主要字段的含义为：字段 `name` 指定与其对应的配置项目的名称，字段 `set` 指向一个回调函数，而字段 `offset` 指定转换后控制值的存放位置。

以上面的 `daemon` 配置项目为例，当遇到配置文件里的 `daemon` 项目名时，nginx 就调用 `ngx_conf_set_flag_slot` 回调函数对其项目值进行解析，并根据其是 `on` 还是 `off` 把 `ngx_core_conf_t` 的 `daemon` 字段置为 1 或者 0，这样就完成了从配置项目信息到 nginx 内部实际值的转换过程。当然，这还有其它一些细节未说，下面再具体来看：

`ngx_command_s` 结构体的 `type` 字段指定该配置项的多种相关信息，比如：

1. 该配置的类型：NGX\_CONF\_FLAG 表示该配置项目有一个布尔类型的值，例如 `daemon` 就是一个布尔类型的配置项目，其值为 `on` 或者 `off`；NGX\_CONF\_BLOCK 表示该配置项目为复杂配置项，因此其有一个由大括号组织起来的多值块，比如配置项 `http`、`events` 等。
2. 该配置项目的配置值的 token 个数：NGX\_CONF\_NOARGS、NGX\_CONF\_TAKE1、NGX\_CONF\_TAKE2、……、NGX\_CONF\_TAKE7，分别表示该配置项的配置值没有 token、一个、两个、……、七个 token；NGX\_CONF\_TAKE12、NGX\_CONF\_TAKE123、NGX\_CONF\_1MORE 等这些表示该配置项的配置值的 token 个数不定，分别为 1 个或 2 个、1 个或 2 个或 3 个、一个以上。
3. 可以该配置项目可处在的上下文：NGX\_MAIN\_CONF（配置文件最外层，不包含其内的类似于 `http` 这样的配置块内部，即不向内延伸，其他上下文都有这个特性）、NGX\_EVENT\_CONF（event 配置块）、NGX\_HTTP\_MAIN\_CONF（http 配置块）、NGX\_HTTP\_SRV\_CONF（http 的 server 指令配置块）、NGX\_HTTP\_LOC\_CONF（http 的 location 指令配置块）、NGX\_HTTP\_SIF\_CONF（http 的在 server 配置块内的 if 指令配置块）、NGX\_HTTP\_LIF\_CONF（http 的在 location 配置块内的 if 指令配置块）、NGX\_HTTP\_LMT\_CONF（http 的 limit\_except 指令配置块）、NGX\_HTTP\_UPS\_CONF（http 的 upstream 指令配置块）、NGX\_MAIL\_MAIN\_CONF（mail 配

置块)、NGX\_MAIL\_SRV\_CONF (mail 的 server 指令配置块), 等等。

字段 conf 被 NGX\_HTTP\_MODULE 类型模块所用, 该字段指定当前配置项所在的大致位置, 取值为 NGX\_HTTP\_MAIN\_CONF\_OFFSET、NGX\_HTTP\_SRV\_CONF\_OFFSET、NGX\_HTTP\_LOC\_CONF\_OFFSET 三者之一; 其它模块基本不用该字段, 直接指定为 0。

字段 offset 指定该配置项值的精确存放位置, 一般指定为某一个结构体变量的字段偏移 (利用 offsetof 宏), 对于复杂配置项目, 例如 server, 它不用保存配置项值, 或者说它本身无法保存, 亦可以说是因为它的值被分得更细小而被单个保存起来, 此时字段 offset 指定为 0 即可。

字段 post 在大多数情况下都为 NULL, 但在某些特殊配置项中也会指定其值, 而且多为回调函数指针, 例如 auth\_basic、connection\_pool\_size、request\_pool\_size、optimize\_host\_names、client\_body\_in\_file\_only 等配置项。

每个模块都把自己所需要的配置项目的对应 ngx\_command\_s 结构体变量组成一个数组, 并以 ngx\_xxx\_xxx\_commands 的形式命名, 该数组以元素 ngx\_null\_command 作为结束哨兵。

## 配置文件解析流程

下面开始对 nginx 配置信息的整个解析流程进行描述, 假设我们以命令:

```
nginx -c /usr/local/nginx/conf/nginx.conf
```

启动 nginx, 而配置文件 nginx.conf 也比较简单, 如下所示:

```
06: Filename : nginx.conf
07: worker_processes 2;
08: error_log logs/error.log debug;
09: events {
10:     use epoll;
11:     worker_connections 1024;
12: }
13: http {
14:     include mime.types;
15:     default_type application/octet-stream;
16:     server {
17:         listen 8888;
18:         server_name localhost;
19:         location / {
20:             root html;
21:             index index.html index.htm;
22:         }
23:         error_page 404 /404.html;
```

```

24:         error_page 500 502 503 504 /50x.html;
25:         location = /50x.html {
26:             root html;
27:         }
28:     }
29: }

```

```

00: Filename : mime.types
01: types {
02:     text/html    html htm shtml;
03:     text/css     css;
04:     text/xml     xml;
05:     image/gif    gif;
06:     image/jpeg   jpeg jpg;
07:     application/x-javascript js;
08: ...
09: }

```

首先，抹掉一些前枝末节，我们直接跟着 nginx 的启动流程进入到与配置信息相关的函数调用处：

```

main -> ngx_init_cycle -> ngx_conf_parse:
267: Filename : ngx_cycle.c
268: if (ngx_conf_parse(&conf, &cycle->conf_file) != NGX_CONF_OK) {
269:     environ = senv;
270:     ngx_destroy_cycle_pools(&conf);
271:     return NULL;
272: }

```

此处调用 `ngx_conf_parse` 函数传入了两个参数，第一个参数为 `ngx_conf_s` 变量，关于这个变量我们在他处再讲，而第二个参数就是保存的配置文件路径的字符串 `/usr/local/nginx/conf/nginx.conf`。`ngx_conf_parse` 函数是执行配置文件解析的关键函数，其原型申明如下：

```
char *ngx_conf_parse(ngx_conf_t *cf, ngx_str_t *filename);
```

它是一个间接递归函数，也就是说虽然我们在该函数体内看不到直接的对其本身的调用，但是它执行的一些其它函数（比如 `ngx_conf_handler`）内又会调用到 `ngx_conf_parse` 函数，从而形成递归，这一般在处理复杂配置项和一些特殊配置指令时发生，比如指令 `include`、`events`、`http`、`server`、`location` 等。

`ngx_conf_parse` 函数体代码量不算太多，但是它照样也将配置内容的解析过程分得很清楚，总体来看分成三个步骤：

- 1，判断当前解析状态；
- 2，读取配置标记 `token`；
- 3，当读取了合适数量的标记 `token` 之后对其进行实际的处理，也就是将配置值转换为

nginx 内对应控制变量的值。

当进入到 `ngx_conf_parse` 函数时, 首先做的第一步是判断当前解析过程处在一个什么样的状态, 这有三种可能:

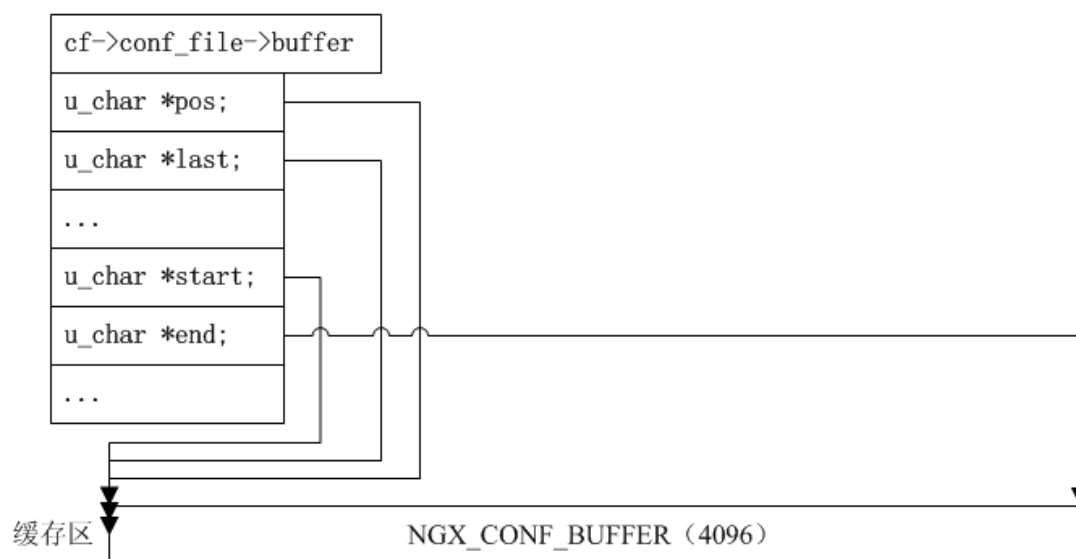
- a, 正要开始解析一个配置文件: 即此时的参数 `filename` 指向一个配置文件路径字符串, 需要函数 `ngx_conf_parse` 打开该文件并获取相关的文件信息以便下面代码读取文件内容并进行解析, 除了在上面介绍的 nginx 启动时开始主配置文件解析时属于这种情况, 还有当遇到 `include` 指令时也将以这种状态调用 `ngx_conf_parse` 函数, 因为 `include` 指令表示一个新的配置文件要开始解析。状态标记为 `type = parse_file;`。
- b, 正要开始解析一个复杂配置项值: 即此时配置文件已经打开并且也已经对文件部分进行了解析, 当遇到复杂配置项比如 `events`、`http` 等时, 这些复杂配置项的处理函数又会递归的调用 `ngx_conf_parse` 函数, 此时解析的内容还是来自当前的配置文件, 因此无需再次打开它, 状态标记为 `type = parse_block;`。
- c, 正要开始解析命令行参数配置项值, 这在对用户通过命令行 `-g` 参数输入的配置信息进行解析时处于这种状态, 如: `nginx -g 'daemon on;'`, nginx 在调用 `ngx_conf_parse` 函数对命令行参数配置信息 `'daemon on;'` 进行解析时就是这种状态, 状态标记为 `type = parse_param;`。

在判断好当前解析状态之后就开始读取配置文件内容, 前面已经说到配置文件都是有一个个 `token` 标记组成的, 因此接下来就是循环从配置文件里读取标记, 而 `ngx_conf_read_token` 函数就是做这个事情的:

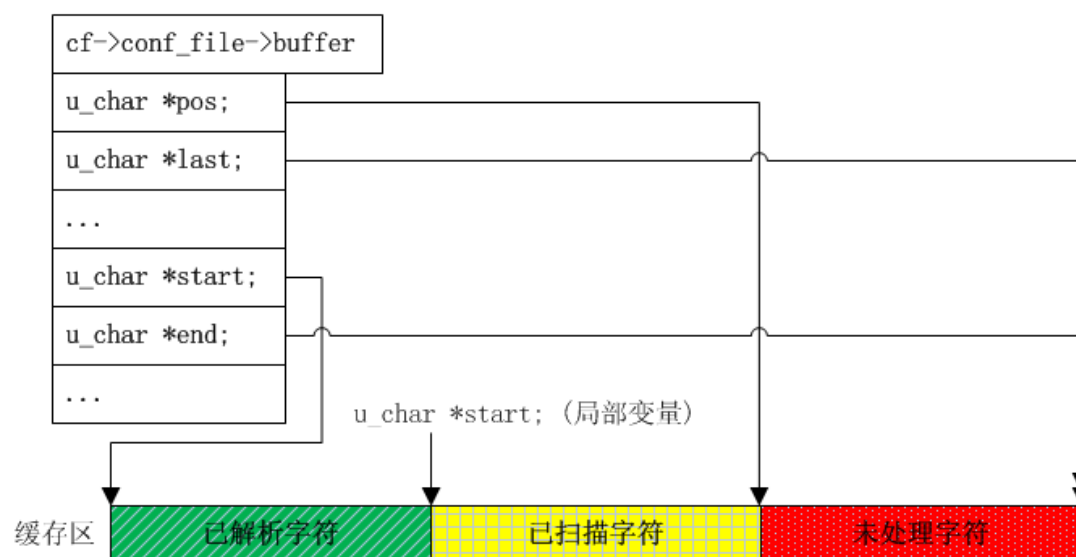
```
rc = ngx_conf_read_token(cf);
```

函数 `ngx_conf_read_token` 对配置文件内容逐个字符扫描并解析为单个的 `token`, 当然, 该函数并不会频繁的去读取配置文件, 它每次将从文件内读取足够多的内容以填满一个大小为 `NGX_CONF_BUFFER` (4096) 的缓存区 (除了最后一次, 即配置文件剩余内容本来就不够了), 这个缓存区在函数 `ngx_conf_parse` 内申请并保存引用到变量 `cf->conf_file->buffer` 内, 函数 `ngx_conf_read_token` 反复使用该缓存区, 该缓存区可能有如下一些状态:

初始状态, 即函数 `ngx_conf_parse` 内申请后的初始状态:

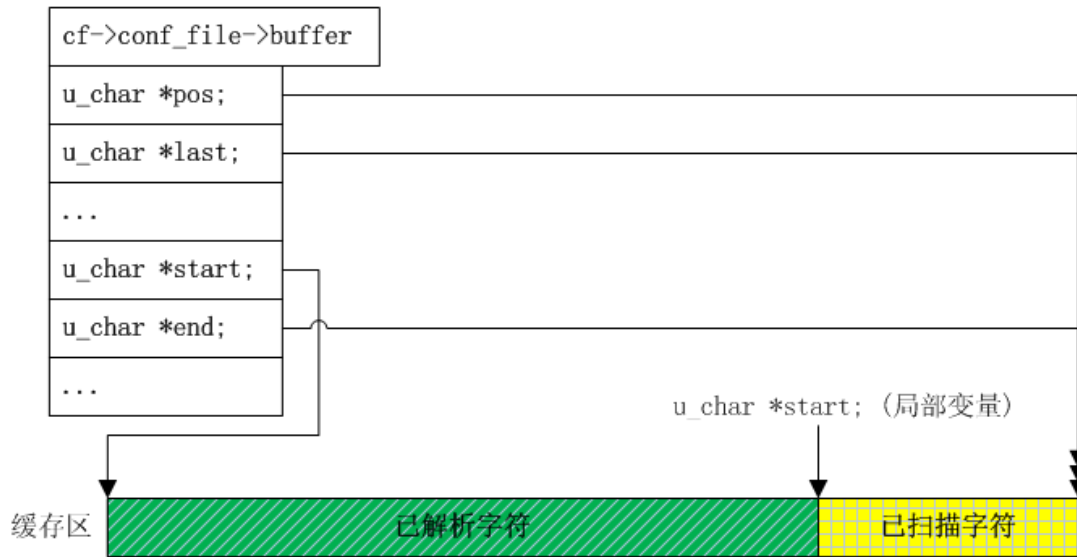


处理过程中的中间状态，有一部分配置内容已经被解析为一个个 token 并保存起来，而有一部分内容正要被组合成 token，还有一部分内容等待处理：

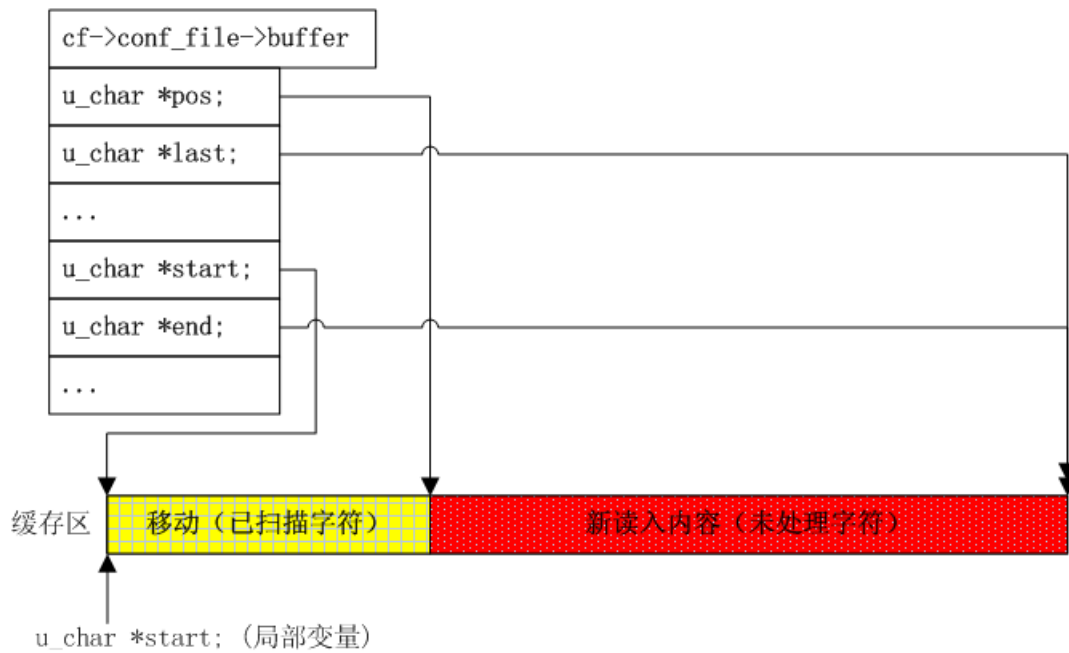


已解析字符和已扫描字符都属于已处理字符，但它们又是不同的，已解析字符表示这些字符已经被作为 token 额外的保存起来了，所以这些字符已经完全没用了；而已扫描字符表示这些字符还未组成一个完整的 token，所以它们还不能被丢弃。

当缓存区里的字符都处理完时，需要继续从打开的配置文件中读取新的内容到缓存区，此时的临界状态为：

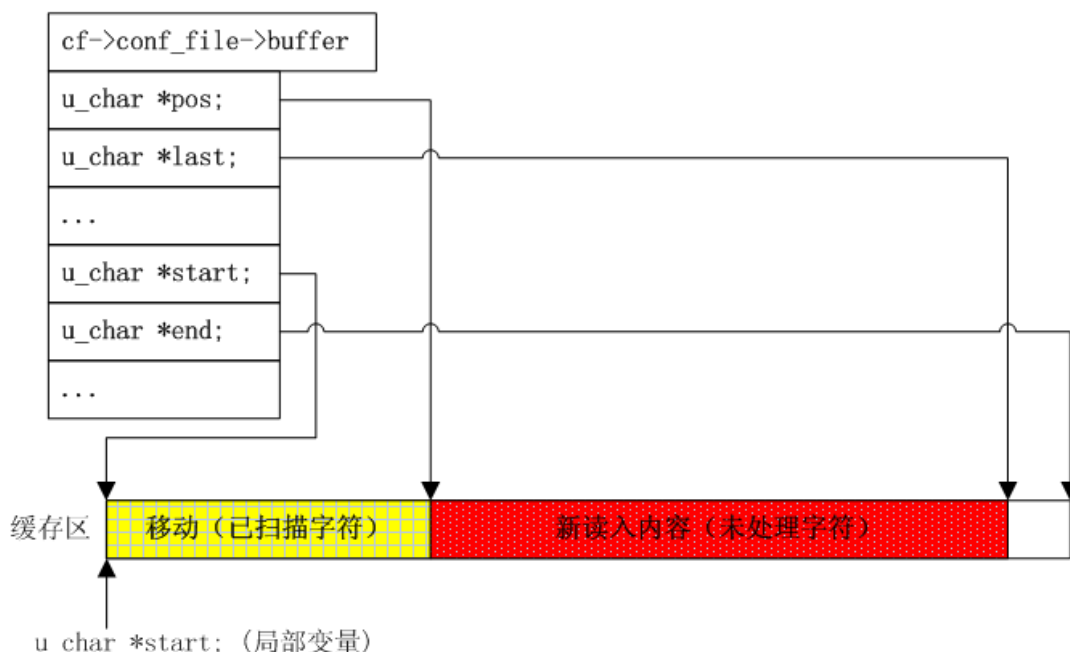


前面图示说过，已解析字符已经没用了，因此我们可以将已扫描但还未组成 token 的字符移动到缓存区的前面，然后从配置文件内读取内容填满缓存区剩余的空间，情况如下：



如果最后一次读取配置文件内容不够，那么情况就是下面这样：





函数 `ngx_conf_read_token` 在读取了合适数量的标记 `token` 之后就开始下一步骤,即对这些标记进行实际的处理,那多少才算是读取了合适数量的标记呢?区别对待,对于简单配置项则是读取其全部的标记,也就是遇到配置项结束标记分号;`;`为止,此时一条简单配置项的所有标记都已经被读取并存放在 `cf->args` 数组内,因此可以开始下一步骤进行实际的处理;对于复杂配置项则是读完其配置块前的所有标记,即遇到大括号`{`为止,此时复杂配置项处理函数所需要的标记都已读取到,而对于配置块`{}`内的标记将在接下来的函数 `ngx_conf_parse` 递归调用中继续处理,这可能是一个反复的过程。当然,函数 `ngx_conf_read_token` 也可能在其它情况下提前返回,比如配置文件格式出错、文件处理完(遇到文件结束)、块配置处理完(遇到大括号`}`),这几种返回情况的处理都很简单,不多详叙。

`ngx_conf_read_token` 函数如何识别并将 `token` 缓存在 `cf->args` 数组中的逻辑还是比较简单的。首先是对配置文件临时缓存区内容的调整(如有必要),这对应前面几个图示的缓存区状态;接着通过缓存区从前往后的扫描整个配置文件的内容,对每一个字符与前面已扫描字符的组合进行有效性检查并进行一些状态旗标切换,比如 `d_quoted` 旗标置 1 则表示当前处于双引号字符串后, `last_space` 旗标置 1 则表示前一个字符为空白字符(包括空格、回车、`tab` 等),……,这些旗标能大大方便接下来的字符有效性组合检查,比如前面的 `nginx.conf` 配置文件的第 5 行末尾多加了个分号(即有 2 个分号),那么启动 `nginx` 将报错:

```
nginx: [emerg] unexpected ";" in /usr/local/nginx/conf/nginx.conf:5
```

再接下来就是判断当前已扫描字符是否能够组成一个 `token` 标记,两个双引号、两个单引号、两个空白字符之间的字符就能够组成一个 `token` 标记,此时就在 `cf->args` 数组内申请对应的存储空间并进行 `token` 标记字符串拷贝,从而完成一个 `token` 标记的解析与读取工作;此时根据情况要么继续进行下一个 `token` 标记的解析与读取,要么返回到 `ngx_conf_parse` 函

数内进行实际的处理。

列表看一下 ngx\_conf\_parse 函数在解析 nginx.conf 配置文件时每次调用 ngx\_conf\_read\_token 后的 cf->args 里存储的内容是什么（这通过 gdb 调试 nginx 时在 ngx\_conf\_file.c:185 处加断点就很容易看到这些信息），这会大大帮助对后续内容的理解：

次数	返回值 rc	cf->args 存储内容
第 1 次	NGX_OK	(gdb) p (*cf->args)->nelts \$43 = 2 (gdb) p *((ngx_str_t*)((cf->args)->elts)) \$44 = {len = 16, data = 0x80ec0c8 "worker_processes"} (gdb) p *(ngx_str_t*)((cf->args)->elts + sizeof(ngx_str_t)) \$45 = {len = 1, data = 0x80ec0da "2"}
第 2 次	NGX_OK	(gdb) p (*cf->args)->nelts \$46 = 3 (gdb) p *((ngx_str_t*)((cf->args)->elts)) \$47 = {len = 9, data = 0x80ec0dd "error_log"} (gdb) p *(ngx_str_t*)((cf->args)->elts + sizeof(ngx_str_t)) \$48 = {len = 14, data = 0x80ec0e8 "logs/error.log"} (gdb) p *(ngx_str_t*)((cf->args)->elts + 2*sizeof(ngx_str_t)) \$49 = {len = 5, data = 0x80ec0f8 "debug"}
第 3 次	NGX_CONF_BLOCK_START	(gdb) p (*cf->args)->nelts \$52 = 1 (gdb) p *((ngx_str_t*)((cf->args)->elts)) \$53 = {len = 6, data = 0x80ec11f "events"}
第...次	.....	.....
第 6 次	NGX_CONF_BLOCK_DONE	(gdb) p (*cf->args)->nelts \$58 = 0
第...次	.....	.....
第 n 次	NGX_CONF_BLOCK_START	(gdb) p (*cf->args)->nelts \$74 = 2 (gdb) p *((ngx_str_t*)((cf->args)->elts)) \$75 = {len = 8, data = 0x80f7392 "location"} (gdb) p *(ngx_str_t*)((cf->args)->elts + sizeof(ngx_str_t)) \$76 = {len = 1, data = 0x80f739c "/" }
第...次	.....	.....

第末次	NGX_CONF_FILE_DONE	(gdb) p (*cf->args)->nelts \$65 = 0
-----	--------------------	--

ngx\_conf\_read\_token 函数的返回值决定了 ngx\_conf\_parse 函数接下来的进一步处理:

情况	返回值 rc	ngx_conf_parse 函数一般情况处理
情况 1	NGX_ERROR	解析异常, return NGX_CONF_ERROR;
情况 2	NGX_CONF_BLOCK_DONE NGX_CONF_FILE_DONE	解析正常, return NGX_CONF_OK
情况 3	NGX_OK NGX_CONF_BLOCK_START	调用 ngx_conf_handler 进行配置文件配置到 nginx 内部控制变量的转换; 继续下一轮 for 循环处理。

讨论情况 3, 我们知道此时解析转换所需要 token 都已经保存到 cf->args 内, 那么接下来就将这些 token 转换为 nginx 内控制变量的值, 执行此逻辑的主要是 ngx\_conf\_handler 函数, 不过在此之前会首先判断 cf->handler 回调函数是否存在, 该回调函数存在的目的是针对类似于 “text/html html htm shtml;” 和 “text/css css;” 这样的 types 配置项, 这些配置项的主要特点是众多且变化不定 (一般可被用户自由配置) 但格式又基本统一, 往往以 key/values 的形式存在, 更重要的是对于这些配置项, nginx 的处理也很简单, 只是拷贝到对应的变量内, 所以这时一般会提供一个统一的 cf->handler 回调函数做这个工作。比如 types 指令的处理函数 ngx\_http\_core\_types 内就对 cf->handler 赋值为 ngx\_http\_core\_type, 这些里面的 mime.types 设置全部由该函数统一处理。

配置转换核心函数 ngx\_conf\_handler 的调入被传入了两个参数, ngx\_conf\_t 类型的 cf 包含有不少重要的信息, 比如转换所需要 token 就保存在 cf->args 内, 而第二个参数无需多说, 记录的是最近一次 token 解析函数 ngx\_conf\_read\_token 的返回值。

前面说过 nginx 的每一个配置指令都对应一个 ngx\_command\_s 数据类型变量, 记录着该配置指令的解析回调函数、转换值存储位置等, 而每一个模块又都把自身所相关的所有指令以数组的形式组织起来, 所有函数 ngx\_conf\_handler 首先做的就是查找当前指令所对应的 ngx\_command\_s 变量, 这通过循环遍历各个模块的指令数组即可, 由于 nginx 的所有模块也是以数组的形式组织起来的, 所有在 ngx\_conf\_handler 函数体内我们可以看到有两个 for 循环的遍历查找:

```
for (i = 0; ngx_modules[i]; i++) {
    ...
    cmd = ngx_modules[i]->commands;
    for (/* void */; cmd->name.len; cmd++) {
        ...
    }
}
```

两个 for 循环的结束判断之所以可以这样写, 是因为这些数组都带有对应的末尾哨兵。具体代码里面还有一些有效性判断 (比如当前模块类型、指令名称、项目值个数、指令位置)

等操作，虽然繁琐但并没有难点所以忽略不讲，直接看里面的函数调用：

```
393: Filename : ngx_conf_file.c
```

```
394: rv = cmd->set(cf, cmd, conf);
```

当代码执行到这里，所以 `ngx` 已经查找到配置指令所对应的 `ngx_command_s` 变量 `cmd`，所以这里就开始调用回调函数进行处理，以配置项目 “`worker_processes 2;`” 为例，对应的 `ngx_command_s` 变量为：

```
69: Filename : ngx.c
```

```
70: { ngx_string("worker_processes"),
```

```
71:     NGX_MAIN_CONF|NGX_DIRECT_CONF|NGX_CONF_TAKE1,
```

```
72:     ngx_conf_set_num_slot,
```

```
73:     0,
```

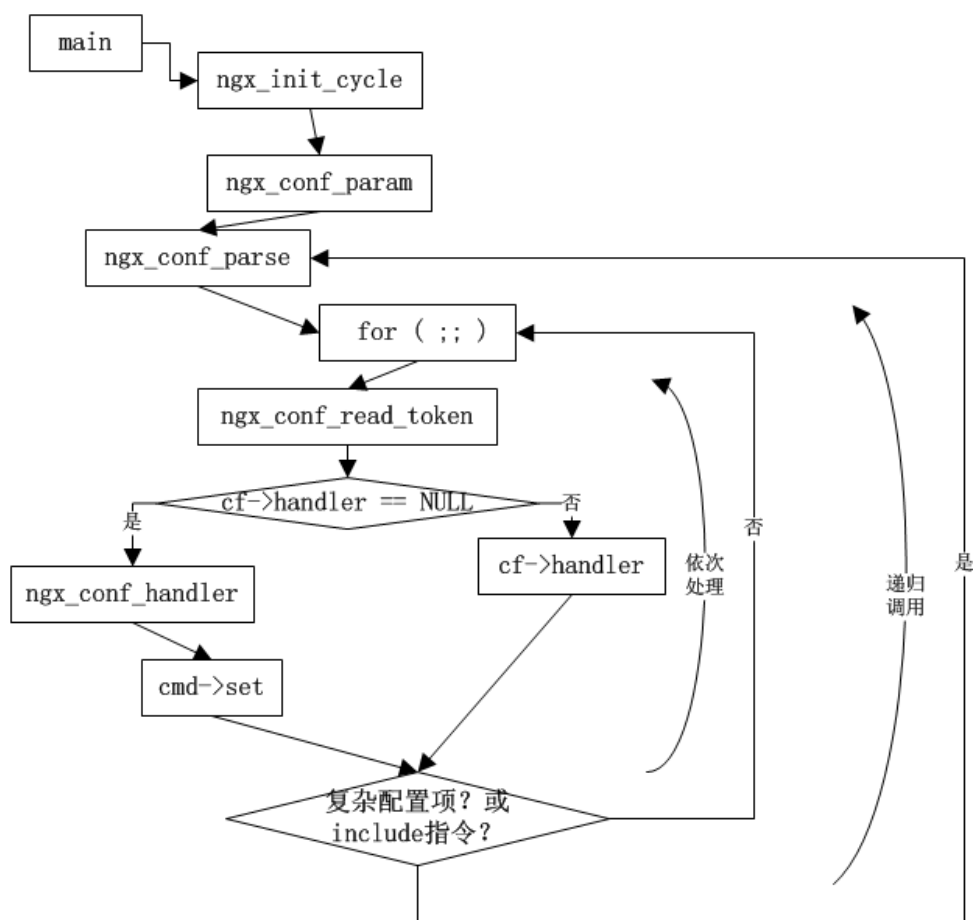
```
74:     offsetof(ngx_core_conf_t, worker_processes),
```

```
75:     NULL },
```

那么其回调函数为 `ngx_conf_set_num_slot`，这是一个比较公共的配置项处理函数，也就是那种数字的配置项目都可以使用该函数进行转换，该函数的内部逻辑非常简单，首先找到转换后值的存储位置，然后利用 `ngx_atoi` 函数把字符串的数字转换为整型的数字，存储到对应位置；这就完成了从配置文件里的 “`worker_processes 2;`” 到 `ngx` 里 `ngx_core_conf_t` 结构体类型变量 `conf` 的 `worker_processes` 字段控制值的转换。

`worker_processes` 指令的回调处理函数比较简单，对于复杂配置项，比如 `server` 指令的回调处理函数 `ngx_http_core_server` 就要复杂得多，比如它会申请内存空间（以便存储其包含的简单配置项的控制值）、会调用 `ngx_conf_parse` 等，这些就留在需要的时候再做阐述吧。

对于 `ngx` 配置文件的解析流程基本就是如此，上面的介绍忽略了很多细节，前面也说过，事实上对于配置信息解析的代码（即各种各样的回调函数 `cmd->set` 的具体实现）占去了 `ngx` 大量的源代码，而我们这里并没有做过多的分析，仅例举了 `worker_processes` 配置指令的简单解析过程。虽然对于不同的配置项，解析代码会根据自身应用不同而不同，但基本框架就是如此了。最后，看一个 `ngx` 配置文件解析的流程图，如下：



## 配置信息组织结构

这里讲的配置信息已不再是配置文件里的内容（比如 `daemon off;`），而是指在 `nginx` 的执行环境里作为特定变量值的存在（比如 `ngx_flag_t daemon;`）。虽然前面已经描述了从各个配置项到特定变量值的转换过程，但并没有详细阐明这些控制变量的整体组织结构，下面就尝试描述这部分内容。

`nginx` 内部对配置信息的组织首先是根据上下级别来区分的，也就是所谓的配置上下文，以 `http` 服务为例，最外层是 `main` 上下文、`http` 指令的 `block` 块内为 `http` 上下文、接着是 `server` 上下文、`location` 上下文，之所以说是按上下级别来区分是因为 `main`、`http`、`server`、`location` 之间存在严格的包含与被包含关系，比如 `http` 包含 `server`、`server` 包含 `location`，这个无需累述；配置信息的组织还是按模块来划分的，这体现在每一平行级别上，也就是说对于所有 `main` 上下文里的配置，是根据模块来划分组织的，这是自然而然的事情，因为 `nginx` 代码本身也进行了模块化划分，而用户传递进来的配置信息说到底要被这些模块代码使用，为了让模块更方便的找到与自己相关的配置信息，那么直接根据模块来组织配置信息是合理的。不会出现多个模块共用一个配置值的情况呢？按理不会，如果出现这种情况就说明模块的划

分不恰当导致模块之间耦合性太强。看具体实现，首先是：

```

187: Filename : ngx_cycle.c
188: cycle->conf_ctx = ngx_palloc(pool, ngx_max_module * sizeof(void *));
189: ...
215: for (i = 0; ngx_modules[i]; i++) {
216:     if (ngx_modules[i]->type != NGX_CORE_MODULE) {
217:         continue;
218:     }
219:
220:     module = ngx_modules[i]->ctx;
221:
222:     if (module->create_conf) {
223:         rv = module->create_conf(cycle);
224:         if (rv == NULL) {
225:             ngx_destroy_pool(pool);
226:             return NULL;
227:         }
228:         cycle->conf_ctx[ngx_modules[i]->index] = rv;
229:     }
230: }
231: ...
251: conf.ctx = cycle->conf_ctx;
252: ...
262: if (ngx_conf_param(&conf) != NGX_CONF_OK) {
263: ...
268: if (ngx_conf_parse(&conf, &cycle->conf_file) != NGX_CONF_OK) {
269: ...

```

第 188 行代码申请存储模块配置信息的内存空间，可以看到这是一个指针数组，数组元素个数为 `ngx_max_module`，刚好一个指针元素可以对应一个模块，后续这些指针就指向其对应模块配置信息的具体存储位置。

第 215 行的 `for` 循环主要是为了调用核心模块的 `create_conf()` 函数，创建实际的配置信息存储空间。为什么先只处理核心模块呢？因为核心模块才是基本模块，它们的配置空间必须首先创建，以便作为其它非核心模块的依赖。对于 `for` 循环内两个 `if` 判断的理解，是因为不是所有的模块都是核心模块，也不是所有的核心模块都有 `create_conf()` 函数，比如虽然模块 `ngx_http_module` 和模块 `ngx_mail_module` 都是核心模块，但它们却并没有 `create_conf()` 函数，因为这两个模块是否真正使用依赖于具体的配置文件，如果配置文件里并没有配置 `http`，但 `nginx` 代码却先在这里把 `http` 的配置信息存储空间申请出来而后面又完全不用，那岂不是多此一举？所以，这两个核心模块的配置信息存储空间会在配置文件的解析过程中根据需要申请。第 223 行的存储空间若创建成功，那么第 228 行就把它赋值给对应的指针元素，完成前面所说的那样。



以核心模块 `ngx_core_module` 为例,从名字就可以看出这是一个特别基础且重要的核心模块,模块序号 `index` 为 0,而 `create_conf` 回调指针指向函数 `ngx_core_module_create_conf()`:

```
924: Filename : nginx.c
925: static void *
926: ngx_core_module_create_conf(ngx_cycle_t *cycle)
927: {
928:     ngx_core_conf_t  *ccf;
929:
930:     ccf = ngx_palloc(cycle->pool, sizeof(ngx_core_conf_t));
931:     ...
945:     ccf->daemon = NGX_CONF_UNSET;
946:     ccf->master = NGX_CONF_UNSET;
947:     ...
970:     return ccf;
971: }
```

这个函数主要做了一件事情,申请内存空间、初始内存空间并返回内存空间的指针引用。注意类似于 `NGX_CONF_UNSET` 这样的初始赋值,这很重要,根据名称就能猜出这些值可用来判断用户是否有在配置文件里对这些配置项做过设置,因为这些值都是特殊值-1(用户的合法设置不会有-1的情况),所以如果用户没做设置,那么在配置文件解析完后,对应的字段值仍然为-1,如果此时在其它配置设定下,正常运行 `nginx` 需要这些字段,那么就需给这些字段设置对应的默认值。设置默认值的处理在模块的回调函数 `init_conf()`内,在配置文件解析完(有的只是对应的配置块解析完,比如 `http`、`events` 配置块,前面提到的 `create_conf()`也是如此,比如 `http` 配置块的 `create_main_conf()`、`init_main_conf()`等,但默认值的设定肯定是在对应的依赖配置内容已经全部解析完后才进行的)后就会调用该函数:

```
278: Filename : ngx_cycle.c
279:     for (i = 0; ngx_modules[i]; i++) {
280: ...
286:         if (module->init_conf) {
287:             if (module->init_conf(cycle, cycle->conf_ctx[ngx_modules[i]->index])
288:                 == NGX_CONF_ERROR)
289: ...
```

看看核心模块 `ngx_core_module` 的默认值设置 `ngx_core_module_init_conf()`函数:

```
973: Filename : nginx.c
974: static char *
975: ngx_core_module_init_conf(ngx_cycle_t *cycle, void *conf)
976: {
977:     ngx_core_conf_t  *ccf = conf;
978:
979:     ngx_conf_init_value(ccf->daemon, 1);
980:     ngx_conf_init_value(ccf->master, 1);
981: ...
```

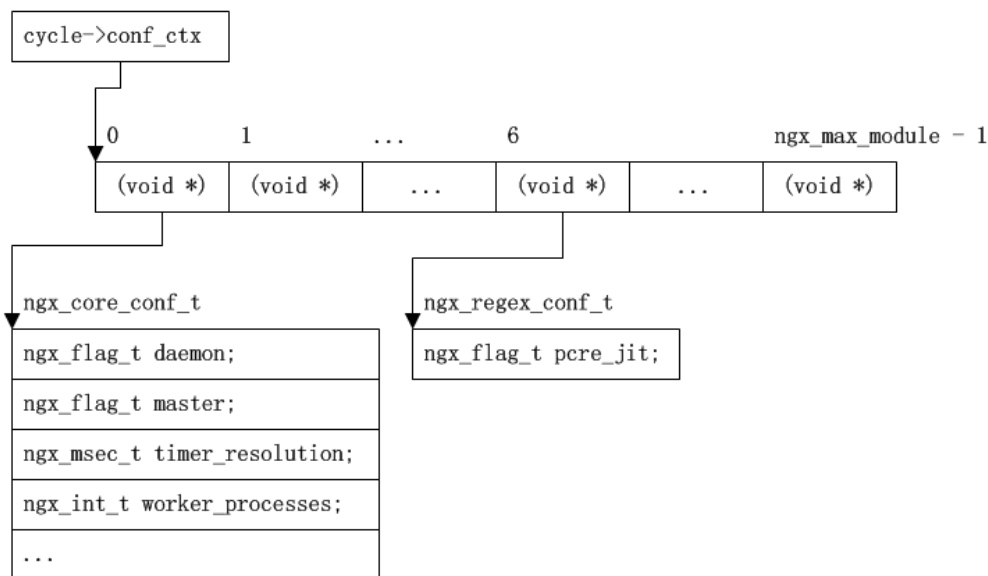
```

229: Filename : ngx_conf_file.h
230: #define ngx_conf_init_value(conf, default)          \
231:     if (conf == NGX_CONF_UNSET) {                    \
232:         conf = default;                               \
233:     }

```

前后一连贯，这部分逻辑就应该很容易懂了，比如如果用户没有对 `daemon` 做设置，那么它的值就还是 `NGX_CONF_UNSET`，进行就把它设置为 `default` 默认值，也就是 1。其它字段的默认值处理也与此类似。

回过头来接着看，前面提到的两段相关源码执行之后，我们目前所了解的配置信息最基本组织结构如下图所示：



可以看到只有两个核心模块 `ngx_core_module` 和 `ngx_regex_module` 有对应的 `create_conf` 回调函数，申请的配置存储空间“挂载”在对应的数组元素下。当然，这只是我这里的 `nginx` 模块情况（请参考附录 A），也许你那因为 `configure` 编译设置不同而有所不同，不过可以肯定结构都是这样了。

再来看源文件 `ngx_cycle.c` 的第 251 行和第 268 行（第 262 行是对通过 `nginx` 命令行传过来的配置信息的处理，和第 268 行将执行的逻辑一样，而且应该是更简单一点，所以略过），因为 `cycle->conf_ctx` 是唯一能正确找到配置存储空间的指针，不能把它弄乱，所以把它赋值给 `conf.ctx` 供后续使用，`conf.ctx` 也就是类似于一个临时变量，不管后续代码怎样修改它（这个值也的确会随着配置文件的解析、配置上下文的切换而变化），我们的 `cycle->conf_ctx` 不变，如第 268 行所看到得那样，`ngx_conf_parse()` 的第一个参数就是 `conf` 的引用，该函数再通过函数调用，把 `conf` 又传递到函数 `ngx_conf_handler()` 内：

```

101: Filename : ngx_conf_file.c
102: char *
103: ngx_conf_parse(ngx_conf_t *cf, ngx_str_t *filename)
104: {

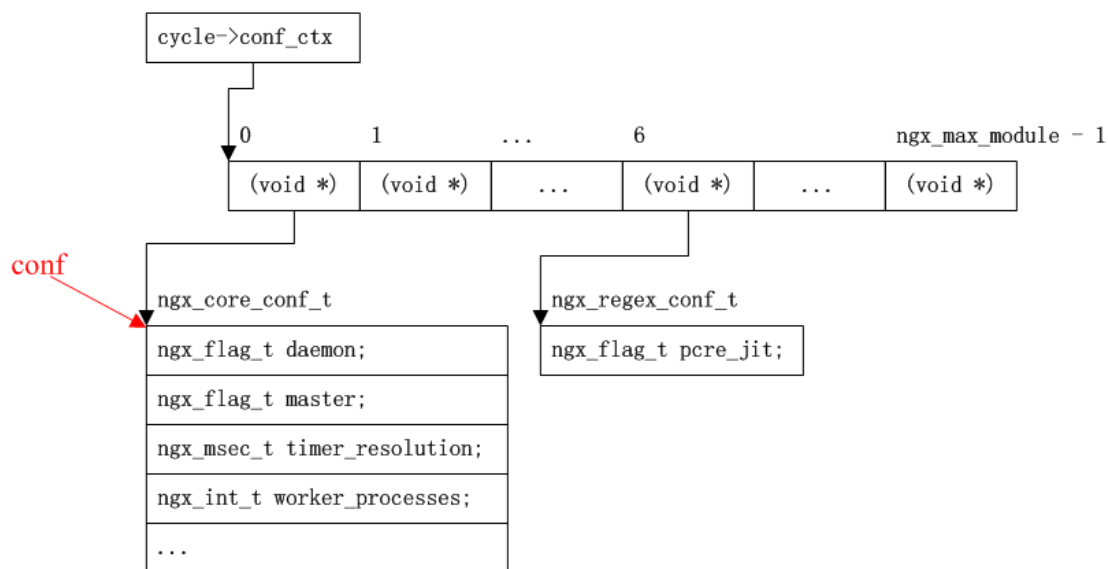
```

```

105: ...
244:         rc = ngx_conf_handler(cf, rc);
277: }
278:
279:
280: static ngx_int_t
281: ngx_conf_handler(ngx_conf_t *cf, ngx_int_t last)
282: {
283: ...
376:         /* set up the directive's configuration context */
377:
378:         conf = NULL;
379:
380:         if (cmd->type & NGX_DIRECT_CONF) {
381:             conf = ((void **) cf->ctx)[ngx_modules[i]->index];
382:
383:         } else if (cmd->type & NGX_MAIN_CONF) {
384:             conf = &(((void **) cf->ctx)[ngx_modules[i]->index]);
385:
386:         } else if (cf->ctx) {
387:             confp = *(void **) ((char *) cf->ctx + cmd->conf);
388:
389:             if (confp) {
390:                 conf = confp[ngx_modules[i]->ctx_index];
391:             }
392:         }
393:
394:         rv = cmd->set(cf, cmd, conf);
395: ...
431: }

```

第 378-392 行的代码为我们关注的重点，看第 380 行的 if 判断，什么样的配置项类型是 NGX\_DIRECT\_CONF 的？搜索一下 nginx 的所有代码，发现只有核心模块的配置项才可能是这个类型，比如 ngx\_core\_module 模块的 daemon 和 master\_process 等、ngx\_openssl\_module 模块的 ssl\_engine、ngx\_regex\_module 模块的 pcre\_jit。从前面分析，我们已经知道这些核心模块的配置存储空间已经申请了，所有其配置项的转换后值已有存储的地方，看第 381 行给 conf 赋值语句，以 ngx\_core\_module 模块为例，那么 conf 指针的指向当前如下所示：



```

41: Filename : nginx.c
42:     { ngx_string("master_process"),
43:       NGX_MAIN_CONF|NGX_DIRECT_CONF|NGX_CONF_FLAG,
44:       ngx_conf_set_flag_slot,
45:       0,
46:       offsetof(ngx_core_conf_t, master),
47:       NULL },

```

```

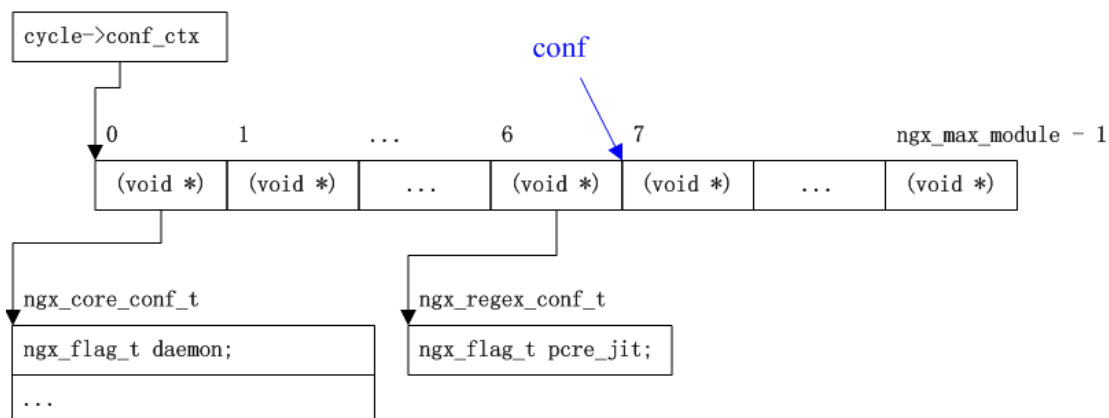
1041: Filename : ngx_conf_file.c
1042: char *
1043: ngx_conf_set_flag_slot(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
1044: {
1045:     char *p = conf;
1046:     ...
1048:     ngx_flag_t *fp;
1049:     ...
1051:     fp = (ngx_flag_t *) (p + cmd->offset);
1052:     ...
1059:     if (ngx_strcasecmp(value[1].data, (u_char *) "on") == 0) {
1060:         *fp = 1;
1061:     } else if (ngx_strcasecmp(value[1].data, (u_char *) "off") == 0) {
1062:         *fp = 0;
1063:     }
1064:     ...

```

上面两段代码显示了配置项 `master_process` 的转换与存储过程，第 1045 与 1051 行结合起来找到 `master_process` 转换后值的存储位置，而 1059 到 1063 完成转换（on 为 1，off 为 0）与存储。

接着看 `ngx_conf_file.c` 源码的第 383 行，有哪些配置项被打上了 `NGX_MAIN_CONF` 标签

而又不是 NGX\_DIRECT\_CONF 的？http、mail、events、error\_log 等，其中前面三个的处理比较类似，以 http 配置项的处理为例，我们知道 ngx\_http\_module 虽然是核心模块，但是其配置存储空间是还没有实际申请的，所以看第 384 行给 conf 进行赋值的语句右值是数组元素的地址，由于 ngx\_http\_module 模块对应 7 号数组元素，所以 conf 指针的指向当前如下所示：

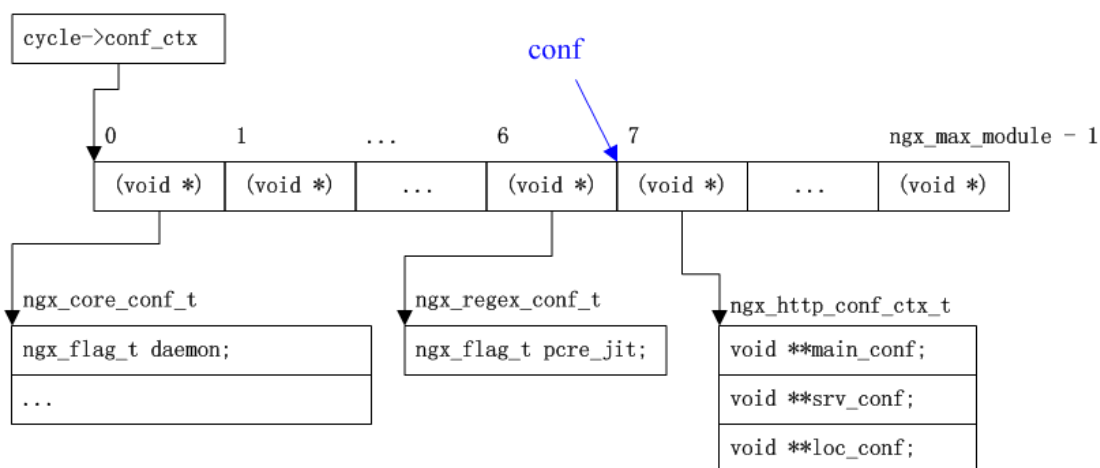


```

83: Filename : ngx_http.c
84:     { ngx_string("http"),
85:       NGX_MAIN_CONF|NGX_CONF_BLOCK|NGX_CONF_NOARGS,
86:       ngx_http_block,
87:       0,
88:       0,
89:       NULL },
90: ...
118: static char *
119: ngx_http_block(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
120: {
121: ...
125:     ngx_http_conf_ctx_t      *ctx;
126: ...
132:     ctx = ngx_palloc(cf->pool, sizeof(ngx_http_conf_ctx_t));
133: ...
137:     *(ngx_http_conf_ctx_t **) conf = ctx;
138:

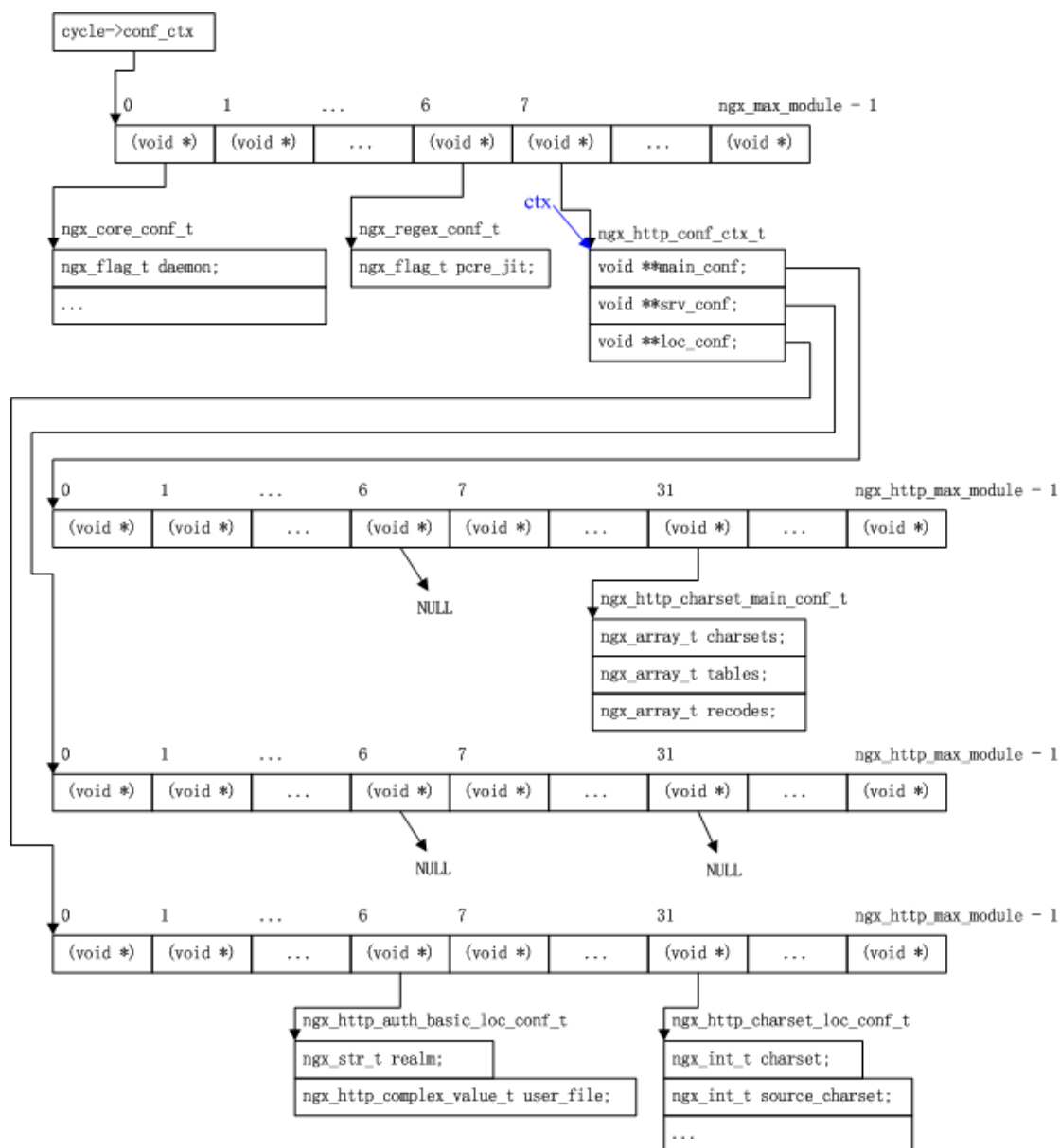
```

第 132 行申请了内存空间，而第 137 行通过 conf 参数间接的把这块内存空间“挂载”在 7 号数组元素下。对于多级指针，大多数人都容易搞混乱，如果没有理解，请仔细思考一下上面的指针操作。经过 ngx\_http\_block 的处理，我们能看到的配置信息最基本组织结构如下图所示：



对于 `ngx_http_module` 模块的内部配置，除了 `main_conf` 配置外，为什么还有 `srv_conf`、`loc_conf` 是因为这两个字段里存储的配置信息是针对 `server`、`location` 应用的 `http` 全局配置。这些配置信息在结构上的组织和 `cycle->conf_ctx` 类似，仍然是根据模块来划分，当然只是 `NGX_HTTP_MODULE` 类型的模块，如果要画个图示，那么就是这样：





NGX\_HTTP\_MODULE 类型模块具有哪种范围域的配置信息就将申请的内存空间“挂载”在对应的数组元素下（如果它在 http 上下文环境里配置），虽然大多数模块都只有一种，比如 ngx\_http\_auth\_basic\_module 模块只有 loc\_conf 配置项，但 ngx\_http\_charset\_filter\_module 模块却有 main\_conf 和 loc\_conf 两类配置项，如上图中显示的那样（在整个 NGX\_HTTP\_MODULE 类型模块中排序中，ngx\_http\_auth\_basic\_module 模块序号为 6、ngx\_http\_charset\_filter\_module 模块序号为 31，上图中只画出了这个两个示例模块的情况）。继续看 ngx\_http\_block 函数的处理：

```

117: Filename : ngx_http.c
118: static char *
119: ngx_http_block(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
120: {
121: ...
218:     pcf = *cf;

```

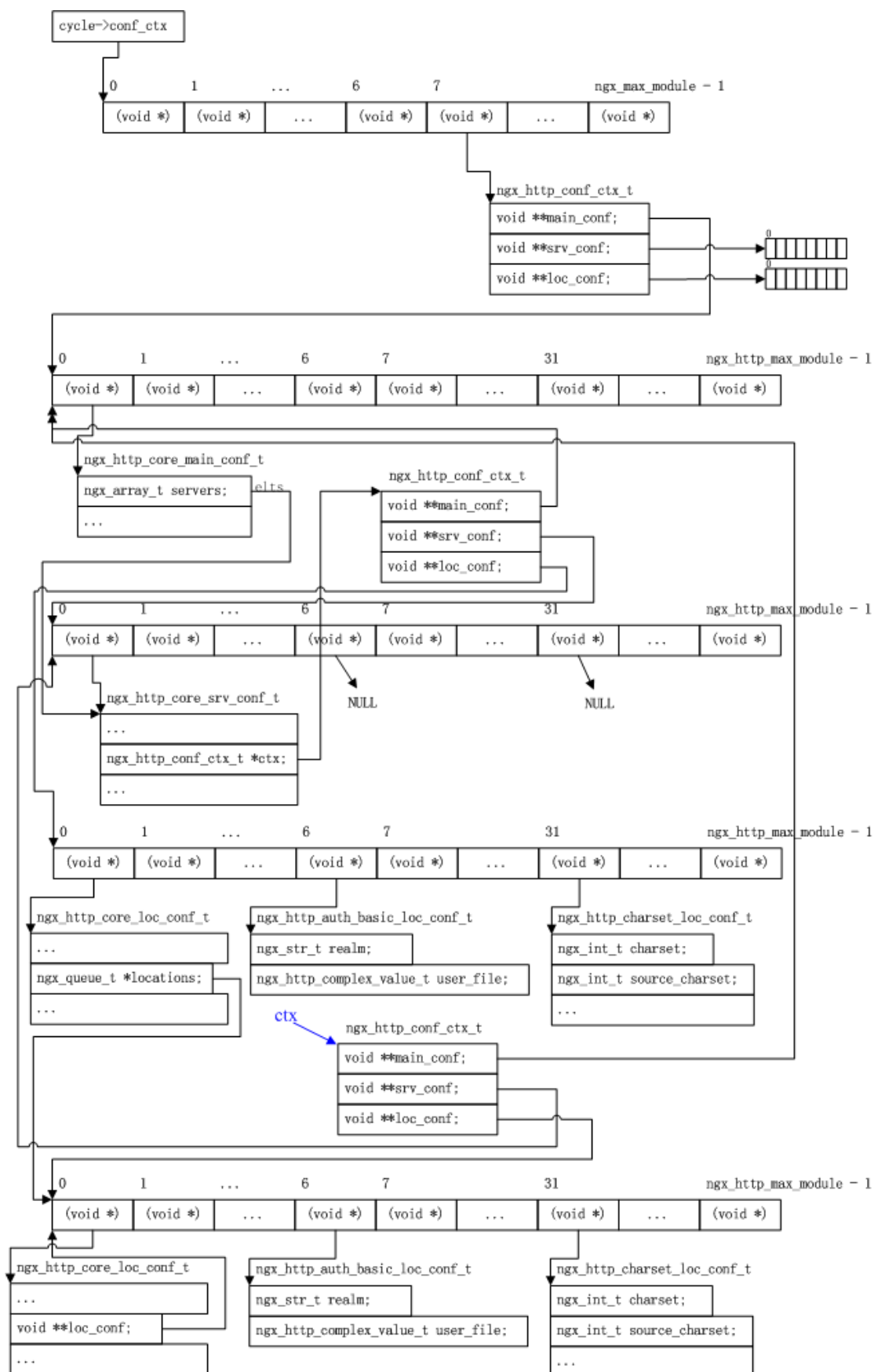
```
219:     cf->ctx = ctx;
220: ...
235:     /* parse inside the http{} block */
236:
237:     cf->module_type = NGX_HTTP_MODULE;
238:     cf->cmd_type = NGX_HTTP_MAIN_CONF;
239:     rv = ngx_conf_parse(cf, NULL);
240: ...
325:     *cf = pcf;
326: ...
```

第 218 行把 cf 值（注意指针取值符号\*，所以这里是进行的结构体赋值操作）保存起来，而第 325 行进行恢复，前面曾说过在配置文件解析的过程中，cf->ctx 会随着上下文的切换而改变，第 219 行就可以看到这点，此时 cf->ctx 和上图中蓝色箭头指向一致。第 239 行调入到 ngx\_conf\_parse 后，当前配置上下文环境就从 main 切换到 http，如果在接下来的解析过程中遇到 server 指令，其指令处理函数 ngx\_http\_core\_server()，类似于 http 指令的处理，对于 server 上下文这一同级别的所有配置同样也是按照模块划分来组织的：

在 `server` 上下文里不再有 `http` 全局配置，所以其 `main_conf` 字段直接指向 `http` 上下文的 `main_conf` 即可。

```
2784:         cf->cmd_type = NGX_HTTP_SRV_CONF;
2785:
2786:         rv = ngx_conf_parse(cf, NULL);
2787:
2788:         *cf = pcf;
2789:     ...
```

第 2786 行调入到 `ngx_conf_parse()` 后，当前配置上下文环境就从 `http` 切换到 `server`，如果在接下来的解析过程中遇到 `location` 指令，其指令处理函数 `ngx_http_core_location ()`，类似于 `http` 指令、`server` 指令的处理，对于 `location` 上下文这一同级别的所有配置同样也是按照模块划分来组织：



依旧是进行上下文的切换（第 3007 和 3008 行），然后调用 `ngx_conf_parse()` 函数继续处理：

```

2824:  Filename : ngx_http_core_module.c
2825:  static char *
2826:  ngx_http_core_location(ngx_conf_t *cf, ngx_command_t *cmd, void *dummy)
2827:  {
2828:  ...
3007:      save = *cf;
3008:      cf->ctx = ctx;
3009:      cf->cmd_type = NGX_HTTP_LOC_CONF;
3010:
3011:      rv = ngx_conf_parse(cf, NULL);
3012:
3013:      *cf = save;
3014:  ...

```

可以看到不管是 http 上下文还是 server 上下文、location 上下文, 调入到 ngx\_conf\_parse() 函数内后, cf->ctx 指向的都是一个 ngx\_http\_conf\_ctx\_t 结构体, 如果此时从 ngx\_conf\_parse() 函数再调入到 ngx\_conf\_handler() 函数, 此时情况是怎么样呢? 回过头来看 ngx\_conf\_file.c 源码的第 386 行, 这是第三种情况, 在前面两个 if 都不匹配的情况下再来进行这个判断, 通过查看 http 模块配置项的 type 字段发现这些配置项的 ngx\_conf\_handler() 函数处理都会进入到这个判断里, 看个实例:

```

138: Filename : ngx_http_charset_filter_module.c
139:  { ngx_string("charset"),
140:
141:      NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF
142:      |NGX_HTTP_LIF_CONF|NGX_CONF_TAKE1,
143:      ngx_http_set_charset_slot,
144:      NGX_HTTP_LOC_CONF_OFFSET,
145:      offsetof(ngx_http_charset_loc_conf_t, charset),
146:      NULL },

```

配置项 charset 的 type 既不包含 NGX\_DIRECT\_CONF 旗标又不包含 NGX\_MAIN\_CONF 旗标, 所以进入到第 386 行的判断里:

```

385: Filename : ngx_conf_file.c
386:  } else if (cf->ctx) {
387:      confp = *(void **) ((char *) cf->ctx + cmd->conf);
388:
389:      if (confp) {
390:          conf = confp[ngx_modules[i]->ctx_index];
391:      }
392:  }

```

从配置项 charset 的 type 字段里还可以看出它可以在多个上下文里使用, 但如前所述, 不管当前是在哪个上下文里, cf->ctx 指向的都是一个 ngx\_http\_conf\_ctx\_t 结构体, 配置项 charset 的 conf 字段为 NGX\_HTTP\_LOC\_CONF\_OFFSET, 也就是:

51: Filename : ngx\_http\_config.h

52: #define NGX\_HTTP\_LOC\_CONF\_OFFSET           offsetof(ngx\_http\_conf\_ctx\_t,  
loc\_conf)

即取 ngx\_http\_conf\_ctx\_t 结构体的字段 loc\_conf 偏移量，那么第 387 行代码也就是获取指针字段 loc\_conf 所指向的数组，再由第 390 行根据模块序号获取对应的数组元素，这就和本节最开始讲述的情况统一起来了。



## 第四章 模块解析

### Nginx 模块综述

nginx 的模块非常之多，可以认为所有代码都是以模块的形式组织，这包括核心模块和功能模块，针对不同的应用场合，并非所有的功能模块都要被用到，附录 A 给出的是默认 configure（即简单的 http 服务器应用）下被连接的模块，这里虽说是模块连接，但 nginx 不会像 apache 或 lighttpd 那样在编译时生成 so 动态库而在程序执行时再进行动态加载，nginx 模块源文件会在生成 nginx 时就直接被编译到其二进制执行文件中，所以如果要选用不同的功能模块，必须对 nginx 做重新配置和编译。对于功能模块的选择，如果要修改默认值，需要在进行 configure 时进行指定，比如新增 http\_flv 功能模块（默认是没有这个功能的，各个选项的默认值可以在文件 auto/options 内看到）：

```
[root@localhost nginx-1.2.0]# ./configure --with-http_flv_module
```

执行后，生成的 objs/nginx\_modules.c 文件内就包含有对 ngx\_http\_flv\_module 模块的引用了，要再去掉 http\_flv 功能模块，则需要重新 configure，即不带 --with-http\_flv\_module 配置后再编译生成新的 nginx 执行程序。通过执行 ./configure --help，我们可以看到更多的配置选项。

虽然 Nginx 模块有很多，并且每个模块实现的功能各不相同，但是根据模块的功能性质，可以将它们分为四个类别：

- 1, handlers: 处理客户端请求并产生待响应内容，比如 ngx\_http\_static\_module 模块，负责客户端的静态页面请求处理并将对应的磁盘文件准备为响应内容输出。
- 2, filters: 对 handlers 产生的响应内容做各种过滤处理（即是增删改），比如模块 ngx\_http\_not\_modified\_filter\_module，对待响应内容进行过滤检测，如果通过时间戳判断出前后两次请求的响应内容没有发生任何改变，那么可以直接响应“304 Not Modified”状态标识，让客户端使用缓存即可，而原本待发送的响应内容将被清除掉。
- 3, upstream: 如果存在后端真实服务器，nginx 可利用 upstream 模块充当反向代理（Proxy）的角色，对客户端发起的请求只负责进行转发（当然也包括后端真实服务器响应的回转），比如 ngx\_http\_proxy\_module 就为标准的代理模块。
- 4, load-balance: 在 nginx 充当中间代理时，由于后端真实服务器往往多于一个，对于某一次客户端的请求，如何选择对应的后端真实服务器来进行处理，这就有类似于 ngx\_http\_upstream\_ip\_hash\_module 这样的模块来实现不同的负载均衡算法（Load Balance）。

对于这几类模块，我们马上会分别进行详细介绍并分析各自典型代表模块，不过在此之前先从 nginx 模块源码上来进行直观认识。前面讲过 nginx 的所有代码都是以模块形式进行

组织，而封装 nginx 模块的结构体为 ngx\_module\_s，定义如下：

```

110: Filename : ngx_conf_file.h
111: struct ngx_module_s {
112:     ngx_uint_t      ctx_index;    //当前模块在同类模块中的序号
113:     ngx_uint_t      index;        //当前模块在所有模块中的序号
114:     ...
120:     ngx_uint_t      version;      //当前模块版本号
121:
122:     void             *ctx;         //指向当前模块特有的数据
123:     ngx_command_t    *commands;    //指向当前模块配置项解析数组
124:     ngx_uint_t       type;         //模块类型
125:     //以下为模块回调函数，回调时机可根据函数名看出
126:     ngx_int_t        (*init_master)(ngx_log_t *log);
127:     ...
128: };
11:  Filename : ngx_core.h
12:  typedef struct ngx_module_s      ngx_module_t;
    
```

结构体 ngx\_module\_s 值得关注的几个字段分别为 ctx、commands、type，其中 commands 字段表示当前模块可以解析的配置项目，这在配置文件解析一章做过详细描述；表示模块类型的 type 值只有 5 种可能的值，而同一类型模块的 ctx 指向的数据类型也相同：

序号	type 值	ctx 指向数据类型
1	NGX_CORE_MODULE	ngx_core_module_t
2	NGX_EVENT_MODULE	ngx_event_module_t
3	NGX_CONF_MODULE	NULL
4	NGX_HTTP_MODULE	ngx_http_module_t
5	NGX_MAIL_MODULE	ngx_mail_module_t

上表中第三列里的数据类型非常重要，它们的字段基本都是一些回调函数，这些回调函数会在其模块对应的配置文件解析过程前/中/后会适时的被调用，做一些内存准备、初始化、配置值检查、初始值填充与合并、回调函数挂载等初始工作，以 ngx\_http\_core\_module 模块为例，该模块 type 类型为 NGX\_HTTP\_MODULE，ctx 指向的 ngx\_http\_module\_t 结构体变量 ngx\_http\_core\_module\_ctx：

```

785: Filename : ngx_http_core_module.c
786: static ngx_http_module_t ngx_http_core_module_ctx = {
787:     ngx_http_core_preconfiguration,    /* preconfiguration */
788:     NULL,                               /* postconfiguration */
789:
790:     ngx_http_core_create_main_conf,    /* create main configuration */
791:     ngx_http_core_init_main_conf,      /* init main configuration */
792:
793:     ngx_http_core_create_srv_conf,     /* create server configuration */
    
```

```

794:     ngx_http_core_merge_srv_conf,           /* merge server configuration */
795:
796:     ngx_http_core_create_loc_conf,           /* create location configuration */
797:     ngx_http_core_merge_loc_conf             /* merge location configuration */
798: };

```

根据上面代码注释，可以很明显的看出各个回调函数的回调时机，比如函数 `ngx_http_core_preconfiguration()` 将在进行 http 块配置解析前被调用，所以在 `ngx_http_block()` 函数里可以看到这样的代码：

```

117: Filename : ngx_http.c
118: static char *
119: ngx_http_block(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
120: ...
228:     if (module->preconfiguration) {
229:         if (module->preconfiguration(cf) != NGX_OK) {
230:             return NGX_CONF_ERROR;
231:         }
232:     }
233: ...
239:     rv = ngx_conf_parse(cf, NULL);
240: ...
309:     if (module->postconfiguration) {
310:         if (module->postconfiguration(cf) != NGX_OK) {
311:             return NGX_CONF_ERROR;
312:         }
313:     }
314: ...

```

至于这些回调函数内的具体逻辑，如前所述一般是一些初始或默认值填充工作，但也有回调函数挂载的设置，比如 `ngx_http_static_module` 模块的 `postconfiguration` 字段回调函数 `ngx_http_static_init()` 就是将自己的处理函数 `ngx_http_static_handler()` 挂载在 http 处理状态机上，但总体来看这毕竟都只是一些简单的初始准备工作，不多累述。

## Handler 模块

对于客户端 http 请求的处理过程，为了获得更强的控制能力，Nginx 将其细分为多个阶段，每一个阶段可以有零个或多个回调函数进行专门处理，当我们在编写自己的 `handlers` 类型模块时，必须把模块功能处理函数挂载在正确的阶段点上，如前面所述的模块 `ngx_http_static_module` 就将自己的模块功能处理函数 `ngx_http_static_handler()` 挂载在 `NGX_HTTP_CONTENT_PHASE` 阶段。这在提供很大灵活性的同时，也极大的增加了编写自定义模块的困难，不过在详细了解每一个处理阶段之后，这种困难也许没有想象中的那么

大。

Http 请求处理过程一共分为 11 个阶段，每一个阶段对应的处理功能都比较单一，这样能让 nginx 模块代码更为内聚：

序号	阶段宏名	阶段描述
0	NGX_HTTP_POST_READ_PHASE	读取请求内容阶段
1	NGX_HTTP_SERVER_REWRITE_PHASE	Server 请求地址重写阶段
2	NGX_HTTP_FIND_CONFIG_PHASE	配置查找阶段
3	NGX_HTTP_REWRITE_PHASE	Location 请求地址重写阶段
4	NGX_HTTP_POST_REWRITE_PHASE	请求地址重写提交阶段
5	NGX_HTTP_PREACCESS_PHASE	访问权限检查准备阶段
6	NGX_HTTP_ACCESS_PHASE	访问权限检查阶段
7	NGX_HTTP_POST_ACCESS_PHASE	访问权限检查提交阶段
8	NGX_HTTP_TRY_FILES_PHASE	配置项 try_files 处理阶段
9	NGX_HTTP_CONTENT_PHASE	内容产生阶段
10	NGX_HTTP_LOG_PHASE	日志模块处理阶段

并非每一个阶段都能去挂载自定义的回调函数，比如 NGX\_HTTP\_TRY\_FILES\_PHASE 阶段就是针对配置项 try\_files 的特定处理阶段，而 NGX\_HTTP\_FIND\_CONFIG\_PHASE、NGX\_HTTP\_POST\_ACCESS\_PHASE 与 NGX\_HTTP\_POST\_REWRITE\_PHASE 这三个阶段也是为了完成 nginx 特定的功能，就算给这几个阶段加上回调函数，也永远不会被调用。一般情况下，我们的自定义模块回调函数挂载在 NGX\_HTTP\_CONTENT\_PHASE 阶段的情况比较多，毕竟大部分情况下的业务需求是修改 HTTP 响应数据，nginx 自身的产生响应内容的模块，像 ngx\_http\_static\_module、ngx\_http\_random\_index\_module、ngx\_http\_index\_module、ngx\_http\_gzip\_static\_module、ngx\_http\_dav\_module 等都是挂载在这个阶段。

大多数情况下，功能模块会在其对应配置解析完后的回调函数，也就是 ngx\_http\_module\_t 结构体的 postconfiguration 字段指向的函数内将当前模块的回调功能函数挂载到这 11 个阶段的其中一个上，看个示例：

```

16: Filename : ngx_http_static_module.c
17: ngx_http_module_t ngx_http_static_module_ctx = {
18:     NULL,                                     /* preconfiguration */
19:     ngx_http_static_init,                     /* postconfiguration */
20:     ...
270: static ngx_int_t
271: ngx_http_static_init(ngx_conf_t *cf)
272: {
273:     ngx_http_handler_pt      *h;
274:     ngx_http_core_main_conf_t *cmcf;
275:

```

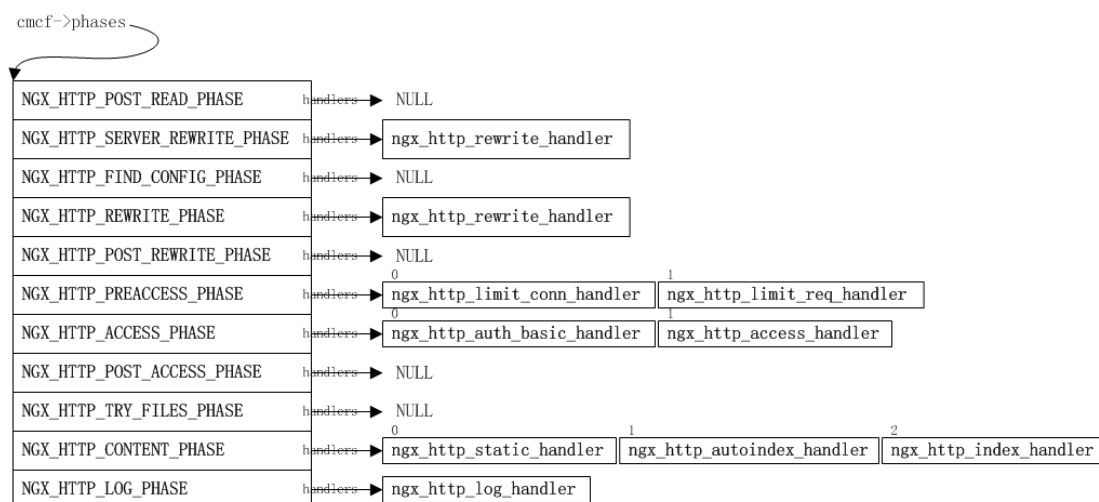
```

276:    cmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_core_module);
277:
278:    h=ngx_array_push(&cmcf->phases[NGX_HTTP_CONTENT_PHASE].handlers);
279:    if (h == NULL) {
280:        return NGX_ERROR;
281:    }
282:
283:    *h = ngx_http_static_handler;
284:
285:    return NGX_OK;
286: }

```

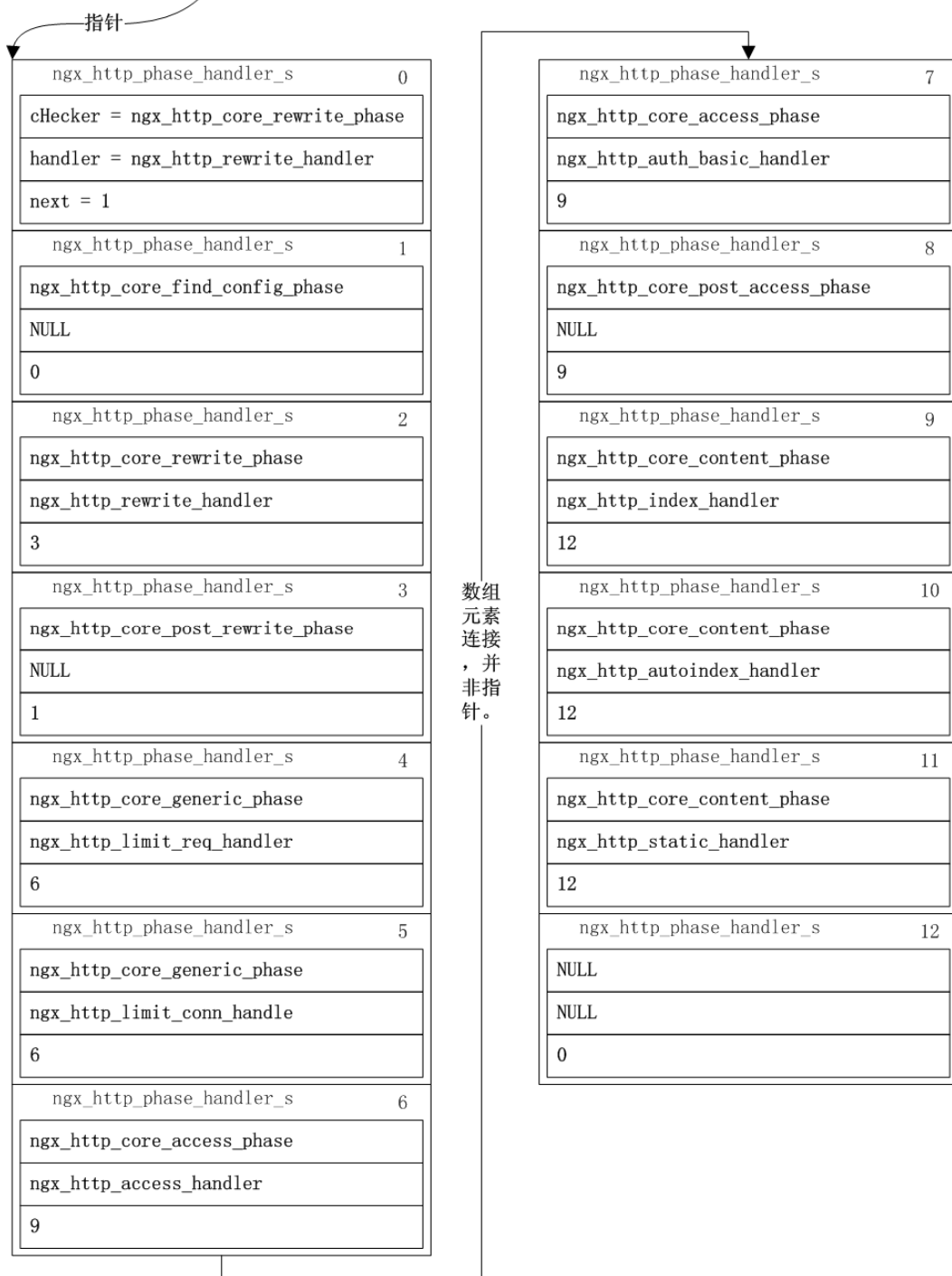
在模块 `ngx_http_static_module` 的 `postconfiguration` 回调函数 `ngx_http_static_init()` 内, 将 `ngx_http_static_module` 模块的核心功能函数 `ngx_http_static_handler()` 挂载在 Http 请求处理流程中的 `NGX_HTTP_CONTENT_PHASE` 阶段。这样, 当一个客户端的 http 静态页面请求发送到 `nginx` 服务器, `nginx` 就能够调用到我们这里注册的 `ngx_http_static_handler()` 函数, 具体怎么做呢? 接着看。

各个功能模块将其自身的功能函数挂载在 `cmcf->phases` 后, 内部的情况如下图所示:



回调函数会根据选用模块的不同而不同, 上图中显示的是在如附录 A 所示的模块选用下的情况。这些回调函数的调用是有条件的, 调用后也要做一些根据返回值的处理, 比如某次处理能否进入到阶段 `NGX_HTTP_CONTENT_PHASE` 的回调函数中处理, 这需要一个事前判断, 所以在函数 `ngx_http_init_phase_handlers()` 里对所有这些回调函数进行一次重组:

cmcf->phase\_engine.handlers



这里不过多描述 `ngx_http_init_phase_handlers()` 函数如何对这些回调函数进行的重组，因为对照上图并利用 `gdb` 跟踪一下也就清楚了，但从上图中可以看到，该函数只把有回调函数的处理阶段给提取了出来，同时利用 `ngx_http_phase_handler_t` 结构体数组对这些回调函数进行重组，不仅加上了进入回调函数的条件判断 `checker` 函数，而且通过 `next` 字段的使用，把原本的二维数组实现转化为可直接在一维函数数组内部跳动；一般来讲，二维数组的遍历需要两层循环，而遍历一维函数数组就只需一层循环了，所以加上 `next` 字段也并非无的放

矢。

再来看对 http 请求进行分阶段处理核心函数 ngx\_http\_core\_run\_phases:

```
863: Filename : ngx_http_core_module.c
864: void
865: ngx_http_core_run_phases(ngx_http_request_t *r)
866: {
867:     ngx_int_t          rc;
868:     ngx_http_phase_handler_t  *ph;
869:     ngx_http_core_main_conf_t  *cmcf;
870:
871:     cmcf = ngx_http_get_module_main_conf(r, ngx_http_core_module);
872:
873:     ph = cmcf->phase_engine.handlers;
874:
875:     while (ph[r->phase_handler].checker) {
876:
877:         rc = ph[r->phase_handler].checker(r, &ph[r->phase_handler]);
878:
879:         if (rc == NGX_OK) {
880:             return;
881:         }
882:     }
883: }
```

注意 while 循环代码并结合前面的分析, 可以看到这是一个超简单的遍历处理。r->phase\_handler 标志当前处理的序号, 对一个客户端请求处理的最开始时刻, 该值当然就是 0 了, while 循环判断如果存在 checker 函数 (末尾数组元素的 checker 函数为 NULL), 那么就调用该 checker 函数并有可能进而调用对应的回调函数, 以 NGX\_HTTP\_ACCESS\_PHASE 阶段的 ngx\_http\_core\_access\_phase() 函数为例:

```
1087: Filename : ngx_http_core_module.c
1088: ngx_int_t
1089: ngx_http_core_access_phase(ngx_http_request_t *r, ngx_http_phase_handler_t *ph)
1090: {
1091:     ...
1094:     if (r != r->main) {
1095:         r->phase_handler = ph->next;
1096:         return NGX_AGAIN;
1097:     }
1098:     ...
1102:     rc = ph->handler(r);
1103:
1104:     if (rc == NGX_DECLINED) {
1105:         r->phase_handler++;
1106:         return NGX_AGAIN;
    }
```



```

1107:     }
1108:
1109:     if (rc == NGX_AGAIN || rc == NGX_DONE) {
1110:         return NGX_OK;
1111:     }
1112:
1113:     ...
1142:     /* rc == NGX_ERROR || rc == NGX_HTTP_... */
1143:
1144:     ngx_http_finalize_request(r, rc);
1145:     return NGX_OK;
1146: }

```

第 1094 行是一个回调函数准入判断，如果当前是子请求，那么第 1095 行代码让状态机直接进入到了下一个处理阶段；第 1102 行进行回调处理，也就是执行功能模块的功能函数，如果第 1104 行判断成功则表示当前回调拒绝处理或者说不符合它的处理条件，那么第 1105 行将处理移到一下回调函数（注意：处理阶段可能会发生迁移，比如当前回调函数已经是当前阶段的最后一个回调函数，那么调用下一个回调函数时就进入到下一个阶段）；如果第 1109 行判断成功则表示当前回调需要再次调用或已经成功处理，但此处与前两处返回不同，首先并没有进行自增 `phase_handler` 变量，其次是这里返回 `NGX_OK` 会导致 `ngx_http_core_run_phases()` 函数里的循环处理会退出，这表示状态机的继续处理需要等待更进一步的事件发生，这可以是子请求结束、socket 描述符变得可写、超时发生等，并且再进入到状态机处理函数时，仍将从当前回调开始；第 1142 行后表示发生错误（比如 `NGX_ERROR`、`NGX_HTTP_FORBIDDEN`、`NGX_HTTP_UNAUTHORIZED` 等）后的处理流程。

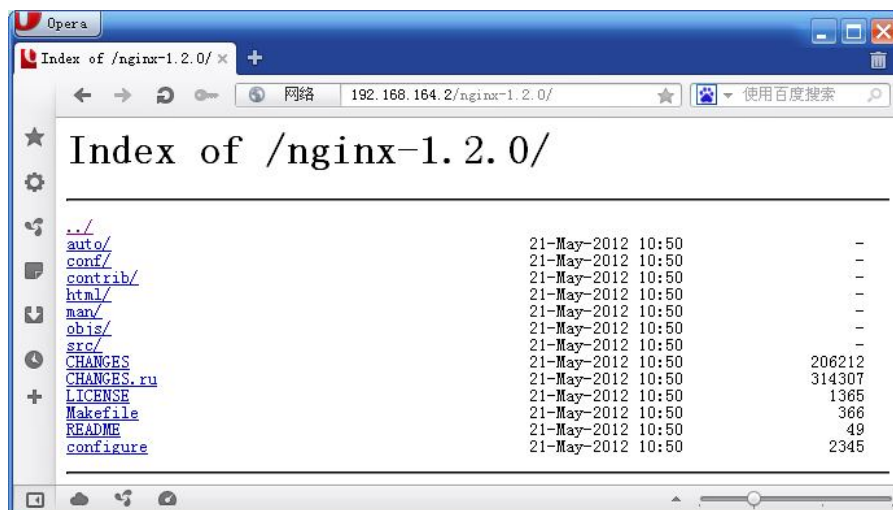
可以看到，一个功能模块的 `handler` 函数可以返回多种类型的值，并且这些值有其固有的含义：

序号	返回值	含义
1	<code>NGX_OK</code>	当前阶段已经被成功处理，必须进入到下一个阶段
2	<code>NGX_DECLINED</code>	当前回调不处理当前情况，进入到下一个回调处理
3	<code>NGX_AGAIN</code>	当前处理所需资源不足，需要等待所依赖事件发生
4	<code>NGX_DONE</code>	当前处理结束，仍需等待进一步事件发生后做处理
5	<code>NGX_ERROR</code> , <code>NGX_HTTP_...</code>	当前回调处理发生错误，需要进入到异常处理流程

值得说明的是，上表只是一般情况下的含义，针对具体的阶段，我们最好仔细对照它的 `checker` 函数，看 `checker` 函数内对回调函数返回值的具体处理是怎样的。

由于回调函数的返回值会影响到同一阶段的后续回调函数的处理与否，而 `nginx` 又采用先进后出的方案，即先注册的模块，其回调函数反而后执行，所以回调函数或者说模块的前后顺序非常重要。以 `NGX_HTTP_CONTENT_PHASE` 阶段的三个回调函数为例，在附录 A

显示的模块列表里可以看到三个相关模块的注册顺序是 `ngx_http_static_module`、`ngx_http_autoindex_module`、`ngx_http_index_module`，而从前面的图中看到回调函数顺序却是 `ngx_http_index_handler`、`ngx_http_autoindex_handler`、`ngx_http_static_handler`，这个顺序是合理的，当我们打开 `nginx` 服务器时，如果直接访问的是一个目录，那么 `nginx` 先是查看当前目录下是否存在 `index.html/index.htm/index.php` 等这样的默认显示页面，这是回调函数 `ngx_http_index_handler()` 的工作；如果不存在默认显示页面，那么就查看是否允许生成类似于下图这样的列表页面：



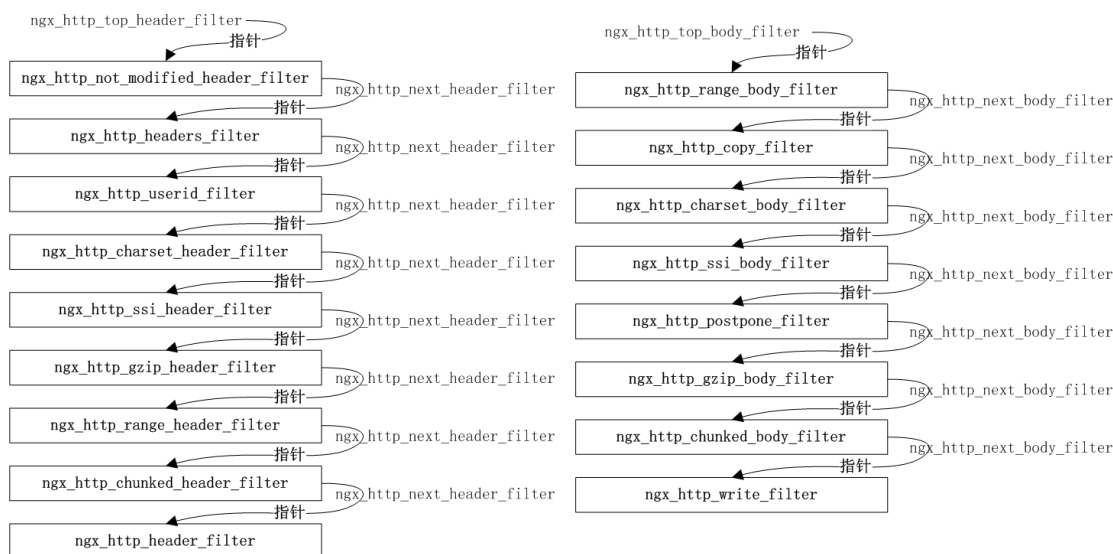
这又是属于 `ngx_http_autoindex_handler()` 函数的工作，而 `ngx_http_static_handler()` 回调函数则是根据客户端静态页面请求查找对应的页面文件并组成待响应内容；可以看到这三个回调函数虽然都挂载在 `NGX_HTTP_CONTENT_PHASE` 阶段，但各自实现的功能本身就存在有先后关系，如果函数 `ngx_http_autoindex_handler()` 在 `ngx_http_index_handler()` 函数之前，那么对于本就存在默认显示页面的目录进行列表显示，这就是非常明显的逻辑错误。

## Filter 模块

对于 `Http` 请求处理 `handlers` 产生的响应内容，在输出到客户端之前需要做过滤处理，这些过滤处理对于完整功能的增强实现与性能的提升是非常有必要的，比如如果没有过滤模块 `ngx_http_chunked_filter_module`，那么就无法支持完整的 `HTTP 1.1` 协议的 `chunk` 功能；如果没有 `ngx_http_not_modified_filter_module` 过滤模块，那么就无法让客户端使用本地缓存来提高性能；诸如这些都需要过滤模块的支持。由于响应数据包括响应头和响应体，所以与此相对应，任一 `filter` 模块必须提供处理响应头的 `header` 过滤功能函数（比如 `ngx_http_not_modified_filter_module` 模块提供的 `ngx_http_not_modified_header_filter()` 函数）或处理响应体的 `body` 过滤功能函数（比如 `ngx_http_copy_filter_module` 模块提供的 `ngx_http_copy_filter()` 函数）或两者皆有（比如 `ngx_http_chunked_filter_module` 模块提供的 `ngx_http_chunked_he`

ader\_filter()函数和 ngx\_http\_chunked\_body\_filter()函数)。

所有的 header 过滤功能函数和 body 过滤功能函数会分别组成各自的两条过滤链，如下图所示（使用附录 A 所列模块）：



这两条过滤链怎么形成的呢？在源文件 ngx\_http.c 里，可以看到定义了这样的两个函数指针变量：

```

71: Filename : ngx_http.c
72: ngx_int_t (*ngx_http_top_header_filter) (ngx_http_request_t *r);
73: ngx_int_t (*ngx_http_top_body_filter) (ngx_http_request_t *r, ngx_chain_t *ch);
    
```

这是整个 nginx 范围内可见的全局变量；然后在每一个 filter 模块内，我们还会看到类似于这样的定义（如果当前模块只有 header 过滤功能函数或只有 body 过滤功能函数，那么如下定义也就只有相应的那个变量）：

```

52: Filename : ngx_http.c
53: static ngx_http_output_header_filter_pt ngx_http_next_header_filter;
54: static ngx_http_output_body_filter_pt ngx_http_next_body_filter;
    
```

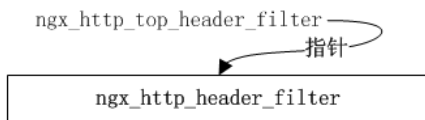
注意到 static 修饰符，也就是说这两个变量是属于模块范围内可见的局部变量。有了这些函数指针变量，再在各个 filter 模块的 postconfiguration 回调函数（该函数会在其对应配置解析完后被调用做一些设置工作，前面已经描述过）内，全局变量与局部变量的巧妙赋值使得最终行成了两条过滤链。以 header 过滤链为例，通过附录 A 的模块列表 ngx\_modules 变量，可以看到 ngx\_http\_header\_filter\_module 是具有 header 过滤功能函数的序号最小的过滤模块，其 postconfiguration 回调函数如下：

```

616: Filename : ngx_http_header_filter_module.c
617: static ngx_int_t
618: ngx_http_header_filter_init(ngx_conf_t *cf)
619: {
620:     ngx_http_top_header_filter = ngx_http_header_filter;
621:
    
```

```
622:     return NGX_OK;
623: }
```

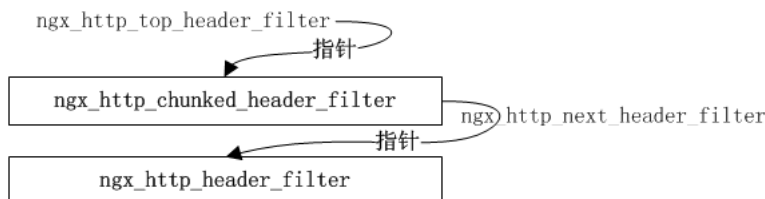
ngx\_http\_top\_header\_filter 指向其 header 过滤功能函数 ngx\_http\_header\_filter，此时 header 过滤链表现为如下形式：



接着 nginx 初始化再继续执行到下一序号的带有 header 过滤功能函数的过滤模块的 postconfiguration 回调函数：

```
231: Filename : ngx_http_chunked_filter_module.c
232: static ngx_int_t
233: ngx_http_chunked_filter_init(ngx_conf_t *cf)
234: {
235:     ngx_http_next_header_filter = ngx_http_top_header_filter;
236:     ngx_http_top_header_filter = ngx_http_chunked_header_filter;
237: ...
```

无需对上面两行代码做过多解释，此时 header 过滤链表现为如下形式：



其它过滤模块的类此加入，逐步形成最终的完整 header 过滤链；当然，body 过滤链的形成过程也与此类似。两条过滤链形成后，其对应的调用入口分别在函数 ngx\_http\_send\_header()和函数 ngx\_http\_output\_filter()内：

```
1888: Filename : ngx_http_core_module.c
1889: ngx_int_t
1890: ngx_http_send_header(ngx_http_request_t *r)
1891: {
1892: ...
1897:     return ngx_http_top_header_filter(r);
1898: }
1899:
1901: ngx_int_t
1902: ngx_http_output_filter(ngx_http_request_t *r, ngx_chain_t *in)
1903: {
1904: ...
1912:     rc = ngx_http_top_body_filter(r, in);
1913: ...
1919:     return rc;
1920: }
```

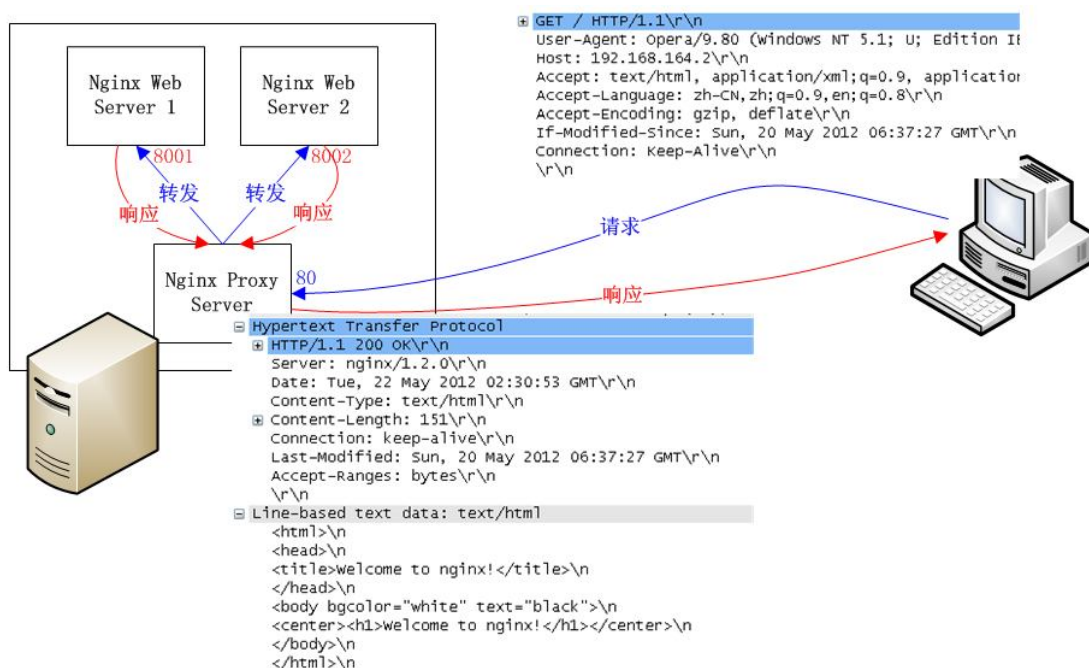
这两个函数非常简单，主要是通过过滤链的链头函数指针全局变量进入到两条过滤链内，进而依次执行链上的各个函数。比如这里 ngx\_http\_top\_header\_filter 指向的是 ngx\_http\_not\_modified\_header\_filter() 函数，因此进入到该函数内执行，而在该函数的执行过程中又会根据情况，继续通过当前模块内的函数指针局部变量 ngx\_http\_next\_header\_filter 间接的调用到 header 过滤链的下一个过滤函数，这对保证过滤链的前后承接是非常必要的，除非我们遇到无法继续处理的错误，此时只有返回 NGX\_ERROR 这样的值：

```
51: Filename : ngx_http_not_modified_filter_module.c
52: static ngx_int_t
53: ngx_http_not_modified_header_filter(ngx_http_request_t *r)
54: {
55: ...
70:     return ngx_http_next_header_filter(r);
71: }
```

根据 HTTP 协议具备的响应头影响或决定响应体内容的特点，所以一般是先对响应头进行过滤，根据头过滤处理返回值再对响应体进行过滤处理，如果在响应头过滤处理中出错或某些特定情况下，响应体过滤处理可以不用再进行。

## Upstream 模块

upstream 模块的典型应用是反向代理，这里就以 ngx\_http\_proxy\_module 模块为例。假定我们有如下这样的实例环境，客户端对服务器 80 端口的请求都被 Nginx Proxy Server 转发到另外两个真实的 Nginx Web Server 实例上进行处理（下图是实验环境，Web Server 和 Proxy Server 都只是 Nginx 进程，并且运行在同一台服务器上）：



那么, Nginx Proxy Server 的核心配置多半是这样:

```
00: Filename : nginx.conf.upstream
01: ...
02: http {
03: ...
04:     upstream load_balance {
05:         server localhost:8001;
06:         server localhost:8002;
07:     }
08:
09:     server {
10:         listen 80;
11:         location / {
12:             proxy_buffering off;
13:             proxy_pass http://load_balance;
14:         }
15:     }
16: }
```

上面的 `proxy_buffering off` 配置是为了禁用 nginx 反向代理的缓存功能, 保证客户端的每次请求都被转发到后端真实服务器, 以便我们每次跟踪分析的 nginx 执行流程更加简单且完整。而另外两个配置指令 `upstream` 和 `proxy_pass` 在此处显得更为重要, 其中 `upstream` 配置指令的回调处理函数为 `ngx_http_upstream()`, 该函数除了申请内存、设置初始值等之外, 最主要的动作就是切换配置上下文并调用 `ngx_conf_parse()` 函数继续进行配置解析:

```
4160:   Filename : ngx_http_upstream.c
4161:       pcf = *cf;
4162:       cf->ctx = ctx;
4163:       cf->cmd_type = NGX_HTTP_UPS_CONF;
4164:
4165:       rv = ngx_conf_parse(cf, NULL);
4166:   ...
4173:       if (uscf->servers == NULL) {
```

进入到 `upstream` 配置块内, 最主要的配置指令也就是 `server`, 其对应的处理函数为 `ngx_http_upstream_server()`, 对于每一个后端真实服务器, 除了其 `uri` 地址外, 还有诸如 `down`、`weight`、`max_fails`、`fail_timeout`、`backup` 这样的可选参数, 所有这些都需要 `ngx_http_upstream_server()` 函数来处理。

在 `ngx_http_upstream.c` 的第 4173 行下个断点, 我们可以看到这里给出示例的解析结果:

```
(gdb) p *(ngx_http_upstream_server_t *)uscf->servers->elts
$20 = {addr = 0x80f73d8, naddr = 2, weight = 1, max_fails = 1, fail_timeout = 10, down = 0, backup = 0}
(gdb) p *(ngx_http_upstream_server_t *)uscf->servers->elts + sizeof(ngx_http_upstream_server_t)
$21 = {addr = 0x80f7460, naddr = 2, weight = 1, max_fails = 1, fail_timeout = 10, down = 0, backup = 0}
(gdb) p (*(ngx_http_upstream_server_t *)uscf->servers->elts)->addr
$22 = {sockaddr = 0x80f73f8, socklen = 16, name = {len = 14, data = 0x80f7408 "127.0.0.1:8001"}}
(gdb) p (*(ngx_http_upstream_server_t *)uscf->servers->elts + sizeof(ngx_http_upstream_server_t))->addr
$23 = {sockaddr = 0x80f7480, socklen = 16, name = {len = 14, data = 0x80f7490 "127.0.0.1:8002"}}
```



另外一个重要配置指令 `proxy_pass` 主要出现在 `location` 配置上下文中，而其对应的处理函数为 `ngx_http_proxy_pass()`，抹去该函数内的众多细节，我们重点关注两个赋值语句：

```
3336:  Filename : ngx_http_proxy_module.c
3337:  static char *
3338:  ngx_http_proxy_pass(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
3339:  {
3340:  ...
3356:      clcf->handler = ngx_http_proxy_handler;
3357:  ...
3425:      plcf->upstream.upstream = ngx_http_upstream_add(cf, &u, 0);
```

上面片段代码里的第一个赋值语句给当前 `location` 的 `http` 处理设置回调函数，而第二个赋值语句则是查找（没有找到则会创建，比如如果配置文件中 `upstream` 指令出现在 `proxy_pass` 指令的后面）其对应的 `upstream` 配置，我们这里就一个名为 `load_balance` 的 `upstream`，所以找到的配置就是它了：

```
(gdb) p *uscfp[1]
$3 = {peer = {init_upstream = 0, init = 0, data = 0x0}, srv_conf = 0x80f6a30, servers = 0x80f7384, flags = 63, host = {
  len = 12, data = 0x80f69e9 "load_balance"}, file_name = 0x80eb7ef "/usr/local/nginx/conf/nginx.conf.upstream",
  line = 12, port = 0, default_port = 0}
```

前面曾提到，Nginx 将对客户端的 `http` 请求处理分为多个阶段，而其中有个 `NGX_HTTP_FIND_CONFIG_PHASE` 阶段主要就是做配置查找处理，如果当前请求 `location` 设置了 `upstream`，即回调函数指针 `clcf->handler` 不为空，则表示对该 `location` 的请求需要后端真实服务器来处理：

```
949: Filename : ngx_http_core_module.c
950: ngx_int_t
951: ngx_http_core_find_config_phase(ngx_http_request_t *r,
952:     ngx_http_phase_handler_t *ph)
953: {
954: ...
981:     ngx_http_update_location_config(r);
982: ...
1439: void
1440: ngx_http_update_location_config(ngx_http_request_t *r)
1441: {
1442: ...
1519:     if (clcf->handler) {
1520:         r->content_handler = clcf->handler;
1521:     }
1522: }
```

在其它有 `location` 更新的情况下，比如 `redirect` 重定向 `location` 或 `named` 命名 `location` 或 `if` 条件 `location` 等，此时也会调用 `ngx_http_update_location_config()` 函数进行 `location` 配置更新。我们知道 `upstream` 模块的主要功能是产生响应数据，虽然这些响应数据来自后端真实



服务器，所以在 NGX\_HTTP\_CONTENT\_PHASE 阶段的 checker 函数 ngx\_http\_core\_content\_phase() 内，我们可以看到在 r->content\_handler 不为空的情况下会优先对 r->content\_handler 函数指针进行回调：

```

1385:   Filename : ngx_http_core_module.c
1386:   ngx_int_t
1387:   ngx_http_core_content_phase(ngx_http_request_t *r,
1388:       ngx_http_phase_handler_t *ph)
1389:   {
1390:   ...
1394:       if (r->content_handler) {
1395:           r->write_event_handler = ngx_http_request_empty_handler;
1396:           ngx_http_finalize_request(r, r->content_handler(r));
1397:           return NGX_OK;
1398:       }
1399:   ...

```

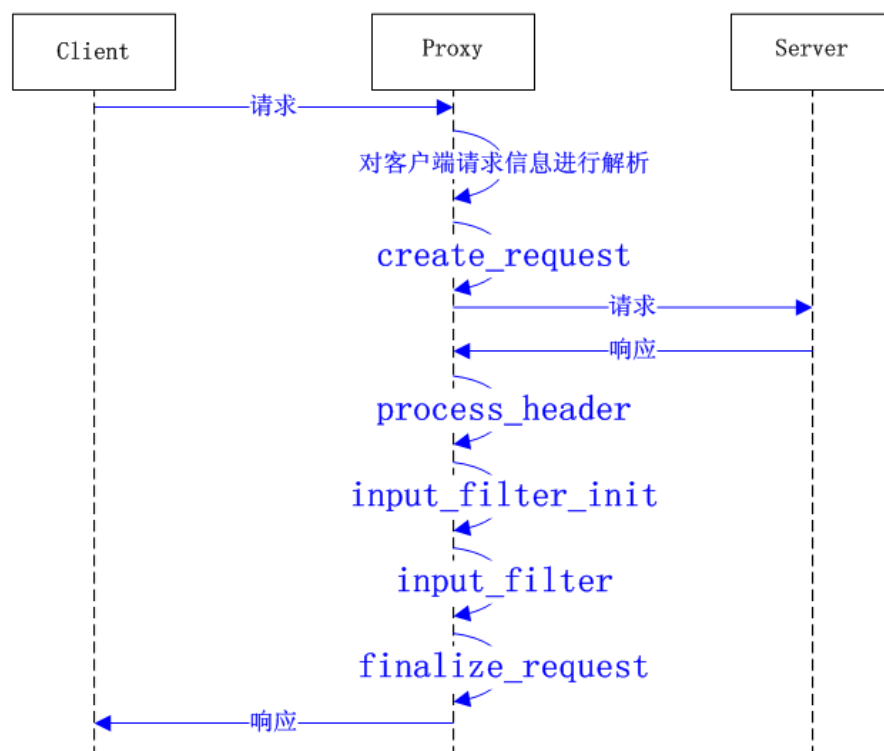
第 1394 行，如果 r->content\_handler 不为空，即存在 upstream，那么进入处理，注意第 1397 行直接返回 NGX\_OK，也即不再调用挂在该阶段的其它模块回调函数，所以说 upstream 模块的优先级是最高的。根据前面的回调赋值，调用 r->content\_handler() 指针函数，实质上就是执行函数 ngx\_http\_proxy\_handler()，直到这里，我们才真正走进 upstream 代理模块的处理逻辑里。

对于任何一个 Upstream 模块而言，最核心的实现主要是 7 个回调函数，upstream 代理模块自然也不例外，它实现并注册了这 7 个回调函数：

回调指针	函数功能	upstream 代理模块
create_request	根据 nginx 与后端服务器通信协议（比如 HTTP、Memcache），将客户端的 HTTP 请求信息转换为对应的发送到后端服务器的真实请求。	ngx_http_proxy_create_request 由于 nginx 与后端服务器通信协议也为 HTTP，所以直接拷贝客户端的请求头、请求体（如果有）到变量 r->upstream->request_bufs 内。
process_header	根据 nginx 与后端服务器通信协议，将后端服务器返回的头部信息转换为对客户端响应的 HTTP 响应头。	ngx_http_proxy_process_status_line 此时后端服务器返回的头部信息已经保存在变量 r->upstream->buffer 内，将这串字符串解析为 HTTP 响应头存储到变量 r->upstream->headers_in 内。
input_filter_init	根据前面获得的后端服务器返回的头部信息，为进一步处理后端服务器将返回的响应体做初始准备工作。	ngx_http_proxy_input_filter_init 根据已解析的后端服务器返回的头部信息，设置需进一步处理的后端服务器将返回的响应体的长度，该值保存在变量 r->

		upstream->length 内。
input_filter	正式处理后端服务器返回的响应体。	ngx_http_proxy_non_buffered_copy_filter 本次收到的响应体数据长度为 bytes，数据长度存储在 r->upstream->buffer 内，把它加入到 r->upstream->out_bufs 响应数据链等待发送给客户端。
finalize_request	正常结束与后端服务器的交互，比如剩余待取数据长度为 0 或读到 EOF 等，之后就会调用该函数。由于 nginx 会自动完成与后端服务器交互的清理工作，所以该函数一般仅做下日志，标识响应正常结束。	ngx_http_proxy_finalize_request 记录一条日志，标识正常结束与后端服务器的交互，然后函数返回。
reinit_request	对交互重新初始化，比如当 nginx 发现一台后端服务器出错无法正常完成处理，需要尝试请求另一台后端服务器时就会调用该函数。	ngx_http_proxy_reinit_request 设置初始值，设置回调指针，处理比较简单。
abort_request	异常结束与后端服务器的交互后就会调用该函数。大部分情况下，该函数仅做下日志，标识响应异常结束。	ngx_http_proxy_abort_request 记录一条日志，标识异常结束与后端服务器的交互，然后函数返回。

上表格中前面 5 个函数执行的先后次序如下图所示，由于在 Client/Proxy/Server 之间，一次请求/响应数据可以发送多次（下图中只画出一发就发送完毕的情况），所以下图中对应的函数也可能被执行多次，不过一般情况下，这 5 个函数执行的先后次序就是这样了。



这些回调函数如何夹杂到 nginx 中被调用并不需要完全搞清楚，要写一个 Upstream 模块，我们只要实现上面提到的这 7 个函数即可，当然，可以看到最主要的也就是 `create_request`、`process_header` 和 `input_filter` 这三个回调，它们实现从 HTTP 协议到 Nginx 与后端服务器之间交互协议的来回转换，使得在用户看来，他访问的就是一台功能完整的 Web 服务器，而也许事实上，显示在他面前的数据来自 Memcache 或别的什么服务器。

## Load-balance 模块

Load-balance 模块可以称之为辅助模块，与前面介绍的以处理请求/响应数据为目标的三种模块完全不同，它的目标明确且单一，即如何从多台后端服务器中选择出一台合适的服务器来处理当前请求。

要实现一个具体的 Load-balance 模块，依旧只需实现如下 4 个回调函数即可：

回调指针	函数功能	round_robin 模块	ip_hash 模块
uscf->peer.init_upstream	解析配置文件过程中被调用，根据 upstream 里各个 server 配置项做初始准备工作，另外的核心工作是设置回调指针 us->peer.init。配置文件	ngx_http_upstream_init_round_robin 设置: us->peer.init = ngx_http_upstream_init_round_robin_peer;	ngx_http_upstream_init_ip_hash 设置: us->peer.init = ngx_http_upstream_init_ip_hash_peer;

	解析完后就不再被调用。		
us->peer.init	在每一次 nginx 准备转发客户端请求到后端服务器前都会调用该函数，该函数为本次转发选择合适的后端服务器做初始准备工作，另外的核心工作是设置回调指针 r->upstream->peer.get 和 r->upstream->peer.free 等。	ngx_http_upstream_init_round_robin_peer 设置: r->upstream->peer.get = ngx_http_upstream_get_round_robin_peer; r->upstream->peer.free = ngx_http_upstream_free_round_robin_peer;	ngx_http_upstream_init_ip_hash_peer 设置: r->upstream->peer.get = ngx_http_upstream_get_ip_hash_peer; r->upstream->peer.free 为空。
r->upstream->peer.get	在每一次 nginx 准备转发客户端请求到后端服务器前都会调用该函数，该函数实现具体的为本次转发选择合适后端服务器的算法逻辑，即完成选择获取合适后端服务器的功能。	ngx_http_upstream_get_round_robin_peer 加权选择当前权值最高（即从各方面综合比较更有能力处理当前请求）的后端服务器。	ngx_http_upstream_get_ip_hash_peer 根据 ip 哈希值选择后端服务器。
r->upstream->peer.free	在每一次 nginx 完成与后端服务器之间的交互后都会调用该函数。如果选择算法有前后依赖性，比如加权选择，那么需要做一些数值更新操作；如果选择算法没有前后依赖性，比如 ip 哈希，那么该函数可为空；	ngx_http_upstream_free_round_robin_peer 更新相关数值，比如 rrp->current 等。	空

Nginx 默认采用 round\_robin 加权算法，如果要选择其它负载均衡算法，必须在 upstream 的配置上下文中明确指定。比如采用 ip\_hash 算法的 upstream 配置如下所示：

```
00: Filename : nginx.conf
01: ...
20:         upstream load_balance {
```

```
21:             ip_hash;
22:             server localhost:8001;
23: ...
```

在配置项 `ip_hash` 的处理函数里，会给 `uscf->peer.init_upstream` 函数指针赋值上 `ip_hash` 模块提供的回调函数，这样在 Nginx 后续处理过程中才能调到 `ip_hash` 模块的功能逻辑里。

## 第五章 事件机制

### I/O 多路复用模型

各种平台下支持的各种 I/O 事件处理机制在 `nginx` 内部都被进行了统一封装，这样不论 `nginx` 被用在何种平台都以最高效的方式运行，下表列出了 `nginx` 具体的支持情况：

名称	特点
<code>select</code>	标准的 IO 复用模型，几乎所有的类 <code>unix</code> 系统上都有提供，但性能相对较差。如果在当前系统平台找不到更优的 IO 复用模型，那么 <code>nginx</code> 默认编译并使用 <code>select</code> 复用模型，我们也可以通过使用 <code>--with-select_module</code> 或 <code>--without-select_module</code> 配置选项来启用或禁用 <code>select</code> 复用模型模块的编译。
<code>poll</code>	标准的 IO 复用模型，但理论上比 <code>select</code> 复用模型要优。同 <code>select</code> 复用模型类似，可以通过使用 <code>--with-poll_module</code> 或 <code>--without-poll_module</code> 配置选项来启用或禁用 <code>poll</code> 复用模型模块的编译。
<code>epoll</code>	系统 <code>Linux 2.6+</code> 上正式提供的性能更为优秀的 IO 复用模型。
<code>kqueue</code>	在系统 <code>FreeBSD 4.1+</code> ， <code>OpenBSD 2.9+</code> ， <code>NetBSD 2.0</code> 和 <code>MacOS X</code> 上特有的性能更优秀的 IO 复用模型。
<code>eventport</code>	在系统 <code>Solaris 10</code> 上可用的高性能 IO 复用模型。
<code>/dev/poll</code>	在系统 <code>Solaris 7 11/99+</code> ， <code>HP/UX 11.22+</code> ( <code>eventport</code> )， <code>IRIX 6.5.15+</code> 和 <code>Tru64 UNIX 5.1A+</code> 上可用的高性能 IO 复用模型。
<code>rtsig</code>	实时信号 ( <code>real time signals</code> ) 模型，在 <code>Linux 2.2.19+</code> 系统上可用。可以通过使用 <code>--with-rtsig_module</code> 配置选项来启用 <code>rtsig</code> 模块的编译。
<code>aio</code>	异步 I/O ( <code>Asynchronous Input and Output</code> ) 模型，通过异步 IO 函数，如 <code>aio_read</code> 、 <code>aio_write</code> 、 <code>aio_cancel</code> 、 <code>aio_error</code> 、 <code>aio_fsync</code> 、 <code>aio_return</code> 等实现。

上表给出的 8 种 I/O 事件处理机制中，前 6 种属于本节将介绍的 I/O 多路复用模型，而后两种机制，实时信号和异步 I/O 比较特殊，在此不做过多的描述，本文其它地方也不做考虑；

不论哪种 I/O 多路复用模型，基本的原理是相同的，它们都能让应用程序可以同时对多个 I/O 端口进行监控以判断其上的操作是否已经顺利完成，达到时间复用的目的。举个例子，如果要监控来之 10 根不同地方的水管 (I/O 端口) 是否有水流出来 (是否可读)，那么需要 10 个人 (10 个线程或 10 处代码) 来做这件事情；如果利用某种技术 (比如摄像头) 把这 10 根水管的状态情况统一传达到某个点，那么就只需要 1 人在那个点进行监控就行了，而类似于 `select()` 或 `epoll_wait()` 这样的系统调用就类似于摄像头的功能，应用程序将阻塞在这些系统调用上，而不是阻塞在某一处的 I/O 系统调用上。这个例子虽然粗糙，但应该是把 I/O

多路复用模型的基本特点给描述出来了。

不同的平台有支持不同的 I/O 多路复用模型，我认为对它们一个个进行讲解是不必要的，因为通过查 man 手册或 Google 都能找到更详细的资料，所以我们直接看 nginx 对这些 I/O 多路复用模型的封装与使用。在 nginx 源码里，I/O 多路复用模型被封装在一个名为 `ngx_event_actions_t` 的结构体里，该结构体包含的字段主要就是回调函数，将各个 I/O 多路复用模型的功能接口进行统一：

ngx_event_actions_t 接口	说明
init	初始化
add	将某描述符的某个事件（可读/可写）添加到多路复用监控里
del	将某描述符的某个事件（可读/可写）从多路复用监控里删除
enable	启用对某个指定事件的监控
disable	禁用对某个指定事件的监控
add_conn	将指定连接关联的描述符加入到多路复用监控里
del_conn	将指定连接关联的描述符从多路复用监控里删除
process_changes	监控的事件发生变化，只有 kqueue 会用到这个接口
process_events	阻塞等待事件发生，对发生的事件进行逐个处理
done	回收资源

由于 I/O 多路复用模型各自具体实现的不同，上表中列出的一些回调接口，在 Nginx 的各个 I/O 多路复用处理模块里可能并没有对应的处理，但几个最基本的接口，比如 `add/del/process_events` 肯定都会有实现。为了方便使用任何一种事件处理机制，nginx 定义了一个类型为 `ngx_event_actions_t` 的全局变量 `ngx_event_actions`，并且还定义了几个宏：

```

44: Filename : ngx_event.c
45: ngx_event_actions_t    ngx_event_actions;

447: Filename : ngx_event.h
448: #define ngx_process_changes    ngx_event_actions.process_changes
449: #define ngx_process_events      ngx_event_actions.process_events
450: #define ngx_done_events          ngx_event_actions.done
451:
452: #define ngx_add_event            ngx_event_actions.add
453: #define ngx_del_event            ngx_event_actions.del
454: #define ngx_add_conn             ngx_event_actions.add_conn
455: #define ngx_del_conn             ngx_event_actions.del_conn
    
```

这样，nginx 要将某个事件添加到多路复用监控里，只需调用 `ngx_add_event()` 函数即可，至于这个函数对应到哪个具体的 I/O 多路复用处理模块上，在这里可以毫不关心。

当然，我们做分析还是要知道 `ngx_add_event()` 函数是怎么关联到具体的 I/O 多路复用处理模块的，而不难看出，关键点是全局变量 `ngx_event_actions` 的值。给全局变量 `ngx_event`



\_actions 进行赋值出现在各个事件处理模块的初始化函数内，比如 epoll 模块：

```

147: Filename : ngx_epoll_module.c
148: ngx_event_module_t  ngx_epoll_module_ctx = {
149:     &epoll_name,
150:     ngx_epoll_create_conf,          /* create configuration */
151:     ngx_epoll_init_conf,            /* init configuration */
152:
153:     {
154:         ngx_epoll_add_event,        /* add an event */
155:         ngx_epoll_del_event,        /* delete an event */
156:         ngx_epoll_add_event,        /* enable an event */
157:         ngx_epoll_del_event,        /* disable an event */
158:         ngx_epoll_add_connection,   /* add an connection */
159:         ngx_epoll_del_connection,   /* delete an connection */
160:         NULL,                       /* process the changes */
161:         ngx_epoll_process_events,    /* process the events */
162:         ngx_epoll_init,              /* init the events */
163:         ngx_epoll_done,              /* done the events */
164:     }
165: };
166: ...
288: static ngx_int_t
289: ngx_epoll_init(ngx_cycle_t *cycle, ngx_msec_t timer)
290: {
291: ...
327:     ngx_event_actions = ngx_epoll_module_ctx.actions;

```

在其它事件处理模块的初始化函数内也可以找到这样的赋值语句，所以一旦设定 nginx 使用某个事件处理模块，经过事件处理模块的初始化后，就把全局变量 ngx\_event\_actions 指向了它的封装，比如从上面 epoll 模块的源代码来看，调用 ngx\_add\_event()函数对应执行的就是 ngx\_epoll\_add\_event()函数。

设定 nginx 使用哪个事件处理机制是通过在 event 块里使用 use 指令来指定的，该配置指令对应的处理函数为 ngx\_event\_use()，在经过相关验证（比如重复指定、对应的事件处理模块是否存在等）后，就会把对应的事件处理模块序号记录在配置变量 ecf->use 内。如果不进行主动指定，那么 nginx 就会根据当前系统平台选择一个合适的事件处理模块，并且同样把其序号记录在配置变量 ecf->use 内，其相关逻辑实现在函数 ngx\_event\_core\_init\_conf()内。

在工作进程的初始化函数 ngx\_worker\_process\_init()内会调用事件核心模块的初始化函数 ngx\_event\_process\_init()，而在该函数内，根据配置变量 ecf->use 记录的值，进而调用到对应事件处理模块的初始化函数，比如 epoll 模块的 ngx\_epoll\_init()函数：

```

582: Filename : ngx_event.c
583: static ngx_int_t
584: ngx_event_process_init(ngx_cycle_t *cycle)

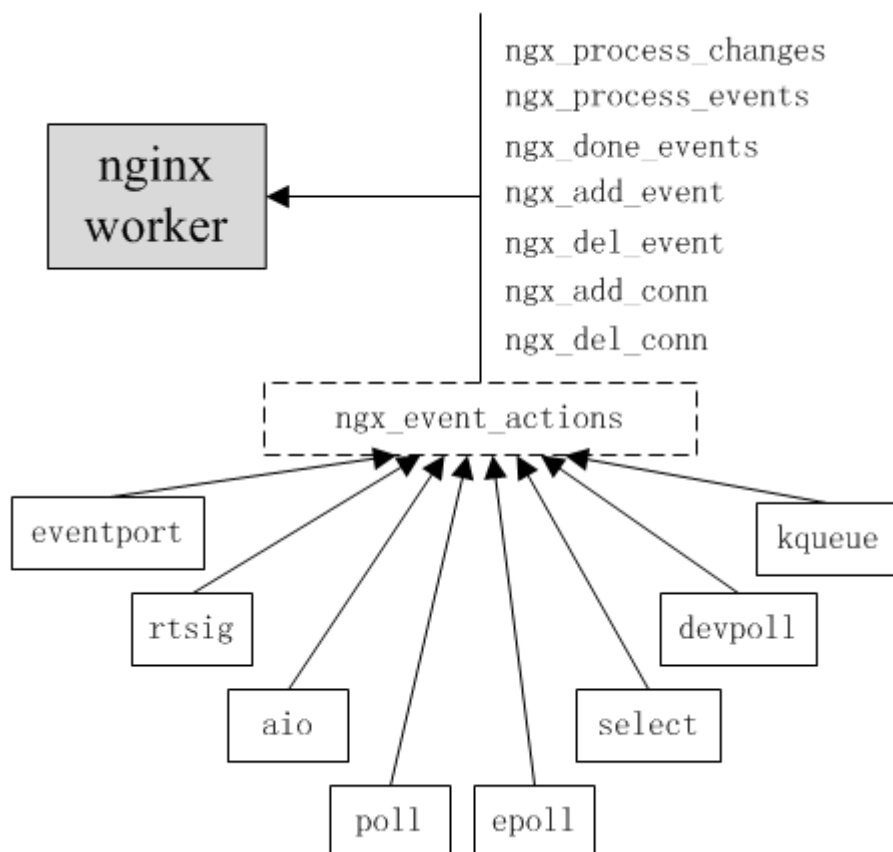
```

```

585: {
586: ...
617:     for (m = 0; ngx_modules[m]; m++) {
618:         if (ngx_modules[m]->type != NGX_EVENT_MODULE) {
619:             continue;
620:         }
621:
622:         if (ngx_modules[m]->ctx_index != ecf->use) {
623:             continue;
624:         }
625:
626:         module = ngx_modules[m]->ctx;
627:
628:         if (module->actions.init(cycle, ngx_timer_resolution) != NGX_OK) {
629:             /* fatal */
630:             exit(2);
631:         }
632:
633:         break;
634:     }

```

至此，nginx 内对 I/O 多路复用模型的整体封装，前后才真正衔接起来，下面是一个粗略的框图：

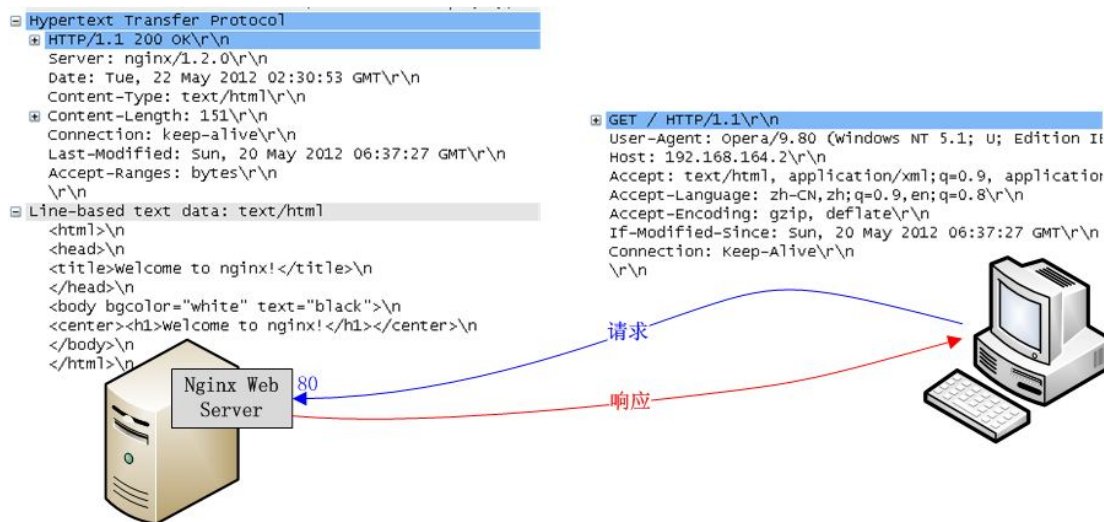


## 事件处理

Nginx 内事件封装所对应的结构体为 `ngx_event_t`，在该结构体内可以看到很多位域字段，凭经验即可知道它们都是用作旗标，即标记事件当前是否处在某种状态；除去这些旗标字段，与事件本身联系更为紧密的是回调接口 `handler` 字段，该字段直接指定了当事件发生时，nginx 该如何进行处理。

我们所关注的事件基本都是依附在 `socket` 描述符上的，而随着处理流程的不断变化，在 `socket` 描述符上所关注的事件也会发生改变；比如，对于一个新建连接 `socket`，一开始必定是关注其可读事件，以便从客户端获取请求信息，当读取完所有请求信息并且被 nginx 正常处理后，又将关注该 `socket` 的可写事件，从而可以将对应的响应信息顺利发送给客户端；即便是关注的同一个事件，根据当前处理阶段的不同，其事件处理回调函数也可能不同，这很容易理解，比如同是新建连接 `socket` 的可读事件，但处理客户端请求头的回调函数与处理客户端请求体的回调函数肯定不是同一个；下面就分析一个客户端请求/服务端响应的完整流程，看在这个过程中，关注事件如何变化，回调函数又如何变化。

这是一个非常简单的流程，客户端浏览器（比如 ie）发送请求（请求某静态页面）到服务器，服务器也就是 nginx 程序，nginx 从磁盘文件系统读取静态文件发送给客户端，流程结束。



当客户端浏览器发送请求到 nginx 时，nginx 就将调用监听套接口对应的事件处理函数 `ngx_event_accept()`，在该函数内将创建一个新的关联当前请求连接的套接口；在这个套接口上，nginx 关注的事件以及回调函数列表（通过 `gdb` 的 `watch` 指令抓取）如下：

序号	关注事件类型	对应的回调函数
1	读	<code>ngx_http_init_request()</code>
2	写	<code>ngx_http_empty_handler()</code>

3	读	ngx_http_process_request_line()
4	读	ngx_http_process_request_headers()
5	读	ngx_http_request_handler()
6	写	ngx_http_request_handler()
7	写	ngx_http_empty_handler()
8	读	ngx_http_keepalive_handler()

accept()新建的套接口最先关注的当然是读事件，以便从客户端获取请求信息，其回调函数为 ngx\_http\_init\_request()，一旦读到客户端请求信息就开始进行初始化等准备工作；此时不关注写事件，所以写事件的回调函数为 ngx\_http\_empty\_handler()，什么也不做，仅打印一条日志；接下来对请求头、请求头处理依次进行，一旦处理结束就开始关注写事件，此时的写事件回调函数同为 ngx\_http\_request\_handler()，将响应数据全部发回给客户端后，将写事件的回调函数又置为 ngx\_http\_empty\_handler()；最后，关注读事件等待客户端的下一个请求，此时的回调处理函数为 ngx\_http\_keepalive\_handler()，表示当前是在与客户端保持 keep alive 状态；如果客户端有新的请求数据发到，那么在 ngx\_http\_keepalive\_handler()函数内将读到对应的数据，并且调用 ngx\_http\_init\_request()做初始化，开始一个新的请求处理。如果此时客户端关闭了连接，那么 nginx 同样也将获得一个可读事件，调用 ngx\_http\_keepalive\_handler()函数处理却读取不到数据，于是关闭连接、回收资源，函数返回；这部分相关逻辑如下所示：

```

2662:  Filename : ngx_event.c
2663:  static void
2664:  ngx_http_keepalive_handler(ngx_event_t *rev)
2665:  {
2666:  ...
2730:      n = c->recv(c, b->last, size);
2731:  ...
2743:      if (n == 0) {
2744:          ngx_log_error(NGX_LOG_INFO, c->log, ngx_socket_errno,
2745:              "client %V closed keepalive connection", &c->addr_text);
2746:          ngx_http_close_connection(c);
2747:          return;
2748:      }
2749:  ...
2767:      ngx_http_init_request(rev);
2768:  }
```

Nginx 对事件的处理耦合性太强，对上一步骤、当前处理步骤以及下一步骤都必须仔细把握，否则回调设置错了，一切就乱了；当然，这也可以说它灵活，只要你乐意，插入几个自编的模块到处理步骤里是非常简单的事情。

## 负载均衡

在一般情况下，配置 nginx 执行时，工作进程都会有多个，由于各个工作进程相互独立的接收客户端请求、处理、响应，所以就可能会出现负载不均衡的情况，比如 1 个工作进程当前有 3000 个请求等待处理，而另 1 个进程当前却只有 300 个请求等待处理，nginx 采取了哪些均衡措施来避免这种情况就是本节将要讨论的内容。

从上一节内容可以看到，nginx 工作进程的主要任务就是处理事件，而事件的最初源头来之监听套接口，所以一旦某个工作进程独自拥有了某个监听套接口，那么所有来之该监听套接口的客户端请求都将被这个工作进程处理；当然，如果是多个工作进程同时拥有某个监听套接口，那么一旦该监听套接口出现有某客户端请求，此时就将引发所有拥有该监听套接口的工作进程去争抢这个请求，能争抢到的肯定只有某一个工作进程，而其它工作进程注定要无获而返，这种现象即为惊群 (thundering herd)。关于惊群是否已经被 Linux 内核所处理，这里不做深入考究，但可以肯定的是，要进行负载均衡，最基本的着手点也就是监听套接口，nginx 是不是这样做的呢？下面来看。

在 nginx 源码里能看到这样一个名为 ngx\_use\_accept\_mutex 的变量，可以说它就是 nginx 均衡措施的基本，该变量是整型类型，具体定义如下：

```
53: Filename : ngx_event.c
54: ngx_uint_t      ngx_use_accept_mutex;
```

该变量的赋值语句在函数 ngx\_event\_process\_init() 内，也就是每个工作进程开始时的初始化函数，前后调用关系如下：

```
ngx_worker_process_cycle() -> ngx_worker_process_init() -> ngx_event_process_init()
```

在函数 ngx\_event\_process\_init() 内，可以看到只有多进程模型下，并且工作进程数目大于 1、用户设置开启负载均衡的情况下才设置该变量为 1，否则为 0：

```
596: Filename : ngx_event.c
597:     if (ccf->master && ccf->worker_processes > 1 && ccf->accept_mutex) {
598:         ngx_use_accept_mutex = 1;
599:         ngx_accept_mutex_held = 0;
600:         ngx_accept_mutex_delay = ccf->accept_mutex_delay;
601:
602:     } else {
603:         ngx_use_accept_mutex = 0;
604:     }
```

前两个条件很容易理解，只有有多个进程才有均衡的概念，而对于 ccf->accept\_mutex 字段的判断主要是提供用户便利，可以关闭该功能，因为既然均衡策略也有相应的代码逻辑，难保在某些情况下其本身的消耗也许会得不偿失；当然，该字段默认为 1，在配置初始化函数 ngx\_event\_core\_init\_conf() 内，有这么一句：ngx\_conf\_init\_value(ccf->accept\_mutex, 1);

一旦变量 ngx\_use\_accept\_mutex 值为 1，也就开启了 nginx 负载均衡策略，此时在每个

工作进程的初始化函数 `ngx_event_process_init()`内，所有监听套接口都不会被加入到其事件监控机制里，如下第 828 和 829 行的代码跳过了所有监听套接口的监听事件加入：

```

745: Filename : ngx_event.c
746:     for (i = 0; i < cycle->listening.nelts; i++) {
747: ...
828:         if (ngx_use_accept_mutex) {
829:             continue;
830:         }
831: ...
838:         if (ngx_add_event(rev, NGX_READ_EVENT, 0) == NGX_ERROR) {
839:             return NGX_ERROR;
840:         }
841: ...
845:     }
```

而真正将监听套接口（即客户端请求）加入到事件监控机制是在函数 `ngx_process_events_and_timers()`里。在前面的进程模型一节，曾提到工作进程的主要执行体是一个无限 `for` 循序，而在该循环内最重要的函数调用就是 `ngx_process_events_and_timers()`，所以可以想象在该函数内动态添加或删除监听套接口是一种很灵活的方式；如果当前工作进程负载比较小，就将监听套接口加入到自身的事件监控机制里，从而带来新的客户端请求；而如果当前工作进程负载比较大，就将监听套接口从自身的事件监控机制里删除，避免引入新的客户端请求而带来更大的负载；当然，并不是想加就加、想删就删，这需要利用锁机制来做互斥与同步，既避免监听套接口被同时加入到多个进程的事件监控机制里，又避免监听套接口在某时刻没有被任何一个进程监控。

看函数 `ngx_process_events_and_timers()`源码，这里有一段至关重要的代码：

```

222: Filename : ngx_event.c
223:     if (ngx_use_accept_mutex) {
224:         if (ngx_accept_disabled > 0) {
225:             ngx_accept_disabled--;
226:
227:         } else {
228:             if (ngx_trylock_accept_mutex(cycle) == NGX_ERROR) {
229:                 return;
230:             }
231:
232:             if (ngx_accept_mutex_held) {
233:                 flags |= NGX_POST_EVENTS;
234:
235:             } else {
236:                 if (timer == NGX_TIMER_INFINITE
237:                     || timer > ngx_accept_mutex_delay)
238:                     {
```



```

239:                timer = ngx_accept_mutex_delay;
240:            }
241:        }
242:    }
243: }

```

可以看到这段代码只有在开启负载均衡（即 `ngx_use_accept_mutex = 1;`）后才生效，在该逻辑内，首先通过检测变量 `ngx_accept_disabled` 值是否大于 0 来判断当前进程是否已经过载，为什么可以这样判断需要理解变量 `ngx_accept_disabled` 值的含义，这在 `accept()` 接受新请求连接的处理函数 `ngx_event_accept()` 内可以看到：

```

17: Filename : ngx_event_accept.c
18: void
19: ngx_event_accept(ngx_event_t *ev)
20: {
21: ...
107:     ngx_accept_disabled = ngx_cycle->connection_n / 8
108:                          - ngx_cycle->free_connection_n;

```

其中 `ngx_cycle->connection_n` 表示一个工作进程的最大可承受连接数，可以通过 `worker_connections` 指令配置，其默认值为 512，在工作进程配置初始化函数 `ngx_event_core_init_conf()` 内有这样的语句：

```

12: Filename : ngx_event.c
13: #define DEFAULT_CONNECTIONS 512
1244:     ngx_conf_init_uint_value(ecf->connections, DEFAULT_CONNECTIONS);
1245:     cycle->connection_n = ecf->connections;

```

另外一个变量 `ngx_cycle->free_connection_n` 则表示当前可用连接数，假设当前活动连接数为  $x$ ，那么该值为：`ngx_cycle->connection_n - x`；故此 `ngx_accept_disabled` 的值为：

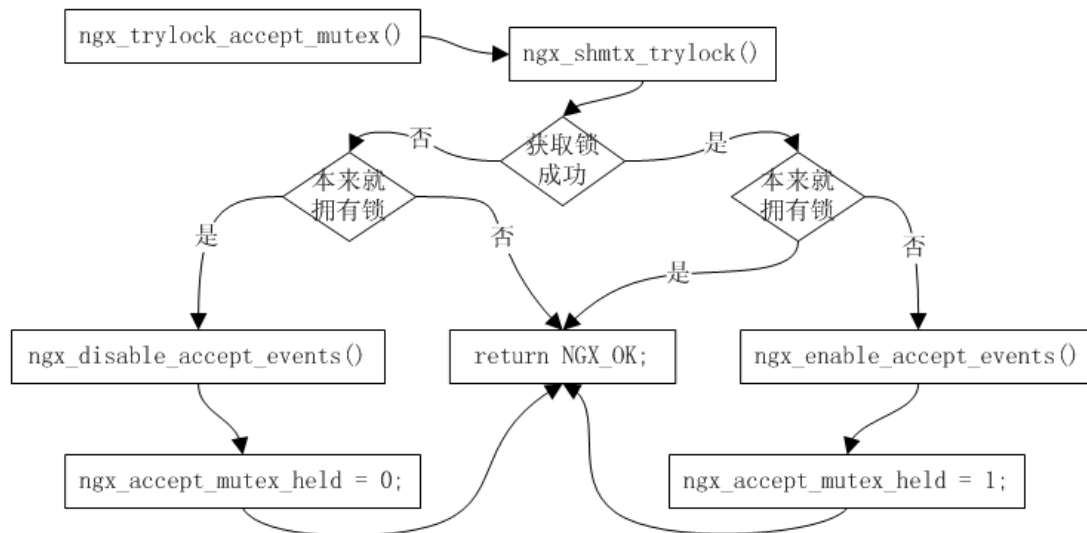
$$\text{ngx\_accept\_disabled} = x - \text{ngx\_cycle->connection\_n} * 7 / 8;$$

也就是说如果当前活动连接数 ( $x$ ) 超过最大可承受连接数的  $7/8$ ，则表示发生过载，变量 `ngx_accept_disabled` 值将大于 0，并且该值越大表示负载越重。

回过头来看函数 `ngx_process_events_and_timers()` 内的代码，当进程处于过载状态时，所做的工作仅仅只是对变量 `ngx_accept_disabled` 自减 1（第 225 行），这表示既然经过了一轮事件处理，那么负载肯定有所减小，所以也要相应的调整变量 `ngx_accept_disabled` 的值；经过一段时间，`ngx_accept_disabled` 又会降到 0 以下，便可争用锁获取新的请求连接。

如果进程并没有处于过载状态，那么就会去争用锁（第 228 行），当然，实际上是争用监听套接口的拥有权，争锁成功就会把所有监听套接口（注意：是所有的监听套接口，它们总是作为一个整体被加入或删除，下同）加入到自身的事件监控机制里（如果原本不在），争锁失败就会把监听套接口从自身的事件监控机制里删除（如果原本就在）；从函数 `ngx_trylock_accept_mutex()` 的内部实现可以看到这一点，代码非常容易理解，画个流程图表示（剔除了异常流程）：





变量 `ngx_accept_mutex_held` 的值用于标识当前是否拥有锁，注意这一点很重要，因为接着看第 232-241 行的代码就是针对如此的处理；如果当前拥有锁，则给 `flags` 变量打个 `NGX_POST_EVENTS` 标记，这表示所有发生的事件都将延后处理（POST 有表示在...之后的意思）。这是任何架构设计都必须遵守的一个约定，即持锁者必须尽量缩短自身持锁的时间，nginx 的设计也不例外，所以也照此把大部分事件延后到释放锁之后再去做处理，缩短自身持锁的时间能让其它进程尽可能的有机会获取到锁；如果当前进程没有拥有锁，那么就把事件监控机制阻塞点（比如 `epoll_wait`）的超时时间限制在一个比较短的范围内（即 `ngx_accept_mutex_delay`，可通过指令 `accept_mutex_delay` 配置，默认值为 500 毫秒），超时更快，那么也就更频繁的从阻塞中跳出来，也就有更多的机会去争抢到互斥锁。

没有拥有锁的进程接下来的操作与无负载均衡情况没有什么不同，所以下面开始重点拥有锁的进程对事件的处理，这也就是前面提到的延迟处理。当一个事件发生时，一般处理（即不做延迟）会立即调用事件对应的回调函数，而延迟处理则会将该事件以链表的形式缓存起来，可以看 `epoll` 模型里的代码作为示例：

```

556: Filename : ngx_epoll_module.c
557: static ngx_int_t
558: ngx_epoll_process_events(ngx_cycle_t *cycle, ngx_msec_t timer, ngx_uint_t flags)
559: {
560: ...
672:     if (flags & NGX_POST_EVENTS) {
673:         queue = (ngx_event_t **) (rev->accept ?
674:             &ngx_posted_accept_events : &ngx_posted_events);
675:
676:         ngx_locked_post_event(rev, queue);
677:
678:     } else {
679:         rev->handler(rev);
680:     }

```

```

681: ...
706:         if (flags & NGX_POST_EVENTS) {
707:             ngx_locked_post_event(wev, &ngx_posted_events);
708:
709:         } else {
710:             wev->handler(wev);
711:         }

```

第 679 和 710 行是直接调用事件回调函数进行处理，而另外的代码是进行事件缓存，即加到 `ngx_posted_accept_events` 链表（新建连接事件，也就是监听套接口上的发生的可读事件）或 `ngx_posted_events` 链表。

回到我们讨论的最初函数 `ngx_process_events_and_timers()`，看最后一点相关内容：

```

199: Filename : ngx_event.c
200: void
201: ngx_process_events_and_timers(ngx_cycle_t *cycle)
202: {
203: ...
247:     (void) ngx_process_events(cycle, timer, flags);
248: ...
254:     if (ngx_posted_accept_events) {
255:         ngx_event_process_posted(cycle, &ngx_posted_accept_events);
256:     }
257:
258:     if (ngx_accept_mutex_held) {
259:         ngx_shmtx_unlock(&ngx_accept_mutex);
260:     }
261: ...
269:     if (ngx_posted_events) {
270: ...
274:         ngx_event_process_posted(cycle, &ngx_posted_events);
275: ...
276:     }

```

在 `ngx_process_events()` 函数调用里已经将所有事件延迟保存，接下来先处理新建连接缓存事件 `ngx_posted_accept_events`，此时还不能释放锁，因为我们还在处理监听套接口上的事件，还要读取上面的请求数据，所以必须独占，一旦缓存的新建连接事件全部被处理完就必须马上释放持有的锁。请求的具体处理与响应是最消耗时间的，不过在此之前已经释放了持有的锁后，所以即使慢一点也不会影响到其它进程。补充两点：第一，如果在处理新建连接事件的过程中，在监听套接口上又来了新的请求会怎么样？这没有关系，当前进程只处理已缓存的事件，新的请求将被阻塞在监听套接口上，会等到下一轮被哪个进程争取到锁并加到事件处理机制监控里时才会触发而被抓取出来。第二，第 259 行只是释放锁而并没有将监听套接口从事件处理机制监控里删除，所以有可能在接下来处理 `ngx_posted_events` 缓存事件

的过程中, 互斥锁被另外一个进程争抢到并且把所有监听套接口加入到它的事件处理机制监控里, 因此严格来说, 在同一时刻, 监听套接口可能被多个进程拥有, 但是, 在同一时刻, 监听套接口只可能被一个进程监控 (也就是 `epoll_wait()` 这种), 因此进程在处理完 `ngx_posted_events` 缓存事件后去争用锁, 发现锁被其它进程占有而争用失败, 会把所有监听套接口从自身的事件处理机制监控里删除, 然后才去进行事件监控。

最后, 说一下 `nginx` 在多核平台上针对负载均衡所做的工作, 也就是 `worker_cpu_affinity` 指令, 利用该指令可以将各个工作进程固定在指定的 CPU 核上执行。关于多核平台的优化, 说起来内容比较多, 但最核心的思路就是 `per-cpu` 化处理, 小到程序内部变量, 大到架构设计都是如此, 只有这样才有可能做到性能按 CPU 线性扩展, 对于 `nginx` 用到的 `cpu_affinity`, 即 `cpu` 亲和性, 也是如此。`cpu affinity`, 简单点说就是让某一段代码/数据尽量在指定的某一个或几个 `cpu` 核心上长时间运行/计算的机制。`nginx` 这里用到的把工作进程绑定到指定 `cpu` 是 `cpu affinity` 的其中一种应用, 另外一种典型应用就是网卡收发包时硬中断的多 `cpu` 绑定, 等等, 这样做的最直观好处就是能够大大提高 `cpu cache` 的命中率, 提高性能。关于 `cpu affinity` 的 api 使用介绍以及 `cpu cache` 对性能的影响, 请参考: <http://lenky.info/?p=1262>、<http://lenky.info/?p=310>、<http://lenky.info/?p=1784>, 以及相应系统的 man 手册。下面仅看一下 `nginx` 内 `cpu affinity` 的使用配置, 其实非常简单, 首先根据系统 `cpu` 个数设定工作进程数目, 我这里只有两个核, 所以就指定 2 个工作进程 (也可以指定 4、6、8 等, 一般情况下肯定是与 `cpu` 数成倍数), 并且要让工作进程 0 运行在 0 号 CPU 上, 工作进程 1 运行在 1 号 CPU 上 (都是从 0 开始编号):

```
00: Filename : nginx.conf
01: worker_processes 2;
02: worker_cpu_affinity 01 10;
```

`worker_cpu_affinity` 指令的配置值是位图表示法, 从前往后分别是 0 号工作进程、1 号工作进程、...、n 号工作进程的 `cpu` 二进制掩码 (各个掩码之间用空格隔开), 所以这里 0 号工作进程的 `cpu` 掩码为 01, 表示其使用 0 号 `cpu`, 1 号工作进程的 `cpu` 掩码为 10, 表示其使用 1 号 `cpu`; 如果哪个工作进程的 `cpu` 掩码为 11, 则表示其既使用 0 号 `cpu`, 又使用 1 号 `cpu`。

使用这个配置文件来执行 `nginx`, 利用 `ps` 的 -F 选项查看:

```
[root@localhost ~]# ps -efHF | grep UID | grep -v grep
UID      PID  PPID  C   SZ   RSS  PSR  STIME TTY          TIME CMD
[root@localhost ~]# ps -efHF | grep nginx | grep -v grep
root      2223    1    0  1291   564    1  00:49 ?           00:00:00 nginx: master process ./nginx
nobody    2224   2223    0  1329   892    0  00:49 ?           00:00:00 nginx: worker process
nobody    2225   2223    0  1329   800    1  00:49 ?           00:00:00 nginx: worker process
```

PSR 列对应的就是进程所在 `cpu` 号, 可以看到 0 号工作进程 (即 2224) 的 `cpu` 号为 0, 而 1 号工作进程 (即 2225) 的 `cpu` 号为 1。

将配置修改一下: `worker_cpu_affinity 10 01`;; 重启 `nginx` 再看:

```
[root@localhost ~]# ps -efHF | grep nginx | grep -v grep
root      2413      1  0  1291   568  1  01:12 ?        00:00:00 nginx: master process ./nginx
nobody    2414    2413  0  1329   836  1  01:12 ?        00:00:00 nginx: worker process
nobody    2415    2413  0  1329   880  0  01:12 ?        00:00:00 nginx: worker process
```

## 超时管理

事件超时意味着等待的事件没有在指定的时间内到达，nginx 有必要对这些可能发生超时的事件（下面统称为超时事件对象）进行统一管理，并在发生事件超时做出相应的处理，比如回收资源，返回错误等。举个具体例子来说，当客户端对 nginx 发出请求连接后，nginx 就会 `accept()` 并建立对应的连接对象 `connection`、读取客户端请求的头部信息，而读取这个头部信息显然是要求在一定的时间内完成，如果在一个有限的时间内没有读取到头部信息或读取的头部信息不完整，那么 nginx 就无法进行正常处理，并且应该认为这是一个错误/非法的请求，直接返回错误信息（"Request time out" (408)）并释放相应的资源，如果 nginx 不这样做，那么针对如此的恶意攻击就很容易实施。当然，其它需要进行事件超时监控的地方还有很多，比如读取客户端请求数据、回写响应数据、管道通信等等，下面就看 nginx 是如何对这些超时事件对象进行统一超时管理的。

对于超时管理，无非要解决两个问题：第一，超时事件对象的组织，nginx 采用的是红黑树（本节如无特殊说明，提到红黑树就是指这颗树）；第二，超时事件对象的超时检测，nginx 提供了两种方案，一种是定时检测机制，通过设置定时器，争取在每过一定的时间就对红黑树管理的所有超时事件对象进行一次超时扫描检测。另一种方案是先计算出距离当前最快发生超时的时间是多少，假设时间为 `t` 秒，那么就等待 `t` 秒（其实是事件处理模型阻塞 `t` 秒）后去进行一次超时检测。

先看超时事件对象的组织结构红黑树，我们知道 nginx 把事件封装在一个名为 `ngx_event_s` 的结构体内，而该结构体有几个字段与 nginx 的超时管理联系紧密：

```
37: Filename : ngx_event.h
38: struct ngx_event_s {
39: ...
67:     unsigned          timeout:1;
68:     unsigned          timer_set:1;
69: ...
134:     ngx_rbtree_node_t  timer;
```

其中 `timeout` 域字段用于标识当前事件是否已经超时，0 为没有超时；`timer_set` 域字段用于标识当前事件是否已经加入到红黑树管理，需要对其是否超时做监控，0 为没有加入；而 `timer` 字段，很容易看出它属于红黑树节点类型变量，红黑树就是通过该字段来组织所有的超时事件对象。

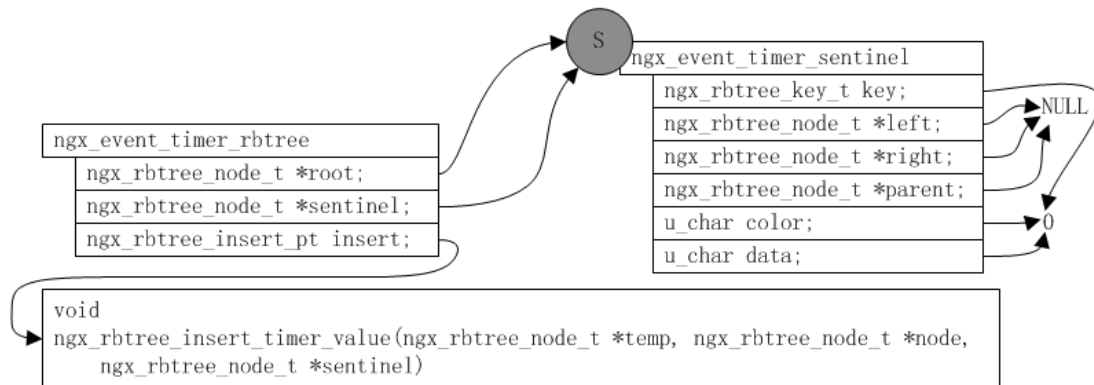
nginx 设置了两个全局变量以便在程序的任何地方都能快速的访问到这颗红黑树：

```
17: Filename : ngx_event_timer.c
```

```
18: ngx_thread_volatile ngx_rbtrees_t ngx_event_timer_rbtrees;
19: static ngx_rbtrees_node_t          ngx_event_timer_sentinel;
```

ngx\_event\_timer\_rbtrees 封装了整棵红黑树结构，而 ngx\_event\_timer\_sentinel 属于红黑树节点类型变量，在红黑树的操作过程中被当作哨兵使用，同时注意到它是 static 的，所以作用域仅限于 ngx\_event\_timer.c 源文件内。

红黑树的初始化函数 ngx\_event\_timer\_init() 是在 ngx\_event\_process\_init() 函数内被调用，所以每个工作进程都会在自身的初始化时建立这颗红黑树：



当需要对某个事件进行超时监控时，就会把它加入到这个红黑树内。仍以之前的例子来说，在 nginx 调用 accept() 接受到客户端请求并建立对应的连接对象 connection 后，在连接对象的初始化函数 ngx\_http\_init\_connection() 内，可以找到这么一行代码：

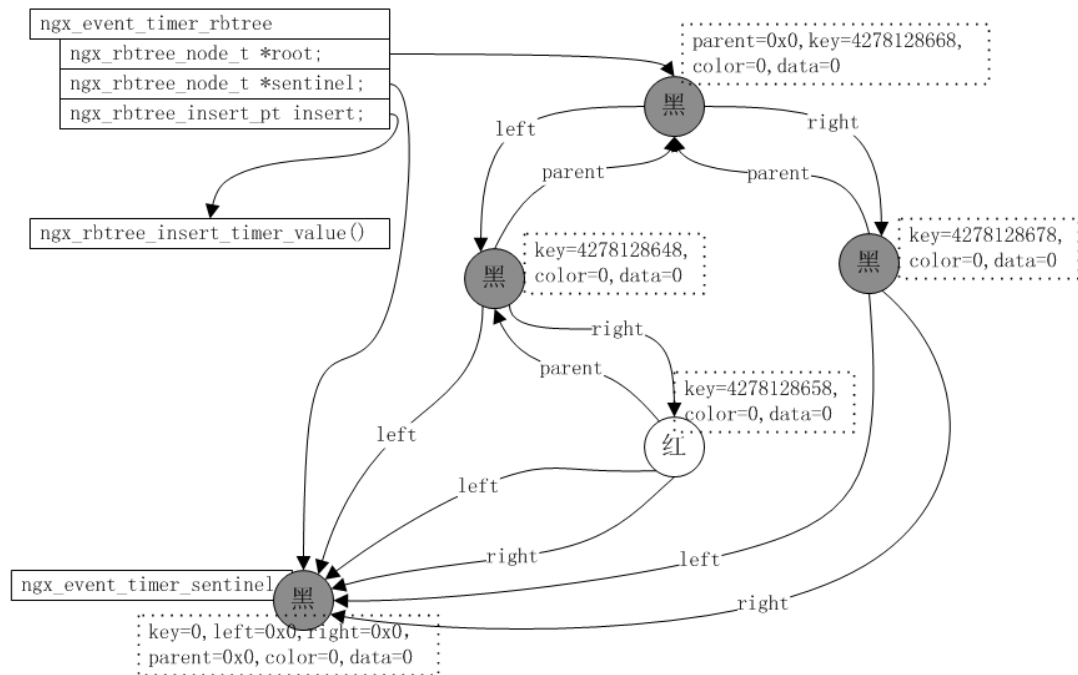
```
224: Filename : ngx_http_request.c
225:     ngx_add_timer(rev, c->listening->post_accept_timeout);
```

这也就是将 rev 事件（触发该事件即表示客户端传来请求头等信息）对象加入到红黑树内进行超时管理，同时给它指定的超时时限为 c->listening->post\_accept\_timeout（该变量的值可由用户通过 client\_header\_timeout 指令进行配置，默认情况下是 60000 毫秒）。

函数 ngx\_add\_timer() 完成将一个超时事件对象加入到红黑树的具体逻辑，代码非常的简单，首先在对应树节点的 key 字段里记录超时时刻（在后续进行超时检测扫描时就需要该字段来进行时刻的先后比较），然后判断该超时事件对象是否已经加入到红黑树，如果是的话则需要先调用函数 ngx\_del\_timer() 将它从红黑树里移除，最后再调用 ngx\_rbtrees\_insert() 函数将超时事件对象真正加入到红黑树。另外可以看到，这种加入是间接性的，根据前面的介绍可知，每个事件对象封装结构体都有一个 timer 字段，并且其为 ngx\_rbtrees\_node\_t 类型变量，加入到红黑树的就是这个字段，而非事件对象结构体本身。当然，可以通过利用 offsetof 宏来根据该 timer 字段快速方便的找到其所在的对应事件对象结构体，所以并不用为这种设计而担心。

具有四个节点的红黑树描述如下图所示，从该图中可以看到两点：第一，可以通过全局变量 ngx\_event\_timer\_rbtrees.root 快速定位到该红黑树的根节点；第二，从该红黑树根节点从左或从右遍历下去，最后都将到达全局变量 ngx\_event\_timer\_sentinel 指定的末端树节点，这

也是前面称 ngx\_event\_timer\_sentinel 为哨兵节点的原因所在。



通过红黑树，nginx 对那些需要关注其是否超时的事件对象就有了统一的管理，nginx 可以选择在合适的时机对事件计时红黑树管理的事件进行一次超时检测，对于超时了的事件对象就进行相应的处理，这在前面曾提到过 nginx 的超时检测方案有两种，下面就来分别介绍。

Nginx 具体使用哪种超时检测方案主要取决于一个配置指令 timer\_resolution，比如：

03: Filename : nginx.conf

04: timer\_resolution 100ms;

反映到 nginx 内，也就是全局变量 ngx\_timer\_resolution 的值为 100，再接下来分析，就又要看工作进程的核心处理函数 ngx\_process\_events\_and\_timers():

199: Filename : ngx\_event.c

200: void

201: ngx\_process\_events\_and\_timers(ngx\_cycle\_t \*cycle)

202: {

203: ...

206: if (ngx\_timer\_resolution) {

207: timer = NGX\_TIMER\_INFINITE;

208: flags = 0;

209:

210: } else {

211: timer = ngx\_event\_find\_timer();

212: flags = NGX\_UPDATE\_TIME;

213: ...

247: (void) ngx\_process\_events(cycle, timer, flags);

可以看到 ngx\_timer\_resolution 变量是否非 0 主要影响了两个变量的值：timer 和 flags。



先看非 0 情况，也就是超时检测方案 1，此时 flags 值为 0，可以认为这表示对其它地方代码逻辑无附加影响，而 timer 为无限大（即：#define NGX\_TIMER\_INFINITE (ngx\_msec\_t)-1），而该值在 ngx\_process\_events()函数内将被用作事件处理机制可以等待的最长时间，那么将 timer 设置为无限大会使得工作进程在事件处理机制里会无限阻塞而导致超时事件得不到及时处理么？当然不会，先不说正常情况下，事件处理机制肯定会监控到某些 I/O 事件发生，即便是因为服务器太空闲，没有任何 I/O 事件发生，工作进程也不会无限阻塞，因为工作进程在一开始就设置好了一个定时器，这实现在初始化函数 ngx\_event\_process\_init()内，关于这个函数前面曾多次提到，所以下面直接看相关代码：

```
642: Filename : ngx_event.c
643:         sa.sa_handler = ngx_timer_signal_handler;
644: ...
652:         itv.it_interval.tv_sec = ngx_timer_resolution / 1000;
653:         itv.it_interval.tv_usec = (ngx_timer_resolution % 1000) * 1000;
654:         itv.it_value.tv_sec = ngx_timer_resolution / 1000;
655:         itv.it_value.tv_usec = (ngx_timer_resolution % 1000) * 1000;
656:
657:         if (setitimer(ITIMER_REAL, &itv, NULL) == -1) {
```

通过 setitimer()函数设置的定时器会自动循环，所以每隔 ngx\_timer\_resolution 毫秒，工作进程就将收到一个定时事件，将其从事件处理机制的阻塞等待里唤醒出来（如果它正处于阻塞状态）。定时事件的回调函数为 ngx\_timer\_signal\_handler()，该函数简单扼要，仅设置一下标记：ngx\_event\_timer\_alarm = 1；，这倒非常符合信号中断处理函数的特点。

只有在 ngx\_event\_timer\_alarm 为 1 的情况下，工作进程才会更新它的时间，也就是工作进程的时间粒度为 ngx\_timer\_resolution：

```
573: Filename : ngx_epoll_module.c
574:         events = epoll_wait(ep, event_list, (int) nevents, timer);
575:
576:         err = (events == -1) ? ngx_errno : 0;
577:
578:         if (flags & NGX_UPDATE_TIME || ngx_event_timer_alarm) {
579:             ngx_time_update();
580:         }
```

从上面代码可以看到，就算工作进程被 I/O 事件唤醒而执行到第 578 行，但只要 ngx\_event\_timer\_alarm 不为 1 就不会（从前面可知，在这里讨论的超时检测方案 1 下，第 578 行前半句判断为假）执行时间更新函数 ngx\_time\_update()，从而导致下面的第 262 行也为假，超时检测函数 ngx\_event\_expire\_timers()不会也不会被执行到：

```
244: Filename : ngx_event.c
245:         delta = ngx_current_msec;
246:
247:         (void) ngx_process_events(cycle, timer, flags);
```



```

248:
249:     delta = ngx_current_msec - delta;
250: ...
262:     if (delta) {
263:         ngx_event_expire_timers();
264:     }

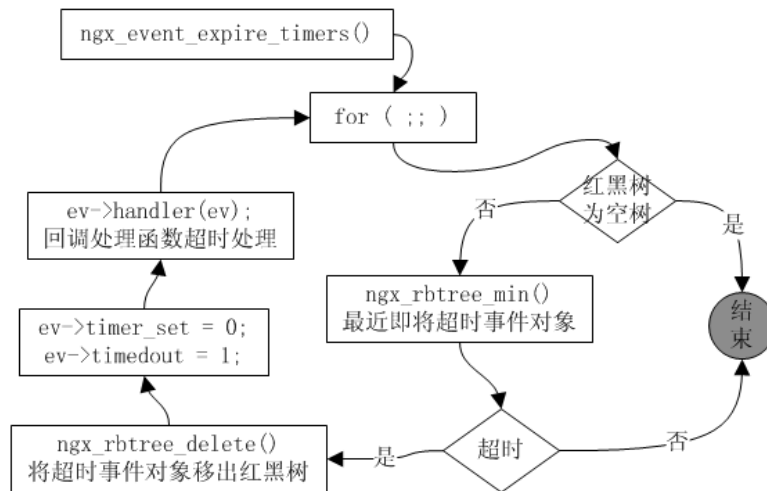
```

如果经过了 `ngx_timer_resolution` 毫秒，执行了定时函数 `ngx_timer_signal_handler()`，设置了 `ngx_event_timer_alarm` 值为 1，又更新了时间，那么第 263 行的超时检测函数 `ngx_event_expire_timers()` 自然会被执行到，这无需多说，下面再来看 `ngx_timer_resolution` 为 0 的情况，即超时检测方案 2。

在超时检测方案 2 里，`timer` 的值被设置为最快发生超时的事件对象的超时时刻与当前时刻的时间差。举个例子来说，比如红黑树管理着三个事件 a、b、c，它们分别将在 5000、6000、7000 毫秒后超时，那么距离当前最快发生超时的就是事件 a，而事件 a 的超时时刻与当前时刻的时间差为 5000 毫秒，因此变量 `timer` 的值就将被设置 5000。`timer` 值的具体计算实现在函数 `ngx_event_find_timer()` 内，该函数从红黑树内找到 `key` 值最小（`key` 值记录的就是事件的超时时刻，那么该值最小的节点表示的也就是距离当前最快发生超时的事件）的节点，然后用该节点的 `key` 值减去当前时刻（`ngx_current_msec`，事实上由于该值并不是完全实时的，所以和精确的当前时刻会有一些偏差，不过不影响）即得到预期的 `timer` 值。预期的 `timer` 值可能为负数，这表示已经有事件超时了，因此直接将 `timer` 值设置为 0，那么事件处理机制在开始监控 I/O 事件时会立即返回，以便能马上处理这些超时事件；另一个变量 `flags` 被标记为 `NGX_UPDATE_TIME`，从前面第 578 行代码可以看到函数 `ngx_time_update()` 将被执行，时间被更新，也就是说事件处理机制每次返回都会更新时间，如果 I/O 事件比较多（比如客户端请求非常的多），那么将会导致比较频繁的调用 `gettimeofday()` 系统函数，这也可以说是超时检测方案 2 的最大缺点。

回过头来看超时检测方案 1，简单直观、容易理解，但有可能导致一些超时事件得不到及时的处理，不过这并不会造成多大问题，如果不放心则可以根据应用环境通过配置指令 `timer_resolution` 适当的调整一下 `ngx_timer_resolution` 值即可。

来看最后一个需要讨论的问题，即对超时事件对象是否超时需进行的扫描检测以及对已超时事件对象的处理。由于工作进程利用红黑树来组织管理超时事件对象，因此检测是否有事件对象超时并不需要遍历扫描所有的超时事件对象，而直接找到最近的即将超时的超时事件对象，判断其是否超时，如果超时则将其移出红黑树、设置其超时标记（即将 `ev->timed_out` 置为 1）并调用该事件对应的回调处理函数进行处理，处理完了再判断第二近的即将超时的超时事件对象，如此反复，直到遇到某个超时事件对象还未超时或所有超时事件对象都已超时并处理完毕就结束检测；这整个逻辑具体实现都在函数 `ngx_event_expire_timers()` 内，流程图如下所示：



## 第六章 变量机制

### 初识变量

前面曾讲过 nginx 配置文件的解析过程，也就是 nginx 如何在启动的过程中对用户设定的配置文件进行解析，并将配置文件中的各个配置项与配置值转换为对应的 nginx 内部变量值，从而能让 nginx 按照用户预想的情况去运行。

如果只是一些比较简单并且确定的功能配置需求，那么 nginx 用户能够很方便的做出相应的设定，比如用户想要设置工作进程数为 2 个，那么配置文件中这样写即可：worker\_processes 2;; 与此同理，nginx 也很容易做到按用户的配置要求去执行，比如这里 nginx 也只需执行且仅执行 2 次 fork()函数来生成工作进程即可，具体实现可利用 for 循环并通过控制上限值来做到：

```
360: Filename : ngx_process_cycle.c
361:     for (i = 0; i < n; i++) {
362: ...
365:         ngx_spawn_process(cycle, ngx_worker_process_cycle, NULL,
366:                             "worker process", type);
```

在上面的源代码里，for 循环的条件判断上限值 n（也就是 ccf->worker\_processes）即为 2，它是通过解析配置项 worker\_processes 时根据用户的具体设定而赋值的。

如果是更高级一点的功能配置，比如当请求连接的客户端是 ie 浏览器时，nginx 能自动将请求文件重定向到/msie 目录下，那么 nginx 用户在配置文件里又该如何去表达这个逻辑呢？熟悉 nginx 的用户肯定知道要实现这个需求，我们可以这样配置（来之官方 wiki 文档示例：<http://wiki.nginx.org/HttpRewriteModule#if>）：

```
49: Filename : nginx.conf
50:     if ($http_user_agent ~ MSIE) {
51:         rewrite ^(.*)$ /msie/$1 break;
52:     }
```

这样，我们用非 ie 浏览器访问该 web 站点时，请求的文件来之其根目录，而用 ie 浏览器访问该 web 站点时，请求的文件却来之其根目录下的 msie 文件夹（事实上，如果用 ie 浏览器做目录访问，即后面不带文件名，如果 nginx 配置了 index 模块，那么访问可能会出现这样的错误：2012/05/25 11:19:25 [error] 4274#0: \*3 open() "/usr/local/nginx/web/msie/msie//index.html" failed (2: No such file or directory), client: 192.168.164.1, server: localhost, request: "GET / HTTP/1.1", host: "192.168.164.2"，可以看到是因为被映射了两次，即首先根目录匹配，由/映射为/msie/，然后被 index 模块改为/msie//index.html 后重定向，又匹配到 if 条件被再次映射为/msie/msie//index.html 而导致路径错乱。关于这个错误以及官方提到的可以考虑用 try\_files 替代 if 等暂不做过多讨论，本节仅以此作为示例讨论 nginx 变量）。

从上面的配置文件相关内容来看，对于稍懂一点编程知识的人来说，直观上这并没有什

么难以理解的地方，无非先一个判断客户端是否为 ie 浏览器，是则将 URI 重定向到 msie，否则继续原 URI 的操作，这看似非常简单的逻辑却至少需要一个东西的支撑，也就是必须要有一个符号（或别的什么）来代表客户端浏览器，nginx 用户才能在配置文件里表达类似“当‘客户端浏览器’是什么，nginx 就该怎么样，如果不是，nginx 又该怎么样”这样的语义，而这个符号也就是本节将要重点介绍的 nginx 变量，如上面示例配置中的 \$http\_user\_agent 就是一个 nginx 变量。

对于 nginx 而言，变量是指配置文件中以\$开头的标识符（整个本章都不涉及 SSI 模块的变量，因为其比较独特，留待后面篇章专讲），这和编程语言 PHP 里的变量命名要求基本一致，当然，nginx 变量的功能等各个方面肯定都要相对简单得多，这是不言而喻的，够用就好，毕竟 nginx 的主要功能不在这里。

和其它编程语言里的变量意义一致，nginx 的变量也同样是指明有一块内存空间，其存放了会根据情况发生变化的动态值。比如，对于变量 \$http\_user\_agent 所代表的一块内存空间而言，客户端用 ie 浏览器访问时，其内存放的值为 MSIE，用非 ie 浏览器访问时，其内存放的值也许就变化为 Opera 或 Safari 等（根据客户端浏览器类型而定），但肯定就不是 MSIE 了，否则上下文中的 if 判断逻辑将失去它的作用，用户的设置也将失效。

不像 PHP 或 C 语言那样拥有众多的变量类型，nginx 只有一种变量类型，即字符串，而且既然变量是用在配置文件中，那么根据曾在配置解析一章的讲解，变量值字符串加或者不加引号，加双引号或单引号都没有什么影响，除非字符串内包含有空格，需要利用引号或用转义字符（\）将它前后的字符连成一个字符串。

Nginx 变量所代表的内存里存放的字符串当然不是凭空生成的，就像是在 C 语言里，我们定义一个变量后总会直接或间接的给它赋值，否则读取出来的就是垃圾数据，所以 nginx 变量也会被赋值，不过这种赋值大部分情况下是自动的，并且是延后的。

自动赋值的意思很简单，比如在上面的示例中，在整个配置文件内，我都没有对变量 \$http\_user\_agent 进行赋值操作，但是却可以直接拿它来用，因为我知道在每一个客户端请求连接里，这个变量都会自动的被 nginx 赋值，要么为 MSIE，或 Opera、或 Safari 等，当然，这大家都知道原因，因为它是 nginx 内部变量。其实，我们实际使用中，大部分情况也就是使用内部变量，一方面在于 nginx 提供的内部变量非常的多，基本考虑了大多数使用场景，另一方面，如果你使用外部变量（或称之为自定义变量），那么就得给它赋值，如果是将一个确定的值（或内部变量）赋值给它，那么在使用这个变量的地方用这个确定的值（或内部变量）就行了，何必多此一举，除非是要根据特殊逻辑组织多个不同的确定值和（或）内部变量在一起成一个新的变量，不过这种情况一般也都比较少。

延后赋值，专业术语叫惰性求值（Lazy Evaluation），其实说清楚了也容易懂，它是从性能上的考虑。nginx 光内部变量就有好几十个，如果每一个客户端请求，nginx 都去给它们赋好值，但是配置文件里却有根本没用到，这岂不是大大的性能浪费？所以，对于大部分

变量，只有真正去读它的值时，nginx 才会临时执行一段代码先给它赋上相应的值，然后再将结果返回（当然还有其它细节，比如如果之前 nginx 已经给它赋好了值并且有效，就不用做第二次赋值直接返回即可，等），这种优化与编程中的另一种常见技术，即写时复制（Copy On Write）有异曲同工之妙。

内部变量意味着变量名是预先定义好的，Nginx 目前具体提供有哪些预定义好的内部变量以及每个变量的含义在官方 wiki 文档（比如：<http://wiki.nginx.org/HttpCoreModule#Variables>、[http://wiki.nginx.org/HttpGeoipModule#geoip\\_country](http://wiki.nginx.org/HttpGeoipModule#geoip_country)）上可以查看，也可以通过源代码（检索关键字：ngx\_http\_variable\_t）根据变量名的英文单词猜测其代表的大致含义。除了 http 核心模块 ngx\_http\_core\_module 提供了大量的内部变量之外，其它模块比如 ngx\_http\_fastcgi\_module、ngx\_http\_geoip\_module 等也有一些内部变量，如果我们自己开发 nginx 模块，自然也可以提供类似这样的内部变量供用户在 nginx 配置文件里使用。

除了内部变量之外，与之相对的就是外部变量（或称之为自定义变量），外部变量是 nginx 用户在配置文件里定义的变量，因此变量名可由用户随意设定，当然也是要以\$开头，并且得注意不要覆盖内部变量名。目前 nginx 主要是通过 ngx\_http\_rewrite\_module 模块的 set 指令来添加外部变量，当然也有其它模块比如 ngx\_http\_geo\_module 来新增外部变量，这些在后面其它章节的分析中会看到其具体的实现。

## 支撑机制

任意一个变量，都有其变量名和变量值，nginx 与此对应的封装分别为结构体 ngx\_http\_variable\_s 和 ngx\_variable\_value\_t:

```

16: Filename : ngx_http_variables.h
17: typedef ngx_variable_value_t  ngx_http_variable_value_t;
35: struct ngx_http_variable_s {
36:     ngx_str_t                name;    /* must be first to build the hash */
37:     ngx_http_set_variable_pt  set_handler;
38:     ngx_http_get_variable_pt  get_handler;
39:     uintptr_t                 data;
40:     ngx_uint_t                flags;
41:     ngx_uint_t                index;
42: };
27: Filename : ngx_string.h
28: typedef struct {
29:     unsigned    len:28;
30:
31:     unsigned    valid:1;
32:     unsigned    no_cacheable:1;
33:     unsigned    not_found:1;

```

```

34:     unsigned    escape:1;
35:
36:     u_char      *data;
37: } ngx_variable_value_t;

```

可以看到这两个结构体并非只是简单的包含其名与值，还有其它相关的辅助字段，甚至结构体 `ngx_http_variable_s` 本身就包含一个 `data` 字段，看似是用来存放变量值的地方，那为什么又还要一个专门的 `ngx_variable_value_t` 结构体来封装 `nginx` 变量值呢？关于这个问题，在本节后面的讲解中会逐步清晰，这里暂且不讲。

在进行配置解析之前，`nginx` 会统计其支持的所有内部变量，也即在每个模块的回调函数 `module->preconfiguration` 内，将模块自身支持的内部变量统一加入到 `http` 核心配置 `ngx_http_core_main_conf_t` 的 `variables_keys` 字段内：

```

149: Filename : ngx_http_core.module.h
150: typedef struct {
151: ...
157:     ngx_hash_t          variables_hash;
158:
159:     ngx_array_t          variables;      /* ngx_http_variable_t */
160: ...
168:     ngx_hash_keys_arrays_t *variables_keys;
169: ...
175: } ngx_http_core_main_conf_t;

```

就以 `http` 核心模块 `ngx_http_core_module` 为例，其模块的 `preconfiguration` 回调函数为 `ngx_http_core_preconfiguration()`，该函数就一条语句：调用 `ngx_http_variables_add_core_vars()` 函数，从而将自身支持的所有内部变量（组织在 `ngx_http_core_variables` 数组内）加入到 `cmcf->variables_keys` 变量内：

```

2014:  Filename : ngx_http_variables.c
2015:  ngx_int_t
2016:  ngx_http_variables_add_core_vars(ngx_conf_t *cf)
2017:  {
2018:  ...
2022:      cmcf = ngx_http_conf_get_module_main_conf(cf, ngx_http_core_module);
2023:
2024:      cmcf->variables_keys = ngx_palloc(cf->temp_pool,
2025:                                       sizeof(ngx_hash_keys_arrays_t));
2026:  ...
2039:      for (v = ngx_http_core_variables; v->name.len; v++) {
2040:          rc = ngx_hash_add_key(cmcf->variables_keys, &v->name, v,
2041:                               NGX_HASH_READONLY_KEY);
2042:  ...

```

上面代码中，函数 `ngx_hash_add_key()` 是实际执行往变量 `cmcf->variables_keys` 内进行新增操作的函数，除了 `http` 核心模块 `ngx_http_core_module` 以外，其它模块都会这么直接或间



接的把自身支持的内部变量加到 `cmcf->variables_keys` 内，再比如 `ngx_http_proxy_module` 模块，其相关执行过程如下：

```
ngx_http_proxy_add_variables() -> ngx_http_add_variable() -> ngx_hash_add_key()
```

其中 `ngx_http_proxy_add_variables()` 是 `ngx_http_proxy_module` 模块的 `preconfiguration` 回调函数。不仅是内部变量，用户自定义的外部变量在配置文件的解析过程中也会被添加到 `cmcf->variables_keys` 内，这从外部变量的主要设置指令 `set` 的回调函数 `ngx_http_rewrite_set()` 的内部实现即可看出：

```
ngx_http_rewrite_set() -> ngx_http_add_variable() -> ngx_hash_add_key()
```

总之，当 `nginx` 解析配置正常结束时，所有的变量都被集中在 `cmcf->variables_keys` 内，那这有什么作用呢？继续来看。

`Nginx` 在配置文件的解析过程中，会遇到用户使用变量的情况，如最前面的配置示例中使用了变量 `$http_user_agent`，所有这些被用户在配置文件里使用的变量都会先通过 `ngx_http_get_variable_index()` 函数而被添加到 `cmcf->variables` 内（配置文件中出现：`set $file t_a`；，在这里这个 `$file` 变量既是定义，又是使用，先定义它，然后把字符串 `"t_a"` 赋值给它，这也是一种使用，所以它会被加入到 `cmcf->variables` 内，可以简单的认为 `nginx` 在解析配置文件的过程中遇到的所有变量都会被加入到 `cmcf->variables` 内；有些变量虽然没有出现在配置文件内，但是以 `nginx` 默认设置的形式出现在源代码里，比如 `ngx_http_log_module` 模块内的 `ngx_http_combined_fmt` 全局静态变量里就出现了一些 `nginx` 变量，也会被加入到 `cmcf->variables` 中；另外，有些变量是模块自身特有的，比如 `ngx_http_log_module` 模块内的 `$time_local` 变量，其模块自身具体专有逻辑来独自处理，从而没有加入到 `cmcf->variables` 内；`nginx` 的哲学是怎么高效就怎么做，除非是对代码框架影响特别大，这也是我们在看源代码的过程中要注意的，所以我的描述也只能针对大多数情况，即便是我在叙述的过程中使用了“全”、“都”这样的字词也不代表就是绝对如此），这和我前面描述的一致，虽然 `nginx` 默认提供的变量有很多，但只需把我们在配置文件里真正用到了的变量给挑出来。当配置文件解析完后，所有用到的变量也被集中起来了，所有这些变量需要检查其是否合法，因为 `nginx` 不能让用户在配置文件里使用一个非法的变量，这就需要 `cmcf->variables_keys` 的帮忙。

这个合法性检测逻辑很简单，实现在函数 `ngx_http_variables_init_vars()` 内，其遍历 `cmcf->variables` 内收集的所有已使用变量，逐个去已定义变量 `cmcf->variables_keys` 集合里查找，如果找到则表示用户使用无误，如果没找到，则需要注意，这还只能说明它可能是一个非法变量，因为有一点之前一直没讲，那就是有一部分变量虽然没有包含在 `cmcf->variables_keys` 内，但是它们却合法，这部分变量是以 `"http_"`、`"sent_http_"`、`"upstream_http_"`、`"cookie_"`、`"arg_"` 开头的五类变量，这些变量庞大并且不可预知，不可能提前定义并收集到 `cmcf->variables_keys` 内，比如以 `"arg_"` 开头代表的参数类变量会根据客户端请求 `uri` 时附带的参数不同而不同，一个类似于 `"http://192.168.164.2/?pageid=2"` 这样的请求就会自动生成变量 `$arg_pa`



geid, 因此还需判断用户在配置文件里使用的变量是否在这五类变量里, 具体怎么判断也就是检测用户使用的变量名前面几个字符是否与它们一致(这也间接说明, 用户自定义变量时不要以这些字符开头)。当然, 如果用户在配置文件里使用了变量\$arg\_pageid, 而客户端请求时却并没有带上 pageid 参数, 此时也只不过是变量\$arg\_pageid 值为空而已, 但它总算是合法, 但如果提示类似如下这样的错误, 请需检查配置文件内变量名是否书写正确:

```
nginx: [emerg] unknown "x_var_test" variable
```

函数 ngx\_http\_variables\_init\_vars()在对已使用变量进行合法性检测的同时, 对于合法的使用变量会将其对应的三个主要字段设置好, 即 get\_handler()回调、data 数据、flags 旗标, 从前面给出的结构体 ngx\_http\_variable\_s 定义来看, name 存储的是变量名字符串, index 存储的是该变量在 cmcf->variables 内的下标(通过函数 ngx\_http\_get\_variable\_index()获得), 这两个都是不变的, 而 set\_handler()回调目前只在使用 set 配置指令构造脚本引擎时才会用到, 而那里直接使用 cmcf->variables\_keys 里对应变量的该字段, 并且一旦配置文件解析完毕, set\_handler()回调也就用不上了, 所以只有剩下的三个字段才需要做赋值操作, 即从 cmcf->variables\_keys 里对应变量的对应字段拷贝过来, 或是另外五类变量就根据不同类别进行固定的赋值。

先看 flags 旗标字段, 这里涉及到的旗标主要是两个: 一个为 NGX\_HTTP\_VAR\_CHANGEABLE, 表示该变量可重复添加, 该标记影响的逻辑主要是变量添加函数 ngx\_http\_add\_variable()。比如如下配置不会出错, 因为 set 指令新增的变量都是 NGX\_HTTP\_VAR\_CHANGEABLE 的:

```
49: Filename : nginx.conf
50:         set $file t_a;
51:         set $file t_b;
```

此时, set 指令会重复添加变量\$file (其实, 第 51 行并不会新增变量\$file, 因为在新增的过程中发现已经有该变量了, 并且是 NGX\_HTTP\_VAR\_CHANGEABLE 的, 所以就返回该变量使用), 并且其最终值将为 t\_b。如果新增一个不是 NGX\_HTTP\_VAR\_CHANGEABLE 的变量\$t\_var, 那么 nginx 将提示 the duplicate "t\_var" variable 后退出执行;

另一个标记为 NGX\_HTTP\_VAR\_NOCACHEABLE, 表示该变量不可缓存, 我们都知道, 所有这些变量基本都是跟随客户端请求的每个连接而变的, 比如变量\$http\_user\_agent 会随着客户端使用浏览器的不同而不同, 但是在客户端的同一个连接里, 这个变量肯定不会发生改变, 即不可能一个连接前半是 IE 浏览器而后半是 Opera 浏览器, 所以这个变量是可缓存的, 在处理这个客户端连接的整个过程中, 变量\$http\_user\_agent 值计算一次就行了, 后续使用可直接使用其缓存。然而, 有一些变量, 因为 nginx 本身的内部处理会发生变化, 比如变量\$uri, 虽然客户端发过来的请求连接 URI 是/thread-3760675-2-1.html, 但通过 rewrite 一转换却变成了/thread.php?id=3760675&page=2&floor=1, 也即是变量\$uri 发生了改变, 所以对于变量\$uri, 每次使用都必须进行主动计算(即调用回调 get\_handler()函数), 该标记

影响的逻辑主要是变量取值函数 `ngx_http_get_flushed_variable()`。当然，如果我们明确知道当前的细节情况，此时从性能上考虑，也不一定就非要去重新计算获取值，比如刚刚通过主动计算获取了变量 `$uri` 的值，接着马上又去获取变量 `$uri` 的值（这种情况当然有，例如连续将 `$uri` 变量的值赋值给另外两个不同变量），此时可使用另外一个取值函数 `ngx_http_get_indexed_variable()`，直接取值而不考虑是否可缓存标记。

再来看 `data` 数据字段，这个字段指向存放该变量值的地方，具体点说是指向结构体 `ngx_http_request_t` 变量 `r` 中的某个字段。我们知道（或者将要知道，下文会讲到）一个 `nginx` 变量总是与具体的 `http` 请求绑定在一起的，一个 `http` 请求总有一个与之对应的 `ngx_http_request_t` 变量 `r`，该变量 `r` 内存放有大量的与当前 `http` 请求相关的信息，而大部分 `nginx` 变量的值又是与 `http` 请求相关的，简而言之，`nginx` 内置变量的值大部分直接或间接的来之变量 `r` 的某些字段内。举个例子，`nginx` 内部变量 `$args` 表示的是客户端 `GET` 请求时 `uri` 里的参数，熟悉结构体 `ngx_http_request_t` 定义的人知道该结构体有一个 `ngx_str_t` 类型字段为 `args`，其内存放的就是 `GET` 请求参数，所以内部变量 `$args` 的这个 `data` 字段就是指向变量 `r` 里的 `args` 字段，表示其数据来之这里。这是直接的情况，那么间接的情况呢？看 `nginx` 内部变量 `$remote_port`，这个变量表示客户端端口号，这个值在结构体 `ngx_http_request_t` 内没有直接的字段对应，但是肯定同样也是来之 `ngx_http_request_t` 变量 `r` 里，怎么去获取就看 `get_handler()` 函数的实现，此时 `data` 数据字段没什么作用，值为 0。

最后来看 `get_handler()` 回调字段，这个字段主要实现获取变量值的功能。前面讲了 `nginx` 内置变量的值都是有默认来源的，如果是简单的直接存放在某个地方（上面讲的内部变量 `$args` 情况），那么不要这个 `get_handler()` 回调函数倒还可以，通过 `data` 字段指向的地址读取；但是如果比较复杂，虽然知道这个值存放在哪儿，但是却需要比较复杂的逻辑获取（上面讲的内部变量 `$remote_port` 情况），此时就必须靠回调函数 `get_handler()` 来执行这部分逻辑。总之，不管简单或复杂，回调函数 `get_handler()` 帮我们去在合适的地方通过合适的方式，获取到该内部变量的值，这也是为什么我们并没有给 `nginx` 内部变量赋值，却又能读到值，因为有这个回调函数的存在。来看看这两个示例变量的 `data` 字段与 `get_handler()` 回调字段情况：

```

191: Filename : ngx_http_variables.c
192:     { ngx_string("args"),
193:       ngx_http_variable_request_set,
194:       ngx_http_variable_request,
195:       offsetof(ngx_http_request_t, args),
196:       NGX_HTTP_VAR_CHANGEABLE|NGX_HTTP_VAR_NOCACHEABLE,0},
197: ...
555: static ngx_int_t
556: ngx_http_variable_request(ngx_http_request_t *r, ngx_http_variable_value_t *v,
557:   uintptr_t data)
558: {
559: ...

```

```

561:    s = (ngx_str_t *) ((char *) r + data);
562:
563:    if (s->data) {
564: ...
568:        v->data = s->data;

```

因为 data 字段的帮助, 变量 \$args 的 get\_handler() 回调函数 ngx\_http\_variable\_request() 的实现非常的简单。

```

155: Filename : ngx_http_variables.c
156:    { ngx_string("remote_port"), NULL, ngx_http_variable_remote_port, 0, 0, 0 },
157: ...
1039: static ngx_int_t
1040: ngx_http_variable_remote_port(ngx_http_request_t *r,
1041:     ngx_http_variable_value_t *v, uintptr_t data)
1042: {
1043:     ngx_uint_t      port;
1044:     ...
1059:     switch (r->connection->sockaddr->sa_family) {
1060:
1061:     #if (NGX_HAVE_INET6)
1062:         case AF_INET6:
1063:             sin6 = (struct sockaddr_in6 *) r->connection->sockaddr;
1064:             port = ntohs(sin6->sin6_port);
1065:             break;
1066:     #endif

```

再看变量 \$remote\_port 的 get\_handler() 回调函数 ngx\_http\_variable\_remote\_port() 的处理就比较麻烦了, 上面只给出了部分代码, 它根据不同的情况做不同的处理, 此时 data 字段也没用了。

一并再来看下 set\_handler(), 这个回调目前只被使用在 set 指令里, 组成脚本引擎的一个步骤, 提供给用户在配置文件里可以修改内置变量的值, 带有 set\_handler() 接口的变量非常的少, 比如变量 \$args、\$limit\_rate, 这类变量一定会带上 NGX\_HTTP\_VAR\_CHANGEABLE 标记, 否则这个接口毫无意义, 因为既然不能修改, 何必提供修改接口? 也会带上 NGX\_HTTP\_VAR\_NOCACHEABLE 标记, 因为既然会被修改, 自然也是不可缓存的。下面看看变量 \$args 的 set\_handler() 接口函数 ngx\_http\_variable\_request\_set():

```

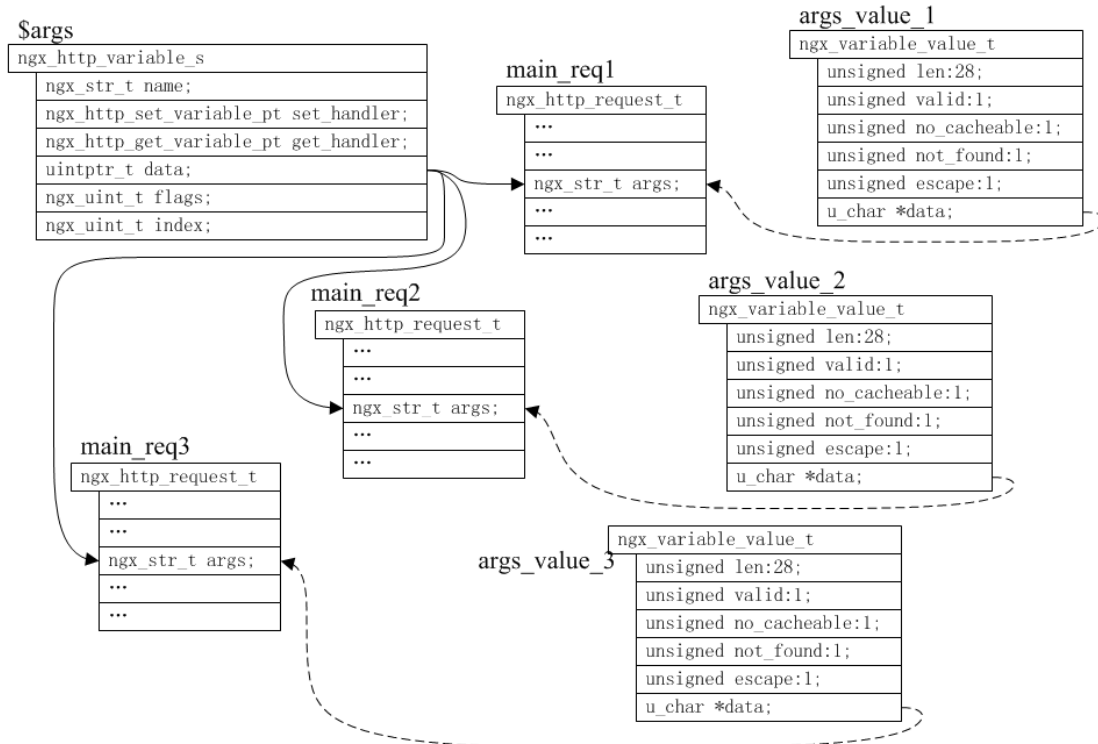
577: Filename : ngx_http_variables.c
578: static void
579: ngx_http_variable_request_set(ngx_http_request_t *r,
580:     ngx_http_variable_value_t *v, uintptr_t data)
581: {
582:     ngx_str_t *s;
583:
584:     s = (ngx_str_t *) ((char *) r + data);
585:
586:     s->len = v->len;

```

```
587:     s->data = v->data;
588: }
```

直接修改了结构体 `ngx_http_request_t` 变量 `r` 里的 `args` 字段（因为 `data` 会指向那里）。由此可以看到，不管从哪方面来讲，`data` 字段都只是一个辅助 `get_handler()`、`set_handler()` 回调处理的指示字段，在调用这两个回调函数时，会把 `data` 指定传递进来，以明确指定变量值来源的地方，简化和统一这两个回调函数的逻辑，所以你能看到大多数变量的 `get_handler()` 回调字段都是指向 `ngx_http_variable_header()`、`ngx_http_variable_request()` 这样的通用函数。其实，如果你有必要，`data` 字段完全可以设置其它值以便传到 `get_handler()`、`set_handler()` 这两个回调处理函数里，这就回答了前面的疑问：为什么结构体 `ngx_http_variable_s` 里已经包含有一个 `data` 字段了，`nginx` 还要弄一个专门的 `ngx_variable_value_t` 结构体封装来 `nginx` 变量值，因为“这个”`data` 字段不是我们设想的“那个”`data` 字段。

是否可以把 `ngx_variable_value_t` 结构体的所有字段都移到结构体 `ngx_http_variable_s` 内，将变量值和变量名组织在一起呢？非要这样做（假设合并而成的结构体为 `ngx_http_variable_name_value_t`，有些重复字段要改一下，比如 `ngx_variable_value_t` 里的 `data` 改为 `value_data` 等），当然可以，但是如果那样设计的话，以现在的代码逻辑，在 `nginx` 里使用 `nginx` 变量名时，所有 `ngx_variable_value_t` 这些字段是否都会浪费（即它们用不上）？而当使用 `nginx` 变量值时，那所有的 `ngx_http_variable_s` 那些字段又是多余（因为，此时那些字段也用不上）？举个例子，合并之后，对于变量 `$args`，就有个对应的结构体变量 `ngx_http_variable_name_value_t` 来统一描述它的名称和值，而我们知道变量是与请求相关联的，这也就是说 `nginx` 工作进程当前有处理个客户端请求正在处理，就有多少份 `$args` 变量，假设当前有 3 个客户端请求在处理，从而变量 `$args` 也就有三份，对应结构体 `ngx_http_variable_name_value_t` 里的关于对变量名的描述就有三份，这岂不是大大内存浪费？这也违背高性能设计里同一份数据只存一份的设计原则（因为存放多份一样的数据，不管是生成、更新、维护都麻烦）。按照现在 `nginx` 对变量的设计，三个请求的 `$args` 变量如下所示，可以看到 `$args` 变量名只存一份，而 `$args` 变量值根据每个请求而存三份，虚线箭头是指各个 `$args` 变量值根据 `$args` 变量名的 `data` 字段与 `http` 请求对象的 `args` 字段关联起来（调用 `get_handler()`、`set_handler()` 回调函数时，会把当前 `http` 请求对象 `r` 传递进去）：



如果合二结构体为一个，那么就是如下这样的情况，相比现在的设计，多次保存\$args变量名就是对内存的一种浪费：



现在，我们知道在 nginx 内部，对于多个变量，其变量名只会保存一次，那么怎么把变量名和变量值对应起来呢？也就是说，比如要读取变量的值，该利用哪个变量名的 get\_han



lder()回调函数呢？关键点就在变量名里的 index 字段，关于这个字段在前面说过，它的值来之将变量添加 cmcf->variables 内时所对应的数组下标，比如假定 cmcf->variables 数组内当前已有 6 个 ngx 变量，如果此时再新增一个使用变量 \$a，那么 \$a 的 index 就是 6（注意下标的序号是从 0 开始）。当然，在这里，为什么说 index 字段很关键，下面继续来看就会理解了。

继续来看函数 ngx\_http\_variables\_init\_vars()后面的逻辑，可以看到 cmcf->variables\_keys 变量指 NULL，其原本实际所占的内存空间因为在 cf->temp\_pool 内（函数 ngx\_http\_variables\_add\_core\_vars()的第 2024 行），所以在初始化基本结束后也会被释放掉（函数 ngx\_init\_cycle()的第 717 行）：

```
41: Filename : ngx_cycle.c
42: ngx_cycle_t *
43: ngx_init_cycle(ngx_cycle_t *old_cycle)
44: {
45: ...
717:     ngx_destroy_pool(conf.temp_pool);
```

因此，关于 ngx 变量，到最后，我们就剩下了一个 cmcf->variables 数组，里面存放了所有用户用到的变量，但是要清楚 cmcf->variables 数组存放的只是有可能被用到的变量，因为在实际处理客户端请求的过程中，根据请求的不同（比如请求地址、传递参数等）执行的具体路径也不相同，所以实际用到的变量也不相同。另外，刚刚讲了，cmcf->variables 数组存放的只是各个变量名（以及相关属性、回调字段），其变量值是通过另外一个结构体 ngx\_variable\_value\_t 变量来存储的，所以必须为这个变量申请对应的内存空间。这在 ngx 处理每一个客户端请求时的初始化函数 ngx\_http\_init\_request()内创建了这个存储空间：

```
236: Filename : ngx_http_request.c
237: static void
238: ngx_http_init_request(ngx_event_t *rev)
239: {
240: ...
478:     r->variables = ngx_palloc(r->pool, cmcf->variables.nelts
479:                                * sizeof(ngx_http_variable_value_t));
```

这个变量和 cmcf->variables 是一一对应的，形成 var\_name 与 var\_value 对，所以两个数组里的同一个下标位置元素刚好就是相互对应的变量名和变量值，而我们在使用某个变量时总会先通过函数 ngx\_http\_get\_variable\_index()获得它在变量名数组里的 index 下标，也就是变量名里的 index 字段值，然后利用这个 index 下标进而去变量值数组里取对应的值，这就解释了前面所提到的疑问。

对于子请求，虽然有独立的 ngx\_http\_request\_t 对象 r，但是却没有额外创建的 r->variables，和父请求（或者说主请求）是共享的，这在 ngx\_http\_subrequest()函数里可以看到相应的代码：

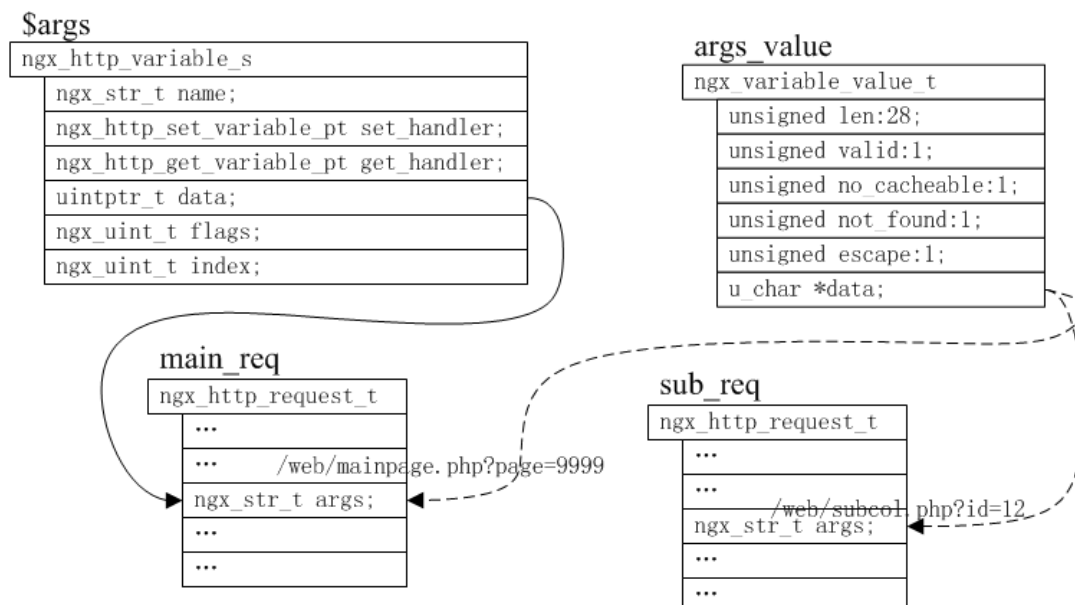
```
2365:     Filename : ngx_http_core_module.c
```

```

2366: ngx_int_t
2367: ngx_http_subrequest(ngx_http_request_t *r,
2368:     ngx_str_t *uri, ngx_str_t *args, ngx_http_request_t **psr,
2369:     ngx_http_post_subrequest_t *ps, ngx_uint_t flags)
2370: {
2371:     ...
2373:     ngx_http_request_t      *sr;
2374:     ...
2386:     sr = ngx_palloc(r->pool, sizeof(ngx_http_request_t));
2387:     ...
2455:     sr->variables = r->variables;

```

针对子请求，虽然重新创建了 `ngx_http_request_t` 变量 `sr`，但子请求的 `nginx` 变量值数组 `sr->variables` 却是直接指向父请求的 `r->variables`，其实这并不难理解，因为父子请求的大部分变量值都是一样的，当然没必要申请另外的空间，而对于那些父子请求之间可能会有不同变量值的变量，又有 `NGX_HTTP_VAR_NOCACHEABLE` 标记的存在，所以也不会有什么。比如变量 `$args`，在父请求里去访问该变量值时，发现该变量是不可缓存的，于是就调用 `get_handler()` 函数从 `main_req` 对象的 `args` 字段（即 `r->args`）里去取，此时得到的是 `page=9999`；而在子请求里去访问该变量值时，发现该变量是不可缓存的，于是也调用 `get_handler()` 函数从 `sub_req` 对象的 `args` 字段（即 `sr->args`，注意对象 `sr` 与 `r` 之间是分割开的）里去取，此时得到的是 `id=12`；因而，在获取父子请求之间可变变量的值时，并不会相互干扰：



关于 `nginx` 变量的基本支撑机制就大概是上面介绍的这些，另外值得说明的是，函数 `ngx_http_variables_init_vars()` 里还有一些没提到的代码以及相关逻辑，这包括旗标 `NGX_HTTP_VAR_INDEXED`、`NGX_HTTP_VAR_NOHASH`、变量 `cmcf->variables_hash` 以及取值函数 `ngx_http_get_variable()` 等，它们都是为 `SSI` 模块实现而设计的，所以本章暂且不讲，否则夹杂在一起反而搞混，这里仅提醒注意一下，在 `SSI` 模块专章时再回头来看这部分。



## 脚本引擎

有了对变量支撑机制的了解，下面就直接进入脚本引擎的主题，可通过“set \$file t\_a;”这个非常简单的实例来描述脚本引擎的大致情况。该实例虽然简单，但已包含脚本引擎处理的基本过程，更复杂一点的情况无非也就是回调处理多几重、相关数据多一点而已。

nginx 在解析配置文件时遇到“set \$file t\_a;”这句配置项就会执行 set 指令相应的回调函数 ngx\_http\_rewrite\_set(), 下面开始逐步分析。

首先，value 字符串数组（其实它本身只是一个字符串指针，因为它指向的是数组变量 cf->args 的 elts 字段，所以可以认为它是一个数组。类似于这种细节，后面都不再一一解释，请根据上下文环境自行理解）包含有三个元素，分别为 set、\$file、t\_a，其中 set 是指令符号，抛开不管，所以第一个被处理的字符串为 \$file，我们知道 set 是用来设置自定义变量的，所以先判断变量名是否合法（即第一个字符是否为 \$ 符号），合法则利用函数 ngx\_http\_add\_variable() 将它加入到变量集 cmcf->variables\_keys 里，同时利用函数 ngx\_http\_get\_variable\_index() 将它也加入到已使用变量集 cmcf->variables 内并获取它的对应下标 index，以便后续使用它。这些都是准备工作，其相关代码如下：

```
891: Filename : ngx_http_rewrite_module.c
892: static char *
893: ngx_http_rewrite_set(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
894: {
895: ...
905:     if (value[1].data[0] != '$') {
906: ...
914:     v = ngx_http_add_variable(cf, &value[1], NGX_HTTP_VAR_CHANGEABLE);
915: ...
919:     index = ngx_http_get_variable_index(cf, &value[1]);
```

接下来就是构建“set \$file t\_a;”所对应的脚本引擎，脚本引擎是一系列的回调函数以及相关数据（它们被组织成 ngx\_http\_script\_xxx\_code\_t 这样的结构体，代表各种不同功能的操作步骤），被保存在变量 lcf->codes 数组内，而 ngx\_http\_rewrite\_loc\_conf\_t 类型变量 lcf 是与当前 location 相关联的，所以这个脚本引擎只有当客户端请求访问当前这个 location 时才会被启动执行。如下配置中，“set \$file t\_a;”构建的脚本引擎只有当客户端请求访问/t 目录时才会被触发，如果当客户端请求访问根目录时则与它毫无关系：

```
13: Filename : nginx.conf
14:     location / {
15:         root web;
16:     }
17:     location /t {
18:         set $file t_a;
```

```
19:         }
```

这也可以说是 nginx 变量惰性求值特性的根本来源，没触发脚本引擎或没执行到的脚本引擎路径，自然不会去计算其相关变量的值。

在函数 ngx\_http\_rewrite\_set()接下来的逻辑里就如何去构建相对应的脚本引擎，“set \$file t\_a;” 配置语句比较简单，略去过多无关重要的细节，仅关注与其相关的关键执行代码路径，第一个重点关注逻辑在函数 ngx\_http\_script\_value\_code\_t()内：

```
963: Filename : ngx_http_rewrite_module.c
964: static char *
965: ngx_http_rewrite_value(ngx_conf_t *cf, ngx_http_rewrite_loc_conf_t *lcf,
966:     ngx_str_t *value)
967: {
968: ...
976:     val = ngx_http_script_start_code(cf->pool, &lcf->codes,
977:         sizeof(ngx_http_script_value_code_t));
978: ...
988:     val->code = ngx_http_script_value_code;
989:     val->value = (uintptr_t) n;
990:     val->text_len = (uintptr_t) value->len;
991:     val->text_data = (uintptr_t) value->data;
```

函数 ngx\_http\_script\_start\_code()利用 ngx\_array\_push\_n()在 lcf->codes 数组内申请了 sizeof(ngx\_http\_script\_value\_code\_t)个元素，注意每个元素的大小为一个字节，所以其实也就是为 ngx\_http\_script\_value\_code\_t 类型变量 val 申请存储空间（很棒的技巧）。接着第 988 行开始为保存回调函数以及相关数据。

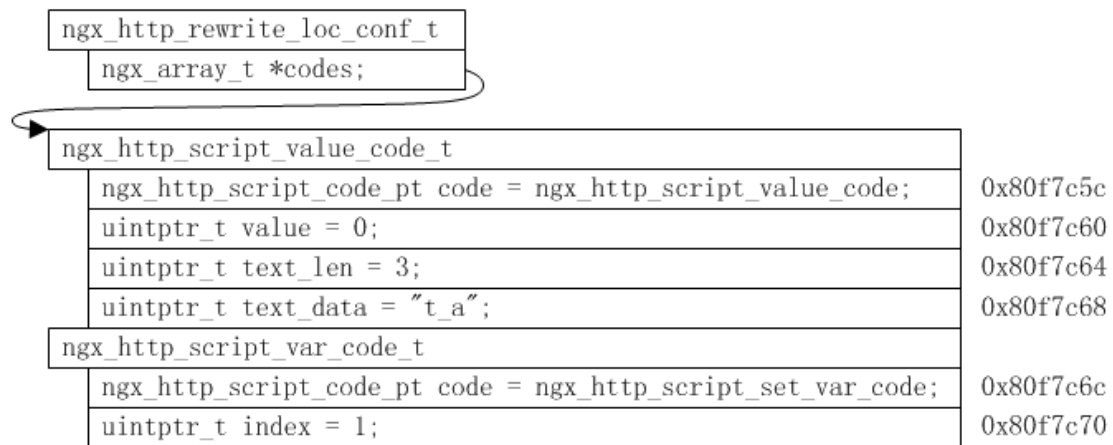
第二个重点关注的逻辑在函数 ngx\_http\_rewrite\_set()内，其继续保存 ngx\_http\_script\_xx\_code\_t 类结构体变量：

```
891: Filename : ngx_http_rewrite_module.c
892: static char *
893: ngx_http_rewrite_set(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
894: {
895: ...
933:     if (ngx_http_rewrite_value(cf, lcf, &value[2]) != NGX_CONF_OK) {
934: ...
951:     vcode = ngx_http_script_start_code(cf->pool, &lcf->codes,
952:         sizeof(ngx_http_script_var_code_t));
953: ...
957:     vcode->code = ngx_http_script_set_var_code;
958:     vcode->index = (uintptr_t) index;
```

逻辑很简单，利用函数 ngx\_http\_script\_start\_code()为 ngx\_http\_script\_var\_code\_t 类型变量 vcode 申请存储空间，然后保存回调函数以及相关数据。

上面具体代码执行路径被我略去了，总之，结果就是如下图示这样，nginx 创建了两个

结构体变量，并且设置好了字段值：



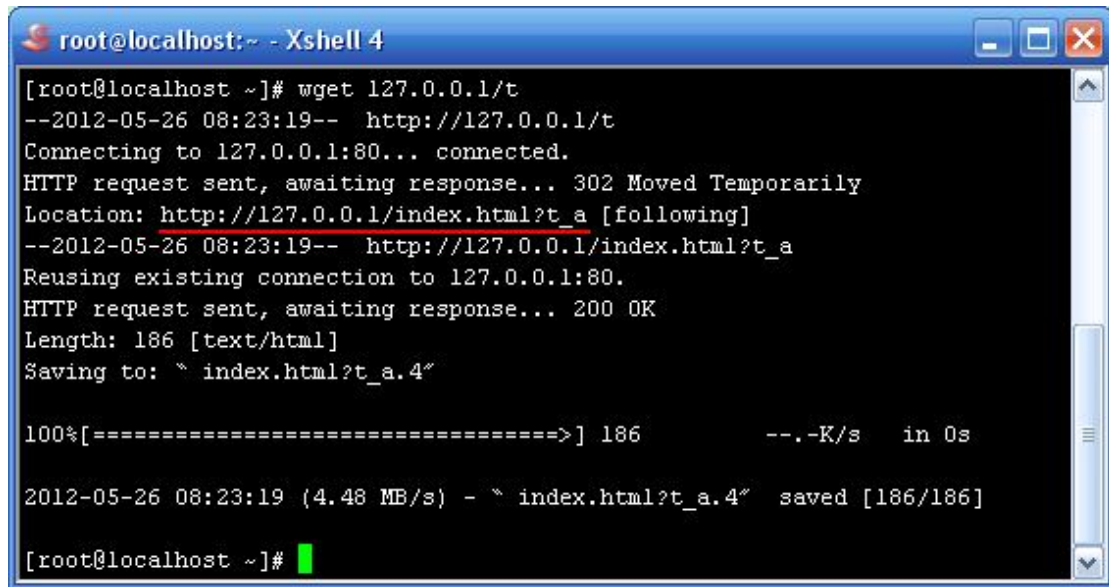
可以看到这两个结构体变量在地址空间上是连续存储的（图中，我特意把每个结构体字段的地址给标了出来），这一点非常重要，因为在脚本引擎实际执行时，回调函数前后的依次调用就靠这个来保证。到这里，关于配置项目 `set $file t_a`；而言，整个 `set` 指令就已完成了它原本的功能，对应的回调函数 `ngx_http_rewrite_set()`构建了这么一个脚本引擎的基础结构（每一个结构体变量代表脚本引擎的一个步骤），但这个脚本引擎还没‘跑’起来。要让这个脚本引擎跑起来，我们把这个配置项目放到配置文件的某一个 `location` 下，然后去请求这个 `location`，此时 `nginx` 就会要执行这个配置语句，对应的脚本引擎自然也就‘跑’起来了。

为了判断脚本引擎‘跑’起来后的效果，我们需要查看变量 `$file` 的值，这可以借助互联网上提供的第三方开源模块，比如 `echo` 模块 (<http://wiki.nginx.org/HttpEchoModule>)，不过我们这里可以灵活利用一下 `rewrite` 指令即可，在配置文件里设定如下配置项：

```

13: Filename : nginx.conf
14:         location / {
15:             root web;
16:         }
17:         location /t {
18:             set $file t_a;
19:             rewrite ^(.*)$ /index.html?$file redirect;
20:             root html;
21:         }
    
```

这样，任何对 `t` 目录的访问都被无条件的重定向到根目录，并且将变量 `$file` 的内容（这里也就是“`t_a`”）以参数的形式带过去。由于 `redirect` 指令会以 `http` 状态码 `302` 来指示浏览器重新请求新的 `URI`，因此我们能在浏览器地址栏里间接的看到 `$file` 的值，比如 `wget` 看到的情况：



```

root@localhost:~ - Xshell 4
[root@localhost ~]# wget 127.0.0.1/t
--2012-05-26 08:23:19-- http://127.0.0.1/t
Connecting to 127.0.0.1:80... connected.
HTTP request sent, awaiting response... 302 Moved Temporarily
Location: http://127.0.0.1/index.html?t_a [following]
--2012-05-26 08:23:19-- http://127.0.0.1/index.html?t_a
Reusing existing connection to 127.0.0.1:80.
HTTP request sent, awaiting response... 200 OK
Length: 186 [text/html]
Saving to: `index.html?t_a.4'

100%[=====>] 186          --.-K/s   in 0s

2012-05-26 08:23:19 (4.48 MB/s) - `index.html?t_a.4' saved [186/186]

[root@localhost ~]#

```

前面章节曾讲过，nginx 将对客户端的连接请求响应处理分成 11 个阶段，每一个阶段可以有零个或多个回调函数进行专门处理，而在这里，当客户端对/t 目录进行的请求访问时，nginx 执行到 NGX\_HTTP\_REWRITE\_PHASE 阶段的回调函数 ngx\_http\_rewrite\_handler()时，就会触发该 location 上脚本引擎的执行：

```

135: Filename : ngx_http_rewrite_module.c
136: static ngx_int_t
137: ngx_http_rewrite_handler(ngx_http_request_t *r)
138: {
139: ...
166:     e->sp = ngx_palloc(r->pool,
167:                         rlc->stack_size * sizeof(ngx_http_variable_value_t));
168: ...
172:     e->ip = rlc->codes->elts;
173: ...
178:     while (*(uintptr_t *) e->ip) {
179:         code = *(ngx_http_script_code_pt *) e->ip;
180:         code(e);
181:     }

```

脚本引擎的执行逻辑也非常的简单，因为刚提到脚本引擎各步骤在内存地址空间上连续，所以前一步骤的回调执行完后，指针偏移到下一步，然后判断是否有效，有效则接着执行，如此反复。由于每个步骤自身占据多大空间只有自己清楚，因此回调指针的偏移操作是由各个步骤来处理的，以这里的实例来看，第一个步骤对应的是结构体 ngx\_http\_script\_value\_code\_t 变量，回调函数为 ngx\_http\_script\_value\_code()：

```

1650: Filename : ngx_http_script.c
1651: void
1652: ngx_http_script_value_code(ngx_http_script_engine_t *e)
1653: {

```

```

1654:      ngx_http_script_value_code_t  *code;
1655:
1656:      code = (ngx_http_script_value_code_t *) e->ip;
1657:
1658:      e->ip += sizeof(ngx_http_script_value_code_t);
1659:
1660:      e->sp->len = code->text_len;
1661:      e->sp->data = (u_char *) code->text_data;
1662:      ...
1666:      e->sp++;
1667:  }

```

很容易看出来，上面代码中的第 1658 行就是做回调指针偏移操作，加上当前结构体 `ngx_http_script_value_code_t` 变量大小即可。另外，这也隐含的默认所有的 `ngx_http_script_xx_code_t` 结构体第一个字段必定为回调指针，如果我们添加自己的脚本引擎功能步骤，这点就需要注意。

第一步骤的回调函数 `ngx_http_script_value_code()` 处理完后，转到 `ngx_http_rewrite_handler()` 函数的第 178 行判断，为真，所以接着执行结构体 `ngx_http_script_var_code_t` 变量的回调函数 `ngx_http_script_set_var_code()`，同样做相应的偏移，再判断就会进入到 `rewrite` 指令所对应的处理步骤里。先不管后面步骤，只看与 `set` 指令相关的两个步骤，我们知道 `set` 指令是让 `nginx` 用户给变量赋值，这里“`set $file t_a;`”即是将字符串“`t_a`”赋值给变量 `$file`，所以这个逻辑也就是实现在刚才的那两个步骤里，具体来说是两个函数 `ngx_http_script_value_code()` 与 `ngx_http_script_set_var_code()`。

在继续分析之前，需要先提一个变量 `e->sp`，它是一个数组，在 `ngx_http_rewrite_handler()` 函数的第 166 行申请空间，就是通过它来在脚本引擎的各个步骤之间进行数据的传递。对于它的使用，有点类似于 C 语言函数调用栈帧，存入传递值就压栈，取传递值就退栈。比如看上面 `ngx_http_script_value_code()` 函数的实现代码，它是将用户设定的值（用户在配置文件里设定的字符串“`t_a`”以及长度在 `nginx` 解析配置文件时存在了 `ngx_http_script_value_code_t` 结构体变量的相关字段内）存起来，所以在第 1660、1661 以及 1666 行的代码，就是转存用户设定值并压栈（注意栈顶数据为空）。而函数 `ngx_http_script_set_var_code()` 就是取值退栈：

```

1669:  Filename : ngx_http_script.c
1670:  void
1671:  ngx_http_script_set_var_code(ngx_http_script_engine_t *e)
1672:  {
1673:      ...
1676:      code = (ngx_http_script_var_code_t *) e->ip;
1677:
1678:      e->ip += sizeof(ngx_http_script_var_code_t);
1679:      ...

```

```
1682:      e->sp--;
1683:
1684:      r->variables[code->index].len = e->sp->len;
1685:      ...
1688:      r->variables[code->index].data = e->sp->data;
```

变量 `code->index` 表示 nginx 变量 `$file` 在 `cmcf->variables` 数组内的下标，对应每个请求的变量值存储空间就为 `r->variables[code->index]`，这里从栈中取出数据并进行变量实际赋值。

基本过程就是，利用 `ngx_http_script_value_code()` 函数将 `"t_a"` 存储到临时空间 (`e->sp` 栈)，然后利用函数 `ngx_http_script_set_var_code()` 从临时空间 (`e->sp` 栈) 取值放到变量 `$file` 内，整个 `set` 指令的逻辑工作得以完成。

更复杂一点的 nginx 配置被解析后生成的脚本引擎及其执行，与上面的介绍并无特别大的差异，只是在脚本引擎的具体生成过程中可能会涉及到正则式的处理，比如：

```
# rewrite /download/*/mp3/*.any_ext to /download/*/mp3/*.mp3
rewrite ^/(download/.*)/mp3/(.*)\..*$ /$1/mp3/$2.mp3 break;
```

前面的 `"^/(download/.*)/mp3/(.*)\..*$"` 就是一个正则匹配，`^` 表示开头，`$` 表示结尾，(`download/.*`) 与 (`.*`) 分别对应后面的变量 `$1`，`$2`，像这个路径：`/download/20120805/mp3/sample.txt`，其对应的变量 `$1` 的值为 `download/20120805`，变量 `$2` 的值为 `sample`，所以 `rewrite` 后的路径为 `/download/20120805/mp3/sample.mp3`。关于这方面的更多内容不做过多介绍，对于复杂脚本引擎感兴趣的或遇到实际问题的，可自行查看 MAN 手册和 nginx 相关源代码，我相信有了前面介绍的基础知识，那不会太难理解，无非是细节代码繁琐一点。

## 执行顺序

关于 nginx 变量 (或者说是其所在的脚本引擎) 的执行顺序，这是一个值得关注的话题，因为不理解它的内在原理，就容易让人在 nginx 配置文件里实际使用变量时出现困惑；但对于 nginx 本身来说，这也是自然而然的事情，在前面的模块解析一章曾描述过 nginx 将对客户端请求的处理分成 11 个阶段，每一个阶段前后按序执行，那么与此对应的 nginx 变量也将受此影响，而出现貌似不合常理的异常情况。举个实例来说，假设在 nginx 配置文件里有这么一段配置 (这段配置在实际使用中毫无用处，这里仅作问题描述)：

```
49: Filename : nginx.conf
50:      location / {
51:          root    web;
52:          set $file index1.html;
53:          index $file;
54:      ...
55:          set $file index2.html;
56:      ...
```

第 52 行设置变量 `$file` 的值为 `index1.html`，第 53 行再通过 `index` 配置指令来指定根目录



的首页文件为变量\$file（也就是 index1.html），这是我们原本的意图。在接下来的配置里，变量\$file 的值又被修改作为它用，比如也许被修改为 logs/root\_access.log，然后用户 access\_log 配置指令来指定根目录的访问日志文件。这里为了作对比演示，我们就直接把它设置为 index2.html，并且 index1.html 和 index2.html 的文件内容也非常简单，分别为：

```
[root@localhost web]# cat index1.html
```

```
<center><h1>1</h1></center>
```

```
[root@localhost web]# cat index2.html
```

```
<center><h1>2</h1></center>
```

利用这个配置文件执行 nginx 后，通过 curl 命令来请求访问该根目录：

```
[root@localhost nginx-1.2.0]# ./objs/nginx -c /usr/local/nginx/conf/nginx.conf
[root@localhost nginx-1.2.0]# curl 127.0.0.1
<center><h1>2</h1></center>
[root@localhost nginx-1.2.0]#
```

奇怪的发现，nginx 返回的内容来之文件 index2.html，完全超出我们原本的设想，这不是 nginx 的 bug 呢？当然不是，其真实原因正是由于受到变量执行顺序的影响。

前面已经说过 nginx 对客户端的请求是分阶段处理的，配置文件里使用到的 nginx 变量会跟随处理阶段的向前推进而逐个被执行到，而与它在配置文件里的具体前后位置并没有关系（当然，必须都在本次会执行到的路径上）。由于在 nginx 启动阶段，通过对配置文件的逐行解析，会把属于同一阶段的变量集中在一起。如在上面的实例中，虽然两条 set 指令使用的 \$file 变量跨越了 index 指令使用的 \$file 变量，但在配置文件解析后，其效果变成了类似于这样：

REWRITE_PHASE	set \$file index1.html;
	set \$file index2.html;
...	
CONTENT_PHASE	index \$file;

当一个客户端请求过来时，在 REWRITE\_PHASE 阶段，将依次执行“set \$file index1.html;”、“set \$file index2.html;”，再到 CONTENT\_PHASE 阶段执行 ngx\_http\_index\_module 模块的逻辑时，\$file 变量的值已经是 index2.html，所以 nginx 返回给客户端的才是文件 index2.html 的内容。

上面给出的只是一个非常简单的例子，但是也较为清楚的说明了 nginx 变量的执行顺序及其内在原因。如果继续举例也没有太大必要，毕竟原理就这么简单，我们在实际进行 nginx 配置时，也就要特别注意配置文件里都使用了哪些 nginx 变量，每个 nginx 变量都使用在哪些配置指令里，避免出现受变量执行顺序的隐含影响，导致 nginx 工作不正常的情况。



## 第七章 请求处理

### 创建监听套接口

前面章节曾陆陆续续的提到过 nginx 对客户端请求的处理，但不甚连贯，所以本章就尝试把这个请求处理响应过程完整的描述一遍。下面先来看后 http 请求处理的前置准备工作，也就是监听套接口的创建以及组织等。

创建哪些监听套接口当然是由用户来指定的，nginx 提供的配置指令为 `listen`（仅关注 http 模块：<http://wiki.nginx.org/HttpCoreModule#listen>），该指令功能非常的丰富，不过在大部分情况下，我们都用得比较简单，一般是指定监听 ip 和端口号（因为 http 协议是基于 tcp，所以这里自然也就是 tcp 端口），比如：`listen 192.168.1.1:80;`，这表示 nginx 仅监听目的 ip 是 192.168.1.1 且端口是 80 的 http 请求；如果主机上还有一个 192.168.1.2 的 ip 地址，那么客户端对该地址的 80 端口访问将被拒绝，要让该地址也正常访问需同样把该 ip 加入：`listen 192.168.1.2:80;`，如果有更多 ip，这样逐个加入比较麻烦，因而另一种更偷懒的配置方法是只指定端口号：`listen 80;`，那么此时任意目的 ip 都可以访问到。不过，这两种不同的配置方式会影响到 nginx 创建监听套接口的数目，前一种方式 nginx 会对应的创建多个监听套接口，而后一种方式，由于 `listen 80;` 包含了所有的目标 ip，所以创建一个监听套接口就足以，即便是配置文件里还有 `listen 192.168.1.1:80;` 这样的配置。看看实例，感性的认识一下：

```
30: Filename : nginx.conf
15:     server {
16:         listen      80;
17:     ...
34:     server {
35:         listen      192.168.1.1:80;
36:
```

上面配置中有两个 `server`，第一个配置 `listen` 的目标 ip 为任意（只要是本主机有的），第二个配置 `listen` 的目标 ip 为 192.168.1.1，但是 nginx 在创建监听套接口时却只创建了一个：

```
[root@localhost html]# netstat -ntap | grep nginx
tcp    0    0    0.0.0.0:80    0.0.0.0:*    LISTEN    13040/nginx
```

如果将第 16 行配置改为 `listen 192.168.1.2:80;`，那么此时的 nginx 将创建两个监听套接口：

```
[root@localhost nginx]# netstat -ntpa | grep nginx
tcp    0    0    192.168.1.1:80    0.0.0.0:*    LISTEN    13145/nginx
tcp    0    0    192.168.1.2:80    0.0.0.0:*    LISTEN    13145/nginx
```

再来看 nginx 代码的具体实现，配置指令 `listen` 的使用上下文为 `server`，其对应的处理函数为 `ngx_http_core_listen()`，该函数本身的功能比较单一，主要是解析 `listen` 指令并将对应的结果存到变量 `lsopt`（可能有人注意到这是一个局部变量，不过没关系，在后面的函数调

用里通过结构体赋值的方式，将它的值全部复制给另外一个变量 `addr->opt` 内，最后调用函数 `ngx_http_add_listen()`，这才是此处关注的核心函数，它将所有的 `listen` 配置以 `[port,addr]` 的形式组织在 `http` 核心配置 `ngx_http_core_main_conf_t` 下的 `ports` 数组字段内。另外，如果有一个 `server` 内没有配置监听端口，那么 `nginx` 会自动创建一个变量 `lsopt` 并给出一些默认值，然后调用函数 `ngx_http_add_listen()` 将其组织到 `ports` 数组字段内，这在 `server` 块配置的回调函数 `ngx_http_core_server()` 最后可以看到：

```

2700:   Filename : ngx_http_core_module.c
2701:   static char *
2702:   ngx_http_core_server(ngx_conf_t *cf, ngx_command_t *cmd, void *dummy)
2703:   {
2704:   ...
2790:       if (rv == NGX_CONF_OK && !cscf->listen) {
2791:           ngx_memzero(&lsopt, sizeof(ngx_http_listen_opt_t));
2792:       ...
2795:           sin->sin_family = AF_INET;
2796:       ...
2799:           sin->sin_port = htons((getuid() == 0) ? 80 : 8000);
2800:   #endif
2801:           sin->sin_addr.s_addr = INADDR_ANY;
2802:       ...
2816:           if (ngx_http_add_listen(cf, cscf, &lsopt) != NGX_OK) {
2817:       ...

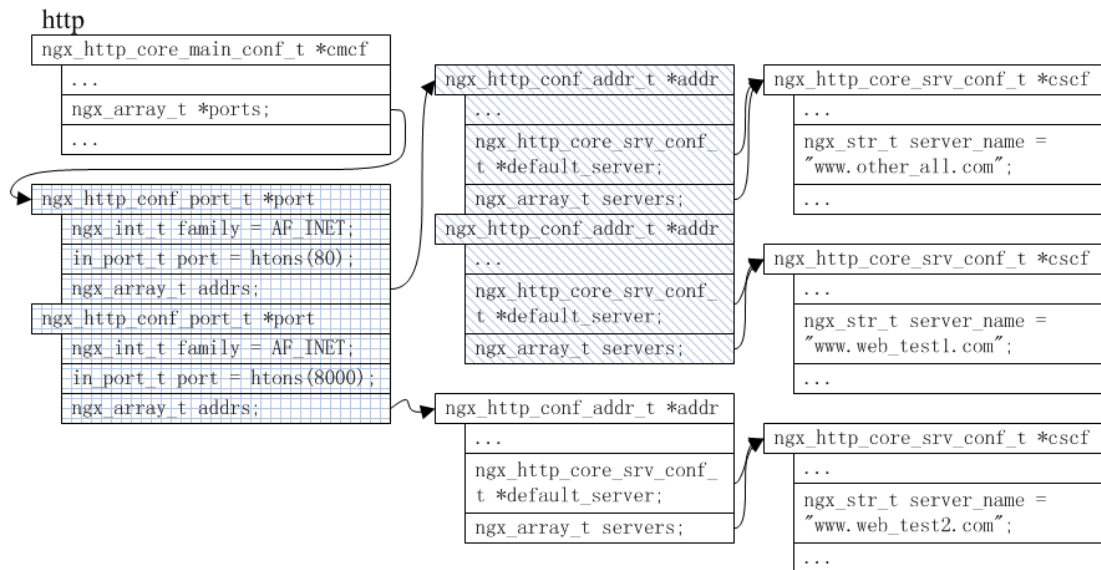
```

默认的设置是 `ipv4` 协议族、`80` 或 `8000` 端口、任意目的 `ip`，所以不管怎么样，一个 `server` 配置至少有一个监听套接口。回过头来看 `ngx_http_add_listen()` 函数以及相关逻辑，具体代码并没有什么难以理解的地方，我们直接看一个实例以及对应的图示，这样更直观且能把握全局。仍接着前面的实例，再加一个 `server` 配置：

```

00:   Filename : nginx.conf
15:       server {
16:           listen      80;
17:           server_name  www.other_all.com;
18:   ...
34:       server {
35:           listen      192.168.1.1:80;
36:           server_name  www.web_test1.com;
37:   ...
53:       server {
54:           listen      192.168.1.2:8000;
55:           server_name  www.web_test2.com;
56:   ...

```



当 nginx 的 http 配置块全部解析完后，所有的监听套接口信息（包括用户主动 listen 配置或 nginx 默认添加）都已被收集起来，先按 port 端口分类形成数组存储在 cmcf->ports 内，然后再在每一个 port 内按 ip 地址分类形成数组存储在 port->addrs 内，也就是一个[port, addr]的二维划分，如上图所示。附带说一下，其实一个[port, addr]可以对应多个 server 配置块，但这里的实例中 server 配置块只有一个，所以也就是默认配置块 default\_server；对应 server 配置块的多少并不会影响到监听套接口的创建逻辑，因为创建监听套接口依赖的是[port, addr]本身，而非它对应的 server 配置块。回到刚才的思路，在 http 配置指令的回调函数 ngx\_http\_block()最后，也就是 http 配置块全部解析完后，将调用 ngx\_http\_optimize\_servers()函数‘创建’对应的监听套接口，之所以打上引号是因为这里还只是名义上的创建，也就是创建了每个监听套接口所对应的结构体变量 ngx\_listening\_s，并以数组的形式组织在全局变量 cycle->listening 内，具体的函数调用关系如下：

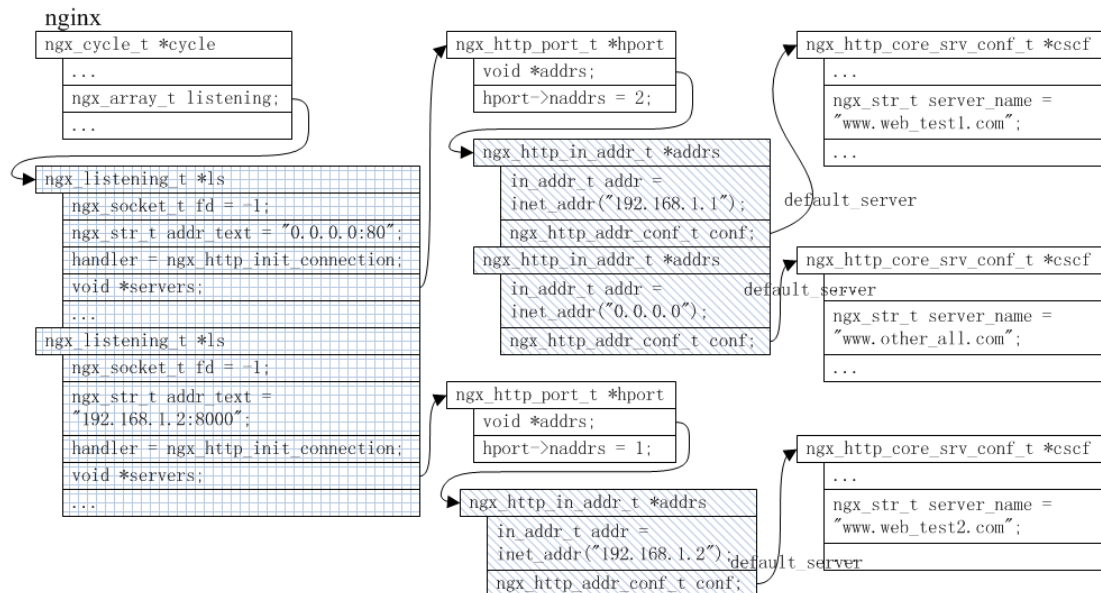
ngx\_http\_optimize\_servers() -> ngx\_http\_init\_listening() -> ngx\_http\_add\_listening() -> ngx\_create\_listening()

先关注两点：第一，如果某端口上有任意目的 ip 的 listen 配置，那么在该端口上只会创建一个结构体变量 ngx\_listening\_s，不管是否还有其他 ip 在该端口上的 listen 配置，进而在后面创建监听套接口描述符时也只创建一个，这在前面的演示实例里验证过这种情况，相关逻辑代码实现在函数 ngx\_http\_init\_listening()内，比如局部变量 bind\_wildcard，也包括前面函数中对 ip 地址排序的准备工作等。第二，ngx\_listening\_s 结构体变量 ls 的回调字段 handler 被设置为 ngx\_http\_init\_connection，注意到这点可以帮助我们在看后面的逻辑时，代码回调该函数时，就能清楚它实际执行的是哪个函数。

另外要关注的是监听套接口与 server 配置块的关联，这是必须的，因为当监听套接口上一个客户端请求到达时，nginx 必须知道它对应的 server 配置才能做进一步处理，这部分逻辑主要在函数 ngx\_http\_add\_addrs()内（以 ipv4 为例），调用关系如下：

ngx\_http\_optimize\_servers() -> ngx\_http\_init\_listening() -> ngx\_http\_add\_addrs()

到此, 在上面所举示例里, 所有等待创建的监听套接口以及相关数据组织结构如下所示 ("www.web\_test1.com"排到了"www.other\_all.com"的前面, 是因为 ngx\_sort()排序的缘故):



在所有配置解析完并且做了一些其它初始化工作后, 就开始真正的监听套接口描述符创建以及特性设置, 也就是调用诸如 socket()、setsockopt()、bind()、listen()等这样的系统函数, 相关逻辑实现在函数 ngx\_open\_listening\_sockets()和 ngx\_configure\_listening\_sockets()内, 而这两个函数在 nginx 的初始化函数 ngx\_init\_cycle()内靠结尾处被调用。从函数 ngx\_open\_listening\_sockets()内代码实现, 可以看到就是遍历 cycle->listening 数组内每一个 ls 元素进行逐个创建, 而 ngx\_configure\_listening\_sockets()内的描述符特性设置也是如此:

```

267: Filename : ngx_connection.c
268: ngx_int_t
269: ngx_open_listening_sockets(ngx_cycle_t *cycle)
270: {
271: ...
292:     ls = cycle->listening.elts;
293:     for (i = 0; i < cycle->listening.nelts; i++) {
294:         ...
312:         s = ngx_socket(ls[i].sockaddr->sa_family, ls[i].type, 0);
313: ...

```

在这两个函数执行完之后, cycle->listening 数组的每一个 ls 元素, 其 fd 就不再是-1, 而是一个可用的监听套接口描述符, 并且该描述符根据用户设置赋予了不同的特性, 比如收包缓存区大小, 发包缓存区大小等。

关于监听套接口, nginx 主进程的工作就做完了, 接下来主进程通过 fork()创建子进程, 也就是工作进程, 它们将全部继承这些已初始化好的监听套接口。在每个工作进程的事件初始化函数 ngx\_event\_process\_init()内, 对每一个监听套接口创建对应的 connection 连接对象

(为什么不直接用一个 `event` 事件对象呢? 主要是考虑到可以传递更多信息到函数 `ngx_event_accept()` 内, 并且这个连接对象虽然并没有对应的客户端, 但可以与 `accept()` 创建的连接套接口统一起来, 因为连接套接口对应的是 `connection` 连接对象), 并利用该 `connection` 的 `read` 事件对象 (因为在监听套接口上触发的肯定是读事件):

```
582: Filename : ngx_event.c
583: static ngx_int_t
584: ngx_event_process_init(ngx_cycle_t *cycle)
585: {
586: ...
745:     ls = cycle->listening.elts;
746:     for (i = 0; i < cycle->listening.nelts; i++) {
747:
748:         c = ngx_get_connection(ls[i].fd, cycle->log);
749:         ...
759:         rev = c->read;
760:         ...
762:         rev->accept = 1;
763:         ...
826:         rev->handler = ngx_event_accept;
```

`read` 事件对象的回调处理函数为 `ngx_event_accept()`, 请记住它。一切都准备就绪, 接下来就是对 `rev` 事件对象进行监控, 即将监听套接口所对应的事件对象加入到 `nginx` 的事件处理模型里, 那什么时候加入呢? 如果没有启动 `accept_mutex`, 那么在函数 `ngx_event_process_init()` 末尾就会通过 `ngx_add_event()` 将它加入到事件监控机制内:

```
837: Filename : ngx_event.c
838:         if (ngx_add_event(rev, NGX_READ_EVENT, 0) == NGX_ERROR) {
```

否则就需在核心执行函数 `ngx_process_events_and_timers()` 内进行竞争, 抢占到 `accept_mutex` 锁后才会把它加入:

```
353: Filename : ngx_event_accept.c
354:         if (ngx_add_event(c->read, NGX_READ_EVENT, 0) == NGX_ERROR) {
```

在事件机制一章的负载均衡一节有对这两方面的详细描述, 而在这里我们需特别注意的是, 这些事件对象是以水平触发的方式加入到事件监控机制内的, 这意味着一个进程一次没有处理完的客户端连接请求可以再次被捕获。

举个例子, 我们知道在默认情况下, `nginx` 一次只 `accept()` 一个请求 (即 `multi_accept` 为 `off`), 如果某个监听套接口 `A` 上同时来了两个客户端请求连接, 触发可读事件被工作进程捕获, 工作进程 `accept()` 处理了一个请求后, 重新阻塞在事件机制监控处 (比如 `epoll_wait()`), 但事实上, 监听套接口 `A` 上还有一个客户端请求连接没有被处理, 如果监听套接口 `A` 不是以水平触发而是以边缘触发加入到事件监控机制, 此时监听套接口 `A` 虽然可读却无法触发可读事件而让 `epoll_wait()` 返回, 除非监听套接口 `A` 上又来了新的请求重新触发可读事件, 但这无疑会导致连接请求得不到及时处理并逐渐累积, 到最后该监听套接口 `A` 彻底失效。



理解这个例子需要对 `epoll` 事件模型的水平触发 LT 与边缘触发 ET 两种模式特性有一定的了解，但这里我也不多详叙，不清楚的请翻阅 `man` 手册或 `Google`。

另外，可以看到 `nginx` 主进程在创建完工作进程后并没有关闭这些监听套接口，但主进程却又并没有进行 `accept()` 客户端请求连接，那么是否会导致一些客户端请求失败呢？答案当然是否定的，虽然主进程也拥有那些监听套接口，并且它也能收到客户端的请求，但是主进程并没有监控这些监听套接口上的事件，没有去读取客户端的请求数据。既然主进程没有去读监听套接口上的数据，那么数据就阻塞在那里，等待任意一个工作进程捕获到对应的可读事件后，进而就可以去处理并响应客户端请求。至于主进程为什么要保留（不关闭）那些监听套接口，是因为再后续再创建新工作进程（比如某工作进程异常退出，主进程收到 `SIGCHLD` 信号）时，还要把这些监听套接口传承过去。

## 创建连接套接口

当有客户端发起请求连接，监控监听套接口的事件管理机制就会捕获到可读事件，工作进程便执行对应的回调函数 `ngx_event_accept()`，从而开始连接套接口的创建工作。

函数 `ngx_event_accept()` 的整体逻辑都比较简单，但是有两个需要解析的处理。首先是每次处理调用 `accept()` 的次数，默认情况下也就是调用 `accept()` 一次，即工作进程每次捕获到监听套接口上的可读事件后，只接受一个服务请求，如果同时收到多个客户端请求，那么除第一个以外的请求需等到再一次触发事件才能被 `accept()` 接受。但是，如果用户配置有 `multi_accept on;`，那么工作进程每次捕获到监听套接口上的可读事件后，将反复 `accept()`，一次接受所有的客户端服务请求。这样看起来，似乎“一次接受所有的客户端服务请求”更高效，可为什么它却不是默认配置呢？

举个例子就懂了，假设两个工作进程 A 和 B 相互争用监听套接口并对其上的客户端服务请求进行处理，假定两者争用成功的概率都为 50%，但是进程 A 的运气有点差，亦或者说运气有点好，反正不管怎么说，每次进程 A 对监听套接口争用成功时，总是同时有很多个客户端请求到达，而进程 B 却每次只有少数的几个请求，这样几个循环下来，进程 A 就非常繁忙了，工作进程之间的负载没有得到均衡，所以默认情况下，工作进程一次只 `accept()` 一个客户端服务请求。相关的逻辑代码如下所示（系统调用 `accept4()` 与 `accept()` 差别不大，请查 `man` 手册，下面代码以使用 `accept()` 为例）：

```
17: Filename : ngx_event_accept.c
18: void
19: ngx_event_accept(ngx_event_t *ev)
20: {
21: ...
40:     ev->available = ecf->multi_accept;
41: ...
```

```

50:      do {
51:  ...
61:          s = accept(lc->fd, (struct sockaddr *) sa, &socklen);
62:  ...
64:          if (s == -1) {
65:  ...
70:              return;
71:  ...
290:      } while (ev->available);

```

当配置文件中有 `multi_accept on` 时, 对应的解析值 `ecf->multi_accept` 为 1, 从而 `ev->available` 值为 1, 所以这个 `do{}while` 是一个死循环, 直到 `accept()` 接受不到客户端请求时, 即返回值 `s` 等于 -1 时, 循环才得以退出。在未配置 `multi_accept` 或 `multi_accept` 为 off 的情况下, `ev->available` 值为 0, 此时循环主体自然也就只执行一次。

函数 `accept()` 调用成功接受客户端请求后, 就通过函数 `ngx_get_connection()` 申请对应的连接对象, 做一些初始赋值等, 简单明了而无需多说, 但有一个需要解析的处理是 `deferred_accept`, 相关代码如下:

```

204: Filename : ngx_event_accept.c
205:      if (ev->deferred_accept) {
206:          rev->ready = 1;
207:  ...
284:      ls->handler(c);

```

`ev->deferred_accept` 值的最初影响设置是 `listen` 配置的附属项目里, 前面曾讲过 `listen` 配置项非常的复杂, 有大量的附属项目提供用户来指定这个监听套接口的相关属性, 而 `deferred` 就是其中的一个, 带有该附属项目的对应监听套接口描述符会被设置 `TCP_DEFER_ACCEPT` 特性, 并且对应到这里的 `ev->deferred_accept` 值为 1 (前后是怎样的转换与逐步赋值略过不讲, 翻下代码很容易理解)。 `TCP_DEFER_ACCEPT` 特性意味当工作进程 `accept()` 这个监听套接口上的客户端请求时, 请求的数据内容已经到达了, 所以这里第 206 行将 `rev->ready` 设置为 1, 表示数据准备就绪。最后执行的 `ls->handler` 回调也就是函数 `ngx_http_init_connection()`, 这是在很早之前赋值 (还记得么? 上一节提到过) 上的。

```

181: Filename : ngx_http_request.c
182: void
183: ngx_http_init_connection(ngx_connection_t *c)
184: {
185:  ...
206:      rev->handler = ngx_http_init_request;
207:  ...
213:      if (rev->ready) {
214:          /* the deferred accept(), rtsig, aio, iocp */
215:
216:          if (ngx_use_accept_mutex) {

```



```

217:         ngx_post_event(rev, &ngx_posted_events);
218:         return;
219:     }
220:
221:     ngx_http_init_request(rev);
222:     return;
223: }
224:
225:     ngx_add_timer(rev, c->listening->post_accept_timeout);
226:
227:     if (ngx_handle_read_event(rev, 0) != NGX_OK) {
228: ...

```

函数 `ngx_http_init_connection()` 很简单，但注意到 `rev->ready`，如果它为 0，则将事件对象 `rev` 加入到超时管理机制和事件监控机制，等超时或请求数据到达。如果 `rev->ready` 为 1，也就是监听套接口描述符使用刚才讲到的 `TCP_DEFER_ACCEPT` 特性，`accept()` 接受服务请求后，请求数据已经准备好了，当然是开始着手处理。第 216 行的 `if` 判断为真则意味着有加锁，所以先把该事件对象加到 `ngx_posted_events` 链表，返回解锁后再进行处理，否则在第 221 行就开始处理，这部分逻辑结合事件机制一章的描述应该容易理解，不过需注意在 `rev->ready` 为 1 的处理情况下，到此时为止，我们新建连接对象都还没有被加入到事件监控机制里，因为当前我们是知道有数据可读，如果运气好，需要的所有请求数据都已经全部到达了，读取数据处理请求然后响应即可，就没有必要把连接对象加到事件监控机制里，却什么作用都没起到又把它从事件监控机制里删除；只有当进行数据读取时，发现所需要的请求数据没有全部到达，此时才将连接对象加到事件监控机制里，等待进一步数据到达时以便获得事件通知，所以在后面的 `ngx_http_read_request_header()` 类似函数内能看到 `ngx_handle_read_event()` 这样的函数调用：

```

1139:     Filename : ngx_http_request.c
1140:     static ssize_t
1141:     ngx_http_read_request_header(ngx_http_request_t *r)
1142:     {
1143:     ...
1144:         if (n == NGX_AGAIN) {
1145:         ...
1146:             ngx_add_timer(rev, cscf->client_header_timeout);
1147:         ...
1148:         if (ngx_handle_read_event(rev, 0) != NGX_OK) {

```

如上所示，在读到 `NGX_AGAIN` 时，也就是需要的数据没有全部到达，于是将事件对象 `rev` 加入到超时管理机制和事件监控机制，以等待后续数据可读事件或超时。

## HTTP 请求处理

函数 `ngx_http_init_request()`，正式开始对一个客户端服务请求进行处理与响应工作。该函数的主要工作仍然只是做处理准备：建立 `http` 连接对象 `ngx_http_connection_t`、`http` 请求对象 `ngx_http_request_t`、找到对应的 `server` 配置 `default_server`、大量的初始化赋值操作，最后执行回调函数 `ngx_http_process_request_line()`，进入到 `http` 请求头的处理中：

```

236: Filename : ngx_http_request.c
237: static void
238: ngx_http_init_request(ngx_event_t *rev)
239: {
240: ...
276:         hc = ngx_palloc(c->pool, sizeof(ngx_http_connection_t));
277: ...
295:         r = ngx_palloc(c->pool, sizeof(ngx_http_request_t));
296: ...
380:         addr = port->addr;
381:         addr_conf = &addr[0].conf;
382: ...
388:         /* the default server configuration for the address:port */
389:         cscf = addr_conf->default_server;
390: ...
395:         rev->handler = ngx_http_process_request_line;
396: ...
519:         rev->handler(rev);
520: }
```

查找该请求该被端口上的哪个 `ip` 地址处理是通过对比目的 `ip` 地址来进行的(前面讲过，如果某端口上设置有任何 `ip` 监听，比如 `*:80`，那么即便还有其他指定 `ip` 的监听，比如 `192.168.1.1:80`，也只会创建一个监听套接口，所以对于该套接口上接收到的连接请求首先要进行目的 `ip` 匹配)，这部分代码很容易理解，就是一个 `for` 循环遍历查找，要注意的是由于 `ipv4` 的地址只有 32 位可以直接比较，但 `ipv6` 的地址有 128 位，所以需采用 `memcmp()` 比较，上面没有显示这部分代码，给出的第 380、381 行代码是监听套接口上只有一个目的 `ip` 的情况，此时直接使用它，并且第 389 行取用该地址上的默认 `server` 配置。如果客户端请求对应的 `server` 不是这个默认的会怎么样？不用当心这种情况，因为在后面还会有处理，比如 `ngx_http_find_virtual_server()` 函数，具体请参见下一章内容。

在继续下面的内容讲解前，有必要先介绍一下 `HTTP` 协议，当然，关于 `HTTP` 协议方面的内容，如果展开来说一时半会说不完，也有专门的书籍，比如《O'Reilly - HTTP Pocket Reference》、《O'Reilly - HTTP The Definitive Guide》、《Sams - HTTP Developers Handbook》等，所以这里仅以 RFC 2616（对应 `HTTP 1.1`：<http://www.ietf.org/rfc/rfc2616.txt>）为依据简单介绍一下 `HTTP` 请求响应数据的格式。

根据 RFC 2616 内容可知, HTTP 请求消息 (也包括响应消息。消息, 即 `message`, 可简单认为就是上面提到的请求响应数据的学术名称, 我说数据是笼统说法, 请勿拘泥这些名词概念, 毕竟我不是在写学术论文, ☺) 是利用 RFC 822 (<http://www.ietf.org/rfc/rfc822.txt>) 定义的常用消息格式来传输实体 (消息的负载, 即真正有价值的数据)。这种常用消息格式就是由开始行 (`start-line`), 零个或多个头域 (经常被称作 “头”)、一个指示头域结束的空行 (一个仅包含 CRLF 的 “空” 行) 以及一个可有可无的消息主体 (`message-body`)。当然, RFC 2616 文档里对 HTTP 请求响应消息格式描述得更具体一点, 其中请求消息格式的 BNF (巴科斯诺尔范式) 表示如下:

```
Request          = Request-Line          ; Section 5.1
                  *(( general-header      ; Section 4.5
                    | request-header      ; Section 5.3
                    | entity-header ) CRLF) ; Section 7.1
                  CRLF
                  [ message-body ]       ; Section 4.3
Request-Line     = Method SP Request-URI SP HTTP-Version CRLF
Method           = "OPTIONS"             ; Section 9.2
                  | "GET"                 ; Section 9.3
                  | "HEAD"                ; Section 9.4
                  | "POST"                ; Section 9.5
                  | "PUT"                 ; Section 9.6
                  | "DELETE"              ; Section 9.7
                  | "TRACE"               ; Section 9.8
                  | "CONNECT"             ; Section 9.9
                  | extension-method
extension-method  = token
Request-URI      = "*" | absoluteURI | abs_path | authority
```

可以看到, 工作进程收到的客户端请求头部数据以 Request-Line 开始 (GET / HTTP/1.0\r\n), 接着是不定数的请求头部 (User-Agent: Wget/1.12 (linux-gnu)\r\nAccept: \*/\*\r\nHost: www.web\_test2.com\r\nConnection: Keep-Alive\r\n), 最后以一个空行结束 (\r\n)。

而函数 `ngx_http_process_request_line()` 处理的数据就是客户端发送过来的 http 请求头中的 Request-Line, 这个过程可分为三步: 读取 Request-Line 数据、解析 Request-Line、存储解析结果并设置相关值。当然, 这个过程实际执行时可能重复多次 (比如 Request-Line 数据分多次到达监听套接口), 所以函数内的实现是一个 `for ( ;; )` 循环。下面逐一简单来看下各个步骤:

第一步, 读取 Request-Line 数据。通过函数 `ngx_http_read_request_header()` 将数据读到缓存区 `r->header_in` 内。比如执行 `wget www.web\_test2.com` 请求时, 调试 `nginx` 对应工作进程打印的数据如下:

```
(gdb) p r->header_in->pos
```

```
$8 = (u_char *) 0x98bdef8 "GET / HTTP/1.0\r\nUser-Agent: Wget/1.12 (linux-gnu)\r\n
Accept: */*\r\nHost: www.web_test2.com\r\nConnection: Keep-Alive\r\n\r\n"
```

一次就把整个请求头部数据读到了，当然也就包括完整的 Request-Line 数据（即：GET / HTTP/1.0\r\n）。另外说一句，可以看到命令 wget 默认是以 HTTP 1.0 协议发送请求，不过不影响（下面仍以它的数据为例），nginx 也支持 HTTP 1.0 协议，也可以用 curl 命令 [curl www.web\\_test2.com](http://www.web_test2.com) 进行请求：

```
(gdb) p r->header_in->pos
```

```
$9 = (u_char *) 0x98bdef8 "GET / HTTP/1.1\r\nUser-Agent: curl/7.19.7 (i686-pc-linux
-gnu) libcurl/7.19.7 NSS/3.12.7.0 zlib/1.2.3 libidn/1.18 libssh2/1.2.2\r\nHost: www.web_test2.
com\r\nAccept: */*\r\n\r\n"
```

刚才提到，由于客户端请求头部数据可能分多次到达，所以缓存区 r->header\_in 内可能还有一些上一次没解析完的头部数据，所以会存在数据的移动等操作，不过也都比较简单，仅提一下而略过不讲。

第二步，解析 Request-Line。对读取到的 Request-Line 数据进行解析的工作实现在函数 ngx\_http\_parse\_request\_line() 内。由于 Request-Line 数据有严格的 BNF 对应，所以其解析过程虽然繁琐，但并无不好理解的地方。

第三步，存储解析结果并设置相关值。在 Request-Line 的解析过程中会有一些赋值操作，但更多的是在成功解析后，ngx\_http\_request\_t 对象 r 内的相关字段值都将被设置，比如 uri (/)、method\_name (GET)、http\_protocol (HTTP/1.0) 等。

Request-Line 解析成功，即函数 ngx\_http\_parse\_request\_line() 返回 NGX\_OK，意味着这初步算是一个合法的 http 请求，接下来就开始解析其它请求头（general-header、request-header、entity-header）：

```
706: Filename : ngx_http_request.c
707: static void
708: ngx_http_process_request_line(ngx_event_t *rev)
709: {
710: ...
732:     for ( ;; ) {
733: ...
735:         n = ngx_http_read_request_header(r);
736: ...
742:         rc = ngx_http_parse_request_line(r, r->header_in);
743:
744:         if (rc == NGX_OK) {
745: ...
796:             if (ngx_list_init(&r->headers_in.headers, r->pool, 20,
797:                             sizeof(ngx_table_elt_t))
798: ...
```

```
915:         rev->handler = ngx_http_process_request_headers;
916:         ngx_http_process_request_headers(rev);
```

函数 `ngx_http_process_request_headers()` 对每一个请求头的处理步骤与函数 `ngx_http_process_request_line()` 处理 Request-Line 的情况类似，也是分为三步：读取数据（对应函数 `ngx_http_read_request_header()`），如果数据已经从监听套接口描述符读到缓存区了，那么无需再读）、解析数据（对应函数 `ngx_http_parse_header_line()`）、存储解析结果。

在第二步骤中，函数 `ngx_http_parse_header_line()` 解析的每一个请求头都会放到 `r->headers_in.headers` 内，看看 gdb 断点后捕获到的实例数据：

```
(gdb) p r->headers_in.headers.part
$34 = {elts = 0x98be5d4, nelts = 4, next = 0x0}
(gdb) p *(ngx_table_elt_t *)r->headers_in.headers.part.elts
$35 = {hash = 486342275, key = {len = 10, data = 0x98bdf08 "User-Agent"}, value = {len = 21, data = 0x98bdf14 "Wget/1.12 (linux-gnu)"}, ...}
(gdb) p *(ngx_table_elt_t *)r->headers_in.headers.part.elts + sizeof(ngx_table_elt_t) *
1)
$36 = {hash = 2871506184, key = {len = 6, data = 0x98bdf2b "Accept"}, value = {len = 3, data = 0x98bdf33 "*/"}, ...}
(gdb) p *(ngx_table_elt_t *)r->headers_in.headers.part.elts + sizeof(ngx_table_elt_t) *
2)
$37 = {hash = 3208616, key = {len = 4, data = 0x98bdf38 "Host"}, value = {len = 17, data = 0x98bdf3e "www.web_test2.com"}, ...}
(gdb) p *(ngx_table_elt_t *)r->headers_in.headers.part.elts + sizeof(ngx_table_elt_t) *
3)
$38 = {hash = 3519315678, key = {len = 10, data = 0x98bdf51 "Connection"}, value = {len = 10, data = 0x98bdf5d "Keep-Alive"}, ...}
```

如果请求头有对应的回调处理函数还会被做进一步处理，所有可以被 nginx 识别并处理的请求头定义在数组 `ngx_http_headers_in` 内，比如：

```
80: Filename : ngx_http_request.c
81: ngx_http_header_t ngx_http_headers_in[] = {
82:     { ngx_string("Host"), offsetof(ngx_http_headers_in_t, host),
83:       ngx_http_process_host },
84:
85:     { ngx_string("Connection"), offsetof(ngx_http_headers_in_t, connection),
86:       ngx_http_process_connection },
87: ...
```

而在函数 `ngx_http_process_request_headers()` 内的具体实现如下：

```
955: Filename : ngx_http_request.c
956: static void
957: ngx_http_process_request_headers(ngx_event_t *rev)
958: {
959: ...
1038:         rc = ngx_http_parse_header_line(r, r->header_in,
```

```

1039:                                     cscf->underscores_in_headers);
1040:
1041:             if (rc == NGX_OK) {
1042:                 ..
1056:                 h = ngx_list_push(&r->headers_in.headers);
1057:                 ...
1085:                 hh = ngx_hash_find(&cmcf->headers_in_hash, h->hash,
1086:                                     h->lowercase_key, h->key.len);
1087:                 ...
1088:                 if (hh && hh->handler(r, h, hh->offset) != NGX_OK) {
1089:                     ...

```

第 1041 行为真则表示一个请求头被成功解析，在 1056 行先把它加入到 `r->headers_in.headers` 内，然后在 `cmcf->headers_in_hash`（该变量对应 `ngx_http_headers_in` 变量）内查找该请求头能否被 `nginx` 处理，比如 "Host" 请求头就能够被 `nginx` 处理，从而调用其对应的处理函数 `ngx_http_process_host()`。

当函数 `ngx_http_parse_header_line()` 返回 `NGX_HTTP_PARSE_HEADER_DONE` 时，表示所有的请求头都已经处理完成（最后一个被处理的请求头为 `entity-header`），客户端的具体请求已经基本被理解（可能还有请求体，比如 `POST` 时），`nginx` 开始进入到内部处理，即开始执行各种模块 `Handler`，不过在此之前，通过调用 `ngx_http_process_request_header()` 函数先做了一个简单的检查：

```

1099:     Filename : ngx_http_request.c
1100:             if (rc == NGX_HTTP_PARSE_HEADER_DONE) {
1101:                 ...
1110:                 rc = ngx_http_process_request_header(r);
1111:
1112:                 if (rc != NGX_OK) {
1113:                     return;
1114:                 ...
1116:                 ngx_http_process_request(r);

```

函数 `ngx_http_process_request_header()` 的检查比较简单，比如如果客户端使用 `HTTP 1.1` 协议发送请求却没有带上 "Host" 请求头则直接返回错误（`HTTP 1.1` 协议明确要求必须有 "Host" 请求头）；客户端发送 `TRACE` 请求则也返回错误（`TRACE` 请求用于调试跟踪，`nginx` 不支持）；等。

调用函数 `ngx_http_process_request()` 也就是开始执行各种模块 `Handler`，也就是那个前面章节曾提到过的“状态机”：

```

ngx_http_process_request() -> ngx_http_handler() -> ngx_http_core_run_phases()
874: Filename : ngx_http_core_module.c
875:     while (ph[r->phase_handler].checker) {
876:
877:         rc = ph[r->phase_handler].checker(r, &ph[r->phase_handler]);

```



```
878:
879:     if(rc == NGX_OK) {
880:         return;
881:     }
882: }
```

对一个客户端请求的处理，终于衔接到 nginx 的 Handler 模块来了，各个 Handler 模块的处理在前面章节已经描述过，所以这里不再多讲。对于一个访问静态页面的 GET 类型请求，最终会被 ngx\_http\_static\_module 模块的 ngx\_http\_static\_handler() 函数捕获，该函数组织待响应的数据，然后调用 ngx\_http\_output\_filter() 经过 nginx 过滤链后将数据发送到客户端，此时一个请求的处理与响应也就完成，所以当回到 ngx\_http\_process\_request() 函数的最末，调用到函数 ngx\_http\_run\_posted\_requests() 内时，因为 c->destroyed 为真而直接退出。

请求处理响应完后，调用函数 ngx\_http\_finalize\_request() 进行清理工作，由于 nginx 对内存的使用采用内存池的方式，所以回收起来非常的简单，也不会出现内存泄露，还有一些其它清理工作，比如从事件超时红黑树里移除等也都比较简单，无需多说。

## HTTP 数据响应

http 响应消息也分为 head 头部和 body 主体，和请求消息一致，也是头部信息先发送，然后才是主体信息。本节仍以简单的 GET 请求静态页面为例，来看看 nginx 如何对客户端做出数据响应。

前面提到，简单的 GET 请求静态页面会最终被 ngx\_http\_static\_module 模块实际处理，执行的函数为 ngx\_http\_static\_handler()，该函数首先要做的当然是找到请求静态页面所对应的磁盘文件，这通过组合当前 location 配置的根目录与 GET 请求里的绝对 URI 即可得到该磁盘文件的绝对路径。

接着通过绝对路径打开该磁盘文件，并且通过文件属性来设置相关响应头，比如通过文件大小来设置 Content-Length 响应头（这里还只是设置对应的字段值，并非创建实际的响应头，下同），告诉客户端接收数据的长度；通过文件修改时间来设置 Last-Modified 响应头，那么客户端下次再请求该静态文件时可带上该时间戳，那时 nginx 就有可能直接返回 304 状态码，让客户端直接使用本地缓存，从而提高性能；等等。发送响应体需要一些内存资源，这会在发送响应头以前分配好，因为如果内存申请失败可提前异常返回，避免可能出现响应头已经发送出去后却发现发送响应体所需要的内存资源却没法成功申请的情况。当然，发送响应头还需要经过 nginx 的过滤链，这是通过函数：

```
ngx_http_send_header() -> ngx_http_top_header_filter()
```

逐步顺链调用下去，过滤链上的回调函数可能会对响应头数据进行检测、截获、新增、修改和删除等操作，不管怎样，一般情况下，执行流程会走到过滤链最末端的两个函数内：



`ngx_http_header_filter() -> ngx_http_write_filter()`

其中函数 `ngx_http_header_filter()` 完成响应头字符串数据的组织工作。该函数申请一个 buf 缓存块，然后根据最初设置以及经过过滤链的修改后的相关响应头字段值，组织响应头数据以字符串的形式存储在该缓存块内，下面是在该函数接近末尾的地方，用 gdb 捕获到的数据：

```
(gdb) p b->pos
```

```
$39 = (u_char *) 0x98be920 "HTTP/1.1 200 OK\r\nServer: nginx/1.2.0\r\nDate: Sun, 27 May 2012 13:58:31 GMT\r\nContent-Type: text/html\r\nContent-Length: 219\r\nLast-Modified: Fri, 25 May 2012 15:20:11 GMT\r\nConnection: keep-alive\r\nAccept-Ranges: bytes\r\n\r\n"
```

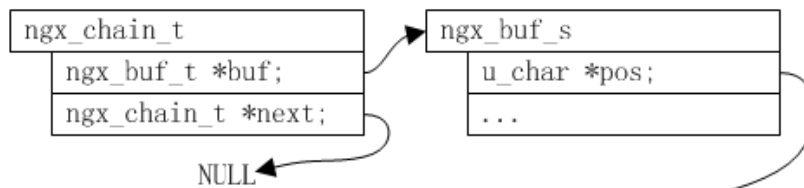
该缓存块被接入到发送链变量 `out`（注意这是一个局部变量）内，之后进入到函数 `ngx_http_write_filter()` 进行“写出”操作，打上引号是因为此处只有在满足某些条件的情况下才会执行实际的数据写出：

```
46: Filename : ngx_http_write_filter_module.c
47: ngx_int_t
48: ngx_http_write_filter(ngx_http_request_t *r, ngx_chain_t *in)
49: {
50: ...
172: /*
173:  * avoid the output if there are no last buf, no flush point,
174:  * there are the incoming bufs and the size of all bufs
175:  * is smaller than "postpone_output" directive
176:  */
177:
178: if (!last && !flush && in && size < (off_t) clcf->postpone_output) {
179:     return NGX_OK;
180: }
```

可以看到如果没有带最后一个缓存块（`last`）并且没有要求强制写出（`flush`）并且当前有新加缓存块（`in` 为真）并且当前缓存块总数据大小小于设定值（`clcf->postpone_output`），此时可直接返回 `NGX_OK`，这意味着会有数据马上跟来（所以该 `if` 语句为什么会有对 `in` 是否为真的判断就是因为这个原因，如果当前都没有新加入数据，那么也不要期待下一步会马上有数据加入，因此基于这种思路，从时延上考虑，就需要立即写出，从而这整个 `if` 判断为假），所以此次可以不写。之所以这么做，当然还是从性能上考虑，在其它章节我们可以看到不管是用哪种读/写方式，总还是要进行用户空间与内核空间的切换，性能损耗比较大，所以读/写操作能省一次就一次。

在我们的示例里，或者说是客户端访问服务器静态页面的这种情况下，那么此时一般就是从低 179 行退出返回了，但是在该函数前面的逻辑里，我们的待发送缓存块（即包含响应头数据的字符串数据）被连接到 `r->out` 链内了，这样做是必须的，毕竟传入进来的 `out` 发送链是个局部变量。此时的情况如下：

r->out



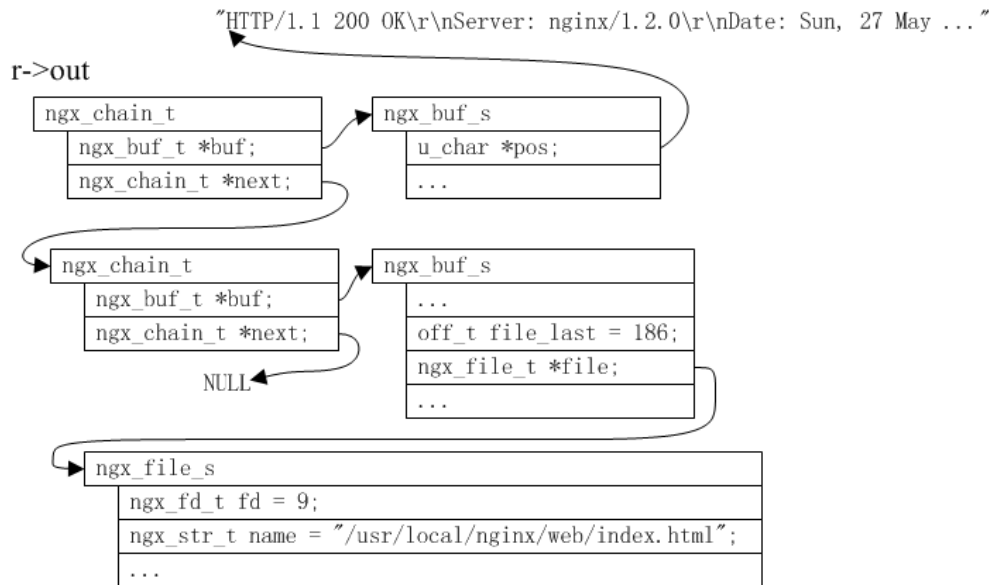
"HTTP/1.1 200 OK\r\nServer: nginx/1.2.0\r\nDate: Sun, 27 May ..."

函数依次返回后到函数 ngx\_http\_static\_handler()内继续执行，看一下相关的完整代码：

```

47: Filename : ngx_http_static_module.c
48: static ngx_int_t
49: ngx_http_static_handler(ngx_http_request_t *r)
50: {
51: ...
235:     b = ngx_palloc(r->pool, sizeof(ngx_buf_t));
236: ...
240:     b->file = ngx_palloc(r->pool, sizeof(ngx_file_t));
241: ...
245:     rc = ngx_http_send_header(r);
246: ...
255:     b->last_buf = (r == r->main) ? 1: 0;
256: ...
258:     b->file->fd = of.fd;
259:     b->file->name = path;
260: ...
266:     return ngx_http_output_filter(r, &out);
267: }
    
```

代码第 235、240、245 行在前面已经描述过了，第 255 行的 last\_buf 会被置 1，即由于当前请求就是主请求，第 258、259 行在后面实现将静态文件写出到客户端时会要用到，第 266 行开始进入到 body 过滤链，最后也进入到函数 ngx\_http\_write\_filter()内，同样，该函数的前面逻辑把这个新缓存块也加入到 r->out 链内，但是在判断是否要实际写出的 if 判断时，由于 last 标记为真，所以此时的确需要做数据写出操作。此时的 r->out 链情况如下所示：



在进行实际的数据写出操作时，会有一些其它与本节无关的细节，比如限速、一次没完全写完等需设置定时器再写，撇开这些而关注我们的重点函数：

241: Filename : ngx\_http\_write\_filter\_module.c

242: chain = c->send\_chain(c, r->out, limit);

回调指针 send\_chain 根据系统环境的不同而指向不同的函数，关于这点在其它章节会讲到，在我这里指向的是 ngx\_linux\_sendfile\_chain() 函数，该函数遍历 r->out 链上的每一个缓存块，根据缓存块里的数据类型调用不同的系统接口函数将数据写出到客户端。比如这里，对于第一个缓存块，其内数据是存放在内存中的字符串数据（响应头），所以调用系统接口函数 writev() 将其写出；对于第二个缓存块，其相关联数据是磁盘上文件系统内某个文件（该文件已被打开，对应文件描述符存放在 buf->file->fd 内）的内容，对于这些数据的写出采用的是系统调用 sendfile()。相关代码为：

36: Filename : ngx\_linux\_sendfile\_chain.c

37: ngx\_chain\_t \*

38: ngx\_linux\_sendfile\_chain(ngx\_connection\_t \*c, ngx\_chain\_t \*in, off\_t limit)

39: {

40: ...

78: for ( ;; ) {

79: ...

92: for (cl = in; cl && send < limit; cl = cl->next) {

93: ...

248: if (file) {

249: ...

264: rc = sendfile(c->fd, file->file->fd, &offset, file\_size);

265: ...

293: } else {

294: rc = writev(c->fd, header.elts, header.nelts);

客户端需要的数据都发送出去了,那么剩下的工作也就是进行连接关闭和一些连接相关资源的清理,当然,如果需要与客户端进行 keepalive,那么会执行函数 `ngx_http_set_keepalive()` 保留一些可重用的资源,这样在客户端新的请求到达时,处理能更快速。不过,对于一个客户端请求的处理与响应,到此就已经算是完满了。

## 附录 A

```
00: Filename : objs/nginx_modules.c
01:
02: #include <ngx_config.h>
03: #include <ngx_core.h>
04:
05:
06:
07: extern ngx_module_t  ngx_core_module;
08: extern ngx_module_t  ngx_errlog_module;
09: extern ngx_module_t  ngx_conf_module;
10: extern ngx_module_t  ngx_events_module;
11: extern ngx_module_t  ngx_event_core_module;
12: extern ngx_module_t  ngx_epoll_module;
13: extern ngx_module_t  ngx_regex_module;
14: extern ngx_module_t  ngx_http_module;
15: extern ngx_module_t  ngx_http_core_module;
16: extern ngx_module_t  ngx_http_log_module;
17: extern ngx_module_t  ngx_http_upstream_module;
18: extern ngx_module_t  ngx_http_static_module;
19: extern ngx_module_t  ngx_http_autoindex_module;
20: extern ngx_module_t  ngx_http_index_module;
21: extern ngx_module_t  ngx_http_auth_basic_module;
22: extern ngx_module_t  ngx_http_access_module;
23: extern ngx_module_t  ngx_http_limit_conn_module;
24: extern ngx_module_t  ngx_http_limit_req_module;
25: extern ngx_module_t  ngx_http_geo_module;
26: extern ngx_module_t  ngx_http_map_module;
27: extern ngx_module_t  ngx_http_split_clients_module;
28: extern ngx_module_t  ngx_http_referer_module;
29: extern ngx_module_t  ngx_http_rewrite_module;
30: extern ngx_module_t  ngx_http_proxy_module;
31: extern ngx_module_t  ngx_http_fastcgi_module;
32: extern ngx_module_t  ngx_http_uwsgi_module;
33: extern ngx_module_t  ngx_http_scgi_module;
34: extern ngx_module_t  ngx_http_memcached_module;
35: extern ngx_module_t  ngx_http_empty_gif_module;
36: extern ngx_module_t  ngx_http_browser_module;
37: extern ngx_module_t  ngx_http_upstream_ip_hash_module;
38: extern ngx_module_t  ngx_http_upstream_keepalive_module;
39: extern ngx_module_t  ngx_http_write_filter_module;
40: extern ngx_module_t  ngx_http_header_filter_module;
41: extern ngx_module_t  ngx_http_chunked_filter_module;
```

```
42: extern ngx_module_t  ngx_http_range_header_filter_module;
43: extern ngx_module_t  ngx_http_gzip_filter_module;
44: extern ngx_module_t  ngx_http_postpone_filter_module;
45: extern ngx_module_t  ngx_http_ssi_filter_module;
46: extern ngx_module_t  ngx_http_charset_filter_module;
47: extern ngx_module_t  ngx_http_userid_filter_module;
48: extern ngx_module_t  ngx_http_headers_filter_module;
49: extern ngx_module_t  ngx_http_copy_filter_module;
50: extern ngx_module_t  ngx_http_range_body_filter_module;
51: extern ngx_module_t  ngx_http_not_modified_filter_module;
52:
53: ngx_module_t *ngx_modules[] = {
54:     &ngx_core_module,
55:     &ngx_errlog_module,
56:     &ngx_conf_module,
57:     &ngx_events_module,
58:     &ngx_event_core_module,
59:     &ngx_epoll_module,
60:     &ngx_regex_module,
61:     &ngx_http_module,
62:     &ngx_http_core_module,
63:     &ngx_http_log_module,
64:     &ngx_http_upstream_module,
65:     &ngx_http_static_module,
66:     &ngx_http_autoindex_module,
67:     &ngx_http_index_module,
68:     &ngx_http_auth_basic_module,
69:     &ngx_http_access_module,
70:     &ngx_http_limit_conn_module,
71:     &ngx_http_limit_req_module,
72:     &ngx_http_geo_module,
73:     &ngx_http_map_module,
74:     &ngx_http_split_clients_module,
75:     &ngx_http_referer_module,
76:     &ngx_http_rewrite_module,
77:     &ngx_http_proxy_module,
78:     &ngx_http_fastcgi_module,
79:     &ngx_http_uwsgi_module,
80:     &ngx_http_scgi_module,
81:     &ngx_http_memcached_module,
82:     &ngx_http_empty_gif_module,
83:     &ngx_http_browser_module,
84:     &ngx_http_upstream_ip_hash_module,
85:     &ngx_http_upstream_keepalive_module,
```

```
86:    &ngx_http_write_filter_module,
87:    &ngx_http_header_filter_module,
88:    &ngx_http_chunked_filter_module,
89:    &ngx_http_range_header_filter_module,
90:    &ngx_http_gzip_filter_module,
91:    &ngx_http_postpone_filter_module,
92:    &ngx_http_ssi_filter_module,
93:    &ngx_http_charset_filter_module,
94:    &ngx_http_userid_filter_module,
95:    &ngx_http_headers_filter_module,
96:    &ngx_http_copy_filter_module,
97:    &ngx_http_range_body_filter_module,
98:    &ngx_http_not_modified_filter_module,
99:    NULL
100: };
101:
```



## 附录 B

```
worker_processes 1;

events {
    worker_connections 1024;
}

http {
    include mime.types;
    default_type application/octet-stream;

    sendfile on;
    keepalive_timeout 65;

    server {
        listen 80;
        server_name localhost;

        location / {
            root html;
            index index.html index.htm;
        }

        error_page 500 502 503 504 /50x.html;
        location = /50x.html {
            root html;
        }
    }
}
```