# nginx的请求处理阶段 (90%)

## 接收请求流程 (99%)

### http请求格式简介 (99%)

首先介绍一下rfc2616中定义的http请求基本格式：

Request = Request-Line   
 \* (( general-header   
 | request-header   
 | entity-header ) CRLF)   
 CRLF  
 [ message-body ]

第一行是请求行（request line），用来说明请求方法，要访问的资源以及所使用的HTTP版本：

Request-Line = Method SP Request-URI SP HTTP-Version CRLF

请求方法（Method）的定义如下，其中最常用的是GET，POST方法：

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

要访问的资源由统一资源地位符URI(Uniform Resource Identifier)确定，它的一个比较通用的组成格式（rfc2396）如下：

<scheme>://<authority><path>?<query>

一般来说根据请求方法（Method）的不同，请求URI的格式会有所不同，通常只需写出path和query部分。

http版本(version)定义如下，现在用的一般为1.0和1.1版本：

HTTP/<major>.<minor>

请求行的下一行则是请求头，rfc2616中定义了3种不同类型的请求头，分别为general-header，request-header和entity-header，每种类型rfc中都定义了一些通用的头，其中entity-header类型可以包含自定义的头。

### 请求头读取 (99%)

这一节介绍nginx中请求头的解析，nginx的请求处理流程中，会涉及到2个非常重要的数据结构，ngx\_connection\_t和ngx\_http\_request\_t，分别用来表示连接和请求，这2个数据结构在本书的前篇中已经做了比较详细的介绍，没有印象的读者可以翻回去复习一下，整个请求处理流程从头到尾，对应着这2个数据结构的分配，初始化，使用，重用和销毁。

nginx在初始化阶段，具体是在init process阶段的ngx\_event\_process\_init函数中会为每一个监听套接字分配一个连接结构（ngx\_connection\_t），并将该连接结构的读事件成员（read）的事件处理函数设置为ngx\_event\_accept，并且如果没有使用accept互斥锁的话，在这个函数中会将该读事件挂载到nginx的事件处理模型上（poll或者epoll等），反之则会等到init process阶段结束，在工作进程的事件处理循环中，某个进程抢到了accept锁才能挂载该读事件。

static ngx\_int\_t  
ngx\_event\_process\_init(ngx\_cycle\_t \*cycle)  
{  
 ...  
  
 /\* 初始化用来管理所有定时器的红黑树 \*/  
 if (ngx\_event\_timer\_init(cycle->log) == NGX\_ERROR) {  
 return NGX\_ERROR;  
 }  
 /\* 初始化事件模型 \*/  
 for (m = 0; ngx\_modules[m]; m++) {  
 if (ngx\_modules[m]->type != NGX\_EVENT\_MODULE) {  
 continue;  
 }  
  
 if (ngx\_modules[m]->ctx\_index != ecf->use) {  
 continue;  
 }  
  
 module = ngx\_modules[m]->ctx;  
  
 if (module->actions.init(cycle, ngx\_timer\_resolution) != NGX\_OK) {  
 /\* fatal \*/  
 exit(2);  
 }  
  
 break;  
 }  
  
 ...  
  
 /\* for each listening socket \*/  
 /\* 为每个监听套接字分配一个连接结构 \*/  
 ls = cycle->listening.elts;  
 for (i = 0; i < cycle->listening.nelts; i++) {  
  
 c = ngx\_get\_connection(ls[i].fd, cycle->log);  
  
 if (c == NULL) {  
 return NGX\_ERROR;  
 }  
  
 c->log = &ls[i].log;  
  
 c->listening = &ls[i];  
 ls[i].connection = c;  
  
 rev = c->read;  
  
 rev->log = c->log;  
 /\* 标识此读事件为新请求连接事件 \*/  
 rev->accept = 1;  
  
 ...  
  
#if (NGX\_WIN32)  
  
 /\* windows环境下不做分析，但原理类似 \*/  
  
#else  
 /\* 将读事件结构的处理函数设置为ngx\_event\_accept \*/  
 rev->handler = ngx\_event\_accept;  
 /\* 如果使用accept锁的话，要在后面抢到锁才能将监听句柄挂载上事件处理模型上 \*/  
 if (ngx\_use\_accept\_mutex) {  
 continue;  
 }  
 /\* 否则，将该监听句柄直接挂载上事件处理模型 \*/  
 if (ngx\_event\_flags & NGX\_USE\_RTSIG\_EVENT) {  
 if (ngx\_add\_conn(c) == NGX\_ERROR) {  
 return NGX\_ERROR;  
 }  
  
 } else {  
 if (ngx\_add\_event(rev, NGX\_READ\_EVENT, 0) == NGX\_ERROR) {  
 return NGX\_ERROR;  
 }  
 }  
  
#endif  
  
 }  
  
 return NGX\_OK;  
}

当一个工作进程在某个时刻将监听事件挂载上事件处理模型之后，nginx就可以正式的接收并处理客户端过来的请求了。这时如果有一个用户在浏览器的地址栏内输入一个域名，并且域名解析服务器将该域名解析到一台由nginx监听的服务器上，nginx的事件处理模型接收到这个读事件之后，会交给之前注册好的事件处理函数ngx\_event\_accept来处理。

在ngx\_event\_accept函数中，nginx调用accept函数，从已连接队列得到一个连接以及对应的套接字，接着分配一个连接结构（ngx\_connection\_t），并将新得到的套接字保存在该连接结构中，这里还会做一些基本的连接初始化工作：

1, 首先给该连接分配一个内存池，初始大小默认为256字节，可通过connection\_pool\_size指令设置；

2, 分配日志结构，并保存在其中，以便后续的日志系统使用；

3, 初始化连接相应的io收发函数，具体的io收发函数和使用的事件模型及操作系统相关；

4, 分配一个套接口地址（sockaddr），并将accept得到的对端地址拷贝在其中，保存在sockaddr字段；

5, 将本地套接口地址保存在local\_sockaddr字段，因为这个值是从监听结构ngx\_listening\_t中可得，而监听结构中保存的只是配置文件中设置的监听地址，但是配置的监听地址可能是通配符\*，即监听在所有的地址上，所以连接中保存的这个值最终可能还会变动，会被确定为真正的接收地址；

6, 将连接的写事件设置为已就绪，即设置ready为1，nginx默认连接第一次为可写；

7, 如果监听套接字设置了TCP\_DEFER\_ACCEPT属性，则表示该连接上已经有数据包过来，于是设置读事件为就绪；

8, 将sockaddr字段保存的对端地址格式化为可读字符串，并保存在addr\_text字段；

最后调用ngx\_http\_init\_connection函数初始化该连接结构的其他部分。

ngx\_http\_init\_connection函数最重要的工作是初始化读写事件的处理函数：将该连接结构的写事件的处理函数设置为ngx\_http\_empty\_handler，这个事件处理函数不会做任何操作，实际上nginx默认连接第一次可写，不会挂载写事件，如果有数据需要发送，nginx会直接写到这个连接，只有在发生一次写不完的情况下，才会挂载写事件到事件模型上，并设置真正的写事件处理函数，这里后面的章节还会做详细介绍；读事件的处理函数设置为ngx\_http\_init\_request，此时如果该连接上已经有数据过来（设置了deferred accept)，则会直接调用ngx\_http\_init\_request函数来处理该请求，反之则设置一个定时器并在事件处理模型上挂载一个读事件，等待数据到来或者超时。当然这里不管是已经有数据到来，或者需要等待数据到来，又或者等待超时，最终都会进入读事件的处理函数-ngx\_http\_init\_request。

ngx\_http\_init\_request函数主要工作即是初始化请求，由于它是一个事件处理函数，它只有唯一一个ngx\_event\_t \*类型的参数，ngx\_event\_t 结构在nginx中表示一个事件，事件处理的上下文类似于一个中断处理的上下文，为了在这个上下文得到相关的信息，nginx中一般会将连接结构的引用保存在事件结构的data字段，请求结构的引用则保存在连接结构的data字段，这样在事件处理函数中可以方便的得到对应的连接结构和请求结构。进入函数内部看一下，首先判断该事件是否是超时事件，如果是的话直接关闭连接并返回；反之则是指之前accept的连接上有请求过来需要处理。

ngx\_http\_init\_request函数首先在连接的内存池中为该请求分配一个ngx\_http\_request\_t结构，这个结构将用来保存该请求所有的信息。分配完之后，这个结构的引用会被包存在连接的hc成员的request字段，以便于在长连接或pipelined请求中复用该请求结构。在这个函数中，nginx根据该请求的接收端口和地址找到一个默认虚拟服务器配置（listen指令的default\_server属性用来标识一个默认虚拟服务器，否则监听在相同端口和地址的多个虚拟服务器，其中第一个定义的则为默认）。

nginx配置文件中可以设置多个监听在不同端口和地址的虚拟服务器（每个server块对应一个虚拟服务器），另外还根据域名（server\_name指令可以配置该虚拟服务器对应的域名）来区分监听在相同端口和地址的虚拟服务器，每个虚拟服务器可以拥有不同的配置内容，而这些配置内容决定了nginx在接收到一个请求之后如何处理该请求。找到之后，相应的配置被保存在该请求对应的ngx\_http\_request\_t结构中。注意这里根据端口和地址找到的默认配置只是临时使用一下，最终nginx会根据域名找到真正的虚拟服务器配置，随后的初始化工作还包括：

1, 将连接的读事件的处理函数设置为ngx\_http\_process\_request\_line函数，这个函数用来解析请求行，将请求的read\_event\_handler设置为ngx\_http\_block\_reading函数，这个函数实际上什么都不做（当然在事件模型设置为水平触发时，唯一做的事情就是将事件从事件模型监听列表中删除，防止该事件一直被触发），后面会说到这里为什么会将read\_event\_handler设置为此函数；

2, 为这个请求分配一个缓冲区用来保存它的请求头，地址保存在header\_in字段，默认大小为1024个字节，可以使用client\_header\_buffer\_size指令修改，这里需要注意一下，nginx用来保存请求头的缓冲区是在该请求所在连接的内存池中分配，而且会将地址保存一份在连接的buffer字段中，这样做的目的也是为了给该连接的下一次请求重用这个缓冲区，另外如果客户端发过来的请求头大于1024个字节，nginx会重新分配更大的缓存区，默认用于大请求的头的缓冲区最大为8K，最多4个，这2个值可以用large\_client\_header\_buffers指令设置，后面还会说到请求行和一个请求头都不能超过一个最大缓冲区的大小；

3, 为这个请求分配一个内存池，后续所有与该请求相关的内存分配一般都会使用该内存池，默认大小为4096个字节，可以使用request\_pool\_size指令修改；

4, 为这个请求分配响应头链表，初始大小为20；

5, 创建所有模块的上下文ctx指针数组，变量数据；

6, 将该请求的main字段设置为它本身，表示这是一个主请求，nginx中对应的还有子请求概念，后面的章节会做详细的介绍；

7, 将该请求的count字段设置为1，count字段表示请求的引用计数；

8, 将当前时间保存在start\_sec和start\_msec字段，这个时间是该请求的起始时刻，将被用来计算一个请求的处理时间（request time），nginx使用的这个起始点和apache略有差别，nginx中请求的起始点是接收到客户端的第一个数据包的事件开始，而apache则是接收到客户端的整个request line后开始算起；

9, 初始化请求的其他字段，比如将uri\_changes设置为11，表示最多可以将该请求的uri改写10次，subrequests被设置为201，表示一个请求最多可以发起200个子请求；

做完所有这些初始化工作之后，ngx\_http\_init\_request函数会调用读事件的处理函数来真正的解析客户端发过来的数据，也就是会进入ngx\_http\_process\_request\_line函数中处理。

#### 解析请求行 (99%)

ngx\_http\_process\_request\_line函数的主要作用即是解析请求行，同样由于涉及到网络IO操作，即使是很短的一行请求行可能也不能被一次读完，所以在之前的ngx\_http\_init\_request函数中，ngx\_http\_process\_request\_line函数被设置为读事件的处理函数，它也只拥有一个唯一的ngx\_event\_t \*类型参数，并且在函数的开头，同样需要判断是否是超时事件，如果是的话，则关闭这个请求和连接；否则开始正常的解析流程。先调用ngx\_http\_read\_request\_header函数读取数据。

由于可能多次进入ngx\_http\_process\_request\_line函数，ngx\_http\_read\_request\_header函数首先检查请求的header\_in指向的缓冲区内是否有数据，有的话直接返回；否则从连接读取数据并保存在请求的header\_in指向的缓存区，而且只要缓冲区有空间的话，会一次尽可能多的读数据，读到多少返回多少；如果客户端暂时没有发任何数据过来，并返回NGX\_AGAIN，返回之前会做2件事情：

1，设置一个定时器，时长默认为60s，可以通过指令client\_header\_timeout设置，如果定时事件到达之前没有任何可读事件，nginx将会关闭此请求；

2，调用ngx\_handle\_read\_event函数处理一下读事件-如果该连接尚未在事件处理模型上挂载读事件，则将其挂载上；

如果客户端提前关闭了连接或者读取数据发生了其他错误，则给客户端返回一个400错误（当然这里并不保证客户端能够接收到响应数据，因为客户端可能都已经关闭了连接），最后函数返回NGX\_ERROR；

如果ngx\_http\_read\_request\_header函数正常的读取到了数据，ngx\_http\_process\_request\_line函数将调用ngx\_http\_parse\_request\_line函数来解析，这个函数根据http协议规范中对请求行的定义实现了一个有限状态机，经过这个状态机，nginx会记录请求行中的请求方法（Method），请求uri以及http协议版本在缓冲区中的起始位置，在解析过程中还会记录一些其他有用的信息，以便后面的处理过程中使用。如果解析请求行的过程中没有产生任何问题，该函数会返回NGX\_OK；如果请求行不满足协议规范，该函数会立即终止解析过程，并返回相应错误号；如果缓冲区数据不够，该函数返回NGX\_AGAIN。

在整个解析http请求的状态机中始终遵循着两条重要的原则：减少内存拷贝和回溯。

内存拷贝是一个相对比较昂贵的操作，大量的内存拷贝会带来较低的运行时效率。nginx在需要做内存拷贝的地方尽量只拷贝内存的起始和结束地址而不是内存本身，这样做的话仅仅只需要两个赋值操作而已，大大降低了开销，当然这样带来的影响是后续的操作不能修改内存本身，如果修改的话，会影响到所有引用到该内存区间的地方，所以必须很小心的管理，必要的时候需要拷贝一份。

这里不得不提到nginx中最能体现这一思想的数据结构，ngx\_buf\_t，它用来表示nginx中的缓存，在很多情况下，只需要将一块内存的起始地址和结束地址分别保存在它的pos和last成员中，再将它的memory标志置1，即可表示一块不能修改的内存区间，在另外的需要一块能够修改的缓存的情形中，则必须分配一块所需大小的内存并保存其起始地址，再将ngx\_buf\_t的temporary标志置1，表示这是一块能够被修改的内存区域。

再回到ngx\_http\_process\_request\_line函数中，如果ngx\_http\_parse\_request\_line函数返回了错误，则直接给客户端返回400错误； 如果返回NGX\_AGAIN，则需要判断一下是否是由于缓冲区空间不够，还是已读数据不够。如果是缓冲区大小不够了，nginx会调用ngx\_http\_alloc\_large\_header\_buffer函数来分配另一块大缓冲区，如果大缓冲区还不够装下整个请求行，nginx则会返回414错误给客户端，否则分配了更大的缓冲区并拷贝之前的数据之后，继续调用ngx\_http\_read\_request\_header函数读取数据来进入请求行自动机处理，直到请求行解析结束；

如果返回了NGX\_OK，则表示请求行被正确的解析出来了，这时先记录好请求行的起始地址以及长度，并将请求uri的path和参数部分保存在请求结构的uri字段，请求方法起始位置和长度保存在method\_name字段，http版本起始位置和长度记录在http\_protocol字段。还要从uri中解析出参数以及请求资源的拓展名，分别保存在args和exten字段。接下来将要解析请求头，将在下一小节中接着介绍。

#### 解析请求头 (99%)

在ngx\_http\_process\_request\_line函数中，解析完请求行之后，如果请求行的uri里面包含了域名部分，则将其保存在请求结构的headers\_in成员的server字段，headers\_in用来保存所有请求头，它的类型为ngx\_http\_headers\_in\_t：

typedef struct {  
 ngx\_list\_t headers;  
  
 ngx\_table\_elt\_t \*host;  
 ngx\_table\_elt\_t \*connection;  
 ngx\_table\_elt\_t \*if\_modified\_since;  
 ngx\_table\_elt\_t \*if\_unmodified\_since;  
 ngx\_table\_elt\_t \*user\_agent;  
 ngx\_table\_elt\_t \*referer;  
 ngx\_table\_elt\_t \*content\_length;  
 ngx\_table\_elt\_t \*content\_type;  
  
 ngx\_table\_elt\_t \*range;  
 ngx\_table\_elt\_t \*if\_range;  
  
 ngx\_table\_elt\_t \*transfer\_encoding;  
 ngx\_table\_elt\_t \*expect;  
  
#if (NGX\_HTTP\_GZIP)  
 ngx\_table\_elt\_t \*accept\_encoding;  
 ngx\_table\_elt\_t \*via;  
#endif  
  
 ngx\_table\_elt\_t \*authorization;  
  
 ngx\_table\_elt\_t \*keep\_alive;  
  
#if (NGX\_HTTP\_PROXY || NGX\_HTTP\_REALIP || NGX\_HTTP\_GEO)  
 ngx\_table\_elt\_t \*x\_forwarded\_for;  
#endif  
  
#if (NGX\_HTTP\_REALIP)  
 ngx\_table\_elt\_t \*x\_real\_ip;  
#endif  
  
#if (NGX\_HTTP\_HEADERS)  
 ngx\_table\_elt\_t \*accept;  
 ngx\_table\_elt\_t \*accept\_language;  
#endif  
  
#if (NGX\_HTTP\_DAV)  
 ngx\_table\_elt\_t \*depth;  
 ngx\_table\_elt\_t \*destination;  
 ngx\_table\_elt\_t \*overwrite;  
 ngx\_table\_elt\_t \*date;  
#endif  
  
 ngx\_str\_t user;  
 ngx\_str\_t passwd;  
  
 ngx\_array\_t cookies;  
  
 ngx\_str\_t server;  
 off\_t content\_length\_n;  
 time\_t keep\_alive\_n;  
  
 unsigned connection\_type:2;  
 unsigned msie:1;  
 unsigned msie6:1;  
 unsigned opera:1;  
 unsigned gecko:1;  
 unsigned chrome:1;  
 unsigned safari:1;  
 unsigned konqueror:1;  
} ngx\_http\_headers\_in\_t;

接着，该函数会检查进来的请求是否使用的是http0.9，如果是的话则使用从请求行里得到的域名，调用ngx\_http\_find\_virtual\_server（）函数来查找用来处理该请求的虚拟服务器配置，之前通过端口和地址找到的默认配置不再使用，找到相应的配置之后，则直接调用ngx\_http\_process\_request（）函数处理该请求，因为http0.9是最原始的http协议，它里面没有定义任何请求头，显然就不需要读取请求头的操作。

if (r->host\_start && r->host\_end) {  
  
 host = r->host\_start;  
 n = ngx\_http\_validate\_host(r, &host,  
 r->host\_end - r->host\_start, 0);  
  
 if (n == 0) {  
 ngx\_log\_error(NGX\_LOG\_INFO, c->log, 0,  
 "client sent invalid host in request line");  
 ngx\_http\_finalize\_request(r, NGX\_HTTP\_BAD\_REQUEST);  
 return;  
 }  
  
 if (n < 0) {  
 ngx\_http\_close\_request(r, NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);  
 return;  
 }  
  
 r->headers\_in.server.len = n;  
 r->headers\_in.server.data = host;  
}  
  
if (r->http\_version < NGX\_HTTP\_VERSION\_10) {  
  
 if (ngx\_http\_find\_virtual\_server(r, r->headers\_in.server.data,  
 r->headers\_in.server.len)  
 == NGX\_ERROR)  
 {  
 ngx\_http\_close\_request(r, NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);  
 return;  
 }  
  
 ngx\_http\_process\_request(r);  
 return;  
}

当然，如果是1.0或者更新的http协议，接下来要做的就是读取请求头了，首先nginx会为请求头分配空间，ngx\_http\_headers\_in\_t结构的headers字段为一个链表结构，它被用来保存所有请求头，初始为它分配了20个节点，每个节点的类型为ngx\_table\_elt\_t，保存请求头的name/value值对，还可以看到ngx\_http\_headers\_in\_t结构有很多类型为ngx\_table\_elt\_t\*的指针成员，而且从它们的命名可以看出是一些常见的请求头名字，nginx对这些常用的请求头在ngx\_http\_headers\_in\_t结构里面保存了一份引用，后续需要使用的话，可以直接通过这些成员得到，另外也事先为cookie头分配了2个元素的数组空间，做完这些内存准备工作之后，该请求对应的读事件结构的处理函数被设置为ngx\_http\_process\_request\_headers，并随后马上调用了该函数。

if (ngx\_list\_init(&r->headers\_in.headers, r->pool, 20,  
 sizeof(ngx\_table\_elt\_t))  
 != NGX\_OK)  
{  
 ngx\_http\_close\_request(r, NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);  
 return;  
}  
  
  
if (ngx\_array\_init(&r->headers\_in.cookies, r->pool, 2,  
 sizeof(ngx\_table\_elt\_t \*))  
 != NGX\_OK)  
{  
 ngx\_http\_close\_request(r, NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);  
 return;  
}  
  
c->log->action = "reading client request headers";  
  
rev->handler = ngx\_http\_process\_request\_headers;  
ngx\_http\_process\_request\_headers(rev);

ngx\_http\_process\_request\_headers函数循环的读取所有的请求头，并保存和初始化和请求头相关的结构，下面详细分析一下该函数：

因为nginx对读取请求头有超时限制，ngx\_http\_process\_request\_headers函数作为读事件处理函数，一并处理了超时事件，如果读超时了，nginx直接给该请求返回408错误：

if (rev->timedout) {  
 ngx\_log\_error(NGX\_LOG\_INFO, c->log, NGX\_ETIMEDOUT, "client timed out");  
 c->timedout = 1;  
 ngx\_http\_close\_request(r, NGX\_HTTP\_REQUEST\_TIME\_OUT);  
 return;  
 }

读取和解析请求头的逻辑和处理请求行差不多，总的流程也是循环的调用ngx\_http\_read\_request\_header（）函数读取数据，然后再调用一个解析函数来从读取的数据中解析请求头，直到解析完所有请求头，或者发生解析错误为主。当然由于涉及到网络io，这个流程可能发生在多个io事件的上下文中。

接着来细看该函数，先调用了ngx\_http\_read\_request\_header（）函数读取数据，如果当前连接并没有数据过来，再直接返回，等待下一次读事件到来，如果读到了一些数据则调用ngx\_http\_parse\_header\_line（）函数来解析，同样的该解析函数实现为一个有限状态机，逻辑很简单，只是根据http协议来解析请求头，每次调用该函数最多解析出一个请求头，该函数返回4种不同返回值，表示不同解析结果：

1，返回NGX\_OK，表示解析出了一行请求头，这时还要判断解析出的请求头名字里面是否有非法字符，名字里面合法的字符包括字母，数字和连字符（-），另外如果设置了underscores\_in\_headers指令为on，则下划线也是合法字符，但是nginx默认下划线不合法，当请求头里面包含了非法的字符，nginx默认只是忽略这一行请求头；如果一切都正常，nginx会将该请求头及请求头名字的hash值保存在请求结构体的headers\_in成员的headers链表,而且对于一些常见的请求头，如Host，Connection，nginx采用了类似于配置指令的方式，事先给这些请求头分配了一个处理函数，当解析出一个请求头时，会检查该请求头是否有设置处理函数，有的话则调用之，nginx所有有处理函数的请求头都记录在ngx\_http\_headers\_in全局数组中：

typedef struct {  
 ngx\_str\_t name;  
 ngx\_uint\_t offset;  
 ngx\_http\_header\_handler\_pt handler;  
} ngx\_http\_header\_t;  
  
ngx\_http\_header\_t ngx\_http\_headers\_in[] = {  
 { ngx\_string("Host"), offsetof(ngx\_http\_headers\_in\_t, host),  
 ngx\_http\_process\_host },  
  
 { ngx\_string("Connection"), offsetof(ngx\_http\_headers\_in\_t, connection),  
 ngx\_http\_process\_connection },  
  
 { ngx\_string("If-Modified-Since"),  
 offsetof(ngx\_http\_headers\_in\_t, if\_modified\_since),  
 ngx\_http\_process\_unique\_header\_line },  
  
 { ngx\_string("If-Unmodified-Since"),  
 offsetof(ngx\_http\_headers\_in\_t, if\_unmodified\_since),  
 ngx\_http\_process\_unique\_header\_line },  
  
 { ngx\_string("User-Agent"), offsetof(ngx\_http\_headers\_in\_t, user\_agent),  
 ngx\_http\_process\_user\_agent },  
  
 { ngx\_string("Referer"), offsetof(ngx\_http\_headers\_in\_t, referer),  
 ngx\_http\_process\_header\_line },  
  
 { ngx\_string("Content-Length"),  
 offsetof(ngx\_http\_headers\_in\_t, content\_length),  
 ngx\_http\_process\_unique\_header\_line },  
  
 { ngx\_string("Content-Type"),  
 offsetof(ngx\_http\_headers\_in\_t, content\_type),  
 ngx\_http\_process\_header\_line },  
  
 { ngx\_string("Range"), offsetof(ngx\_http\_headers\_in\_t, range),  
 ngx\_http\_process\_header\_line },  
  
 { ngx\_string("If-Range"),  
 offsetof(ngx\_http\_headers\_in\_t, if\_range),  
 ngx\_http\_process\_unique\_header\_line },  
  
 { ngx\_string("Transfer-Encoding"),  
 offsetof(ngx\_http\_headers\_in\_t, transfer\_encoding),  
 ngx\_http\_process\_header\_line },  
  
 { ngx\_string("Expect"),  
 offsetof(ngx\_http\_headers\_in\_t, expect),  
 ngx\_http\_process\_unique\_header\_line },  
  
#if (NGX\_HTTP\_GZIP)  
 { ngx\_string("Accept-Encoding"),  
 offsetof(ngx\_http\_headers\_in\_t, accept\_encoding),  
 ngx\_http\_process\_header\_line },  
  
 { ngx\_string("Via"), offsetof(ngx\_http\_headers\_in\_t, via),  
 ngx\_http\_process\_header\_line },  
#endif  
  
 { ngx\_string("Authorization"),  
 offsetof(ngx\_http\_headers\_in\_t, authorization),  
 ngx\_http\_process\_unique\_header\_line },  
  
 { ngx\_string("Keep-Alive"), offsetof(ngx\_http\_headers\_in\_t, keep\_alive),  
 ngx\_http\_process\_header\_line },  
  
#if (NGX\_HTTP\_PROXY || NGX\_HTTP\_REALIP || NGX\_HTTP\_GEO)  
 { ngx\_string("X-Forwarded-For"),  
 offsetof(ngx\_http\_headers\_in\_t, x\_forwarded\_for),  
 ngx\_http\_process\_header\_line },  
#endif  
  
#if (NGX\_HTTP\_REALIP)  
 { ngx\_string("X-Real-IP"),  
 offsetof(ngx\_http\_headers\_in\_t, x\_real\_ip),  
 ngx\_http\_process\_header\_line },  
#endif  
  
#if (NGX\_HTTP\_HEADERS)  
 { ngx\_string("Accept"), offsetof(ngx\_http\_headers\_in\_t, accept),  
 ngx\_http\_process\_header\_line },  
  
 { ngx\_string("Accept-Language"),  
 offsetof(ngx\_http\_headers\_in\_t, accept\_language),  
 ngx\_http\_process\_header\_line },  
#endif  
  
#if (NGX\_HTTP\_DAV)  
 { ngx\_string("Depth"), offsetof(ngx\_http\_headers\_in\_t, depth),  
 ngx\_http\_process\_header\_line },  
  
 { ngx\_string("Destination"), offsetof(ngx\_http\_headers\_in\_t, destination),  
 ngx\_http\_process\_header\_line },  
  
 { ngx\_string("Overwrite"), offsetof(ngx\_http\_headers\_in\_t, overwrite),  
 ngx\_http\_process\_header\_line },  
  
 { ngx\_string("Date"), offsetof(ngx\_http\_headers\_in\_t, date),  
 ngx\_http\_process\_header\_line },  
#endif  
  
 { ngx\_string("Cookie"), 0, ngx\_http\_process\_cookie },  
  
 { ngx\_null\_string, 0, NULL }  
};

ngx\_http\_headers\_in数组当前包含了25个常用的请求头，每个请求头都设置了一个处理函数，其中一部分请求头设置的是公共处理函数，这里有2个公共处理函数，ngx\_http\_process\_header\_line和ngx\_http\_process\_unique\_header\_line。 先来看一下处理函数的函数指针定义：

typedef ngx\_int\_t (\*ngx\_http\_header\_handler\_pt)(ngx\_http\_request\_t \*r,  
 ngx\_table\_elt\_t \*h, ngx\_uint\_t offset);

它有3个参数，r为对应的请求结构，h为指向该请求头在headers\_in.headers链表中对应节点的指针，offset为该请求头对应字段在ngx\_http\_headers\_in\_t结构中的偏移。

再来看ngx\_http\_process\_header\_line函数：

static ngx\_int\_t  
ngx\_http\_process\_header\_line(ngx\_http\_request\_t \*r, ngx\_table\_elt\_t \*h,  
 ngx\_uint\_t offset)  
{  
 ngx\_table\_elt\_t \*\*ph;  
  
 ph = (ngx\_table\_elt\_t \*\*) ((char \*) &r->headers\_in + offset);  
  
 if (\*ph == NULL) {  
 \*ph = h;  
 }  
  
 return NGX\_OK;  
}

这个函数只是简单将该请求头在ngx\_http\_headers\_in\_t结构中保存一份引用。ngx\_http\_process\_unique\_header\_line功能类似，不同点在于该函数会检查这个请求头是否是重复的，如果是的话，则给该请求返回400错误。

ngx\_http\_headers\_in数组中剩下的请求头都有自己特殊的处理函数，这些特殊的函数根据对应的请求头有一些特殊的处理，下面拿Host头的处理函数ngx\_http\_process\_host做一下介绍：

static ngx\_int\_t  
ngx\_http\_process\_host(ngx\_http\_request\_t \*r, ngx\_table\_elt\_t \*h,  
 ngx\_uint\_t offset)  
{  
 u\_char \*host;  
 ssize\_t len;  
  
 if (r->headers\_in.host == NULL) {  
 r->headers\_in.host = h;  
 }  
  
 host = h->value.data;  
 len = ngx\_http\_validate\_host(r, &host, h->value.len, 0);  
  
 if (len == 0) {  
 ngx\_log\_error(NGX\_LOG\_INFO, r->connection->log, 0,  
 "client sent invalid host header");  
 ngx\_http\_finalize\_request(r, NGX\_HTTP\_BAD\_REQUEST);  
 return NGX\_ERROR;  
 }  
  
 if (len < 0) {  
 ngx\_http\_close\_request(r, NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);  
 return NGX\_ERROR;  
 }  
  
 if (r->headers\_in.server.len) {  
 return NGX\_OK;  
 }  
  
 r->headers\_in.server.len = len;  
 r->headers\_in.server.data = host;  
  
 return NGX\_OK;  
}

此函数的目的也是保存Host头的快速引用，它会对Host头的值做一些合法性检查，并从中解析出域名，保存在headers\_in.server字段，实际上前面在解析请求行时，headers\_in.server可能已经被赋值为从请求行中解析出来的域名，根据http协议的规范，如果请求行中的uri带有域名的话，则域名以它为准，所以这里需检查一下headers\_in.server是否为空，如果不为空则不需要再赋值。

其他请求头的特殊处理函数，不再做介绍，大致都是根据该请求头在http协议中规定的意义及其值设置请求的一些属性，必备后续使用。

对一个合法的请求头的处理大致为如上所述；

2，返回NGX\_AGAIN，表示当前接收到的数据不够，一行请求头还未结束，需要继续下一轮循环。在下一轮循环中，nginx首先检查请求头缓冲区header\_in是否已满，如够满了，则调用ngx\_http\_alloc\_large\_header\_buffer（）函数分配更多缓冲区，下面分析一下ngx\_http\_alloc\_large\_header\_buffer函数：

static ngx\_int\_t  
ngx\_http\_alloc\_large\_header\_buffer(ngx\_http\_request\_t \*r,  
 ngx\_uint\_t request\_line)  
{  
 u\_char \*old, \*new;  
 ngx\_buf\_t \*b;  
 ngx\_http\_connection\_t \*hc;  
 ngx\_http\_core\_srv\_conf\_t \*cscf;  
  
 ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,  
 "http alloc large header buffer");  
  
 /\*  
 \* 在解析请求行阶段，如果客户端在发送请求行之前发送了大量回车换行符将  
 \* 缓冲区塞满了，针对这种情况，nginx只是简单的重置缓冲区，丢弃这些垃圾  
 \* 数据，不需要分配更大的内存。  
 \*/  
 if (request\_line && r->state == 0) {  
  
 /\* the client fills up the buffer with "\r\n" \*/  
  
 r->request\_length += r->header\_in->end - r->header\_in->start;  
  
 r->header\_in->pos = r->header\_in->start;  
 r->header\_in->last = r->header\_in->start;  
  
 return NGX\_OK;  
 }  
  
 /\* 保存请求行或者请求头在旧缓冲区中的起始地址 \*/  
 old = request\_line ? r->request\_start : r->header\_name\_start;  
  
 cscf = ngx\_http\_get\_module\_srv\_conf(r, ngx\_http\_core\_module);  
  
 /\* 如果一个大缓冲区还装不下请求行或者一个请求头，则返回错误 \*/  
 if (r->state != 0  
 && (size\_t) (r->header\_in->pos - old)  
 >= cscf->large\_client\_header\_buffers.size)  
 {  
 return NGX\_DECLINED;  
 }  
  
 hc = r->http\_connection;  
  
 /\* 首先在ngx\_http\_connection\_t结构中查找是否有空闲缓冲区，有的话，直接取之 \*/  
 if (hc->nfree) {  
 b = hc->free[--hc->nfree];  
  
 ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,  
 "http large header free: %p %uz",  
 b->pos, b->end - b->last);  
  
 /\* 检查给该请求分配的请求头缓冲区个数是否已经超过限制，默认最大个数为4个 \*/  
 } else if (hc->nbusy < cscf->large\_client\_header\_buffers.num) {  
  
 if (hc->busy == NULL) {  
 hc->busy = ngx\_palloc(r->connection->pool,  
 cscf->large\_client\_header\_buffers.num \* sizeof(ngx\_buf\_t \*));  
 if (hc->busy == NULL) {  
 return NGX\_ERROR;  
 }  
 }  
  
 /\* 如果还没有达到最大分配数量，则分配一个新的大缓冲区 \*/  
 b = ngx\_create\_temp\_buf(r->connection->pool,  
 cscf->large\_client\_header\_buffers.size);  
 if (b == NULL) {  
 return NGX\_ERROR;  
 }  
  
 ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,  
 "http large header alloc: %p %uz",  
 b->pos, b->end - b->last);  
  
 } else {  
 /\* 如果已经达到最大的分配限制，则返回错误 \*/  
 return NGX\_DECLINED;  
 }  
  
 /\* 将从空闲队列取得的或者新分配的缓冲区加入已使用队列 \*/  
 hc->busy[hc->nbusy++] = b;  
  
 /\*  
 \* 因为nginx中，所有的请求头的保存形式都是指针（起始和结束地址），  
 \* 所以一行完整的请求头必须放在连续的内存块中。如果旧的缓冲区不能  
 \* 再放下整行请求头，则分配新缓冲区，并从旧缓冲区拷贝已经读取的部分请求头，  
 \* 拷贝完之后，需要修改所有相关指针指向到新缓冲区。  
 \* status为0表示解析完一行请求头之后，缓冲区正好被用完，这种情况不需要拷贝  
 \*/  
 if (r->state == 0) {  
 /\*  
 \* r->state == 0 means that a header line was parsed successfully  
 \* and we do not need to copy incomplete header line and  
 \* to relocate the parser header pointers  
 \*/  
  
 r->request\_length += r->header\_in->end - r->header\_in->start;  
  
 r->header\_in = b;  
  
 return NGX\_OK;  
 }  
  
 ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,  
 "http large header copy: %d", r->header\_in->pos - old);  
  
 r->request\_length += old - r->header\_in->start;  
  
 new = b->start;  
  
 /\* 拷贝旧缓冲区中不完整的请求头 \*/  
 ngx\_memcpy(new, old, r->header\_in->pos - old);  
  
 b->pos = new + (r->header\_in->pos - old);  
 b->last = new + (r->header\_in->pos - old);  
  
 /\* 修改相应的指针指向新缓冲区 \*/  
 if (request\_line) {  
 r->request\_start = new;  
  
 if (r->request\_end) {  
 r->request\_end = new + (r->request\_end - old);  
 }  
  
 r->method\_end = new + (r->method\_end - old);  
  
 r->uri\_start = new + (r->uri\_start - old);  
 r->uri\_end = new + (r->uri\_end - old);  
  
 if (r->schema\_start) {  
 r->schema\_start = new + (r->schema\_start - old);  
 r->schema\_end = new + (r->schema\_end - old);  
 }  
  
 if (r->host\_start) {  
 r->host\_start = new + (r->host\_start - old);  
 if (r->host\_end) {  
 r->host\_end = new + (r->host\_end - old);  
 }  
 }  
  
 if (r->port\_start) {  
 r->port\_start = new + (r->port\_start - old);  
 r->port\_end = new + (r->port\_end - old);  
 }  
  
 if (r->uri\_ext) {  
 r->uri\_ext = new + (r->uri\_ext - old);  
 }  
  
 if (r->args\_start) {  
 r->args\_start = new + (r->args\_start - old);  
 }  
  
 if (r->http\_protocol.data) {  
 r->http\_protocol.data = new + (r->http\_protocol.data - old);  
 }  
  
 } else {  
 r->header\_name\_start = new;  
 r->header\_name\_end = new + (r->header\_name\_end - old);  
 r->header\_start = new + (r->header\_start - old);  
 r->header\_end = new + (r->header\_end - old);  
 }  
  
 r->header\_in = b;  
  
 return NGX\_OK;  
}

当ngx\_http\_alloc\_large\_header\_buffer函数返回NGX\_DECLINED时，表示客户端发送了一行过大的请求头，或者是整个请求头部超过了限制，nginx会返回494错误，注意到nginx在返回494错误之前将请求的lingering\_close标识置为了1，这样做的目的是在返回响应之前丢弃掉客户端发过来的其他数据；

3，返回NGX\_HTTP\_PARSE\_INVALID\_HEADER，表示请求头解析过程中遇到错误，一般为客户端发送了不符合协议规范的头部，此时nginx返回400错误；

4，返回NGX\_HTTP\_PARSE\_HEADER\_DONE，表示所有请求头已经成功的解析，这时请求的状态被设置为NGX\_HTTP\_PROCESS\_REQUEST\_STATE，意味着结束了请求读取阶段，正式进入了请求处理阶段，但是实际上请求可能含有请求体，nginx在请求读取阶段并不会去读取请求体，这个工作交给了后续的请求处理阶段的模块，这样做的目的是nginx本身并不知道这些请求体是否有用，如果后续模块并不需要的话，一方面请求体一般较大，如果全部读取进内存，则白白耗费大量的内存空间，另一方面即使nginx将请求体写进磁盘，但是涉及到磁盘io，会耗费比较多时间。所以交由后续模块来决定读取还是丢弃请求体是最明智的办法。

读取完请求头之后，nginx调用了ngx\_http\_process\_request\_header（）函数，这个函数主要做了两个方面的事情，一是调用ngx\_http\_find\_virtual\_server（）函数查找虚拟服务器配置；二是对一些请求头做一些协议的检查。比如对那些使用http1.1协议但是却没有发送Host头的请求，nginx给这些请求返回400错误。还有nginx现在的版本并不支持chunked格式的输入，如果某些请求申明自己使用了chunked格式的输入（请求带有值为chunked的transfer\_encoding头部)，nginx给这些请求返回411错误。等等。

最后调用ngx\_http\_process\_request（）函数处理请求,至此，nginx请求头接收流程就介绍完毕。

### 请求体读取(100%)

上节说到nginx核心本身不会主动读取请求体，这个工作是交给请求处理阶段的模块来做，但是nginx核心提供了ngx\_http\_read\_client\_request\_body()接口来读取请求体，另外还提供了一个丢弃请求体的接口-ngx\_http\_discard\_request\_body()，在请求执行的各个阶段中，任何一个阶段的模块如果对请求体感兴趣或者希望丢掉客户端发过来的请求体，可以分别调用这两个接口来完成。这两个接口是nginx核心提供的处理请求体的标准接口，如果希望配置文件中一些请求体相关的指令（比如client\_body\_in\_file\_only，client\_body\_buffer\_size等）能够预期工作，以及能够正常使用nginx内置的一些和请求体相关的变量（比如$request\_body和$request\_body\_file），一般来说所有模块都必须调用这些接口来完成相应操作，如果需要自定义接口来处理请求体，也应尽量兼容nginx默认的行为。

#### 读取请求体

请求体的读取一般发生在nginx的content handler中，一些nginx内置的模块，比如proxy模块，fastcgi模块，uwsgi模块等，这些模块的行为必须将客户端过来的请求体（如果有的话）以相应协议完整的转发到后端服务进程，所有的这些模块都是调用了ngx\_http\_read\_client\_request\_body()接口来完成请求体读取。值得注意的是这些模块会把客户端的请求体完整的读取后才开始往后端转发数据。

由于内存的限制，ngx\_http\_read\_client\_request\_body()接口读取的请求体会部分或者全部写入一个临时文件中，根据请求体的大小以及相关的指令配置，请求体可能完整放置在一块连续内存中，也可能分别放置在两块不同内存中，还可能全部存在一个临时文件中，最后还可能一部分在内存，剩余部分在临时文件中。下面先介绍一下和这些不同存储行为相关的指令：

client\_body\_buffer\_size

: 设置缓存请求体的buffer大小，默认为系统页大小的2倍，当请求体的大小超过此大小时，nginx会把请求体写入到临时文件中。可以根据业务需求设置合适的大小，尽量避免磁盘io操作;

client\_body\_in\_single\_buffer

: 指示是否将请求体完整的存储在一块连续的内存中，默认为off，如果此指令被设置为on，则nginx会保证请求体在不大于client\_body\_buffer\_size设置的值时，被存放在一块连续的内存中，但超过大小时会被整个写入一个临时文件;

client\_body\_in\_file\_only

: 设置是否总是将请求体保存在临时文件中，默认为off，当此指定被设置为on时，即使客户端显式指示了请求体长度为0时，nginx还是会为请求创建一个临时文件。

接着介绍ngx\_http\_read\_client\_request\_body()接口的实现，它的定义如下：

ngx\_int\_t  
ngx\_http\_read\_client\_request\_body(ngx\_http\_request\_t \*r,  
 ngx\_http\_client\_body\_handler\_pt post\_handler)

该接口有2个参数，第1个为指向请求结构的指针，第2个为一个函数指针，当请求体读完时，它会被调用。之前也说到根据nginx现有行为，模块逻辑会在请求体读完后执行，这个回调函数一般就是模块的逻辑处理函数。ngx\_http\_read\_client\_request\_body()函数首先将参数r对应的主请求的引用加1，这样做的目的和该接口被调用的上下文有关，一般而言，模块是在content handler中调用此接口，一个典型的调用如下：

static ngx\_int\_t  
ngx\_http\_proxy\_handler(ngx\_http\_request\_t \*r)  
{  
 ...  
 rc = ngx\_http\_read\_client\_request\_body(r, ngx\_http\_upstream\_init);  
  
  
 if (rc >= NGX\_HTTP\_SPECIAL\_RESPONSE) {  
 return rc;  
 }  
  
 return NGX\_DONE;  
}

上面的代码是在porxy模块的content handler，ngx\_http\_proxy\_handler()中调用了ngx\_http\_read\_client\_request\_body()函数，其中ngx\_http\_upstream\_init()被作为回调函数传入进接口中，另外nginx中模块的content handler调用的上下文如下：

ngx\_int\_t  
ngx\_http\_core\_content\_phase(ngx\_http\_request\_t \*r,  
 ngx\_http\_phase\_handler\_t \*ph)  
{  
 ...  
 if (r->content\_handler) {  
 r->write\_event\_handler = ngx\_http\_request\_empty\_handler;  
 ngx\_http\_finalize\_request(r, r->content\_handler(r));  
 return NGX\_OK;  
 }  
 ...  
}

上面的代码中，content handler调用之后，它的返回值作为参数调用了ngx\_http\_finalize\_request()函数，在请求体没有被接收完全时，ngx\_http\_read\_client\_request\_body()函数返回值为NGX\_AGAIN，此时content handler，比如ngx\_http\_proxy\_handler()会返回NGX\_DONE，而NGX\_DONE作为参数传给ngx\_http\_finalize\_request()函数会导致主请求的引用计数减1，所以正好抵消了ngx\_http\_read\_client\_request\_body()函数开头对主请求计数的加1。

接下来回到ngx\_http\_read\_client\_request\_body()函数，它会检查该请求的请求体是否已经被读取或者被丢弃了，如果是的话，则直接调用回调函数并返回NGX\_OK，这里实际上是为子请求检查，子请求是nginx中的一个概念，nginx中可以在当前请求中发起另外一个或多个全新的子请求来访问其他的location，关于子请求的具体介绍会在后面的章节作详细分析，一般而言子请求不需要自己去读取请求体。

函数接着调用ngx\_http\_test\_expect()检查客户端是否发送了Expect: 100-continue头，是的话则给客户端回复\"HTTP/1.1 100 Continue\"，根据http 1.1协议，客户端可以发送一个Expect头来向服务器表明期望发送请求体，服务器如果允许客户端发送请求体，则会回复\"HTTP/1.1 100 Continue\"，客户端收到时，才会开始发送请求体。

接着继续为接收请求体做准备工作，分配一个ngx\_http\_request\_body\_t结构，并保存在r->request\_body，这个结构用来保存请求体读取过程用到的缓存引用，临时文件引用，剩余请求体大小等信息，它的定义如下:

typedef struct {  
 ngx\_temp\_file\_t \*temp\_file;  
 ngx\_chain\_t \*bufs;  
 ngx\_buf\_t \*buf;  
 off\_t rest;  
 ngx\_chain\_t \*to\_write;  
 ngx\_http\_client\_body\_handler\_pt post\_handler;  
} ngx\_http\_request\_body\_t;

temp\_file

: 指向储存请求体的临时文件的指针；

bufs

: 指向保存请求体的链表头；

buf

: 指向当前用于保存请求体的内存缓存；

rest

: 当前剩余的请求体大小；

post\_handler

: 保存传给ngx\_http\_read\_client\_request\_body()函数的回调函数。

做好准备工作之后，函数开始检查请求是否带有content\_length头，如果没有该头或者客户端发送了一个值为0的content\_length头，表明没有请求体，这时直接调用回调函数并返回NGX\_OK即可。当然如果client\_body\_in\_file\_only指令被设置为on，且content\_length为0时，该函数在调用回调函数之前，会创建一个空的临时文件。

进入到函数下半部分，表明客户端请求确实表明了要发送请求体，该函数会先检查是否在读取请求头时预读了请求体，这里的检查是通过判断保存请求头的缓存(r->header\_in)中是否还有未处理的数据。如果有预读数据，则分配一个ngx\_buf\_t结构，并将r->header\_in中的预读数据保存在其中，并且如果r->header\_in中还有剩余空间，并且能够容下剩余未读取的请求体，这些空间将被继续使用，而不用分配新的缓存，当然甚至如果请求体已经被整个预读了，则不需要继续处理了，此时调用回调函数后返回。

如果没有预读数据或者预读不完整，该函数会分配一块新的内存（除非r->header\_in还有足够的剩余空间），另外如果request\_body\_in\_single\_buf指令被设置为no，则预读的数据会被拷贝进新开辟的内存块中，真正读取请求体的操作是在ngx\_http\_do\_read\_client\_request\_body()函数，该函数循环的读取请求体并保存在缓存中，如果缓存被写满了，其中的数据会被清空并写回到临时文件中。当然这里有可能不能一次将数据读到，该函数会挂载读事件并设置读事件handler为ngx\_http\_read\_client\_request\_body\_handler，另外nginx核心对两次请求体的读事件之间也做了超时设置，client\_body\_timeout指令可以设置这个超时时间，默认为60秒，如果下次读事件超时了，nginx会返回408给客户端。

最终读完请求体后，ngx\_http\_do\_read\_client\_request\_body()会根据配置，将请求体调整到预期的位置(内存或者文件)，所有情况下请求体都可以从r->request\_body的bufs链表得到，该链表最多可能有2个节点，每个节点为一个buffer，但是这个buffer的内容可能是保存在内存中，也可能是保存在磁盘文件中。另外$request\_body变量只在当请求体已经被读取并且是全部保存在内存中，才能取得相应的数据。

#### 丢弃请求体

一个模块想要主动的丢弃客户端发过的请求体，可以调用nginx核心提供的ngx\_http\_discard\_request\_body()接口，主动丢弃的原因可能有很多种，如模块的业务逻辑压根不需要请求体 ，客户端发送了过大的请求体，另外为了兼容http1.1协议的pipeline请求，模块有义务主动丢弃不需要的请求体。总之为了保持良好的客户端兼容性，nginx必须主动丢弃无用的请求体。下面开始分析ngx\_http\_discard\_request\_body()函数：

ngx\_int\_t  
ngx\_http\_discard\_request\_body(ngx\_http\_request\_t \*r)  
{  
 ssize\_t size;  
 ngx\_event\_t \*rev;  
  
 if (r != r->main || r->discard\_body) {  
 return NGX\_OK;  
 }  
  
 if (ngx\_http\_test\_expect(r) != NGX\_OK) {  
 return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;  
 }  
  
 rev = r->connection->read;  
  
 ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, rev->log, 0, "http set discard body");  
  
 if (rev->timer\_set) {  
 ngx\_del\_timer(rev);  
 }  
  
 if (r->headers\_in.content\_length\_n <= 0 || r->request\_body) {  
 return NGX\_OK;  
 }  
  
 size = r->header\_in->last - r->header\_in->pos;  
  
 if (size) {  
 if (r->headers\_in.content\_length\_n > size) {  
 r->header\_in->pos += size;  
 r->headers\_in.content\_length\_n -= size;  
  
 } else {  
 r->header\_in->pos += (size\_t) r->headers\_in.content\_length\_n;  
 r->headers\_in.content\_length\_n = 0;  
 return NGX\_OK;  
 }  
 }  
  
 r->read\_event\_handler = ngx\_http\_discarded\_request\_body\_handler;  
  
 if (ngx\_handle\_read\_event(rev, 0) != NGX\_OK) {  
 return NGX\_HTTP\_INTERNAL\_SERVER\_ERROR;  
 }  
  
 if (ngx\_http\_read\_discarded\_request\_body(r) == NGX\_OK) {  
 r->lingering\_close = 0;  
  
 } else {  
 r->count++;  
 r->discard\_body = 1;  
 }  
  
 return NGX\_OK;  
}

由于函数不长，这里把它完整的列出来了，函数的开始同样先判断了不需要再做处理的情况：子请求不需要处理，已经调用过此函数的也不需要再处理。接着调用ngx\_http\_test\_expect() 处理http1.1 expect的情况，根据http1.1的expect机制，如果客户端发送了expect头，而服务端不希望接收请求体时，必须返回417(Expectation Failed)错误。nginx并没有这样做，它只是简单的让客户端把请求体发送过来，然后丢弃掉。接下来，函数删掉了读事件上的定时器，因为这时本身就不需要请求体，所以也无所谓客户端发送的快还是慢了，当然后面还会讲到，当nginx已经处理完该请求但客户端还没有发送完无用的请求体时，nginx会在读事件上再挂上定时器。

客户端如果打算发送请求体，就必须发送content-length头，所以函数会检查请求头中的content-length头，同时还会查看其他地方是不是已经读取了请求体。如果确实有待处理的请求体，函数接着检查请求头buffer中预读的数据，预读的数据会直接被丢掉，当然如果请求体已经被全部预读，函数就直接返回了。

接下来，如果还有剩余的请求体未处理，该函数调用ngx\_handle\_read\_event()在事件处理机制中挂载好读事件，并把读事件的处理函数设置为ngx\_http\_discarded\_request\_body\_handler。做好这些准备之后，该函数最后调用ngx\_http\_read\_discarded\_request\_body()接口读取客户端过来的请求体并丢弃。如果客户端并没有一次将请求体发过来，函数会返回，剩余的数据等到下一次读事件过来时，交给ngx\_http\_discarded\_request\_body\_handler()来处理，这时，请求的discard\_body将被设置为1用来标识这种情况。另外请求的引用数(count)也被加1，这样做的目的是客户端可能在nginx处理完请求之后仍未完整发送待发送的请求体，增加引用是防止nginx核心在处理完请求后直接释放了请求的相关资源。

ngx\_http\_read\_discarded\_request\_body()函数非常简单，它循环的从链接中读取数据并丢弃，直到读完接收缓冲区的所有数据，如果请求体已经被读完了，该函数会设置读事件的处理函数为ngx\_http\_block\_reading，这个函数仅仅删除水平触发的读事件，防止同一事件不断被触发。

最后看一下读事件的处理函数ngx\_http\_discarded\_request\_body\_handler，这个函数每次读事件来时会被调用，先看一下它的源码：

void  
ngx\_http\_discarded\_request\_body\_handler(ngx\_http\_request\_t \*r)  
{  
 ...  
  
 c = r->connection;  
 rev = c->read;  
  
 if (rev->timedout) {  
 c->timedout = 1;  
 c->error = 1;  
 ngx\_http\_finalize\_request(r, NGX\_ERROR);  
 return;  
 }  
  
 if (r->lingering\_time) {  
 timer = (ngx\_msec\_t) (r->lingering\_time - ngx\_time());  
  
 if (timer <= 0) {  
 r->discard\_body = 0;  
 r->lingering\_close = 0;  
 ngx\_http\_finalize\_request(r, NGX\_ERROR);  
 return;  
 }  
  
 } else {  
 timer = 0;  
 }  
  
 rc = ngx\_http\_read\_discarded\_request\_body(r);  
  
 if (rc == NGX\_OK) {  
 r->discard\_body = 0;  
 r->lingering\_close = 0;  
 ngx\_http\_finalize\_request(r, NGX\_DONE);  
 return;  
 }  
  
 /\* rc == NGX\_AGAIN \*/  
  
 if (ngx\_handle\_read\_event(rev, 0) != NGX\_OK) {  
 c->error = 1;  
 ngx\_http\_finalize\_request(r, NGX\_ERROR);  
 return;  
 }  
  
 if (timer) {  
  
 clcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_core\_module);  
  
 timer \*= 1000;  
  
 if (timer > clcf->lingering\_timeout) {  
 timer = clcf->lingering\_timeout;  
 }  
  
 ngx\_add\_timer(rev, timer);  
 }  
}

函数一开始就处理了读事件超时的情况，之前说到在ngx\_http\_discard\_request\_body()函数中已经删除了读事件的定时器，那么什么时候会设置定时器呢？答案就是在nginx已经处理完该请求，但是又没有完全将该请求的请求体丢弃的时候（客户端可能还没有发送过来），在ngx\_http\_finalize\_connection()函数中，如果检查到还有未丢弃的请求体时，nginx会添加一个读事件定时器，它的时长为lingering\_timeout指令所指定，默认为5秒，不过这个时间仅仅两次读事件之间的超时时间，等待请求体的总时长为lingering\_time指令所指定，默认为30秒。这种情况中，该函数如果检测到超时事件则直接返回并断开连接。同样，还需要控制整个丢弃请求体的时长不能超过lingering\_time设置的时间，如果超过了最大时长，也会直接返回并断开连接。

如果读事件发生在请求处理完之前，则不用处理超时事件，也不用设置定时器，函数只是简单的调用ngx\_http\_read\_discarded\_request\_body()来读取并丢弃数据。

## 多阶段处理请求

读取完请求头后，nginx进入请求的处理阶段。简单的情况下，客户端发送过的统一资源定位符(url)对应服务器上某一路径上的资源，web服务器需要做的仅仅是将url映射到本地文件系统的路径，然后读取相应文件并返回给客户端。但这仅仅是最初的互联网的需求，而如今互联网出现了各种各样复杂的需求，要求web服务器能够处理诸如安全及权限控制，多媒体内容和动态网页等等问题。这些复杂的需求导致web服务器不再是一个短小的程序，而变成了一个必须经过仔细设计，模块化的系统。nginx良好的模块化特性体现在其对请求处理流程的多阶段划分当中，多阶段处理流程就好像一条流水线，一个nginx进程可以并发的处理处于不同阶段的多个请求。nginx允许开发者在处理流程的任意阶段注册模块，在启动阶段，nginx会把各个阶段注册的所有模块处理函数按序的组织成一条执行链。

nginx实际把请求处理流程划分为了11个阶段，这样划分的原因是将请求的执行逻辑细分，各阶段按照处理时机定义了清晰的执行语义，开发者可以很容易分辨自己需要开发的模块应该定义在什么阶段，下面介绍一下各阶段:

NGX\_HTTP\_POST\_READ\_PHASE

: 接收完请求头之后的第一个阶段，它位于uri重写之前，实际上很少有模块会注册在该阶段，默认的情况下，该阶段被跳过；

NGX\_HTTP\_SERVER\_REWRITE\_PHASE

: server级别的uri重写阶段，也就是该阶段执行处于server块内，location块外的重写指令，前面的章节已经说明在读取请求头的过程中nginx会根据host及端口找到对应的虚拟主机配置；

NGX\_HTTP\_FIND\_CONFIG\_PHASE

: 寻找location配置阶段，该阶段使用重写之后的uri来查找对应的location，值得注意的是该阶段可能会被执行多次，因为也可能有location级别的重写指令；

NGX\_HTTP\_REWRITE\_PHASE

: location级别的uri重写阶段，该阶段执行location基本的重写指令，也可能会被执行多次；

NGX\_HTTP\_POST\_REWRITE\_PHASE

: location级别重写的后一阶段，用来检查上阶段是否有uri重写，并根据结果跳转到合适的阶段；

NGX\_HTTP\_PREACCESS\_PHASE

: 访问权限控制的前一阶段，该阶段在权限控制阶段之前，一般也用于访问控制，比如限制访问频率，链接数等；

NGX\_HTTP\_ACCESS\_PHASE

: 访问权限控制阶段，比如基于ip黑白名单的权限控制，基于用户名密码的权限控制等；

NGX\_HTTP\_POST\_ACCESS\_PHASE

: 访问权限控制的后一阶段，该阶段根据权限控制阶段的执行结果进行相应处理；

NGX\_HTTP\_TRY\_FILES\_PHASE

: try\_files指令的处理阶段，如果没有配置try\_files指令，则该阶段被跳过；

NGX\_HTTP\_CONTENT\_PHASE

: 内容生成阶段，该阶段产生响应，并发送到客户端；

NGX\_HTTP\_LOG\_PHASE

: 日志记录阶段，该阶段记录访问日志。

### 多阶段执行链

nginx按请求处理的执行顺序将处理流程划分为多个阶段，一般每个阶段又可以注册多个模块处理函数，nginx按阶段将这些处理函数组织成了一个执行链，这个执行链保存在http主配置（ngx\_http\_core\_main\_conf\_t）的phase\_engine字段中，phase\_engine字段的类型为ngx\_http\_phase\_engine\_t：

typedef struct {  
 ngx\_http\_phase\_handler\_t \*handlers;  
 ngx\_uint\_t server\_rewrite\_index;  
 ngx\_uint\_t location\_rewrite\_index;  
} ngx\_http\_phase\_engine\_t;

其中handlers字段即为执行链，实际上它是一个数组，而每个元素之间又被串成链表，从而允许执行流程向前，或者向后的阶段跳转，执行链节点的数据结构定义如下：

struct ngx\_http\_phase\_handler\_s {  
 ngx\_http\_phase\_handler\_pt checker;  
 ngx\_http\_handler\_pt handler;  
 ngx\_uint\_t next;  
};

其中checker和handler都是函数指针，相同阶段的节点具有相同的checker函数，handler字段保存的是模块处理函数，一般在checker函数中会执行当前节点的handler函数，但是例外的是NGX\_HTTP\_FIND\_CONFIG\_PHASE，NGX\_HTTP\_POST\_REWRITE\_PHASE，NGX\_HTTP\_POST\_ACCESS\_PHASE和NGX\_HTTP\_TRY\_FILES\_PHASE这4个阶段不能注册模块函数。next字段为快速跳跃索引，多数情况下，执行流程是按照执行链顺序的往前执行，但在某些执行阶段的checker函数中由于执行了某个逻辑可能需要回跳至之前的执行阶段，也可能需要跳过之后的某些执行阶段，next字段保存的就是跳跃的目的索引。

和建立执行链相关的数据结构都保存在http主配置中，一个是phases字段，另外一个是phase\_engine字段。其中phases字段为一个数组，它的元素个数等于阶段数目，即每个元素对应一个阶段。而phases数组的每个元素又是动态数组（ngx\_array\_t），每次模块注册处理函数时只需要在对应阶段的动态数组增加一个元素用来保存处理函数的指针。由于在某些执行阶段可能需要向后，或者向前跳转，简单的使用2个数组并不方便，所以nginx又组织了一个执行链，保存在了phase\_engine字段，其每个节点包含一个next域用来保存跳跃目的节点的索引，而执行链的建立则在nginx初始化的post config阶段之后调用ngx\_http\_init\_phase\_handlers函数完成，下面分析一下该函数：

static ngx\_int\_t  
ngx\_http\_init\_phase\_handlers(ngx\_conf\_t \*cf, ngx\_http\_core\_main\_conf\_t \*cmcf)  
{  
 ngx\_int\_t j;  
 ngx\_uint\_t i, n;  
 ngx\_uint\_t find\_config\_index, use\_rewrite, use\_access;  
 ngx\_http\_handler\_pt \*h;  
 ngx\_http\_phase\_handler\_t \*ph;  
 ngx\_http\_phase\_handler\_pt checker;  
  
 cmcf->phase\_engine.server\_rewrite\_index = (ngx\_uint\_t) -1;  
 cmcf->phase\_engine.location\_rewrite\_index = (ngx\_uint\_t) -1;  
 find\_config\_index = 0;  
 use\_rewrite = cmcf->phases[NGX\_HTTP\_REWRITE\_PHASE].handlers.nelts ? 1 : 0;  
 use\_access = cmcf->phases[NGX\_HTTP\_ACCESS\_PHASE].handlers.nelts ? 1 : 0;  
  
 n = use\_rewrite + use\_access + cmcf->try\_files + 1 /\* find config phase \*/;  
  
 for (i = 0; i < NGX\_HTTP\_LOG\_PHASE; i++) {  
 n += cmcf->phases[i].handlers.nelts;  
 }  
  
 ph = ngx\_pcalloc(cf->pool,  
 n \* sizeof(ngx\_http\_phase\_handler\_t) + sizeof(void \*));  
 if (ph == NULL) {  
 return NGX\_ERROR;  
 }  
  
 cmcf->phase\_engine.handlers = ph;  
 n = 0;  
  
 for (i = 0; i < NGX\_HTTP\_LOG\_PHASE; i++) {  
 h = cmcf->phases[i].handlers.elts;  
  
 switch (i) {  
  
 case NGX\_HTTP\_SERVER\_REWRITE\_PHASE:  
 if (cmcf->phase\_engine.server\_rewrite\_index == (ngx\_uint\_t) -1) {  
 cmcf->phase\_engine.server\_rewrite\_index = n;  
 }  
 checker = ngx\_http\_core\_rewrite\_phase;  
  
 break;  
  
 case NGX\_HTTP\_FIND\_CONFIG\_PHASE:  
 find\_config\_index = n;  
  
 ph->checker = ngx\_http\_core\_find\_config\_phase;  
 n++;  
 ph++;  
  
 continue;  
  
 case NGX\_HTTP\_REWRITE\_PHASE:  
 if (cmcf->phase\_engine.location\_rewrite\_index == (ngx\_uint\_t) -1) {  
 cmcf->phase\_engine.location\_rewrite\_index = n;  
 }  
 checker = ngx\_http\_core\_rewrite\_phase;  
  
 break;  
  
 case NGX\_HTTP\_POST\_REWRITE\_PHASE:  
 if (use\_rewrite) {  
 ph->checker = ngx\_http\_core\_post\_rewrite\_phase;  
 ph->next = find\_config\_index;  
 n++;  
 ph++;  
 }  
  
 continue;  
  
 case NGX\_HTTP\_ACCESS\_PHASE:  
 checker = ngx\_http\_core\_access\_phase;  
 n++;  
 break;  
  
 case NGX\_HTTP\_POST\_ACCESS\_PHASE:  
 if (use\_access) {  
 ph->checker = ngx\_http\_core\_post\_access\_phase;  
 ph->next = n;  
 ph++;  
 }  
  
 continue;  
  
 case NGX\_HTTP\_TRY\_FILES\_PHASE:  
 if (cmcf->try\_files) {  
 ph->checker = ngx\_http\_core\_try\_files\_phase;  
 n++;  
 ph++;  
 }  
  
 continue;  
  
 case NGX\_HTTP\_CONTENT\_PHASE:  
 checker = ngx\_http\_core\_content\_phase;  
 break;  
  
 default:  
 checker = ngx\_http\_core\_generic\_phase;  
 }  
  
 n += cmcf->phases[i].handlers.nelts;  
  
 for (j = cmcf->phases[i].handlers.nelts - 1; j >=0; j--) {  
 ph->checker = checker;  
 ph->handler = h[j];  
 ph->next = n;  
 ph++;  
 }  
 }  
  
 return NGX\_OK;  
}

首先需要说明的是cmcf->phases数组中保存了在post config之前注册的所有模块函数，上面的函数先计算执行链的节点个数，并分配相应的空间，前面提到有4个阶段不能注册模块，并且POST\_REWRITE和POST\_ACCESS这2个阶段分别只有在REWRITE和ACCESS阶段注册了模块时才存在，另外TRY\_FILES阶段只有在配置了try\_files指令的时候才存在，最后FIND\_CONFIG阶段虽然不能注册模块，但它是必须存在的，所以在计算执行链节点数时需要考虑这些因素。

分配好内存之后，开始建立链表，过程很简单，遍历每个阶段注册的模块函数，为每个阶段的节点赋值checker函数，handler函数，以及next索引。最终建立好的执行链如下图：

(暂缺)

SERVER\_REWRITE阶段的节点的next域指向FIND\_CONFIG阶段的第1个节点，REWRITE阶段的next域指向POST\_REWRITE阶段的第1个节点，而POST\_REWRITE阶段的next域则指向FIND\_CONFIG，因为当出现location级别的uri重写时，可能需要重新匹配新的location，PREACCESS阶段的next域指向ACCESS域，ACCESS和POST\_ACCESS阶段的next域则是则是指向CONTENT阶段，当然如果TRY\_FILES阶段存在的话，则是指向TRY\_FILES阶段，最后CONTENT阶段的next域指向LOG阶段，当然next域是每个阶段的checker函数根据该阶段的需求来使用的，没有需要时，checker函数可能都不会使用到它。

### POST\_READ阶段

POST\_READ阶段是nginx处理请求流程中第一个可以添加模块函数的阶段，任何需要在接收完请求头之后立刻处理的逻辑可以在该阶段注册处理函数。nginx源码中只有realip模块在该阶段注册了函数，当nginx前端多了一个7层负载均衡层，并且客户端的真实ip被前端保存在请求头中时，该模块用来将客户端的ip替换为请求头中保存的值。realip模块之所以在POST\_READ阶段执行的原因是它需要在其他模块执行之前悄悄的将客户端ip替换为真实值，而且它需要的信息仅仅只是请求头。一般很少有模块需要注册在POST\_READ阶段，realip模块默认没有编译进nginx。

POST\_READ阶段的checker函数是ngx\_http\_core\_generic\_phase，这个函数是nginx phase默认的checker函数，后面的PREACCESS phase也是用checker，下面对它做一下介绍：

ngx\_int\_t  
ngx\_http\_core\_generic\_phase(ngx\_http\_request\_t \*r, ngx\_http\_phase\_handler\_t \*ph)  
{  
 ngx\_int\_t rc;  
  
 /\*  
 \* generic phase checker,  
 \* used by the post read and pre-access phases  
 \*/  
  
 ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,  
 "generic phase: %ui", r->phase\_handler);  
  
 rc = ph->handler(r);  
  
 if (rc == NGX\_OK) {  
 r->phase\_handler = ph->next;  
 return NGX\_AGAIN;  
 }  
  
 if (rc == NGX\_DECLINED) {  
 r->phase\_handler++;  
 return NGX\_AGAIN;  
 }  
  
 if (rc == NGX\_AGAIN || rc == NGX\_DONE) {  
 return NGX\_OK;  
 }  
  
 /\* rc == NGX\_ERROR || rc == NGX\_HTTP\_... \*/  
  
 ngx\_http\_finalize\_request(r, rc);  
  
 return NGX\_OK;  
}

这个函数逻辑非常简单，调用该phase注册的handler函数，需要注意的是该函数对handler返回值的处理，一般而言handler返回：

NGX\_OK

: 表示该阶段已经处理完成，需要转入下一个阶段；

NG\_DECLINED

: 表示需要转入本阶段的下一个handler继续处理；

NGX\_AGAIN, NGX\_DONE

: 表示需要等待某个事件发生才能继续处理（比如等待网络IO），此时Nginx为了不阻塞其他请求的处理，必须中断当前请求的执行链，等待事件发生之后继续执行该handler；

NGX\_ERROR

: 表示发生了错误，需要结束该请求。

checker函数根据handler函数的不同返回值，给上一层的ngx\_http\_core\_run\_phases函数返回NGX\_AGAIN或者NGX\_OK，如果期望上一层继续执行后面的phase则需要确保checker函数不是返回NGX\_OK，不同checker函数对handler函数的返回值处理还不太一样，开发模块时需要确保相应阶段的checker函数对返回值的处理在你的预期之内。

### SERVER\_REWRITE阶段

SERVER\_REWRITE阶段是nginx中第一个必须经历的重要phase，请求进入此阶段时已经找到对应的虚拟主机（server）配置。nginx的rewrite模块在这个阶段注册了一个handler，rewrite模块提供url重写指令rewrite，变量设置指令set，以及逻辑控制指令if、break和return，用户可以在server配置里面，组合这些指令来满足自己的需求，而不需要另外写一个模块，比如将一些前缀满足特定模式的uri重定向到一个固定的url，还可以根据请求的属性来决定是否需要重写或者给用户发送特定的返回码。rewrite提供的逻辑控制指令能够满足一些简单的需求，针对一些较复杂的逻辑可能需要注册handler通过独立实现模块的方式来满足。

需要注意该阶段和后面的REWRITE阶段的区别，在SERVER\_REWRITE阶段中，请求还未被匹配到一个具体的location中。该阶段执行的结果（比如改写后的uri）会影响后面FIND\_CONFIG阶段的执行。另外这个阶段也是内部子请求执行的第一个阶段。 SERVER\_REWRITE阶段的checker函数是ngx\_http\_core\_rewrite\_phase：

ngx\_int\_t  
ngx\_http\_core\_rewrite\_phase(ngx\_http\_request\_t \*r, ngx\_http\_phase\_handler\_t \*ph)  
{  
 ngx\_int\_t rc;  
  
 ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,  
 "rewrite phase: %ui", r->phase\_handler);  
  
 rc = ph->handler(r);  
  
 if (rc == NGX\_DECLINED) {  
 r->phase\_handler++;  
 return NGX\_AGAIN;  
 }  
  
 if (rc == NGX\_DONE) {  
 return NGX\_OK;  
 }  
  
 /\* NGX\_OK, NGX\_AGAIN, NGX\_ERROR, NGX\_HTTP\_... \*/  
  
 ngx\_http\_finalize\_request(r, rc);  
  
 return NGX\_OK;  
}

这个函数和上面说的ngx\_http\_core\_generic\_phase函数流程基本一致，唯一的区别就是对handler返回值的处理稍有不同，比如这里对NGX\_OK的处理是调用ngx\_http\_finalize\_request结束请求，所以再强调一下，handler函数的返回值一定要根据不同phase的checker函数来设置。Nginx的rewrite模块会挂上一个名为ngx\_http\_rewrite\_handler的handler。

### FIND\_CONFIG阶段

FIND\_CONFIG阶段顾名思义就是寻找配置阶段，具体一点就是根据uri查找location配置，实际上就是设置r->loc\_conf，在此之前r->loc\_conf使用的server级别的，查找location过程由函数ngx\_http\_core\_find\_location完成，具体查找流流程这里不再赘述，可以参考上一章关于location管理的内容，值得注意的是当ngx\_http\_core\_find\_location函数返回NGX\_DONE时，Nginx会返回301，将用户请求做一个重定向，这种情况仅发生在该location使用了proxy\_pass/fastcgi/scgi/uwsgi/memcached模块，且location的名字以/符号结尾，并且请求的uri为该location除/之外的前缀，比如对location /xx/，如果某个请求/xx访问到该location，则会被重定向为/xx/。另外Nginx中location可以标识为internal，即内部location，这种location只能由子请求或者内部跳转访问。

找到location配置后，Nginx调用了ngx\_http\_update\_location\_config函数来更新请求相关配置，其中最重要的是更新请求的content handler，不同location可以有自己的content handler。

最后，由于有REWRITE\_PHASE的存在，FIND\_CONFIG阶段可能会被执行多次。

### REWRITE阶段

REWRITE阶段为location级别的重写，这个阶段的checker和SERVER\_REWRITE阶段的是同一个函数，而且Nginx的rewrite模块对这2个阶段注册的是同一个handler，2者唯一区别就是执行时机不一样，REWRITE阶段为location级别的重写，SERVER\_REWRITE执行之后是FIND\_CONFIG阶段，REWRITE阶段执行之后是POST\_REWRITE阶段。

### POST\_REWRITE阶段

该阶段不能注册handler，仅仅只是检查上一阶段是否做了uri重写，如果没有重写的话，直接进入下一阶段；如果有重写的话，则利用next跳转域往前跳转到FIND\_CONFIG阶段重新执行。Nginx对uri重写次数做了限制，默认是10次。

### PREACCESS阶段

进入该阶段表明Nginx已经将请求确定到了某一个location(当该server没有任何location时，也可能是server），如论如何请求的loc\_conf配置已经确定下来，该阶段一般用来做资源控制，默认情况下，诸如ngx\_http\_limit\_conn\_module，ngx\_http\_limit\_req\_module等模块会在该阶段注册handler，用于控制连接数，请求速率等。PREACCESS阶段使用的checker是默认的ngx\_http\_core\_generic\_phase函数。

### ACCESS阶段

该阶段的首要目的是做权限控制，默认情况下，Nginx的ngx\_http\_access\_module和ngx\_http\_auth\_basic\_module模块分别会在该阶段注册一个handler。

ACCESS阶段的checker是ngx\_http\_core\_access\_phase函数，此函数对handler返回值的处理大致和ngx\_http\_core\_generic\_phase一致，特殊的地方是当clcf->satisfy为NGX\_HTTP\_SATISFY\_ALL，也就是需要满足该阶段注册的所有handler的验证时，某个handler返回NGX\_OK时还需要继续处理本阶段的其他handler。clcf->satisfy的值可以使用satisfy指令指定。

### POST\_ACCESS阶段

POST\_ACCESS和POST\_REWRITE阶段一样，只是处理一下上一阶段的结果，而不能挂载自己的handler，具体为如果ACCESS阶段返回了NGX\_HTTP\_FORBIDDEN或NGX\_HTTP\_UNAUTHORIZED（记录在r->access\_code字段），该阶段会结束掉请求。

### TRY\_FILES阶段

TRY\_FILES阶段仅当配置了try\_files指令时生效，实际上该指令不常用，它的功能是指定一个或者多个文件或目录，最后一个参数可以指定为一个location或一个返回码，当设置了该指令时，TRY\_FILES阶段调用checker函数ngx\_http\_core\_try\_files\_phase来依此检查指定的文件或目录是否存在，如果本地文件系统存在某个文件或目录则退出该阶段继续执行下面的阶段，否则内部重定向到最后一个参数指定的location或返回指定的返回码。

该阶段也不能注册handler。

### CONTENT阶段

CONTENT阶段可以说是整个执行链中最重要的阶段，请求从这里开始执行业务逻辑并产生响应，下面来分析一下它的checker函数：

ngx\_int\_t  
ngx\_http\_core\_content\_phase(ngx\_http\_request\_t \*r,  
 ngx\_http\_phase\_handler\_t \*ph)  
{  
 size\_t root;  
 ngx\_int\_t rc;  
 ngx\_str\_t path;  
  
 if (r->content\_handler) {  
 r->write\_event\_handler = ngx\_http\_request\_empty\_handler;  
 ngx\_http\_finalize\_request(r, r->content\_handler(r));  
 return NGX\_OK;  
 }  
  
 ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, r->connection->log, 0,  
 "content phase: %ui", r->phase\_handler);  
  
 rc = ph->handler(r);  
  
 if (rc != NGX\_DECLINED) {  
 ngx\_http\_finalize\_request(r, rc);  
 return NGX\_OK;  
 }  
  
 /\* rc == NGX\_DECLINED \*/  
  
 ph++;  
  
 if (ph->checker) {  
 r->phase\_handler++;  
 return NGX\_AGAIN;  
 }  
  
 /\* no content handler was found \*/  
  
 if (r->uri.data[r->uri.len - 1] == '/') {  
  
 if (ngx\_http\_map\_uri\_to\_path(r, &path, &root, 0) != NULL) {  
 ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, 0,  
 "directory index of \"%s\" is forbidden", path.data);  
 }  
  
 ngx\_http\_finalize\_request(r, NGX\_HTTP\_FORBIDDEN);  
 return NGX\_OK;  
 }  
  
 ngx\_log\_error(NGX\_LOG\_ERR, r->connection->log, 0, "no handler found");  
  
 ngx\_http\_finalize\_request(r, NGX\_HTTP\_NOT\_FOUND);  
 return NGX\_OK;  
}

CONTENT阶段有些特殊，它不像其他阶段只能执行固定的handler链，还有一个特殊的content\_handler，每个location可以有自己独立的content handler，而且当有content handler时，CONTENT阶段只会执行content handler，不再执行本阶段的handler链。

默认情况下，Nginx会在CONTENT阶段的handler链挂上index模块，静态文件处理模块等的handler。另外模块还可以设置独立的content handler，比如ngx\_http\_proxy\_module的proxy\_pass指令会设置一个名为ngx\_http\_proxy\_handler的content handler。

接下来看一下上面的checker函数的执行流程，首先检查是否设置了r->content\_handler，如果设置了的话，则执行它，需要注意的是在执行它之前，Nginx将r->write\_event\_handler设置为了ngx\_http\_request\_empty\_handler，先看一下设置r->write\_event\_handler之前的值是什么，在ngx\_http\_handler函数中它被设置为ngx\_http\_core\_run\_phases，而ngx\_http\_core\_run\_phases会运行每个阶段的checker函数。正常流程中，如果某个阶段需要等待某个写事件发生时，该阶段的handler会返回NGX\_OK来中断ngx\_http\_core\_run\_phases的运行，等到下次写事件过来时，会继续执行之前阶段的handler；当执行r->content\_handler的流程时，Nginx默认模块会去处理r->write\_event\_handler的值，也就是假设r->content\_handler只能执行1次，如果模块设置的content handler涉及到IO操作，就需要合理的设置处理读写事件的handler（r->read\_event\_handler和r->write\_event\_handler）。

还有一个需要注意的点是r->content\_handler执行之后，Nginx直接用其返回值调用了ngx\_http\_finalize\_request函数，Nginx将一大堆耦合的逻辑都集中在了这个函数当中，包括长连接，lingering\_close，子请求等的处理都涉及到该函数，后面会有一节单独介绍这个函数。这里需要提醒的是r->content\_handler如果并未完成整个请求的处理，而只是需要等待某个事件发生而退出处理流程的话，必须返回一个合适的值传给ngx\_http\_finalize\_request，一般而言是返回NGX\_DONE，而且需要将请求的引用计数（r->count）加1，确保ngx\_http\_finalize\_request函数不会将该请求释放掉。

函数的其他部分处理走handler链的情况，特殊的地方是CONTENT阶段是ngx\_http\_core\_run\_phases函数跑的最后一个阶段，如果最后一个handler返回NGX\_DECLINED，此时Nginx会给客户端返回NGX\_HTTP\_FORBIDDEN（403）或NGX\_HTTP\_NOT\_FOUND（404）。

### LOG阶段

LOG阶段主要的目的就是记录访问日志，进入该阶段表明该请求的响应已经发送到系统发送缓冲区。另外这个阶段的handler链实际上并不是在ngx\_http\_core\_run\_phases函数中执行，而是在释放请求资源的ngx\_http\_free\_request函数中运行，这样做的原因实际是为了简化流程，因为ngx\_http\_core\_run\_phases可能会执行多次，而LOG阶段只需要再请求所有逻辑都结束时运行一次，所以在ngx\_http\_free\_request函数中运行LOG阶段的handler链是非常好的选择。具体的执行的函数为ngx\_http\_log\_request：

static void  
ngx\_http\_log\_request(ngx\_http\_request\_t \*r)  
{  
 ngx\_uint\_t i, n;  
 ngx\_http\_handler\_pt \*log\_handler;  
 ngx\_http\_core\_main\_conf\_t \*cmcf;  
  
 cmcf = ngx\_http\_get\_module\_main\_conf(r, ngx\_http\_core\_module);  
  
 log\_handler = cmcf->phases[NGX\_HTTP\_LOG\_PHASE].handlers.elts;  
 n = cmcf->phases[NGX\_HTTP\_LOG\_PHASE].handlers.nelts;  
  
 for (i = 0; i < n; i++) {  
 log\_handler[i](r);  
 }  
}

函数非常简单，仅仅是遍历LOG阶段的handler链，逐一执行，而且不会检查返回值。LOG阶段和其他阶段的不同点有两个，一是执行点是在ngx\_http\_free\_request中，二是这个阶段的所有handler都会被执行。

至此，Nginx请求处理的多阶段执行链的各个阶段都已经介绍完毕，弄清楚每个阶段的执行时机以及每个阶段的不同特点对写模块非常重要。

## Nginx filter

在CONTENT阶段产生的数据被发往客户端（系统发送缓存区）之前，会先经过过滤。Nginx的filter的工作方式和做鱼有些类似。比如一条鱼，可以把它切成鱼片（也可以切块，切泥），然后通过不同的烹饪方法就得到水煮鱼或者日式生鱼片或者废了等等。同样是一条鱼，加工得到的结果却截然不同，就是因为中间不同的工序赋予了这条鱼各种属性。Nginx的filter也是一个道理，前面的Handler好比这条鱼，filter负责加工，最后得到的HTTP响应就会各种各样，格式可以是JSON或者YAML，内容可能多一些或者少一些，HTTP属性可各异，可以选择压缩，甚至内容可以被丢弃。

对应HTTP请求的响应头和响应体，Nginx分别设置了header filter和body filter。两种机制都是采用链表的方式，不同过滤模块对应链表的一个节点，一般而言一个模块会同时注册header filter和body filter。一个典型的filter模块，比如gzip模块使用类似如下的代码来注册：

static ngx\_http\_output\_header\_filter\_pt ngx\_http\_next\_header\_filter;  
static ngx\_http\_output\_body\_filter\_pt ngx\_http\_next\_body\_filter;  
  
...  
  
static ngx\_int\_t  
ngx\_http\_gzip\_filter\_init(ngx\_conf\_t \*cf)  
{  
 ngx\_http\_next\_header\_filter = ngx\_http\_top\_header\_filter;  
 ngx\_http\_top\_header\_filter = ngx\_http\_gzip\_header\_filter;  
  
 ngx\_http\_next\_body\_filter = ngx\_http\_top\_body\_filter;  
 ngx\_http\_top\_body\_filter = ngx\_http\_gzip\_body\_filter;  
  
 return NGX\_OK;  
}

上面的代码中，gzip模块首先在模块的开头声明了两个static类型的全局变量ngx\_http\_next\_header\_filter和ngx\_http\_next\_body\_filter，在ngx\_http\_gzip\_filter\_init函数中，这二个变量分别被赋值为ngx\_http\_top\_header\_filter及ngx\_http\_top\_body\_filter。而后二者定义在ngx\_http.c，并在ngx\_http.h头文件中被导出。ngx\_http\_top\_header\_filter和ngx\_http\_top\_body\_filter实际上是filter链表的头结点，每次注册一个新的filter模块时，它们的值先被保存在新模块的内部全局变量ngx\_http\_next\_header\_filter及ngx\_http\_next\_body\_filter，然后被赋值为新模块注册的filter函数，而且Nginx filter是先从头节点开始执行，所以越晚注册的模块越早执行。

采用默认编译选项，Nginx默认编译的模块如下：

ngx\_module\_t \*ngx\_modules[] = {  
 &ngx\_core\_module,  
 &ngx\_errlog\_module,  
 &ngx\_conf\_module,  
 &ngx\_events\_module,  
 &ngx\_event\_core\_module,  
 &ngx\_epoll\_module,  
 &ngx\_regex\_module,  
 &ngx\_http\_module,  
 &ngx\_http\_core\_module,  
 &ngx\_http\_log\_module,  
 &ngx\_http\_upstream\_module,  
 &ngx\_http\_static\_module,  
 &ngx\_http\_autoindex\_module,  
 &ngx\_http\_index\_module,  
 &ngx\_http\_auth\_basic\_module,  
 &ngx\_http\_access\_module,  
 &ngx\_http\_limit\_conn\_module,  
 &ngx\_http\_limit\_req\_module,  
 &ngx\_http\_geo\_module,  
 &ngx\_http\_map\_module,  
 &ngx\_http\_split\_clients\_module,  
 &ngx\_http\_referer\_module,  
 &ngx\_http\_rewrite\_module,  
 &ngx\_http\_proxy\_module,  
 &ngx\_http\_fastcgi\_module,  
 &ngx\_http\_uwsgi\_module,  
 &ngx\_http\_scgi\_module,  
 &ngx\_http\_memcached\_module,  
 &ngx\_http\_empty\_gif\_module,  
 &ngx\_http\_browser\_module,  
 &ngx\_http\_upstream\_ip\_hash\_module,  
 &ngx\_http\_upstream\_keepalive\_module,  
 &ngx\_http\_write\_filter\_module, /\* 最后一个body filter，负责往外发送数据 \*/  
 &ngx\_http\_header\_filter\_module, /\* 最后一个header filter，负责在内存中拼接出完整的http响应头，  
 并调用ngx\_http\_write\_filter发送 \*/  
 &ngx\_http\_chunked\_filter\_module, /\* 对响应头中没有content\_length头的请求，强制短连接（低于http 1.1）  
 或采用chunked编码（http 1.1) \*/  
 &ngx\_http\_range\_header\_filter\_module, /\* header filter，负责处理range头 \*/  
 &ngx\_http\_gzip\_filter\_module, /\* 支持流式的数据压缩 \*/  
 &ngx\_http\_postpone\_filter\_module, /\* body filter，负责处理子请求和主请求数据的输出顺序 \*/  
 &ngx\_http\_ssi\_filter\_module, /\* 支持过滤SSI请求，采用发起子请求的方式，去获取include进来的文件 \*/  
 &ngx\_http\_charset\_filter\_module, /\* 支持添加charset，也支持将内容从一种字符集转换到另外一种字符集 \*/  
 &ngx\_http\_userid\_filter\_module, /\* 支持添加统计用的识别用户的cookie \*/  
 &ngx\_http\_headers\_filter\_module, /\* 支持设置expire和Cache-control头，支持添加任意名称的头 \*/  
 &ngx\_http\_copy\_filter\_module, /\* 根据需求重新复制输出链表中的某些节点  
 （比如将in\_file的节点从文件读出并复制到新的节点），并交给后续filter  
 进行处理 \*/  
 &ngx\_http\_range\_body\_filter\_module, /\* body filter，支持range功能，如果请求包含range请求，  
 那就只发送range请求的一段内容 \*/  
 &ngx\_http\_not\_modified\_filter\_module, /\* 如果请求的if-modified-since等于回复的last-modified值，  
 说明回复没有变化，清空所有回复的内容，返回304 \*/   
 NULL  
};

从模块的命名可以很容易看出哪些模块是filter模块，一般而言Nginx的filter模块名以filter\_module结尾，普通的模块名以module结尾。上面的列表从下往上看，ngx\_http\_not\_modified\_filter\_module实际上filter链的第一个节点，而ngx\_http\_write\_filter\_module是最后一个节点。filter模块的执行顺序特别重要，比如数据经过gzip模块后就变成了压缩之后的数据，如果在gzip模块后面运行的filter模块需要再查看数据的原始内容就不可能了（除非再做解压），第三方模块会被Nginx注册在ngx\_http\_copy\_filter\_module之后，ngx\_http\_headers\_filter\_module之前。这样设定的原因是为了确保一些模块比如gzip filter，chunked filter，copy filter运行在filter链的开头或尾部。

### header filter分析

通常Nginx调用ngx\_http\_send\_header函数来发送响应头，看下它的实现：

ngx\_int\_t  
ngx\_http\_send\_header(ngx\_http\_request\_t \*r)  
{  
 if (r->err\_status) {  
 r->headers\_out.status = r->err\_status;  
 r->headers\_out.status\_line.len = 0;  
 }  
  
 return ngx\_http\_top\_header\_filter(r);  
}

上面的代码中调用了ngx\_http\_top\_header\_filter，也就是header filter的头节点，按照上一节介绍的顺序，ngx\_http\_not\_modified\_filter\_module是最后一个注册的filter模块，而最后定义的会最先执行，初始化之后，它实际上是ngx\_http\_not\_modified\_header\_filter函数：

static ngx\_int\_t  
ngx\_http\_not\_modified\_header\_filter(ngx\_http\_request\_t \*r)  
{  
 if (r->headers\_out.status != NGX\_HTTP\_OK  
 || r != r->main  
 || r->headers\_out.last\_modified\_time == -1)  
 {  
 return ngx\_http\_next\_header\_filter(r);  
 }  
  
 if (r->headers\_in.if\_unmodified\_since) {  
 return ngx\_http\_test\_precondition(r);  
 }  
  
 if (r->headers\_in.if\_modified\_since) {  
 return ngx\_http\_test\_not\_modified(r);  
 }  
  
 return ngx\_http\_next\_header\_filter(r);  
}

而在ngx\_http\_not\_modified\_header\_filter函数中，它会调用模块内部定义的函数指针变量ngx\_http\_next\_header\_filter，而该变量保存的是上一模块注册的header filter函数，同样的下一个header filter函数内部也会调用其模块内部的ngx\_http\_next\_header\_filter，直到调用到最后一个header filter - ngx\_http\_header\_filter。

ngx\_http\_header\_filter，这个filter负责计算响应头的总大小，并分配内存，组装响应头，并调用ngx\_http\_write\_filter发送。Nginx中，header filter只会被调用一次，ngx\_http\_header\_filter函数中首先会检查r->header\_sent标识是否已经被设置，如果是的话，则直接返回；否则设置该标识，并发送响应头。另外如果是子请求的话，也会直接退出函数。

### body filter分析

Nginx中通常调用ngx\_http\_output\_filter函数来发送响应体，它的实现如下：

ngx\_int\_t  
ngx\_http\_output\_filter(ngx\_http\_request\_t \*r, ngx\_chain\_t \*in)  
{  
 ngx\_int\_t rc;  
 ngx\_connection\_t \*c;  
  
 c = r->connection;  
  
 ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, c->log, 0,  
 "http output filter \"%V?%V\"", &r->uri, &r->args);  
  
 rc = ngx\_http\_top\_body\_filter(r, in);  
  
 if (rc == NGX\_ERROR) {  
 /\* NGX\_ERROR may be returned by any filter \*/  
 c->error = 1;  
 }  
  
 return rc;  
}

body filter链调用的原理和header filter一样，和ngx\_http\_send\_header函数不同的是，上面的函数多了一个类型为ngx\_chain\_t \*的参数，因为Nginx实现的是流式的输出，并不用等到整个响应体都生成了才往客户端发送数据，而是产生一部分内容之后将其组织成链表，调用ngx\_http\_output\_filter发送，并且待发送的内容可以在文件中，也可以是在内存中，Nginx会负责将数据流式的，高效的传输出去。而且当发送缓存区满了时，Nginx还会负责保存未发送完的数据，调用者只需要对新数据调用一次ngx\_http\_output\_filter即可。

#### ngx\_http\_copy\_filter\_module分析

ngx\_http\_copy\_filter\_module是响应体过滤链（body filter）中非常重要的一个模块，这个filter模块主要是来将一些需要复制的buf（可能在文件中，也可能在内存中）重新复制一份交给后面的filter模块处理。先来看它的初始化函数：

static ngx\_int\_t  
ngx\_http\_copy\_filter\_init(ngx\_conf\_t \*cf)  
{  
 ngx\_http\_next\_body\_filter = ngx\_http\_top\_body\_filter;  
 ngx\_http\_top\_body\_filter = ngx\_http\_copy\_filter;  
  
 return NGX\_OK;  
}

可以看到，它只注册了body filter，而没有注册header filter，也就是说只有body filter链中才有这个模块。

该模块有一个命令，命令名为output\_buffers，用来配置可用的buffer数和buffer大小，它的值保存在copy filter的loc conf的bufs字段，默认数量为1，大小为32768字节。这个参数具体的作用后面会做介绍。

Nginx中，一般filter模块可以header filter函数中根据请求响应头设置一个模块上下文（context），用来保存相关的信息，在body filter函数中使用这个上下文。而copy filter没有header filter，因此它的context的初始化也是放在body filter中的，而它的ctx就是ngx\_output\_chain\_ctx\_t，为什么名字是output\_chain呢，这是因为copy filter的主要逻辑的处理都放在ngx\_output\_chain模块中，另外这个模块在core目录下，而不是属于http目录。

接下来看一下上面说到的context结构：

struct ngx\_output\_chain\_ctx\_s {  
 ngx\_buf\_t \*buf; /\* 保存临时的buf \*/  
 ngx\_chain\_t \*in; /\* 保存了将要发送的chain \*/  
 ngx\_chain\_t \*free; /\* 保存了已经发送完毕的chain，以便于重复利用 \*/  
 ngx\_chain\_t \*busy; /\* 保存了还未发送的chain \*/  
  
 unsigned sendfile:1; /\* sendfile标记 \*/  
 unsigned directio:1; /\* directio标记 \*/  
#if (NGX\_HAVE\_ALIGNED\_DIRECTIO)  
 unsigned unaligned:1;  
#endif  
 unsigned need\_in\_memory:1; /\* 是否需要在内存中保存一份(使用sendfile的话，  
 内存中没有文件的拷贝的，而我们有时需要处理文件，  
 此时就需要设置这个标记) \*/  
 unsigned need\_in\_temp:1; /\* 是否需要在内存中重新复制一份，不管buf是在内存还是文件,  
 这样的话，后续模块可以直接修改这块内存 \*/  
#if (NGX\_HAVE\_FILE\_AIO)  
 unsigned aio:1;  
  
 ngx\_output\_chain\_aio\_pt aio\_handler;  
#endif  
  
 off\_t alignment;  
  
 ngx\_pool\_t \*pool;  
 ngx\_int\_t allocated; /\* 已经分别的buf个数 \*/  
 ngx\_bufs\_t bufs; /\* 对应loc conf中设置的bufs \*/  
 ngx\_buf\_tag\_t tag; /\* 模块标记，主要用于buf回收 \*/  
  
 ngx\_output\_chain\_filter\_pt output\_filter; /\* 一般是ngx\_http\_next\_filter,也就是继续调用filter链 \*/  
 void \*filter\_ctx; /\* 当前filter的上下文，  
 这里是由于upstream也会调用output\_chain \*/  
};

为了更好的理解context结构每个域的具体含义，接下来分析filter的具体实现：

static ngx\_int\_t  
ngx\_http\_copy\_filter(ngx\_http\_request\_t \*r, ngx\_chain\_t \*in)  
{  
 ngx\_int\_t rc;  
 ngx\_connection\_t \*c;  
 ngx\_output\_chain\_ctx\_t \*ctx;  
 ngx\_http\_core\_loc\_conf\_t \*clcf;  
 ngx\_http\_copy\_filter\_conf\_t \*conf;  
  
 c = r->connection;  
  
 ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, c->log, 0,  
 "http copy filter: \"%V?%V\"", &r->uri, &r->args);  
  
 /\* 获取ctx \*/  
 ctx = ngx\_http\_get\_module\_ctx(r, ngx\_http\_copy\_filter\_module);  
  
 /\* 如果为空，则说明需要初始化ctx \*/  
 if (ctx == NULL) {  
 ctx = ngx\_pcalloc(r->pool, sizeof(ngx\_output\_chain\_ctx\_t));  
 if (ctx == NULL) {  
 return NGX\_ERROR;  
 }  
  
 ngx\_http\_set\_ctx(r, ctx, ngx\_http\_copy\_filter\_module);  
  
 conf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_copy\_filter\_module);  
 clcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_core\_module);  
  
 /\* 设置sendfile \*/  
 ctx->sendfile = c->sendfile;  
 /\* 如果request设置了filter\_need\_in\_memory的话，ctx的这个域就会被设置 \*/  
 ctx->need\_in\_memory = r->main\_filter\_need\_in\_memory  
 || r->filter\_need\_in\_memory;  
 /\* 和上面类似 \*/  
 ctx->need\_in\_temp = r->filter\_need\_temporary;  
  
 ctx->alignment = clcf->directio\_alignment;  
  
 ctx->pool = r->pool;  
 ctx->bufs = conf->bufs;  
 ctx->tag = (ngx\_buf\_tag\_t) &ngx\_http\_copy\_filter\_module;  
 /\* 可以看到output\_filter就是下一个body filter节点 \*/  
 ctx->output\_filter = (ngx\_output\_chain\_filter\_pt)  
 ngx\_http\_next\_body\_filter;  
 /\* 此时filter ctx为当前的请求 \*/  
 ctx->filter\_ctx = r;  
  
 ...  
  
 if (in && in->buf && ngx\_buf\_size(in->buf)) {  
 r->request\_output = 1;  
 }  
 }  
  
 ...  
  
 for ( ;; ) {  
 /\* 最关键的函数，下面会详细分析 \*/  
 rc = ngx\_output\_chain(ctx, in);  
  
 if (ctx->in == NULL) {  
 r->buffered &= ~NGX\_HTTP\_COPY\_BUFFERED;  
  
 } else {  
 r->buffered |= NGX\_HTTP\_COPY\_BUFFERED;  
 }  
  
 ...  
  
 return rc;  
 }  
}

上面的代码去掉了AIO相关的部分，函数首先设置并初始化context，接着调用ngx\_output\_chain函数，这个函数实际上包含了copy filter模块的主要逻辑，它的原型为：

ngx\_int\_t ngx\_output\_chain(ngx\_output\_chain\_ctx\_t *ctx, ngx\_chain\_t*in)

分段来看它的代码，下面这段代码是一个快捷路径（short path），也就是说当能直接确定所有的in chain都不需要复制的时，可以直接调用output\_filter来交给剩下的filter去处理：

if (ctx->in == NULL && ctx->busy == NULL) {  
  
 /\*  
 \* the short path for the case when the ctx->in and ctx->busy chains  
 \* are empty, the incoming chain is empty too or has the single buf  
 \* that does not require the copy  
 \*/  
  
 if (in == NULL) {  
 return ctx->output\_filter(ctx->filter\_ctx, in);  
 }  
  
 if (in->next == NULL  
#if (NGX\_SENDFILE\_LIMIT)  
 && !(in->buf->in\_file && in->buf->file\_last > NGX\_SENDFILE\_LIMIT)  
#endif  
 && ngx\_output\_chain\_as\_is(ctx, in->buf))  
 {  
 return ctx->output\_filter(ctx->filter\_ctx, in);  
 }  
}

上面可以看到了一个函数ngx\_output\_chain\_as\_is，这个函数很关键，下面还会再次被调用，这个函数主要用来判断是否需要复制buf。返回1,表示不需要拷贝，否则为需要拷贝：

static ngx\_inline ngx\_int\_t  
ngx\_output\_chain\_as\_is(ngx\_output\_chain\_ctx\_t \*ctx, ngx\_buf\_t \*buf)  
{  
 ngx\_uint\_t sendfile;  
  
 /\* 是否为特殊buf（special buf），是的话返回1，也就是不用拷贝 \*/  
 if (ngx\_buf\_special(buf)) {  
 return 1;  
 }  
  
 /\* 如果buf在文件中，并且使用了directio的话，需要拷贝buf \*/  
 if (buf->in\_file && buf->file->directio) {  
 return 0;  
 }  
  
 /\* sendfile标记 \*/  
 sendfile = ctx->sendfile;  
  
#if (NGX\_SENDFILE\_LIMIT)  
 /\* 如果pos大于sendfile的限制，设置标记为0 \*/  
 if (buf->in\_file && buf->file\_pos >= NGX\_SENDFILE\_LIMIT) {  
 sendfile = 0;  
 }  
  
#endif  
  
 if (!sendfile) {  
 /\* 如果不走sendfile，而且buf不在内存中，则我们就需要复制到内存一份 \*/  
 if (!ngx\_buf\_in\_memory(buf)) {  
 return 0;  
 }  
  
 buf->in\_file = 0;  
 }  
  
 /\* 如果需要内存中有一份拷贝，而并不在内存中，此时返回0，表示需要拷贝 \*/  
 if (ctx->need\_in\_memory && !ngx\_buf\_in\_memory(buf)) {  
 return 0;  
 }  
  
 /\* 如果需要内存中有可修改的拷贝，并且buf存在于只读的内存中或者mmap中，则返回0 \*/   
 if (ctx->need\_in\_temp && (buf->memory || buf->mmap)) {  
 return 0;  
 }  
  
 return 1;  
}

上面有两个标记要注意，一个是need\_in\_memory ，这个主要是用于当使用sendfile的时候，Nginx并不会将请求文件拷贝到内存中，而有时需要操作文件的内容，此时就需要设置这个标记。然后后面的body filter就能操作内容了。

第二个是need\_in\_temp，这个主要是用于把本来就存在于内存中的buf复制一份可修改的拷贝出来，这里有用到的模块有charset，也就是编解码 filter。

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

/\* add the incoming buf to the chain ctx->in \*/  
  
if (in) {  
 if (ngx\_output\_chain\_add\_copy(ctx->pool, &ctx->in, in) == NGX\_ERROR) {  
 return NGX\_ERROR;  
 }  
}

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

先来看chain的重用。关键的几个结构以及域：ctx的free，busy以及ctx->pool的chain域。

其中每次发送没有发完的chain就放到busy中，而已经发送完毕的就放到free中，而最后会调用 ngx\_free\_chain来将free的chain放入到pool->chain中,而在ngx\_alloc\_chain\_link中，如果pool->chain中存在chain的话，就不用malloc了，而是直接返回pool->chain，相关的代码如下：

/\* 链接cl到pool->chain中 \*/   
#define ngx\_free\_chain(pool, cl) \   
 cl->next = pool->chain; \   
 pool->chain = cl   
  
/\* 从pool中分配chain \*/  
ngx\_chain\_t \*   
ngx\_alloc\_chain\_link(ngx\_pool\_t \*pool)   
{   
 ngx\_chain\_t \*cl;   
  
 cl = pool->chain;   
 /\* 如果cl存在，则直接返回cl \*/  
 if (cl) {   
 pool->chain = cl->next;   
 return cl;   
 }   
 /\* 否则才会malloc chain \*/   
 cl = ngx\_palloc(pool, sizeof(ngx\_chain\_t));   
 if (cl == NULL) {   
 return NULL;   
 }   
  
 return cl;   
}

然后是buf的重用，严格意义上来说buf的重用是从free中的chain中取得的，当free中的buf被重用，则这个buf对应的chain就会被链接到ctx->pool中，从而这个chain就会被重用。也就是说首先考虑的是buf的重用，只有当这个chain的buf确定不需要被重用(或者说已经被重用)的时候，chain才会被链接到ctx->pool中被重用。

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

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

/\* out为最终需要传输的chain，也就是交给剩下的filter处理的chain \*/  
out = NULL;   
/\* last\_out为out的最后一个chain \*/   
last\_out = &out;   
last = NGX\_NONE;   
  
for ( ;; ) {   
  
 /\* 开始遍历chain \*/   
 while (ctx->in) {   
  
 /\* 取得当前chain的buf大小 \*/   
 bsize = ngx\_buf\_size(ctx->in->buf);   
  
 /\* 跳过bsize为0的buf \*/   
 if (bsize == 0 && !ngx\_buf\_special(ctx->in->buf)) {   
 ngx\_debug\_point();   
  
 ctx->in = ctx->in->next;   
  
 continue;   
 }   
  
 /\* 判断是否需要复制buf \*/   
 if (ngx\_output\_chain\_as\_is(ctx, ctx->in->buf)) {   
  
 /\* move the chain link to the output chain \*/   
 /\* 如果不需要复制，则直接链接chain到out，然后继续循环 \*/   
 cl = ctx->in;   
 ctx->in = cl->next;   
  
 \*last\_out = cl;   
 last\_out = &cl->next;   
 cl->next = NULL;   
  
 continue;   
 }   
  
 /\* 到达这里，说明我们需要拷贝buf，这里buf最终都会被拷贝进ctx->buf中，  
 因此这里先判断ctx->buf是否为空 \*/   
 if (ctx->buf == NULL) {   
  
 /\* 如果为空，则取得buf，这里要注意，一般来说如果没有开启directio的话，  
 这个函数都会返回NGX\_DECLINED \*/   
 rc = ngx\_output\_chain\_align\_file\_buf(ctx, bsize);   
  
 if (rc == NGX\_ERROR) {   
 return NGX\_ERROR;   
 }   
  
 /\* 大部分情况下，都会落入这个分支 \*/   
 if (rc != NGX\_OK) {   
  
 /\* 准备分配buf，首先在free中寻找可以重用的buf \*/  
 if (ctx->free) {   
  
 /\* get the free buf \*/   
 /\* 得到free buf \*/   
 cl = ctx->free;   
 ctx->buf = cl->buf;   
 ctx->free = cl->next;   
 /\* 将要重用的chain链接到ctx->poll中，以便于chain的重用 \*/   
 ngx\_free\_chain(ctx->pool, cl);   
  
 } else if (out || ctx->allocated == ctx->bufs.num) {   
 /\* 如果已经等于buf的个数限制，则跳出循环，发送已经存在的buf。  
 这里可以看到如果out存在的话，nginx会跳出循环，然后发送out，  
 等发送完会再次处理，这里很好的体现了nginx的流式处理 \*/   
 break;   
  
 } else if (ngx\_output\_chain\_get\_buf(ctx, bsize) != NGX\_OK) {   
 /\* 上面这个函数也比较关键，它用来取得buf。接下来会详细看这个函数 \*/   
 return NGX\_ERROR;   
 }   
 }   
 }   
 /\* 从原来的buf中拷贝内容或者从文件中读取内容 \*/  
 rc = ngx\_output\_chain\_copy\_buf(ctx);  
  
 if (rc == NGX\_ERROR) {  
 return rc;  
 }  
  
 if (rc == NGX\_AGAIN) {  
 if (out) {  
 break;  
 }  
  
 return rc;  
 }  
  
 /\* delete the completed buf from the ctx->in chain \*/  
  
 if (ngx\_buf\_size(ctx->in->buf) == 0) {  
 ctx->in = ctx->in->next;  
 }  
 /\* 分配新的chain节点 \*/  
 cl = ngx\_alloc\_chain\_link(ctx->pool);  
 if (cl == NULL) {  
 return NGX\_ERROR;  
 }  
  
 cl->buf = ctx->buf;  
 cl->next = NULL;  
 \*last\_out = cl;  
 last\_out = &cl->next;  
 ctx->buf = NULL;   
 }   
 ...  
}

上面的代码分析的时候有个很关键的函数，那就是ngx\_output\_chain\_get\_buf，这个函数当没有可重用的buf时用来分配buf。

如果当前的buf位于最后一个chain，则需要特殊处理，一是buf的recycled域，另外是将要分配的buf的大小。

先来说recycled域，这个域表示当前的buf需要被回收。而一般情况下Nginx(比如在非last buf)会缓存一部分buf(默认是1460字节)，然后再发送，而设置了recycled的话，就不会让它缓存buf，也就是尽量发送出去，然后以供回收使用。 因此如果是最后一个buf，则不需要设置recycled域的，否则的话，需要设置recycled域。

然后就是buf的大小。这里会有两个大小，一个是需要复制的buf的大小，一个是配置文件中设置的大小。如果不是最后一个buf，则只需要分配配置中设置的buf的大小就行了。如果是最后一个buf，则就处理不太一样，下面的代码会看到：

static ngx\_int\_t   
ngx\_output\_chain\_get\_buf(ngx\_output\_chain\_ctx\_t \*ctx, off\_t bsize)   
{   
 size\_t size;   
 ngx\_buf\_t \*b, \*in;   
 ngx\_uint\_t recycled;   
  
 in = ctx->in->buf;   
 /\* 可以看到这里分配的buf，每个buf的大小是配置文件中设置的size \*/   
 size = ctx->bufs.size;   
 /\* 默认有设置recycled域 \*/   
 recycled = 1;   
 /\* 如果当前的buf是属于最后一个chain的时候，需要特殊处理 \*/   
 if (in->last\_in\_chain) {   
 /\* 如果buf大小小于配置指定的大小，则直接按实际大小分配，不设置回收标记 \*/  
 if (bsize < (off\_t) size) {   
  
 /\*   
 \* allocate a small temp buf for a small last buf   
 \* or its small last part   
 \*/   
 size = (size\_t) bsize;   
 recycled = 0;   
  
 } else if (!ctx->directio   
 && ctx->bufs.num == 1   
 && (bsize < (off\_t) (size + size / 4)))   
 {   
 /\*   
 \* allocate a temp buf that equals to a last buf,   
 \* if there is no directio, the last buf size is lesser   
 \* than 1.25 of bufs.size and the temp buf is single   
 \*/   
  
 size = (size\_t) bsize;   
 recycled = 0;   
 }   
 }   
 /\* 开始分配buf内存 \*/   
 b = ngx\_calloc\_buf(ctx->pool);   
 if (b == NULL) {   
 return NGX\_ERROR;   
 }   
  
 if (ctx->directio) {   
 /\* directio需要对齐 \*/   
  
 b->start = ngx\_pmemalign(ctx->pool, size, (size\_t) ctx->alignment);   
 if (b->start == NULL) {   
 return NGX\_ERROR;   
 }   
  
 } else {   
 /\* 大部分情况会走到这里 \*/   
 b->start = ngx\_palloc(ctx->pool, size);   
 if (b->start == NULL) {   
 return NGX\_ERROR;   
 }   
 }   
  
 b->pos = b->start;   
 b->last = b->start;   
 b->end = b->last + size;   
 /\* 设置temporary \*/   
 b->temporary = 1;   
 b->tag = ctx->tag;   
 b->recycled = recycled;   
  
 ctx->buf = b;   
 /\* 更新allocated,可以看到每分配一个就加1 \*/   
 ctx->allocated++;   
  
 return NGX\_OK;   
}

分配新的buf和chain，并调用ngx\_output\_chain\_copy\_buf拷贝完数据之后，Nginx就将新的chain链表交给下一个body filter继续处理：

if (out == NULL && last != NGX\_NONE) {  
  
 if (ctx->in) {  
 return NGX\_AGAIN;  
 }  
  
 return last;  
}  
  
last = ctx->output\_filter(ctx->filter\_ctx, out);  
  
if (last == NGX\_ERROR || last == NGX\_DONE) {  
 return last;  
}  
  
ngx\_chain\_update\_chains(ctx->pool, &ctx->free, &ctx->busy, &out,  
 ctx->tag);  
last\_out = &out;

在其他body filter处理完之后，ngx\_output\_chain函数还需要更新chain链表，以便回收利用，ngx\_chain\_update\_chains函数主要是将处理完毕的chain节点放入到free链表，没有处理完毕的放到busy链表中，另外这个函数用到了tag，它只回收copy filter产生的chain节点。

#### ngx\_http\_write\_filter\_module分析

ngx\_http\_write\_filter\_module是最后一个body filter，可以看到它的注册函数的特殊性：

static ngx\_int\_t  
ngx\_http\_write\_filter\_init(ngx\_conf\_t \*cf)  
{  
 ngx\_http\_top\_body\_filter = ngx\_http\_write\_filter;  
  
 return NGX\_OK;  
}

ngx\_http\_write\_filter\_module是第一个注册body filter的模块，于是它也是最后一个执行的body filter模块。

直接来看ngx\_http\_write\_filter，下面的代码中去掉了一些调试代码：

ngx\_int\_t  
ngx\_http\_write\_filter(ngx\_http\_request\_t \*r, ngx\_chain\_t \*in)  
{  
 off\_t size, sent, nsent, limit;  
 ngx\_uint\_t last, flush;  
 ngx\_msec\_t delay;  
 ngx\_chain\_t \*cl, \*ln, \*\*ll, \*chain;  
 ngx\_connection\_t \*c;  
 ngx\_http\_core\_loc\_conf\_t \*clcf;  
  
 c = r->connection;  
  
 if (c->error) {  
 return NGX\_ERROR;  
 }  
  
 size = 0;  
 flush = 0;  
 last = 0;  
 ll = &r->out;  
  
 /\* find the size, the flush point and the last link of the saved chain \*/  
  
 for (cl = r->out; cl; cl = cl->next) {  
 ll = &cl->next;  
  
#if 1  
 if (ngx\_buf\_size(cl->buf) == 0 && !ngx\_buf\_special(cl->buf)) {  
 return NGX\_ERROR;  
 }  
#endif  
  
 size += ngx\_buf\_size(cl->buf);  
  
 if (cl->buf->flush || cl->buf->recycled) {  
 flush = 1;  
 }  
  
 if (cl->buf->last\_buf) {  
 last = 1;  
 }  
 }  
  
 /\* add the new chain to the existent one \*/  
  
 for (ln = in; ln; ln = ln->next) {  
 cl = ngx\_alloc\_chain\_link(r->pool);  
 if (cl == NULL) {  
 return NGX\_ERROR;  
 }  
  
 cl->buf = ln->buf;  
 \*ll = cl;  
 ll = &cl->next;  
  
#if 1  
 if (ngx\_buf\_size(cl->buf) == 0 && !ngx\_buf\_special(cl->buf)) {  
 return NGX\_ERROR;  
 }  
#endif  
  
 size += ngx\_buf\_size(cl->buf);  
  
 if (cl->buf->flush || cl->buf->recycled) {  
 flush = 1;  
 }  
  
 if (cl->buf->last\_buf) {  
 last = 1;  
 }  
 }  
  
 \*ll = NULL;  
  
 clcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_core\_module);  
  
 /\*  
 \* avoid the output if there are no last buf, no flush point,  
 \* there are the incoming bufs and the size of all bufs  
 \* is smaller than "postpone\_output" directive  
 \*/  
  
 if (!last && !flush && in && size < (off\_t) clcf->postpone\_output) {  
 return NGX\_OK;  
 }  
 /\* 如果请求由于被限速而必须延迟发送时，设置一个标识后退出 \*/  
 if (c->write->delayed) {  
 c->buffered |= NGX\_HTTP\_WRITE\_BUFFERED;  
 return NGX\_AGAIN;  
 }  
 /\* 如果buffer总大小为0，而且当前连接之前没有由于底层发送接口的原因延迟，  
 则检查是否有特殊标记 \*/  
 if (size == 0 && !(c->buffered & NGX\_LOWLEVEL\_BUFFERED)) {  
 /\* last\_buf标记，表示请求体已经发送结束 \*/  
 if (last) {  
 r->out = NULL;  
 c->buffered &= ~NGX\_HTTP\_WRITE\_BUFFERED;  
  
 return NGX\_OK;  
 }  
 /\* flush生效，而且又没有实际数据，则清空当前的未发送队列 \*/  
 if (flush) {  
 do {  
 r->out = r->out->next;  
 } while (r->out);  
  
 c->buffered &= ~NGX\_HTTP\_WRITE\_BUFFERED;  
  
 return NGX\_OK;  
 }  
  
 return NGX\_ERROR;  
 }  
 /\*　请求有速率限制，则计算当前可以发送的大小 \*/  
 if (r->limit\_rate) {  
 limit = r->limit\_rate \* (ngx\_time() - r->start\_sec + 1)  
 - (c->sent - clcf->limit\_rate\_after);  
  
 if (limit <= 0) {  
 c->write->delayed = 1;  
 ngx\_add\_timer(c->write,  
 (ngx\_msec\_t) (- limit \* 1000 / r->limit\_rate + 1));  
  
 c->buffered |= NGX\_HTTP\_WRITE\_BUFFERED;  
  
 return NGX\_AGAIN;  
 }  
  
 if (clcf->sendfile\_max\_chunk  
 && (off\_t) clcf->sendfile\_max\_chunk < limit)  
 {  
 limit = clcf->sendfile\_max\_chunk;  
 }  
  
 } else {  
 limit = clcf->sendfile\_max\_chunk;  
 }  
  
 sent = c->sent;  
 /\* 发送数据 \*/  
 chain = c->send\_chain(c, r->out, limit);  
  
 if (chain == NGX\_CHAIN\_ERROR) {  
 c->error = 1;  
 return NGX\_ERROR;  
 }  
 /\* 更新限速相关的信息 \*/  
 if (r->limit\_rate) {  
  
 nsent = c->sent;  
  
 if (clcf->limit\_rate\_after) {  
  
 sent -= clcf->limit\_rate\_after;  
 if (sent < 0) {  
 sent = 0;  
 }  
  
 nsent -= clcf->limit\_rate\_after;  
 if (nsent < 0) {  
 nsent = 0;  
 }  
 }  
  
 delay = (ngx\_msec\_t) ((nsent - sent) \* 1000 / r->limit\_rate);  
  
 if (delay > 0) {  
 limit = 0;  
 c->write->delayed = 1;  
 ngx\_add\_timer(c->write, delay);  
 }  
 }  
  
 if (limit  
 && c->write->ready  
 && c->sent - sent >= limit - (off\_t) (2 \* ngx\_pagesize))  
 {  
 c->write->delayed = 1;  
 ngx\_add\_timer(c->write, 1);  
 }  
 /\* 更新输出链，释放已经发送的节点 \*/  
 for (cl = r->out; cl && cl != chain; /\* void \*/) {  
 ln = cl;  
 cl = cl->next;  
 ngx\_free\_chain(r->pool, ln);  
 }  
  
 r->out = chain;  
 /\* 如果数据未发送完毕，则设置一个标记 \*/  
 if (chain) {  
 c->buffered |= NGX\_HTTP\_WRITE\_BUFFERED;  
 return NGX\_AGAIN;  
 }  
  
 c->buffered &= ~NGX\_HTTP\_WRITE\_BUFFERED;  
 /\* 如果由于底层发送接口导致数据未发送完全，且当前请求没有其他数据需要发送，  
 此时要返回NGX\_AGAIN，表示还有数据未发送 \*/  
 if ((c->buffered & NGX\_LOWLEVEL\_BUFFERED) && r->postponed == NULL) {  
 return NGX\_AGAIN;  
 }  
  
 return NGX\_OK;  
}

Nginx将待发送的chain链表保存在r->out，上面的函数先检查之前未发送完的链表中是否有flush，recycled以及last\_buf标识，并计算所有buffer的大小，接着对新输入的chain链表做同样的事情，并将新链表加到r->out的队尾。

如果没有输出链表中没有被标识为最后一块buffer的节点，而且没有需要flush或者急着回收的buffer，并且当前队列中buffer总大小不够postpone\_output指令设置的大小（默认为1460字节）时，函数会直接返回。

ngx\_http\_write\_filter会调用c->send\_chain往客户端发送数据，c->send\_chain的取值在不同操作系统，编译选项以及协议下（https下用的是ngx\_ssl\_send\_chain）会取不同的函数，典型的linux操作系统下，它的取值为ngx\_linux\_sendfile\_chain，也就是最终会调用这个函数来发送数据。它的函数原型为：

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

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

首先看一下这个函数定义的一些重要局部变量：

send表示将要发送的buf已经已经发送的大小；

sent表示已经发送的buf的大小；

prev\_send表示上一次发送的大小，也就是已经发送的buf的大小；

fprev 和prev-send类似，只不过是file类型的；

complete表示是否buf被完全发送了，也就是sent是否等于send - prev\_send；

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

struct iovec \*iov, headers[NGX\_HEADERS] 这个主要是用于sendfile和writev的参数，这里注意上面header数组保存的就是iovec。

下面看函数开头的一些初始化代码：

wev = c->write;   
  
if (!wev->ready) {   
 return in;   
}   
  
/\* the maximum limit size is 2G-1 - the page size \*/  
  
if (limit == 0 || limit > (off\_t) (NGX\_SENDFILE\_LIMIT - ngx\_pagesize)) {   
 limit = NGX\_SENDFILE\_LIMIT - ngx\_pagesize;   
}   
  
  
send = 0;   
  
/\* 设置header，也就是in memory的数组 \*/   
header.elts = headers;   
header.size = sizeof(struct iovec);   
header.nalloc = NGX\_HEADERS;   
header.pool = c->pool;

下面这段代码就是处理in memory的部分，然后将buf放入对应的iovec数组，处理核心思想就是合并内存连续并相邻的buf(不管是in memory还是in file)：

for (cl = in; cl && send < limit; cl = cl->next) {   
  
 if (ngx\_buf\_special(cl->buf)) {   
 continue;   
 }   
  
 /\* 如果既不在内存中，又不在文件中，则返回错误 \*/  
 if (!ngx\_buf\_in\_memory(cl->buf) && !cl->buf->in\_file) {  
 return NGX\_CHAIN\_ERROR;  
 }   
  
 /\* 如果不只是在buf中，这是因为有时in file的buf可能需要内存中也有拷贝，  
 如果一个buf同时in memoey和in file的话，Nginx会把它当做in file来处理 \*/   
 if (!ngx\_buf\_in\_memory\_only(cl->buf)) {   
 break;   
 }   
  
 /\* 得到buf的大小 \*/  
 size = cl->buf->last - cl->buf->pos;   
  
 /\* 大于limit的话修改为size \*/   
 if (send + size > limit) {   
 size = limit - send;   
 }  
  
 /\* 如果prev等于pos，则说明当前的buf的数据和前一个buf的数据是连续的 \*/   
 if (prev == cl->buf->pos) {   
 iov->iov\_len += (size\_t) size;   
  
 } else {   
 if (header.nelts >= IOV\_MAX) {  
 break;  
 }  
 /\* 否则说明是不同的buf，因此增加一个iovc \*/   
 iov = ngx\_array\_push(&header);   
 if (iov == NULL) {   
 return NGX\_CHAIN\_ERROR;   
 }   
  
 iov->iov\_base = (void \*) cl->buf->pos;   
 iov->iov\_len = (size\_t) size;   
 }   
  
 /\* 这里可以看到prev保存了当前buf的结尾 \*/   
 prev = cl->buf->pos + (size\_t) size;   
 /\* 更新发送的大小 \*/   
 send += size;   
}

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

/\* 如果header的大小不为0则说明前面有需要发送的buf，  
 并且数据大小已经超过限制则跳过in file处理 \*/  
if (header.nelts == 0 && cl && cl->buf->in\_file && send < limit) {   
 /\* 得到file   
 file = cl->buf;   
  
 /\* 开始合并 \*/   
 do {   
 /\* 得到大小 \*/   
 size = cl->buf->file\_last - cl->buf->file\_pos;   
  
 /\* 如果太大则进行对齐处理 \*/   
 if (send + size > limit) {   
 size = limit - send;   
  
 aligned = (cl->buf->file\_pos + size + ngx\_pagesize - 1)   
 & ~((off\_t) ngx\_pagesize - 1);   
  
 if (aligned <= cl->buf->file\_last) {   
 size = aligned - cl->buf->file\_pos;   
 }   
 }   
  
 /\* 设置file\_size \*/   
 file\_size += (size\_t) size;   
 /\* 设置需要发送的大小 \*/   
 send += size;   
 /\* 和上面的in memory处理一样就是保存这次的last \*/   
 fprev = cl->buf->file\_pos + size;   
 cl = cl->next;   
  
 } while (cl   
 && cl->buf->in\_file   
 && send < limit   
 && file->file->fd == cl->buf->file->fd   
 && fprev == cl->buf->file\_pos);   
}

然后就是发送部分，这里in file使用sendfile，in memory使用writev。处理逻辑比较简单，就是发送后判断发送成功的大小

if (file) {   
#if 1   
 if (file\_size == 0) {   
 ngx\_debug\_point();   
 return NGX\_CHAIN\_ERROR;   
 }   
#endif   
#if (NGX\_HAVE\_SENDFILE64)   
 offset = file->file\_pos;   
#else   
 offset = (int32\_t) file->file\_pos;   
#endif   
  
 /\* 数据在文件中则调用sendfile发送数据 \*/  
 rc = sendfile(c->fd, file->file->fd, &offset, file\_size);   
  
 ...  
  
 /\* 得到发送成功的字节数 \*/   
 sent = rc > 0 ? rc : 0;   
  
} else {  
 /\* 数据在内存中则调用writev发送数据 \*/   
 rc = writev(c->fd, header.elts, header.nelts);   
  
 ...  
 /\* 得到发送成功的字节数 \*/  
 sent = rc > 0 ? rc : 0;   
}

接下来就是需要根据发送成功的字节数来更新chain：

/\* 如果send - prev\_send == sent则说明该发送的都发完了 \*/   
if (send - prev\_send == sent) {   
 complete = 1;   
}   
/\* 更新congnect的sent域 \*/   
c->sent += sent;   
  
/\* 开始重新遍历chain，这里是为了防止没有发送完全的情况，  
 此时我们就需要切割buf了 \*/   
for (cl = in; cl; cl = cl->next) {   
  
 if (ngx\_buf\_special(cl->buf)) {   
 continue;   
 }   
  
 if (sent == 0) {   
 break;   
 }   
 /\* 得到buf size \*/   
 size = ngx\_buf\_size(cl->buf);   
  
 /\* 如果大于当前的size，则说明这个buf的数据已经被完全发送完毕了，  
 因此更新它的域 \*/   
 if (sent >= size){   
 /\* 更新sent域 \*/   
 sent -= size;   
 /\* 如果在内存则更新pos \*/   
 if (ngx\_buf\_in\_memory(cl->buf)) {   
 cl->buf->pos = cl->buf->last;   
 }   
 /\* 如果在file中则更显file\_pos \*/   
 if (cl->buf->in\_file) {   
 cl->buf->file\_pos = cl->buf->file\_last;   
 }   
  
 continue;   
 }   
  
 /\* 到这里说明当前的buf只有一部分被发送出去了，因此只需要修改指针。  
 以便于下次发送 \*/   
 if (ngx\_buf\_in\_memory(cl->buf)) {   
 cl->buf->pos += (size\_t) sent;   
 }   
 /\* 同上 \*/  
 if (cl->buf->in\_file) {   
 cl->buf->file\_pos += sent;   
 }   
  
 break;   
}

最后一部分是一些是否退出循环的判断。这里要注意，Nginx中如果发送未完全的话，将会直接返回，返回的就是没有发送完毕的chain，它的buf也已经被更新。然后Nginx返回去处理其他的事情，等待可写之后再次发送未发送完的数据：

if (eintr) {   
 continue;   
}   
/\* 如果未完成，则设置wev->ready为0后返回 \*/   
if (!complete) {   
 wev->ready = 0;   
 return cl;   
}   
/\* 发送数据超过限制，或没有数据了 \*/  
if (send >= limit || cl == NULL) {   
 return cl;   
}   
/\* 更新in，也就是开始处理下一个chain \*/  
in = cl;

## subrequest原理解析 (99%)

子请求并不是http标准里面的概念，它是在当前请求中发起的一个新的请求，它拥有自己的ngx\_http\_request\_t结构，uri和args。一般来说使用subrequest的效率可能会有些影响，因为它需要重新从server rewrite开始走一遍request处理的PHASE，但是它在某些情况下使用能带来方便，比较常用的是用subrequest来访问一个upstream的后端，并给它一个ngx\_http\_post\_subrequest\_t的回调handler，这样有点类似于一个异步的函数调用。对于从upstream返回的数据，subrequest允许根据创建时指定的flag，来决定由用户自己处理(回调handler中)还是由upstream模块直接发送到out put filter。简单的说一下subrequest的行为，nginx使用subrequest访问某个location，产生相应的数据，并插入到nginx输出链的相应位置（创建subrequest时的位置），下面用nginx代码内的addition模块(默认未编译进nginx核心，请使用--with-http\_addition\_module选项包含此模块)来举例说明一下：

location /main.htm {  
 # content of main.htm: main  
 add\_before\_body /hello.htm;  
 add\_after\_body /world.htm;  
}  
location /hello.htm {  
 #content of hello.htm: hello  
}  
location /world.htm {  
 #content of world.htm: world  
}

访问/main.htm，将得到如下响应：

hello  
main  
world

上面的add\_before\_body指令发起一个subrequest来访问/hello.htm，并将产生的内容(hello)插入主请求响应体的开头，add\_after\_body指令发起一个subrequest访问/world.htm，并将产生的内容(world)附加在主请求响应体的结尾。addition模块是一个filter模块，但是subrequest既可以在phase模块中使用，也可以在filter模块中使用。

在进行源码解析之前，先来想想如果是我们自己要实现subrequest的上述行为，该如何来做？subrequest还可能有自己的subrequest，而且每个subrequest都不一定按照其创建的顺序来输出数据，所以简单的采用链表不好实现，于是进一步联想到可以采用树的结构来做，主请求即为根节点，每个节点可以有自己的子节点，遍历某节点表示处理某请求，自然的可以想到这里可能是用后根(序)遍历的方法，没错，实际上Igor采用树和链表结合的方式实现了subrequest的功能，但是由于节点（请求）产生数据的顺序不是固定按节点创建顺序(左->右)，而且可能分多次产生数据，不能简单的用后根(序)遍历。Igor使用了2个链表的结构来实现，第一个是每个请求都有的postponed链表，一般情况下每个链表节点保存了该请求的一个子请求，该链表节点定义如下：

struct ngx\_http\_postponed\_request\_s {  
 ngx\_http\_request\_t \*request;  
 ngx\_chain\_t \*out;  
 ngx\_http\_postponed\_request\_t \*next;  
};

可以看到它有一个request字段，可以用来保存子请求，另外还有一个ngx\_chain\_t类型的out字段，实际上一个请求的postponed链表里面除了保存子请求的节点，还有保存该请求自己产生的数据的节点，数据保存在out字段；第二个是posted\_requests链表，它挂载了当前需要遍历的请求（节点）， 该链表保存在主请求（根节点）的posted\_requests字段，链表节点定义如下：

struct ngx\_http\_posted\_request\_s {  
 ngx\_http\_request\_t \*request;  
 ngx\_http\_posted\_request\_t \*next;  
};

在ngx\_http\_run\_posted\_requests函数中会顺序的遍历主请求的posted\_requests链表：

void  
ngx\_http\_run\_posted\_requests(ngx\_connection\_t \*c)  
{  
 ...  
 for ( ;; ) {  
 /\* 连接已经断开，直接返回 \*/  
 if (c->destroyed) {  
 return;  
 }  
  
 r = c->data;  
 /\* 从posted\_requests链表的队头开始遍历 \*/  
 pr = r->main->posted\_requests;  
  
 if (pr == NULL) {  
 return;  
 }  
  
  
 /\* 从链表中移除即将要遍历的节点 \*/  
 r->main->posted\_requests = pr->next;  
 /\* 得到该节点中保存的请求 \*/  
 r = pr->request;  
  
 ctx = c->log->data;  
 ctx->current\_request = r;  
  
 ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, c->log, 0,  
 "http posted request: \"%V?%V\"", &r->uri, &r->args);  
 /\* 遍历该节点（请求） \*/  
 r->write\_event\_handler(r);  
 }  
}

ngx\_http\_run\_posted\_requests函数的调用点后面会做说明。

了解了一些实现的原理，来看代码就简单多了，现在正式进行subrequest的源码解析， 首先来看一下创建subrequest的函数定义：

ngx\_int\_t  
ngx\_http\_subrequest(ngx\_http\_request\_t \*r,  
 ngx\_str\_t \*uri, ngx\_str\_t \*args, ngx\_http\_request\_t \*\*psr,  
 ngx\_http\_post\_subrequest\_t \*ps, ngx\_uint\_t flags)

参数r为当前的请求，uri和args为新的要发起的uri和args，当然args可以为NULL，psr为指向一个ngx\_http\_request\_t指针的指针，它的作用就是获得创建的子请求，ps的类型为ngx\_http\_post\_subrequest\_t，它的定义如下：

typedef struct {  
 ngx\_http\_post\_subrequest\_pt handler;  
 void \*data;  
} ngx\_http\_post\_subrequest\_t;  
  
typedef ngx\_int\_t (\*ngx\_http\_post\_subrequest\_pt)(ngx\_http\_request\_t \*r,  
 void \*data, ngx\_int\_t rc);

它就是之前说到的回调handler，结构里面的handler类型为ngx\_http\_post\_subrequest\_pt，它是函数指针，data为传递给handler的额外参数。再来看一下ngx\_http\_subrequest函数的最后一个是flags，现在的源码中实际上只有2种类型的flag，分别为NGX\_HTTP\_SUBREQUEST\_IN\_MEMORY和NGX\_HTTP\_SUBREQUEST\_WAITED，第一个就是指定文章开头说到的子请求的upstream处理数据的方式，第二个参数表示如果该子请求提前完成(按后序遍历的顺序)，是否设置将它的状态设为done，当设置该参数时，提前完成就会设置done，不设时，会让该子请求等待它之前的子请求处理完毕才会将状态设置为done。

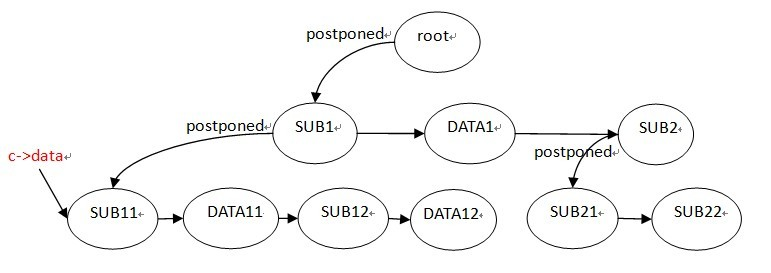
进入ngx\_http\_subrequest函数内部看看：

{  
 ...  
 /\* 解析flags， subrequest\_in\_memory在upstream模块解析完头部，  
 发送body给downsstream时用到 \*/  
 sr->subrequest\_in\_memory = (flags & NGX\_HTTP\_SUBREQUEST\_IN\_MEMORY) != 0;  
 sr->waited = (flags & NGX\_HTTP\_SUBREQUEST\_WAITED) != 0;  
  
 sr->unparsed\_uri = r->unparsed\_uri;  
 sr->method\_name = ngx\_http\_core\_get\_method;  
 sr->http\_protocol = r->http\_protocol;  
  
 ngx\_http\_set\_exten(sr);  
 /\* 主请求保存在main字段中 \*/  
 sr->main = r->main;  
 /\* 父请求为当前请求 \*/   
 sr->parent = r;  
 /\* 保存回调handler及数据，在子请求执行完，将会调用 \*/  
 sr->post\_subrequest = ps;  
 /\* 读事件handler赋值为不做任何事的函数，因为子请求不用再读数据或者检查连接状态；  
 写事件handler为ngx\_http\_handler，它会重走phase \*/  
 sr->read\_event\_handler = ngx\_http\_request\_empty\_handler;  
 sr->write\_event\_handler = ngx\_http\_handler;  
  
 /\* ngx\_connection\_s的data字段比较关键，它保存了当前可以向out chain输出数据的请求，  
 具体意义后面会做详细介绍 \*/  
 if (c->data == r && r->postponed == NULL) {  
 c->data = sr;  
 }  
 /\* 默认共享父请求的变量，当然你也可以根据需求在创建完子请求后，再创建子请求独立的变量集 \*/  
 sr->variables = r->variables;  
  
 sr->log\_handler = r->log\_handler;  
  
 pr = ngx\_palloc(r->pool, sizeof(ngx\_http\_postponed\_request\_t));  
 if (pr == NULL) {  
 return NGX\_ERROR;  
 }  
  
 pr->request = sr;  
 pr->out = NULL;  
 pr->next = NULL;  
 /\* 把该子请求挂载在其父请求的postponed链表的队尾 \*/  
 if (r->postponed) {  
 for (p = r->postponed; p->next; p = p->next) { /\* void \*/ }  
 p->next = pr;  
  
 } else {  
 r->postponed = pr;  
 }  
 /\* 子请求为内部请求，它可以访问internal类型的location \*/  
 sr->internal = 1;  
 /\* 继承父请求的一些状态 \*/  
 sr->discard\_body = r->discard\_body;  
 sr->expect\_tested = 1;  
 sr->main\_filter\_need\_in\_memory = r->main\_filter\_need\_in\_memory;  
  
 sr->uri\_changes = NGX\_HTTP\_MAX\_URI\_CHANGES + 1;  
  
 tp = ngx\_timeofday();  
 r->start\_sec = tp->sec;  
 r->start\_msec = tp->msec;  
  
 r->main->subrequests++;  
 /\* 增加主请求的引用数，这个字段主要是在ngx\_http\_finalize\_request调用的一些结束请求和  
 连接的函数中使用 \*/  
 r->main->count++;  
  
 \*psr = sr;  
 /\* 将该子请求挂载在主请求的posted\_requests链表队尾 \*/  
 return ngx\_http\_post\_request(sr, NULL);  
}

到这时，子请求创建完毕，一般来说子请求的创建都发生在某个请求的content handler或者某个filter内，从上面的函数可以看到子请求并没有马上被执行，只是被挂载在了主请求的posted\_requests链表中，那它什么时候可以执行呢？之前说到posted\_requests链表是在ngx\_http\_run\_posted\_requests函数中遍历，那么ngx\_http\_run\_posted\_requests函数又是在什么时候调用？它实际上是在某个请求的读（写）事件的handler中，执行完该请求相关的处理后被调用，比如主请求在走完一遍PHASE的时候会调用ngx\_http\_run\_posted\_requests，这时子请求得以运行。

这时实际还有1个问题需要解决，由于nginx是多进程，是不能够随意阻塞的（如果一个请求阻塞了当前进程，就相当于阻塞了这个进程accept到的所有其他请求，同时该进程也不能accept新请求），一个请求可能由于某些原因需要阻塞（比如访问io），nginx的做法是设置该请求的一些状态并在epoll中添加相应的事件，然后转去处理其他请求，等到该事件到来时再继续处理该请求，这样的行为就意味着一个请求可能需要多次执行机会才能完成，对于一个请求的多个子请求来说，意味着它们完成的先后顺序可能和它们创建的顺序是不一样的，所以必须有一种机制让提前完成的子请求保存它产生的数据，而不是直接输出到out chain，同时也能够让当前能够往out chain输出数据的请求及时的输出产生的数据。作者Igor采用ngx\_connection\_t中的data字段，以及一个body filter，即ngx\_http\_postpone\_filter，还有ngx\_http\_finalize\_request函数中的一些逻辑来解决这个问题。

下面用一个图来做说明，下图是某时刻某个主请求和它的所有子孙请求的树结构：



图中的root节点即为主请求，它的postponed链表从左至右挂载了3个节点，SUB1是它的第一个子请求，DATA1是它产生的一段数据，SUB2是它的第2个子请求，而且这2个子请求分别有它们自己的子请求及数据。ngx\_connection\_t中的data字段保存的是当前可以往out chain发送数据的请求，文章开头说到发到客户端的数据必须按照子请求创建的顺序发送，这里即是按后序遍历的方法（SUB11->DATA11->SUB12->DATA12->(SUB1)->DATA1->SUB21->SUB22->(SUB2)->(ROOT)），上图中当前能够往客户端（out chain）发送数据的请求显然就是SUB11，如果SUB12提前执行完成，并产生数据DATA121，只要前面它还有节点未发送完毕，DATA121只能先挂载在SUB12的postponed链表下。这里还要注意一下的是c->data的设置，当SUB11执行完并且发送完数据之后，下一个将要发送的节点应该是DATA11，但是该节点实际上保存的是数据，而不是子请求，所以c->data这时应该指向的是拥有改数据节点的SUB1请求。

下面看下源码具体是怎样实现的，首先是ngx\_http\_postpone\_filter函数：

static ngx\_int\_t  
ngx\_http\_postpone\_filter(ngx\_http\_request\_t \*r, ngx\_chain\_t \*in)  
{  
 ...  
 /\* 当前请求不能往out chain发送数据，如果产生了数据，新建一个节点，  
 将它保存在当前请求的postponed队尾。这样就保证了数据按序发到客户端 \*/  
 if (r != c->data) {   
  
 if (in) {  
 ngx\_http\_postpone\_filter\_add(r, in);  
 return NGX\_OK;  
 }  
 ...  
 return NGX\_OK;  
 }  
 /\* 到这里，表示当前请求可以往out chain发送数据，如果它的postponed链表中没有子请求，也没有数据，  
 则直接发送当前产生的数据in或者继续发送out chain中之前没有发送完成的数据 \*/  
 if (r->postponed == NULL) {   
  
 if (in || c->buffered) {  
 return ngx\_http\_next\_filter(r->main, in);  
 }  
 /\* 当前请求没有需要发送的数据 \*/  
 return NGX\_OK;  
 }  
 /\* 当前请求的postponed链表中之前就存在需要处理的节点，则新建一个节点，保存当前产生的数据in，  
 并将它插入到postponed队尾 \*/  
 if (in) {   
 ngx\_http\_postpone\_filter\_add(r, in);  
 }  
 /\* 处理postponed链表中的节点 \*/  
 do {   
 pr = r->postponed;  
 /\* 如果该节点保存的是一个子请求，则将它加到主请求的posted\_requests链表中，  
 以便下次调用ngx\_http\_run\_posted\_requests函数，处理该子节点 \*/  
 if (pr->request) {  
  
 ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, c->log, 0,  
 "http postpone filter wake \"%V?%V\"",  
 &pr->request->uri, &pr->request->args);  
  
 r->postponed = pr->next;  
  
 /\* 按照后序遍历产生的序列，因为当前请求（节点）有未处理的子请求(节点)，  
 必须先处理完改子请求，才能继续处理后面的子节点。  
 这里将该子请求设置为可以往out chain发送数据的请求。 \*/  
 c->data = pr->request;  
 /\* 将该子请求加入主请求的posted\_requests链表 \*/  
 return ngx\_http\_post\_request(pr->request, NULL);  
 }  
 /\* 如果该节点保存的是数据，可以直接处理该节点，将它发送到out chain \*/  
 if (pr->out == NULL) {  
 ngx\_log\_error(NGX\_LOG\_ALERT, c->log, 0,  
 "http postpone filter NULL output",  
 &r->uri, &r->args);  
  
 } else {  
 ngx\_log\_debug2(NGX\_LOG\_DEBUG\_HTTP, c->log, 0,  
 "http postpone filter output \"%V?%V\"",  
 &r->uri, &r->args);  
  
 if (ngx\_http\_next\_filter(r->main, pr->out) == NGX\_ERROR) {  
 return NGX\_ERROR;  
 }  
 }  
  
 r->postponed = pr->next;  
  
 } while (r->postponed);  
  
 return NGX\_OK;  
}

再来看ngx\_http\_finalzie\_request函数：

void  
ngx\_http\_finalize\_request(ngx\_http\_request\_t \*r, ngx\_int\_t rc)   
{  
 ...  
 /\* 如果当前请求是一个子请求，检查它是否有回调handler，有的话执行之 \*/  
 if (r != r->main && r->post\_subrequest) {  
 rc = r->post\_subrequest->handler(r, r->post\_subrequest->data, rc);  
 }  
  
 ...  
  
 /\* 子请求 \*/  
 if (r != r->main) {   
 /\* 该子请求还有未处理完的数据或者子请求 \*/  
 if (r->buffered || r->postponed) {  
 /\* 添加一个该子请求的写事件，并设置合适的write event hander，  
 以便下次写事件来的时候继续处理，这里实际上下次执行时会调用ngx\_http\_output\_filter函数，  
 最终还是会进入ngx\_http\_postpone\_filter进行处理 \*/  
 if (ngx\_http\_set\_write\_handler(r) != NGX\_OK) {  
 ngx\_http\_terminate\_request(r, 0);  
 }  
  
 return;  
 }  
 ...  
  
 pr = r->parent;  
  
  
 /\* 该子请求已经处理完毕，如果它拥有发送数据的权利，则将权利移交给父请求， \*/  
 if (r == c->data) {   
  
 r->main->count--;  
  
 if (!r->logged) {  
  
 clcf = ngx\_http\_get\_module\_loc\_conf(r, ngx\_http\_core\_module);  
  
 if (clcf->log\_subrequest) {  
 ngx\_http\_log\_request(r);  
 }  
  
 r->logged = 1;  
  
 } else {  
 ngx\_log\_error(NGX\_LOG\_ALERT, c->log, 0,  
 "subrequest: \"%V?%V\" logged again",  
 &r->uri, &r->args);  
 }  
  
 r->done = 1;  
 /\* 如果该子请求不是提前完成，则从父请求的postponed链表中删除 \*/  
 if (pr->postponed && pr->postponed->request == r) {  
 pr->postponed = pr->postponed->next;  
 }  
 /\* 将发送权利移交给父请求，父请求下次执行的时候会发送它的postponed链表中可以  
 发送的数据节点，或者将发送权利移交给它的下一个子请求 \*/  
 c->data = pr;   
  
 } else {  
 /\* 到这里其实表明该子请求提前执行完成，而且它没有产生任何数据，则它下次再次获得  
 执行机会时，将会执行ngx\_http\_request\_finalzier函数，它实际上是执行  
 ngx\_http\_finalzie\_request（r,0），也就是什么都不干，直到轮到它发送数据时，  
 ngx\_http\_finalzie\_request函数会将它从父请求的postponed链表中删除 \*/  
 r->write\_event\_handler = ngx\_http\_request\_finalizer;  
  
 if (r->waited) {  
 r->done = 1;  
 }  
 }  
 /\* 将父请求加入posted\_request队尾，获得一次运行机会 \*/  
 if (ngx\_http\_post\_request(pr, NULL) != NGX\_OK) {  
 r->main->count++;  
 ngx\_http\_terminate\_request(r, 0);  
 return;  
 }  
  
 return;  
 }  
 /\* 这里是处理主请求结束的逻辑，如果主请求有未发送的数据或者未处理的子请求，  
 则给主请求添加写事件，并设置合适的write event hander，  
 以便下次写事件来的时候继续处理 \*/  
 if (r->buffered || c->buffered || r->postponed || r->blocked) {  
  
 if (ngx\_http\_set\_write\_handler(r) != NGX\_OK) {  
 ngx\_http\_terminate\_request(r, 0);  
 }  
  
 return;  
 }  
  
 ...  
}

## https请求处理解析

### nginx支持ssl简介

nginx-1.2.0编译时默认是不支持ssl协议的，需要通过编译指令来开启对其支持：

./configure --with-http\_ssl\_module

在nginx源码中，ssl相关代码用宏定义变量NGX\_HTTP\_SSL来控制是否开启。这给我们查找和阅读ssl相关代码带来了方便，如下:

#if NGX\_HTTP\_SSL  
/\* http ssl code \*/  
#endif}

ssl协议工作在tcp协议与http协议之间。nginx在支持ssl协议时，需要注意三点，其他时候只要正常处理http协议即可:

1. tcp连接建立时，在tcp连接上建立ssl连接
2. tcp数据接收后，将收到的数据解密并将解密后的数据交由正常http协议处理流程
3. tcp数据发送前，对(http)数据进行加密，然后再发送

以下章节将分别介绍这三点。

### ssl连接建立(ssl握手)

#### 对ssl连接建立的准备

更具ssl协议规定，在正式发起数据收发前，需要建立ssl连接，连接建立过程既ssl握手。nginx在创建和初始化http请求阶段的同时为tcp连接建立做准备，主要流程在ngx\_http\_init\_request函数中实现:

static void  
ngx\_http\_init\_request(ngx\_event\_t \*rev)  
{  
  
...  
  
#if (NGX\_HTTP\_SSL)  
  
 {  
 ngx\_http\_ssl\_srv\_conf\_t \*sscf;  
  
 sscf = ngx\_http\_get\_module\_srv\_conf(r, ngx\_http\_ssl\_module);  
 if (sscf->enable || addr\_conf->ssl) {  
  
 /\* c->ssl不为空时，表示请求复用长连接(已经建立过ssl连接) \*/  
 if (c->ssl == NULL) {  
  
 c->log->action = "SSL handshaking";  
  
 /\*  
 \* nginx.conf中开启ssl协议(listen 443 ssl;)，  
 \* 却没用设置服务器证书(ssl\_certificate <certificate\_path>;)  
 \*/  
 if (addr\_conf->ssl && sscf->ssl.ctx == NULL) {  
 ngx\_log\_error(NGX\_LOG\_ERR, c->log, 0,  
 "no \"ssl\_certificate\" is defined "  
 "in server listening on SSL port");  
 ngx\_http\_close\_connection(c);  
 return;  
 }  
  
 /\*   
 \* 创建ngx\_ssl\_connection\_t并初始化  
 \* openssl库中关于ssl连接的初始化  
 \*/  
 if (ngx\_ssl\_create\_connection(&sscf->ssl, c, NGX\_SSL\_BUFFER)  
 != NGX\_OK)  
 {  
 ngx\_http\_close\_connection(c);  
 return;  
 }  
  
 rev->handler = ngx\_http\_ssl\_handshake;  
 }  
  
 /\* ssl加密的数据必须读到内存中 \*/  
 r->main\_filter\_need\_in\_memory = 1;  
 }  
 }  
  
#endif  
  
...  
  
}

ngx\_http\_init\_request大部分流程已经在前面章节分析过了，这个函数主要负责初始化http请求，此时并没有实际解析http请求。若发来的请求是经由ssl协议加密的，直接解析http请求就会出错。ngx\_http\_init\_request中ssl协议相关处理流程:

1，首先判断c->ssl是否为空。若不为空：说明这里是http长连接的情况，ssl连接已经在第一个请求进入时建立了。这里只要复用这个ssl连接即可，跳过ssl握手阶段。

2.(1)，若c->ssl为空：需要进行ssl握手来建立连接。此时调用ngx\_ssl\_create\_connection为ssl连接建立做准备。

ngx\_ssl\_create\_connection 简化代码如下:

ngx\_int\_t  
ngx\_ssl\_create\_connection(ngx\_ssl\_t \*ssl, ngx\_connection\_t \*c, ngx\_uint\_t flags)  
{  
 ngx\_ssl\_connection\_t \*sc;  
  
 /\* ngx\_ssl\_connection\_t是nginx对ssl连接的描述结构，记录了ssl连接的信息和状态 \*/  
 sc = ngx\_pcalloc(c->pool, sizeof(ngx\_ssl\_connection\_t));  
  
 sc->buffer = ((flags & NGX\_SSL\_BUFFER) != 0);  
  
 /\* 创建openssl库中对ssl连接的描述结构 \*/  
 sc->connection = SSL\_new(ssl->ctx);  
  
 /\* 关联(openssl库)ssl连接到tcp连接对应的socket \*/  
 SSL\_set\_fd(sc->connection, c->fd);  
  
 if (flags & NGX\_SSL\_CLIENT) {  
 /\* upstream中发起对后端的ssl连接，指明nginx ssl连接是客户端 \*/  
 SSL\_set\_connect\_state(sc->connection);  
  
 } else {  
 /\* 指明nginx ssl连接是服务端 \*/  
 SSL\_set\_accept\_state(sc->connection);  
 }  
  
 /\* 关联(openssl库)ssl连接到用户数据(当前连接c) \*/  
 SSL\_set\_ex\_data(sc->connection, ngx\_ssl\_connection\_index, c);  
  
 c->ssl = sc;  
  
 return NGX\_OK;  
}

2.(2)，设置连接读事件处理函数为ngx\_http\_ssl\_handshake，这将改变后续处理http请求的正常流程为：先进行ssl握手，再正常处理http请求。

3，标明当前待发送的数据须在内存中，以此可以让ssl对数据进行加密。由于开启了ssl协议，对发送出去的数据要进行加密，这就要求待发送的数据必须在内存中。 标识r->main\_filter\_need\_in\_memory为1，可以让后续数据发送前，将数据读取到内存中 (防止在文件中的数据通过sendfile直接发送出去，而没有加密）。

#### 实际ssl握手阶段

由于在ngx\_http\_init\_request中将连接读事件处理函数设置成ngx\_http\_ssl\_handshake，当连接中有可读数据时，将会进入ngx\_http\_ssl\_handshake来处理(若未开启ssl，将进入ngx\_http\_process\_request\_line直接解析http请求）

在ngx\_http\_ssl\_handshake中，来进行ssl握手:

1，首先判断连接是否超时，如果超时则关闭连接

static void  
ngx\_http\_process\_request(ngx\_http\_request\_t \*r)  
{  
 if (rev->timedout) {  
 ngx\_log\_error(NGX\_LOG\_INFO, c->log, NGX\_ETIMEDOUT, "client timed out");  
 c->timedout = 1;  
 ngx\_http\_close\_request(r, NGX\_HTTP\_REQUEST\_TIME\_OUT);  
 return;  
 }

2，首字节预读：从tcp连接中查看一个字节(通过MSG\_PEEK查看tcp连接中数据，但不会实际读取该数据)，若tcp连接中没有准备好的数据，则重新添加读事件退出等待新数据到来。

n = recv(c->fd, (char \*) buf, 1, MSG\_PEEK);  
  
if (n == -1 && ngx\_socket\_errno == NGX\_EAGAIN) {  
  
 if (!rev->timer\_set) {  
 ngx\_add\_timer(rev, c->listening->post\_accept\_timeout);  
 }  
  
 if (ngx\_handle\_read\_event(rev, 0) != NGX\_OK) {  
 ngx\_http\_close\_request(r, NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);  
 }  
  
 return;  
}

3，首字节探测：若成功查看1个字节数据，通过该首字节来探测接受到的数据是ssl握手包还是http数据。根据ssl协议规定，ssl握手包的首字节中包含有ssl协议的版本信息。nginx根据此来判断是进行ssl握手还是返回正常处理http请求(实际返回应答400 BAD REQUEST)。

if (n == 1) {  
 if (buf[0] & 0x80 /\* SSLv2 \*/ || buf[0] == 0x16 /\* SSLv3/TLSv1 \*/) {  
 ngx\_log\_debug1(NGX\_LOG\_DEBUG\_HTTP, rev->log, 0,  
 "https ssl handshake: 0x%02Xd", buf[0]);  
  
 /\*   
 \* 调用ngx\_ssl\_handshake函数进行ssl握手，连接双方会在ssl握手时交换相  
 \* 关数据(ssl版本，ssl加密算法，server端的公钥等) 并正式建立起ssl连接。  
 \* ngx\_ssl\_handshake函数内部对openssl库进行了封装。  
 \* 调用SSL\_do\_handshake()来进行握手，并根据其返回值判断ssl握手是否完成  
 \* 或者出错。  
 \*/  
 rc = ngx\_ssl\_handshake(c);  
  
 /\*  
 \* ssl握手可能需要多次数据交互才能完成。  
 \* 如果ssl握手没有完成，ngx\_ssl\_handshake会根据具体情况(如需要读取更  
 \* 多的握手数据包，或者需要发送握手数据包）来重新添加读写事件  
 \*/  
 if (rc == NGX\_AGAIN) {  
  
 if (!rev->timer\_set) {  
 ngx\_add\_timer(rev, c->listening->post\_accept\_timeout);  
 }  
  
 c->ssl->handler = ngx\_http\_ssl\_handshake\_handler;  
 return;  
 }  
  
 /\*  
 \* 若ssl握手完成或者出错，ngx\_ssl\_handshake会返回NGX\_OK或者NGX\_ERROR, 然后ngx\_http\_ssl\_handshake调用  
 \* ngx\_http\_ssl\_handshake\_handler以继续处理  
 \*/  
  
 ngx\_http\_ssl\_handshake\_handler(c);  
  
 return;  
  
 } else {  
 ngx\_log\_debug0(NGX\_LOG\_DEBUG\_HTTP, rev->log, 0,  
 "plain http");  
  
 r->plain\_http = 1;  
 }  
}

需要特别注意，如果ssl握手完成，ngx\_ssl\_handshake会替换连接的读写接口。这样，后续需要读写数据时，替换的接口会对数据进行加密解密。详细代码见下:

ngx\_int\_t  
ngx\_ssl\_handshake(ngx\_connection\_t \*c)  
{  
 n = SSL\_do\_handshake(c->ssl->connection);  
 /\* 返回1表示ssl握手成功 \*/  
 if (n == 1) {   
...  
 c->ssl->handshaked = 1;  
  
 c->recv = ngx\_ssl\_recv;  
 c->send = ngx\_ssl\_write;  
 c->recv\_chain = ngx\_ssl\_recv\_chain;  
 c->send\_chain = ngx\_ssl\_send\_chain;  
  
 return NGX\_OK;  
 }  
...  
}

4，探测为http协议：正常的http协议包处理直接调用ngx\_http\_process\_request\_line处理http请求，并将读事件处理函数设置成ngx\_http\_process\_request\_line。(实际处理结果是向客户端返回400 BAD REQUET，在ngx\_http\_process\_request中又对r->plain\_http标志的单独处理。)

c->log->action = "reading client request line";  
  
rev->handler = ngx\_http\_process\_request\_line;  
ngx\_http\_process\_request\_line(rev);

} /\* end of ngx\_http\_process\_request() \*/

5，当ssl握手成功或者出错时，调用ngx\_http\_ssl\_handshake\_handler函数。

5.(1)，若ssl握手完成 (c->ssl->handshaked由ngx\_ssl\_handshake()确定握手完成后设为1)，设置读事件处理函数为ngx\_http\_process\_request\_line，并调用此函数正常处理http请求。

5.(2)，若ssl握手没完成（则说明ssl握手出错），则返回400 BAD REQUST给客户端。

至此，ssl连接已经建立，此后在ngx\_http\_process\_request中会读取数据并解密然后正常处理http请求。

static void  
ngx\_http\_ssl\_handshake\_handler(ngx\_connection\_t \*c)  
{  
 ngx\_http\_request\_t \*r;  
  
 if (c->ssl->handshaked) {  
  
 /\*  
 \* The majority of browsers do not send the "close notify" alert.  
 \* Among them are MSIE, old Mozilla, Netscape 4, Konqueror,  
 \* and Links. And what is more, MSIE ignores the server's alert.  
 \*  
 \* Opera and recent Mozilla send the alert.  
 \*/  
  
 c->ssl->no\_wait\_shutdown = 1;  
  
 c->log->action = "reading client request line";  
  
 c->read->handler = ngx\_http\_process\_request\_line;  
 /\* STUB: epoll edge \*/ c->write->handler = ngx\_http\_empty\_handler;  
  
 ngx\_http\_process\_request\_line(c->read);  
  
 return;  
 }  
  
 r = c->data;  
  
 ngx\_http\_close\_request(r, NGX\_HTTP\_BAD\_REQUEST);  
  
 return;  
}

#### ssl协议接受数据

ngx\_http\_process\_request中处理http请求，需要读取和解析http协议。而实际数据读取是通过c->recv()函数来读取的，此函数已经在ngx\_ssl\_handshake中被替换成ngx\_ssl\_recv了。

ngx\_ssl\_recv函数中调用openssl库函数SSL\_read()来读取并解密数据，简化后如下：

ssize\_t ngx\_ssl\_recv(ngx\_connection\_t \*c, u\_char \*buf, size\_t size)  
{  
...  
 n = SSL\_read(c->ssl->connection, buf, size);  
...  
 return n;  
}

#### ssl协议发送数据

当nginx发送数据时，如使用ngx\_output\_chain函数发送缓存的http数据缓存链时，通过调用c->send\_chain()来发送数据。这个函数已经在ngx\_ssl\_handshake中被设置成ngx\_ssl\_send\_chain了。ngx\_ssl\_send\_chain会进一步调用ngx\_ssl\_write。而ngx\_ssl\_write调用openssl库SSL\_write函数来加密并发送数据。

/\* ngx\_output\_chain  
 \* -> ..  
 \* -> ngx\_chain\_writer  
 \* -> c->send\_chain (ngx\_ssl\_send\_chain)   
 \* -> ngx\_ssl\_write  
 \*/  
ssize\_t ngx\_ssl\_write(ngx\_connection\_t \*c, u\_char \*data, size\_t size)  
{  
...  
 n = SSL\_write(c->ssl->connection, data, size);  
...  
 return n;  
}