Nginx 源码分析之 Filter 与 Upstream

第一章 Filter 模块

1.1 过滤模块简介

1. 执行时间和内容

过滤（filter）模块是过滤响应头和内容的模块，可以对回复的头和内容进行处理。它的处理时间在获取回复内容之后，向用户发送响应之前。它的处理过程分为两个阶段，过滤 HTTP 回复的头部和主体，在这两个阶段可以分别对头部和主体进行修改。
在代码中有类似的函数：

```
ngx_http_top_header_filter(r);
ngx_http_top_body_filter(r, in);
```

就是分别对头部和主体进行过滤的函数。所有模块的响应内容要返回给客户端，都必须调用这两个接口。

2. 执行顺序

过滤模块的调用是有顺序的，它的顺序在编译的时候就决定了。控制编译的脚本位于 auto/modules 中，当你编译完 Nginx 以后，可以在 objs 目录下面看到一个 ngx_modules.c 的文件。打开这个文件，有类似的代码：

```
ngx_module_t *ngx_modules[] = {
        ...
        &ngx_http_write_filter_module,
        &ngx_http_header_filter_module,
        &ngx_http_chunked_filter_module,
        &ngx_http_range_header_filter_module,
        &ngx_http_gzip_filter_module,
        &ngx_http_postpone_filter_module,
        &ngx_http_ssi_filter_module,
        &ngx_http_charset_filter_module,
        &ngx_http_userid_filter_module,
        &ngx_http_headers_filter_module,
        &ngx_http_copy_filter_module,
        &ngx_http_range_body_filter_module,
        &ngx_http_not_modified_filter_module,
```

```
        NULL
};
```

从 write_filter 到 not_modified_filter，模块的执行顺序是反向的。也就是说最早执行的是 not_modified_filter，然后各个模块依次执行。所有第三方的模块只能加入到 copy_filter 和 headers_filter 模块之间执行。

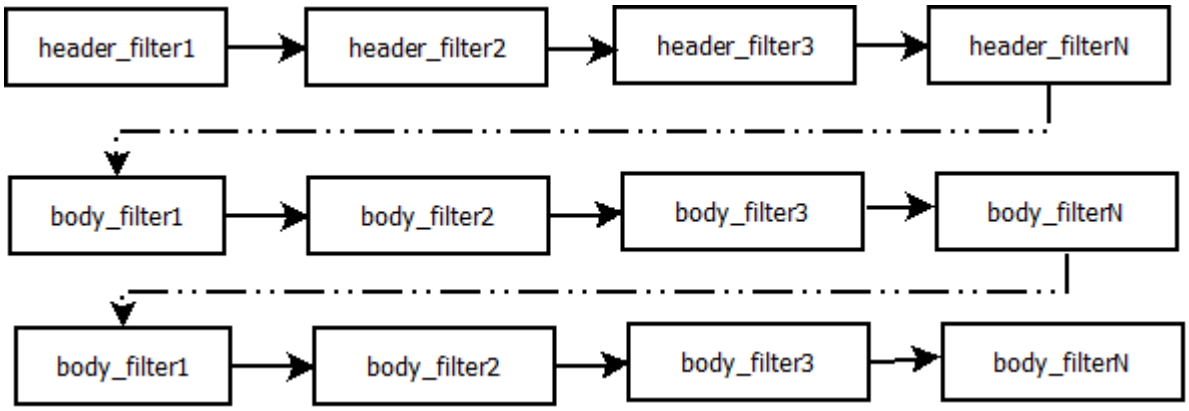
Nginx 执行的时候是怎么按照次序依次来执行各个过滤模块呢？它采用了一种很隐晦的方法，即通过局部的全局变量。比如，在每个 filter 模块，很可能看到如下代码：

```
static ngx_http_output_header_filter_pt
ngx_http_next_header_filter;
static ngx_http_output_body_filter_pt
ngx_http_next_body_filter;

...

ngx_http_next_header_filter = ngx_http_top_header_filter;
ngx_http_top_header_filter = ngx_http_example_header_filter;

ngx_http_next_body_filter = ngx_http_top_body_filter;
ngx_http_top_body_filter = ngx_http_example_body_filter;
```

ngx_http_top_header_filter 是一个全局变量。当编译进一个 filter 模块的时候，就被赋值为当前 filter 模块的处理函数。而 ngx_http_next_header_filter 是一个局部全局变量，它保存了编译前上一个 filter 模块的处理函数。所以整体看来，就像用全局变量组成的一条单向链表。

每个模块想执行下一个过滤函数，只要调用一下 ngx_http_next_header_filter 这个局部变量。而整个过滤模块链的入口，需要调用 ngx_http_top_header_filter 这个全局变量。ngx_http_top_body_filter 的行为与 header fitler 类似。

响应头和响应体过滤函数的执行顺序如下所示：



这图只表示了 head_filter 和 body_filter 之间的执行顺序，在 header_filter 和 body_filter 处理函数之间，在 body_filter 处理函数之间，可能还有其他执行代码。

3. 模块编译

Nginx 可以方便的加入第三方的过滤模块。在过滤模块的目录里，首先需要加入 config 文件，文件的内容如下：

```
ngx_addon_name=ngx_http_example_filter_module
HTTP_AUX_FILTER_MODULES="$HTTP_AUX_FILTER_MODULES
ngx_http_example_filter_module"
NGX_ADDON_SRCS="$NGX_ADDON_SRCS
$ngx_addon_dir/ngx_http_example_filter_module.c"
```

说明把这个名为 ngx_http_example_filter_module 的过滤模块加入，ngx_http_example_filter_module.c 是该模块的源代码。
注意 HTTP_AUX_FILTER_MODULES 这个变量与一般的内容处理模块不同。

1.2 过滤模块的分析

1. 相关结构体

ngx_chain_t 结构非常简单，是一个单向链表：

```
typedef struct ngx_chain_s ngx_chain_t;

        struct ngx_chain_s {
                ngx_buf_t    *buf;
                ngx_chain_t  *next;
        };
```

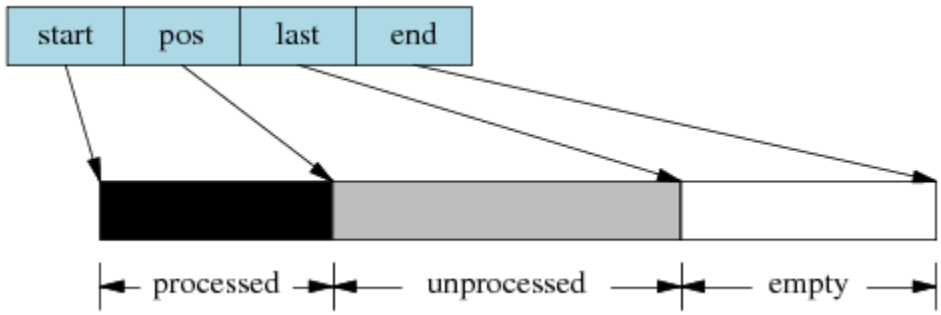
在过滤模块中，所有输出的内容都是通过一条单向链表所组成。这种单向链表的设计，正好应和了 Nginx 流式的输出模式。每次 Nginx 都是读到一部分的内容，就放到链表，然后输出出去。这种设计的好处是简单，非阻塞，但是相应的问题就是跨链表的内容操作非常麻烦，如果需要跨链表，很多时候都只能缓存链表的内容。
单链表负载的就是 ngx_buf_t，这个结构体使用非常广泛，先让我们看下该结构体的代码：

```
struct ngx_buf_s {
        u_char   *pos;  /* 当前 buffer 真实内容的起始位置 */
        u_char   *last; /* 当前 buffer 真实内容的结束位置 */
        off_t    file_pos; /* 在文件中真实内容的起始位置    */
        off_t    file_last; /* 在文件中真实内容的结束位置    */
        u_char   *start;    /* buffer 内存的开始分配的位置 */
        u_char   *end;      /* buffer 内存的结束分配的位置 */
        ngx_buf_tag_t    tag;       /* buffer 属于哪个模块的标志 */
        ngx_file_t      *file;      /* buffer 所引用的文件 */
```

```
        /* 用来引用替换过后的 buffer，以便当所有 buffer 输出以后，
         * 这个影子 buffer 可以被释放。
         */
        ngx_buf_t       *shadow;

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

        /*
         * the buf's content is in a memory cache or in a read
only memory
         * and must not be changed
         */
        unsigned         memory:1;

        /* the buf's content is mmap()ed and must not be changed
*/
        unsigned         mmap:1;

        unsigned         recycled:1; /* 内存可以被输出并回收 */
        unsigned         in_file:1;  /* buffer 的内容在文件中 */
        /* 马上全部输出 buffer 的内容，gzip 模块里面用得比较多 */
        unsigned         flush:1;
        /* 基本上是一段输出链的最后一个 buffer 带的标志，标示可以输
出，
         * 有些零长度的 buffer 也可以置该标志
         */
        unsigned         sync:1;
        /* 所有请求里面最后一块 buffer，包含子请求 */
        unsigned         last_buf:1;
        /* 当前请求输出链的最后一块 buffer         */
        unsigned         last_in_chain:1;
        /* shadow 链里面的最后 buffer，可以释放 buffer 了 */
        unsigned         last_shadow:1;
        /* 是否是暂存文件 */
        unsigned         temp_file:1;

        /* 统计用，表示使用次数 */
        /* STUB */ int   num;
};
```

一般 buffer 结构体可以表示一块内存，内存的起始和结束地址分别用 start 和 end 表示，pos 和 last 表示实际的内容。如果内容已经处理过了，pos 的位置就可以往后移动。如果读取到新的内容，last 的位置就会往后移动。所以 buffer 可以在多次调用过程中使用。如果 last 等

于 end，就说明这块内存已经用完了。如果 pos 等于 last，说明内存已经处理完了。下面是一个简单的示意图，说明 buffer 中指针的用法：



2. 响应头过滤函数

响应头过滤函数主要的用处就是处理 HTTP 响应的头，可以根据实际情况对于响应头进行修改或者添加删除。响应头过滤函数先于响应体过滤函数，而且只调用一次，所以一般可作过滤模块的初始化工作。

响应头过滤函数的入口只有一个：

```
ngx_int_t
ngx_http_send_header(ngx_http_request_t *r)
{
        ...

        return ngx_http_top_header_filter(r);
}
```

该函数向客户端发送回复的时候调用，然后按前一节所述的执行顺序。该函数的返回值一般是 NGX_OK，NGX_ERROR 和 NGX_AGAIN，分别表示处理成功，失败和未完成。可以把 HTTP 响应头的存储方式想象成一个 hash 表，在 Nginx 内部可以很方便地查找和修改各个响应头部，ngx_http_header_filter_module 过滤模块把所有的 HTTP 头组合成一个完整的 buffer，最终 ngx_http_write_filter_module 过滤模块把 buffer 输出。

按照前一节过滤模块的顺序，依次讲解如下：

filter module	**description**
ngx_http_not_modified_filter_module	默认打开，如果请求的 if-modified-since 等于回复的 last-modified 间值，说明回复没有变化，清空所有回复的内容，返回 304。
ngx_http_range_body_filter_module	默认打开，只是响应体过滤函数，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_copy_filter_module	始终打开，只是响应体过滤函数， 主要工作是把文件中内容读到内存中，以便进行处理。
ngx_http_headers_filter_module	始终打开，可以设置 expire 和 Cache-control 头，可以添加任意名称的头

filter module	description
ngx_http_userid_filter_module	默认关闭，可以添加统计用的识别用户的 cookie。
ngx_http_charset_filter_module	默认关闭，可以添加 charset，也可以将内容从一种字符集转换到另外一种字符集，不支持多字节字符集。
ngx_http_ssi_filter_module	默认关闭，过滤 SSI 请求，可以发起子请求，去获取 include 进来的文件
ngx_http_postpone_filter_module	始终打开，用来将子请求和主请求的输出链合并
ngx_http_gzip_filter_module	默认关闭，支持流式的压缩内容
ngx_http_range_header_filter_module	默认打开，只是响应头过滤函数，用来解析 range 头，并产生 range 响应的头。
ngx_http_chunked_filter_module	默认打开，对于 HTTP/1.1 和缺少 content-length 的回复自动打开。
ngx_http_header_filter_module	始终打开，用来将所有 header 组成一个完整的 HTTP 头。
ngx_http_write_filter_module	始终打开，将输出链拷贝到 r->out 中，然后输出内容。

3. 响应体过滤函数

响应体过滤函数是过滤响应主体的函数。ngx_http_top_body_filter 这个函数每个请求可能会被执行多次，它的入口函数是 ngx_http_output_filter，比如：

```
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;

    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;
}
```

ngx_http_output_filter 可以被一般的静态处理模块调用，也有可能是在 upstream 模块里面被调用，对于整个请求的处理阶段来说，他们处于的用处都是一样的，就是把响应内容过滤，

然后发给客户端。具体模块的响应体过滤函数的格式类似这样：

```
static int
ngx_http_example_body_filter(ngx_http_request_t *r, ngx_chain_t
*in)
{
        ...

        return ngx_http_next_body_filter(r, in);
}
```

该函数的返回值一般是 NGX_OK，NGX_ERROR 和 NGX_AGAIN，分别表示处理成功，失败和未完成。

4. 主要功能介绍

响应的主体内容就存于单链表 in，链表一般不会太长，有时 in 参数可能为 NULL。in 中存有 buf 结构体中，对于静态文件，这个 buf 大小默认是 32K；对于反向代理的应用，这个 buf 可能是 4k 或者 8k。为了保持内存的低消耗，Nginx 一般不会分配过大的内存，处理的原则是收到一定的数据，就发送出去。一个简单的例子，可以看看 Nginx 的 chunked_filter 模块，在没有 content-length 的情况下，chunk 模块可以流式（stream）的加上长度，方便浏览器接收和显示内容。

在响应体过滤模块中，尤其要注意的是 buf 的标志位，完整描述可以在“相关结构体”这个节中看到。如果 buf 中包含 last 标志，说明是最后一块 buf，可以直接输出并结束请求了。如果有 flush 标志，说明这块 buf 需要马上输出，不能缓存。如果整块 buffer 经过处理完以后，没有数据了，你可以把 buffer 的 sync 标志置上，表示只是同步的用处。

当所有的过滤模块都处理完毕时，在最后的 write_fitler 模块中，Nginx 会将 in 输出链拷贝到 r->out 输出链的末尾，然后调用 sendfile 或者 writev 接口输出。由于 Nginx 是非阻塞的 socket 接口，写操作并不一定会成功，可能会有部分数据还残存在 r->out。在下次的调用中，Nginx 会继续尝试发送，直至成功。

5. 发出子请求

Nginx 过滤模块一大特色就是可以发出子请求，也就是在过滤响应内容的时候，你可以发送新的请求，Nginx 会根据你调用的先后顺序，将多个回复的内容拼接成正常的响应主体。一个简单的例子可以参考 addition 模块。

Nginx 是如何保证父请求和子请求的顺序呢？当 Nginx 发出子请求时，就会调用 ngx_http_subrequest 函数，将子请求插入父请求的 r->postponed 链表中。子请求会在主请求执行完毕时获得依次调用。子请求同样会有一个请求所有的生存期和处理过程，也会进入过滤模块流程。

关键点是在 postpone_filter 模块中，它会拼接主请求和子请求的响应内容。r->postponed 按次序保存有父请求和子请求，它是一个链表，如果前面一个请求未完成，那后一个请求内容就不会输出。当前一个请求完成时并输出时，后一个请求才可输出，当所有的子请求都完成时，所有的响应内容也就输出完毕了。

6. 一些优化措施

Nginx 过滤模块涉及到的结构体，主要就是 chain 和 buf，非常简单。在日常的过滤模块中，这两类结构使用非常频繁，Nginx 采用类似 freelist 重复利用的原则，将使用完毕的 chain 或者 buf 结构体，放置到一个固定的空闲链表里，以待下次使用。
比如，在通用内存池结构体中，pool->chain 变量里面就保存着释放的 chain。而一般的 buf 结构体，没有模块间公用的空闲链表池，都是保存在各模块的缓存空闲链表池里面。对于 buf 结构体，还有一种 busy 链表，表示该链表中的 buf 都处于输出状态，如果 buf 输出完毕，这些 buf 就可以释放并重复利用了。

功能	函数名
chain 分配	ngx_alloc_chain_link
chain 释放	ngx_free_chain
buf 分配	ngx_chain_get_free_buf
buf 释放	ngx_chain_update_chains

7. 过滤内容的缓存

由于 Nginx 设计流式的输出结构，当我们需要对响应内容作全文过滤的时候，必须缓存部分的 buf 内容。该类过滤模块往往比较复杂，比如 sub，ssi，gzip 等模块。这类模块的设计非常灵活，我简单讲一下设计原则：

1. 输入链 in 需要拷贝操作，经过缓存的过滤模块，输入输出链往往已经完全不一样了，所以需要拷贝，通过 ngx_chain_add_copy 函数完成。
2. 一般有自己的 free 和 busy 缓存链表池，可以提高 buf 分配效率。
3. 如果需要分配大块内容，一般分配固定大小的内存卡，并设置 recycled 标志，表示可以重复利用。
4. 原有的输入 buf 被替换缓存时，必须将其 buf->pos 设为 buf->last，表明原有的 buf 已经被输出完毕。或者在新建立的 buf，将 buf->shadow 指向旧的 buf，以便输出完毕时及时释放旧的 buf。

第二章 Upstream 模块

2.1 Upstream 模块

nginx 模块一般被分成三大类：handler、filter 和 upstream。前面的章节中，读者已经了解了 handler、filter。利用这两类模块，可以使 nginx 轻松完成任何单机工作。而本章介绍的 upstream 模块，将使 nginx 跨越单机的限制，完成网络数据的接收、处理和转发。
数据转发功能，为 nginx 提供了跨越单机的横向处理能力，使 nginx 摆脱只能为终端节点提供单一功能的限制，而使它具备了网路应用级别的拆分、封装和整合的战略功能。在云模型大行其道的今天，数据转发是 nginx 有能力构建一个网络应用的关键组件。当然，鉴于开发

成本的问题，一个网络应用的关键组件一开始往往会采用高级编程语言开发。但是当系统到达一定规模，并且需要更重视性能的时候，为了达到所要求的性能目标，高级语言开发出的组件必须进行结构化修改。此时，对于修改代价而言，nginx 的 upstream 模块呈现出极大的吸引力，因为它天生就快。作为附带，nginx 的配置系统提供的层次化和松耦合使得系统的扩展性也达到比较高的程度。言归正传，下面介绍 upstream 的写法。

从本质上说，upstream 属于 handler，只是他不产生自己的内容，而是通过请求后端服务器得到内容，所以才称为 upstream（上游）。请求并取得响应内容的整个过程已经被封装到 nginx 内部，所以 upstream 模块只需要开发若干回调函数，完成构造请求和解析响应等具体的工作。这些回调函数如下表所示：

create_request	生成发送到后端服务器的请求缓冲（缓冲链），在初始化 upstream 时使用。
reinit_request	在某台后端服务器出错的情况，nginx 会尝试另一台后端服务器。 nginx 选定新的服务器以后，会先调用此函数，以重新初始化 upstream 模块的工作状态，然后再次进行 upstream 连接。
process_header	处理后端服务器返回的信息头部。所谓头部是与 upstream server 通信的协议规定的，比如 HTTP 协议的 header 部分，或者 memcached 协议的响应状态部分。
abort_request	在客户端放弃请求时被调用。不需要在函数中实现关闭后端服务 器连接的功能，系统会自动完成关闭连接的步骤，所以一般此函 数不会进行任何具体工作。
finalize_request	正常完成与后端服务器的请求后调用该函数，与 abort_request 相同，一般也不会进行任何具体工作。
input_filter	处理后端服务器返回的响应正文。nginx 默认的 input_filter 会 将收到的内容封装成为缓冲区链 ngx_chain。该链由 upstream 的 out_bufs 指针域定位，所以开发人员可以在模块以外通过该指针 得到后端服务器返回的正文数据。memcached 模块实现了自己的 input_filter，在后面会具体分析这个模块。
input_filter_init	初始化 input filter 的上下文。nginx 默认的 input_filter_init 直接返回。

2.1.1 Memcached 模块分析

Memcached 是一款高性能的分布式 cache 系统，得到了非常广泛的应用。memcached 定义了一套私有通信协议，使得不能通过 HTTP 请求来访问 memcached。但协议本身简单高效，而且 memcached 使用广泛，所以大部分现代开发语言和平台都提供了 memcached 支持，方便开发者使用 memcached。

Nginx 提供了 ngx_http_memcached 模块，提供从 memcached 读取数据的功能，而不提供向 memcache 写数据的功能。作为 web 服务器，这种设计是可以接受的。
下面，我们开始分析 ngx_http_memcached 模块，一窥 upstream 的奥秘。

1. Handler 模块

初看 memcached 模块，大家可能觉得并无特别之处。如果稍微细看，甚至觉得有点像 handler 模块，当大家看到这段代码以后，必定疑惑为什么会跟 handler 模块一模一样。

```
clcf = ngx_http_conf_get_module_loc_conf(cf,
ngx_http_core_module);
clcf->handler = ngx_http_memcached_handler;
```

因为 upstream 模块使用的就是 handler 模块的接入方式。同时，upstream 模块的指令系统的设计也是遵循 handler 模块的基本规则：配置该模块才会执行该模块。

```
{ ngx_string("memcached_pass"),
  NGX_HTTP_LOC_CONF|NGX_HTTP_LIF_CONF|NGX_CONF_TAKE1,
  ngx_http_memcached_pass,
  NGX_HTTP_LOC_CONF_OFFSET,
  0,
  NULL }
```

所以大家觉得眼熟是好事，说明大家对 Handler 的写法已经很熟悉了。

2. Upstream 模块

那么，upstream 模块的特别之处究竟在哪里呢？答案是就在模块处理函数的实现中。upstream 模块的处理函数进行的操作都包含一个固定的流程。在 memcached 的例子中，可以观察 ngx_http_memcached_handler 的代码，可以发现，这个固定的操作流程是：

1. 创建 upstream 数据结构。

```
if (ngx_http_upstream_create(r) != NGX_OK) {
    return NGX_HTTP_INTERNAL_SERVER_ERROR;
}
```

2. 设置模块的 tag 和 schema。

schema 现在只会用于日志，tag 会用于 buf_chain 管理。

```
u = r->upstream;

ngx_str_set(&u->schema, "memcached://");
u->output.tag = (ngx_buf_tag_t) &ngx_http_memcached_module;
```

3. 设置 upstream 的后端服务器列表数据结构。

```
mlcf = ngx_http_get_module_loc_conf(r,
ngx_http_memcached_module);
u->conf = &mlcf->upstream;
```

4. 设置 upstream 回调函数。

在这里列出的代码稍稍调整了代码顺序。

```
u->create_request = ngx_http_memcached_create_request;
u->reinit_request = ngx_http_memcached_reinit_request;
u->process_header = ngx_http_memcached_process_header;
u->abort_request = ngx_http_memcached_abort_request;
u->finalize_request = ngx_http_memcached_finalize_request;
u->input_filter_init = ngx_http_memcached_filter_init;
u->input_filter = ngx_http_memcached_filter;
```

5. 创建并设置 upstream 环境数据结构。

```
ctx = ngx_palloc(r->pool, sizeof(ngx_http_memcached_ctx_t));
if (ctx == NULL) {
    return NGX_HTTP_INTERNAL_SERVER_ERROR;
}

ctx->rest = NGX_HTTP_MEMCACHED_END;
ctx->request = r;

ngx_http_set_ctx(r, ctx, ngx_http_memcached_module);

u->input_filter_ctx = ctx;
```

6. 完成 upstream 初始化并进行收尾工作。

```
r->main->count++;
ngx_http_upstream_init(r);
return NGX_DONE;
```

任何 upstream 模块，简单如 memcached，复杂如 proxy、fastcgi 都是如此。不同的 upstream 模块在这 6 步中的最大差别会出现在第 2、3、4、5 上。其中第 2、4 两步很容易理解，不同的模块设置的标志和使用的回调函数肯定不同。第 5 步也不难理解，只有第 3 步是最为晦涩的，不同的模块在取得后端服务器列表时，策略的差异非常大，有如 memcached 这样简单明了的，也有如 proxy 那样逻辑复杂的。这个问题先记下来，等把 memcached 剖析清楚了，再单独讨论。

第 6 步是一个常态。将 count 加 1，然后返回 NGX_DONE。nginx 遇到这种情况，虽然会认为当前请求的处理已经结束，但是不会释放请求使用的内存资源，也不会关闭与客户端的连接。之所以需要这样，是因为 nginx 建立了 upstream 请求和客户端请求之间一对一的关系，在后续使用 ngx_event_pipe 将 upstream 响应发送回客户端时，还要使用到这些保存着客户端信息的数据结构。这部分会在后面的原理篇做具体介绍，这里不再展开。

将 upstream 请求和客户端请求进行一对一绑定，这个设计有优势也有缺陷。优势就是简化模块开发，可以将精力集中在模块逻辑上，而缺陷同样明显，一对一的设计很多时候都不能满足复杂逻辑的需要。对于这一点，将会在后面的原理篇来阐述。

3. 回调函数

前面剖析了 memcached 模块的骨架，现在开始逐个解决每个回调函数。

1. ngx_http_memcached_create_request：很简单的按照设置的内容生成一个 key，接着生成一个“get $key”的请求，放在 r->upstream->request_bufs 里面。
2. ngx_http_memcached_reinit_request：无需初始化。
3. ngx_http_memcached_abort_request：无需额外操作。
4. ngx_http_memcached_finalize_request：无需额外操作。
5. ngx_http_memcached_process_header：模块的业务重点函数。

Memcached 协议的头部信息被定义为第一行文本，可以找到这段代码证明：

```
for (p = u->buffer.pos; p < u->buffer.last; p++) {
    if ( * p == LF) {
    goto found;
}
```

如果在已读入缓冲的数据中没有发现 LF(‘n’)字符，函数返回 NGX_AGAIN，表示头部未完全读入，需要继续读取数据。nginx 在收到新的数据以后会再次调用该函数。

nginx 处理后端服务器的响应头时只会使用一块缓存，所有数据都在这块缓存中，所以解析头部信息时不需要考虑头部信息跨越多块缓存的情况。而如果头部过大，不能保存在这块缓存中，nginx 会返回错误信息给客户端，并记录 error log，提示缓存不够大。

process_header 的重要职责是将后端服务器返回的状态翻译成返回给客户端的状态。例如，在 ngx_http_memcached_process_header 中，有这样几段代码：

```
r->headers_out.content_length_n = ngx_atoof(len, p - len - 1);

u->headers_in.status_n = 200;
u->state->status = 200;

u->headers_in.status_n = 404;
u->state->status = 404;
```

u->state 用于计算 upstream 相关的变量。比如 u->state->status 将被用于计算变量“upstream_status”的值。u->headers_in 将被作为返回给客户端的响应返回状态码。而第一行则是设置返回给客户端的响应的长度。

在这个函数中不能忘记的一件事情是处理完头部信息以后需要将读指针 pos 后移，否则这段数据也将被复制到返回给客户端的响应的正文中，进而导致正文内容不正确。

```
u->buffer.pos = p + 1;
```

process_header 函数完成响应头的正确处理，应该返回 NGX_OK。如果返回 NGX_AGAIN，表示未读取完整数据，需要从后端服务器继续读取数据。返回 NGX_DECLINED 无意义，其他任何返回值都被认为是出错状态，nginx 将结束 upstream 请求并返回错误信息。

6. ngx_http_memcached_filter_init：修正从后端服务器收到的内容长度。因为在处理 header 时没有加上这部分长度。
7. ngx_http_memcached_filter：memcached 模块是少有的带有处理正文的回调函数的模块。因为 memcached 模块需要过滤正文末尾 CRLF “END” CRLF，所以实现了自己的 filter 回调函数。处理正文的实际意义是将从后端服务器收到的正文有效内容封装成 ngx_chain_t，并加在 u->out_bufs 末尾。nginx 并不进行数据拷贝，而是建立 ngx_buf_t 数据结构指向这些数据内存区，然后由 ngx_chain_t 组织这些 buf。这种实现避免了内存大量搬迁，也是 nginx 高效的奥秘之一。

upstream 模块是从 handler 模块发展而来，指令系统和模块生效方式与 handler 模块无异。不同之处在于，upstream 模块在 handler 函数中设置众多回调函数。实际工作都是由这些回调函数完成的。每个回调函数都是在 upstream 的某个固定阶段执行，各司其职，大部分回调函数一般不会真正用到。upstream 最重要的回调函数是 create_request、process_header 和 input_filter，他们共同实现了与后端服务器的协议的解析部分。

2.2 负载均衡模块

负载均衡模块用于从”upstream”指令定义的后端主机列表中选取一台主机。nginx 先使用负载均衡模块找到一台主机，再使用 upstream 模块实现与这台主机的交互。为了方便介绍负载均衡模块，做到言之有物，以下选取 nginx 内置的 ip hash 模块作为实际例子进行分析。

1. 配置

要了解负载均衡模块的开发方法，首先需要了解负载均衡模块的使用方法。因为负载均衡模块与之前书中提到的模块差别比较大，所以我们从配置入手比较容易理解。
在配置文件中，我们如果需要使用 ip hash 的负载均衡算法。我们需要写一个类似下面的配置：

```
upstream test {
    ip_hash;

    server 192.168.0.1;
    server 192.168.0.2;
}
```

从配置我们可以看出负载均衡模块的使用场景： 1. 核心指令”ip_hash”只能在 upstream {}中使用。这条指令用于通知 nginx 使用 ip hash 负载均衡算法。如果没加这条指令，nginx 会使用默认的 round robin 负载均衡模块。请各位读者对比 handler 模块的配置，是不是有共同点？ 2. upstream {}中的指令可能出现在”server”指令前，可能出现在”server”指令后，也可能出现在两条”server”指令之间。各位读者可能会有疑问，有什么差别么？那么请各位读者尝试下面这个配置：

```
upstream test {
    server 192.168.0.1 weight=5;
    ip_hash;
    server 192.168.0.2 weight=7;
}
```

神奇的事情出现了：

```
nginx: [emerg] invalid parameter "weight=7" in nginx.conf:103
configuration file nginx.conf test failed
```

可见 ip_hash 指令的确能影响到配置的解析。

2. 指令

配置决定指令系统，现在就来看 ip_hash 的指令定义：

```
static ngx_command_t  ngx_http_upstream_ip_hash_commands[] = {

    { ngx_string("ip_hash"),
      NGX_HTTP_UPS_CONF|NGX_CONF_NOARGS,
      ngx_http_upstream_ip_hash,
      0,
      0,
      NULL },

    ngx_null_command
};
```

没有特别的东西，除了指令属性是 NGX_HTTP_UPS_CONF。这个属性表示该指令的适用范围是 upstream{}。

3. 钩子

以从前面的章节得到的经验，大家应该知道这里就是模块的切入点了。负载均衡模块的钩子代码都是有规律的，这里通过 ip_hash 模块来分析这个规律。

```
static char *
ngx_http_upstream_ip_hash(ngx_conf_t *cf, ngx_command_t *cmd,
void *conf)
{
    ngx_http_upstream_srv_conf_t  *uscf;

    uscf = ngx_http_conf_get_module_srv_conf(cf,
ngx_http_upstream_module);

    uscf->peer.init_upstream = ngx_http_upstream_init_ip_hash;

    uscf->flags = NGX_HTTP_UPSTREAM_CREATE
                |NGX_HTTP_UPSTREAM_MAX_FAILS
                |NGX_HTTP_UPSTREAM_FAIL_TIMEOUT
                |NGX_HTTP_UPSTREAM_DOWN;

    return NGX_CONF_OK;
}
```

这段代码中有两点值得我们注意。一个是 uscf->flags 的设置，另一个是设置 init_upstream 回调。

设置 uscf->flags

1. NGX_HTTP_UPSTREAM_CREATE：创建标志，如果含有创建标志的话，nginx 会检查重复创建，以及必要参数是否填写；
2. NGX_HTTP_UPSTREAM_MAX_FAILS：可以在 server 中使用 max_fails 属性；
3. NGX_HTTP_UPSTREAM_FAIL_TIMEOUT：可以在 server 中使用 fail_timeout 属性；
4. NGX_HTTP_UPSTREAM_DOWN：可以在 server 中使用 down 属性；

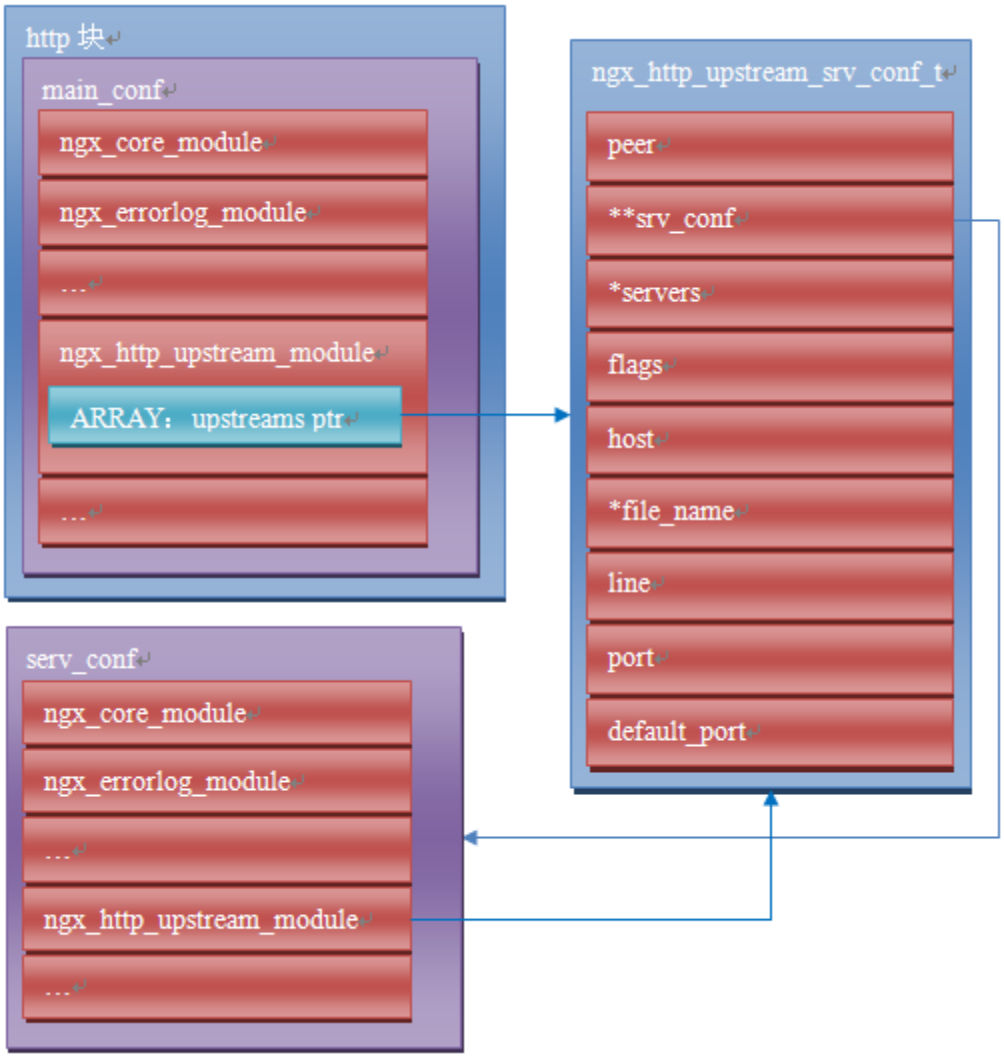
此外还有下面属性：

5. NGX_HTTP_UPSTREAM_WEIGHT：可以在 server 中使用 weight 属性；
6. NGX_HTTP_UPSTREAM_BACKUP：可以在 server 中使用 backup 属性。

聪明的读者如果联想到刚刚遇到的那个神奇的配置错误，可以得出一个结论：在负载均衡模块的指令处理函数中可以设置并修改 upstream{}中”server”指令支持的属性。这是一个很重要的性质，因为不同的负载均衡模块对各种属性的支持情况都是不一样的，那么就需要在解析配置文件的时候检测出是否使用了不支持的负载均衡属性并给出错误提示，这对于提升系统维护性是很有意义的。但是，这种机制也存在缺陷，正如前面的例子所示，没有机制能够追加检查在更新支持属性之前已经配置了不支持属性的”server”指令。

设置 init_upstream 回调

nginx 初始化 upstream 时，会在 ngx_http_upstream_init_main_conf 函数中调用设置的回调函数初始化负载均衡模块。这里不太好理解的是 uscf 的具体位置。通过下面的示意图，说明 upstream 负载均衡模块的配置的内存布局。



从图上可以看出，MAIN_CONF 中 ngx_upstream_module 模块的配置项中有一个指针数组 upstreams，数组中的每个元素对应就是配置文件中每一个 upstream{}的信息。更具体的将会在后面的原理篇讨论。

4. 初始化配置

init_upstream 回调函数执行时需要初始化负载均衡模块的配置，还要设置一个新钩子，这个钩子函数会在 nginx 处理每个请求时作为初始化函数调用，关于这个新钩子函数的功能，后面会有详细的描述。这里，我们先分析 IP hash 模块初始化配置的代码：

```
ngx_http_upstream_init_round_robin(cf, us);
us->peer.init = ngx_http_upstream_init_ip_hash_peer;
```

这段代码非常简单：IP hash 模块首先调用另一个负载均衡模块 Round Robin 的初始化函数，然后再设置自己的处理请求阶段初始化钩子。实际上几个负载均衡模块可以组成一条链表，每次都是从链首的模块开始进行处理。如果模块决定不处理，可以将处理权交给链表中的下一个模块。这里，IP hash 模块指定 Round Robin 模块作为自己的后继负载均衡模块，所以在自己的初始化配置函数中也对 Round Robin 模块进行初始化。

5. 初始化请求

nginx 收到一个请求以后，如果发现需要访问 upstream，就会执行对应的 peer.init 函数。这是在初始化配置时设置的回调函数。这个函数最重要的作用是构造一张表，当前请求可以使用的 upstream 服务器被依次添加到这张表中。之所以需要这张表，最重要的原因是如果 upstream 服务器出现异常，不能提供服务时，可以从这张表中取得其他服务器进行重试操作。此外，这张表也可以用于负载均衡的计算。之所以构造这张表的行为放在这里而不是在前面初始化配置的阶段，是因为 upstream 需要为每一个请求提供独立隔离的环境。
为了讨论 peer.init 的核心，我们还是看 IP hash 模块的实现：

```
r->upstream->peer.data = &iphp->rrp;

ngx_http_upstream_init_round_robin_peer(r, us);

r->upstream->peer.get = ngx_http_upstream_get_ip_hash_peer;
```

第一行是设置数据指针，这个指针就是指向前面提到的那张表；
第二行是调用 Round Robin 模块的回调函数对该模块进行请求初始化。面前已经提到，一个负载均衡模块可以调用其他负载均衡模块以提供功能的补充。
第三行是设置一个新的回调函数 get。该函数负责从表中取出某个服务器。除了 get 回调函数，还有另一个 r->upstream->peer.free 的回调函数。该函数在 upstream 请求完成后调用，负责做一些善后工作。比如我们需要维护一个 upstream 服务器访问计数器，那么可以在 get 函数中对其加 1，在 free 中对其减 1。如果是 SSL 的话，nginx 还提供两个回调函数 peer.set_session 和 peer.save_session。一般来说，有两个切入点实现负载均衡算法，其一是在这里，其二是在 get 回调函数中。

6. peer.get 和 peer.free 回调函数

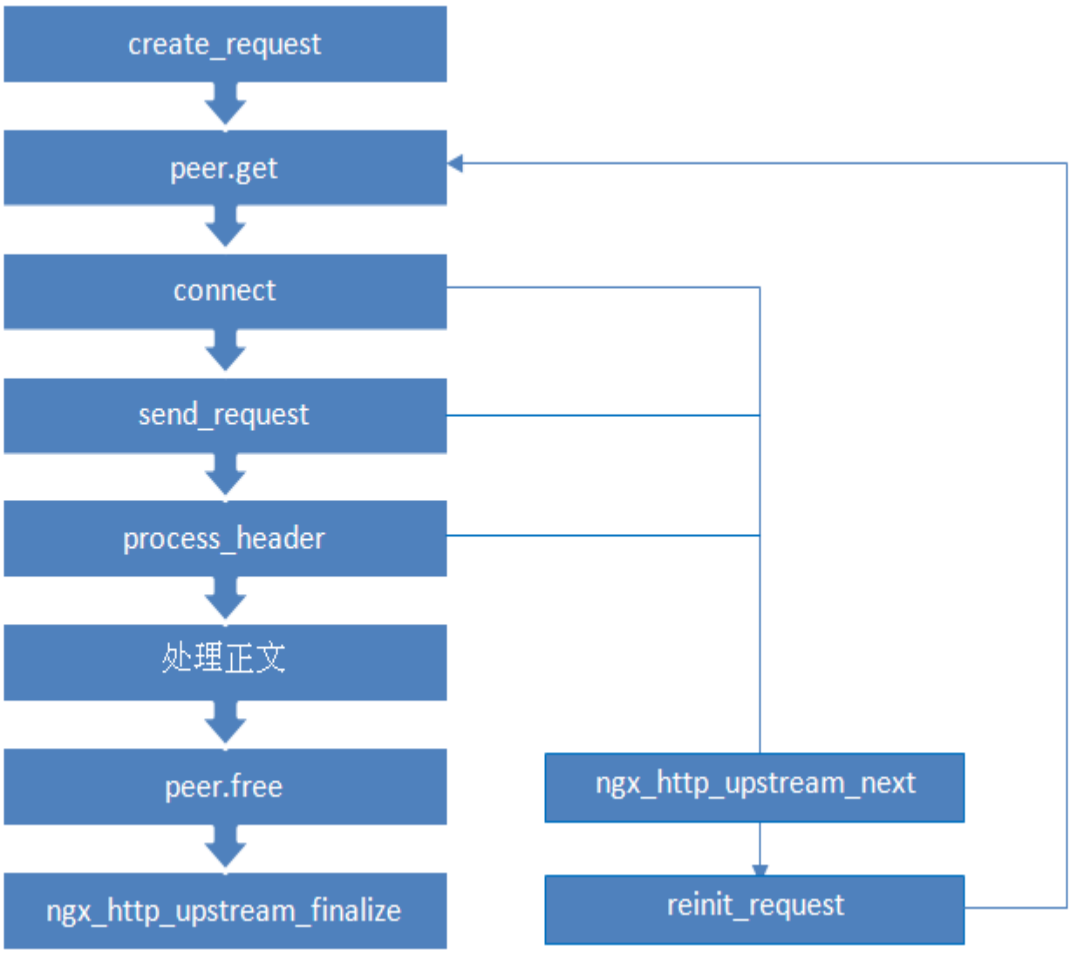
这两个函数是负载均衡模块最底层的函数，负责实际获取一个连接和回收一个连接的预备操作。之所以说是预备操作，是因为在这两个函数中，并不实际进行建立连接或者释放连接的动作，而只是执行获取连接的地址或维护连接状态的操作。需要理解的清楚一点，在 peer.get 函数中获取连接的地址信息，并不代表这时连接一定没有被建立，相反的，通过 get 函数的返回值，nginx 可以了解是否存在可用连接，连接是否已经建立。这些返回值总结如下：

返回值	说明	nginx 后续动作
NGX_DONE	得到了连接地址信息，并且连接已经建立。	直接使用连接，发送数据。
NGX_OK	得到了连接地址信息，但连接并未建立。	建立连接，如连接不能立即建立，设置事件， 暂停执行本请求，执行别的请求。
NGX_BUSY	所有连接均不可用。	返回 502 错误至客户端。

各位读者看到上面这张表，可能会有几个问题浮现出来：

Q:	什么时候连接是已经建立的？
A:	使用后端 keepalive 连接的时候，连接在使用完以后并不关闭，而是存放在一个队列中，新的请求只需要从队列中取出连接，这些连接都是已经准备好的。

Q:	什么叫所有连接均不可用？
A:	初始化请求的过程中，建立了一张表，get 函数负责每次从这张表中不重复的取出一个连接，当无法从表中取得一个新的连接时，即所有连接均不可用。
Q:	对于一个请求，peer.get 函数可能被调用多次么？
A:	正式如此。当某次 peer.get 函数得到的连接地址连接不上，或者请求对应的服务器得到异常响应，nginx 会执行 ngx_http_upstream_next，然后可能再次调用 peer.get 函数尝试别的连接。upstream 整体流程如下：



负载均衡模块的配置区集中在 upstream{}块中。负载均衡模块的回调函数体系是以 init_upstream 为起点，经历 init_peer，最终到达 peer.get 和 peer.free。其中 init_peer 负责建立每个请求使用的 server 列表，peer.get 负责从 server 列表中选择某个 server（一般是不重复选择），而 peer.free 负责 server 释放前的资源释放工作。最后，这一节通过一张图将 upstream 模块和负载均衡模块在请求处理过程中的相互关系展现出来。