Emiller 的 Nginx 模块开发指南

作者：Evan Miller 草稿：2009 年 8 月 13 日
译者：姚伟斌 草稿：2009 年 9 月 21 日

内容目录

翻译说明：

在 nginx 的模块编写过程中，时常苦于文档的不足，而源代码中又没多少注释。只有 Emiller 的这篇英文文档带我入门，在自己研读的过程中，就想将其翻译出来，让其他人能快速的浏览，但是如果你想更深入的进入 nginx 的代码开发，最好是多读 nginx 的代码。因其内部的代码是经常改变的，所以本文有可能已经过时。

由于本人英语水平一般，接触 Nginx 时间不多，翻译中碰到的错误在所难免，如果您觉得哪里翻译得不对，请跟我联系：yaoweibin2008@163.com

翻译词汇对照表：

Backend：后端服务器。

Buffer：缓冲区。

Callback：回调函数，一般来说是将某个回调函数赋值给某个函数指针

CHAIN OF RESPONSIBILITY：接力链表。

Context：上下文，有前后工作环境的意思，主要是前期配置值初始化。

Filter：过滤模块/函数，模块和函数的概念似乎有点模糊不清。

Handler: 处理模块/函数，另外也有指向具体的处理函数的指针或句柄的意思。

Installation：原意为安装，我觉得还是译作注册好点。

Load-balancer：负载均衡模块/函数。

Location：指目录位置，比如 http://img.blog.163.com/photo/中的“/photo”目录。

Master：主进程，由主进程产生 worker 进程，同时也可以而监视 worker 进程的动态，worker 因为异常而退出的时候也可以重启一个新的 worker 进程。

Reference：一般译作“引用”，不过很多时候，似乎还不如称作“指针”来的

直接些。

Request：HTTP 服务请求。

Response：HTTP 服务回复。

Server-side：服务端。

Upstream：上游服务（器），有时亦称 backend。

Worker：工作进程，模块真正发挥作用的地方。

0. 预备知识

对于 C 语言，你应该十分熟悉，对于结构体和预处理命令应有深入的理解，不至于见到大量的指针和函数引用就惊慌失措。如果觉得需要补习，就多看看 K&R（C 语言的语法书）。

如果你对于 HTTP 协议已经有了基本概念，那是很有好处的。毕竟你正在 Web 服务器上做开发。

你应该熟悉 Nginx 的配置文件。如果不熟悉，也没关系，这里有一些基本理解：Nginx 配置文件主要分成四部分：*main（全局设置）*、*server（主机设置）* 、*upstream（上游服务器设置）*和 *location（URL 匹配特定位置后的设置）*。每部分包含若干个指令。main 部分设置的指令将影响其它所有部分；server 部分的指令主要用于指定主机和端口；upstream 的指令用于设置后端服务器；location 部分用于匹配网页位置（比如，根目录“/”，“/images”等等） 。Location 部分会继承 server 部分的指令，而 server 部分的指令会继承 main 部分；upstream 既不继承指令也不会影响其他部分。它有自己的特殊指令，不需要在其他地方的应用。在下面很多地方都会涉及这四个部分，不要忘记哟。

让我们开始吧。

1. Nginx 模块任务委派的主要轮廓

Nginx 模块主要有 3 种角色：

- *handlers(处理模块)* 用于处理 HTTP 请求，然后产生输出
- *filters（过滤模块）* 过滤 handler 产生的输出
- *load-balancers（负载均衡模块）*当有多于一台的后端备选服务器时，选择一台转发 HTTP 请求

模块可以做任何你分配给 web 服务器的实际工作：当 Nginx 发送文件或者转发请求到其他服务器，有**处理模块**为其服务；当需要 Nginx 把输出压缩或者在服务端加一些东西，可以用**过滤模块**；还有一些 Nginx 的核心模块主要负责管理网络层和应用层协议，以及针对特定应用的一系列模块。Nginx 集中式的体系结构让你可以随心所欲地实现一些功能强大的内部单元。

注意：Nginx 不像 apache，模块不是动态添加的（换句话就是说，所有的模块都要预先编译进 Nginx 的二进制可执行文件）。

模块是如何被调用的？典型的讲，当服务器启动，每个处理模块都有机会映射到配置文件中定义的特定位置；如果有多个处理模块映射到同一个位置时，只

有一个会“赢”（聪明如你，当然不会让这些冲突产生）。处理模块主要以三种形式返回：正常、错误、或者放弃处理这个请求而让默认处理模块来处理。

如果处理模块把请求反向代理到后端的服务器，就变成另外一类的模块：**负载均衡模块**。负载均衡模块的配置中有一组后端服务器，当一个 HTTP 请求过来时，它决定哪台服务器应当获得这个请求。Nginx 的负载均衡模块默认采用两种方法：轮转法，它处理请求的方式就像纸牌游戏一样从头到尾分发；IP 哈希法，在众多请求的情况下，它确保来自同一个 IP 的请求会分发到相同的后端服务器。当然还有一些第三方的负载均衡方法，你可以从这里找到。

如果处理模块没有产生错误，过滤模块将被调用。过滤模块能映射到每个位置，而且过滤模块是有先后顺序的，它们的执行顺序在编译时决定。过滤模块是经典的“接力链表（CHAIN OF RESPONSIBILITY）”模型：一个过滤模块被调用，完成其工作，然后调用下一个过滤模块，直到最后一个过滤模块。一般来说压缩模块是比较靠后的，不然压缩以后的内容是很难用来读的。最后，Nginx 发出回复。

真正 cool 的地方是在过滤模块链中，每个过滤模块不会等上一个过滤模块全部完成：它能把前一个过滤模块的输出作为其处理内容；有点像 Unix 中的流水线。过滤模块能以 buffer（缓冲区）为单位进行操作，这些 buffer 一般都是一页（4K）大小，当然你也可以在 nginx.conf 文件中进行配置。这意味着，比如模块可以压缩来自后端服务器的回复，然后像流一样的到达客户端，直到整个回复发送完成。

所以总结下上面的内容，一个典型的处理周期是这样的：
客户端发送 HTTP 请求->Nginx 根据配置选择一个合适的处理模块->（如果有）负载均衡模块选择一台后端服务器，并负责完成后端的发送接收过程->处理模块进行处理并把输出缓冲放到第一个过滤模块上->第一个过滤模块处理后输出给第二个过滤模块->然后第二个过滤模块又到第三个->依此类推->最后把回复发给客户端。

我说“典型”这个词是因为 Nginx 的模块调用是具有很强的定制性的。模块开发者需要花很多精力精确定义模块在何时如何产生作用（我认为是件不容易的事）。模块的调用事实上通过一系列的回调函数来实现，很多很多。名义上来说，你的函数可在以下时段执行某些功能：

- 当服务读配置文件之前
- 读存在 location 和 server 或其他任何部分的每一个配置指令
- 当 Nginx 初始化全局部分的配置时
- 当 Nginx 初始化主机部分（比如主机/端口）的配置时
- 当 Nginx 将全局部分的配置与主机部分的配置合并的时候
- 当 Nginx 初始化位置部分配置的时候
- 当 Nginx 将其上层主机配置与位置部分配置合并的时候
- 当 Nginx 的主（master）进程开始的时候
- 当一个新的工作进程（worker）开始的时候

- 当一个工作进程退出的时候
- 当主进程退出的时候
- 处理请求
- 过滤回复的头部
- 过滤回复的主体
- 选择一台后端服务器
- 初始化到后端服务器的请求
- 重新初始化到后端的服务器的请求
- 处理来自后端服务器的回复
- 完成与后端服务器的交互

难以置信，有点应接不暇！有这么多的功能任你处置，而你只需通过多组有用的钩子（由函数指针组成的结构体）和相应的实现函数。现在让我们开始接触一些模块吧。

2. Nginx 模块的组成

我说过， Nginx 模块的使用是很灵活的。本段描述的东西会经常出现。它引导你理解模块，也可以成为你开始写模块的参考。

2.1. 模块的配置结构体

模块的配置结构体的定义有三种，分别是全局，主机和位置的配置结构体。大多数 HTTP 模块仅仅需要一个位置的配置结构体。名称约定是这样的：

`ngx_http_<module name>_(main|srv|loc)_conf_t`。这里有一个来自 `dav` 模块的例子：

```
typedef struct {
    ngx_uint_t  methods;
    ngx_flag_t  create_full_put_path;
    ngx_uint_t  access;
} ngx_http_dav_loc_conf_t;
```

注意 Nginx 有一些特别的类型（如：`ngx_uint_t` 和 `ngx_flag_t`），可能是一些基本类型的别名。（如果你好奇的话，可以看这里：core/ngx_config.h）这些类型用在配置结构体的情形很多。

2.2. 模块的指令

模块的指令出现在静态数组 ngx_command_t。这里有一个例子，说明它们是如何被定义的，来自我写的一个小模块：

```
static ngx_command_t  ngx_http_circle_gif_commands[] = {
    { ngx_string("circle_gif"),
      NGX_HTTP_LOC_CONF|NGX_CONF_NOARGS,
      ngx_http_circle_gif,
      NGX_HTTP_LOC_CONF_OFFSET,
      0,
      NULL },

    { ngx_string("circle_gif_min_radius"),
      NGX_HTTP_MAIN_CONF|NGX_HTTP_SRV_CONF|NGX_HTTP_LOC_CONF|
NGX_CONF_TAKE1,
      ngx_conf_set_num_slot,
      NGX_HTTP_LOC_CONF_OFFSET,
      offsetof(ngx_http_circle_gif_loc_conf_t, min_radius),
      NULL },
      ...
      ngx_null_command
};
```

这是 ngx_command_t 的函数原型（也就是我们定义的那些结构体），出自 core/ngx_conf_file.h：

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

参数多了点，但每个参数都是有用的：

name 是指令的字符串（也就是包含指令名称），不包含空格（有空格的话就是命令的参数了）。数据类型是 ngx_str_t，经常用来进行字符串实例化（比如：ngx_str("proxy_pass")）。注意：ngx_str_t 结构体由包含有字符串的 data 成员和表示字符串长度的 len 成员组成。Nginx 用这个数据类型来存放字符串。

type 是标识的集合，表明这个指令在哪里出现是合法的、指令的参数个数。应用

中，标识一般是下面多个值的位或：

- NGX_HTTP_MAIN_CONF: 指令出现在全局配置部分是合法的
- NGX_HTTP_SRV_CONF: 指令在主机配置部分出现是合法的
- NGX_HTTP_LOC_CONF: 指令在位置配置部分出现是合法的
- NGX_HTTP_UPS_CONF: 指令在上游服务器配置部分出现是合法的

- NGX_CONF_NOARGS: 指令没有参数
- NGX_CONF_TAKE1: 指令读入一个参数
- NGX_CONF_TAKE2: 指令读入两个参数
- ...
- NGX_CONF_TAKE7: 指令读入七个参数

- NGX_CONF_FLAG: 指令读入一个布尔型数据
- NGX_CONF_1MORE: 指令至少读入 1 个参数
- NGX_CONF_2MORE: 指令至少读入 2 个参数

这里有很多另外的选项：core/ngx_conf_file.h。

结构体成员 set 是一个函数指针，用来设定模块的配置；典型地，这个函数会转化读入指令传进来的参数，然后将合适的值保存到配置结构体。这个设定函数有三个参数：

1. 指向 ngx_conf_t 结构体的指针，包含从配置文件中指令传过来的参数。
2. 指向当前 ngx_command_t 结构体的指针
3. 指向自定义模块配置结构体的指针

这个设定函数在指令被遇到的时候就会调用。在自定义的配置结构体中，Nginx 提供了多个函数用来保存特定类型的数据，这些函数包含有：

- ngx_conf_set_flag_slot：将"on"或"off"转化为
- ngx_conf_set_str_slot：将字符串保存为ngx_str_t类型
- ngx_conf_set_num_slot：解析一个数字并保存为int型
- ngx_conf_set_size_slot：解析一个数据大小(如："8k"，"1m"）并保存为 size_t类型

还有另外一些，很容易查到（看，core/ngx_conf_file.h）。模块也可以把它们自己函数的引用放在这里，但这样内嵌的类型不是很好。

那这些内嵌函数怎么知道要把值保存在哪里呢？ngx_command_t接下来的两个成员conf和offset正好可用。conf告诉Nginx把这个值是放在全局配置部分、主机配置部分还是位置配置部分（用NGX_HTTP_MAIN_CONF_OFFSET，NGX_HTTP_SRV_CONF_OFFSET或NGX_HTTP_LOC_CONF_OFFSET）。然后offset确定到底是保存在结构体的哪个位置。

最后，post指向模块在读配置的时候需要的一些零碎变量。一般它是NULL。

这个ngx_command_t 数组在读入ngx_null_command后停止。

2.3. 模块的上下文

静态的ngx_http_module_t结构体，包含一大把函数引用。用来创建三个部分的配置和合并配置。一般结构体命名为ngx_http_<module name>_module_ctx。以此，这些函数引用包括：

- 在读入配置文件前调用
- 在读入配置文件后调用
- 在创建全局部分配置时调用（比如，用来分配空间和设置默认值）
- 在初始化全局部分的配置时调用（比如，把原来的默认值用nginx.conf读到的值来覆盖）
- 在创建主机部分的配置时调用
- 与全局部分配置合并时调用
- 创建位置部分的配置时掉用
- 与主机部分配置合并时调用

这些函数参数不同，依赖于它们的功能。这里有这个结构体的定义，摘自http/ngx_http_config.h，你可以看到属性各不同的回调函数：

```
typedef struct {
```

```
    ngx_int_t   (*preconfiguration)(ngx_conf_t *cf);
    ngx_int_t   (*postconfiguration)(ngx_conf_t *cf);

    void       *(*create_main_conf)(ngx_conf_t *cf);
    char       *(*init_main_conf)(ngx_conf_t *cf, void *conf);

    void       *(*create_srv_conf)(ngx_conf_t *cf);
    char       *(*merge_srv_conf)(ngx_conf_t *cf, void *prev, void
*conf);

    void       *(*create_loc_conf)(ngx_conf_t *cf);
    char       *(*merge_loc_conf)(ngx_conf_t *cf, void *prev, void
*conf);
} ngx_http_module_t;
```

如果某些函数不用，可以设定为NULL，Nginx会剔除它。

大多数处理模块只使用最后两个：一个函数用来为特定的位置部分的配置结构体分配内存（称为ngx_http_<module name>_create_loc_conf），另外一个函数用来设定默认值和与继承过来的配置合并(称为ngx_http_<module name>_merge_loc_conf)。这个合并函数负责检验读入的数值是否有效，并设定一些默认值。

这里有一个模块上下文结构体的例子：

```
static ngx_http_module_t  ngx_http_circle_gif_module_ctx = {
    NULL,                          /* preconfiguration */
    NULL,                          /* postconfiguration */

    NULL,                          /* create main configuration */
    NULL,                          /* init main configuration */

    NULL,                          /* create server configuration */
    NULL,                          /* merge server configuration */

    ngx_http_circle_gif_create_loc_conf,  /* create location
configuration */
    ngx_http_circle_gif_merge_loc_conf /* merge location
configuration */

};
```

现在开始讲得更深一点。这些配置回调函数看其来很相似，几乎所有模块都一样，而且Nginx的API都会用到这个部分，所以值得了解。

2.3.1. 创建位置结构体（create_loc_conf）

create_loc_conf 函数骨架看起来像这个样子，摘自我写的 circle_gif 模块（看源代码）。它的参数是结构体 ngx_conf_t ，返回更新的模块配置结构体（例子中是 ngx_http_circle_gif_loc_conf_t）。

```
static void *
ngx_http_circle_gif_create_loc_conf(ngx_conf_t *cf)
{
    ngx_http_circle_gif_loc_conf_t  *conf;

    conf = ngx_pcalloc(cf->pool,
sizeof(ngx_http_circle_gif_loc_conf_t));
    if (conf == NULL) {
        return NGX_CONF_ERROR;
    }
    conf->min_radius = NGX_CONF_UNSET_UINT;
    conf->max_radius = NGX_CONF_UNSET_UINT;
    return conf;
}
```

第一个需要注意的是 Nginx 的内存分配，只要使用 ngx_palloc（malloc 的包装函数）或 ngx_pcalloc（calloc 的包装函数）就不需要关心内存的释放。UNSET 常量有：NGX_CONF_UNSET_UINT, NGX_CONF_UNSET_PTR, NGX_CONF_UNSET_SIZE, NGX_CONF_UNSET_MSEC, 和一起的 NGX_CONF_UNSET。UNSET 告诉合并函数这些值还没有被初始化过，需要设定其参数。

2.3.2. 初始化结构体（merge_loc_conf）

这是用在 circle_gif 模块的合并函数：

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

    ngx_conf_merge_uint_value(conf->min_radius, prev->min_radius,
10);
    ngx_conf_merge_uint_value(conf->max_radius, prev->max_radius,
20);
```

```
    if (conf->min_radius < 1) {
        ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
            "min_radius must be equal or more than 1");
        return NGX_CONF_ERROR;
    }
    if (conf->max_radius < conf->min_radius) {
        ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
            "max_radius must be equal or more than min_radius");
        return NGX_CONF_ERROR;
    }

    return NGX_CONF_OK;
}
```

首先需要注意的是Nginx为不同的类型提供了良好的合并函数（ngx_conf_merge_<data type>_value）；这些参数含义是：

1、当前参数变量
2、如果第一个参数没有被设置
3、如果第一个和第二个参数都没有设置时的默认值

结果会保存在第一个参数中。有效的合并函数包括ngx_conf_merge_size_value, ngx_conf_merge_msec_value，在core/ngx_conf_file.h里有完整的列出。

有个问题：因为第一个参数是传值的，这些值是如何写到第一个参数中？

回答：这些函数其实都是预处理命令（在真正编译之前，它们会被扩展成一些if语句）。

注意错误是如何产生的；这些函数把错误信息写到log文件中，然后返回NGX_CONF_ERROR。返回代码会中止服务的启动。（因为被标识为NGX_LOG_EMERG，这些消息也会被会输出到标准输出；core/ngx_log.h有完整的日志级别。）

2.4. 模块定义

第二步，我们间接的介绍更多一层，结构体ngx_module_t。这个结构体变量命名为ngx_http_<module name>_module。它包含有模块的主要内容和指令的执行部分，也有一些回调函数（退出线程，退出进程，等等）。这些函数的定义是把数据处理关联到特定模块的关键。模块定义像这个样子：

```
ngx_module_t  ngx_http_<module name>_module = {
    NGX_MODULE_V1,
    &ngx_http_<module name>_module_ctx, /* module context */
    ngx_http_<module name>_commands,    /* module directives */
    NGX_HTTP_MODULE,                /* module type */
    NULL,                           /* init master */
    NULL,                           /* init module */
    NULL,                           /* init process */
    NULL,                           /* init thread */
    NULL,                           /* exit thread */
    NULL,                           /* exit process */
    NULL,                           /* exit master */
    NGX_MODULE_V1_PADDING
};
```

…仅仅替换掉合适的模块名称就可以了。在进程/线程退出的时候，模块可以添加一些回调函数来运行，但大多数模块用不到。（这些回调函数的参数，可以参考 core/ngx_conf_file.h .）

2.5. 模块注册

有两种途径来注册（Installation）模块：处理模块经常是通过指令的回调函数来注册，过滤模块通过模块上下文结构体中的 postconfigration 回调函数来注册。具体你可以下面具体模块中会提到。

3. 处理模块、过滤模块和 负载均衡模块

接下来我们把模块的细节放到显微镜下，看它们到底怎么运行的。

3.1. 剖析处理模块(非代理)

处理模块一般做四样东西：获得位置配置结构体，产生合适的回复，发送 HTTP 头部和发送 HTTP 主体。它只有一个变量--请求结构体。这个结构体有很多关于客户端请求的有用信息，比如请求方法（request method），URI 和请求头部。我们会一步一步分析整个过程。

3.1.1. 获得位置配置结构体

这部分很简单。所有你需要做的是根据当前的请求结构体和模块定义，调用

ngx_http_get_module_loc_conf，获得当前的配置结构体。这是我写的 circle gif hanlder 函数的相关部分。

```
static ngx_int_t
ngx_http_circle_gif_handler(ngx_http_request_t *r)
{
    ngx_http_circle_gif_loc_conf_t  *circle_gif_config;
    circle_gif_config = ngx_http_get_module_loc_conf(r,
ngx_http_circle_gif_module);
    ...
```

现在我们就能访问在合并函数中初始化的各个变量。

3.1.2. 产生回复

这部分很有趣，也是模块真正起作用的地方。

请求结构体在这里也很有用，特别是这些成员变量：

```
typedef struct {
...
/* the memory pool, used in the ngx_palloc functions */
    ngx_pool_t                        *pool;
    ngx_str_t                          uri;
    ngx_str_t                          args;
    ngx_http_headers_in_t              headers_in;

...
} ngx_http_request_t;
```

uri 是请求的路径，比如："/query.cgi"。

args 是在问号之后请求的参数(比如 "name=john")。

headers_in 有很多有用的东西，如 cookie 和浏览器信息。如果你感兴趣，请看这里 http/ngx_http_request.h 。

这里应该有足够的信息来产生有用的输出。ngx_http_request_t 结构体的完整定义在 http/ngx_http_request.h。

3.1.3. 发送 HTTP 头部

回复头部存在于被称为 headers_out 的结构体中。它包含在请求结构体中。这个处理函数生成头部变量，然后调用 ngx_http_send_header(r)函数，下面列出些有用的部分：

```
typedef stuct {
...
    ngx_uint_t                        status;
    size_t                            content_type_len;
    ngx_str_t                         content_type;
    ngx_table_elt_t                  *content_encoding;
    off_t                             content_length_n;
    time_t                            date_time;
    time_t                            last_modified_time;
..
} ngx_http_headers_out_t;
```

(另外一些你可以在这里找到： http/ngx_http_request.h.)

举个例子，如果一个模块把 Content-Type 需要设定为“image/gif”，Content-Length 为 100，然后返回 200 OK 的回复，代码将是这样的：

```
    r->headers_out.status = NGX_HTTP_OK;
    r->headers_out.content_length_n = 100;
    r->headers_out.content_type.len = sizeof("image/gif") - 1;
    r->headers_out.content_type.data = (u_char *) "image/gif";
    ngx_http_send_header(r);
```

上面的设定方式针对大多数参数都是有效的。但一些头部的变量设定要比上面的例子要麻烦；比如，content_encoding 含有类型(ngx_table_elt_t*)，这时模块必须为它分配内存。可以用一个叫 ngx_list_push 的函数来做。它需要传入一个 ngx_list_t 变量（与数组类似），然后返回一个 list 中的新成员（类型是 ngx_table_elt_t）。下面的代码把 Content-Encoding 设定为“deflate”，然后把头部发出。

```
    r->headers_out.content_encoding = ngx_list_push(&r-
>headers_out.headers);
    if (r->headers_out.content_encoding == NULL) {
        return NGX_ERROR;
    }
```

```
    r->headers_out.content_encoding->hash = 1;
    r->headers_out.content_encoding->key.len = sizeof("Content-
Encoding") - 1;
    r->headers_out.content_encoding->key.data = (u_char *) "Content-
Encoding";
    r->headers_out.content_encoding->value.len = sizeof("deflate") -
1;
    r->headers_out.content_encoding->value.data = (u_char *)
"deflate";
    ngx_http_send_header(r);
```

当头部有多个值的时候，这个机制会经常用到；它（理论上讲）使得过滤模块添加、删除某个值而保留其他值的时候更加容易，在操纵字符串的时候，不需要把字符串重新排序。

3.1.4. 发送 HTTP 主体

现在模块已经产生了一个回复，把它放到内存中。需要为回复分配一块特别的 buffer，并把这个 buffer 连接到一个链表，然后调用“send body”函数发送。

这些链表有什么用？在 Nginx 中，处理模块和过滤模块在处理完成后产生的回复都包含在缓冲中，每次产生一个 buffer；每个链表成员保存指向下一个成员的指针，如果是最后的 buffer，就置为 NULL。这里我们简单地假定只有一个 buffer 成员。

首先，模块声明一块 buffer 和一条链表：

```
    ngx_buf_t    *b;
    ngx_chain_t   out;
```

第二步是分配缓冲，然后指向我们的回复数据：

```
    b = ngx_pcalloc(r->pool, sizeof(ngx_buf_t));
    if (b == NULL) {
        ngx_log_error(NGX_LOG_ERR, r->connection->log, 0,
            "Failed to allocate response buffer.");
        return NGX_HTTP_INTERNAL_SERVER_ERROR;
    }
```

```
    b->pos = some_bytes; /* first position in memory of the data */
    b->last = some_bytes + some_bytes_length; /* last position */
```

```
    b->memory = 1; /* content is in read-only memory */
    /* (i.e., filters should copy it rather than rewrite in place) */

    b->last_buf = 1; /* there will be no more buffers in the request
*/
```

现在模块 buffer 添加到了链表上：

```
    out.buf = b;
    out.next = NULL;
```

最后，我们把主体发送出去，返回值是 output_filter 函数对整个链表的返回状态。

```
    return ngx_http_output_filter(r, &out);
```

缓冲链表是一个典型的 Nginx IO 模型，你必须清楚它们是如何工作的。

问题：为什么会有变量 last_buf，什么时候我们才能说这条链表结束了，并把 next 设为 NULL？

回答：链表可能不完全的，比如，有多个 buffer 的时候，但是不是所有的 buffer 都在这个请求或者回复中。所以一些 buffer 是链表的结尾，而不是请求的结尾。这意味着模块判断是否是请求的结尾，并设置相应的值。

3.2. 上游模块剖析（又称代理模块）

我已经帮你了解了如何用处理模块来产生回复。你可能用一小块 C 代码就可以完成，但有时你想和另外一台服务器通信（比如，你写一个模块来实现另一种协议）。你可能需要自己做所有的网络编程。如果收到部分的回复，需要等待余下的回复数据，你会怎么办？你不会想阻塞整个事件处理循环吧？这样会毁掉 Nginx 的良好性能！幸运的是，Nginx 可以使用某种机制将回复映射到一台后端服务器（叫做“upstream”），你的模块可以与另外的服务器通信而不需要任何请求。这个小节将会描述模块是如何与上游服务器通信，比如 Memcached, FastCGI 或其它 HTTP 服务器。

3.2.1.代理模块回调函数的概要

与其他模块的处理函数不一样，代理模块的处理函数仅作少量的实际工作。它也不会调用 ngx_http_output_filter，仅仅设定一些回调函数。这些函数在代理服务等待接受和发送的时候被调用。总共有 6 个：

- create_request 生成发送到上游服务器的请求（或者一条缓冲链）
- reinit_request 在后端服务器被重置的情况下（在create_request被第二次调用之前） 被调用
- process_header 处理上游服务器的回复
- abort_request 在客户端放弃请求的时候被调用
- finalize_request 在Nginx完成从上游服务器读入回复并解析完成以后被调用
- input_filter 这是一个主体过滤函数，在产生回复主体时调用(比如把尾部删掉)

这些函数是怎么附加上去的？下面是一个例子，简单版本的代理模块处理函数：

```
static ngx_int_t
ngx_http_proxy_handler(ngx_http_request_t *r)
{
    ngx_int_t                    rc;
    ngx_http_upstream_t         *u;
    ngx_http_proxy_loc_conf_t  *plcf;

    plcf = ngx_http_get_module_loc_conf(r, ngx_http_proxy_module);

/* set up our upstream struct */
    u = ngx_pcalloc(r->pool, sizeof(ngx_http_upstream_t));
    if (u == NULL) {
        return NGX_HTTP_INTERNAL_SERVER_ERROR;
    }

    u->peer.log = r->connection->log;
    u->peer.log_error = NGX_ERROR_ERR;

    u->output.tag = (ngx_buf_tag_t) &ngx_http_proxy_module;

    u->conf = &plcf->upstream;

/* attach the callback functions */
    u->create_request = ngx_http_proxy_create_request;
    u->reinit_request = ngx_http_proxy_reinit_request;
    u->process_header = ngx_http_proxy_process_status_line;
    u->abort_request = ngx_http_proxy_abort_request;
    u->finalize_request = ngx_http_proxy_finalize_request;

    r->upstream = u;

    rc = ngx_http_read_client_request_body(r,
```

```
ngx_http_upstream_init);

    if (rc >= NGX_HTTP_SPECIAL_RESPONSE) {
        return rc;
    }

    return NGX_DONE;
}
```

这个函数只是些打扫工作，重要的部分在回调函数，另外对于 `ngx_http_read_client_request_body` 也要注意。当结束从客户端读以后，这个函数用来设定另外一个回调函数。

每个回调函数都是怎么做的？一般来说 `reinit_request`, `abort_request`, 和 `finalize_request` 会设定或者重新设定一些内部状态，可能只有几行代码。真正的工作是在 `create_request` 和 `process_header` 完成。

3.2.2. create_request 回调函数

为了简单起见，我们假定有个上游服务器读入一个字符，然后打印出两个字符。我们的函数会怎么写呢？

`create_request` 需要为每个字符分配缓冲区，然后为缓冲分配一条链表，最后把链表挂到 upstream 结构体。看起来像这样：

```
static ngx_int_t
ngx_http_character_server_create_request(ngx_http_request_t *r)
{
/* make a buffer and chain */
    ngx_buf_t *b;
    ngx_chain_t *cl;

    b = ngx_create_temp_buf(r->pool, sizeof("a") - 1);
    if (b == NULL)
        return NGX_ERROR;

    cl = ngx_alloc_chain_link(r->pool);
    if (cl == NULL)
        return NGX_ERROR;

/* hook the buffer to the chain */
    cl->buf = b;
/* chain to the upstream */
```

```
    r->upstream->request_bufs = cl;

/* now write to the buffer */
    b->pos = "a";
    b->last = b->pos + sizeof("a") - 1;

    return NGX_OK;
}
```

看起来不错，是吗？当然，在现实中，你可能需要在很多方面需要用到URI。r->uri是一个ngx_str_t类型，GET的参数在r->args里面，不要忘记你还可以访问请求结构体的头部和cookie。

3.2.3. process_header回调函数

现在讲process_header，就像create_request把链表指针挂到请求结构体上去一样，process_header把回复指针移到客户端可以接收到的部分。

这里有一个小例子，读进两个字符的回复。让我们假定第一个字符是“状态”字符。如果是问号，我们可以返回404 File Not Found给客户端，并把余下的字符抛弃掉。如果是空格，我们可以返回给客户端一个200 OK。好吧，这不是很有用的协议，但是用来说明原理还是可以的。那我们怎么来写这个process_header函数呢？

```
static ngx_int_t
ngx_http_character_server_process_header(ngx_http_request_t *r)
{
    ngx_http_upstream_t       *u;
    u = r->upstream;

    /* read the first character */
    switch(u->buffer.pos[0]) {
        case '?':
            r->header_only = 1; /* suppress this buffer from the
client */
            u->headers_in.status_n = 404;
            break;
        case ' ':
            u->buffer.pos++; /* move the buffer to point to the next
character */
            u->headers_in.status_n = 200;
            break;
    }
```

```
    return NGX_OK;
}
```

就是这样，操作头部，改变指针，完成。注意 `headers_in` 是回复的头部结构体，就像我们以前看到过的一样（http/ngx_http_request.h），在代理模块中，操作这些头部是很流行的。真正的代理模块会对头部做很多事情，不仅仅是错误处理，这就看你的想法了。

但是，如果我们从上游服务器获得的回复头部不在一个缓冲区中，那怎么办?

3.2.4. 状态保持

记得我曾经说过，`abort_request`, `reinit_request`, 和 `finalize_request` 可以用来重设内部状态么？那是因为许多代理模块有内部状态。这个状态需要自定义一个**上下文结构体**，以便用来标记到现在为止我们到底从上游服务器读到了什么。这个跟前面提到的“模块上下文”是不一样的。模块上下文只是些已经确定了的类型，而这里的自定义上下文却可以包含任何你想要的成员或者数据（这是你的结构体）。这个上下文结构体应该在 `create_request` 函数里面实例化，可能像这样：

```
    ngx_http_character_server_ctx_t   *p;   /* my custom context
struct */

    p = ngx_pcalloc(r->pool,
sizeof(ngx_http_character_server_ctx_t));
    if (p == NULL) {
        return NGX_HTTP_INTERNAL_SERVER_ERROR;
    }

    ngx_http_set_ctx(r, p, ngx_http_character_server_module);
```

最后一行实际上把这个自定义上下文结构体注册到了这个特定的请求和模块名称上去，以便以后拿来用。当你需要这个结构体的时候（也可能在其它任何回调函数中），这样做：

```
    ngx_http_proxy_ctx_t  *p;
    p = ngx_http_get_module_ctx(r, ngx_http_proxy_module);
```

指针 `p` 就可以得到当前的状态。给它赋值，重设它，把它增加，把它减少，所有数据放在那里，由你操纵。当从上游服务器不断返回一块块的数据的时候，这里刚好可以用支持状态机，而不用阻塞基本的事件处理循环。很好，很强大!

我已经把我知道的关于处理模块的东西都说了。是时候移驾过滤模块了，讲讲输出过滤链表的成员。头部过滤函数可以修改 HTTP 头，主体过滤函数可以修改回复内容。

3.3.处理模块的注册

处理模块的注册通过添加代码到指令中的回调函数。比如，我的 circle gif `ngx_command_t` 结构体看起来这样的：

```
    { ngx_string("circle_gif"),
      NGX_HTTP_LOC_CONF|NGX_CONF_NOARGS,
      ngx_http_circle_gif,
      0,
      0,
      NULL }
```

回调函数是第三个成员，在这个例子中是 `ngx_http_circle_gif` 函数。须知这个回调函数参数分别是配置结构体（`ngx_conf_t`，保存用户的参数），相关的 `ngx_command_t` 结构体，以及一个指向模块自定义的配置结构体。在我的 circle gif 模块，这个函数看来像这样：

```
static char *
ngx_http_circle_gif(ngx_conf_t *cf, ngx_command_t *cmd, void *conf)
{
    ngx_http_core_loc_conf_t  *clcf;

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

    return NGX_CONF_OK;
}
```

函数中有两步，获得这个位置配置的“核心”结构体，然后为它分配一个处理函数。是不是很简单，呵呵。

4 过滤模块

4.1. 剖析头部过滤函数

头部过滤函数由下面三个基础部分组成：

1. 决定是否操纵这个回复

2. 操纵这个回复

3. 调用下一个过滤函数

举个例子，这里有一个简单版本的“没有修改过的”头部过滤函数。如果客户端的 If-Modfied-Since 头部与回复的 Last-Modified 头部匹配，就把状态设为 304 Not Modified。头部过滤函数只有一个参数 ngx_http_request_t，但它可以让我们访问到客户端的头部和不久要发送的回复头部。

```
static
ngx_int_t ngx_http_not_modified_header_filter(ngx_http_request_t *r)
{
    time_t  if_modified_since;

    if_modified_since = ngx_http_parse_time(
r->headers_in.if_modified_since->value.data,
r->headers_in.if_modified_since->value.len);

/* step 1: decide whether to operate */
    if (if_modified_since != NGX_ERROR &&
        if_modified_since == r->headers_out.last_modified_time) {

/* step 2: operate on the header */
        r->headers_out.status = NGX_HTTP_NOT_MODIFIED;
        r->headers_out.content_type.len = 0;
        ngx_http_clear_content_length(r);
        ngx_http_clear_accept_ranges(r);
    }

/* step 3: call the next filter */
    return ngx_http_next_header_filter(r);
}
```

headers_out 结构体与处理模块中的一样（http/ngx_http_request.h），也可以随便改。

4.2. 剖析主体过滤函数

因为有了缓冲链表，主体过滤函数可能有点棘手，该函数每次只能操作一个缓

冲区（链表成员）。模块必须决定是否覆盖输入缓冲区，或者把缓冲去替换掉并分配一个新的缓冲，还是在原来的缓冲区后面插入一个新缓冲，这是个问题。复杂的时候，模块会接收到几个不完全的缓冲区，还需要继续操纵。不幸的是，Nginx 不提供高层次的 API 来操作缓冲区链表，所以主体过滤函数可能很难懂（也很难写）。但是，这里有些你可以看到在实际中用到的操作。

一个主体 filter 的函数原型看起来这样：（源代码摘自 Nginx 的"chunked" 过滤模块）

```
static ngx_int_t ngx_http_chunked_body_filter(ngx_http_request_t *r,
ngx_chain_t *in);
```

第一个参数是我们的老朋友请求结构体。第二个参数是指向当前链表头部的指针（可能有 0、1 或者多个缓冲）。

举个简单例子。假设我们像在每个请求的最后插入文本“<!-- Served by Nginx -->”。首先，我们需要确定回复的最后一个缓冲区是否包含在我们的缓冲区链表中。就像我说的那样，现在还没有一个完美的 API，所以我们必须自己做循环：

```
    ngx_chain_t *chain_link;
    int chain_contains_last_buffer = 0;

    for ( chain_link = in; chain_link->next != NULL; chain_link =
chain_link->next ) {
        if (chain_link->buf->last_buf)
            chain_contains_last_buffer = 1;
    }
```

如果没有找到最后的缓冲区，就返回：

```
    if (!chain_contains_last_buffer)
        return ngx_http_next_body_filter(r, in);
```

很好，现在最后一个缓冲区已经存在链表中了。我们分配一个新缓冲区：

```
    ngx_buf_t    *b;
    b = ngx_calloc_buf(r->pool);
    if (b == NULL) {
        return NGX_ERROR;
    }
```

然后把数据放进去：

```
    b->pos = (u_char *) "<!-- Served by Nginx -->";
    b->last = b->pos + sizeof("<!-- Served by Nginx -->") - 1;
```

把这个缓冲区挂到新的链表成员：

```
    ngx_chain_t   added_link;

    added_link.buf = b;
    added_link.next = NULL;
```

最后，把这个新的链表成员挂到先前链表的末尾：

```
    chain_link->next = added_link;
```

然后根据变化重设 last_buf 变量：

```
    chain_link->buf->last_buf = 0;
    added_link->buf->last_buf = 1;
```

返回修改过的链表，进入下一个输出过滤函数：

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

其实函数可以比我们做的多得多，比如，mod_perl ($response->body =~ s/$/<!-- Served by mod_perl -->/)，缓冲区链表功能强大，可以让程序员在以此处理数据，以便客户端可以尽快获得数据。不管怎样，按照我的观点，Nginx 实在应该有个干净的接口，程序员也可以摆脱链表中不一致的状态。到现在为止，我们操纵它是有风险的。

4.3 过滤函数的注册

过滤模块通过在读入配置后调用的函数（postconfigration）来注册。它们一般包括两种过滤函数：*头部过滤函（header filters）*处理 HTTP 头，*主体过滤函数（body filters）*处理主体。它们在同一个位置注册。

以 chunked filter 模块为例，它的模块上下文是这样的：

```
static ngx_http_module_t  ngx_http_chunked_filter_module_ctx = {
    NULL,                                  /* preconfiguration */
    ngx_http_chunked_filter_init,          /* postconfiguration */
```

```
    ...
};
```

下面是 ngx_http_chunked_filter_init 函数做的一些事情：

```
static ngx_int_t
ngx_http_chunked_filter_init(ngx_conf_t *cf)
{
    ngx_http_next_header_filter = ngx_http_top_header_filter;
    ngx_http_top_header_filter = ngx_http_chunked_header_filter;

    ngx_http_next_body_filter = ngx_http_top_body_filter;
    ngx_http_top_body_filter = ngx_http_chunked_body_filter;

    return NGX_OK;
}
```

这里到底做了些什么？如果你还记得，过滤模块组成了一条“接力链表”。当一个处理模块产生一个回复，它会调用两个函数：ngx_http_output_filter，调用全局函数 ngx_http_top_body_filter；ngx_http_send_header，调用全局函数 ngx_top_header_filter。

ngx_http_top_body_filter 和 ngx_http_top_header_filter 是主体和头部各自的“链表头部”。在这个链上每个成员都有指向下一个成员的函数指针（这个指针称为 ngx_http_next_body_filter 和 ngx_http_next_header_filter）。当一个过滤模块执行完成后，它就调用下一个过滤模块，直到一个特殊的“写”过滤模块被调用，它把整个 HTTP 包裹起来。在 filter_init 函数中，可以看到是把模块本身添加到过滤模块链中；它用 next 变量保存了一个指向旧的“top”过滤模块，然后声明自己是新的“top”过滤模块。（所以，最后注册进过滤链的模块是最先被执行的。）

作者边注：这是如何正常工作的呢？

原来每个过滤函数用下面的语句或者错误代码返回：

```
return ngx_http_next_body_filter();
```

（注意：ngx_http_next_header_filter 是局部全局变量，所以每个 filter 模块中该变量的值都不一样）所以，如果过滤链表达到最后一个成员（特别定义的），就会返回一个 OK。如果出现错误，整条链表会被缩短，Nginx 产生合适的错误消息。这是一条单向链表，仅仅使用函数引用，就可以实现快速的错误返回。聪明。

5. 剖析负载均衡模块

负载均衡模块决定哪个后端服务器可以分到特定的请求；现有的实现有通过轮转法或者对请求的某些部分进行哈希处理。这一节将会介绍负载均衡模块的注册和调用，并以 upstream_hash 模块为例子（源代码）。Upstream_hash 模块对在 nginx.conf 中确定的某个变量进行哈希。

一个负载均衡模块有以下六个方面：

1. 激活配置指令需要调用一个*注册函数 registration function*（比如：`hash`）
2. 注册函数定义合法的服务器选项（比如：`weight=`），同时注册*上游主机初始化函数 upstream initialization function*
3. 上游主机初始化函数在配置确认好以后被调用：
   a) 解析服务器名称，指向特定的 IP 地址
   b) 为每个 socket 分配空间
   c) 设定*同伴初始化函数 peer initialization function* 的回调入口
4. 同伴初始化函数在每个请求来临的时候调用一次，设置一个负载均衡函数可以进入和操作的数据结构体。
5. *负载均衡函数 load-balancing function* 决定这个请求的去向；在客户端请求来时至少被调用一次（如果后端服务器回复失败，可能要更多次数）。这里发生的东西很有趣。
6. 最后，在和特定的后端服务器结束通信的时候，*同伴释放函数 peer release function* 会更新统计（通信是否成功或失败）。

有很多内容，我将会一点点得展开。

5.1.激活指令

指令声明，既确定了他们的有效存在位置，也给出会调用哪个函数。对于负载均衡的模块应该有的标识 `NGX_HTTP_UPS_CONF`，以便 Nginx 知道这个指令只在 `upstream` 配置部分有效。它应该提供一个指向注册函数的指针。下面是

upstream_hash 模块的指令声明：

```
    { ngx_string("hash"),
      NGX_HTTP_UPS_CONF|NGX_CONF_NOARGS,
      ngx_http_upstream_hash,
      0,
      0,
      NULL },
```

其他没什么了。

5.2.注册函数

上面的回调函数其实是个注册函数，这样称呼是因为它把 upstream 初始化函数注册到了 upstream 配置结构体中。另外，注册函数定义了在特定的 upstream 块中 server 指令一些选项（比如：weight=, fail_timeout=）。这里是 upstream_hash 模块的注册函数

```
ngx_http_upstream_hash(ngx_conf_t *cf, ngx_command_t *cmd, void
*conf)
 {
    ngx_http_upstream_srv_conf_t  *uscf;
    ngx_http_script_compile_t      sc;
    ngx_str_t                     *value;
    ngx_array_t                   *vars_lengths, *vars_values;

    value = cf->args->elts;

    /* the following is necessary to evaluate the argument to "hash"
as a $variable */
    ngx_memzero(&sc, sizeof(ngx_http_script_compile_t));

    vars_lengths = NULL;
    vars_values = NULL;

    sc.cf = cf;
    sc.source = &value[1];
    sc.lengths = &vars_lengths;
    sc.values = &vars_values;
    sc.complete_lengths = 1;
    sc.complete_values = 1;

    if (ngx_http_script_compile(&sc) != NGX_OK) {
```

```
        return NGX_CONF_ERROR;
    }
    /* end of $variable stuff */

    uscf = ngx_http_conf_get_module_srv_conf(cf,
ngx_http_upstream_module);

    /* the upstream initialization function */
    uscf->peer.init_upstream = ngx_http_upstream_init_hash;

    uscf->flags = NGX_HTTP_UPSTREAM_CREATE;

    /* OK, more $variable stuff */
    uscf->values = vars_values->elts;
    uscf->lengths = vars_lengths->elts;

    /* set a default value for "hash_method" */
    if (uscf->hash_function == NULL) {
        uscf->hash_function = ngx_hash_key;
    }

    return NGX_CONF_OK;
}
```

经过上面的函数，我们很直接得取得了$variable，分配一个回调函数，设定一些标识。有哪些标识是可用的呢？

- NGX_HTTP_UPSTREAM_CREATE:在 upstream 块中有有 server 指令。我想象不出你会不用它。
- NGX_HTTP_UPSTREAM_WEIGHT:让 server 指令获取 weight=
- NGX_HTTP_UPSTREAM_MAX_FAILS:允许 max_fails=选项
- NGX_HTTP_UPSTREAM_FAIL_TIMEOUT:允许 fail_timeout=选项
- NGX_HTTP_UPSTREAM_DOWN:允许 down 选项
- NGX_HTTP_UPSTREAM_BACKUP:允许 backup 选项

每个模块都可以设置这些配置选项。用哪些取决于你用它来干嘛。它们不会自动加上去；所有的失败逻辑由模块作者来维护。以后可能会有更多。现在为止，我们还没有完成对回调函数的追踪。接下来，我们讲讲 upstream 初始化函数。

5.3.上游主机初始化函数

上游主机初始化函数的用处是解析主机名称，为每个 socket 分配空间，并分配回调函数。看看 upstream_hash 是如何做的：

```
ngx_int_t
ngx_http_upstream_init_hash(ngx_conf_t *cf,
ngx_http_upstream_srv_conf_t *us)
{
    ngx_uint_t                          i, j, n;
    ngx_http_upstream_server_t      *server;
    ngx_http_upstream_hash_peers_t  *peers;

    /* set the callback */
    us->peer.init = ngx_http_upstream_init_upstream_hash_peer;

    if (!us->servers) {
        return NGX_ERROR;
    }

    server = us->servers->elts;

    /* figure out how many IP addresses are in this upstream block.
*/
    /* remember a domain name can resolve to multiple IP addresses.
*/
    for (n = 0, i = 0; i < us->servers->nelts; i++) {
        n += server[i].naddrs;
    }

    /* allocate space for sockets, etc */
    peers = ngx_pcalloc(cf->pool,
sizeof(ngx_http_upstream_hash_peers_t)
            + sizeof(ngx_peer_addr_t) * (n - 1));

    if (peers == NULL) {
        return NGX_ERROR;
    }

    peers->number = n;

    /* one port/IP address per peer */
    for (n = 0, i = 0; i > us->servers->nelts; i++) {
```

```
        for (j = 0; j < server[i].naddrs; j++, n++) {
            peers->peer[n].sockaddr = server[i].addrs[j].sockaddr;
            peers->peer[n].socklen = server[i].addrs[j].socklen;
            peers->peer[n].name = server[i].addrs[j].name;
        }
    }

    /* save a pointer to our peers for later */
    us->peer.data = peers;

    return NGX_OK;
}
```

这个函数可能多于一个功能。大部分工作看起来很抽象，但事实上不是，那些是我们要用到的。一个简单的策略就是调用另外一个 upstream 模块的初始化函数，让它做所有的繁重工作，覆盖下面的 `us->peer.init` 回调函数。比如，看看这里：

http/modules/ngx_http_upstream_ip_hash_module.c。

关键点是设置同伴初始化函数的指针，比如这里的 `ngx_http_upstream_init_upstream_hash_peer`。

5.4.同伴初始化函数

每当客户端请求的时候，同伴初始化函数就被调用一次。它设定一个结构体，模块会用它选择一个合适的服务器来服务这个请求；该结构体在后端服务器重试的时候是一致的。所以很容易在这里追踪服务器的连接失败或计算过的哈希值。按照约定，这个结构体称为 `ngx_http_upstream_<module name>_peer_data_t`。

此外，同伴初始化函数设定两个回调函数：

- `get`: 负载均衡函数
- `free`:同伴释放函数（通常在连接结束的时候更新一些统计信息）

看起来还不够，它还初始化了一个名叫 `tries` 的变量。只要 `tries` 是正的，Nginx 就会重试这个负载均衡函数。当 `tries` 是零的时候，Nginx 会放弃。这取决于 `get` 和 `free` 函数的 `tries` 设定的初始值。

这里是摘自 upstream_hash 模块的同伴初始化函数：

```
static ngx_int_t
```

```
ngx_http_upstream_init_hash_peer(ngx_http_request_t *r,
    ngx_http_upstream_srv_conf_t *us)
{
    ngx_http_upstream_hash_peer_data_t     *uhpd;

    ngx_str_t val;

    /* evaluate the argument to "hash" */
    if (ngx_http_script_run(r, &val, us->lengths, 0, us->values) ==
NULL) {
        return NGX_ERROR;
    }

    /* data persistent through the request */
    uhpd = ngx_pcalloc(r->pool,
sizeof(ngx_http_upstream_hash_peer_data_t)
        + sizeof(uintptr_t)
          * ((ngx_http_upstream_hash_peers_t *)us->peer.data)->number
                / (8 * sizeof(uintptr_t)));
    if (uhpd == NULL) {
        return NGX_ERROR;
    }

    /* save our struct for later */
    r->upstream->peer.data = uhpd;

    uhpd->peers = us->peer.data;

    /* set the callbacks and initialize "tries" to "hash_again" + 1*/
    r->upstream->peer.free = ngx_http_upstream_free_hash_peer;
    r->upstream->peer.get = ngx_http_upstream_get_hash_peer;
    r->upstream->peer.tries = us->retries + 1;

    /* do the hash and save the result */
    uhpd->hash = us->hash_function(val.data, val.len);

    return NGX_OK;
}
```

看起来不错，现在我们准备选择一台上游服务器了。

5.5.负载均衡函数

主要部分开始了，真正的大餐哦。在这里模块选择一台上游服务器。负载均衡函数的原型是这样的：

```
static ngx_int_t
ngx_http_upstream_get_<module_name>_peer(ngx_peer_connection_t *pc,
void *data);
```

Data 是我们的结构体，包含客户端的连接等有效信息。Pc 包含我们要连接的服务器信息。负载均衡函数的主要工作就是往 pc->sockaddr, pc->socklen, 和 pc->name 里面填进数据。如果你熟悉网络编程，那么对这些变量应该会很熟悉；但它们在这里不是很重要。我们不必留意他们代表什么；只需要知道去哪里找到合适的值填进去。

这个函数需要找到一列服务器，选择一个，把值赋给 pc。让我们看看 upstream_hash 模块是怎么做的。

此前在函数调用 ngx_http_upstream_init_hash ，upstream_hash 模块已经把服务器列表存在结构体 ngx_http_upstream_hash_peer_data_t 中。这个结构体现在是有效的：

```
    ngx_http_upstream_hash_peer_data_t *uhpd = data;
```

现在同伴列表存在 uhpd->peers->peer，根据分配好的哈希值和服务器数目，从这个数组中选择一台同伴服务器。

```
    ngx_peer_addr_t *peer = &uhpd->peers->peer[uhpd->hash % uhpd-
>peers->number];
```

终于大功告成：

```
    pc->sockaddr = peers->sockaddr;
    pc->socklen  = peers->socklen;
    pc->name     = &peers->name;

    return NGX_OK;
```

如果负载均衡函数返回 NGX_OK，它表明，继续尝试这台服务器。如果返回 NGX_BUSY，意味着所有后台主机均无效，Nginx 应该再尝试一次。

但是，我们如何知道服务器无效？或者不想再尝试了呢？

5.6. 同伴释放函数

同伴释放函数在上游连接发生之后运行；它的目的是记录失败数。这里是它的函数原型：

```
void
ngx_http_upstream_free_<module name>_peer(ngx_peer_connection_t *pc,
void *data,
    ngx_uint_t state);
```

前面两个参数跟我们在负载均衡函数看到的一样，第三个参数是 state 变量，标识连接是否成功。它可能是 NGX_PEER_FAILED 和 NGX_PEER_NEXT 的位或（要么连接失败，要么连接成功但是应用返回错误）。零表示连接成功。

这取决于模块作者在碰到这些失败事件的时候会做什么。如果完全不做，那结果应该存进 data，它指向每个请求的数据结构体

如果不想让 Nginx 在请求来的时候继续尝试负载均衡，关键你要在这个函数中把 pc->tries 设为 0。所以最简单的同伴释放函数看起来是这样的：

```
    pc->tries = 0;
```

如果在到达后端服务器过程中碰到错误，客户端会收到 502 Bad Proxy 错误。

这里有个更复杂的例子，摘自 upstream_hash 模块。如果后端连接失败，它会在位向量中标识失败（称作 tried，是一个 uintptr_t 数组），然后选择一台新的后端服务器，直到找到一台不失败的。

```
#define ngx_bitvector_index(index) index / (8 * sizeof(uintptr_t))
#define ngx_bitvector_bit(index) (uintptr_t) 1 << index % (8 *
sizeof(uintptr_t))

static void
ngx_http_upstream_free_hash_peer(ngx_peer_connection_t *pc, void
*data,
    ngx_uint_t state)
{
    ngx_http_upstream_hash_peer_data_t  *uhpd = data;
    ngx_uint_t                           current;

    if (state & NGX_PEER_FAILED
            && --pc->tries)
    {
```

```
        /* the backend that failed */
        current = uhpd->hash % uhpd->peers->number;

       /* mark it in the bit-vector */
        uhpd->tried[ngx_bitvector_index(current)] |=
ngx_bitvector_bit(current);

        do { /* rehash until we're out of retries or we find one that
hasn't been tried */
            uhpd->hash = ngx_hash_key((u_char *)&uhpd->hash,
sizeof(ngx_uint_t));
            current = uhpd->hash % uhpd->peers->number;
        } while ((uhpd->tried[ngx_bitvector_index(current)] &
ngx_bitvector_bit(current)) && --pc->tries);
    }
}
```

只要为负载均衡函数 `uhpd->hash` 找到一个新值就可以了，所以上面的函数是可行的。

许多应用不需要重试或高可用的逻辑，你看上面，这只需要提供几行代码。

6. 完成并编译自定义模块

现在，你应该准备看 nginx 的真实模块，并想知道到底 Nginx 是怎么运行的。看 Nginx 的源代码是必须的，选一个功能和你想要做的相近的模块，看懂它，然后模仿它，实现它，是不很很简单。

Emiller 不是写一个深入阅读 Nginx 模块的文章，绝对不是，这是一篇应用为主的文章，我记录自己的感悟，并写出自己的模块，并与全世界分享。

首先，你需要一个目录来放你的模块，最好不要放在 nginx 的代码目录里面。你的目录里面至少需要两个文件：

- "config"
- "ngx_http_<your module>_module.c"

config 文件会在./configure 配置时被包含，它需要一些配置。

过滤模块的"config"文件：

```
ngx_addon_name=ngx_http_<your module>_module
```

```
HTTP_AUX_FILTER_MODULES="$HTTP_AUX_FILTER_MODULES ngx_http_<your module>_module"
NGX_ADDON_SRCS="$NGX_ADDON_SRCS $ngx_addon_dir/ngx_http_<your module>_module.c"
```

其他模块的"config"文件:

```
ngx_addon_name=ngx_http_<your module>_module
HTTP_MODULES="$HTTP_MODULES ngx_http_<your module>_module"
NGX_ADDON_SRCS="$NGX_ADDON_SRCS $ngx_addon_dir/ngx_http_<your module>_module.c"
```

至于 C 的源文件，我建议直接把类似模块的代码拷过来，然后重命名一下，修改一下，满足你的要求就可以了。

现在开始编译:

```
./configure --add-module=path/to/your/new/module/directory
make
make install
```

如果你需要添加库文件，只要在 config 文件中加入这样的命令:

```
CORE_LIBS="$CORE_LIBS -lfoo"
```

foo 是你要的加入的库名称。

如果你有什么好东东完成了，发到 nginx 的邮件列表，与大家分享吧。

7.高级话题

高级话题已经单独转移到其他地方了，你想看的话，请点击这里。

中文版本修改日志

2009 年 9 月 21 日，第二次修改。
2008 年 9 月 20 日，第一次翻译。