tengine.taobao.org

upstream模块 — Nginx开发从入门到精通

upstream模块 (100%)¶

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。

upstream模块接口¶

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。

memcached模块分析¶

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

nginx提供了ngx_http_memcached模块，提供从memcache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。

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。

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。

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。memcache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。

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。

本节回顾¶

这一节介绍了upstream模块的基本组成。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。

负载均衡模块 (100%)¶

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。

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。

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{}。

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：

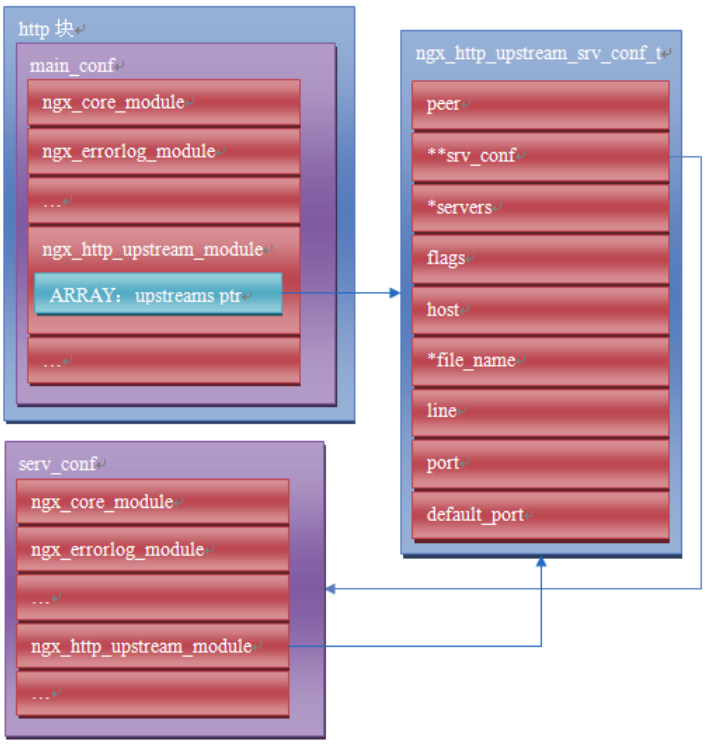
5. NGX_HTTP_UPSTREAM_WEIGHT：可以在server中使用weight属性；
6. NGX_HTTP_UPSTREAM_BACKUP：可以在server中使用backup属性。

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。

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。

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。

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？

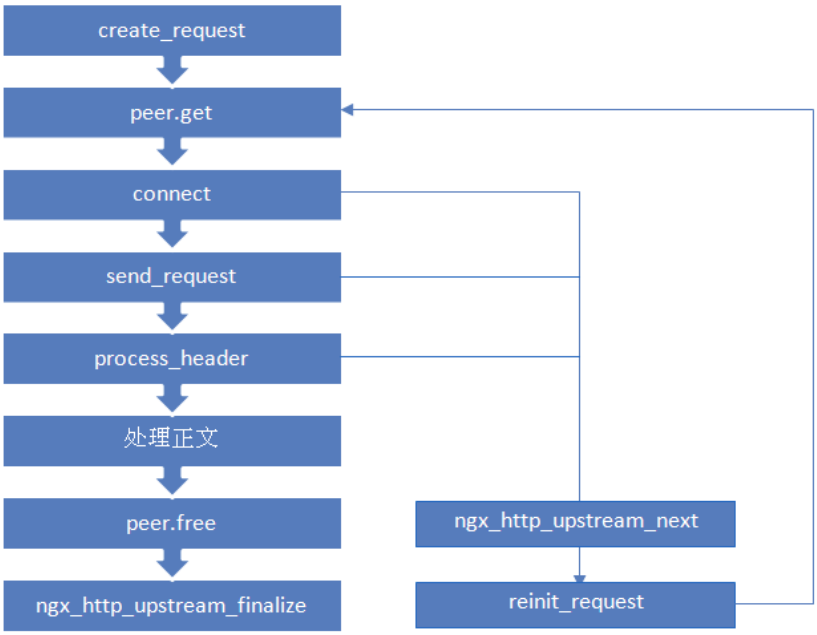
A: 使用后端keepalive连接的时候，连接在使用完以后并不关闭，而是存放在一个队列中，新的请求只需要从队列中取出连接，这些连接都是已经准备好的。

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：



本节回顾¶

这一节介绍了负载均衡模块的基本组成。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。