[Nginx Upstream Keepalive 分析](http://blog.csdn.net/lizhiqiang50/article/details/7551292)

标签： [nginx](http://www.csdn.net/tag/nginx)[cache](http://www.csdn.net/tag/cache)[socket](http://www.csdn.net/tag/socket)[session](http://www.csdn.net/tag/session)[server](http://www.csdn.net/tag/server)[module](http://www.csdn.net/tag/module)

2012-05-09 19:34 5922人阅读 [评论](http://blog.csdn.net/lizhiqiang50/article/details/7551292#comments)(4) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/lizhiqiang50/article/details/7551292#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

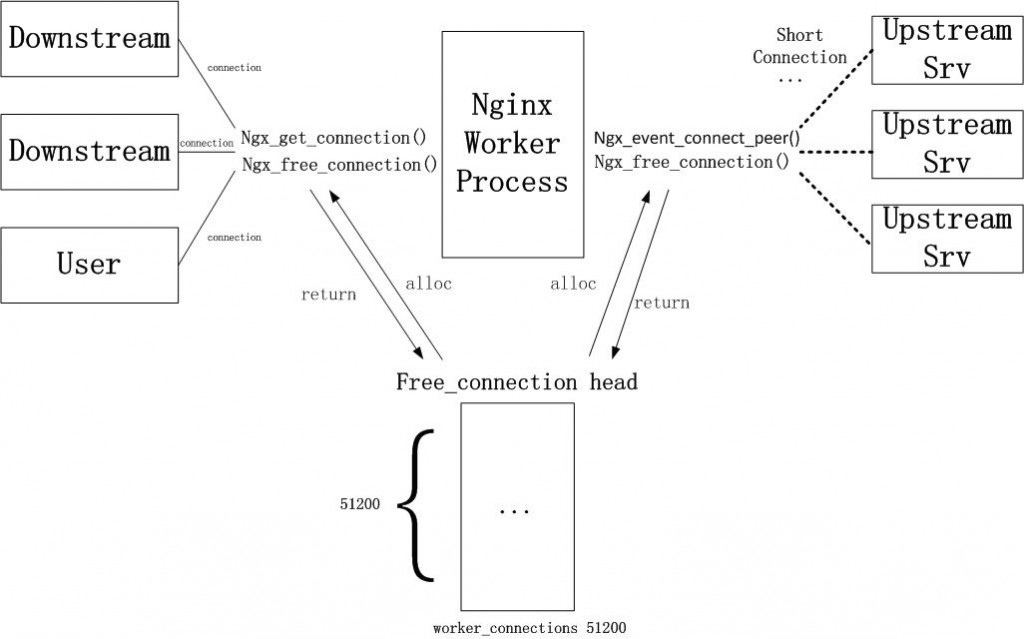
Nginx（2） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

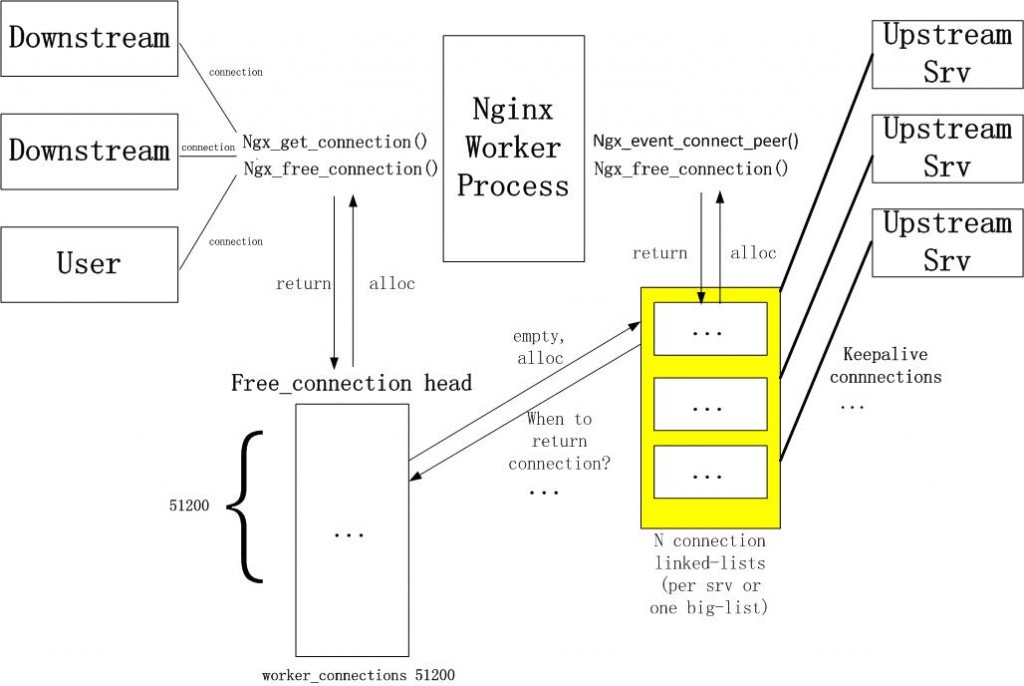
目录[(?)[+]](http://blog.csdn.net/lizhiqiang50/article/details/7551292)

**1.  实现原理**

Nginx 1.1.14版本以前upstream连接建立和获取的机制如下图所示，Nginx会在一开始创建connection pool（进程间不共享，可以避免锁），提供给所有向前/后的连接。



如果要实现upstream长连接，则每个进程需要另外一个connection pool，里面都是长连接。一旦与后端服务器建立连接，则在当前请求连接结束之后不会立即关闭连接，而是把用完的连接保存在一个keepalive connection pool里面，以后每次需要建立向后连接的时候，只需要从这个连接池里面找，如果找到合适的连接的话，就可以直接来用这个连接，不需要重新创建socket或者发起connect()。这样既省下建立连接时在握手的时间消耗，又可以避免TCP连接的slow start。如果在keepalive连接池找不到合适的连接，那就按照原来的步骤重新建立连接。假设连接查找时间可以忽略不计，那么这种方法肯定是有益而无害的（当然，需要少量额外的内存）。



**2.  相关配置**

Nginx Upstream长连接由upstream模式下的keepalive指令控制，并指定可用于长连接的连接数，配置样例如下：

upstream **http\_backend** {

    server 127.0.0.1:8080;

**keepalive 16**;

}

server {

    ...

    location /http/ {

        proxy\_pass http://**http\_backend**;

**proxy\_http\_version 1.1;**

**proxy\_set\_header Connection "";**

        ...

   }

}

目前Nginx只支持反向代理到upstream下配置的server，不支持直接由proxy\_pass指令配置的server，更不支持proxy\_pass参数中包含变量的情况。此外，为支持长连接，需要配置使用HTTP1.1协议（虽然HTTP 1.0可通过设置Connection请求头为“keep-alive”来实现长连接，但这并不推荐）。

此外，由于HTTPPROXY模块默认会将反向代理请求的connection头部设置成Close，因此这里也需要清除connection头部（清除头部即不发送该头部，在HTTP 1.0中默认为长连接）。

**3.  现实细节**

**3.1. 数据结构**

Nginx upstream keepalive的具体实现在文件src/http/modules/ngx\_http\_upstream\_keepalive\_module.c中，该文件中包含以下三个[**数据结构**](http://lib.csdn.net/base/datastructure)：

typedef struct {

       //最大缓存连接个数，由keepalive参数指定（keepalive*connection*）

    ngx\_uint\_t                         max\_cached;

    ngx\_uint\_t                         single;       /\* unsigned:1 \*/

       //长连接队列，其中cache为缓存连接池，free为空闲连接池。初始化时根据keepalive

//指令的参数初始化free队列，后续有连接过来从free队列

//取连接，请求处理结束后将长连接缓存到cache队列，连接被断开（或超时）再从cache

//队列放入free队列

    ngx\_queue\_t                        cache;

    ngx\_queue\_t                        free;

    ngx\_http\_upstream\_init\_pt          original\_init\_upstream;

    ngx\_http\_upstream\_init\_peer\_pt     original\_init\_peer;

}ngx\_http\_upstream\_keepalive\_srv\_conf\_t;

typedef struct {

    ngx\_http\_upstream\_keepalive\_srv\_conf\_t  \*conf;

    ngx\_http\_upstream\_t               \*upstream;

    void                              \*data;

    //保存原始获取peer和释放peer的钩子，它们通常是ngx\_http\_upstream\_get\_round\_robin\_peer和ngx\_http\_upstream\_free\_round\_robin\_peer，nginx负载均衡默认是使用轮询[**算法**](http://lib.csdn.net/base/datastructure)

    ngx\_event\_get\_peer\_pt              original\_get\_peer;

    ngx\_event\_free\_peer\_pt             original\_free\_peer;

#if (NGX\_HTTP\_SSL)

    ngx\_event\_set\_peer\_session\_pt      original\_set\_session;

    ngx\_event\_save\_peer\_session\_pt     original\_save\_session;

#endif

    ngx\_uint\_t                         failed;       /\* unsigned:1 \*/

}ngx\_http\_upstream\_keepalive\_peer\_data\_t;

typedef struct {

    ngx\_http\_upstream\_keepalive\_srv\_conf\_t  \*conf;

    ngx\_queue\_t                        queue;

    ngx\_connection\_t                  \*connection;

       //缓存连接池中保存的后端服务器的地址，后续就是根据相同的socket地址来找出

       //对应的连接，并使用该连接

    socklen\_t                          socklen;

    u\_char                            sockaddr[NGX\_SOCKADDRLEN];

}ngx\_http\_upstream\_keepalive\_cache\_t;

**3.2. 初始化操作**

该模块中只有keepalive一个指令。在nginx解析配置文件遇到upstream模式下的keepalive时，会执行ngx\_http\_upstream\_keepalive函数：

static char \*

ngx\_http\_upstream\_keepalive(ngx\_conf\_t\*cf, ngx\_command\_t \*cmd, void \*conf)

{

……

// 获取upstream模块的server conf

    uscf =ngx\_http\_conf\_get\_module\_srv\_conf(cf, ngx\_http\_upstream\_module);

    kcf = ngx\_http\_conf\_upstream\_srv\_conf(uscf,

                                         ngx\_http\_upstream\_keepalive\_module);

       // 保存原来的初始化upstream的钩子，并设置新的钩子

    kcf->original\_init\_upstream =uscf->peer.init\_upstream

                                  ? uscf->peer.init\_upstream

                                  :ngx\_http\_upstream\_init\_round\_robin;

    uscf->peer.init\_upstream =ngx\_http\_upstream\_init\_keepalive;

       // 设置指令配置的最大缓存连接数

    /\* read options \*/

    value = cf->args->elts;

    n = ngx\_atoi(value[1].data, value[1].len);

    kcf->max\_cached = n;

       ……

    return NGX\_CONF\_ERROR;

}

在upstream模块初始化main conf时，就会根据设置的钩子进行upstream的初始化，该操作是在ngx\_http\_upstream\_init\_main\_conf函数中执行的：

static char \*

ngx\_http\_upstream\_init\_main\_conf(ngx\_conf\_t\*cf, void \*conf)

{

    ngx\_http\_upstream\_main\_conf\_t  \*umcf = conf;

  ……

uscfp = umcf->upstreams.elts;

//遍历upstream数组。upstream的来源包括显性的配置upstream模式（ngx\_http\_upstream）

//和隐性的proxy\_pass指令（ngx\_http\_proxy\_pass），upstream的添加是通过

//ngx\_http\_upstream\_add函数，需要注意的是proxy\_pass指令携带变量参数时不会添加

//upstream

    for (i = 0; i <umcf->upstreams.nelts; i++) {

              //取钩子，并执行钩子。若该upstream来源于包含keepalive指令的upsteam模式，

              //则钩子为前面设置的ngx\_http\_upstream\_init\_keepalive；其他情况均为空，则会

              //使用ngx\_http\_upstream\_init\_round\_robin

        init = uscfp[i]->peer.init\_upstream? uscfp[i]->peer.init\_upstream:

                                           ngx\_http\_upstream\_init\_round\_robin;

        if (init(cf, uscfp[i]) != NGX\_OK) {

            return NGX\_CONF\_ERROR;

        }

}

}

再来看upstreamkeepalive初始化函数ngx\_http\_upstream\_init\_keepalive：

static ngx\_int\_t

ngx\_http\_upstream\_init\_keepalive(ngx\_conf\_t\*cf,

    ngx\_http\_upstream\_srv\_conf\_t \*us)

{

   ……

kcf = ngx\_http\_conf\_upstream\_srv\_conf(us,

                                         ngx\_http\_upstream\_keepalive\_module);

// 先执行原始初始化upstream函数（即ngx\_http\_upstream\_init\_round\_robin），该函数

// 会根据配置的后端地址解析成socket地址，用于连接后端。并设置us->peer.init钩子

// 为ngx\_http\_upstream\_init\_round\_robin\_peer

if (kcf->original\_init\_upstream(cf, us) != NGX\_OK) {

        return NGX\_ERROR;

    }

       // 保存原钩子，并用keepalive的钩子覆盖旧钩子，初始化后端请求的时候会调用这个

       // 新钩子

    kcf->original\_init\_peer =us->peer.init;

    us->peer.init =ngx\_http\_upstream\_init\_keepalive\_peer;

    /\* 申请缓存项，并添加到free队列中，后续用从free队列里面取 \*/

    cached = ngx\_pcalloc(cf->pool,

                sizeof(ngx\_http\_upstream\_keepalive\_cache\_t)\* kcf->max\_cached);

    if (cached == NULL) {

        return NGX\_ERROR;

    }

    ngx\_queue\_init(&kcf->cache);

    ngx\_queue\_init(&kcf->free);

    for (i = 0; i < kcf->max\_cached; i++){

        ngx\_queue\_insert\_head(&kcf->free, &cached[i].queue);

        cached[i].conf = kcf;

    }

    return NGX\_OK;

}

至此，与upstreamkeepalive相关的初始化操作已经全部完毕，接下来反向代理如何使用keepalive连接。

**3.3. 连接的建立/获取**

在nginx收到HTTP请求时，准备将这个请求转发给后端之前，先要与后端建立TCP连接，这些准备操作是在ngx\_http\_upstream\_init\_request函数中：

static void

ngx\_http\_upstream\_init\_request(ngx\_http\_request\_t\*r)

{

       ……

       // 不需要解析地址的情况，包括配置upstream模式下的静态server，静态proxy\_pass

       // 指令，即初始化过程中已经能将地址解析好的情况（静态IP后端或者可通过

       //gethostbyname获取到地址的域名后端）

    if (u->resolved == NULL) {

        uscf = u->conf->upstream;

    } else {

              // 需要解析地址，并且已经解析完毕的情况（已经有socket地址），直接连接后端

        if (u->resolved->sockaddr) {

            if(ngx\_http\_upstream\_create\_round\_robin\_peer(r, u->resolved)

                != NGX\_OK)

            {

               ngx\_http\_upstream\_finalize\_request(r, u,

                                              NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);

                return;

            }

            ngx\_http\_upstream\_connect(r, u);

            return;

        }

              // 需要解析地址，但socket地址还未解析

        host = &u->resolved->host;

        umcf = ngx\_http\_get\_module\_main\_conf(r,ngx\_http\_upstream\_module);

              //先找当前访问的host是否在配置的upstream数组中

        uscfp = umcf->upstreams.elts;

        for (i = 0; i <umcf->upstreams.nelts; i++) {

            uscf = uscfp[i];

            if (uscf->host.len ==host->len

                && ((uscf->port == 0&& u->resolved->no\_port)

                     || uscf->port ==u->resolved->port)

                &&ngx\_memcmp(uscf->host.data, host->data, host->len) == 0)

            {

                goto found;

            }

        }

        if (u->resolved->port == 0) {

            ngx\_log\_error(NGX\_LOG\_ERR,r->connection->log, 0,

                          "no port inupstream \"%V\"", host);

            ngx\_http\_upstream\_finalize\_request(r,u,

                                              NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);

            return;

        }

        temp.name = \*host;

              // 开始进行域名解析

        ctx =ngx\_resolve\_start(clcf->resolver, &temp);

        if (ctx == NULL) {

           ngx\_http\_upstream\_finalize\_request(r, u,

                                              NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);

            return;

        }

        if (ctx == NGX\_NO\_RESOLVER) {

            ngx\_log\_error(NGX\_LOG\_ERR,r->connection->log, 0,

                          "no resolverdefined to resolve %V", host);

           ngx\_http\_upstream\_finalize\_request(r, u, NGX\_HTTP\_BAD\_GATEWAY);

            return;

        }

        ctx->name = \*host;

        ctx->type = NGX\_RESOLVE\_A;

              // 设置DNS解析钩子，解析DNS成功后会执行

        ctx->handler =ngx\_http\_upstream\_resolve\_handler;

        ctx->data = r;

        ctx->timeout =clcf->resolver\_timeout;

        u->resolved->ctx = ctx;

              // 开始处理DNS解析域名

        if (ngx\_resolve\_name(ctx) != NGX\_OK) {

            u->resolved->ctx = NULL;

           ngx\_http\_upstream\_finalize\_request(r, u,

                                              NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);

            return;

        }

        return;

    }

found:

       //调用peer.init初始化对端，若访问请求为配置了keepalive指令的upstream列表中的

       //server，该钩子并初始化成ngx\_http\_upstream\_init\_keepalive\_peer，否则是ngx\_http\_upstream\_init\_round\_robin\_peer。根据上述的代码，可以知道只有不需要解析地址或者能找到upstream的情况才会走到这里

    if (uscf->peer.init(r, uscf) != NGX\_OK){

        ngx\_http\_upstream\_finalize\_request(r,u,

                                          NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);

        return;

    }

    ngx\_http\_upstream\_connect(r, u);

}

可以看出，在不需要解析地址的情况下，初始化peer完毕后（uscf->peer.init）就开始连接后端（ngx\_http\_upstream\_connect），先来看下初始化keepalive peer函数（ngx\_http\_upstream\_init\_keepalive\_peer）：

static ngx\_int\_t ngx\_http\_upstream\_init\_keepalive\_peer(ngx\_http\_request\_t\*r,

    ngx\_http\_upstream\_srv\_conf\_t \*us)

{

   ……

    kcf = ngx\_http\_conf\_upstream\_srv\_conf(us,

                                          ngx\_http\_upstream\_keepalive\_module);

    kp = ngx\_palloc(r->pool,sizeof(ngx\_http\_upstream\_keepalive\_peer\_data\_t));

    if (kp == NULL) {

        return NGX\_ERROR;

    }

       // 先执行原始的初始化peer函数，即ngx\_http\_upstream\_init\_round\_robin\_peer。该

       // 函数内部处理一些与负载均衡相关的操作并分别设置以下四个钩子：

       // r->upstream->peer.get和 r->upstream->peer.free

       // r->upstream->peer.set\_session和 r->upstream->peer.save\_session

    if (kcf->original\_init\_peer(r, us) !=NGX\_OK) {

        return NGX\_ERROR;

    }

       // keepalive模块则保存上述原始钩子，并使用新的各类钩子覆盖旧钩子

    kp->conf = kcf;

    kp->upstream = r->upstream;

    kp->data = r->upstream->peer.data;

    kp->original\_get\_peer =r->upstream->peer.get;

    kp->original\_free\_peer =r->upstream->peer.free;

    r->upstream->peer.data = kp;

    r->upstream->peer.get =ngx\_http\_upstream\_get\_keepalive\_peer;

    r->upstream->peer.free =ngx\_http\_upstream\_free\_keepalive\_peer;

#if(NGX\_HTTP\_SSL)

    kp->original\_set\_session =r->upstream->peer.set\_session;

    kp->original\_save\_session =r->upstream->peer.save\_session;

    r->upstream->peer.set\_session =ngx\_http\_upstream\_keepalive\_set\_session;

    r->upstream->peer.save\_session =ngx\_http\_upstream\_keepalive\_save\_session;

#endif

    return NGX\_OK;

}

在连接后端的函数（ngx\_http\_upstream\_connect）中，会调用ngx\_event\_connect\_peer去建立TCP连接：

ngx\_int\_t ngx\_event\_connect\_peer(ngx\_peer\_connection\_t\*pc)

{

    int                rc;

    ngx\_int\_t          event;

    ngx\_err\_t          err;

    ngx\_uint\_t         level;

    ngx\_socket\_t       s;

    ngx\_event\_t       \*rev, \*wev;

    ngx\_connection\_t  \*c;

       //调用pc->get钩子。如前面所述，若是keepalive upstream，则该钩子是

       //ngx\_http\_upstream\_get\_keepalive\_peer，此时如果存在缓存长连接该函数调用返回的是

       //NGX\_DONE，直接返回上层调用而不会继续往下执行获取新的连接并创建socket，

       //如果不存在缓存的长连接，则会返回NGX\_OK.

       //若是非keepalive upstream，该钩子是ngx\_http\_upstream\_get\_round\_robin\_peer。

    rc = pc->get(pc, pc->data);

    if (rc != NGX\_OK) {

        return rc;

    }

       // 非keepalive upstream或者keepalive upstream为找到缓存连接，则创建socket

    s = ngx\_socket(pc->sockaddr->sa\_family,SOCK\_STREAM, 0);

    ngx\_log\_debug1(NGX\_LOG\_DEBUG\_EVENT,pc->log, 0, "socket %d", s);

    if (s == -1) {

        ngx\_log\_error(NGX\_LOG\_ALERT,pc->log, ngx\_socket\_errno,

                      ngx\_socket\_n "failed");

        return NGX\_ERROR;

    }

c = ngx\_get\_connection(s, pc->log);

……

}

下面看下keepaliveupstream是如何保存缓存连接并当后续使用时如何获取的，这是就是在ngx\_http\_upstream\_get\_keepalive\_peer函数中完成：

static ngx\_int\_t

ngx\_http\_upstream\_get\_keepalive\_peer(ngx\_peer\_connection\_t\*pc, void \*data)

{

    ……

    /\* single pool ofcached connections，只要cache非空，直接从cache里面取，并返回NGX\_DONE\*/

    if (kp->conf->single &&!ngx\_queue\_empty(&kp->conf->cache)) {

        q =ngx\_queue\_head(&kp->conf->cache);

        item = ngx\_queue\_data(q,ngx\_http\_upstream\_keepalive\_cache\_t, queue);

        c = item->connection;

        ngx\_queue\_remove(q);

       ngx\_queue\_insert\_head(&kp->conf->free, q);

              ……

        pc->connection = c;

        pc->cached = 1;

        return NGX\_DONE;

    }

       // 先调用原始getpeer钩子（ngx\_http\_upstream\_get\_round\_robin\_peer）选择后端

    rc = kp->original\_get\_peer(pc,kp->data);

    if (kp->conf->single || rc != NGX\_OK){

        return rc;

    }

    /\* search cache for suitable connection \*/

       // 根据socket地址查找连接cache池，找到直接返回NGX\_DONE，上层调用就不会获取新的连接

    cache = &kp->conf->cache;

    for (q = ngx\_queue\_head(cache);

         q != ngx\_queue\_sentinel(cache);

         q = ngx\_queue\_next(q))

    {

        item = ngx\_queue\_data(q,ngx\_http\_upstream\_keepalive\_cache\_t, queue);

        c = item->connection;

        if (ngx\_memn2cmp((u\_char \*)&item->sockaddr, (u\_char \*) pc->sockaddr,

                         item->socklen,pc->socklen)

            == 0)

        {

            ngx\_queue\_remove(q);

           ngx\_queue\_insert\_head(&kp->conf->free, q);

                     ……

            pc->connection = c;

            pc->cached = 1;

            return NGX\_DONE;

        }

    }

    return NGX\_OK;

}

上述分析了不需要进行解析地址的情况下如何使用keepalive的，当访问的地址需要进行域名解析的情况下（比如proxy\_pass中存在变量，变量的值为一个域名，任何静态域名均会在初始化时候被解析），是无法使用keepalive连接的，请看nginx在解析域名成功后是怎么处理的（即ngx\_http\_upstream\_init\_request函数中设置ctx->handler = ngx\_http\_upstream\_resolve\_handler）：

static void ngx\_http\_upstream\_resolve\_handler(ngx\_resolver\_ctx\_t\*ctx)

{

    ……

       // 创建轮询后端，设置获取peer和释放peer的钩子等（与ngx\_http\_upstream\_init\_round\_robin\_peer的功能有点类似）

    if (ngx\_http\_upstream\_create\_round\_robin\_peer(r,ur) != NGX\_OK){

        ngx\_http\_upstream\_finalize\_request(r,u,

                                          NGX\_HTTP\_INTERNAL\_SERVER\_ERROR);

        return;

    }

    ngx\_resolve\_name\_done(ctx);

    ur->ctx= NULL;

       // 直接开始连接后端，注意该函数中的获取后端的钩子是上述调用中设置的ngx\_http\_upstream\_get\_round\_robin\_peer而不会是ngx\_http\_upstream\_get\_keepalive\_peer

    ngx\_http\_upstream\_connect(r, u);

}

经过上面分析，若想支持任何域名请求都使用keepalive，则需要修改上述函数，需要在这个函数里面调用ngx\_http\_upstream\_init\_keepalive\_peer改变获取/释放连接的接口，然后由于当前nginxkeepalive实现完全依赖于在upstream模式下配置，需要upstream server conf的支持，因此直接调用该初始化函数是无法实现的。

**3.4. 连接的关闭/释放**

在一个HTTP请求处理完毕后，通常会调用ngx\_http\_upstream\_finalize\_request结束请求，并调用释放peer的操作：

static void ngx\_http\_upstream\_finalize\_request(ngx\_http\_request\_t\*r,

    ngx\_http\_upstream\_t \*u, ngx\_int\_t rc)

{

   ……

    u->finalize\_request(r, rc);

       // 如果有设置peer.free钩子，则调用释放peer

       // 非keepalive的情况下该钩子是ngx\_http\_upstream\_free\_round\_robin\_peer

       // keepalive的情况下该钩子是ngx\_http\_upstream\_free\_keepalive\_peer，该钩子会将连接

       // 缓存到长连接cache池，并将u->peer.connection设置成空，防止下面代码关闭连接。

    if (u->peer.free) {

        u->peer.free(&u->peer,u->peer.data, 0);

    }

       // 若与对端的连接为关闭（或未被缓存起来），则关闭连接

    if (u->peer.connection) {

#if(NGX\_HTTP\_SSL)

        if (u->peer.connection->ssl) {

            u->peer.connection->ssl->no\_wait\_shutdown= 1;

            (void)ngx\_ssl\_shutdown(u->peer.connection);

        }

#endif

        if (u->peer.connection->pool) {

           ngx\_destroy\_pool(u->peer.connection->pool);

        }

       ngx\_close\_connection(u->peer.connection);

    }

       //关闭连接后置空

u->peer.connection = NULL;

……

}

接着分析设置keepalive情况下的free peer的函数，即ngx\_http\_upstream\_free\_keepalive\_peer：

static void ngx\_http\_upstream\_free\_keepalive\_peer(ngx\_peer\_connection\_t\*pc, void \*data,

    ngx\_uint\_t state)

{

……

       //通常设置keepalive后连接都是由后端web服务发起的，因此需要添加读事件

    if (ngx\_handle\_read\_event(c->read, 0) !=NGX\_OK) {

        goto invalid;

    }

//如果free队列中可用cache items为空，则从cache队列取一个最近最少使用item，

//将该item对应的那个连接关闭，该item用于保存当前需要释放的连接

    if(ngx\_queue\_empty(&kp->conf->free)) {

        q =ngx\_queue\_last(&kp->conf->cache);

        ngx\_queue\_remove(q);

        item = ngx\_queue\_data(q,ngx\_http\_upstream\_keepalive\_cache\_t, queue);

       ngx\_http\_upstream\_keepalive\_close(item->connection);

} else {

       //free队列不空则直接从队列头取一个item用于保存当前连接

        q= ngx\_queue\_head(&kp->conf->free);

        ngx\_queue\_remove(q);

        item = ngx\_queue\_data(q,ngx\_http\_upstream\_keepalive\_cache\_t, queue);

    }

//缓存当前连接，将item插入cache队列，然后将pc->connection置空，防止上层调用

//ngx\_http\_upstream\_finalize\_request关闭该连接（详见该函数）

    item->connection = c;

   ngx\_queue\_insert\_head(&kp->conf->cache, q);

    pc->connection = NULL;

    if (c->read->timer\_set) {

        ngx\_del\_timer(c->read);

    }

    if (c->write->timer\_set) {

        ngx\_del\_timer(c->write);

}

//设置连接读写钩子。写钩子是一个假钩子（keepalive连接不会由客户端主动关闭）

//读钩子处理关闭keepalive连接的操作（接收到来自后端web服务器的FIN分节）

    c->write->handler =ngx\_http\_upstream\_keepalive\_dummy\_handler;

    c->read->handler =ngx\_http\_upstream\_keepalive\_close\_handler;

       ……

       // 保存socket地址相关信息，后续就是通过查找相同的socket地址来复用该连接

item->socklen = pc->socklen;

    ngx\_memcpy(&item->sockaddr,pc->sockaddr, pc->socklen);

……

}

分析完ngx\_http\_upstream\_free\_keepalive\_peer函数后，在回过头去看ngx\_http\_upstream\_get\_keepalive\_peer就更能理解是如何复用keepalive连接的，free操作将当前连接缓存到cache队列中，并保存该连接对应后端的socket地址，get操作根据想要连接后端的socket地址，遍历查找cache队列，如果找到就使用先前缓存的长连接，未找到就重新建立新的连接。

当free操作发现当前所有cache item用完时（即缓存连接达到上限），会关闭最近未被使用的那个连接，用来缓存新的连接。Nginx官方推荐keepalive的连接数应该配置的尽可能小，以免出现被缓存的连接太多而造成新的连接请求过来时无法获取连接的情况（一个worker进程的总连接池是有限的）。

**3.5. 实现总结**

nginx upstream keepalive实现主要通过当解析到upstream模式下的keepalive命令时，为该upstream改变初始化钩子（ngx\_http\_upstream\_init\_keepalive），而在初始化upstream时又再次改变初始化对端的钩子（ngx\_http\_upstream\_init\_keepalive\_peer），在初始化对端时，再一次改变获取对端和释放对端的钩子（ngx\_http\_upstream\_get\_keepalive\_peer和ngx\_http\_upstream\_free\_keepalive\_peer）。在获取对端时，会先查找所访问的对端是否已经在cache池内（通过对端的socket地址），如果在cache池内则直接使用cache的连接，否则需要建立新的连接；在释放对端连接时，不直接释放连接，而是将连接保存在cache池中，同时使用对端的socket地址标识该对端，方便后续获取时查找。

可以看到，nginxupstream keepalive在缓存连接（free操作）和获取缓存的连接（get操作）时，只是查找匹配后端服务器的地址，而对前端没有任何感知。这就会造成一个问题，当用户1访问某站点后，会建立一个TCP连接，在后端web服务器没有关闭该连接之前，用户2同样访问该站点时，则不用建立TCP连接即可直接访问，也就是说nginx与后端的keepalive连接对前端来说是共享的，这就造成一个性能问题，当几万个用户同时访问同一站点时，这几万个用户与nginx建立了几万条TCP连接，而nginx与后端服务器确有可能只有一条连接，这一条连接需要服务前端的几万个用户，这就大大的影响了系统的性能！

可以在释放对端连接时添加前端IP地址（获取其他标识信息）来标识前端，在获取连接遍历连接cache池时增加前端IP地址查找匹配，这样方能携带前端标识，避免多个前端共用一个后端连接从而影响性能的问题。

**4.  参考文档**

<http://bollaxu.iteye.com/blog/900424>

<http://nginx.org/en/docs/http/ngx_http_upstream_module.html#keepalive>