**tcp/http keep-alive**

[phper](http://ju.outofmemory.cn/feed/4789/) 2016-04-05 **174** 阅读

http://www.firefoxbug.com/index.php/archives/2805/

http://www.firefoxbug.com/index.php/archives/2806/

大家都听过keepalive，但是其实对于keepalive这个词还是很晦涩的，至少我一直都只知道一个大概，直到之前排查线上一些问题，发现keepalive还是有很多玄机的。其实keepalive有两种，一种是TCP层的keepalive，另一种是HTTP层的Keep-Alive。这篇文章先说说tcp层的keepalive

**tcp keepalive**

设想有一种场景：A和B两边通过三次握手建立好TCP连接，然后突然间B就宕机了，之后时间内B再也没有起来。如果B宕机后A和B一直没有数据通信的需求，A就永远都发现不了B已经挂了，那么A的内核里还维护着一份关于A&B之间TCP连接的信息，浪费系统资源。于是在TCP层面引入了keepalive的机制，A会定期给B发空的数据包，通俗讲就是心跳包，一旦发现到B的网络不通就关闭连接。这一点在LVS内尤为明显，因为LVS维护着两边大量的连接状态信息，一旦超时就需要释放连接。

Linux内核对于tcp keepalive的调整主要有以下三个参数

. tcp\_keepalive\_time the interval between the last data packet sent (simple ACKs are not considered data) and the first keepalive probe; after the connection is marked to need keepalive, this counter is not used any further 2. tcp\_keepalive\_intvl the interval between subsequential keepalive probes, regardless of what the connection has exchanged in the meantime 3. tcp\_keepalive\_probes the number of unacknowledged probes to send before considering the connection dead and notifying the application layer

Example

$ cat /proc/sys/net/ipv4/tcp\_keepalive\_time 7200 $ cat /proc/sys/net/ipv4/tcp\_keepalive\_intvl 75 $ cat /proc/sys/net/ipv4/tcp\_keepalive\_probes 9 当tcp发现有tcp\_keepalive\_time(7200)秒未收到对端数据后，开始以间隔tcp\_keepalive\_intvl(75)秒的频率发送的空心跳包，如果连续tcp\_keepalive\_probes(9)次以上未响应代码对端已经down了，close连接

在socket编程时候，可以调用setsockopt指定不同的宏来更改上面几个参数

TCP\_KEEPCNT: tcp\_keepalive\_probes TCP\_KEEPIDLE: tcp\_keepalive\_time TCP\_KEEPINTVL: tcp\_keepalive\_intvl

**Nginx配置tcp keepalive**

Nginx对于keepalive的配置有一大堆，大伙每次看都迷茫了，其实Nginx涉及到tcp层面的keepalive只有一个：so\_keepalive。它属于listen指令的配置参数，具体配置

so\_keepalive=on|off|[keepidle]:[keepintvl]:[keepcnt]

this parameter (1.1.11) configures the “TCP keepalive” behavior for the listening socket. If this parameter is omitted then the operating system’s settings will be in effect for the socket. If it is set to the value “on”, the SO\_KEEPALIVE option is turned on for the socket. If it is set to the value “off”, the SO\_KEEPALIVE option is turned off for the socket. Some operating systems support setting of TCP keepalive parameters on a per-socket basis using the TCP\_KEEPIDLE, TCP\_KEEPINTVL, and TCP\_KEEPCNT socket options. On such systems (currently, Linux 2.4+, NetBSD 5+, and FreeBSD 9.0-STABLE), they can be configured using the keepidle, keepintvl, and keepcnt parameters. One or two parameters may be omitted, in which case the system default setting for the corresponding socket option will be in effect.

* Example

so\_keepalive=30m::10 will set the idle timeout (TCP\_KEEPIDLE) to 30 minutes, leave the probe interval (TCP\_KEEPINTVL) at its system default, and set the probes count (TCP\_KEEPCNT) to 10 probes.

在Nginx的代码里可以看到

./src/http/ngx\_http\_core\_module.c static ngx\_command\_t ngx\_http\_core\_commands[] = { ... // listen 指令解析 -->> call ngx\_http\_core\_listen() { ngx\_string("listen"), NGX\_HTTP\_SRV\_CONF|NGX\_CONF\_1MORE, ngx\_http\_core\_listen, NGX\_HTTP\_SRV\_CONF\_OFFSET, 0, NULL }, ... } static char \* ngx\_http\_core\_listen(ngx\_conf\_t \*cf, ngx\_command\_t \*cmd, void \*conf){ ... // 下面就是 so\_keepalive 后面的参数解析 if (ngx\_strncmp(value[n].data, "so\_keepalive=", 13) == 0) { if (ngx\_strcmp(&value[n].data[13], "on") == 0) { lsopt.so\_keepalive = 1; } else if (ngx\_strcmp(&value[n].data[13], "off") == 0) { lsopt.so\_keepalive = 2; } else { // 自定义系统keepalive的相关设置 ... } if (ngx\_http\_add\_listen(cf, cscf, &lsopt) == NGX\_OK) { return NGX\_CONF\_OK; } } ./src/core/ngx\_connection.c if (ls[i].keepidle) { value = ls[i].keepidle; // 设置 tcp\_keepalive\_time if (setsockopt(ls[i].fd, IPPROTO\_TCP, TCP\_KEEPIDLE, (const void \*) &value, sizeof(int)) == -1) { ngx\_log\_error(NGX\_LOG\_ALERT, cycle->log, ngx\_socket\_errno, "setsockopt(TCP\_KEEPIDLE, %d) %V failed, ignored", value, &ls[i].addr\_text); } } if (ls[i].keepintvl) { value = ls[i].keepintvl; // 设置 tcp\_keepalive\_intvl if (setsockopt(ls[i].fd, IPPROTO\_TCP, TCP\_KEEPINTVL, (const void \*) &value, sizeof(int)) == -1) { ngx\_log\_error(NGX\_LOG\_ALERT, cycle->log, ngx\_socket\_errno, "setsockopt(TCP\_KEEPINTVL, %d) %V failed, ignored", value, &ls[i].addr\_text); } } if (ls[i].keepcnt) { // 设置 tcp\_keepalive\_intvl if (setsockopt(ls[i].fd, IPPROTO\_TCP, TCP\_KEEPCNT, (const void \*) &ls[i].keepcnt, sizeof(int)) == -1) { ngx\_log\_error(NGX\_LOG\_ALERT, cycle->log, ngx\_socket\_errno, "setsockopt(TCP\_KEEPCNT, %d) %V failed, ignored", ls[i].keepcnt, &ls[i].addr\_text); } }

**总结**

这篇文章说了TCP层面的keepalive相关知识以及Nginx的支持tcp keepalive的配置。tcp层面的keepalive存在更多意义上是为了检测两端连接是否正常，注重点是在于连接的本身！要和HTTP层面的keepaplive区分开来，明白这点很重要。

在前面一篇文章中讲了 [TCP的keepalive](http://ju.outofmemory.cn/entry/219957)，这篇文章再讲讲HTTP层面keep-alive。两种keepalive在拼写上面就是不一样的，只是发音一样，于是乎大家就都迷茫了。HTTP层面的keep-alive是我们接触比较多的，也是大家平时口头上的"keepalive"。下面我们就来谈谈HTTP的keep-alive

**短连接&长连接&并行连接**

再说keep-alive之前，先说说HTTP的短连接&长连接。

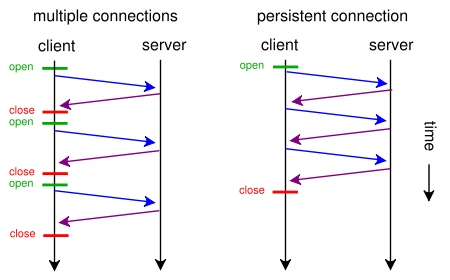
* 短连接

所谓短连接，就是每次请求一个资源就建立连接，请求完成后连接立马关闭。每次请求都经过“创建tcp连接->请求资源->响应资源->释放连接”这样的过程

* 长连接

所谓长连接(persistent connection)，就是只建立一次连接，多次资源请求都复用该连接，完成后关闭。要请求一个页面上的十张图，只需要建立一次tcp连接，然后依次请求十张图，等待资源响应，释放连接。

* 并行连接

所谓并行连接(multiple connections)，其实就是并发的短连接。   


**keep-alive**

具体client和server要从短连接到长连接最简单演变需要做如下改进:

1. client发出的HTTP请求头需要增加Connection:keep-alive字段
2. Web-Server端要能识别Connection:keep-alive字段，并且在http的response里指定Connection:keep-alive字段，告诉client，我能提供keep-alive服务，并且"应允"client我暂时不会关闭socket连接

在HTTP/1.0里，为了实现client到web-server能支持长连接，必须在HTTP请求头里显示指定

Connection:keep-alive

在HTTP/1.1里，就默认是开启了keep-alive，要关闭keep-alive需要在HTTP请求头里显示指定

Connection:close

现在大多数浏览器都默认是使用HTTP/1.1，所以keep-alive都是默认打开的。一旦client和server达成协议，那么长连接就建立好了。

接下来client就给server发送http请求，继续上面的例子:请求十张图片。如果每次"请求->响应"都是独立的，那还好，10张图片的内容都是独立的。但是如果pipeline模式，上一个请求还没响应，下一个请求就发出，这样并发地发出10个请求，对于10个response client要怎么区分呢？而HTTP协议又是没有办法区分的，所以这种情况下必须要求server端地响应是顺序的，通过Conten-Length区分每次请求，这还只是针对静态资源，那对于动态资源无法预知页面大小的情况呢？我还没有深入研究，可以查看 <https://www.byvoid.com/blog/http-keep-alive-header>

另外注意: 指定keep-alive是一种client和server端尽可能需要满足的约定，client和server可以在任意时刻都关闭keep-alive，彼此都不应该受影响。

**Nginx keepa-alive配置**

具体到Nginx的HTTP层的keepalive配置有

* keepalive\_timeout

Syntax: keepalive\_timeout timeout [header\_timeout]; Default: keepalive\_timeout 75s; Context: http, server, location

The first parameter sets a timeout during which a keep-alive client connection will stay open on the server side. The zero value disables keep-alive client connections. The optional second parameter sets a value in the “Keep-Alive: timeout=time” response header field. Two parameters may differ.

* keepalive\_requests

Syntax: keepalive\_requests number; Default: keepalive\_requests 100; Context: http, server, location

Sets the maximum number of requests that can be served through one keep-alive connection. After the maximum number of requests are made, the connection is closed.

可以看看Nginx的关于 keepalive\_timeout 是实现

./src/http/ngx\_http\_request.c static void ngx\_http\_finalize\_connection(ngx\_http\_request\_t \*r){ ... if (!ngx\_terminate && !ngx\_exiting && r->keepalive && clcf->keepalive\_timeout > 0) { ngx\_http\_set\_keepalive(r); return; } ... } static void ngx\_http\_set\_keepalive(ngx\_http\_request\_t \*r){ //如果发现是pipeline请求，判断条件是缓存区里有N和N+1个请求同时存在 if (b->pos < b->last) { /\* the pipelined request \*/ } // 本次请求已经结束，开始释放request对象资源 r->keepalive = 0; ngx\_http\_free\_request(r, 0); c->data = hc; // 如果尝试读取keep-alive的socket返回值不对，可能是客户端close了。那么就关闭socket if (ngx\_handle\_read\_event(rev, 0) != NGX\_OK) { ngx\_http\_close\_connection(c); return; } //开始正式处理pipeline ... rev->handler = ngx\_http\_keepalive\_handler; ... // 设置了一个定时器，触发时间是keepalive\_timeout的设置 ngx\_add\_timer(rev, clcf->keepalive\_timeout); ... } static void ngx\_http\_keepalive\_handler(ngx\_event\_t \*rev){ // 发现超时则关闭socket if (rev->timedout || c->close) { ngx\_http\_close\_connection(c); return; } // 读取keep-alive设置从socket n = c->recv(c, b->last, size); if (n == NGX\_AGAIN) { if (ngx\_handle\_read\_event(rev, 0) != NGX\_OK) { ngx\_http\_close\_connection(c); return; } ... } //此处尚有疑惑？ ngx\_reusable\_connection(c, 0); c->data = ngx\_http\_create\_request(c); // 删除定时器 ngx\_del\_timer(rev); // 重新开始处理请求 rev->handler = ngx\_http\_process\_request\_line; ngx\_http\_process\_request\_line(rev); }

**参考资料**

* <http://nginx.org/en/docs/http/ngx_http_core_module.html>