hanwei_1049

hanwei_1049

博客访问： 616391
博文数量： 229
博客积分： 1698
博客等级： 上尉
技术积分： 2648
用 户 组： 普通用户
注册时间： 2008-12-24 21:49

加关注 短消息
论坛 加好友

个人简介

Linux

文章分类

全部博文（229）
进程间通信（4）
LVS（0）
OpenStack（1）
HTTPS（5）
LUA（1）
版本控制（9）
个人计划（2）
Nginx（26）
MySQL（0）
Trouble Sho（2）
HaProxy（1）
进程调度（0）
ATS（31）
CDN（15）
Redis（0）
TCP/IP协议栈（6）
文件系统/存储（3）
内存管理（12）
系统/脚本（3）
编程相关（8）
攻防研究（17）
体系结构（9）
数据结构（0）
内核相关（16）

TCP TIME_WAIT常见解决方法

2015-03-02 18:35:22

分类： LINUX

转自：http://blog.csdn.net/yunhua_lee/article/details/8146830

1. ——高屋建瓴

tcp连接是网络编程中最基础的概念，基于不同的使用场景，我们一般区分为“长连接”和“短连接”，
长短连接优点和缺点这里就不详细展开了，有心的同学直接去google查询，本文主要关注如何解决tcp短连接TIME_WAIT问题。

短连接最大的优点是方便，特别是脚本语言，由于执行完毕后脚本语言的进程就结束了，基本上都是用短连接。
但短连接最大的缺点是将占用大量的系统资源，例如：本地端口、socket句柄。
导致这个问题的原因其实很简单：tcp协议层并没有长短连接的概念，因此不管长连接还是短连接，连接建立->数据传输->连接关闭的流程和处理都是一样的。

正常的TCP客户端连接在关闭后，会进入一个TIME_WAIT的状态，持续的时间一般在1~4分钟，对于连接数不高的场景，1~4分钟其实并不长，对系统也不会有什么影响，但如果短时间内（例如1s内）进行大量的短连接，则可能出现这样一种情况：客户端所在的操作系统的socket端口和句柄被用尽，系统无法再发起新的连接！
举例来说：假设每秒建立了1000个短连接（Web场景下是很常见的，例如每个请求都去访问memcached），假设TIME_WAIT的时间是1分钟，则1分钟内需要建立6W个短连接，由于TIME_WAIT时间是1分钟，这些短连接1分钟内都处于TIME_WAIT状态，都不会释放，而Linux默认的本地端口范围配置是：
net.ipv4.ip_local_port_range = 32768 61000
不到3W，因此这种情况下新的请求由于没有本地端口就不能建立了。

可以通过如下方式来解决这个问题：
1）可以改为长连接，但代价较大，长连接太多会导致服务器性能问题，而且PHP等脚本语言，需要通过proxy之类的软件才能实现长连接；
2）修改ipv4.ip_local_port_range，增大可用端口范围，但只能缓解问题，不能根本解决问题；
3）客户端程序中设置socket的SO_LINGER选项；
4）客户端机器打开tcp_tw_recycle和tcp_timestamps选项；
5）客户端机器打开tcp_tw_reuse和tcp_timestamps选项；
6）客户端机器设置tcp_max_tw_buckets为一个很小的值；

在解决php连接Memcached的短连接问题过程中，我们主要验证了3）4）5）6）几种方法，采取的是基本功能验证和代码验证，并没有进行性能压力测试验证，
因此实际应用的时候需要注意观察业务运行情况，发现丢包、断连、无法连接等现象时，需要关注是否是因为这些选项导致的。
虽然这几种方法都可以通过google查询到相关信息，但这些信息大部分都是泛泛而谈，而且绝大部分都是人云亦云，没有很大参考价值。我们在定位和处理这些问题过程中，遇到一些疑惑和困难，也花费了一些时间去定位和解决，以下就是相关的经验总结。

2. ——SO_LINGER

SO_LINGER是一个socket选项，通过setsockopt API进行设置，使用起来比较简单，但其实现机制比较复杂，且字面意思上比较难理解。
解释最清楚的当属《Unix网络编程卷1》中的说明（7.5章节），这里简单摘录：
SO_LINGER的值用如下数据结构表示：

```
struct linger {
    int l_onoff; /* 0 = off, nozero = on */
    int l_linger; /* linger time */
};
```

其取值和处理如下：
1）设置 l_onoff为0，则该选项关闭，l_linger的值被忽略，等于内核缺省情况，close调用会立即返回给调用者，如果可能将会传输任何未发送的数据；
2）设置 l_onoff为非0，l_linger为0，则套接口关闭时TCP天折连接，TCP将丢弃保留在套接口发送缓冲区中的任何数据并发送一个RST给对方，
　而不是通常的四分组终止序列，这避免了TIME_WAIT状态；
3）设置 l_onoff 为非0，l_linger为非0，当套接口关闭时内核将拖延一段时间（由l_linger决定）。
　如果套接口缓冲区中仍残留数据，进程将处于睡眠状态，直 到（a）所有数据发送完且被对方确认，之后进行正常的终止序列（描述字访问计数为0）
　或（b）延迟时间到。此种情况下，应用程序检查close的返回值是非常重要的，如果在数据发送完并被确认前时间到，close将返回EWOULDBLOCK错误且套接口发送缓冲区中的任何数据都丢失。
　close的成功返回仅告诉我们发送的数据（和FIN）已由对方TCP确认，它并不能告诉我们对方应用进程是否已读了数据。如果套接口设为非阻塞的，它将不等待close完成。

第一种情况其实和不设置没有区别，第二种情况可以用于避免TIME_WAIT状态，但在Linux上测试的时候，并未发现发送了RST选项，而是正常进行了四步关闭流程，
初步推断是“只有在丢弃数据的时候才发送RST”，如果没有丢弃数据，则走正常的关闭流程。
查看Linux源码，确实有这么一段注释和源码：

点击(此处)折叠或打开

```
=====linux-2.6.37 net/ipv4/tcp.c 1915=====
/* As outlined in RFC 2525, section 2.17, we send a RST here because
 * data was lost. To witness the awful effects of the old behavior of
 * always doing a FIN, run an older 2.1.x kernel or 2.0.x, start a bulk
 * GET in an FTP client, suspend the process, wait for the client to
 * advertise a zero window, then kill -9 the FTP client, wheee...
 * Note: timeout is always zero in such a case.
 */
if (data_was_unread) {
    /* Unread data was tossed, zap the connection. */
    NET_INC_STATS_USER(sock_net(sk), LINUX_MIB_TCPABORTONCLOSE);
    tcp_set_state(sk, TCP_CLOSE);
    tcp_send_active_reset(sk, sk->sk_allocation);
}
```

另外，从原理上来说，这个选项有一定的危险性，可能导致丢数据，使用的时候要小心一些，但我们在实测libmemcached的过程中，没有发现此类现象，
应该是和libmemcached的通讯协议设置有关，也可能是我们的压力不够大，不会出现这种情况。

第三种情况其实就是第一种和第二种的折中处理，且当socket为非阻塞的场景下是没有作用的。
对于应对短连接导致的大量TIME_WAIT连接问题，个人认为第二种处理是最优的选择，libmemcached就是采用这种方式，
从实测情况来看，打开这个选项后，TIME_WAIT连接数为0，且不受网络组网（例如是否虚拟机等）的影响。

3. ——tcp_tw_recycle

【tcp_tw_recycle和tcp_timestamps】
参考官方文档（http://www.kernel.org/doc/Documentation/networking/ip-

• ROSE HA，想说爱你不容易——...
• 记一次sql server 数据库sa用...
• 《Oracle DBA工作笔记》第二...
• oracle本地分区索引跨分区对...
• 半自动化搭建Data Guard的想...

热词专题

• linux+ARM学习路线
• lua编译(linux)

sysctl.txt），tcp_tw_recycle解释如下：
tcp_tw_recycle选项作用为：Enable fast recycling TIME-WAIT sockets. Default value is 0.
tcp_timestamps选项作用为：Enable timestamps as defined in RFC1323. Default value is 1.

这两个选项是linux内核提供的控制选项，和具体的应用程序没有关系，而且网上也能够查询到大量的相关资料，但信息都不够完整，最主要的几个问题如下；
1）快速回收到底有多快？
2）有的资料说只要打开tcp_tw_recycle即可，有的又说要tcp_timestamps同时打开，具体是哪个正确？
3）NAT时可能出现什么问题？

为了回答上面的疑问，只能看代码，看出一些相关的代码供大家参考：

点击(此处)折叠或打开

```
1.  void tcp_time_wait(struct sock *sk, int state, int timeo)
2.  {
3.      struct inet_timewait_sock *tw = NULL;
4.      const struct inet_connection_sock *icsk = inet_csk(sk);
5.      const struct tcp_sock *tp = tcp_sk(sk);
6.      int recycle_ok = 0;
7.
8.      tcp_death_row.period = sysctl_tcp_tw_timeout / INET_TWDR_TWKILL_SLOTS;
9.
10.     //  判断是否快速回收，这里可以看出tcp_tw_recycle和tcp_timestamps两个选项都打开的时候才进行快速回收
11.     if (tcp_death_row.sysctl_tw_recycle && tp->rx_opt.ts_recent_stamp)
12.         recycle_ok = icsk->icsk_af_ops->remember_stamp(sk);
13.
14.     if (tcp_death_row.tw_count < tcp_death_row.sysctl_max_tw_buckets)
15.         tw = inet_twsk_alloc(sk, state);
16.
17.     if (tw != NULL) {
18.         struct tcp_timewait_sock *tcptw = tcp_twsk((struct sock *)tw);
19.         //  计算快速回收的时间，等于 RTO * 3.5，回答第一个问题的关键是RTO（Retransmission Timeout）大概是多
    少
20.         const int rto = (icsk->icsk_rto << 2) - (icsk->icsk_rto >> 1);
21.         ...........
22.         /* Linkage updates. */
23.         __inet_twsk_hashdance(tw, sk, &tcp_hashinfo);
24.
25.         /* Get the TIME_WAIT timeout firing. */
26.         if (timeo < rto)
27.             timeo = rto;
28.         //设置快速回收的时间
29.         if (recycle_ok) {
30.             tw->tw_timeout = rto;
31.         } else {
32.             tw->tw_timeout = sysctl_tcp_tw_timeout;
33.             if (state == TCP_TIME_WAIT)
34.                 timeo = sysctl_tcp_tw_timeout;
35.         }
36.         .......
37. }
```

RFC中有关于RTO计算的详细规定，一共有三个：RFC-793、RFC-2988、RFC-6298，Linux的实现是参考RFC-2988。
对于这些算法的规定和Linuxde 实现，有兴趣的同学可以自己深入研究，实际应用中我们只要记住Linux如下两个边界值：
=====linux-2.6.37 net/ipv4/tcp.c 126================
#define TCP_RTO_MAX ((unsigned)(120*HZ))
#define TCP_RTO_MIN ((unsigned)(HZ/5))
==
这里的HZ是1s，因此可以得出RTO最大是120s，最小是200ms，对于局域网的机器来说，正常情况下RTO基本上就是200ms，因此3.5 RTO就是700ms
也就是说，快速回收是TIME_WAIT的状态持续700ms，而不是正常的2MSL（Linux是1分钟，请参考：include/net/tcp.h 109行TCP_TIMEWAIT_LEN定义）。
实测结果也验证了这个推论，不停的查看TIME_WAIT状态的连接，偶尔能看到1个。

总结一下：
1）快速回收到底有多快？
局域网环境下，700ms就回收；
2）有的资料说只要打开tcp_tw_recycle即可，有的又说要tcp_timestamps同时打开，具体是哪个正确？
需要同时打开，但默认情况下tcp_timestamps就是打开的，所以会有人说只要打开tcp_tw_recycle即可；

综合上面的分析和总结，不能进行快速回收。

附：
1）tcp_timestamps的说明详见RF1323，和TCP的拥塞控制（Congestion control）有关。
2）打开此选项，可能导致无法连接，请参考：http://www.pagefault.info/?p=416
==========

3）NAT时可能出现什么问题？ http://www.pagefault.info/?p=416
今天普空说了一个问题就是如果设置了tcp_tw_recycle，那么如果客户端是NAT出来的，那么就可能会出现连接被直接rst的情况。
然后我google了下，在内核列表也有人说了这个问题 https://lkml.org/lkml/2008/11/15/67。
The big problem is that both are incompatible with NAT. So if you ever talk to any NATed clients don't use it.

点击(此处)折叠或打开

```
#define TCP_PAWS_MSL 60 /* Per-host timestamps are invalidated
                         * after this time. It should be equal
                         * (or greater than) TCP_TIMEWAIT_LEN
                         * to provide reliability equal to one
                         * provided by timewait state.
                         */
#define TCP_PAWS_WINDOW 1 /* Replay window for per-host
                           * timestamps. It must be less than
                           * minimal timewait lifetime.


          /* VJ's idea. We save last timestamp seen
           * from the destination in peer table, when entering
           * state TIME-WAIT, and check against it before
           * accepting new connection request.
           *
           * If "isn" is not zero, this request hit alive
           * timewait bucket, so that all the necessary checks
           * are made in the function processing timewait state.
           */
          if (tmp_opt.saw_tstamp &&
              tcp_death_row.sysctl_tw_recycle &&
              (dst = inet_csk_route_req(sk, &fl4, req)) != NULL &&
              fl4.daddr == saddr &&
              (peer = rt_get_peer((struct rtable *)dst, fl4.daddr)) != NULL) {
              inet_peer_refcheck(peer);
              if ((u32)get_seconds() - peer->tcp_ts_stamp < TCP_PAWS_MSL &&
                  (s32)(peer->tcp_ts - req->ts_recent) >
                              TCP_PAWS_WINDOW) {
                  NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_PAWSPASSIVEREJECTED);
                  goto drop_and_release;
              }
          }
```

可以看到当满足下面所有的条件时，这个syn包将会被丢弃，然后释放相关内存，并发送rst。
1 tcp的option有 time stamp字段.
2 tcp_tw_recycle有设置。
3 在路由表中是否存在完全相同的流(如果打开了xfrm的话，还要比较端口,默认xfrm应该是打开的)，如果存在则直接返回.
4 并且数据包的源地址和新请求的源地址相同.
5 根据路由表以及源地址能够查找到保存的peer(这个可以看我以前的blog，也就是保存了一些连接统计信息).
6 当前时间(接收到syn)比最后一次的时间(time stamp)小于60秒.
7 已经存在peer的最近一次时间戳要大于当前请求进来的时间戳.

从上面可以看到，上面的条件中1/2都是 server端可以控制的，而其他的条件，都是很容易就满足的，因此我们举个例子。

如果客户端是NAT出来的，并且我们server端有打开tcp_tw_recycle ,并且time stamp也没有关闭，那么假设第一个连接进来，然后关闭，此时这个句柄处于time wait状态，然后很快(小于60秒)又一个客户端(相同的源地址，如果打开了xfrm还要相同的端口号)发一个syn包，此时linux内核就会认为这个数据包异常的，因此就会丢掉这个包,并发送rst。

而现在大部分的客户端都是NAT出来的，因此建议tw_recycle还是关闭,或者说server段关闭掉time stamp(/proc/sys/net/ipv4/tcp_timestamps).

4. ——tcp_tw_reuse

tcp_tw_reuse选项的含义如下
（http://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt）：

tcp_tw_reuse - BOOLEAN
Allow to reuse TIME-WAIT sockets for new connections when it is safe from protocol viewpoint. Default value is 0.
1）tcp_tw_reuse选项和tcp_timestamps选项也必须同时打开；
2）重用TIME_WAIT的条件是收到最后一个包后超过1s。

5. ——tcp_max_tw_buckets
参考官方文档（http://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt），解释如下：
tcp_max_tw_buckets - INTEGER
Maximal number of timewait sockets held by system simultaneously.
If this number is exceeded time-wait socket is immediately destroyed and warning is printed.

上一篇：再叙TCP_TIMEWAIT
下一篇：为什么多TCP连接分块下载比单连接下载快？