

解决Linux TIME_WAIT过多造成的问题

1、time_wait的作用：

TIME_WAIT状态存在的理由：

1) 可靠地实现TCP全双工连接的终止 在进行关闭连接四次挥手协议时，最后的ACK是由主动关闭端发出的，如果这个最终的ACK丢失，服务器将重发最终的FIN，因此客户端必须维护状态信息允许它重发最终的ACK。如果不维持这个状态信息，那么客户端将响应RST分节，服务器将此分节解释成一个错误（在java中会抛出connection reset的SocketException）。因而，要实现TCP全双工连接的正常终止，必须处理终止序列四个分节中任何一个分节的丢失情况，主动关闭的客户端必须维持状态信息进入TIME_WAIT状态。

2) 允许老的重复分节在网络中消逝 TCP分节可能由于路由器异常而“迷途”，在迷途期间，TCP发送端可能因确认超时而重发这个分节，迷途的分节在路由器修复后也会被送到最终目的地，这个原来的迷途分节就称为lost duplicate。在关闭一个TCP连接后，马上又重新建立起一个相同的IP地址和端口之间的TCP连接，后一个连接被称为前一个连接的化身（incarnation），那么有可能出现这种情况，前一个连接的迷途重复分组在前一个连接终止后出现，从而被误解成从属于新的化身。为了避免这个情况，TCP不允许处于TIME_WAIT状态的连接启动一个新的化身，因为TIME_WAIT状态持续2MSL，就可以保证当成功建立一个TCP连接的时候，来自连接先前化身的重复分组已经在网络中消逝。

2、大量TIME_WAIT造成的影响：

在高并发短连接的TCP服务器上，当服务器处理完请求后立刻主动正常关闭连接。这个场景下会出现大量socket处于TIME_WAIT状态。如果客户端的并发量持续很高，此时部分客户端就会显示连接不上。

我来解释下这个场景。主动正常关闭TCP连接，都会出现TIMEWAIT。

为什么我们要关注这个高并发短连接呢？有两个方面需要注意：

1. 高并发可以让服务器在短时间范围内同时占用大量端口，而端口有个0~65535的范围，并不是很多，刨除系统和其他服务要用的，剩下的就更少了。
2. 在这个场景中，短连接表示“业务处理+传输数据的时间 远远小于 TIMEWAIT超时的时间”的连接。

这里有个相对长短的概念，比如取一个web页面，1秒钟的http短连接处理完业务，在关闭连接之后，这个业务用过的端口会停留在TIMEWAIT状态几分钟，而这几分钟，其他HTTP请求来临的时候是无法占用此端口的(占着茅坑不拉翔)。单用这个业务计算服务器的利用率会发现，服务器干正经事的时间和端口（资源）被挂着无法被使用的时间的比例是 1：几百，服务器资源严重浪费。（说个题外话，从这个意义出发来考虑服务器性能调优的话，长连接业务的服务就不需要考虑TIMEWAIT状态。同时，如果你对服务器业务场景非常熟悉，你会发现，在实际业务场景中，一般长连接对应的业务的并发量并不会很高。

综合这两个方面，持续的到达一定量的高并发短连接，会使服务器因端口资源不足而拒绝为一部分客户服务。同时，这些端口都是服务器临时分配，无法用SO_REUSEADDR选项解决这个问题。

关于time_wait的反思：

存在即是合理的，既然TCP协议能盛行四十多年，就证明他的设计合理性。所以我们尽可能的使用其原本功能。依靠TIME_WAIT状态来保证我的服务器程序健壮，服务功能正常。那是不是就不要性能了呢？并不是。如果服务器上跑的短连接业务量到了我真的必须处理这个TIMEWAIT状态过多的问题的时候，我的原则是尽量处理，而不是跟TIMEWAIT干上，非先除之而后快。如果尽量处理了，还是解决不了问题，仍然拒绝服务部分请求，那我会采取负载均衡来抗这些高并发的短请求。持续十万并发的短连接请求，两台机器，每台5万个，应该够用了吧。一般的业务量以及国内大部分网站其实并不需要关注这个问题，一句话，达不到时才需要关注这个问题的访问量。

小知识点：

TCP协议发表：1974年12月，卡恩、瑟夫的第一份TCP协议详细说明正式发表。当时美国国防部与三个科学家小组签定了完成TCP/IP的协议，结果由瑟夫领衔的小组捷足先登，首先制定出了通过详细定义的TCP/IP协议标准。当时作了一个试验，将信息包通过点对点的卫星网络，再通过陆地电缆，再通过卫星网络，再由地面传输，贯串欧洲和美国，经过各种电脑系统，全程9.4万公里竟然没有丢失一个数据位，远距离的可靠数据传输证明了TCP/IP协议的成功。

3、案例分析：

首先，根据一个查询TCP连接数，来说明这个问题。

```
1 netstat -ant|awk '/^tcp/ {++S[$NF]} END {for(a in S) print (a,S[a])}'
2 LAST_ACK 14
3 SYN_RECV 348
4 ESTABLISHED 70
5 FIN_WAIT1 229
6 FIN_WAIT2 30
7 CLOSING 33
8 TIME_WAIT 18122
```

状态描述：

- 1 CLOSED：无连接是活动的或正在进行
- 2 LISTEN：服务器在等待进入呼叫
- 3 SYN_RECV：一个连接请求已经到达，等待确认
- 4 SYN_SENT：应用已经开始，打开一个连接



```
5 ESTABLISHED: 正常数据传输状态
6 FIN_WAIT1: 应用说它已经完成
7 FIN_WAIT2: 另一边已同意释放
8 ITMED_WAIT: 等待所有分组死掉
9 CLOSING: 两边同时尝试关闭
10 TIME_WAIT: 另一边已初始化一个释放
11 LAST_ACK: 等待所有分组死掉
```

命令解释：

```
1 先来看看netstat:
2 netstat -n
3 Active Internet connections (w/o servers)
4 Proto Recv-Q Send-Q Local Address Foreign Address State
5 tcp 0 0 123.123.123.123:80 234.234.234.234:12345 TIME_WAIT
6 你实际执行这条命令的时候，可能会得到成千上万条类似上面的记录，不过我们就拿其中的一条就足够了。
7
8 再来看看awk:
9 /^tcp/
10 滤出tcp开头的记录，屏蔽udp，socket等无关记录。
11 state[]相当于定义了一个名叫state的数组
12 NF
13 表示记录的字段数，如上所示的记录，NF等于6
14 $NF
15 表示某个字段的值，如上所示的记录，$NF也就是6，表示第6个字段的值，也就是TIME_WAIT
16 state[$NF]表示数组元素的值，如上所示的记录，就是state[TIME_WAIT] 状态的连接数
17 ++state[$NF]表示把某个数加一，如上所示的记录，就是把state[TIME_WAIT] 状态的连接数加一
18 END
19 表示在最后阶段要执行的命令
20 for(key in state)
21 遍历数组
```

如何尽量处理TIMEWAIT过多？

编辑内核文件/etc/sysctl.conf，加入以下内容：

```
1 net.ipv4.tcp_syncookies = 1 表示开启SYN Cookies。当出现SYN等待队列溢出时，启用cookies来处理，可防范少量SYN攻击，默认为0，表示关闭；
2 net.ipv4.tcp_tw_reuse = 1 表示开启重用。允许将TIME-WAIT sockets重新用于新的TCP连接，默认为0，表示关闭；
3 net.ipv4.tcp_tw_recycle = 1 表示开启TCP连接中TIME-WAIT sockets的快速回收，默认为0，表示关闭。
4 net.ipv4.tcp_fin_timeout 修改系默认的 TIMEOUT 时间
```

然后执行 `/sbin/sysctl -p` 让参数生效。

```
1 /etc/sysctl.conf是一个允许改变正在运行中的Linux系统的接口，它包含一些TCP/IP堆栈和虚拟内存系统的高级选项，修改内核参数永久生效。
```

简单来说，就是打开系统的TIMEWAIT重用和快速回收。

如果以上配置调优后性能还不理想，可继续修改一下配置：

```
1 vi /etc/sysctl.conf
2 net.ipv4.tcp_keepalive_time = 1200
3 #表示当keepalive起用的时候，TCP发送keepalive消息的频度。缺省是2小时，改为20分钟。
4 net.ipv4.ip_local_port_range = 1024 65000
5 #表示用于向外连接的端口范围。缺省情况下很小：32768到61000，改为1024到65000。
6 net.ipv4.tcp_max_syn_backlog = 8192
7 #表示SYN队列的长度，默认为1024，加大队列长度为8192，可以容纳更多等待连接的网络连接数。
8 net.ipv4.tcp_max_tw_buckets = 5000
9 #表示系统同时保持TIME_WAIT套接字的最大数量，如果超过这个数字，TIME_WAIT套接字将立刻被清除并打印警告信息。
10 默认为180000，改为5000。对于Apache、Nginx等服务器，上几行的参数可以很好地减少TIME_WAIT套接字数量，但是对于 Squid，效果却不大。此项参数可以控制
    TIME_WAIT套接字的最大数量，避免Squid服务器被大量的TIME_WAIT套接字拖死。
```

==原理及解决办法=====

执行主动关闭的那端经历了这个状态，并停留MSL（最长分节生命期）的2倍，即2MSL。

TIME_WAIT存在的两个理由：

- 1 可靠的实现TCP全双工连接的终止
- 2 允许老的重复的分节在网络上的消逝

第一个：如果客户端不维持TIME_WAIT状态，那么将响应给服务端一个RST，该分节被服务器解释成一个错误。如果TCP打算执行所有必要的工作以彻底终止某个连接上两个方向的数据流，那么必须正确的处理。执行主动关闭的那一端是处于TIME_WAIT状态的那一端。

第二个：端到端的连接关闭后，过一段时间相同的ip和端口间进行连接，后一个连接成为前一个连接的化身。TCP必须防止来自某个连接的老的重复分组在该连接已经终止后再现，从而被误解成处于同一个连接的某个新的化身。为做到这一点，TCP将不给处于TIME_WAIT状态的连接发起新的化身。

TIME_WAIT状态的持续时间是MSL的2倍，使得某个方向上的分组最多存活MSL秒被丢弃，另一个方向上的应答最多存活MSL秒被丢弃，这样保证每建立一个TCP连接的时候，来自连接先前的化身的老的重复分组都已在网络中消逝。

那么TIME_WAIT状态有什么危害么？

首先要明白两个概念长连接和短连接，

短连接：

我们模拟一下TCP短连接的情况，client向server发起连接请求，server接到请求，然后双方建立连接。client向server发送消息，server回应client，然后一次读写就完成了，这时候双方任何一个都可以发起close操作，不过一般都是client先发起close操作。为什么呢，一般的server不会回复完client后立即关闭连接的，当然不排除有特殊的情况。从上面的描述看，短连接一般只会在client/server间传递一次读写操作

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。

长连接：

接下来我们再模拟一下长连接的情况，client向server发起连接，server接受client连接，双方建立连接。Client与server完成一次读写之后，它们之间的连接并不会主动关闭，后续的读写操作会继续使用这个连接。

首先说一下TCP/IP详解上讲到的TCP保活功能，保活功能主要为服务器应用提供，服务器应用希望知道客户主机是否崩溃，从而可以代表客户使用资源。如果客户已经消失，使得服务器上保留一个半开放的连接，而服务器又在等待来自客户端的数据，则服务器将应远等待客户端的数据，保活功能就是试图在服务器端检测到这种半开放的连接。

如果一个给定的连接在两小时内没有任何的动作，则服务器就向客户发一个探测报文段，客户主机必须处于以下4个状态之一：

客户主机依然正常运行，并从服务器可达。客户的TCP响应正常，而服务器也知道对方是正常的，服务器在两小时后将保活定时器复位。

客户主机已经崩溃，并且关闭或者正在重新启动。在任何一种情况下，客户的TCP都没有响应。服务端将不能收到对探测的响应，并在75秒后超时。服务器总共发送10个这样的探测，每个间隔75秒。如果服务器没有收到一个响应，它就认为客户主机已经关闭并终止连接。

客户主机崩溃并已经重新启动。服务器将收到一个对其保活探测的响应，这个响应是一个复位，使得服务器终止这个连接。

客户机正常运行，但是服务器不可达，这种情况与2类似，TCP能发现的就是没有收到探查的响应。

从上面可以看出，TCP保活功能主要为探测长连接的存活状况，不过这里存在一个问题，存活功能的探测周期太长，还有就是它只是探测TCP连接的存活，属于比较斯文的做法，遇到恶意的连接时，保活功能就不够使了。

在长连接的应用场景下，client端一般不会主动关闭它们之间的连接，Client与server之间的连接如果一直不关闭的话，会存在一个问题，随着客户端连接越来越多，server早晚有扛不住的时候，这时候server端需要采取一些策略，如关闭一些长时间没有读写事件发生的连接，这样可以避免一些恶意连接导致server端服务受损；如果条件再允许就可以以客户端机器为颗粒度，限制每个客户端的最大长连接数，这样可以完全避免某个蛋疼的客户端连累后端服务。

长连接和短连接的产生在于client和server采取的关闭策略，具体的应用场景采用具体的策略，没有十全十美的选择，只有合适的选择。

下面主要讨论TIME_WAIT对短连接的影响：

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14分钟，对于连接数不高的场景，14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；

方法2：

查看系统本地可用端口极限值

```
cat /proc/sys/net/ipv4/ip_local_port_range
```

用这条命令会返回两个数字，默认是：32768 61000，说明这台机器本地能向外连接 $61000 - 32768 = 28232$ 个连接，注意是本地向外连接，不是这台机器的所有连接，不会影响这台机器的80端口的对外连接数。但这个数字会影响到代理服务器（nginx）对app服务器的最大连接数，因为nginx对app是用的异步传输，所以这个环节的连接速度很快，所以堆积的连接就很少。假如nginx对app服务器之间的带宽出了问题或是app服务器有问题，那么可能使连接堆积起来，这时可以通过设定nginx的代理超时时间，来使连接尽快释放掉，一般来说极少能用到28232个连接。

因为有软件使用了40000端口监听，常常出错的话，可以通过设定ip_local_port_range的最小值来解决：

```
echo "40001 61000" > /proc/sys/net/ipv4/ip_local_port_range
```

但是这么做很显然把系统可用端口数减少了，这时可以把ip_local_port_range的最大值往上调，但是好习惯是使用不超过32768的端口来侦听服务，另外也不必要去修改ip_local_port_range数值成1024 65535之类的，意义不大。

方法3：

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 {undefined
    int l_onoff; /* 0 = off, nonzero = 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夭折连接，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

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：

`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) {undefined
/* 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。

方法4：

tcp_tw_recycle和tcp_timestamps】

参考官方文档（<http://www.kernel.org/doc/Documentation/networking/ip-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) 为什么从虚拟机NAT出去发起客户端连接时选项无效，非虚拟机连接就有效？

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：

==linux-2.6.37 net/ipv4/tcp_minisocks.c 269==

```
void tcp_time_wait(struct sock *sk, int state, int timeo)
```

```
{undefined
```

```
struct inet_timewait_sock *tw = NULL;
```

```
const struct inet_connection_sock *icsk = inet_csk(sk);
```

```
const struct tcp_sock *tp = tcp_sk(sk);
```

```
int recycle_ok = 0;
```

```
//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，
```

```
    //而且还有进一步的判断条件，后面会分析，这个进一步的判断条件和第三个问题有关
```

```
if (tcp_death_row.sysctl_tw_recycle && tp->rx_opt.ts_recent_stamp)
```

```
recycle_ok = icsk->icsk_af_ops->remember_stamp(sk);
```

```
if (tcp_death_row.tw_count < tcp_death_row.sysctl_max_tw_buckets)
```

```
tw = inet_twsk_alloc(sk, state);
```

```
if (tw != NULL) {undefined
```

```
struct tcp_timewait_sock *tcptw = tcp_twsk((struct sock *)tw);
```

```
    //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
```

```
const int rto = (icsk->icsk_rto << 2) - (icsk->icsk_rto >> 1);
```

```
//。。。。。。此处省略很多代码。。。。。。
```

```
if (recycle_ok) {undefined
    //设置快速回收的时间
tw->tw_timeout = rto;
} else {undefined
tw->tw_timeout = TCP_TIMEWAIT_LEN;
if (state == TCP_TIME_WAIT)
timeo = TCP_TIMEWAIT_LEN;
}
```

```
//。。。。。。此处省略很多代码。。。。。。
```

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。

最后一个问题是什么从虚拟机发起的连接即使设置了tcp_tw_recycle和tcp_timestamps，也不会快速回收，继续看代码：

tcp_time_wait函数中的代码行：recycle_ok = icsk->icsk_af_ops->remember_stamp(sk);对应的实现如下：

```
=linux-2.6.37 net/ipv4/tcp_ipv4.c 1772=
```

```
int tcp_v4_remember_stamp(struct sock *sk)
```

```

{undefined
    //。。。。。。此处省略很多代码。。。。。。

//当获取对端信息时，进行快速回收，否则不进行快速回收
if (peer) {undefined
if ((s32)(peer->tcp_ts - tp->rx_opt.ts_recent) <= 0 ||
    ((u32)get_seconds() - peer->tcp_ts_stamp > TCP_PAWS_MSL &&
    peer->tcp_ts_stamp <= (u32)tp->rx_opt.ts_recent_stamp)) {undefined
peer->tcp_ts_stamp = (u32)tp->rx_opt.ts_recent_stamp;
peer->tcp_ts = tp->rx_opt.ts_recent;
}
if (release_it)
inet_putpeer(peer);
return 1;
}

return 0;
}

```

上面这段代码应该就是测试的时候虚拟机环境不会释放的原因，当使用虚拟机NAT出去的时候，服务器无法获取隐藏在NAT后的机器信息。生产环境也出现了设置了选项，但TIME_WAIT连接数达到4W多的现象，可能和虚拟机有关，也可能和组网有关。

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；

3) 为什么从虚拟机发起客户端连接时选项无效，非虚拟机连接就有效？

和网络组网有关系，无法获取对端信息时就不进行快速回收；

综合上面的分析和总结，可以看出这种方法不是很保险，在实际应用中可能受到虚拟机、网络组网、防火墙之类的影响从而导致不能进行快速回收。

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>

方法5：

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.

这里的关键在于“协议什么情况下认为是安全的”，由于环境限制，没有办法进行验证，通过看源码简单分析了一下。

`=linux-2.6.37 net/ipv4/tcp_ipv4.c 114=`

```
int tcp_twsk_unique(struct sock *sk, struct sock *sktw, void *twp)
```

```
{undefined
```

```
const struct tcp_timewait_sock *tcptw = tcp_twsk(sktw);
```

```
struct tcp_sock *tp = tcp_sk(sk);
```

```
/* With PAWS, it is safe from the viewpoint
```

```
of data integrity. Even without PAWS it is safe provided sequence  
spaces do not overlap i.e. at data rates <= 80Mbit/sec.
```

Actually, the idea is close to VJ's one, only timestamp cache is

held not per host, but per port pair and TW bucket is used as state holder.

If TW bucket has been already destroyed we fall back to VJ's scheme and use initial timestamp retrieved from peer table.

```
*/
```

//从代码来看，tcp_tw_reuse选项和tcp_timestamps选项也必须同时打开；否则tcp_tw_reuse就不起作用

//另外，所谓的“协议安全”，从代码来看应该是收到最后一个包后超过1s

```
if (tcptw->tw_ts_recent_stamp &&
```

```
(twp == NULL || (sysctl_tcp_tw_reuse &&
    get_seconds() - tcptw->tw_ts_recent_stamp > 1))) {undefined
tp->write_seq = tcptw->tw_snd_nxt + 65535 + 2;
if (tp->write_seq == 0)
tp->write_seq = 1;
tp->rx_opt.ts_recent = tcptw->tw_ts_recent;
tp->rx_opt.ts_recent_stamp = tcptw->tw_ts_recent_stamp;
sock_hold(sktw);
return 1;
}

return 0;
}
```

总结一下：

- 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。

官方手册有一段警告：

It should not be changed without advice/request of technical experts.

对于大部分局域网或者公司内网应用来说，满足条件2) 都是没有问题的，因此官方手册里面的警告其实也没那么可怕：)

方法6：

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.

官方文档没有说明默认值，通过几个系统的简单验证，初步确定默认值是180000。

通过源码查看发现，这个选项比较简单，其实现代码如下：

```
=linux-2.6.37 net/ipv4/tcp_minisocks.c 269==  
void tcp_time_wait(struct sock *sk, int state, int timeo)  
{undefined  
    struct inet_timewait_sock *tw = NULL;  
    const struct inet_connection_sock *icsk = inet_csk(sk);  
    const struct tcp_sock *tp = tcp_sk(sk);  
    int recycle_ok = 0;  
  
    if (tcp_death_row.sysctl_tw_recycle && tp->rx_opt.ts_recent_stamp)  
        recycle_ok = icsk->icsk_af_ops->remember_stamp(sk);  
  
    if (tcp_death_row.tw_count < tcp_death_row.sysctl_max_tw_buckets)  
        tw = inet_twsk_alloc(sk, state);  
  
    if (tw != NULL) {undefined  
        //分配成功，进行TIME_WAIT状态处理，此处略去很多代码  
    } else {undefined  
        //分配失败，不进行处理，只记录日志: TCP: time wait bucket table overflow  
    }  
  
    /* Sorry, if we're out of memory, just CLOSE this  
     * socket up. We've got bigger problems than  
     * non-graceful socket closings.  
     */  
    NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_TCPTIMEWAITOVERFLOW);  
}  
  
tcp_update_metrics(sk);  
tcp_done(sk);  
}
```

实测结果验证，配置为100，TIME_WAIT连接数就稳定在100，且不受组网和其它配置的影响。

官方手册中有一段警告：

This limit exists only to prevent simple DoS attacks, you must not lower the limit artificially, but rather increase it (probably, after increasing installed memory), if network conditions require more than default value.

基本意思是这个用于防止Dos攻击，我们不应该人工减少，如果网络条件需要的话，反而应该增加。

但其实对于我们的局域网或者公司内网应用来说，这个风险并不大。

作者：五月的麦田

出处：<https://www.cnblogs.com/my-show-time/p/16064770.html>

版权：本作品采用「署名-非商业性使用-相同方式共享 4.0 国际」许可协议进行许可。

签名：诗酒趁年华！

分类：网络

标签：TIME_WAIT

 1

 0

支持成功





« 上一篇：[SSH免密码失败原因定位分析](#)

» 下一篇：[安装pip的几种方式](#)

posted @ 2022-03-27 21:56 五月的麦田 阅读(1148) 评论(0) 编辑 收藏 举报

发表评论

编辑 预览

B    



支持 Markdown

自动补全

提交评论 退出 订阅评论 我的博客

[Ctrl+Enter] 快捷键提交

【推荐】园子的商业化努力-AI人才服务：招募AI导师，一起探索AI领域的机会

【推荐】中国云计算领导者：阿里云轻量应用服务器2核2G低至108元/年

【推荐】第五届金蝶云苍穹低代码开发大赛正式启动，百万奖金等你拿！

编辑推荐：

- 秒杀系统常见问题—库存超卖
- 现代 CSS 解决方案：CSS 原生支持的三角函数
- Three.js 进阶之旅：滚动控制模型动画和相机动画
- SQL 优化实战分享
- 分页列表缓存，你真的会吗

即构专区：

- 【活动回顾】后疫情时代，教育行业的增量和变量
- 告别尬聊，解锁秀场+社交新玩法（内含源码+Demo）
- 影响音视频延迟的关键因素（三）： 传输、渲染
- 【技术教程】如何快速实现即构推拉流网络探测及推流测速
- 在线自习室场景爆发，在线教育平台用户时间争夺战打响