inux和windows下TIME\_WAIT过多的解决办法

如果使用了nginx代理，那么系统TIME\_WAIT的数量会变得比较多，这是由于nginx代理使用了短链接的方式和后端交互的原因，使得nginx和后端的ESTABLISHED变得很少而TIME\_WAIT很多。这不但发生在安装nginx的代理服务器上，而且也会使后端的app服务器上有大量的TIME\_WAIT。查阅TIME\_WAIT资料，发现这个状态很多也没什么大问题，但可能因为它占用了系统过多的端口，导致后续的请求无法获取端口而造成障碍。

虽然TIME\_WAIT会造成一些问题，但是要完全枪毙掉它也是不正当的，虽然看起来这么做没什么错。具体可看这篇文档：

http://hi.baidu.com/tim\_bi/blog/item/35b005d784ca91d5a044df1d.html

所以目前看来最好的办法是让每个TIME\_WAIT早点过期。

在linux上可以这么配置：

#让TIME\_WAIT状态可以重用，这样即使TIME\_WAIT占满了所有端口，也不会拒绝新的请求造成障碍

echo "1" > /proc/sys/net/ipv4/tcp\_tw\_reuse

#让TIME\_WAIT尽快回收，我也不知是多久，观察大概是一秒钟

echo "1" > /proc/sys/net/ipv4/tcp\_tw\_recycle

很多文档都会建议两个参数都配置上，但是我发现只用修改tcp\_tw\_recycle就可以解决问题的了，TIME\_WAIT重用TCP协议本身就是不建议打开的。

不能重用端口可能会造成系统的某些服务无法启动，比如要重启一个系统监控的软件，它用了40000端口，而这个端口在软件重启过程中刚好被使用了，就可能会重启失败的。linux默认考虑到了这个问题，有这么个设定：

#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。

因为使用了nginx代理，在windows下也会造成大量TIME\_WAIT，当然windows也可以调整：

在注册表（regedit）的HKEY\_LOCAL\_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters上添加一个DWORD类型的值TcpTimedWaitDelay，值就是秒数，即可。

windows默认是重用TIME\_WAIT，我现在还不知道怎么改成不重用的，本地端口也没查到是什么值，但这些都关系不大，都可以按系统默认运作。

------------------------------------------------------------------------------------------------------------------------

TIME\_WAIT状态

根据TCP协议，主动发起关闭的一方，会进入TIME\_WAIT状态，持续2\*MSL(Max Segment Lifetime)，缺省为240秒，在这个post中简洁的介绍了为什么需要这个状态。

值得一说的是，对于基于TCP的HTTP协议，关闭TCP连接的是Server端，这样，Server端会进入TIME\_WAIT状态，可想而知，对于访问量大的Web Server，会存在大量的TIME\_WAIT状态，假如server一秒钟接收1000个请求，那么就会积压240\*1000=240，000个TIME\_WAIT的记录，维护这些状态给Server带来负担。当然现代操作系统都会用快速的查找算法来管理这些TIME\_WAIT，所以对于新的TCP连接请求，判断是否hit中一个TIME\_WAIT不会太费时间，但是有这么多状态要维护总是不好。

HTTP协议1.1版规定default行为是Keep-Alive，也就是会重用TCP连接传输多个request/response，一个主要原因就是发现了这个问题。还有一个方法减缓TIME\_WAIT压力就是把系统的2\*MSL时间减少，因为240秒的时间实在是忒长了点，对于Windows，修改注册表，在HKEY\_LOCAL\_MACHINE\ SYSTEM\CurrentControlSet\Services\ Tcpip\Parameters上添加一个DWORD类型的值TcpTimedWaitDelay，一般认为不要少于60，不然可能会有麻烦。

对于大型的服务，一台server搞不定，需要一个LB(Load Balancer)把流量分配到若干后端服务器上，如果这个LB是以NAT方式工作的话，可能会带来问题。假如所有从LB到后端Server的IP包的source address都是一样的(LB的对内地址），那么LB到后端Server的TCP连接会受限制，因为频繁的TCP连接建立和关闭，会在server上留下TIME\_WAIT状态，而且这些状态对应的remote address都是LB的，LB的source port撑死也就60000多个(2^16=65536,1~1023是保留端口，还有一些其他端口缺省也不会用），每个LB上的端口一旦进入Server的TIME\_WAIT黑名单，就有240秒不能再用来建立和Server的连接，这样LB和Server最多也就能支持300个左右的连接。如果没有LB，不会有这个问题，因为这样server看到的remote address是internet上广阔无垠的集合，对每个address，60000多个port实在是够用了。

一开始我觉得用上LB会很大程度上限制TCP的连接数，但是实验表明没这回事，LB后面的一台Windows Server 2003每秒处理请求数照样达到了600个，难道TIME\_WAIT状态没起作用？用Net Monitor和netstat观察后发现，Server和LB的XXXX端口之间的连接进入TIME\_WAIT状态后，再来一个LB的XXXX端口的SYN包，Server照样接收处理了，而是想像的那样被drop掉了。翻书，从书堆里面找出覆满尘土的大学时代买的《UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI》，中间提到一句，对于BSD-derived实现，只要SYN的sequence number比上一次关闭时的最大sequence number还要大，那么TIME\_WAIT状态一样接受这个SYN，难不成Windows也算BSD-derived?有了这点线索和关键字(BSD)，找到这个post，在NT4.0的时候，还是和BSD-derived不一样的，不过Windows Server 2003已经是NT5.2了，也许有点差别了。

做个试验，用Socket API编一个Client端，每次都Bind到本地一个端口比如2345，重复的建立TCP连接往一个Server发送Keep-Alive=false的HTTP请求，Windows的实现让sequence number不断的增长，所以虽然Server对于Client的2345端口连接保持TIME\_WAIT状态，但是总是能够接受新的请求，不会拒绝。那如果SYN的Sequence Number变小会怎么样呢？同样用Socket API，不过这次用Raw IP，发送一个小sequence number的SYN包过去，Net Monitor里面看到，这个SYN被Server接收后如泥牛如海，一点反应没有，被drop掉了。

按照书上的说法，BSD-derived和Windows Server 2003的做法有安全隐患，不过至少这样至少不会出现TIME\_WAIT阻止TCP请求的问题，当然，客户端要配合，保证不同TCP连接的sequence number要上涨不要下降。

----------------------------------------------------------------------------------------------------------------------------

Socket中的TIME\_WAIT状态

在高并发短连接的server端，当server处理完client的请求后立刻closesocket此时会出现time\_wait状态然后如果client再并发2000个连接，此时部分连接就连接不上了,用linger强制关闭可以解决此问题，但是linger会导致数据丢失，linger值为0时是强制关闭,无论并发多少多能正常连接上,如果非0会发生部分连接不上的情况!（可调用setsockopt设置套接字的linger延时标志，同时将延时时间设置为0。）

TCP/IP的RFC文档。TIME\_WAIT是TCP连接断开时必定会出现的状态。

是无法避免掉的，这是TCP协议实现的一部分。

在WINDOWS下，可以修改注册表让这个时间变短一些

time\_wait的时间为2msl,默认为4min.

你可以通过改变这个变量:

TcpTimedWaitDelay

把它缩短到30s

TCP要保证在所有可能的情况下使得所有的数据都能够被投递。当你关闭一个socket时，主动关闭一端的socket将进入TIME\_WAIT状态，而被动关闭一方则转入CLOSED状态，这的确能够保证所有的数据都被传输。当一个socket关闭的时候，是通过两端互发信息的四次握手过程完成的，当一端调用close()时，就说明本端没有数据再要发送了。这好似看来在握手完成以后，socket就都应该处于关闭CLOSED状态了。但这有两个问题，首先，我们没有任何机制保证最后的一个ACK能够正常传输，第二，网络上仍然有可能有残余的数据包(wandering duplicates)，我们也必须能够正常处理。

通过正确的状态机，我们知道双方的关闭过程如下

假设最后一个ACK丢失了，服务器会重发它发送的最后一个FIN，所以客户端必须维持一个状态信息，以便能够重发ACK；如果不维持这种状态，客户端在接收到FIN后将会响应一个RST，服务器端接收到RST后会认为这是一个错误。如果TCP协议能够正常完成必要的操作而终止双方的数据流传输，就必须完全正确的传输四次握手的四个节，不能有任何的丢失。这就是为什么socket在关闭后，仍然处于 TIME\_WAIT状态，因为他要等待以便重发ACK。

如果目前连接的通信双方都已经调用了close()，假定双方都到达CLOSED状态，而没有TIME\_WAIT状态时，就会出现如下的情况。现在有一个新的连接被建立起来，使用的IP地址与端口与先前的完全相同，后建立的连接又称作是原先连接的一个化身。还假定原先的连接中有数据报残存于网络之中，这样新的连接收到的数据报中有可能是先前连接的数据报。为了防止这一点，TCP不允许从处于TIME\_WAIT状态的socket建立一个连接。处于TIME\_WAIT状态的socket在等待两倍的MSL时间以后（之所以是两倍的MSL，是由于MSL是一个数据报在网络中单向发出到认定丢失的时间，一个数据报有可能在发送图中或是其响应过程中成为残余数据报，确认一个数据报及其响应的丢弃的需要两倍的MSL），将会转变为CLOSED状态。这就意味着，一个成功建立的连接，必然使得先前网络中残余的数据报都丢失了。

由于TIME\_WAIT状态所带来的相关问题，我们可以通过设置SO\_LINGER标志来避免socket进入TIME\_WAIT状态，这可以通过发送RST而取代正常的TCP四次握手的终止方式。但这并不是一个很好的主意，TIME\_WAIT对于我们来说往往是有利的。

客户端与服务器端建立TCP/IP连接后关闭SOCKET后，服务器端连接的端口

状态为TIME\_WAIT

是不是所有执行主动关闭的socket都会进入TIME\_WAIT状态呢？

有没有什么情况使主动关闭的socket直接进入CLOSED状态呢？

主动关闭的一方在发送最后一个 ack 后

就会进入 TIME\_WAIT 状态 停留2MSL（max segment lifetime）时间

这个是TCP/IP必不可少的，也就是“解决”不了的。

也就是TCP/IP设计者本来是这么设计的

主要有两个原因

1。防止上一次连接中的包，迷路后重新出现，影响新连接

   （经过2MSL，上一次连接中所有的重复包都会消失）

2。可靠的关闭TCP连接

   在主动关闭方发送的最后一个 ack(fin) ，有可能丢失，这时被动方会重新发

   fin, 如果这时主动方处于 CLOSED 状态 ，就会响应 rst 而不是 ack。所以

   主动方要处于 TIME\_WAIT 状态，而不能是 CLOSED 。

TIME\_WAIT 并不会占用很大资源的，除非受到攻击。

还有，如果一方 send 或 recv 超时，就会直接进入 CLOSED 状态

讨厌的 Socket TIME\_WAIT 问题

netstat -n | awk '/^tcp/ {++state[$NF]} END {for(key in state) print key,"\t",state[key]}'

会得到类似下面的结果，具体数字会有所不同：

LAST\_ACK 1

SYN\_RECV 14

ESTABLISHED 79

FIN\_WAIT1 28

FIN\_WAIT2 3

CLOSING 5

TIME\_WAIT 1669

状态：描述

CLOSED：无连接是活动的或正在进行

LISTEN：服务器在等待进入呼叫

SYN\_RECV：一个连接请求已经到达，等待确认

SYN\_SENT：应用已经开始，打开一个连接

ESTABLISHED：正常数据传输状态

FIN\_WAIT1：应用说它已经完成

FIN\_WAIT2：另一边已同意释放

ITMED\_WAIT：等待所有分组死掉

CLOSING：两边同时尝试关闭

TIME\_WAIT：另一边已初始化一个释放

LAST\_ACK：等待所有分组死掉

也就是说，这条命令可以把当前系统的网络连接状态分类汇总。

下面解释一下为啥要这样写：

一个简单的管道符连接了netstat和awk命令。

------------------------------------------------------------------

每个TCP报文在网络内的最长时间，就称为MSL（Maximum Segment Lifetime），它的作用和IP数据包的TTL类似。

RFC793指出，MSL的值是2分钟，但是在实际的实现中，常用的值有以下三种：30秒，1分钟，2分钟。

注意一个问题，进入TIME\_WAIT状态的一般情况下是客户端，大多数服务器端一般执行被动关闭，不会进入TIME\_WAIT状态，当在服务

器端关闭某个服务再重新启动时，它是会进入TIME\_WAIT状态的。

举例：

1.客户端连接服务器的80服务，这时客户端会启用一个本地的端口访问服务器的80，访问完成后关闭此连接，立刻再次访问服务器的

80，这时客户端会启用另一个本地的端口，而不是刚才使用的那个本地端口。原因就是刚才的那个连接还处于TIME\_WAIT状态。

2.客户端连接服务器的80服务，这时服务器关闭80端口，立即再次重启80端口的服务，这时可能不会成功启动，原因也是服务器的连

接还处于TIME\_WAIT状态。

vi /etc/sysctl.conf

net.ipv4.tcp\_syncookies = 1

net.ipv4.tcp\_tw\_reuse = 1

net.ipv4.tcp\_tw\_recycle = 1

net.ipv4.tcp\_fin\_timeout = 30

然后执行 /sbin/sysctl -p 让参数生效。

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；

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；

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。

net.ipv4.tcp\_fin\_timeout 修改系統默认的 TIMEOUT 时间

# TCP/IP详解--TCP连接中TIME\_WAIT状态过多

原创 2014年03月18日 10:30:59

* 标签：
* [网络编程](http://so.csdn.net/so/search/s.do?q=%E7%BD%91%E7%BB%9C%E7%BC%96%E7%A8%8B&t=blog) /
* [c](http://so.csdn.net/so/search/s.do?q=c&t=blog)
* 48116

TIMEWAIT状态本身和应用层的客户端或者服务器是没有关系的。仅仅是主动关闭的一方，在使用FIN|ACK|FIN|ACK四分组正常关闭TCP连接的时候会出现这个TIMEWAIT。服务器在处理客户端请求的时候，如果你的程序设计为服务器主动关闭，那么你才有可能需要关注这个TIMEWAIT状态过多的问题。如果你的服务器设计为被动关闭，那么你首先要关注的是CLOSE\_WAIT。

## 原则

TIMEWAIT并不是多余的。在TCP协议被创造，经历了大量的实际场景实践之后，TIMEWAIT出现了，因为TCP主动关闭连接的一方需要TIMEWAIT状态，它是我们的朋友。这是《UNIX网络编程》的作者----Steven对TIMEWAIT的态度。

## TIMEWAIT是友好的

    TCP要保证在所有可能的情况下使得所有的数据都能够被正确送达。当你关闭一个socket时，主动关闭一端的socket将进入TIME\_WAIT状态，而被动关闭一方则转入CLOSED状态，这的确能够保证所有的数据都被传输。当一个socket关闭的时候，是通过两端四次握手完成的，当一端调用close()时，就说明本端没有数据要发送了。这好似看来在握手完成以后，socket就都可以处于初始的CLOSED状态了，其实不然。原因是这样安排状态有两个问题， 首先，我们没有任何机制保证最后的一个ACK能够正常传输，第二，网络上仍然有可能有残余的数据包(wandering duplicates)，我们也必须能够正常处理。  
TIMEWAIT就是为了解决这两个问题而生的。

1.假设最后一个ACK丢失了，被动关闭一方会重发它的FIN。主动关闭一方必须维持一个有效状态信息（TIMEWAIT状态下维持），以便能够重发ACK。如果主动关闭的socket不维持这种状态而进入CLOSED状态，那么主动关闭的socket在处于CLOSED状态时，接收到FIN后将会响应一个RST。被动关闭一方接收到RST后会认为出错了。如果TCP协议想要正常完成必要的操作而终止双方的数据流传输，就必须完全正确的传输四次握手的四个节，不能有任何的丢失。这就是为什么socket在关闭后，仍然处于TIME\_WAIT状态的第一个原因，因为他要等待以便重发ACK。

2.假设目前连接的通信双方都已经调用了close()，双方同时进入CLOSED的终结状态，而没有走TIME\_WAIT状态。会出现如下问题，现在有一个新的连接被建立起来，使用的IP地址与端口与先前的完全相同，后建立的连接是原先连接的一个完全复用。还假定原先的连接中有数据报残存于网络之中，这样新的连接收到的数据报中有可能是先前连接的数据报。为了防止这一点，TCP不允许新连接复用TIME\_WAIT状态下的socket。处于TIME\_WAIT状态的socket在等待两倍的MSL时间以后（之所以是两倍的MSL，是由于MSL是一个数据报在网络中单向发出到认定丢失的时间，一个数据报有可能在发送途中或是其响应过程中成为残余数据报，确认一个数据报及其响应的丢弃的需要两倍的MSL），将会转变为CLOSED状态。这就意味着，一个成功建立的连接，必然使得先前网络中残余的数据报都丢失了。

## 大量TIMEWAIT在某些场景中导致的令人头疼的业务问题

大量TIMEWAIT出现，并且需要解决的场景  
      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按照主动正常关闭连接。。。这个场景下，会出现大量socket处于TIMEWAIT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。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，假如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:(

## 一对矛盾

TIMEWAIT既友好，又令人头疼。  
但是我们还是要抱着一个友好的态度来看待它，因为它尽它的能力保证了服务器的健壮性。

## 可行而且必须存在，但是不符合原则的解决方式

1. linux没有在sysctl或者proc文件系统暴露修改这个TIMEWAIT超时时间的接口，可以修改内核协议栈代码中关于这个TIMEWAIT的超时时间参数，重编内核，让它缩短超时时间，加快回收；  
2. 利用SO\_LINGER选项的强制关闭方式，发RST而不是FIN，来越过TIMEWAIT状态，直接进入CLOSED状态。详见我的博文[《TCP之选项SO\_LINGER》](http://blog.chinaunix.net/uid-29075379-id-3904022.html)。

## 我如何看待这个问题

为什么说上述两种解决方式我觉得可行，但是不符合原则？  
我首先认为，我要依靠TIMEWAIT状态来保证我的服务器程序健壮，网络上发生的乱七八糟的问题太多了，我先要服务功能正常。  
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，那我会采取分机器的方法，让多台机器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，达不到需要关注这个问题的访问量。  
真正地必须使用上述我认为不合理的方式来解决这个问题的场景有没有呢？答案是有。  
像淘宝、百度、新浪、京东商城这样的站点，由于有很多静态小图片业务，如果过度分服会导致需要上线大量机器，多买机器多花钱，得多配机房，多配备运维工程师来守护这些机器，成本增长非常严重。。。这个时候就要尽一切可能去优化。  
题外话，服务器上的技术问题没有绝对，一切都是为业务需求服务的。

## 如何****尽量****处理TIMEWAIT过多

sysctl改两个内核参数就行了，如下：  
net.ipv4.tcp\_tw\_reuse = 1  
net.ipv4.tcp\_tw\_recycle = 1  
简单来说，就是打开系统的TIMEWAIT重用和快速回收，至于怎么重用和快速回收，这个问题我没有深究，实际场景中这么做确实有效果。用netstat或者ss观察就能得出结论。  
还有些朋友同时也会打开syncookies这个功能，如下：  
net.ipv4.tcp\_syncookies = 1  
打开这个syncookies的目的实际上是：“在服务器资源（并非单指端口资源，拒绝服务有很多种资源不足的情况）不足的情况下，尽量不要拒绝TCP的syn（连接）请求，尽量把syn请求缓存起来，留着过会儿有能力的时候处理这些TCP的连接请求”。  
如果并发量真的非常非常高，打开这个其实用处不大。