高性能网络编程7--tcp连接的内存使用

2014年01月23日 17:47:28 阅读数：34406

当服务器的并发TCP连接数以十万计时，我们就会对一个TCP连接在操作系统内核上消耗的内存多少感兴趣。socket编程方法提供了SO_SNDBUF、SO_RCVBUF这样的接口来设置连接的读写缓存，linux上还提供了以下系统级的配置来整体设置服务器上的TCP内存使用，但这些配置看名字却有些互相冲突、概念模糊的感觉，如下（sysctl -a命令可以查看这些配置）：

```
net.ipv4.tcp_rmem = 8192 87380 16777216
net.ipv4.tcp_wmem = 8192 65536 16777216
net.ipv4.tcp_mem = 8388608 12582912 16777216
net.core.rmem_default = 262144
net.core.wmem_default = 262144
net.core.rmem_max = 16777216
net.core.wmem_max = 16777216
```

还有一些较少被提及的、也跟TCP内存相关的配置：

```
net.ipv4.tcp_moderate_rcvbuf = 1
net.ipv4.tcp_adv_win_scale = 2
```

（注：为方便下文讲述，介绍以上系统配置时前缀省略掉，配置值以空格分隔的多个数字以数组来称呼，例如tcp_rmem[2]表示上面第一行最后一列16777216。）

网上可以找到很多这些系统配置项的说明，然而往往还是让人费解，例如，tcp_rmem[2]和rmem_max似乎都跟接收缓存最大值有关，但它们却可以不一致，究竟有什么区别？或者tcp_wmem[1]和wmem_default似乎都表示发送缓存的默认值，冲突了怎么办？在用抓包软件抓到的syn握手包里，为什么TCP接收窗口大小似乎与这些配置完全没关系？

TCP连接在进程中使用的内存大小千变万化，通常程序较复杂时可能不是直接基于socket编程，这时平台级的组件可能就封装了TCP连接使用到的用户态内存。不同的平台、组件、中间件、网络库都大不相同。而内核态为TCP连接分配内存的算法则是基本不变的，这篇文章将试图说明TCP连接在内核态中会使用多少内存，操作系统使用怎样的策略来平衡宏观的吞吐量与微观的某个连接传输速度。这篇文章也将一如既往的面向应用程序开发者，而不是系统级的内核开发者，所以，不会详细的介绍为了一个TCP连接、一个TCP报文操作系统分配了多少字节的内存，内核级的数据结构也不是本文的关注点，这些也不是应用级程序员的关注点。这篇文章主要描述linux内核为了TCP连接上传输的数据是怎样管理读写缓存的。

一、缓存上限是什么？

（1）先从应用程序编程时可以设置的SO_SNDBUF、SO_RCVBUF说起。

无论何种语言，都对TCP连接提供基于setsockopt方法实现的SO_SNDBUF、SO_RCVBUF，怎么理解这两个属性的意义呢？
SO_SNDBUF、SO_RCVBUF都是个体化的设置，即，只会影响到设置过的连接，而不会对其他连接生效。SO_SNDBUF表示这个连接上的内核写缓存上限。实际上，进程设置的SO_SNDBUF也并不是真的上限，在内核中会把这个值翻一倍再作为写缓存上限使用，我们不需要纠结这种细节，只需要知道，当设置了SO_SNDBUF时，就相当于

划定了所操作的TCP连接上的写缓存能够使用的最大内存。然而，这个值也不是可以由着进程随意设置的，它会受制于系统级的上下限，当它大于上面的系统配置wmem_max（net.core.wmem_max）时，将会被wmem_max替代（同样翻一倍）；而当它特别小时，例如在2.6.18内核中设计的写缓存最小值为2K字节，此时也会被直接替代为2K。

SO_RCVBUF表示连接上的读缓存上限，与SO_SNDBUF类似，它也受制于rmem_max配置项，实际在内核中也是2倍大小作为读缓存的使用上限。SO_RCVBUF设置时也有下限，同样在2.6.18内核中若这个值小于256字节就会被256所替代。

（2）那么，可以设置的SO_SNDBUF、SO_RCVBUF缓存使用上限与实际内存到底有怎样的关系呢？

TCP连接所用内存主要由读写缓存决定，而读写缓存的大小只与实际使用场景有关，在实际使用未达到上限时，SO_SNDBUF、SO_RCVBUF是不起任何作用的。对读缓存来说，接收到一个来自连接对端的TCP报文时，会导致读缓存增加，当然，如果加上报文大小后读缓存已经超过了读缓存上限，那么这个报文会被丢弃从而读缓存大小维持不变。什么时候读缓存使用的内存会减少呢？当进程调用read、recv这样的方法读取TCP流时，读缓存就会减少。因此，读缓存是一个动态变化的、实际用到多少才分配多少的缓冲内存，当这个连接非常空闲时，且用户进程已经把连接上接收到的数据都消费了，那么读缓存使用内存就是0。

写缓存也是同样道理。当用户进程调用send或者write这样的方法发送TCP流时，就会造成写缓存增大。当然，如果写缓存已经到达上限，那么写缓存维持不变，向用户进程返回失败。而每当接收到TCP连接对端发来的ACK确认了报文的成功发送时，写缓存就会减少，这是因为TCP的可靠性决定的，发出去报文后由于担心报文丢失而不会销毁它，可能会由重发定时器来重发报文。因此，写缓存也是动态变化的，空闲的正常连接上，写缓存所用内存通常也为0。

因此，只有当接收网络报文的速度大于应用程序读取报文的速度时，可能使读缓存达到了上限，这时这个缓存使用上限才会起作用。所起作用为：丢弃掉新收到的报文，防止这个TCP连接消耗太多的服务器资源。同样，当应用程序发送报文的速度大于接收对方确认ACK报文的速度时，写缓存可能达到上限，从而使send这样的方法失败，内核不为其分配内存。

二、缓存的大小与TCP的滑动窗口到底有什么关系？

（1）滑动窗口的大小与缓存大小肯定是有关的，但却不是一一对应的关系，更不会与缓存上限具有一一对应的关系。因此，网上很多资料介绍rmem_max等配置设置了滑动窗口的最大值，与我们tcpdump抓包时看到的win窗口值完全不一致，是讲得通的。下面我们来细探其分别在哪里。

读缓存的作用有2个：1、将无序的、落在接收滑动窗口内的TCP报文缓存起来；2、当有序的、可以供应用程序读取的报文出现时，由于应用程序的读取是延时的，所以会把待应用程序读取的报文也保存在读缓存中。所以，读缓存一分为二，一部分缓存无序报文，一部分缓存待延时读取的有序报文。这两部分缓存大小之和由于受制于同一个上限值，所以它们是会互相影响的，当应用程序读取速率过慢时，这块过大的应用缓存将会影响到套接字缓存，使接收滑动窗口缩小，从而通知连接的对端降低发送速度，避免无谓的网络传输。当应用程序长时间不读取数据，造成应用缓存将套接字缓存挤压到没空间，那么连接对端会收到接收窗口为0的通知，告诉对方：我现在消化不了更多的报文了。

反之，接收滑动窗口也是一直在变化的，我们用tcpdump抓三次握手的报文：

```
14:49:52.421674 IP houyi-vm02.dev.sd.aliyun.com.6400 > r14a02001.dg.tbsite.net.54073: S
```

可以看到初始的接收窗口是5792，当然也远小于最大接收缓存（稍后介绍的tcp_rmem[1]）。
这当然是有原因的，TCP协议需要考虑复杂的网络环境，所以使用了慢启动、拥塞窗口（参见高性能网络编程2---TCP消息的发送），建立连接时的初始窗口并不会按照接收缓存的最大值来初始化。这是因为，过大的初始窗口

从宏观角度，对整个网络可能造成过载引发恶性循环，也就是考虑到链路上各环节的诸多路由器、交换机可能扛不住压力不断的丢包（特别是广域网），而微观的TCP连接的双方却只按照自己的读缓存上限作为接收窗口，这样双方的发送窗口（对方的接收窗口）越大就对网络产生越坏的影响。慢启动就是使初始窗口尽量的小，随着接收到对方的有效报文，确认了网络的有效传输能力后，才开始增大接收窗口。

不同的linux内核有着不同的初始窗口，我们以广为使用的linux2.6.18内核为例，在以太网里，MSS大小为1460，此时初始窗口大小为4倍的MSS，简单列下代码（*rcv_wnd即初始接收窗口）：

```
int init_cwnd = 4;
if (mss > 1460*3)
 init_cwnd = 2;
else if (mss > 1460)
 init_cwnd = 3;
if (*rcv_wnd > init_cwnd*mss)
 *rcv_wnd = init_cwnd*mss;
```

大家可能要问，为何上面的抓包上显示窗口其实是5792，并不是1460*4为5840呢？这是因为1460想表达的意义是：将1500字节的MTU去除了20字节的IP头、20字节的TCP头以后，一个最大报文能够承载的有效数据长度。但有些网络中，会在TCP的可选头部里，使用12字节作为时间戳使用，这样，有效数据就是MSS再减去12，初始窗口就是（1460-12）*4=5792，这与窗口想表达的含义是一致的，即：我能够处理的有效数据长度。

在linux3以后的版本中，初始窗口调整到了10个MSS大小，这主要来自于GOOGLE的建议。原因是这样的，接收窗口虽然常以指数方式来快速增加窗口大小（拥塞阀值以下是指数增长的，阀值以上进入拥塞避免阶段则为线性增长，而且，拥塞阀值自身在收到128以上数据报文时也有机会快速增加），若是传输视频这样的大数据，那么随着窗口增加到（接近）最大读缓存后，就会“开足马力”传输数据，但若是通常都是几十KB的网页，那么过小的初始窗口还没有增加到合适的窗口时，连接就结束了。这样相比较大的初始窗口，就使得用户需要更多的时间（RTT）才能传输完数据，体验不好。

那么这时大家可能有疑问，当窗口从初始窗口一路扩张到最大接收窗口时，最大接收窗口就是最大读缓存吗？
不是，因为必须分一部分缓存用于应用程序的延时报文读取。到底会分多少出来呢？这是可配的系统选项，如下：

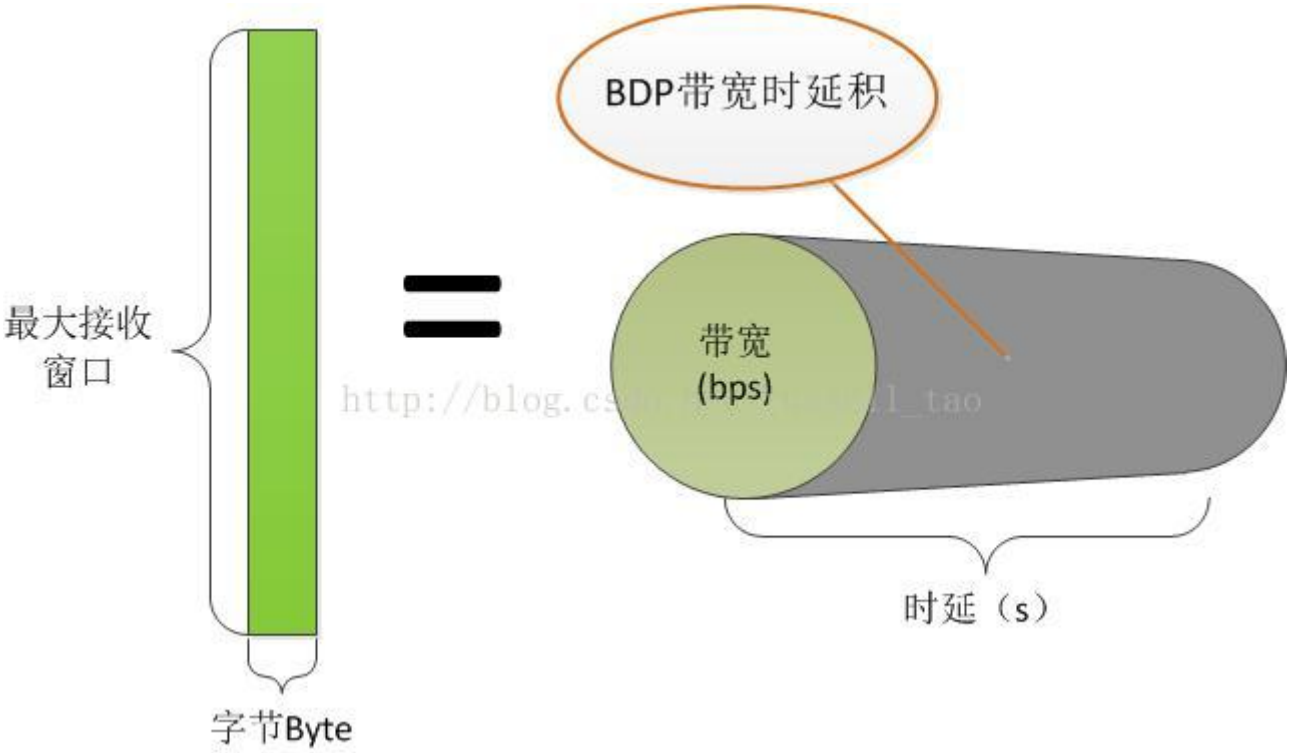
```
net.ipv4.tcp_adv_win_scale = 2
```

这里的tcp_adv_win_scale意味着，将要拿出1/(2^tcp_adv_win_scale)缓存出来做应用缓存。即，默认tcp_adv_win_scale配置为2时，就是拿出至少1/4的内存用于应用读缓存，那么，最大的接收滑动窗口的大小只能到达读缓存的3/4。

（2）最大读缓存到底应该设置到多少为合适呢？

当应用缓存所占的份额通过tcp_adv_win_scale配置确定后，读缓存的上限应当由最大的TCP接收窗口决定。初始窗口可能只有4个或者10个MSS，但在无丢包情形下随着报文的交互窗口就会增大，当窗口过大时，“过大”是什么意思呢？即，对于通讯的两台机器的内存而言不算大，但是对于整个网络负载来说过大了，就会对网络设备引发恶性循环，不断的因为繁忙的网络设备造成丢包。而窗口过小时，就无法充分的利用网络资源。所以，一般会以BDP来设置最大接收窗口（可计算出最大读缓存）。BDP叫做带宽时延积，也就是带宽与网络时延的乘积，例如若我们的带宽为2Gbps，时延为10ms，那么带宽时延积BDP则为2G/8*0.01=2.5MB，所以这样的网络中可以设最大接收窗口为2.5MB，这样最大读缓存可以设为4/3*2.5MB=3.3MB。

为什么呢？因为BDP就表示了网络承载能力，最大接收窗口就表示了网络承载能力内可以不经确认发出的报文。如下图所示：



经常提及的所谓长肥网络，“长”就是是时延长，“肥”就是带宽大，这两者任何一个大时，BDP就大，都应导致最大窗口增大，进而导致读缓存上限增大。所以在长肥网络中的服务器，缓存上限都是比较大的。（当然，TCP原始的16位长度的数字表示窗口虽然有上限，但在RFC1323中定义的弹性滑动窗口使得滑动窗口可以扩展到足够大。）

发送窗口实际上就是TCP连接对方的接收窗口，所以大家可以按接收窗口来推断，这里不再啰嗦。

三、linux的TCP缓存上限自动调整策略

那么，设置好最大缓存限制后就高枕无忧了吗？对于一个TCP连接来说，可能已经充分利用网络资源，使用大窗口、大缓存来保持高速传输了。比如在长肥网络中，缓存上限可能会被设置为几十兆字节，但系统的总内存却是有限的，当每一个连接都全速飞奔使用到最大窗口时，1万个连接就会占用内存到几百G了，这就限制了高并发场景的使用，公平性也得不到保证。我们希望的场景是，在并发连接比较少时，把缓存限制放大一些，让每一个TCP连接开足马力工作；当并发连接很多时，此时系统内存资源不足，那么就把缓存限制缩小一些，使每一个TCP连接的缓存尽量的小一些，以容纳更多的连接。

linux为了实现这种场景，引入了自动调整内存分配的功能，由tcp_moderate_rcvbuf配置决定，如下：
net.ipv4.tcp_moderate_rcvbuf = 1
默认tcp_moderate_rcvbuf配置为1，表示打开了TCP内存自动调整功能。若配置为0，这个功能将不会生效（慎用）。

另外请注意：当我们在编程中对连接设置了SO_SNDBUF、SO_RCVBUF，将会使linux内核不再对这样的连接执行自动调整功能！

那么，这个功能到底是怎样起作用的呢？看以下配置：

```
net.ipv4.tcp_rmem = 8192 87380 16777216
net.ipv4.tcp_wmem = 8192 65536 16777216
net.ipv4.tcp_mem = 8388608 12582912 16777216
```

tcp_rmem[3]数组表示任何一个TCP连接上的读缓存上限，其中tcp_rmem[0]表示最小上限，tcp_rmem[1]表示初始上限（注意，它会覆盖适用于所有协议的rmem_default配置），tcp_rmem[2]表示最大上限。
tcp_wmem[3]数组表示写缓存，与tcp_rmem[3]类似，不再赘述。

tcp_mem[3]数组就用来设定TCP内存的整体使用状况，所以它的值很大（它的单位也不是字节，而是页--4K或者8K等这样的单位！）。这3个值定义了TCP整体内存的无压力值、压力模式开启阀值、最大使用值。以这3个值为标记点则内存共有4种情况：

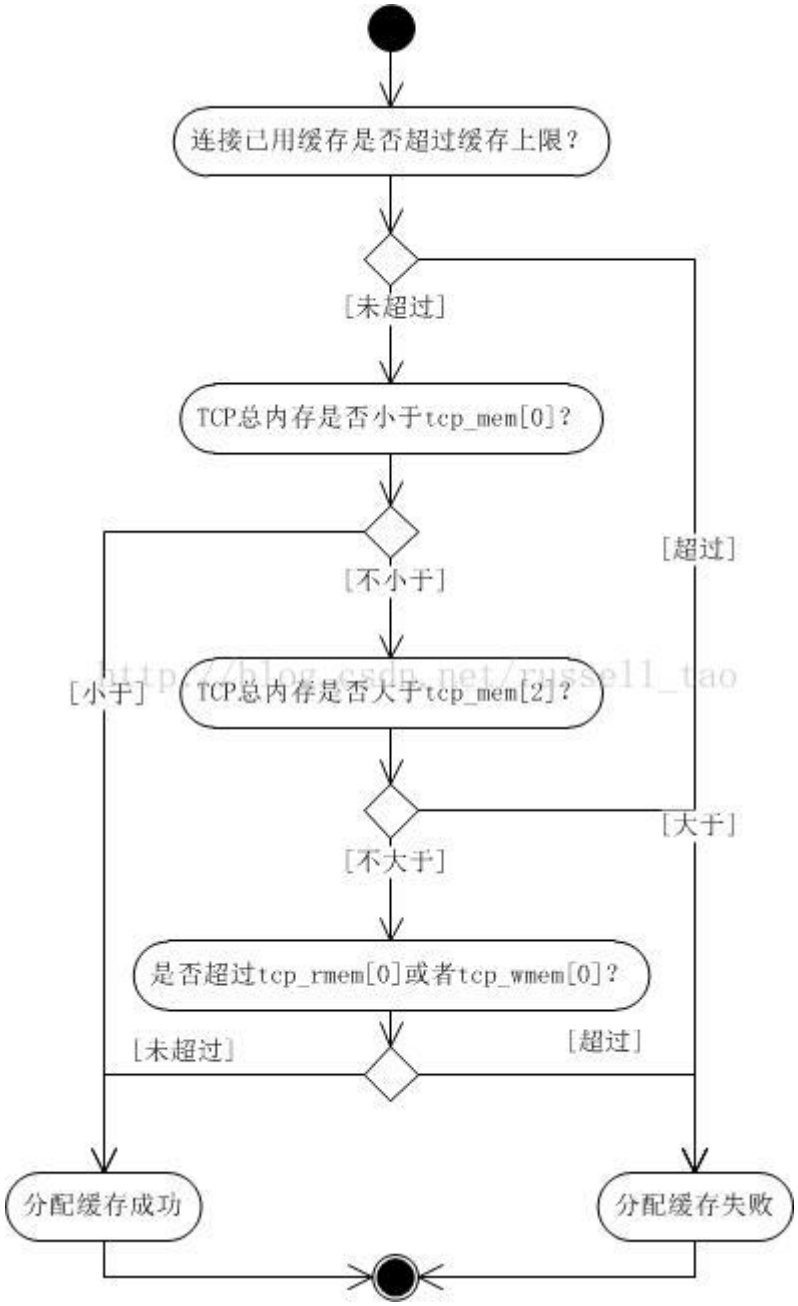
1、当TCP整体内存小于tcp_mem[0]时，表示系统内存总体无压力。若之前内存曾经超过了tcp_mem[1]使系统进入内存压力模式，那么此时也会把压力模式关闭。
这种情况下，只要TCP连接使用的缓存没有达到上限（注意，虽然初始上限是tcp_rmem[1]，但这个值是可变的，下文会详述），那么新内存的分配一定是成功的。

2、当TCP内存在tcp_mem[0]与tcp_mem[1]之间时，系统可能处于内存压力模式，例如总内存刚从tcp_mem[1]之上下来；也可能是在非压力模式下，例如总内存刚从tcp_mem[0]以下上来。
此时，无论是否在压力模式下，只要TCP连接所用缓存未超过tcp_rmem[0]或者tcp_wmem[0]，那么都一定都能成功分配新内存。否则，基本上就会面临分配失败的状况。（注意：还有一些例外场景允许分配内存成功，由于对于我们理解这几个配置项意义不大，故略过。）

3、当TCP内存在tcp_mem[1]与tcp_mem[2]之间时，系统一定处于系统压力模式下。其他行为与上同。

4、当TCP内存在tcp_mem[2]之上时，毫无疑问，系统一定在压力模式下，而且此时所有的新TCP缓存分配都会失败。

下图为需要新缓存时内核的简化逻辑：



当系统在非压力模式下，上面我所说的每个连接的读写缓存上限，才有可能增加，当然最大也不会超过tcp_rmem[2]或者tcp_wmem[2]。相反，在压力模式下，读写缓存上限则有可能减少，虽然上限可能会小于tcp_rmem[0]或者tcp_wmem[0]。

所以，粗略的总结下，对这3个数组可以这么看：
1、只要系统TCP的总体内存超了 tcp_mem[2] ，新内存分配都会失败。
2、tcp_rmem[0]或者tcp_wmem[0]优先级也很高，只要条件1不超限，那么只要连接内存小于这两个值，就保证新内存分配一定成功。
3、只要总体内存不超过tcp_mem[0]，那么新内存在不超过连接缓存的上限时也能保证分配成功。
4、tcp_mem[1]与tcp_mem[0]构成了开启、关闭内存压力模式的开关。在压力模式下，连接缓存上限可能会减少。在非压力模式下，连接缓存上限可能会增加，最多增加到tcp_rmem[2]或者tcp_wmem[2]。