浅谈 TCPIP 网络编程中 socket 的行为

<http://blog.jobbole.com/104695>

我认为，想要熟练掌握Linux下的TCP/IP网络编程，至少有三个层面的知识需要熟悉：

1. TCP/IP协议（如连接的建立和终止、重传和确认、滑动窗口和拥塞控制等等）

2. Socket I/O系统调用（重点如read/write），这是TCP/IP协议在应用层表现出来的行为。

3. 编写Performant, Scalable的服务器程序。包括多线程、IO Multiplexing、非阻塞、异步等各种技术。

关于TCP/IP协议，建议参考Richard Stevens的《TCP/IP Illustrated，vol1》（TCP/IP详解卷1）。

关于第二层面，依然建议Richard Stevens的《Unix network proggramming，vol1》（Unix网络编程卷1），这两本书公认是Unix网络编程的圣经。

至于第三个层面，UNP的书中有所提及，也有著名的C10K问题，业界也有各种各样的框架和解决方案，本人才疏学浅，在这里就不一一敷述。

<http://www.kegel.com/c10k.html>

本文的重点在于第二个层面，主要总结一下Linux下TCP/IP网络编程中的read/write系统调用的行为，知识来源于自己网络编程的粗浅经验和对《Unix网络编程卷1》相关章节的总结。由于本人接触Linux下网络编程时间不长，错误和疏漏再所难免，望看官不吝赐教。

**一. read/write的语义：为什么会阻塞？**

先从write说起：

RO}@@MI~MVUOFA]5][8P4PF

首先，write成功返回，只是buf中的数据被复制到了kernel中的TCP发送缓冲区。至于数据什么时候被发往网络，什么时候被对方主机接收，什么时候被对方进程读取，系统调用层面不会给予任何保证和通知。

write在什么情况下会阻塞？当kernel的该socket的发送缓冲区已满时。对于每个socket，拥有自己的send buffer和receive buffer。从Linux 2.6开始，两个缓冲区大小都由系统来自动调节（autotuning），但一般在default和max之间浮动。



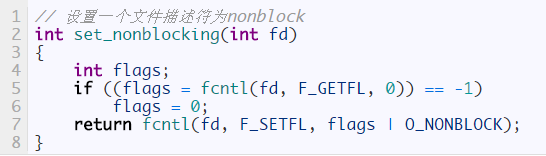
已经发送到网络的数据依然需要暂存在send buffer中，只有收到对方的ack后，kernel才从buffer中清除这一部分数据，为后续发送数据腾出空间。接收端将收到的数据暂存在receive buffer中，自动进行确认。但如果socket所在的进程不及时将数据从receive buffer中取出，最终导致receive buffer填满，由于TCP的滑动窗口和拥塞控制，接收端会阻止发送端向其发送数据。这些控制皆发生在TCP/IP栈中，对应用程序是透明的，应用程序继续发送数据，最终导致send buffer填满，write调用阻塞。

一般来说，由于接收端进程从socket读数据的速度跟不上发送端进程向socket写数据的速度，最终导致发送端write调用阻塞。

而read调用的行为相对容易理解，从socket的receive buffer中拷贝数据到应用程序的buffer中。read调用阻塞，通常是发送端的数据没有到达。

**二. blocking（默认）和nonblock模式下read/write行为的区别：**

将socket fd设置为nonblock（非阻塞）是在服务器编程中常见的做法，采用blocking IO并为每一个client创建一个线程的模式开销巨大且可扩展性不佳（带来大量的切换开销），更为通用的做法是采用线程池+Nonblock I/O+Multiplexing（select/poll，以及Linux上特有的epoll）。



几个重要的结论：

1. read总是在接收缓冲区有数据时立即返回，而不是等到给定的read buffer填满时返回。

只有当receive buffer为空时，blocking模式才会等待，而nonblock模式下会立即返回-1（errno = EAGAIN或EWOULDBLOCK）

2. blocking的write只有在缓冲区足以放下整个buffer时才返回（与blocking read并不相同）

nonblock write则是返回能够放下的字节数，之后调用则返回-1（errno = EAGAIN或EWOULDBLOCK）

对于blocking的write有个特例：当write正阻塞等待时对面关闭了socket，则write则会立即将剩余缓冲区填满并返回所写的字节数，再次调用则write失败（connection reset by peer），这正是下个小节要提到的：

**三. read/write对连接异常的反馈行为：**

对应用程序来说，与另一进程的TCP通信其实是完全异步的过程：

1. 我并不知道对面什么时候、能否收到我的数据

2. 我不知道什么时候能够收到对面的数据

3. 我不知道什么时候通信结束（主动退出或是异常退出、机器故障、网络故障等等）

对于1和2，采用write() -> read() -> write() -> read() ->…的序列，通过blocking read或者nonblock read+轮询的方式，应用程序基于可以保证正确的处理流程。

对于3，kernel将这些事件的“通知”通过read/write的结果返回给应用层。

假设A机器上的一个进程a正在和B机器上的进程b通信：某一时刻a正阻塞在socket的read调用上（或者在nonblock下轮询socket）

当b进程终止时，无论应用程序是否显式关闭了socket（OS会负责在进程结束时关闭所有的文件描述符，对于socket，则会发送一个FIN包到对面）。

”同步通知“：进程a对已经收到FIN的socket调用read，如果已经读完了receive buffer的剩余字节，则会返回EOF:0

”异步通知“：如果进程a正阻塞在read调用上（前面已经提到，此时receive buffer一定为空，因为read在receive buffer有内容时就会返回），则read调用立即返回EOF，进程a被唤醒。

socket在收到FIN后，虽然调用read会返回EOF，但进程a依然可以其调用write，因为根据TCP协议，收到对方的FIN包只意味着对方不会再发送任何消息。 在一个双方正常关闭的流程中，收到FIN包的一端将剩余数据发送给对面（通过一次或多次write），然后关闭socket。

但是事情远远没有想象中简单。优雅地（gracefully)关闭一个TCP连接，不仅仅需要双方的应用程序遵守约定，中间还不能出任何差错。

假如b进程是异常终止的，发送FIN包是OS代劳的，b进程已经不复存在，当机器再次收到该socket的消息时，会回应RST（因为拥有该socket的进程已经终止）。a进程对收到RST的socket调用write时，操作系统会给a进程发送SIGPIPE，默认处理动作是终止进程，知道你的进程为什么毫无征兆地死亡了吧：）

from 《Unix Network programming, vol1》 3rd Edition：

“It is okay to write to a socket that has received a FIN, but it is an error to write to a socket that has received an RST.”

通过以上的叙述，内核通过socket的read/write将双方的连接异常通知到应用层，虽然很不直观，似乎也够用。

这里说一句题外话：

不知道有没有同学会和我有一样的感慨：在写TCP/IP通信时，似乎没怎么考虑连接的终止或错误，只是在read/write错误返回时关闭socket，程序似乎也能正常运行，但某些情况下总是会出奇怪的问题。想完美处理各种错误，却发现怎么也做不对。

原因之一是：socket（或者说TCP/IP栈本身）对错误的反馈能力是有限的。

考虑这样的错误情况：

不同于b进程退出（此时OS会负责为所有打开的socket发送FIN包），当B机器的OS崩溃（注意不同于人为关机，因为关机时所有进程的退出动作依然能够得到保证）/主机断电/网络不可达时，a进程根本不会收到FIN包作为连接终止的提示。

如果a进程阻塞在read上，那么结果只能是永远的等待。

如果a进程先write然后阻塞在read，由于收不到B机器TCP/IP栈的ack，TCP会持续重传12次（时间跨度大约为9分钟），然后在阻塞的read调用上返回错误：ETIMEDOUT/EHOSTUNREACH/ENETUNREACH

假如B机器恰好在某个时候恢复和A机器的通路，并收到a某个重传的pack，因为不能识别所以会返回一个RST，此时a进程上阻塞的read调用会返回错误ECONNREST

恩，socket对这些错误还是有一定的反馈能力的，前提是在对面不可达时你依然做了一次write调用，而不是轮询或是阻塞在read上，那么总是会在重传的周期内检测出错误。如果没有那次write调用，应用层永远不会收到连接错误的通知。

write的错误最终通过read来通知应用层，有点阴差阳错？

**四. 还需要做什么?**

至此，我们知道了仅仅通过read/write来检测异常情况是不靠谱的，还需要一些额外的工作：

1. 使用TCP的KEEPALIVE功能？

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

以上参数的大致意思是：keepalive routine每2小时（7200秒）启动一次，发送第一个probe（探测包），如果在75秒内没有收到对方应答则重发probe，当连续9个probe没有被应答时，认为连接已断。（此时read调用应该能够返回错误，待测试）

但在我印象中keepalive不太好用，默认的时间间隔太长，又是整个TCP/IP栈的全局参数：修改会影响其他进程，Linux的下似乎可以修改per socket的keepalive参数？（希望有使用经验的人能够指点一下），但是这些方法不是portable的。

2. 进行应用层的心跳

严格的网络程序中，应用层的心跳协议是必不可少的。虽然比TCP自带的keep alive要麻烦不少（怎样正确地实现应用层的心跳，我或许会用一篇专门的文章来谈一谈），但有其最大的优点：可控。

当然，也可以简单一点，针对连接做timeout，关闭一段时间没有通信的”空闲“连接。这里可以参考一篇文章：

Muduo 网络编程示例之八：Timing wheel 踢掉空闲连接 by 陈硕

<http://www.cnblogs.com/Solstice/archive/2011/05/04/2036983.html>

参考资料：

《TCP/IP Illustrated, vol 1》 by Richard Stevens

《Unix Network Programming， vol 1》(3rd Edition) by Richard Stevens

Linux TCP tuning

<http://fasterdata.es.net/fasterdata/host-tuning/linux/>

Using TCP keepalive under Linux

<http://tldp.org/HOWTO/TCP-Keepalive-HOWTO/usingkeepalive.html>