高性能网络编程4--TCP连接的关闭

2013年10月26日 12:24:23 阅读数：17242

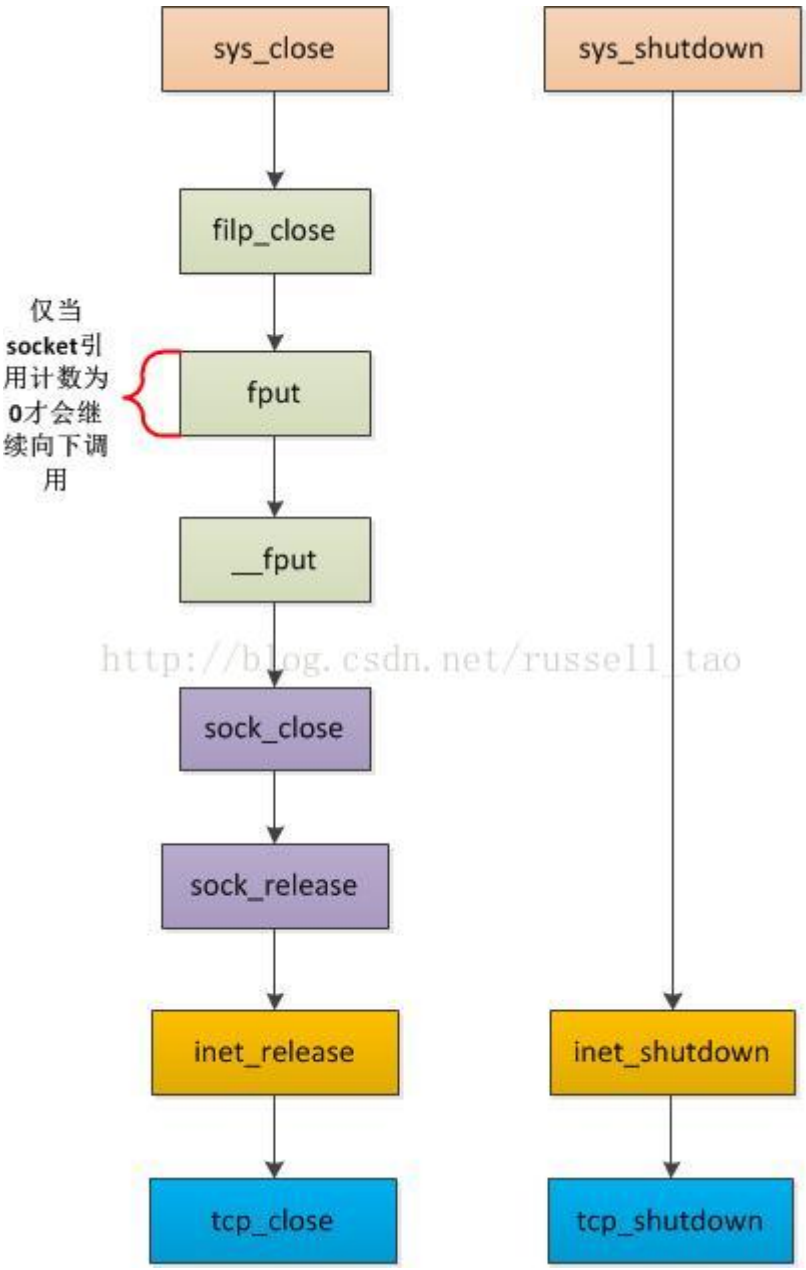
TCP连接的关闭有两个方法close和shutdown，这篇文章将尽量精简的说明它们分别做了些什么。
为方便阅读，我们可以带着以下5个问题来阅读本文：
1、当socket被多进程或者多线程共享时，关闭连接时有何区别？
2、关连接时，若连接上有来自对端的还未处理的消息，会怎么处理？
3、关连接时，若连接上有本进程待发送却未来得及发送出的消息，又会怎么处理？
4、so_linger这个功能的用处在哪？
5、对于监听socket执行关闭，和对处于ESTABLISH这种通讯的socket执行关闭，有何区别？

下面分三部分进行：首先说说多线程多进程关闭连接的区别；再用一幅流程图谈谈close；最后用一幅流程图说说shutdown。

先不提其原理和实现，从多进程、多线程下 close和shutdown方法调用时的区别说起。
看看close与shutdown这两个系统调用对应的内核函数：（参见unistd.h文件）

```
#define __NR_close                               3
__SYSCALL(__NR_close, sys_close)
#define __NR_shutdown                           48
__SYSCALL(__NR_shutdown, sys_shutdown)
```

但sys_close和sys_shutdown这两个系统调用最终是由tcp_close和tcp_shutdown方法来实现的，调用过程如下图所示：



sys_shutdown与多线程和多进程都没有任何关系，而sys_close则不然，上图中可以看到，层层封装调用中有一个方法叫fput，它有一个引用计数，记录这个socket被引用了多少次。在说明多线程或者多进程调用close的区别前，先在代码上简单看下close是怎么调用的，对内核代码没兴趣的同学可以仅看fput方法：

```
void fastcall fput(struct file *file)
{

    if (atomic_dec_and_test(&file->f_count))//检查引用计数，直到为0才会真正去关闭sock
                __fput(file);    }
```

当这个socket的引用计数f_count不为0时，是不会触发到真正关闭TCP连接的tcp_close方法的。
那么，这个引用计数的意义何在呢？为了说明它，先要说道下进程与线程的区别。

大家知道，所谓线程其实就是“轻量级”的进程。创建进程只能是一个进程（父进程）创建另一个进程（子进程），子进程会复制父进程的资源，这里的”复制“针对不同的资源其意义是不同的，例如对内存、文件、TCP连接等。创建进程是由clone系统调用实现的，而创建线程时同样也是clone实现的，只不过clone的参数不同，其行为也很不同。这个话题是很大的，这里我们仅讨论下TCP连接。
在clone系统调用中，会调用方法copy_files来拷贝文件描述符（包括socket）。创建线程时，传入的flag参数中包含标志位CLONE_FILES，此时，线程将会共享父进程中的文件描述符。而创建进程时没有这个标志位，这时，

会把进程打开的所有文件描述符的引用计数加1，即把file数据结构的f_count成员加1，如下：

```
static int copy_files(unsigned long clone_flags, struct task_struct * tsk)
{
        if (clone_flags & CLONE_FILES) {
                goto out;//创建线程
        }
        newf = dup_fd(oldf, &error);
out:
        return error;
}
```

再看看dup_fd方法：

```
static struct files_struct *dup_fd(struct files_struct *oldf, int *errorp)
{
        for (i = open_files; i != 0; i--) {
                struct file *f = *old_fds++;
                if (f) {
                        get_file(f);//创建进程
                }
        }
}
```

get_file宏就会加引用计数。

```
#define get_file(x)     atomic_inc(&(x)->f_count)
```

所以，子进程会将父进程中已经建立的socket加上引用计数。当进程中close一个socket时，只会减少引用计数，仅当引用计数为0时才会触发tcp_close。

到这里，对于第一个问题的close调用自然有了结论：单线程（进程）中使用close与多线程中是一致的，但这两者与多进程的行为并不一致，多进程中共享的同一个socket必须都调用了close才会真正的关闭连接。

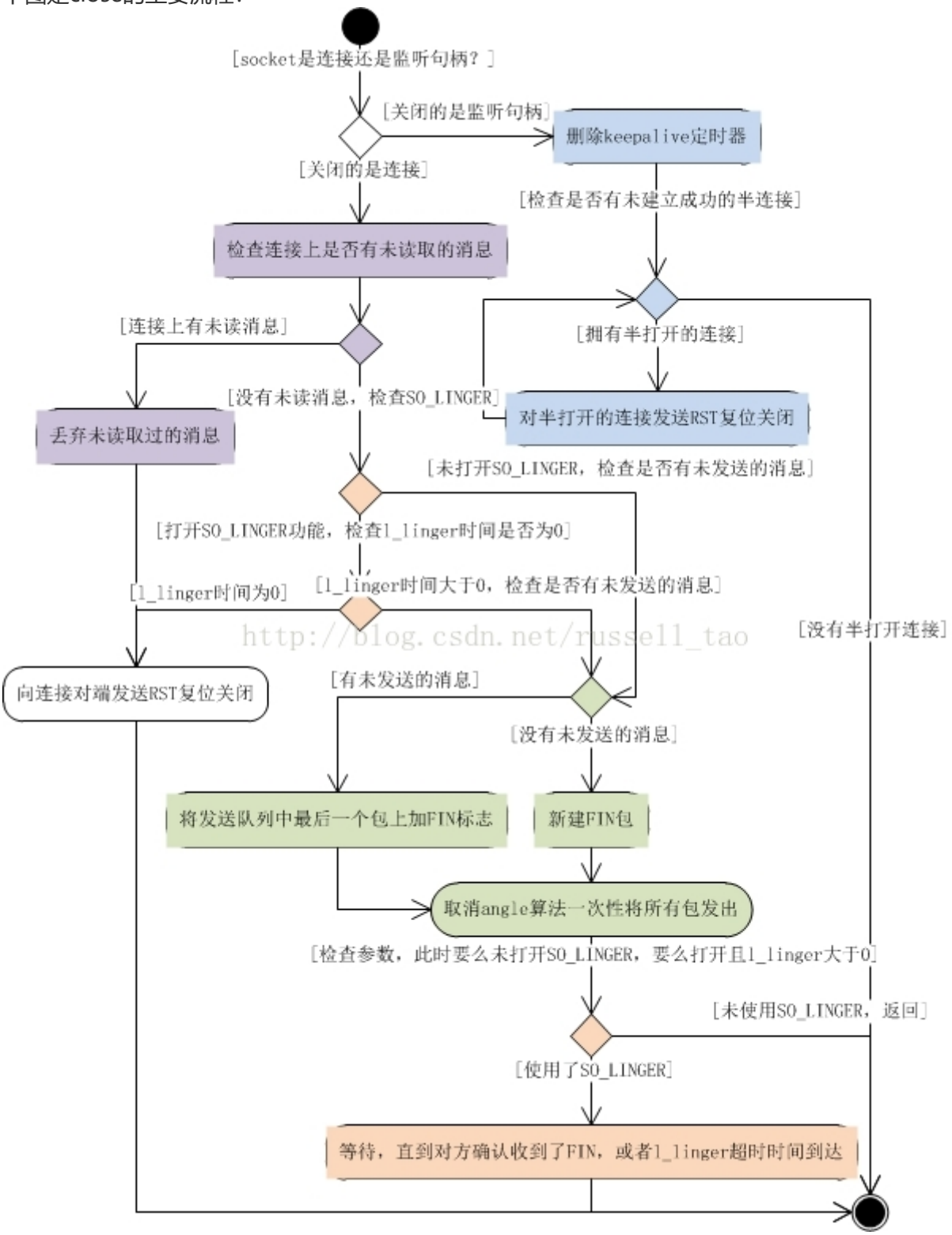
而shutdown则不然，这里是没有引用计数什么事的，只要调用了就会去试图按需关闭连接。所以，调用shutdown与多线程、多进程无关。

下面我们首先深入探讨下close的行为，因为close比较shutdown来说要复杂许多。顺便回答其余四个问题。
TCP连接是一种双工的连接，何谓双工？即连接双方可以并行的发送或者接收消息，而无须顾及对方此时到底在发还是收消息。这样，关闭连接时，就存在3种情形：完全关闭连接；关闭发送消息的功能；关闭接收消息的功能。其中，后两者就叫做半关闭，由shutdown实现（所以 shutdown多出一个参数正是控制关闭发送或者关闭接收），前者由close实现。

TCP连接是一种可靠的连接，在这里可以这么理解：既要确认本机发出的包得到确认，又要确认收到的任何消息都已告知连接的对端。

以下主要从双工、可靠性这两点上理解连接的关闭。

TCP双工的这个特性使得连接的正常关闭需要四次握手，其含义为：主动端关闭了发送的功能；被动端认可；被动端也关闭了发送的功能；主动端认可。
但还存在程序异常的情形，此时，则通过异常的那端发送RST复位报文通知另一端关闭连接。
下图是close的主要流程：



这个图稍复杂，这是因为它覆盖了关闭监听句柄、关闭普通连接、关闭设置了SO_LINGER的连接这三种主要场景。

1）关闭监听句柄
先从最右边的分支说说关闭监听socket的那些事。用于listen的监听句柄也是使用close关闭，关闭这样的句柄含义当然很不同，它本身并不对应着某个TCP连接，但是，附着在它之上的却可能有半成品连接。什么意思呢？之前说过TCP是双工的，它的打开需要三次握手，三次握手也就是3个步骤，其含义为：客户端打开接收、发送的功能；

服务器端认可并也打开接收、发送的功能；客户端认可。当第1、2步骤完成、第3步步骤未完成时，就会在服务器上有许多半连接，close这个操作主要是清理这些连接。
参照上图，close首先会移除keepalive定时器。keepalive功能常用于服务器上，防止僵死、异常退出的客户端占用服务器连接资源。移除此定时器后，若ESTABLISH状态的TCP连接在tcp_keepalive_time时间（如服务器上常配置为2小时）内没有通讯，服务器就会主动关闭连接。
接下来，关闭每一个半连接。如何关闭半连接？这时当然不能发FIN包，即正常的四次握手关闭连接，而是会发送RST复位标志去关闭请求。处理完所有半打开的连接close的任务就基本完成了。

2）关闭普通ESTABLISH状态的连接（未设置so_linger）
首先检查是否有接收到却未处理的消息。
如果close调用时存在收到远端的、没有处理的消息，这时根据close这一行为的意义，是要丢弃这些消息的。但丢弃消息后，意味着连接远端误以为发出的消息已经被本机收到处理了（因为ACK包确认过了），但实际上确是收到未处理，此时也不能使用正常的四次握手关闭，而是会向远端发送一个RST非正常复位关闭连接。这个做法的依据请参考draft-ietf-tcpimpl-prob-03.txt文档3.10节，Failure to RST on close with data pending。所以，这也要求我们程序员在关闭连接时，要确保已经接收、处理了连接上的消息。

如果此时没有未处理的消息，那么进入发送FIN来关闭连接的阶段。
这时，先看看是否有待发送的消息。前一篇已经说过，发消息时要计算滑动窗口、拥塞窗口、angle算法等，这些因素可能导致消息会延迟发送的。如果有待发送的消息，那么要尽力保证这些消息都发出去的。所以，会在最后一个报文中加入FIN标志，同时，关闭用于减少网络中小报文的angle算法，向连接对端发送消息。如果没有待发送的消息，则构造一个报文，仅含有FIN标志位，发送出去关闭连接。

3）使用了so_linger的连接
首先要澄清，为何要有so_linger这个功能？因为我们可能有强可靠性的需求，也就是说，必须确保发出的消息、FIN都被对方收到。例如，有些响应发出后调用close关闭连接，接下来就会关闭进程。如果close时发出的消息其实丢失在网络中了，那么，进程突然退出时连接上发出的RST就可能被对方收到，而且，之前丢失的消息不会有重发来保障可靠性了。
so_linger用来保证对方收到了close时发出的消息，即，至少需要对方通过发送ACK且到达本机。
怎么保证呢？等待！close会阻塞住进程，直到确认对方收到了消息再返回。然而，网络环境又得复杂的，如果对方总是不响应怎么办？所以还需要l_linger这个超时时间，控制close阻塞进程的最长时间。注意，务必慎用so_linger，它会在不经意间降低你程序中代码的执行速度（close的阻塞）。

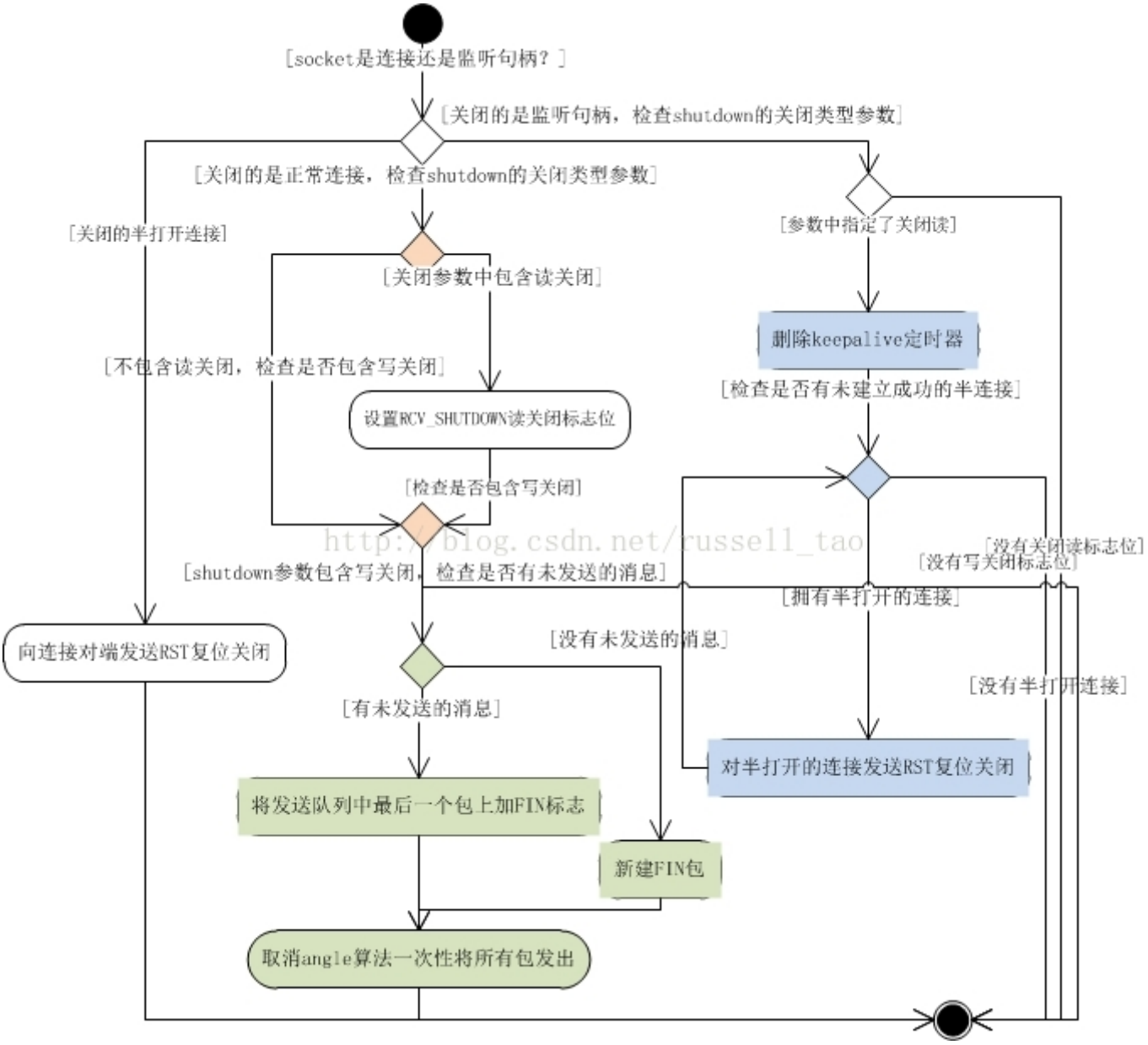
所以，当这个进程设置了so_linger后，前半段依然没变化。检查是否有未读消息，若有则发RST关连接，不会触发等待。接下来检查是否有未发送的消息时与第2种情形一致，设好FIN后关闭angle算法发出。接下来，则会设置最大等待时间l_linger，然后开始将进程睡眠，直到确认对方收到后才会醒来，将控制权交还给用户进程。

这里需要注意，so_linger不是确保连接被四次握手关闭再使close返回，而只是保证我方发出的消息都已被对方收到。例如，若对方程序写的有问题，当它收到FIN进入CLOSE_WAIT状态，却一直不调用close发出FIN，此时，对方仍然会通过ACK确认，我方收到了ACK进入FIN_WAIT2状态，但没收到对方的FIN，我方的close调用却不会再阻塞，close直接返回，控制权交还用户进程。

从上图可知，so_linger还有个偏门的用法，若l_linger超时时间竟被设为0，则不会触发FIN包的发送，而是直接RST复位关闭连接。我个人认为，这种玩法确没多大用处。

最后做个总结。调用close时，可能导致发送RST复位关闭连接，例如有未读消息、打开so_linger但l_linger却为0、关闭监听句柄时半打开的连接。更多时会导致发FIN来四次握手关闭连接，但打开so_linger可能导致close阻塞住等待着对方的ACK表明收到了消息。

最后来看看较为简单的shutdown。



解释下上图：

1）shutdown可携带一个参数，取值有3个，分别意味着：只关闭读、只关闭写、同时关闭读写。

对于监听句柄，如果参数为关闭写，显然没有任何意义。但关闭读从某方面来说是有意义的，例如不再接受新的连接。看看最右边蓝色分支，针对监听句柄，若参数为关闭写，则不做任何事；若为关闭读，则把端口上的半打开连接使用RST关闭，与close如出一辙。

2）若shutdown的是半打开的连接，则发出RST来关闭连接。

3）若shutdown的是正常连接，那么关闭读其实与对端是没有关系的。只要本机把接收掉的消息丢掉，其实就等价于关闭读了，并不一定非要对端关闭写的。实际上，shutdown正是这么干的。若参数中的标志位含有关闭读，只是标识下，当我们调用read等方法时这个标识就起作用了，会使进程读不到任何数据。

4）若参数中有标志位为关闭写，那么下面做的事与close是一致的：发出FIN包，告诉对方，本机不会再发消息了。

以上，就是close与shutdown的主要行为，同时也回答了本文最初的5个问题。下一篇，我们开始讨论多路复用中常见的epoll。