

11 | 优雅地关闭还是粗暴地关闭？

2019-08-26 盛延敏

网络编程实战

[进入课程 >](#)



讲述：冯永吉

时长 12:18 大小 16.91M



你好，我是盛延敏，这里是网络编程实战第 11 讲，欢迎回来。

上一讲我们讲到了 TCP 的四次挥手，其中发起连接关闭的一方会有一段时间处于 TIME_WAIT 状态。那么究竟如何来发起连接关闭呢？这一讲我们就来讨论一下。

我们知道，一个 TCP 连接需要经过三次握手进入数据传输阶段，最后来到连接关闭阶段。在最后的连接关闭阶段，我们需要重点关注的是“半连接”状态。

因为 TCP 是双向的，这里说的方向，指的是数据流的写入 - 读出的方向。

比如客户端到服务器端的方向，指的是客户端通过套接字接口，向服务器端发送 TCP 报文；而服务器端到客户端方向则是另一个传输方向。在绝大多数情况下，TCP 连接都是先关

闭一个方向，此时另外一个方向还是可以正常进行数据传输。


举个例子，客户端主动发起连接的中断，将自己到服务器端的数据流方向关闭，此时，客户端不再往服务器端写入数据，服务器端读完客户端数据后就不会再有新的报文到达。但这并不意味着，TCP 连接已经完全关闭，很有可能的是，服务器端正在对客户端的最后报文进行处理，比如去访问数据库，存入一些数据；或者是计算出某个客户端需要的值，当完成这些操作之后，服务器端把结果通过套接字写给客户端，我们说这个套接字的状态此时是“半关闭”的。最后，服务器端才有条不紊地关闭剩下的半个连接，结束这一段 TCP 连接的使命。

当然，我这里描述的，是服务器端“优雅”地关闭了连接。如果服务器端处理不好，就会导致最后的关闭过程是“粗暴”的，达不到我们上面描述的“优雅”关闭的目标，形成的后果，很可能是服务器端处理完的信息没办法正常传送给客户端，破坏了用户侧的使用场景。

接下来我们就来看看关闭连接时，都有哪些方式呢？

close 函数

首先，我们来看最常见的 close 函数：

 复制代码

```
1 int close(int sockfd)
```

这个函数很简单，对已连接的套接字执行 close 操作就可以，若成功则为 0，若出错则为 -1。

这个函数会对套接字引用计数减一，一旦发现套接字引用计数到 0，就会对套接字进行彻底释放，并且会关闭**TCP 两个方向的数据流**。

套接字引用计数是什么意思呢？因为套接字可以被多个进程共享，你可以理解为我们给每个套接字都设置了一个积分，如果我们通过 fork 的方式产生子进程，套接字就会积分 +1，如果我们调用一次 close 函数，套接字积分就会 -1。这就是套接字引用计数的含义。

close 函数具体是如何关闭两个方向的数据流呢？

在输入方向，系统内核会将该套接字设置为不可读，任何读操作都会返回异常。

在输出方向，系统内核尝试将发送缓冲区的数据发送给对端，并最后向对端发送一个 FIN 报文，接下来如果再对该套接字进行写操作会返回异常。


如果对端没有检测到套接字已关闭，还继续发送报文，就会收到一个 RST 报文，告诉对端：“Hi, 我已经关闭了，别再给我发数据了。”

我们会发现，close 函数并不能帮助我们关闭连接的一个方向，那么如何在需要的时候关闭一个方向呢？幸运的是，设计 TCP 协议的人帮我们想好了解决方案，这就是 shutdown 函数。

shutdown 函数

shutdown 函数的原型是这样的：

```
1 int shutdown(int sockfd, int howto)
```

 复制代码

对已连接的套接字执行 shutdown 操作，若成功则为 0，若出错则为 -1。

howto 是这个函数的设置选项，它的设置有三个主要选项：

SHUT_RD(0)：关闭连接的“读”这个方向，对该套接字进行读操作直接返回 EOF。从数据角度来看，套接字上接收缓冲区已有的数据将被丢弃，如果再有新的数据流到达，会对数据进行 ACK，然后悄悄地丢弃。也就是说，对端还是会接收到 ACK，在这种情况下根本不知道数据已经被丢弃了。

SHUT_WR(1)：关闭连接的“写”这个方向，这就是常被称为“半关闭”的连接。此时，不管套接字引用计数的值是多少，都会直接关闭连接的写方向。套接字上发送缓冲区已有的数据将被立即发送出去，并发送一个 FIN 报文给对端。应用程序如果对该套接字进行写操作会报错。

SHUT_RDWR(2)：相当于 SHUT_RD 和 SHUT_WR 操作各一次，关闭套接字的读和写两个方向。

讲到这里，不知道你是不是有和我当初一样的困惑，使用 SHUT_RDWR 来调用 shutdown 不是和 close 基本一样吗，都是关闭连接的读和写两个方向。

其实，这两个还是有差别的。

第一个差别：close 会关闭连接，并释放所有连接对应的资源，而 shutdown 并不会释放掉套接字和所有的资源。

第二个差别：close 存在引用计数的概念，并不一定导致该套接字不可用；shutdown 则不管引用计数，直接使得该套接字不可用，如果有别的进程企图使用该套接字，将会受到影响。

第三个差别：close 的引用计数导致不一定会发出 FIN 结束报文，而 shutdown 则总是会发出 FIN 结束报文，这在我们打算关闭连接通知对端的时候，是非常重要的。


体会 close 和 shutdown 的差别

下面，我们通过构建一组客户端和服务端程序，来进行 close 和 shutdown 的实验。

客户端程序，从标准输入不断接收用户输入，把输入的字符串通过套接字发送给服务器端，同时，将服务器端的应答显示到标准输出上。

如果用户输入了“close”，则会调用 close 函数关闭连接，休眠一段时间，等待服务器端处理后退；如果用户输入了“shutdown”，调用 shutdown 函数关闭连接的写方向，注意我们不会直接退出，而是会继续等待服务器端的应答，直到服务器端完成自己的操作，在另一个方向上完成关闭。

在这里，我们会第一次接触到 select 多路复用，这里不展开讲，你只需要记住，使用 select 使得我们可以同时完成对连接套接字和标准输入两个 I/O 对象的处理。

 复制代码

```
1 # include "lib/common.h"
2 # define    MAXLINE    4096
3
4 int main(int argc, char **argv) {
5     if (argc != 2) {
6         error(1, 0, "usage: graceclient <IPaddress>");
7     }
```



```

60         }
61         sleep(6);
62         exit(0);
63     } else {
64         int i = strlen(send_line);
65         if (send_line[i - 1] == '\n') {
66             send_line[i - 1] = 0;
67         }
68
69         printf("now sending %s\n", send_line);
70         size_t rt = write(socket_fd, send_line, strlen(send_line));
71         if (rt < 0) {
72             error(1, errno, "write failed ");
73         }
74         printf("send bytes: %zu \n", rt);
75     }
76
77     }
78 }
79
80 }
81
82 }

```

我对这个程序的细节展开解释一下：

第一部分是套接字的创建和 select 初始工作：

9-10 行创建了一个 TCP 套接字；

12-16 行设置了连接的目标服务器 IPv4 地址，绑定到了指定的 IP 和端口；

18-22 行使用创建的套接字，向目标 IPv4 地址发起连接请求；

30-32 行为使用 select 做准备，初始化描述字集合，这部分我会在后面详细解释，这里就不再深入。


第二部分是程序的主体部分，从 33-80 行，使用 select 多路复用观测在连接套接字和标准输入上的 I/O 事件，其中：

38-48 行：当连接套接字上有数据可读，将数据读入到程序缓冲区中。40-41 行，如果有异常则报错退出；42-43 行如果读到服务器端发送的 EOF 则正常退出。

49-77 行：当标准输入上有数据可读，读入后进行判断。如果输入的是“shutdown”，则关闭标准输入的 I/O 事件感知，并调用 shutdown 函数关闭写方向；如果输入的是“close”，则调用 close 函数关闭连接；64-74 行处理正常的输入，将回车符截掉，调用 write 函数，通过套接字将数据发送给服务器端。

服务器端程序稍微简单一点，连接建立之后，打印出接收的字节，并重新格式化后，发送给客户端。

服务器端程序有一点需要注意，那就是对 SIGPIPE 这个信号的处理。后面我会结合程序的结果展开说明。

 复制代码

```
1 #include "lib/common.h"
2
3 static int count;
4
5 static void sig_int(int signo) {
6     printf("\nreceived %d datagrams\n", count);
7     exit(0);
8 }
9
10 int main(int argc, char **argv) {
11     int listenfd;
12     listenfd = socket(AF_INET, SOCK_STREAM, 0);
13
14     struct sockaddr_in server_addr;
15     bzero(&server_addr, sizeof(server_addr));
16     server_addr.sin_family = AF_INET;
17     server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
18     server_addr.sin_port = htons(SERV_PORT);
19
20     int rt1 = bind(listenfd, (struct sockaddr *) &server_addr, sizeof(server_addr));
21     if (rt1 < 0) {
22         error(1, errno, "bind failed ");
23     }
24
25     int rt2 = listen(listenfd, LISTENQ);
26     if (rt2 < 0) {
27         error(1, errno, "listen failed ");
28     }
29
30     signal(SIGINT, sig_int);
31     signal(SIGPIPE, SIG_IGN);
32
33     int connfd;
34     struct sockaddr_in client_addr;
```

```

35     socklen_t client_len = sizeof(client_addr);
36
37     if ((connfd = accept(listenfd, (struct sockaddr *) &client_addr, &client_len)) < 0)
38         error(1, errno, "bind failed ");
39 }
40
41 char message[MAXLINE];
42 count = 0;
43
44 for (;;) {
45     int n = read(connfd, message, MAXLINE);
46     if (n < 0) {
47         error(1, errno, "error read");
48     } else if (n == 0) {
49         error(1, 0, "client closed \n");
50     }
51     message[n] = 0;
52     printf("received %d bytes: %s\n", n, message);
53     count++;
54
55     char send_line[MAXLINE];
56     sprintf(send_line, "Hi, %s", message);
57
58     sleep(5);
59
60     int write_nc = send(connfd, send_line, strlen(send_line), 0);
61     printf("send bytes: %zu \n", write_nc);
62     if (write_nc < 0) {
63         error(1, errno, "error write");
64     }
65 }
66 }

```

服务器端程序的细节也展开解释一下：

第一部分是套接字和连接创建过程：

11-12 行创建了一个 TCP 套接字；


14-18 行设置了本地服务器 IPv4 地址，绑定到了 ANY 地址和指定的端口；

20-40 行使用创建的套接字，以此执行 bind、listen 和 accept 操作，完成连接建立。


第二部分是程序的主体，通过 read 函数获取客户端传送来的数据流，并回送给客户端：

51-52 行显示收到的字符串，在 56 行对原字符串进行重新格式化，之后调用 send 函数将数据发送给客户端。注意，在发送之前，让服务器端程序休眠了 5 秒，以模拟服务器端处理的时间。

我们启动服务器，再启动客户端，依次在标准输入上输入 data1、data2 和 close，观察一段时间后我们看到：

 复制代码

```
1 $./graceclient 127.0.0.1
2 data1
3 now sending data1
4 send bytes:5
5 data2
6 now sending data2
7 send bytes:5
8 Hi,data1
9 close
```

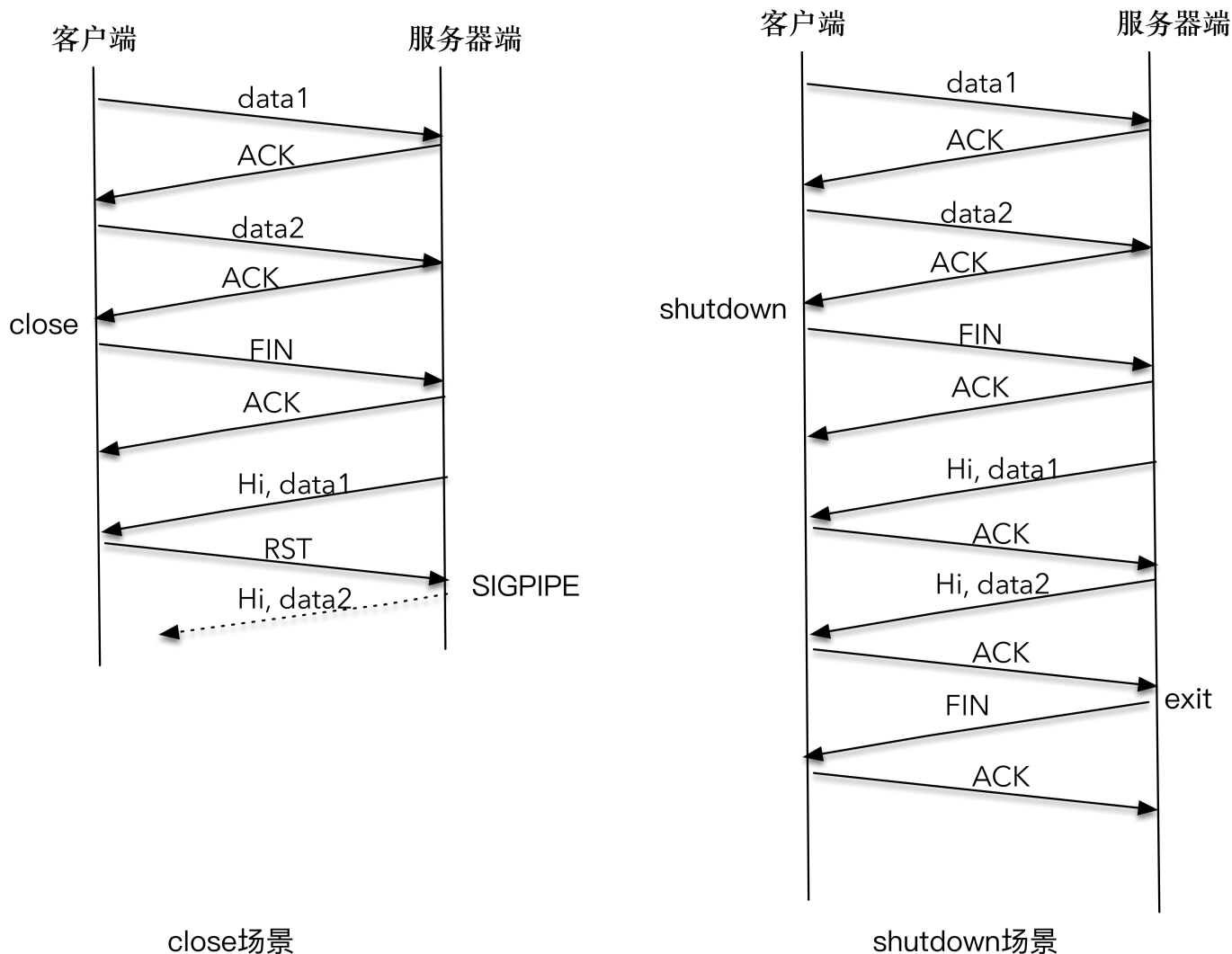
 复制代码

```
1 $./graceserver
2 received 5 bytes: data1
3 send bytes: 9
4 received 5 bytes: data2
5 send bytes: 9
6 client closed
```

客户端依次发送了 data1 和 data2，服务器端也正常接收到 data1 和 data2。在客户端 close 掉整个连接之后，服务器端接收到 SIGPIPE 信号，直接退出。客户端并没有收到服务器端的应答数据。

我在下面放了一张图，这张图详细解释了客户端和服务端交互的时序图。因为客户端调用 close 函数关闭了整个连接，当服务器端发送的“Hi, data1”分组到底时，客户端给回送一个 RST 分组；服务器端再次尝试发送“Hi, data2”第二个应答分组时，系统内核通知 SIGPIPE 信号。这是因为，在 RST 的套接字进行写操作，会直接触发 SIGPIPE 信号。

这回知道你的程序莫名奇妙终止的原因了吧。



我们可以像这样注册一个信号处理函数，对 SIGPIPE 信号进行处理，避免程序莫名退出：

复制代码


```
1 static void sig_pipe(int signo) {
2     printf("\nreceived %d datagrams\n", count);
3     exit(0);
4 }
5 signal(SIGINT, sig_pipe);
```

接下来，再次启动服务器，再启动客户端，依次在标准输入上输入 data1、data2 和 shutdown 函数，观察一段时间后我们看到：

复制代码

```
1 $./graceclient 127.0.0.1
2 data1
3 now sending data1
4 send bytes:5
```

```
5 data2
6 now sending data2
7 send bytes:5
8 shutdown
9 Hi, data1
10 Hi, data2
11 server terminated
```

 复制代码

```
1 $./graceserver
2 received 5 bytes: data1
3 send bytes: 9
4 received 5 bytes: data2
5 send bytes: 9
6 client closed
```

和前面的结果不同，服务器端输出了 data1、data2；客户端也输出了 “Hi,data1” 和 “Hi,data2”，客户端和服务端各自完成了自己的工作后，正常退出。

我们再看下客户端和服务端交互的时序图。因为客户端调用 shutdown 函数只是关闭连接的一个方向，服务器端到客户端的这个方向还可以继续进行数据的发送和接收，所以 “Hi,data1” 和 “Hi,data2” 都可以正常传送；当服务器端读到 EOF 时，立即向客户端发送了 FIN 报文，客户端在 read 函数中感知了 EOF，也进行了正常退出。

总结

在这一讲中，我们讲述了 close 函数关闭连接的方法，使用 close 函数关闭连接有两个需要明确的地方。

close 函数只是把套接字引用计数减 1，未必会立即关闭连接；

close 函数如果在套接字引用计数达到 0 时，立即终止读和写两个方向的数据传送。

基于这两个确定，在期望关闭连接其中一个方向时，应该使用 shutdown 函数。

思考题

和往常一样，给大家留两道思考题。

第一道题，你可以看到在今天的服务器端程序中，直接调用`exit(0)`完成了 FIN 报文的发送，这是为什么呢？为什么不调用 `close` 函数或 `shutdown` 函数呢？

第二道题关于关于信号量处理，今天的程序中，使用的是`SIG_IGN`默认处理，你知道默认处理和自定义函数处理的区别吗？不妨查查资料，了解一下。

欢迎你在评论区写下你的思考，也欢迎把这篇文章分享给你的朋友或者同事，一起来交流。



网络编程实战

从底层到实战，深度解析网络编程

盛延敏

前大众点评云平台首席架构师



新版升级：点击「👤请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 10 | TIME_WAIT：隐藏在细节下的魔鬼

精选留言 (4)

写留言



胡波 allenhu

2019-08-26

老师你好, 上面的例子中有两个地方不是很清楚:

(1) close: client调用close后, 不是会马上关闭两个方向的连接吗, 那为啥"Hi, data1"还能被收到并显示?

(2) shutdown: client调用shutdown只是关闭写方向的连接, 是不是client不能发了只能收? 那为啥server能读到EOF并发送"FIN"给client?

展开 ▾



许童童

2019-08-26

思考题

1.应该是exit后, 操作系统内核协议栈会接管后续的处理。

2.默认处理会exit非0, 也就是错误, 在标准错误上输出信息。自定义函数处理则可以根据自己的需要自行处理。

展开 ▾



许童童

2019-08-26

老师你好, 请问在浏览器中, TCP关闭是由哪一端发起的?



程序水果宝

2019-08-26

signal (SIGINT , sig_pipe) 是不是应该改为signal (SIGPIPE , sig_pipe) , 处理的应该是

