

05 | 使用套接字进行读写：开始交流吧

2019-08-12 盛延敏

网络编程实战

[进入课程 >](#)



讲述：冯永吉

时长 10:18 大小 9.44M



你好，我是盛延敏，这里是网络编程实战第 5 讲，欢迎回来。


在前面的章节中，我们讲述了套接字相关的知识，包括套接字的格式，套接字的创建以及 TCP 连接的建立等。在这一讲里，我来讲一下如何使用创建的套接字收发数据。

连接建立的根本目的是为了数据的收发。拿我们常用的网购场景举例子，我们在浏览商品或者购买货品的时候，并不会察觉到网络连接的存在，但是我们可以真切感觉到数据在客户端和服务端有效的传送，比如浏览商品时商品信息的不断刷新，购买货品时显示购买成功的消息等。

首先我们先来看一下发送数据。

发送数据

发送数据时常用的有三个函数，分别是 `write`、`send` 和 `sendmsg`。

 复制代码

```
1 ssize_t write (int sockfd, const void *buffer, size_t size)
2 ssize_t send (int sockfd, const void *buffer, size_t size, int flags)
3 ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags)
```

每个函数都是单独使用的，使用的场景略有不同：

第一个函数是常见的文件写函数，如果把 `sockfd` 换成文件描述符，就是普通的文件写入。

如果想指定选项，发送带外数据，就需要使用第二个带 `flag` 的函数。所谓带外数据，是一种基于 TCP 协议的紧急数据，用于客户端 - 服务器在特定场景下的紧急处理。

如果想指定多重缓冲区传输数据，就需要使用第三个函数，以结构体 `msghdr` 的方式发送数据。

你看到这里可能会问，既然套接字描述符是一种特殊的描述符，那么在套接字描述符上调用 `write` 函数，应该和在普通文件描述符上调用 `write` 函数的行为是一致的，都是通过描述符句柄写入指定的数据。

乍一看，两者的表现形式是一样，内在的区别还是很不一样的。

对于普通文件描述符而言，一个文件描述符代表了打开的一个文件句柄，通过调用 `write` 函数，操作系统内核帮我们不断地往文件系统中写入字节流。注意，写入的字节流大小通常和输入参数 `size` 的值是相同的，否则表示出错。

对于套接字描述符而言，它代表了一个双向连接，在套接字描述符上调用 `write` 写入的字节数**有可能**比请求的数量少，这在普通文件描述符情况下是不正常的。

产生这个现象的原因在于操作系统内核为读取和发送数据做了很多我们表面上看不到的工作。接下来我拿 `write` 函数举例，重点阐述发送缓冲区的概念。

发送缓冲区

你一定要建立一个概念，当 TCP 三次握手成功，TCP 连接成功建立后，操作系统内核会为每一个连接创建配套的基础设施，比如**发送缓冲区**。

发送缓冲区的大小可以通过套接字选项来改变，当我们的应用程序调用 write 函数时，实际所做的事情是把数据**从应用程序中拷贝到操作系统内核的发送缓冲区中**，并不一定是把数据通过套接字写出去。

这里有几种情况：

第一种情况很简单，操作系统内核的发送缓冲区足够大，可以直接容纳这份数据，那么皆大欢喜，我们的程序从 write 调用中退出，返回写入的字节数就是应用程序的数据大小。

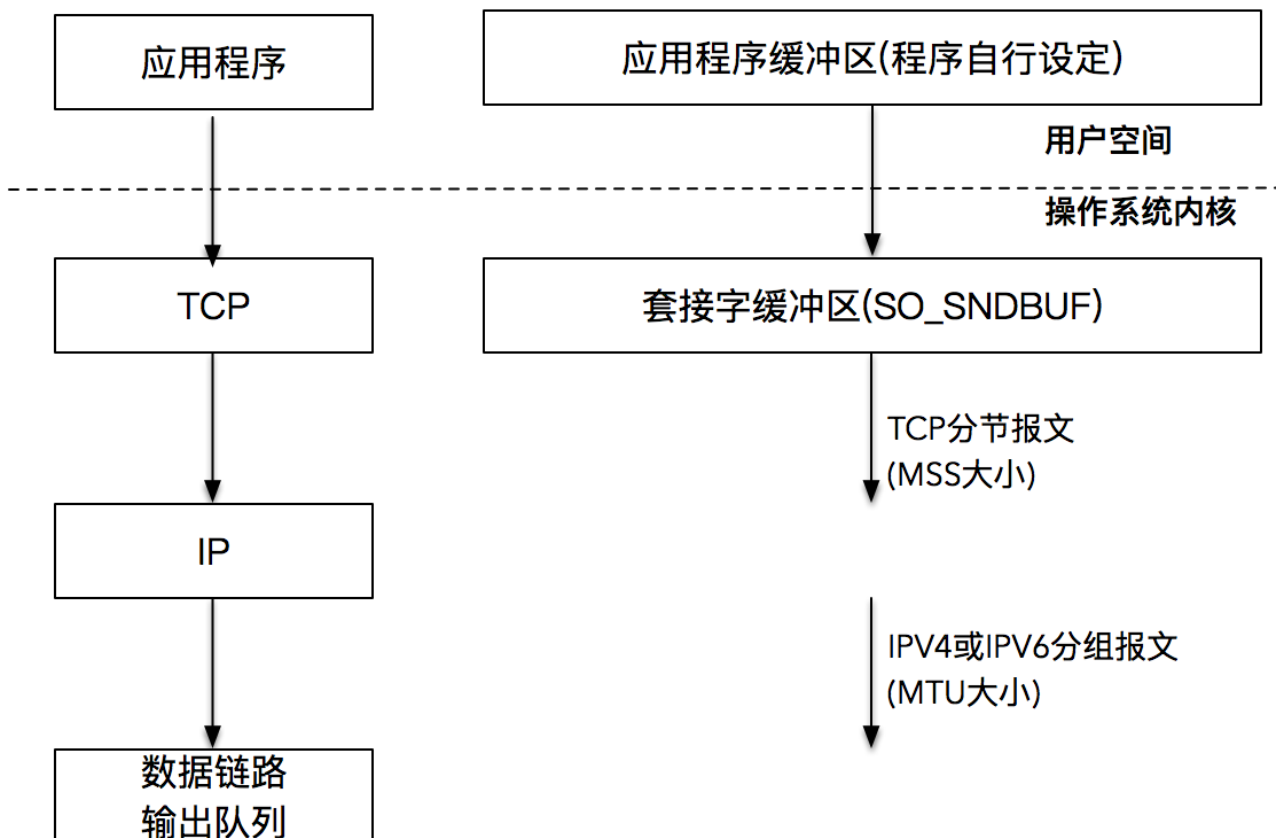
第二种情况是，操作系统内核的发送缓冲区是够大了，不过还有数据没有发送完，或者数据发送完了，但是操作系统内核的发送缓冲区不足以容纳应用程序数据，在这种情况下，你预料的结果是什么呢？报错？还是直接返回？

操作系统内核并不会返回，也不会报错，而是应用程序被阻塞，也就是说应用程序在 write 函数调用处停留，不直接返回。术语“挂起”也表达了相同的意思，不过“挂起”是从操作系统内核角度来说的。

那么什么时候才会返回呢？

实际上，每个操作系统内核的处理是不同的。大部分 UNIX 系统的做法是一直等到可以把应用程序数据完全放到操作系统内核的发送缓冲区中，再从系统调用中返回。怎么理解呢？

别忘了，我们的操作系统内核是很聪明的，当 TCP 连接建立之后，它就开始运作起来。你可以把发送缓冲区想象成一条包裹流水线，有个聪明且忙碌的工人不断地从流水线上取出包裹（数据），这个工人会按照 TCP/IP 的语义，将取出的包裹（数据）封装成 TCP 的 MSS 包，以及 IP 的 MTU 包，最后走数据链路层将数据发送出去。这样我们的发送缓冲区就又空了一部分，于是又可以继续从应用程序搬一部分数据到发送缓冲区里，这样一直进行下去，到某一个时刻，应用程序的数据可以完全放置到发送缓冲区里。在这个时候，write 阻塞调用返回。注意返回的时刻，应用程序数据并没有全部被发送出去，发送缓冲区里还有部分数据，这部分数据会在稍后由操作系统内核通过网络发送出去。




读取数据

我们可以注意到，套接字描述本身和本地文件描述符并无区别，在 **UNIX 的世界里万物都是文件**，这就意味着可以将套接字描述符传递给那些原先为处理本地文件而设计的函数。这些函数包括 `read` 和 `write` 交换数据的函数。

read 函数


让我们先从最简单的 `read` 函数开始看起，这个函数的原型如下：

 复制代码

```
1 ssize_t read (int sockfd, void *buffer, size_t size)
```

`read` 函数要求操作系统内核从套接字描述符 `sockfd` 读取最多多少个字节（`size`），并将结果存储到 `buffer` 中。返回值告诉我们实际读取的字节数目，也有一些特殊情况，如果返回值为 0，表示 EOF（end-of-file），这在网络中表示对端发送了 FIN 包，要处理断连的情况；如果返回值为 -1，表示出错。当然，如果是非阻塞 I/O，情况会略有不同，在后面的提高篇中我们会重点讲述非阻塞 I/O 的特点。

注意这里是最多读取 size 个字节。如果我们想让应用程序每次都读到 size 个字节，就需要编写下面的函数，不断地循环读取。

 复制代码

```
1  /* 从 sockfd 描述符中读取 "size" 个字节。 */
2  ssize_t readn(int fd, void *vptr, size_t size)
3  {
4      size_t  nleft;
5      ssize_t nread;
6      char    *ptr;
7
8      ptr = vptr;
9      nleft = size;
10
11     while (nleft > 0) {
12         if ( (nread = read(fd, ptr, nleft)) < 0) {
13             if (errno == EINTR)
14                 nread = 0;      /* 这里需要再次调用 read */
15             else
16                 return(-1);
17         } else if (nread == 0)
18             break;             /* EOF(End of File) 表示套接字关闭 */
19
20
21         nleft -= nread;
22         ptr   += nread;
23     }
24     return(n - nleft);         /* 返回的是实际读取的字节数 */
25 }
```

对这个程序稍微解释下：

11-25 行的循环条件表示的是，在没读满 size 个字节之前，一直都要循环下去。

13-14 行表示的是非阻塞 I/O 的情况下，没有数据可以读，需要继续调用 read。

17-18 行表示读到对方发出的 FIN 包，表现形式是 EOF，此时需要关闭套接字。

21-22 行，需要读取的字符数减少，缓存指针往下移动。


24 行是在读取 EOF 跳出循环后，返回实际读取的字符数。

缓冲区实验

我们用一个客户端 - 服务器的例子来解释一下读取缓冲区和发送缓冲区的概念。在这个例子中客户端不断地发送数据，服务器端每读取一段数据之后进行休眠，以模拟实际业务处理所需要的时间。

服务器端读取数据程序

下面是服务器端读取数据的程序：

 复制代码

```
1 int main(int argc, char **argv)
2 {
3     int                listenfd, connfd;
4     socklen_t          clilen;
5     struct sockaddr_in cliaddr, servaddr;
6
7     listenfd = socket(AF_INET, SOCK_STREAM, 0);
8
9     bzero(&servaddr, sizeof(servaddr));
10    servaddr.sin_family      = AF_INET;
11    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
12    servaddr.sin_port        = htons(12345);
13
14    /* bind 到本地地址，端口为 12345 */
15    bind(listenfd, (SA *) &servaddr, sizeof(servaddr));
16    /* listen 的 backlog 为 1024 */
17    listen(listenfd, 1024);
18
19    /* 循环处理用户请求 */
20    for ( ; ; ) {
21        clilen = sizeof(cliaddr);
22        connfd = accept(listenfd, (SA *) &cliaddr, &clilen);
23        read_data(connfd); /* 读取数据 */
24        close(connfd);     /* 关闭连接套接字，注意不是监听套接字 */
25    }
26 }
27
28 void read_data(int sockfd)
29 {
30     ssize_t n;
31     char buf[1024];
32
33     int time = 0;
34     for ( ; ; ) {
35         fprintf(stdout, "block in read\n");
36         if ( (n = Readn(sockfd, buf, 1024)) == 0)
37             return; /* connection closed by other end */
38
39         time ++;
```

```

40         fprintf(stdout, "1K read for %d \n", time);
41         usleep(1000);
42     }
43 }

```

对服务器端程序解释如下：


6-17 行先后创建了 socket 套接字，bind 到对应地址和端口，并开始调用 listen 接口监听。

20-25 行循环等待连接，通过 accept 获取实际的连接，并开始读取数据。

28-42 行实际每次读取 1K 数据，之后休眠 1 秒，用来模拟服务器端处理时延。

客户端发送数据程序

下面是客户端发送数据的程序：

 复制代码

```

1 int main(int argc, char **argv)
2 {
3     int                sockfd;
4     struct sockaddr_in servaddr;
5
6     if (argc != 2)
7         err_quit("usage: tcpclient <IPaddress>");
8
9     sockfd = socket(AF_INET, SOCK_STREAM, 0);
10
11     bzero(&servaddr, sizeof(servaddr));
12     servaddr.sin_family = AF_INET;
13     servaddr.sin_port = htons(SERV_PORT);
14     inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
15     connect(sockfd, (SA *)&servaddr, sizeof(servaddr));
16     send_data(stdin, sockfd);
17     exit(0);
18 }
19
20 # define MESSAGE_SIZE 10240000
21 void send_data(FILE *fp, int sockfd)
22 {
23     char * query;
24     query = malloc(MESSAGE_SIZE+1);
25     for(int i=0; i< MESSAGE_SIZE; i++){
26         query[i] = 'a';

```



```
27     }
28     query[MESSAGE_SIZE] = '\0';
29
30     const char *cp;
31     cp = query;
32     remaining = strlen(query);
33     while (remaining) {
34         n_written = send(sockfd, cp, remaining, 0);
35         fprintf(stdout, "send into buffer %ld \n", n_written);
36         if (n_written <= 0) {
37             perror("send");
38             return;
39         }
40         remaining -= n_written;
41         cp += n_written;
42     }
43
44     return;
45 }
```

对客户端程序解释如下：

9-15 行先后创建了 socket 套接字，调用 connect 向对应服务器端发起连接请求。

16 行在连接建立成功后，调用 send_data 发送数据。

23-28 行初始化了一个长度为 MESSAGE_SIZE 的字符串流。

33-42 行调用 send 函数将 MESSAGE_SIZE 长度的字符串流发送出去。

实验一：观察客户端数据发送行为

客户端程序发送了一个很大的字节流，程序运行起来之后，我们会看到服务端不断地在屏幕上打印出读取字节流的过程：


```
1K read for 1208
block in read
1K read for 1209
block in read
1K read for 1210
block in read
1K read for 1211
block in read
1K read for 1212
block in read
1K read for 1213
block in read
1K read for 1214
block in read
1K read for 1215
block in read
1K read for 1216
block in read
1K read for 1217
block in read
1K read for 1218
block in read
1K read for 1219
```

而客户端直到最后所有的字节流发送完毕才打印出下面的一句话，说明在此之前 send 函数一直都是阻塞的，也就是说**阻塞式套接字最终发送返回的实际写入字节数和请求字节数是相等的**。

而关于非阻塞套接字的操作，我会在后面的文章中讲解。

实验二：服务端处理变慢

如果我们把服务端的休眠时间稍微调大，把客户端发送的字节数从 10240000 调整为 1024000，再次运行刚才的例子，我们会发现，客户端很快打印出一句话：

```
send into buffer 1024000
```

但与此同时，服务端读取程序还在屏幕上不断打印读取数据的进度，显示出服务端读取程序还在辛苦地从缓冲区中读取数据。

通过这个例子我想再次强调一下：

发送成功仅仅表示的是数据被拷贝到了发送缓冲区中，并不意味着连接对端已经收到所有的数据。至于什么时候发送到对端的接收缓冲区，或者更进一步说，什么时候被对方应用程序缓冲所接收，对我们而言完全都是透明的。

总结

这一讲重点讲述了通过 send 和 read 来收发数据包，你需要牢记以下两点：

对于 send 来说，返回成功仅仅表示数据写到发送缓冲区成功，并不表示对端已经成功收到。

对于 read 来说，需要循环读取数据，并且需要考虑 EOF 等异常条件。

思考题

最后你不妨思考一下，既然缓冲区如此重要，我们可不可以把缓冲区搞得大大的，这样不就可以提高应用程序的吞吐量了么？你可以想一想这个方法可行吗？另外你可以自己总结一下，一段数据流从应用程序发送端，一直到应用程序接收端，总共经过了多少次拷贝？

欢迎你在评论区与我分享你的答案，如果你理解了套接字读写的过程，也欢迎把这篇文章分享给你的朋友或者同事。

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 04 | TCP三次握手：如何使用套接字格式建立连接？

精选留言 (13)

写留言



Geek_Wison

2019-08-12

老师可以将完整的代码的github地址贴出来吗，我想自己编译调试运行一下。

作者回复: 正在进行中



2



破晓^_^

2019-08-12

无限增大缓冲区肯定不行，文章中已经说过write函数发送数据只是将数据发送到内核缓冲区，而什么时候发送由内核决定。内核缓冲区总是充满数据时会产生粘包问题，同时网络的传输大小MTU也会限制每次发送的大小，最后由于数据堵塞需要消耗大量内存资源，资源使用效率不高。

用户缓冲区到内核缓冲区...

展开 ▾

作者回复: 都是强人 😊



👍 1



业余爱好者

2019-08-12

网络程序的性能瓶颈一般在于服务端，所以老师说的增加缓冲区大小应该指的是服务端。如果应用程序的处理速度跟不上，即使缓冲区再大，也不能在整体上提高太多的吞吐率。性能，从来都只能从整体上优化，简单粗暴地提升某一个指标的效果一般。

首先讲客户端发送的数据拷贝到缓冲区，...

展开 ▾



👍 1



在路上

2019-08-12

个人的想法：

单纯的提高缓冲区应该不是可行的方法，以一台计算机思考，内存毕竟是有限的，为每个套接字都开辟一大块内存，那相应可以创建的套接字的就减少了把

展开 ▾

作者回复: 相当于让仓库变大，可以存储了更多的货物，如果出货的速度有限，会有更多的货物烂在仓库里。



👍 1



星亦辰

2019-08-12

有这么一个问题：

假如客户端往服务端发送了1025个字节，而只有1024个字节是有用的。我在服务端用read 读取1024个字节，最后一定是返回1024。然后，剩下的没有用了，不打算读了，怎么样抛弃多余的内容呢。

作者回复: 那你为啥要传这第1025个字节呢？如果是消息的边界，例如换行，还是要读到这个字符的，读完以后不拷贝到应用程序的缓冲区就可以认为是丢弃了。

2

1



阿西吧

2019-08-12

有哪些必看的书籍，求推荐

...

👍



范龙dragon

2019-08-12

第一个readn函数中第24行返回实际字节数的地方，应该是size-nleft吧，通篇没有看到变量n

...

👍



许童童

2019-08-12

一段数据流从应用程序发送端，一直到应用程序接收端，总共经过了多少次拷贝？

客户端 -> 应用程序缓冲区

应用程序缓冲区 -> 发送缓冲区

接收缓冲区 -> 应用程序缓冲区

应用程序缓冲区 -> 服务端...

展开 ▾

...

👍



许童童

2019-08-12

适当增大缓冲区大小是可以解决一些性能问题的，

但是当服务器的处理速度跟不上的时，增大缓冲区也无济于事，大量的数据在缓冲区内久久得不到处理，所以问题还是得不到解决。

展开 ▾

...

👍



阿西吧

2019-08-12

serv.c: In function 'main' :

serv.c:17: warning: incompatible implicit declaration of built-in function 'bzero'

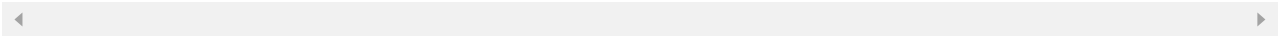
serv.c:24: error: 'SA' undeclared (first use in this function)

serv.c:24: error: (Each undeclared identifier is reported only once

serv.c:24: error: for each function it appears in.)...

展开 ▾

作者回复: 正在准备中.....



大灰狼

2019-08-12

网卡驱动拿到数据，中断后写入收缓冲区

从缓冲区到应用一次。

所以最少两次。

无限大小的缓冲区浪费内存，并不是越大越好。如果缓冲区增大并不能提升收发包效率，可以采取其他方面的优化。

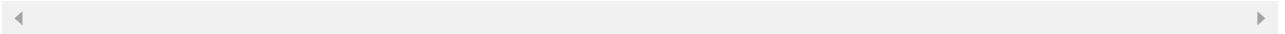


破晓^_^

2019-08-12

老师你好，文中几处贴代码的位置，个人觉得不管是代码片段还是完整程序，应该保持完整性，文中几处都缺少花括号补齐，完整性。

作者回复: 感谢指出，代码会加强审阅



天蝎座的狮子狗

2019-08-12

我遇到一个问题，发送函数send函数返回值竟然比我写的时候传入的buff的bufflen大，不知道为什么😳。

作者回复: 出错了.... 代码贴出来看看

