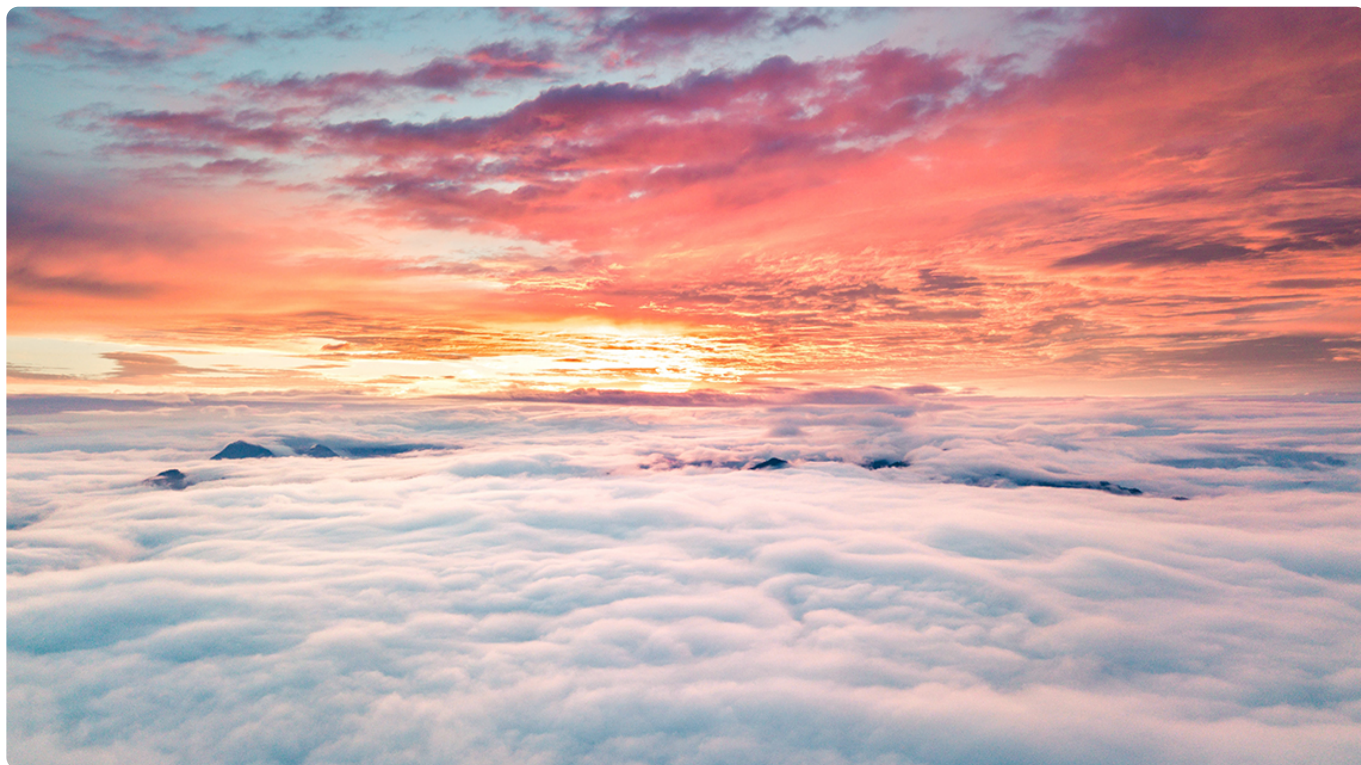


## 16 | 如何理解TCP的“流”？

2019-09-06 盛延敏

网络编程实战

[进入课程 >](#)



讲述：冯永吉

时长 11:31 大小 10.55M



你好，我是盛延敏，这里是网络编程实战第 16 讲，欢迎回来。

上一讲我们讲到了使用 `SO_REUSEADDR` 套接字选项，可以让服务器满足快速重启的需求。在这一讲里，我们回到数据的收发这个主题，谈一谈如何理解 TCP 的数据流特性。

### TCP 是一种流式协议

在前面的章节中，我们讲的都是单个客户端 - 服务器的例子，可能会给你造成一种错觉，好像 TCP 是一种应答形式的数据传输过程，比如发送端一次发送 network 和 program 这样的报文，在前面的例子中，我们看到的结果基本是这样的：

发送端：network ----> 接收端回应：Hi, network

发送端: program -----> 接收端回应: Hi, program


这其实是一个假象, 之所以会这样, 是因为网络条件比较好, 而且发送的数据也比较少。

为了让大家理解 TCP 数据是流式的这个特性, 我们分别从发送端和接收端来阐述。

我们知道, 在发送端, 当我们调用 send 函数完成数据“发送”以后, 数据并没有被真正从网络上发送出去, 只是从应用程序拷贝到了操作系统内核协议栈中, 至于什么时候真正被发送, 取决于发送窗口、拥塞窗口以及当前发送缓冲区的大小等条件。也就是说, 我们不能假设每次 send 调用发送的数据, 都会作为一个整体完整地发送出去。

如果我们考虑实际网络传输过程中的各种影响, 假设发送端陆续调用 send 函数先后发送 network 和 program 报文, 那么实际的发送很有可能是这个样子的。


第一种情况, 一次性将 network 和 program 在一个 TCP 分组中发送出去, 像这样:

 复制代码

```
1 ...xxxnetworkprogramxxx...
```


第二种情况, program 的部分随 network 在一个 TCP 分组中发送出去, 像这样:

TCP 分组 1:

 复制代码

```
1 ...xxxxxnetworkpro
```


TCP 分组 2:

 复制代码

```
1 gramxxxxxxxxxxxx...
```


第三种情况，network 的一部分随 TCP 分组被发送出去，另一部分和 program 一起随另一个 TCP 分组发送出去，像这样。

TCP 分组 1:

 复制代码

```
1 ...xxxxxxxxxxxnet
```

TCP 分组 2:


 复制代码

```
1 workprogramxxx...
```

实际上类似的组合可以枚举出无数种。不管是哪一种，核心的问题就是，我们不知道 network 和 program 这两个报文是如何进行 TCP 分组传输的。换言之，我们在发送数据的时候，不应该假设“数据流和 TCP 分组是一种映射关系”。就好像在前面，我们似乎觉得 network 这个报文一定对应一个 TCP 分组，这是完全不正确的。

如果我们再来看客户端，数据流的特征更明显。

我们知道，接收端缓冲区保留了没有被取走的数据，随着应用程序不断从接收端缓冲区读出数据，接收端缓冲区就可以容纳更多新的数据。如果我们使用 `recv` 从接收端缓冲区读取数据，发送端缓冲区的数据是以字节流的方式存在的，无论发送端如何构造 TCP 分组，接收端最终受到的字节流总是像下面这样：

 复制代码

```
1 xxxxxxxxxxxxxxxxxxxnetworkprogramxxxxxxxxxxxxx
```

关于接收端字节流，有两点需要注意：

第一，这里 network 和 program 的顺序肯定是会保持的，也就是说，先调用 send 函数发送的字节，总在后调用 send 函数发送字节的前面，这个是由 TCP 严格保证的；

第二，如果发送过程中有 TCP 分组丢失，但是其后续分组陆续到达，那么 TCP 协议栈会缓存后续分组，直到前面丢失的分组到达，最终，形成可以被应用程序读取的数据流。

## 网络字节排序

我们知道计算机最终保存和传输，用的都是 0101 这样的二进制数据，字节流在网络上的传输，也是通过二进制来完成的。

从二进制到字节是通过编码完成的，比如著名的 ASCII 编码，通过一个字节 8 个比特对常用的西方字母进行了编码。


这里有一个有趣的问题，如果需要传输数字，比如 0x0201，对应的二进制为 00000010000000001，那么两个字节的的数据到底是先传 0x01，还是相反？



在计算机发展的历史上，对于如何存储这个数据没有形成标准。比如这里讲到的问题，不同的系统就会有两种存法，一种是将 0x02 高字节存放在起始地址，这个叫做**大端字节序** (Big-Endian)。另一种相反，将 0x01 低字节存放在起始地址，这个叫做**小端字节序** (Little-Endian)。

但是在网络传输中，必须保证双方都用同一种标准来表达，这就好比我们打电话时说的是同一种语言，否则双方不能顺畅地沟通。这个标准就涉及到了网络字节序的选择问题，对于网络字节序，必须二选一。我们可以看到网络协议使用的是大端字节序，我个人觉得大端字节序比较符合人类的思维习惯，你可以想象手写一个多位数字，从开始往小位写，自然会先写大位，比如写 12, 1234，这个样子。

为了保证网络字节序一致，POSIX 标准提供了如下的转换函数：

 复制代码


```
1 uint16_t htons (uint16_t hostshort)
2 uint16_t ntohs (uint16_t netshort)
3 uint32_t htonl (uint32_t hostlong)
4 uint32_t ntohl (uint32_t netlong)
```

这里函数中的 n 代表的就是 network，h 代表的是 host，s 表示的是 short，l 表示的是 long，分别表示 16 位和 32 位的整数。

这些函数可以帮助我们在主机（host）和网络（network）的格式间灵活转换。当使用这些函数时，我们并不需要关心主机到底是什么样的字节顺序，只要使用函数给定值进行网络字节序和主机字节序的转换就可以了。

你可以想象，如果碰巧我们的系统本身是大端字节序，和网络字节序一样，那么使用上述所有的函数进行转换的时候，结果都仅仅是一个空实现，直接返回。

比如这样：

 复制代码

```
1 # if __BYTE_ORDER == __BIG_ENDIAN
2 /* The host byte order is the same as network byte order,
3    so these functions are all just identity. */
4 # define ntohl(x) (x)
5 # define ntohs(x) (x)
6 # define htonl(x) (x)
7 # define htons(x) (x)
```

## 报文读取和解析

应该看到，报文是以字节流的形式呈现给应用程序的，那么随之而来的一个问题就是，应用程序如何解读字节流呢？

这就要说到报文格式和解析了。报文格式实际上定义了字节的组织形式，发送端和接收端都按照统一的报文格式进行数据传输和解析，这样就可以保证彼此能够完成交流。

只有知道了报文格式，接收端才能针对性地进行报文读取和解析工作。

报文格式最重要的是如何确定报文的边界。常见的报文格式有两种方法，一种是发送端把要发送的报文长度预先通过报文告知给接收端；另一种是通过一些特殊的字符来进行边界的划分。

## 显式编码报文长度

### 报文格式


下面我们来看一个例子，这个例子是把要发送的报文长度预先通过报文告知接收端，你可以看到文章中有这样一张图。

消息长度	消息类型	请求正文
------	------	------

由图可以看出，这个报文的格式很简单，首先 4 个字节大小的消息长度，其目的是将真正发送的字节流的大小显式通过报文告知接收端，接下来是 4 个字节大小的消息类型，而真正需要发送的数据则紧随其后。

### 发送报文

发送端的程序如下：

 复制代码

```
1 int main(int argc, char **argv) {
2     if (argc != 2) {
3         error(1, 0, "usage: tcpclient <IPaddress>");
4     }
5
6     int socket_fd;
7     socket_fd = socket(AF_INET, SOCK_STREAM, 0);
8
9     struct sockaddr_in server_addr;
10    bzero(&server_addr, sizeof(server_addr));
11    server_addr.sin_family = AF_INET;
12    server_addr.sin_port = htons(SERV_PORT);
13    inet_pton(AF_INET, argv[1], &server_addr.sin_addr);
14
15    socklen_t server_len = sizeof(server_addr);
16    int connect_rt = connect(socket_fd, (struct sockaddr *) &server_addr, server_len);
```

```

17     if (connect_rt < 0) {
18         error(1, errno, "connect failed ");
19     }
20
21     struct {
22         u_int32_t message_length;
23         u_int32_t message_type;
24         char buf[128];
25     } message;
26
27     int n;
28
29     while (fgets(message.buf, sizeof(message.buf), stdin) != NULL) {
30         n = strlen(message.buf);
31         message.message_length = htonl(n);
32         message.message_type = 1;
33         if (send(socket_fd, (char *) &message, sizeof(message.message_length) + sizeof(r
34             0)
35             error(1, errno, "send failure");
36
37     }
38     exit(0);
39 }

```

程序的 1-20 行是常规的创建套接字和地址，建立连接的过程。我们重点往下看，21-25 行就是图示的报文格式转化为结构体，29-37 行从标准输入读入数据，分别对消息长度、类型进行了初始化，注意这里使用了 `htonl` 函数将字节大小转化为了网络字节顺序，这一点很重要。最后我们看到 23 行实际发送的字节流大小为消息长度 4 字节，加上消息类型 4 字节，以及标准输入的字符串大小。

## 解析报文：程序

下面给出的是服务器端的程序，和客户端不一样的是，服务器端需要对报文进行解析。

 复制代码

```

1 static int count;
2
3 static void sig_int(int signo) {
4     printf("\nreceived %d datagrams\n", count);
5     exit(0);
6 }
7
8
9 int main(int argc, char **argv) {

```



```

10  int listenfd;
11  listenfd = socket(AF_INET, SOCK_STREAM, 0);
12
13  struct sockaddr_in server_addr;
14  bzero(&server_addr, sizeof(server_addr));
15  server_addr.sin_family = AF_INET;
16  server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
17  server_addr.sin_port = htons(SERV_PORT);
18
19  int on = 1;
20  setsockopt(listenfd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));
21
22  int rt1 = bind(listenfd, (struct sockaddr *) &server_addr, sizeof(server_addr));
23  if (rt1 < 0) {
24      error(1, errno, "bind failed ");
25  }
26
27  int rt2 = listen(listenfd, LISTENQ);
28  if (rt2 < 0) {
29      error(1, errno, "listen failed ");
30  }
31
32  signal(SIGPIPE, SIG_IGN);
33
34  int connfd;
35  struct sockaddr_in client_addr;
36  socklen_t client_len = sizeof(client_addr);
37
38  if ((connfd = accept(listenfd, (struct sockaddr *) &client_addr, &client_len)) < 0)
39      error(1, errno, "bind failed ");
40  }
41
42  char buf[128];
43  count = 0;
44
45  while (1) {
46      int n = read_message(connfd, buf, sizeof(buf));
47      if (n < 0) {
48          error(1, errno, "error read message");
49      } else if (n == 0) {
50          error(1, 0, "client closed \n");
51      }
52      buf[n] = 0;
53      printf("received %d bytes: %s\n", n, buf);
54      count++;
55  }
56
57  exit(0);
58
59 }


```



这个程序 1-41 行创建套接字，等待连接建立部分和前面基本一致。我们重点看 42-55 行的部分。45-55 行循环处理字节流，调用 `read_message` 函数进行报文解析工作，并把报文的主体通过标准输出打印出来。

## 解析报文：readn 函数

在了解 `read_message` 工作原理之前，我们先来看第 5 讲就引入的一个函数：`readn`。这里一定要强调的是 `readn` 函数的语义，**读取报文预设大小的字节**，`readn` 调用会一直循环，尝试读取预设大小的字节，如果接收缓冲区数据空，`readn` 函数会阻塞在那里，直到有数据到达。

 复制代码


```
1 size_t readn(int fd, void *buffer, size_t length) {
2     size_t count;
3     ssize_t nread;
4     char *ptr;
5
6     ptr = buffer;
7     count = length;
8     while (count > 0) {
9         nread = read(fd, ptr, count);
10
11         if (nread < 0) {
12             if (errno == EINTR)
13                 continue;
14             else
15                 return (-1);
16         } else if (nread == 0)
17             break;          /* EOF */
18
19         count -= nread;
20         ptr += nread;
21     }
22     return (length - count); /* return >= 0 */
23 }
```

`readn` 函数中使用 `count` 来表示还需要读取的字符数，如果 `count` 一直大于 0，说明还没有满足预设的字符大小，循环就会继续。第 9 行通过 `read` 函数来服务最多 `count` 个字符。11-17 行针对返回值进行出错判断，其中返回值为 0 的情形是 EOF，表示对方连接终

止。19-20 行要读取的字符数减去这次读到的字符数，同时移动缓冲区指针，这样做的目的是为了确认字符数是否已经读取完毕。

## 解析报文: read\_message 函数

有了 readn 函数作为基础，我们再看一下 read\_message 对报文的解析处理：

 复制代码

```
1 size_t read_message(int fd, char *buffer, size_t length) {
2     u_int32_t msg_length;
3     u_int32_t msg_type;
4     int rc;
5
6     rc = readn(fd, (char *) &msg_length, sizeof(u_int32_t));
7     if (rc != sizeof(u_int32_t))
8         return rc < 0 ? -1 : 0;
9     msg_length = ntohl(msg_length);
10
11    rc = readn(fd, (char *) &msg_type, sizeof(msg_type));
12    if (rc != sizeof(u_int32_t))
13        return rc < 0 ? -1 : 0;
14
15    if (msg_length > length) {
16        return -1;
17    }
18
19    rc = readn(fd, buffer, msg_length);
20    if (rc != msg_length)
21        return rc < 0 ? -1 : 0;
22    return rc;
23 }
```

在这个函数中，第 6 行通过调用 readn 函数获取 4 个字节的消息长度数据，紧接着，第 11 行通过调用 readn 函数获取 4 个字节的消息类型数据。第 15 行判断消息的长度是不是太大，如果大到本地缓冲区不能容纳，则直接返回错误；第 19 行调用 readn 一次性读取已知长度的消息体。

## 实验

我们依次启动作为报文解析的服务器一端，以及作为报文发送的客户端。我们看到，每次客户端发送的报文都可以被服务器端解析出来，在标准输出上的结果验证了这一点。

```
1 $./streamserver
2 received 8 bytes: network
3 received 5 bytes: good
```

```
1 $./streamclient
2 network
3 good
```

## 特殊字符作为边界

前面我提到了两种报文格式，另外一种报文格式就是通过设置特殊字符作为报文边界。HTTP 是一个非常好的例子。

请求方法	空格	URL	空格	协议版本	回车	换行
头部字段名	:	值	回车	换行		
头部字段名	:	值	回车	换行		
回车	换行					
请求正文						

HTTP 通过设置回车符、换行符做为 HTTP 报文协议的边界。

下面的 `read_line` 函数就是在尝试读取一行数据，也就是读到回车符 `\r`，或者读到回车换行符 `\r\n` 为止。这个函数每次尝试读取一个字节，第 9 行如果读到了回车符 `\r`，接下来在 11 行的“观察”下看有没有换行符，如果有就在第 12 行读取这个换行符；如果没有读到回车符，就在第 16-17 行将字符放到缓冲区，并移动指针。

```
1 int read_line(int fd, char *buf, int size) {
2     int i = 0;
3     char c = '\0';
```

```
4     int n;
5
6     while ((i < size - 1) && (c != '\n')) {
7         n = recv(fd, &c, 1, 0);
8         if (n > 0) {
9             if (c == '\r') {
10                 n = recv(fd, &c, 1, MSG_PEEK);
11                 if ((n > 0) && (c == '\n'))
12                     recv(fd, &c, 1, 0);
13                 else
14                     c = '\n';
15             }
16             buf[i] = c;
17             i++;
18         } else
19             c = '\n';
20     }
21     buf[i] = '\0';
22
23     return (i);
24 }
```

## 总结

和我们预想的不太一样，TCP 数据流特性决定了字节流本身是没有边界的，一般我们通过显式编码报文长度的方式，以及选取特殊字符区分报文边界的方式来进行报文格式的设计。而对报文解析的工作就是要在知道报文格式的情况下，有效地对报文信息进行还原。

## 思考题

和往常一样，这里给你留两道思考题，供你消化今天的内容。

第一道题关于 HTTP 的报文格式，我们看到，既要处理只有回车的情景，也要处理同时有回车和换行的情景，你知道造成这种情况的原因是什么吗？

第二道题是，我们这里讲到的报文格式，和 TCP 分组的报文格式，有什么区别和联系吗？

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，与他们一起交流一下这两个问题吧。

# 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 15 | 怎么老是出现“地址已经被使用”？

下一篇 17 | TCP并不总是“可靠”的？

## 精选留言 (17)

写留言



wyf2317

2019-09-08

1. windos 和mac linux 的换行不一样。\\r 或\\r\\n
2. 所属的层级不一样，应用层和网络层

但本质上就是人为预定的对二进制数据序列化的方式，没有太大的差别，都是通信协议。



1



iron\_man

2019-09-07

一直有个疑问，趁这堂课向老师请教一下，前面客户端发送消息时，消息长度转成网络序了，后面的消息为何没有转成网络序，如果消息里面含有数字呢？如果消息里面全是字符呢？



1



刘晓林

2019-09-07

老师，关于网络序和主机序，我有3个问题想要请教一下：

- 1.在数据发送的时候，是先发送内存中高地址的数据，还是先发内存中低地址的数据？比如char\* sendline="abcdef", 是从a到f的顺序去发，还是从f到a的顺序去发。
- 2.接受的时候，是现接受到的是网络序中的高地址，还是低地址？
- 3.socket接口，如read，send这些函数，会自动帮我们完成主机序和网络序之间的转换...

展开 ▾



1



1



卫江

2019-09-06

问题1，window与linux平台对于回车换行的编码不一致。

问题2，协议本质来说就是大家协商好，便于沟通的内容形式。所以，tcp与我们自定义的协议本质来说没有什么不用，区别只是针对的业务不同而已！

展开 ▾



1



徐凯

2019-09-06

第一个是不是跟windows文本换行与linux文本换行的字符不同有关 windows上好像是一个换行符一个回车符 linux是一个换行符

展开 ▾



1



Brave Shine

2019-09-06

1. linux和windows在编码http层面的不同？
2. tcp的分组报文是传输层在协议栈层面怎么发tcp包的规范，这里指的是应用层报文

展开 ▾



1



yusuf

2019-09-08

第一个问题是因为windows系统下的回车是\r\n，而类UNIX系统下的回车是\n



刘晓林

2019-09-07

老师，你这里的struct不会有内存对齐的操作吗？这样struct的大小信息和和自行的位置信

息会不会就不对了呀？

展开 ▾



**Chao**

2019-09-06

老师 我什么时候答疑

最大分组 MSL 是 TCP 分组在网络中存活的最长时间这个问题

展开 ▾



**张立华**

2019-09-06

我的操作系统是：centos 7.4 64位操作系统。

short int = 258;

258=0x0102...

展开 ▾



**传说中的成大大**

2019-09-06

```
message.message_length = htonl( n );
```

```
message.message_type = 1;
```

我在自己写代码实现的时候突然想起这两句代码为什么一个需要htonl一个不需要呢？

💬 4



**传说中的成大大**

2019-09-06

网络字节序大端小端问题 具体是大端还是小端应该不是应用层决定的吧 我们只需要发送和接收的时候通过hton\* 和ntoh\*函数进行转换就可以用了吧？

展开 ▾



**锦**

2019-09-06

回答第一个问题，应该是历史原因吧？

回答第二个问题，这里讲的是应用程序层面的报文格式，tcp分组报文格式是协议层面的。

应用层面的报文在先发送到协议栈，然后在协议栈以tcp分组报文发送出去。



展开 ▾



**传说中的成大大**

2019-09-06

第1问:

大胆猜测因为http是以回车换行作为分界符但是在程序在编码过程中有可能随时只产生一个回车符

第2问:

tcp分组格式 是tcp层面的东西 而本文提到的报文是应用层方面的东西,联系在于tcp层面...

展开 ▾



**Geek\_007**

2019-09-06

老师，如果有多个进程都在对一个socket fd进行write操作，一个要写abc,一个要写123。会不会最终发出去的流是ab1c23。这样是不是靠长度和分割符的方式就无效了。

💬 2



**Berry Wang**

2019-09-06

非计算机专业出身的Java程序员看的一知半解，周末好好消化一下。



**( \_ \_ )**

2019-09-06

可能是因为windows下编码换行符号是0d0a吧

展开 ▾

