

## 06 | 嗨，别忘了UDP这个小兄弟

2019-08-14 盛延敏

网络编程实战

[进入课程 >](#)



讲述：冯永吉

时长 09:21 大小 8.58M



你好，我是盛延敏，这里是网络编程实战第 6 讲，欢迎回来。

前面几讲我们讲述了 TCP 方面的编程知识，这一讲我们来讲讲 UDP 方面的编程知识。

如果说 TCP 是网络协议的“大哥”，那么 UDP 可以说是“小兄弟”。这个小兄弟和大哥比，有什么差异呢？

首先，UDP 是一种“数据报”协议，而 TCP 是一种面向连接的“数据流”协议。

TCP 可以用日常生活中打电话的场景打比方，前面也多次用到了这样的例子。在这个例子中，拨打号码，接通电话，开始交流，分别对应了 TCP 的三次握手和报文传送。一旦双方

的连接建立，那么双方对话时，一定知道彼此是谁。这个时候我们就说，这种对话是有上下文的。

同样的，我们也可以给 UDP 找一个类似的例子，这个例子就是邮寄明信片。在这个例子中，发信方在明信片中填上了接收方的地址和邮编，投递到邮局的邮筒之后，就可以不管了。发信方也可以给这个接收方再邮寄第二张、第三张，甚至是第四张明信片，但是这几张明信片之间是没有任何关系的，他们的到达顺序也是不保证的，有可能最后寄出的第四张明信片最先到达接收者的手中，因为没有序号，接收者也不知道这是第四张寄出的明信片；而且，即使接收方没有收到明信片，也没有办法重新邮寄一遍该明信片。

这两个简单的例子，道出了 UDP 和 TCP 之间最大的区别。

TCP 是一个面向连接的协议，TCP 在 IP 报文的基础上，增加了诸如重传、确认、有序传输、拥塞控制等能力，通信的双方是在一个确定的上下文中工作的。

而 UDP 则不同，UDP 没有这样一个确定的上下文，它是一个不可靠的通信协议，没有重传和确认，没有有序控制，也没有拥塞控制。我们可以简单地理解为，在 IP 报文的基础上，UDP 增加的能力有限。

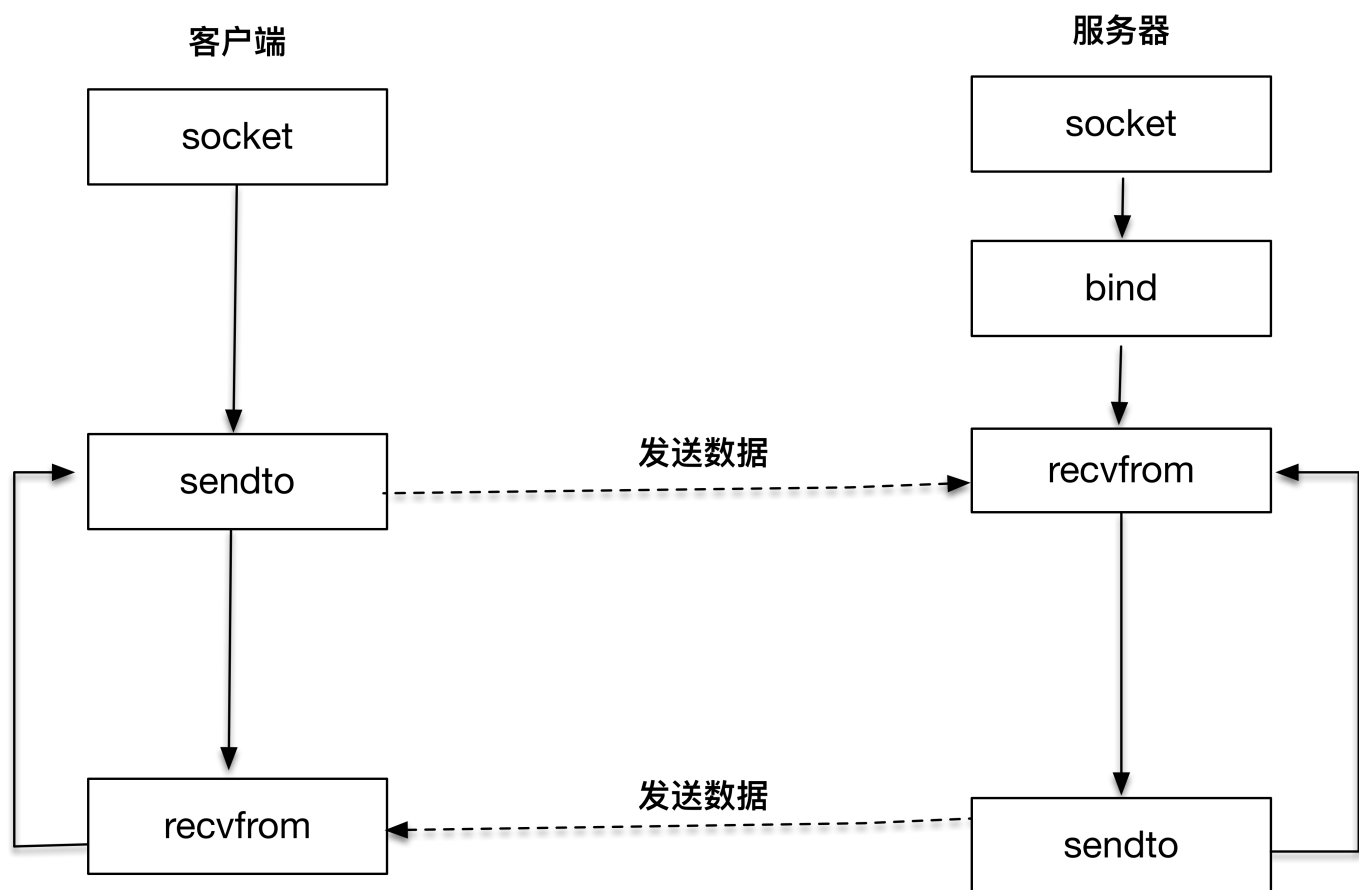
UDP 不保证报文的有效传递，不保证报文的有序，也就是说使用 UDP 的时候，我们需要做好丢包、重传、报文组装等工作。

既然如此，为什么我们还要使用 UDP 协议呢？

答案很简单，因为 UDP 比较简单，适合的场景还是比较多的，我们常见的 DNS 服务，SNMP 服务都是基于 UDP 协议的，这些场景对时延、丢包都不是特别敏感。另外多人通信的场景，如聊天室、多人游戏等，也都会使用到 UDP 协议。


## UDP 编程

UDP 和 TCP 编程非常不同，下面这张图是 UDP 程序设计时的主要过程。



我们看到服务器端创建 UDP 套接字之后，绑定到本地端口，调用 `recvfrom` 函数等待客户端的报文发送；客户端创建套接字之后，调用 `sendto` 函数往目标地址和端口发送 UDP 报文，然后客户端和服务端进入互相应答过程。

`recvfrom` 和 `sendto` 是 UDP 用来接收和发送报文的两个主要函数：

 复制代码

```
1 #include <sys/socket.h>
2
3 ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags,
4                 struct sockaddr *from, socklen_t *addrlen);
5
6 ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags,
7               const struct sockaddr *to, socklen_t *addrlen);
```

我们先来看一下 `recvfrom` 函数。

`sockfd`、`buff` 和 `nbytes` 是前三个参数。`sockfd` 是本地创建的套接字描述符，`buff` 指向本地的缓存，`nbytes` 表示最大接收数据字节。

第四个参数 flags 是和 I/O 相关的参数，这里我们还用不到，设置为 0。

后面两个参数 from 和 addrlen，实际上是返回对端发送方的地址和端口等信息，这和 TCP 非常不一样，TCP 是通过 accept 函数拿到的描述符信息来决定对端的信息。另外 UDP 报文每次接收都会获取对端的信息，也就是说报文和报文之间是没有上下文的。

函数的返回值告诉我们实际接收的字节数。

接下来看一下 sendto 函数。

sendto 函数中的前三个参数为 sockfd、buff 和 nbytes。sockfd 是本地创建的套接字描述符，buff 指向发送的缓存，nbytes 表示发送字节数。第四个参数 flags 依旧设置为 0。

后面两个参数 to 和 addrlen，表示发送的对端地址和端口等信息。

函数的返回值告诉我们实际接收的字节数。

我们知道，TCP 的发送和接收每次都是在一个上下文中，类似这样的过程：

A 连接上: 接收→发送→接收→发送→...


B 连接上: 接收→发送→接收→发送→ ...

而 UDP 的每次接收和发送都是一个独立的上下文，类似这样：

接收 A→发送 A→接收 B→发送 B →接收 C→发送 C→ ...

## UDP 服务端例子

我们先来看一个 UDP 服务器端的例子：

 复制代码

```
1 #include "lib/common.h"
2
3 static int count;
4
5 static void recvfrom_int(int signo) {
```

```

6     printf("\nreceived %d datagrams\n", count);
7     exit(0);
8 }
9
10
11 int main(int argc, char **argv) {
12     int socket_fd;
13     socket_fd = socket(AF_INET, SOCK_DGRAM, 0);
14
15     struct sockaddr_in server_addr;
16     bzero(&server_addr, sizeof(server_addr));
17     server_addr.sin_family = AF_INET;
18     server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
19     server_addr.sin_port = htons(SERV_PORT);
20
21     bind(socket_fd, (struct sockaddr *) &server_addr, sizeof(server_addr));
22
23     socklen_t client_len;
24     char message[MAXLINE];
25     count = 0;
26
27     signal(SIGINT, recvfrom_int);
28
29     struct sockaddr_in client_addr;
30     client_len = sizeof(client_addr);
31     for (;;) {
32         int n = recvfrom(socket_fd, message, MAXLINE, 0, (struct sockaddr *) &client_addr,
33             &client_len);
34         message[n] = 0;
35         printf("received %d bytes: %s\n", n, message);
36
37         char send_line[MAXLINE];
38         sprintf(send_line, "Hi, %s", message);
39
40         sendto(socket_fd, send_line, strlen(send_line), 0, (struct sockaddr *) &client_addr,
41             &client_len);
42
43         count++;
44     }
45 }

```

程序的 12~13 行，首先创建一个套接字，注意这里的套接字类型是 “SOCK\_DGRAM”，表示的是 UDP 数据报。


15~21 行和 TCP 服务器端类似，绑定数据报套接字到本地的一个端口上。

27 行为该服务器创建了一个信号处理函数，以便在响应 “Ctrl+C” 退出时，打印出收到的报文总数。

31 ~ 42 行是该服务器端的主体，通过调用 `recvfrom` 函数获取客户端发送的报文，之后我们对收到的报文进行重新改造，加上 “Hi” 的前缀，再通过 `sendto` 函数发送给客户端对端。

## UDP 客户端例子

接下来我们再来构建一个对应的 UDP 客户端。在这个例子中，从标准输入中读取输入的字符串后，发送给服务端，并且把服务端经过处理的报文打印到标准输出上。

 复制代码

```
1 #include "lib/common.h"
2
3 # define    MAXLINE    4096
4
5 int main(int argc, char **argv) {
6     if (argc != 2) {
7         error(1, 0, "usage: udpclient <IPaddress>");
8     }
9
10    int socket_fd;
11    socket_fd = socket(AF_INET, SOCK_DGRAM, 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_port = htons(SERV_PORT);
17    inet_pton(AF_INET, argv[1], &server_addr.sin_addr);
18
19    socklen_t server_len = sizeof(server_addr);
20
21    struct sockaddr *reply_addr;
22    reply_addr = malloc(server_len);
23
24    char send_line[MAXLINE], recv_line[MAXLINE + 1];
25    socklen_t len;
26    int n;
27
28    while (fgets(send_line, MAXLINE, stdin) != NULL) {
29        int i = strlen(send_line);
30        if (send_line[i - 1] == '\n') {
31            send_line[i - 1] = 0;
32        }
33
```

```

34     printf("now sending %s\n", send_line);
35     size_t rt = sendto(socket_fd, send_line, strlen(send_line), 0, (struct sockaddr
36     if (rt < 0) {
37         error(1, errno, "send failed ");
38     }
39     printf("send bytes: %zu \n", rt);
40
41     len = 0;
42     n = recvfrom(socket_fd, recv_line, MAXLINE, 0, reply_addr, &len);
43     if (n < 0)
44         error(1, errno, "recvfrom failed");
45     recv_line[n] = 0;
46     fputs(recv_line, stdout);
47     fputs("\n", stdout);
48 }
49
50 exit(0);
51 }

```

10~11 行创建一个类型为 “SOCK\_DGRAM” 的套接字。


13~17 行，初始化目标服务器的地址和端口。

28~51 行为程序主体，从标准输入中读取的字符进行处理后，调用 `sendto` 函数发送给目标服务器端，然后再次调用 `recvfrom` 函数接收目标服务器发送过来的新报文，并将其打印到标准输出上。

为了让你更好地理解 UDP 和 TCP 之间的差别，我们模拟一下 UDP 的三种运行场景，你不妨思考一下这三种场景的结果和 TCP 的到底有什么不同？

## 场景一：只运行客户端

如果我们只运行客户端，程序会一直阻塞在 `recvfrom` 上。

 复制代码

```


1 $ ./udpclient 127.0.0.1
2 1
3 now sending g1
4 send bytes: 2
5 < 阻塞在这里 >

```

还记得 TCP 程序吗？如果不开启服务端，TCP 客户端的 connect 函数会直接返回 “Connection refused” 报错信息。而在 UDP 程序里，则会一直阻塞在这里。

## 场景二：先开启服务端，再开启客户端

在这个场景里，我们先开启服务端在端口侦听，然后再开启客户端：

 复制代码

```
1 ./udpservice
2 received 2 bytes: g1
3 received 2 bytes: g2
```

 复制代码

```
1 ./udpclient 127.0.0.1
2 g1
3 now sending g1
4 send bytes: 2
5 Hi, g1
6 g2
7 now sending g2
8 send bytes: 2
9 Hi, g2
```

我们在客户端一次输入 g1、g2，服务器端在屏幕上打印出收到的字符，并且可以看到，我们的客户端也收到了服务端的回应：“Hi, g1” 和 “Hi, g2”。

## 场景三：开启服务端，再一次开启两个客户端

这个实验中，在服务端开启之后，依次开启两个客户端，并发送报文。

服务端：


 复制代码

```
1 ./udpservice
2 received 2 bytes: g1
3 received 2 bytes: g2
```




```
4 received 2 bytes: g3
5 received 2 bytes: g4
```

## 第一个客户端：

 复制代码

```
1 $./udpclient 127.0.0.1
2 now sending g1
3 send bytes: 2
4 Hi, g1
5 g3
6 now sending g3
7 send bytes: 2
8 Hi, g3
```


## 第二个客户端：

 复制代码

```
1 $./udpclient 127.0.0.1
2 now sending g2
3 send bytes: 2
4 Hi, g2
5 g4
6 now sending g4
7 send bytes: 2
8 Hi, g4
```

我们看到，两个客户端发送的报文，依次都被服务端收到，并且客户端也可以收到服务端处理之后的报文。

如果我们此时把服务器端进程杀死，就可以看到信号函数在进程退出之前，打印出服务器端接收到的报文个数。


 复制代码

```
1 $ ./udpserver
2 received 2 bytes: g1
```

```
3 received 2 bytes: g2
4 received 2 bytes: g3
5 received 2 bytes: g4
6 ^C
7 received 4 datagrams
```


之后，我们再重启服务器端进程，并使用客户端 1 和客户端 2 继续发送新的报文，我们可以看到和 TCP 非常不同的结果。

以下就是服务器端的输出，服务器端重启后可以继续收到客户端的报文，这在 TCP 里是不可以的，TCP 断联之后必须重新连接才可以发送报文信息。但是 UDP 报文的“无连接”的特点，可以在 UDP 服务器重启之后，继续进行报文的发送，这就是 UDP 报文“无上下文”的最好说明。

 复制代码

```
1 $ ./udpserver
2 received 2 bytes: g1
3 received 2 bytes: g2
4 received 2 bytes: g3
5 received 2 bytes: g4
6 ^C
7 received 4 datagrams
8 $ ./udpserver
9 received 2 bytes: g5
10 received 2 bytes: g6
```

第一个客户端：


 复制代码

```
1 $ ./udpclient 127.0.0.1
2 now sending g1
3 send bytes: 2
4 Hi, g1
5 g3
6 now sending g3
7 send bytes: 2
8 Hi, g3
9 g5
10 now sending g5
11 send bytes: 2
```

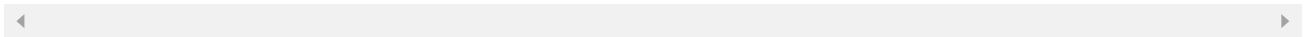
12 Hi, g5



第二个客户端：

 复制代码

```
1 $./udpclient 127.0.0.1
2 now sending g2
3 send bytes: 2
4 Hi, g2
5 g4
6 now sending g4
7 send bytes: 2
8 Hi, g4
9 g6
10 now sending g6
11 send bytes: 2
12 Hi, g6
```



## 总结

在这一讲里，我介绍了 UDP 程序的例子，我们需要重点关注以下两点：

UDP 是无连接的数据报程序，和 TCP 不同，不需要三次握手建立一条连接。

UDP 程序通过 `recvfrom` 和 `sendto` 函数直接收发数据报报文。

## 思考题

最后我给大家留两个思考题吧。在第一个场景中，`recvfrom` 一直处于阻塞状态中，这是非常不合理的，你觉得这种情形应该怎么处理呢？另外，既然 UDP 是请求 - 应答模式的，那么请求中的 UDP 报文最大可以是多大呢？

欢迎你在评论区写下你的思考，我会和你一起讨论。也欢迎把这篇文章分享给你的朋友或者同事，一起讨论一下 UDP 这个协议。

# 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 05 | 使用套接字进行读写：开始交流吧

下一篇 07 | What? 还有本地套接字？

## 精选留言 (15)

写留言



许童童

2019-08-14

第一个场景中，可以添加超时时间做处理，当然你也可以自己实现一个复杂的请求-确认模式，那这样就跟TCP类似了，HTTP/3就是这样做的。

用UDP协议发送时，用sendto函数最大能发送数据的长度为：65535 - IP头(20) - UDP头(8) = 65507字节。用sendto函数发送数据时，如果发送数据长度大于该值，则函数会返...  
展开

1

15



刘明

2019-08-14

1、一直阻塞会导致程序无法正常退出，可以使用接收超时、IO多路复用的超时机制。

2、IP和UDP头中都有16bit的长度字段，最长65535字节，去掉头部长得到UDP数据净荷长度： $65535 - 20 - 8 = 65507$ 字节。

展开 ▾

1

4



**Godblessor**

2019-08-14

recvfrom函数可以设置超时，当限定时间内未接收到服务器端回复，打印超时消息并退出

2

2



**岳文**

2019-08-14

recv from会导致进程一直阻塞，可以通过setsockopt接口设置TIMEOUT 超时后退出。

2

2



**范龙dragon**

2019-08-18

客户端代码的29行sendline数组之前没有初始化数组元素为0，直接用strlen应该会有问题吧，strlen不是以0作为结束标志吗？

展开 ▾

1

1



**江永枫**

2019-08-14

关于阻塞io，可以考虑使用多线程模型去提升性能，或者结合io多路复用来处理能力。

<https://m.php.cn/article/410029.html>

展开 ▾

作者回复: 很快就会讲到了 : )

1

1



**传说中的成大大**

2019-08-17

前些天看完了文章内容 直到今天才把课程上的代码重新实现了一遍并且按照文章的步骤一步一步实现，在重启服务器过后客户端再次发送消息之前客户端其实一直阻塞在等待输入或者recvfrom函数并没有像tcp那样建立连接，这是最好的证据证明udp不需要建立连接

展开 ▾



**Sweety**

2019-08-16

日常打卡.晚了2天.

展开 ▾



**无名**

2019-08-16

对于场景一：只运行客户端

调用sendto也是成功的，我们怎么查看实际的到达记录呢？

展开 ▾

作者回复: debug或加log



**星亦辰**

2019-08-15

udp端口扫描 有没有快速的方法？

展开 ▾

作者回复: 你可以查查资料，我猜就一个一个端口试吧



**Steiner**

2019-08-14

这里sendto的第六个参数应该不是对长度的引用，而是对长度的传值

作者回复: 正解



**传说中的成大大**

2019-08-14

第一问阻塞的情况下可以采用非阻塞套接字方式

第二问应该是65535



**二姐**

2019-08-14

一般udp协议的通信都是用多线程来控制成非阻塞式吧 也就是recieve和send分别放进两个线程里循环调用



**haer**

2019-08-14

那么请求中的 UDP 报文最大可以是多大呢？

1500-20-8=1472

展开 ∨



**zhchnchn**

2019-08-14

如果不开启服务端，TCP 客户端的 connect 函数会直接返回 “Connection refused” 报错信息。而在 UDP 程序里，则会一直阻塞在这里。

-----

这个地方不太理解。请问老师,对TCP来说，收到 “Connection refused” 报错信息，表明收到了服务端的RST包，如果服务端不开启，谁负责发送RST包呢？

展开 ∨

作者回复: 不是RST包，RST的意思是connection reset；这里connection refused是对方的TCP协议栈发送的，工作在操作系统内核中。



1

