

14 | UDP也可以是“已连接”？

2019-09-02 盛延敏

网络编程实战

[进入课程 >](#)



讲述：冯永吉

时长 09:02 大小 8.28M



你好，我是盛延敏，这里是网络编程实战的第 14 讲，欢迎回来。

在前面的基础篇中，我们已经接触到了 UDP 数据报协议相关的知识，在我们的脑海里，已经深深印上了“**UDP 等于无连接协议**”的特性。那么看到这一讲的题目，你是不是觉得有点困惑？没关系，和我一起进入“已连接”的 UDP 的世界，回头再看这个标题，相信你就会恍然大悟。

从一个例子开始

我们先从一个客户端例子开始，在这个例子中，客户端在 UDP 套接字上调用 connect 函数，之后将标准输入的字符串发送到服务器端，并从服务器端接收处理后的报文。当然，和服务器端发送和接收报文是通过调用函数 sendto 和 recvfrom 来完成的。

```
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: udpclient1 <IPaddress>");
7     }
8
9     int socket_fd;
10    socket_fd = socket(AF_INET, SOCK_DGRAM, 0);
11
12    struct sockaddr_in server_addr;
13    bzero(&server_addr, sizeof(server_addr));
14    server_addr.sin_family = AF_INET;
15    server_addr.sin_port = htons(SERV_PORT);
16    inet_pton(AF_INET, argv[1], &server_addr.sin_addr);
17
18    socklen_t server_len = sizeof(server_addr);
19
20    if (connect(socket_fd, (struct sockaddr *) &server_addr, server_len)) {
21        error(1, errno, "connect failed");
22    }
23
24    struct sockaddr *reply_addr;
25    reply_addr = malloc(server_len);
26
27    char send_line[MAXLINE], recv_line[MAXLINE + 1];
28    socklen_t len;
29    int n;
30
31    while (fgets(send_line, MAXLINE, stdin) != NULL) {
32        int i = strlen(send_line);
33        if (send_line[i - 1] == '\n') {
34            send_line[i - 1] = 0;
35        }
36
37        printf("now sending %s\n", send_line);
38        size_t rt = sendto(socket_fd, send_line, strlen(send_line), 0, (struct sockaddr
39        if (rt < 0) {
40            error(1, errno, "sendto failed");
41        }
42        printf("send bytes: %zu \n", rt);
43
44        len = 0;
45        recv_line[0] = 0;
46        n = recvfrom(socket_fd, recv_line, MAXLINE, 0, reply_addr, &len);
47        if (n < 0)
48            error(1, errno, "recvfrom failed");
49        recv_line[n] = 0;
50        fputs(recv_line, stdout);
```

```
51     fputs("\n", stdout);
52 }
53
54     exit(0);
55 }
```

我对这个程序做一个简单的解释：


9-10 行创建了一个 UDP 套接字；

12-16 行创建了一个 IPv4 地址，绑定到指定端口和 IP；

20-22 行调用 connect 将 UDP 套接字和 IPv4 地址进行了“绑定”，这里 connect 函数的名称有点让人误解，其实可能更好的选择是叫做 setpeername；

31-55 行是程序的主体，读取标准输入字符串后，调用 sendto 发送给对端；之后调用 recvfrom 等待对端的响应，并把对端响应信息打印到标准输出。

在没有开启服务端的情况下，我们运行一下这个程序：

 复制代码

```
1 $ ./udpconnectclient 127.0.0.1
2 g1
3 now sending g1
4 send bytes: 2
5 recvfrom failed: Connection refused (111)
```

看到这里你会不会觉得很奇怪？不是说好 UDP 是“无连接”的协议吗？不是说好 UDP 客户端只会阻塞在 recvfrom 这样的调用上吗？怎么这里冒出一个“Connection refused”的错误呢？

别着急，下面就跟着我的思路慢慢去解开这个谜团。

UDP connect 的作用

从前面的例子中，你会发现，我们可以对 UDP 套接字调用 connect 函数，但是和 TCP connect 调用引起 TCP 三次握手，建立 TCP 有效连接不同，UDP connect 函数的调用，

并不会引起和服务端目标端的网络交互，也就是说，并不会触发所谓的“握手”报文发送和应答。

那么对 UDP 套接字进行 connect 操作到底有什么意义呢？

其实上面的例子已经给出了答案，这主要是为了让应用程序能够接收“异步错误”的信息。

如果我们回想一下第 6 篇不调用 connect 操作的客户端程序，在服务器端不开启的情况下，客户端程序是不会报错的，程序只会阻塞在 recvfrom 上，等待返回（或者超时）。

在这里，我们通过对 UDP 套接字进行 connect 操作，将 UDP 套接字建立了“上下文”，该套接字和服务端端的地址和端口产生了联系，正是这种绑定关系给了操作系统内核必要的信息，能够将操作系统内核收到的信息和对应的套接字进行关联。

我们可以展开讨论一下。

事实上，当我们调用 sendto 或者 send 操作函数时，应用程序报文被发送，我们的应用程序返回，操作系统内核接管了该报文，之后操作系统开始尝试往对应的地址和端口发送，因为对应的地址和端口不可达，一个 ICMP 报文会返回给操作系统内核，该 ICMP 报文含有目的地址和端口等信息。

如果我们不进行 connect 操作，建立（UDP 套接字——目的地址 + 端口）之间的映射关系，操作系统内核就没有办法把 ICMP 不可达的信息和 UDP 套接字进行关联，也就没有办法将 ICMP 信息通知给应用程序。

如果我们进行了 connect 操作，帮助操作系统内核从容建立了（UDP 套接字——目的地址 + 端口）之间的映射关系，当收到一个 ICMP 不可达报文时，操作系统内核可以从映射表中找出是哪个 UDP 套接字拥有该目的地址和端口，别忘了套接字在操作系统内部是全局唯一的，当我们在该套接字上再次调用 recvfrom 或 recv 方法时，就可以收到操作系统内核返回的“Connection Refused”的信息。

收发函数

在对 UDP 进行 connect 之后，关于收发函数的使用，很多书籍是这样推荐的：

使用 send 或 write 函数来发送，如果使用 sendto 需要把相关的 to 地址信息置零；

使用 `recv` 或 `read` 函数来接收，如果使用 `recvfrom` 需要把对应的 `from` 地址信息置零。

其实不同的 UNIX 实现对此表现出来的行为不尽相同。


在我的 Linux 4.4.0 环境中，使用 `sendto` 和 `recvfrom`，系统会自动忽略 `to` 和 `from` 信息。在我的 macOS 10.13 中，确实需要遵守这样的规定，使用 `sendto` 或 `recvfrom` 会得到一些奇怪的结果，切回 `send` 和 `recv` 后正常。

考虑到兼容性，我们也推荐这些常规做法。所以在接下来的程序中，我会使用这样的做法来实现。

服务器端 `connect` 的例子

一般来说，服务器端不会主动发起 `connect` 操作，因为一旦如此，服务器端就只能响应一个客户端了。不过，有时候也不排除这样的情形，一旦一个客户端和服务端发送 UDP 报文之后，该服务器端就要服务于这个唯一的客户端。

一个类似的服务器端程序如下：

 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 int main(int argc, char **argv) {
11     int socket_fd;
12     socket_fd = socket(AF_INET, SOCK_DGRAM, 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     bind(socket_fd, (struct sockaddr *) &server_addr, sizeof(server_addr));
21
```

```

22  socklen_t client_len;
23  char message[MAXLINE];
24  message[0] = 0;
25  count = 0;
26
27  signal(SIGINT, recvfrom_int);
28
29  struct sockaddr_in client_addr;
30  client_len = sizeof(client_addr);
31
32  int n = recvfrom(socket_fd, message, MAXLINE, 0, (struct sockaddr *) &client_addr, {
33  if (n < 0) {
34      error(1, errno, "recvfrom failed");
35  }
36  message[n] = 0;
37  printf("received %d bytes: %s\n", n, message);
38
39  if (connect(socket_fd, (struct sockaddr *) &client_addr, client_len)) {
40      error(1, errno, "connect failed");
41  }
42
43  while (strncmp(message, "goodbye", 7) != 0) {
44      char send_line[MAXLINE];
45      sprintf(send_line, "Hi, %s", message);
46
47      size_t rt = send(socket_fd, send_line, strlen(send_line), 0);
48      if (rt < 0) {
49          error(1, errno, "send failed ");
50      }
51      printf("send bytes: %zu \n", rt);
52
53      size_t rc = recv(socket_fd, message, MAXLINE, 0);
54      if (rc < 0) {
55          error(1, errno, "recv failed");
56      }
57
58      count++;
59  }
60
61  exit(0);
62 }

```

我对这个程序做下解释：

11-12 行创建 UDP 套接字；

14-18 行创建 IPv4 地址，绑定到 ANY 和对应端口；

20 行绑定 UDP 套接字和 IPv4 地址;

27 行为该程序注册一个信号处理函数, 以响应 Ctrl+C 信号量操作;


32-37 行调用 `recvfrom` 等待客户端报文到达, 并将客户端信息保持到 `client_addr` 中;

39-41 行调用 `connect` 操作, 将 UDP 套接字和客户端 `client_addr` 进行绑定;

43-59 行是程序的主体, 对接收的信息进行重新处理, 加上“Hi”前缀后发送给客户端, 并持续不断地从客户端接收报文, 该过程一直持续, 直到客户端发送“goodbye”报文为止。

注意这里所有收发函数都使用了 `send` 和 `recv`。

接下来我们实现一个 `connect` 的客户端程序:

 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: udpclient3 <IPaddress>");
7     }
8
9     int socket_fd;
10    socket_fd = socket(AF_INET, SOCK_DGRAM, 0);
11
12    struct sockaddr_in server_addr;
13    bzero(&server_addr, sizeof(server_addr));
14    server_addr.sin_family = AF_INET;
15    server_addr.sin_port = htons(SERV_PORT);
16    inet_pton(AF_INET, argv[1], &server_addr.sin_addr);
17
18    socklen_t server_len = sizeof(server_addr);
19
20    if (connect(socket_fd, (struct sockaddr *) &server_addr, server_len)) {
21        error(1, errno, "connect failed");
22    }
23
24    char send_line[MAXLINE], recv_line[MAXLINE + 1];
25    int n;
26
27    while (fgets(send_line, MAXLINE, stdin) != NULL) {
28        int i = strlen(send_line);
29        if (send_line[i - 1] == '\n') {
30            send_line[i - 1] = 0;
31        }
```

```

32
33     printf("now sending %s\n", send_line);
34     size_t rt = send(socket_fd, send_line, strlen(send_line), 0);
35     if (rt < 0) {
36         error(1, errno, "send failed ");
37     }
38     printf("send bytes: %zu \n", rt);
39
40     recv_line[0] = 0;
41     n = recv(socket_fd, recv_line, MAXLINE, 0);
42     if (n < 0)
43         error(1, errno, "recv failed");
44     recv_line[n] = 0;
45     fputs(recv_line, stdout);
46     fputs("\n", stdout);
47 }
48
49 exit(0);
50 }

```

我对这个客户端程序做一下解读：

9-10 行创建了一个 UDP 套接字；

12-16 行创建了一个 IPv4 地址，绑定到指定端口和 IP；


20-22 行调用 connect 将 UDP 套接字和 IPv4 地址进行了“绑定”；

27-46 行是程序的主体，读取标准输入字符串后，调用 send 发送给对端；之后调用 recv 等待对端的响应，并把对端响应信息打印到标准输出。

注意这里所有收发函数也都使用了 send 和 recv。

接下来，我们先启动服务器端程序，然后依次开启两个客户端，分别是客户端 1、客户端 2，并且让客户端 1 先发送 UDP 报文。

服务器端：


 复制代码

```

1 $ ./udpconnectserver
2 received 2 bytes: g1
3 send bytes: 6


```


客户端 1:

 复制代码

```
1 ./udpconnectclient2 127.0.0.1
2 g1
3 now sending g1
4 send bytes: 2
5 Hi, g1
```

客户端 2:

 复制代码

```
1 ./udpconnectclient2 127.0.0.1
2 g2
3 now sending g2
4 send bytes: 2
5 recv failed: Connection refused (111)
```

我们看到，客户端 1 先发送报文，服务端随之通过 connect 和客户端 1 进行了“绑定”，这样，客户端 2 从操作系统内核得到了 ICMP 的错误，该错误在 recv 函数中返回，显示了“Connection refused”的错误信息。

性能考虑

一般来说，客户端通过 connect 绑定服务端的地址和端口，对 UDP 而言，可以有一定程度的性能提升。

这是为什么呢？

因为如果不使用 connect 方式，每次发送报文都会需要这样的过程：

连接套接字→发送报文→断开套接字→连接套接字→发送报文→断开套接字 →.....

而如果使用 connect 方式，就会变成下面这样：

连接套接字→发送报文→发送报文→.....→最后断开套接字

我们知道，连接套接字是需要一定开销的，比如需要查找路由表信息。所以，UDP 客户端程序通过 connect 可以获得一定的性能提升。

总结

在今天的内容里，我对 UDP 套接字调用 connect 方法进行了深入的分析。之所以对 UDP 使用 connect，绑定本地地址和端口，是为了让我们的程序可以快速获取异步错误信息的通知，同时也可以获得一定性能上的提升。

思考题

在本讲的最后，按照惯例，给大家留两个思考题：

1. 可以对一个 UDP 套接字进行多次 connect 操作吗？你不妨动手试试，看看结果。
2. 如果想使用多播或广播，我们应该怎么去使用 connect 呢？

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，一起交流一下。

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 13 | 小数据包应对之策：理解TCP协议中的动态数据传输

精选留言 (3)

写留言



Liam

2019-09-02

按照老师的说法，只有connect才建立socket和ip地址的映射；那么，如果不进行connect，收到信息后内核又是如何把数据交给对应的socket呢

展开 ∨

2

1



传说中的成大大

2019-09-02

udp 连接套接字 这个是什么过程？ 断开套接字这又是什么过程呢？



酱油君

2019-09-02

老师 文中提到的icmp报文 发源地 是本机的网卡吗？

展开

