一般使用情况下，UDP网络通信的客户端不需要显示的去bind指定ip、port，交给内核进行分配即可，因为一般客户端不需要知道自己的本地的地址信息（也同样适用于TCP客户端）。但是，在客户端程序中bind也是可以使用的。

另外，TCP客户端需要在创建套接字之必须调用connect()函数连接到服务器，之后在发送数据。对于UDP客户端，也同样可以选择使用connect()函数，这涉及到了性能考量。

**1、UDP客户端使用bind()函数**

以前面的UDP客户端访问echo服务器程序为例，我们在使用sendto发送数据前，先bind()到本地的ip、port上。这里仅给出main()函数修改部分代码。

Int main()

{

/// 1、创建socket

int socket\_fd = ::socket(AF\_INET, SOCK\_DGRAM, 0); // udp

… //省略

// 绑定本地地址

sockaddr\_in localaddr;

localaddr.sin\_family = AF\_INET;

localaddr.sin\_addr.s\_addr = inet\_addr("127.0.0.1");

localaddr.sin\_port = htons(9000);

int ret = ::bind(socket\_fd, (const sockaddr \*)&localaddr, sizeof(localaddr));

if (ret == -1){

printf("%s: bind failed. %s \n", \_\_func\_\_,strerror(errno));

return 1;

}

else{

printf("%s: bind success.\n", \_\_func\_\_);

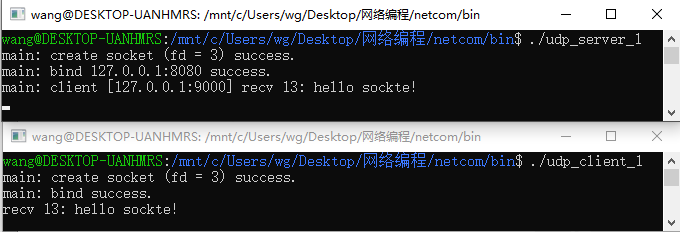
}

// 2、 发送到服务端

… //省略

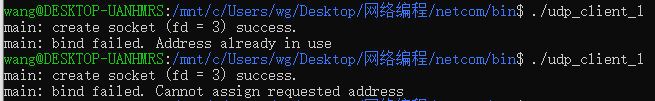
}

结果如下图所示。



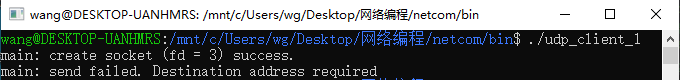
这里是一个成功的示例，绑定在本地ip地址“172.0.0.1”和端口9000上。

但是实际使用中，我们去指定端口会可能出现端口已经被占用导致出错，而需要修改逻辑（尽管可以使用端口服用解决）。例如，绑定在一个已经占用的端口(已经运行的UDP服务端口8080)上、或者不存在的IP地址（如"127.0.0.256"）上，报错分别如下



**2、UDP客户端使用connect()函数**

首先，如果客户端不调用带有目的地址参数的sendto函数，使用weite、send或者无目的地址参数的sendto，程序是无法知道数据需要发送到哪里，导致发送失败。



TCP客户端使用对套接字调用connect()函数会执行三次握手的连接流程，而UDP的connect函数只检查是否存在立即可知的错误（例如一个显然不可达的目的地）

在BSD中文，根据UDP客户端是否使用了connect()成功返回，并记录对端的ip和port，并立即返回结果。根据connect()成功返回与否，有

* **未连接的UDP套接字(unconnected UDP socket)** 新创建UDP套接字默认
* **已连接的UDP套接字(connected UDP socket)** 对UDP套接字调用connect()的结果

对于已连接的UDP套接字，发生了三个变化：

（1）不能在使用带有目的地址的参数的sendto函数发送数据，而需要使用write、send以及不带目的地之的sendto函数，因为任何写到已连接UDP套接字上的任何内容，都自动发送到有connect函数指定的协议地址上。

（2）必须要使用recvfrom以获取数据报的发送者，因为在当前已连接的套接字上在内核中仅接收来自connect后的协议地址数据。因此，也限制了一个已连接的UDP套接字仅能与一个对端交换数据报（确切说是一个IP地址，因为多播或广播是可能的）。

（3）由已连接的UDP套接字引发异步异常会返回给所在进程，而未连接UDP套接字不会接收任何异步错误。

这里修改UDP客户端，添加connect代码段，并设置循环发送接收数据。Main()函数代码如下

int main()

{

/// 1、创建socket

int socket\_fd = ::socket(AF\_INET, SOCK\_DGRAM, 0); // udp

if(socket\_fd == -1){

printf("%s: create socket failed.\n", \_\_func\_\_);

return -1;

}else{

printf("%s: create socket (fd = %d) success.\n", \_\_func\_\_, socket\_fd);

}

// 2、 发送到服务端

sockaddr\_in servaddr;

servaddr.sin\_family = AF\_INET;

servaddr.sin\_addr.s\_addr = inet\_addr("172.0.0.1");

servaddr.sin\_port = htons(8080);

//增加connect()功能

int ret = ::connect(socket\_fd, (const sockaddr\*)&servaddr, sizeof(servaddr));

if(ret < 0) {

printf("%s: connect failed. %s \n", \_\_func\_\_, strerror(errno));

return 0;

}else{

printf("%s: connect %s:%d success.\n", \_\_func\_\_, SRV\_ADDR, SRV\_PORT);

}

int len ;

char buf[1024] = "hello sockte!";

while(1)

{

// 发送(已连接的UDP套接字)

//len = ::write(socket\_fd, buf, strlen(buf));

//len = ::recv(socket\_fd, buf, strlen(buf), 0); //同上

len = ::sendto(socket\_fd, buf, strlen(buf), 0, NULL, NULL); //同上

// 发送（未连接的UDP套接字，不调用connect）

//len = ::sendto(socket\_fd, buf, strlen(buf), 0, (sockaddr\*)&servaddr, sizeof(servaddr));

if(len < 0){

printf("%s: send failed. %s \n", \_\_func\_\_, strerror(errno));

return 1;

}

// 接收

len = ::read(socket\_fd, buf, strlen(buf));

//len = ::recv(socket\_fd, buf, strlen(buf), 0);

if(len < 0){

printf("%s: recv failed. %s \n", \_\_func\_\_, strerror(errno));

return 1;

}

else{

printf("recv %2d: %s\n", len, buf);

}

sleep(1);

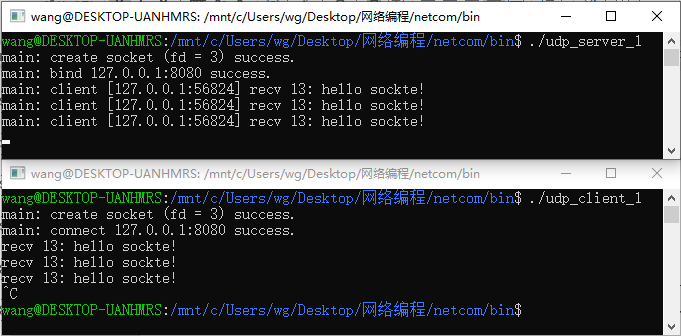
}

// 3、关闭退出

::close(socket\_fd);

}

结果如下



**3、UDP客户端使用已连接UDP套接字性能**

当应用程序在使用一个未连接的UDP套接字上调用sento时，源自berkeley的内核会暂时连接该套接字，发送数据，然后断开连接。那么扩展开来的意思是，在一个未连接的UDP套接字上多次调用sendto发送数据时，会重复进行三个步骤（连接套接字、发送数据、关闭套接字）。

因此，当程序知道要给同一地址发送多个数据时，显示连接套接字效率高，经过connect后多次发送数据，内核仅最开始连接套接字，中间多次发送数据，最后关闭套接字。所以，这种情况下的开销是要小得多的。

在前面提到 ”对于已连接的UDP套接字，不能在使用带有目的地址的参数的sendto函数发送数据“是BSD的规定，实测在**wsl ubuntu18.04中对已连接的UDP套接字是仍然能够使用带目的参数的sendto函数的**。

对于我们实际需求、测试情况，选择是否需要connect使用已连接的UDP套接字。例如，有多个对端要发送数据，建议选择未连接的UDP套接字；而明确一定时间内有大量发送需求到同一个协议地址，建议选择已连接的UDP套接字。