Linux-Socket(TCP 阻塞方式)的使用

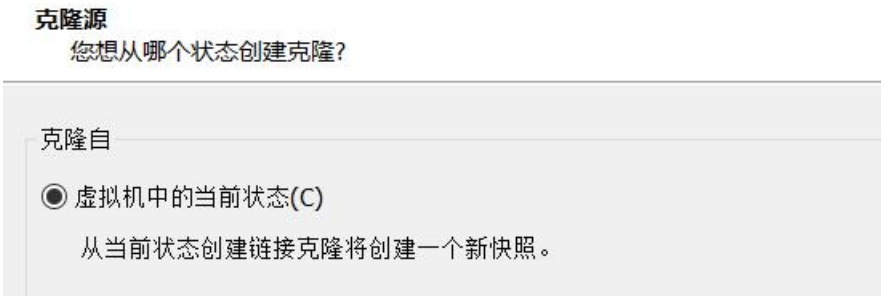
1650373 李子中

0. 补充知识

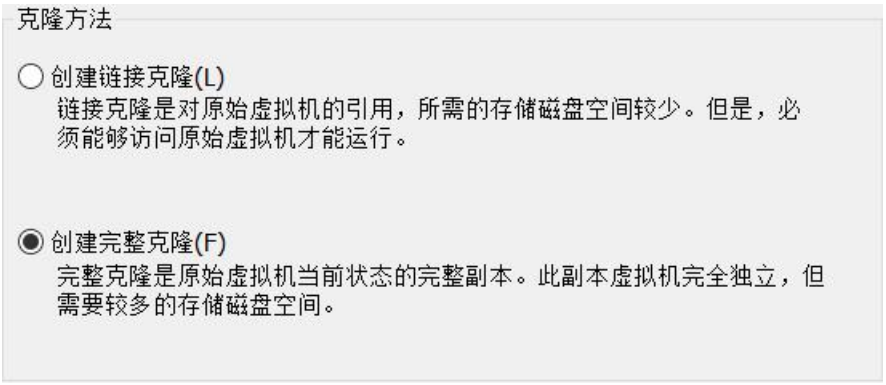
•克隆虚拟机，设置新的虚拟机网卡使其生效

1）克隆虚拟机

在 VMWare 的虚拟机列表中选中要克隆的虚拟机，右键，选择“管理”，选择“克隆”，弹出如下对话框，选择克隆自“虚拟机中的当前状态”



选择“创建完整克隆”



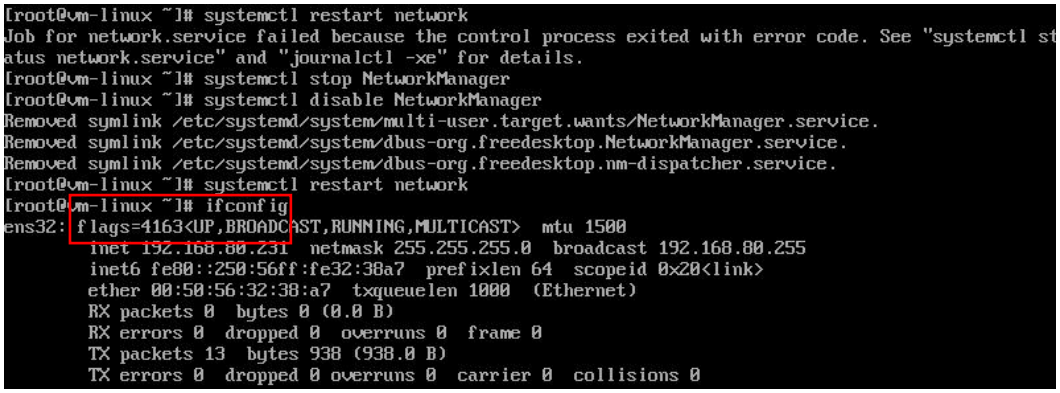
克隆出 CentOS 64 clone

2）重新生成 CenrtOS 64 clone 的 MAC 地址，如下 MAC：00:50:56:32:38:a7

3）进入 clone 机的 ifcfg-ens32 配置文件，将其 ip 地址修改为 192.168.80.231

在最下一行添加 MAC(HWADDR)地址

补充：重启克隆机后依旧连不上网，查阅资料后发现是跟系统自带的 NetworkManager 这个管理套件有关系，关掉就可以解决。



4）使用指令 ping 192.168.80.230，发现两个地址可以相互 ping 通

```
[root@vm-linux ~]# ping 192.168.80.230
PING 192.168.80.230 (192.168.80.230) 56(84) bytes of data.
64 bytes from 192.168.80.230: icmp_seq=1 ttl=64 time=0.489 ms
64 bytes from 192.168.80.230: icmp_seq=2 ttl=64 time=0.358 ms
64 bytes from 192.168.80.230: icmp_seq=3 ttl=64 time=0.386 ms
64 bytes from 192.168.80.230: icmp_seq=4 ttl=64 time=0.297 ms
64 bytes from 192.168.80.230: icmp_seq=5 ttl=64 time=0.238 ms
64 bytes from 192.168.80.230: icmp_seq=6 ttl=64 time=0.292 ms
^C
--- 192.168.80.230 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5002ms
rtt min/avg/max/mdev = 0.238/0.343/0.489/0.082 ms
```

- **在一个网卡上设置多个地址（多地址属于不同网段）**

打开网卡 ens32 的配置文件 /etc/sysconfig/network-scripts/ifcfg-ens32，将原 IPADDR 改为 IPADDR0，PREFIX 改为 PREFIX0。并新增 IPADDR1（新 IP 地址）和 PREFIX1

设置一台虚拟机的新增 IP 地址为 172.18.12.230/24

```
IPADDR0=192.168.80.230
PREFIX0=24
IPV6_PRIVACY=no
########################
IPADDR1=172.18.12.230
PREFIX1=24
```

另一台为 172.18.12.231/24

```
IPADDR0=192.168.80.231
PREFIX0=24
IPV6_PRIVACY=no
HWADDR=00:50:56:32:38:a7
#########################
IPADDR1=172.18.12.231
PREFIX1=24
```

两个同网段新增地址互能 ping 通

```
[root@vm-linux network-scripts]# ping 172.18.12.230
PING 172.18.12.230 (172.18.12.230) 56(84) bytes of data.
64 bytes from 172.18.12.230: icmp_seq=1 ttl=64 time=0.444 ms
64 bytes from 172.18.12.230: icmp_seq=2 ttl=64 time=0.313 ms
64 bytes from 172.18.12.230: icmp_seq=3 ttl=64 time=0.318 ms
64 bytes from 172.18.12.230: icmp_seq=4 ttl=64 time=0.312 ms
^C
--- 172.18.12.230 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3001ms
rtt min/avg/max/mdev = 0.312/0.346/0.444/0.060 ms

[root@vm-linux network-scripts]# ping 172.18.12.231
PING 172.18.12.231 (172.18.12.231) 56(84) bytes of data.
64 bytes from 172.18.12.231: icmp_seq=1 ttl=64 time=0.205 ms
64 bytes from 172.18.12.231: icmp_seq=2 ttl=64 time=0.345 ms
64 bytes from 172.18.12.231: icmp_seq=3 ttl=64 time=0.309 ms
64 bytes from 172.18.12.231: icmp_seq=4 ttl=64 time=0.323 ms
^C
--- 172.18.12.231 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3001ms
rtt min/avg/max/mdev = 0.205/0.295/0.345/0.056 ms
```

1. 写一对 TCP Socket 测试程序，分为 client 和 server，分别运行在不同的虚拟机上

1）Socket 介绍

Linux 中的网络编程是通过 socket 接口来进行的，为 TCP/IP 协议设计的应用层编程接口成为 socket API。在 TCP/IP 协议中，“IP 地址+TCP 或 UDP 端口号”唯一的标识网络通讯中的一个进程，“IP 地址+TCP 或 UDP 端口号”就是 socket。

在 TCP 协议中，建立连接的两个进程——客户端和服务器，各自有一个 socket 标识，则这两个 socket 组成的 socket pair 就唯一标识一个连接。

2）Socket 实现的基本函数

●创建套接字 socket

用于打开一个网络通讯接口，出错返回-1，成功返回一个 socket，应用进程就可调用 read/write 在网络上收发数据。

```
int socket(int domain,int type,int protocol);
//domain:该参数一般被设置为 AF_INET，表示使用的是 IPv4 地址。
//type:该参数也有很多选项，例如 SOCK_STREAM 表示面向流的传输协议，SOCK_DGRAM 表示数据报，我们这里实现的是 TCP，因此选用 SOCK_STREAM。
//protocol:协议类型，一般使用默认，设置为 0
```

●绑定 bind

服务器所监听的网络地址和端口号一般是固定不变的，客户端程序得知服务器程序

地址和端口号后就可以向服务器发起连接，因此服务器需要调用 bind 来绑定一个固定的网络地址和端口号，即将 sockfd（用于网络通讯的文件描述符）和 addr（所描述的地址和端口号）绑定在一起，成功返回 0，出错返回 1.

```
int bind(int sockfd,const struct sockaddr*addr,socklen_t addrlen);
//sockfd：服务器打开的 sock
```

●监听 listen

监听函数仅在服务端使用，listen 声明 sockfd 处于监听状态，并最多允许有 backlog 个客户端处于连接等待状态，若收到更多的连接请求则忽略。listen 成功返回 1，失败返回-1.

```
int listen(int sockfd,int backlog);
//sockfd 的含义与 bind 中的相同。
//backlog 参数解释为内核为次套接口排队的最大数量，这个大小一般为 5~10，不宜太大
```

●接收连接 accept

典型的服务器程序是可以同时服务多个客户端的。当客户端发起连接时，服务器就调用 accept 返回并接收这个连接。如果有大量客户端发起请求而服务器来不及处理，还没有 accept 的客户端就处于连接等待状态。

三次握手完成后，服务器调用 accept 接收连接。如果服务器调用 accept 时还没有客户端的连接请求，就阻塞等待直到有客户端连接上来。

```
int accept(int sockfd,struct sockaddr* addr,socklen_t* addrlen);
//addrlen 是一个传入传出型参数，传入的是调用者的缓冲区 cliaddr 的长度，以避免缓冲区溢出问题；
//传出的是客户端地址结构体的实际长度（有可能没有占满调用者提供的缓冲区）。如果给 cliaddr 参数传 NULL，表示不关心客户端的地址。
```

●请求连接 connect

函数 connect 只需由客户端来调用，调用该函数后表明连接服务器。

```
int connect(int sockfd,const struct sockaddr* addr,socklen_t addrlen);
```

3）客户端程序和服务器程序建立连接的过程

服务器：首先调用 socket（）创建一个套接字用来通讯，其次调用 bind（）对套接字进行绑定，并调用 listen（）来监听端口是否有客户端请求。如果有，调用 accept（）进行连接，否则就继续阻塞式等待直到有客户端连接上来，建立连接后就可以开始通信。

客户端：调用 socket（）分配一个用来通讯的端口，接着调用 connect（）发出 SYN 请求并处于阻塞等待服务器应答状态。客户端收到服务器应答的 SYN-ACK 分段后从 connect（）返回，同时应答一个 ACK 分段，服务器收到后从 accept（）返回，连接建立成功。

4）实现代码

```
//server.c
#include<stdio.h>
```

```c
#include<stdlib.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<unistd.h>
#include<netinet/in.h>
#include<arpa/inet.h>


//_port:端口  _ip:可使用的主机 ip 地址
int startup(int _port, const char *_ip)
{
    //创建 socket
    int sock = socket(AF_INET, SOCK_STREAM, 0);

    //错误返回
    if (sock < 0)
    {
        perror("socket");
        exit(1);
    }

    int opt = 1;
    setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));

    //以 socket_in 结构体填充 socket 信息
    struct sockaddr_in server_sockaddr;
    server_sockaddr.sin_family = AF_INET; //IPV4
    server_sockaddr.sin_port = htons(_port);
    server_sockaddr.sin_addr.s_addr = inet_addr(_ip);
    socklen_t len = sizeof(server_sockaddr);

    //bind 绑定
    if (bind(sock,(struct sockaddr*)&server_sockaddr,len) < 0)
    {
        perror("bind");
        exit(2);
    }

    if (listen(sock, 5) < 0) //允许连接的最大数量为 5
    {
        perror("listen");
        exit(3);
    }
```

```
    return sock;
}

int main(int argc, const char *argv[])
{
    if (argc != 2)
    {
        printf("Usage:%s [local_port]\n", argv[0]);
        return 1;
    }

    //初始化
    int listen_sock = startup(atoi(argv[1]), "192.168.80.230");

    //用来接收客户端的 socket 地址结构体
    struct sockaddr_in remote;
    socklen_t len = sizeof(struct sockaddr_in);

    while (1)
    {
        int sock = accept(listen_sock,(struct sockaddr*)&remote, &len);
        if (sock < 0)
        {
            perror("accept");
            continue;
        }

        //每次建立一个连接后 fork 出一个子进程进行收发数据
        pid_t pid = fork();
        if (pid > 0)
        {
            close(sock);
            while (waitpid(-1, NULL, WNOHANG) > 0);
        }
        else if (pid == 0)
        {
            printf("get a
client,ip:%s,port:%d\n",inet_ntoa(remote.sin_addr),ntohs(remote.sin_port));
            if (fork() > 0)
                exit(0);
            close(listen_sock);
            char buf[1024];
            while (1)
            {
```

```
                ssize_t _s = read(sock, buf, sizeof(buf) - 1);
                if (_s > 0)
                {
                    buf[_s] = 0;
                    printf("client:%s", buf);
                }
                else
                {
                    printf("client is quit!\n");
                    break;
                }
            }
        }
        else
        {
            perror("fork");
            return 2;
        }
    }
    return 0;
}

//client.c
#include<stdio.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<stdlib.h>
#include<netinet/in.h>
#include<arpa/inet.h>
int main(int argc, const char* argv[])
{
    if (argc != 3)
    {
        printf("Usgae:%s [ip] [port]\n", argv[0]);
        return 0;
    }

    //创建一个用来通讯的 socket
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0)
    {
        perror("socket");
        return 1;
    }
```

```
    //需要 connect 的是对端地址，因此这里定义服务器端的地址结构体
    struct sockaddr_in server;
    server.sin_family = AF_INET;
    server.sin_port = htons(atoi(argv[2]));
    server.sin_addr.s_addr = inet_addr(argv[1]);
    socklen_t len = sizeof(struct sockaddr_in);

    if (connect(sock, (struct sockaddr*)&server, len) < 0)
    {
        perror("connect");
        return 2;
    }
    //连接成功进行收数据
    char buf[1024];
    while (1)
    {
        printf("send###");
        fflush(stdout);

        ssize_t _s = read(0, buf, sizeof(buf) - 1);
        buf[_s] = 0;
        write(sock, buf, _s);
    }
    close(sock);
    return 0;
}
```

5）通信

●测试 tcp_server1

执行指令 ./tcp_server1 2021 绑定主机（服务器）IP 地址和端口号 2021，进入等待连接状态（listen）

```
[root@vm-linux 01]# ./tcp_server1 2021
```

开启服务器 IP 地址的新会话，再次执行指令，bind()函数提示地址已被占用，即出错位置在 bind（）函数

```
[root@vm-linux 01]# ./tcp_server1 2021
bind: Address already in use
```

●测试 tcp_client1

执行指令./tcp_client1 192.168.80.232（不存在的 IP 地址），connect()函数报错，没有路径

```
[root@vm-linux d1650373]# ./tcp_client1 192.168.80.232 2021
connect: No route to host
```

执行指令./tcp_client 192.168.80.230 2022（错误的端口号），connect()函数报错，拒绝访问

```
[root@vm-linux d1650373]# ./tcp_client1 192.168.80.230 2022
connect: Connection refused
```

- **连接成功**

服务器执行 ./tcp_server1 2021 开启监听，客户端执行./tcp_client1 192.168.80.230 2021 连接服务器，连接成功后服务器进入 read（recv）状态：

```
[root@vm-linux 01]# ./tcp_server1 2021
get a client,ip:192.168.80.231,port:5513
```

客户端进入 read（recv）状态：

```
[root@vm-linux d1650373]# ./tcp_client1 192.168.80.230 2021
send###
```

用 CTRL+C 中断 client 端，server 端可以检测到连接中断，server 端的 accept（）函数报错

```
accept: Bad file descriptor
accept: Bad file descriptor
accept: Bad file descriptor
accept: Bad file descriptor
accept: Bad file descriptor
accept: Bad file descriptor
accept: Bad file descriptor
accept: Bad accept: Bad file descriptor^C
```

用 CTRL+C 中断 server 端，client 端可以检测到中断，即两次发送失败后退出通信

```
[root@vm-linux d1650373]# ./tcp_client1 192.168.80.230 2021
send###
send###
[root@vm-linux d1650373]# ./tcp_client1 192.168.80.230 2021
```

kill -9 杀死 client 端，server 端可以检测到连接已中断

```
[root@vm-linux 01]# ./tcp_server1 2021
get a client,ip:192.168.80.231,port:5519
client:
client:
client:
client:
client:
client:
client:
client:
client:
client:
client:
已杀死
```

kill -9 杀死 server 端，client 端不会检测到连接已中断

```
[root@vm-linux 01]# ./tcp_server1 2021
get a client,ip:192.168.80.231,port:5525
已杀死
```

```
[root@vm-linux d1650373]# ./tcp_client1 192.168.80.230 2021
send###
send###
send###
send###
send###
send###
send###
```

双方连接成功后，在新的会话中再启动一个 tcp_client1 连接 server，绑定相同端口号，服务器会接收到两个客户端（已设置 SO_REUSEADDR 选项）

```
[root@vm-linux 01]# ./tcp_server1 2021
get a client,ip:192.168.80.231,port:5527
get a client,ip:192.168.80.231,port:5529
```

tcp_server1 运行终止后，立刻再次启动，绑定相同端口号，可以成功（setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt));）

```
[root@vm-linux 01]# ./tcp_server1 2021
^C
[root@vm-linux 01]# ./tcp_server1 2021
^C
[root@vm-linux 01]# ./tcp_server1 2021
get a client,ip:192.168.80.231,port:5535
get a client,ip:192.168.80.231,port:5537
^C
```

- **SO_REUSEADDR 作用：**让一个端口释放后立即就可以被再次使用。（一般来说（不加 SO_REUSEADDR)，一个端口释放后回等待两分钟才能被再次使用）

- 将虚拟机 IP 地址设置为 192.168.80.230/24 和 192.168.80.231/24，VMNet8 网卡设置为 192.168.100.0/24,发现两台虚拟机之间可以相互 ping 通(同一网段下的虚拟机之间可以相互通信，不牵扯到宿主机，与虚拟网卡无关）

```
以太网适配器 VMware Network Adapter VMnet8:

   连接特定的 DNS 后缀 . . . . . :
   本地链接 IPv6 地址. . . . . : fe80::c957:4038:608:5aad%4
   IPv4 地址 . . . . . . . . . . . . : 192.168.100.1
   子网掩码 . . . . . . . . . . . . : 255.255.255.0
   默认网关. . . . . . . . . . . . . :
```

```
[root@vm-linux ~]# ping 192.168.80.230
PING 192.168.80.230 (192.168.80.230) 56(84) bytes of data.
64 bytes from 192.168.80.230: icmp_seq=1 ttl=64 time=0.303 ms
64 bytes from 192.168.80.230: icmp_seq=2 ttl=64 time=0.304 ms
64 bytes from 192.168.80.230: icmp_seq=3 ttl=64 time=0.303 ms
64 bytes from 192.168.80.230: icmp_seq=4 ttl=64 time=0.337 ms
^C
--- 192.168.80.230 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3000ms
rtt min/avg/max/mdev = 0.303/0.311/0.337/0.026 ms
```

```
[root@vm-linux ~]# ping 192.168.80.231
PING 192.168.80.231 (192.168.80.231) 56(84) bytes of data.
64 bytes from 192.168.80.231: icmp_seq=1 ttl=64 time=0.465 ms
64 bytes from 192.168.80.231: icmp_seq=2 ttl=64 time=0.342 ms
64 bytes from 192.168.80.231: icmp_seq=3 ttl=64 time=0.319 ms
64 bytes from 192.168.80.231: icmp_seq=4 ttl=64 time=0.299 ms
64 bytes from 192.168.80.231: icmp_seq=5 ttl=64 time=0.377 ms
^C
--- 192.168.80.231 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4001ms
rtt min/avg/max/mdev = 0.299/0.360/0.465/0.060 ms
```

2. 写一对 TCP Socket 测试程序，要求 client 连接 server 端的时候使用固定端口号，通过 main 函数带参方式传入

修改题 1 的 tcp_client1.c，使用 bind（）函数绑定 client 端端口

```
struct sockaddr_in client;
    client.sin_family = AF_INET;
    client.sin_addr.s_addr = htonl(INADDR_ANY);
    client.sin_port = htons(atoi(argv[1]));
    //绑定 client 端口号
    if(bind(sock,(struct sockaddr*)&client,sizeof(client))<0)
    {
            perror("bind");
        return 3;
    }
```

- 令 client 连接 server 时使用 3000 号端口

```
[root@vm-linux d1650373]# chmod 777 tcp_client2
[root@vm-linux d1650373]# ./tcp_client2 3000 192.168.80.230 2025
send###
send###
send###
```

- server 端接收到 client 端口号 3000 的连接，打印相关信息

```
[root@vm-linux 02]# ./tcp_server2 2025
get a client,ip:192.168.80.231,port:3000
client:
client:
```

3. 写一对 TCP Socket 测试程序，server 打印本机所有 IP 地址，选择要绑定的地址后发起连接

- 指令 ./tcp_server3 showip 可显示本机所有 IP 地址

```
[root@vm-linux 03]# ./tcp_server3 showip
The ips : 127.0.0.1; 192.168.80.230; 172.18.12.230; 192.168.1.232
```

```
//获取 IP 地址
int get_ip()
{
    char ipAddr[MAX_LENGTH];
    ipAddr[0] = '\0';

    struct ifaddrs *ifAddrStruct = NULL;
    void *tmpAddrPtr = NULL;

    if (getifaddrs(&ifAddrStruct) != 0)
    {
        printf("quit getifaddrs\n");
        return -1;
    }

    struct ifaddrs * iter = ifAddrStruct;
    while (iter != NULL)
    {
        if (iter->ifa_addr->sa_family == AF_INET) {
            tmpAddrPtr = &((struct sockaddr_in *)iter->ifa_addr)->sin_addr;
            char addressBuffer[INET_ADDRSTRLEN];
            inet_ntop(AF_INET, tmpAddrPtr, addressBuffer, INET_ADDRSTRLEN);
            if (strlen(ipAddr) + strlen(addressBuffer) < MAX_LENGTH - 1)
            {
                if (strlen(ipAddr) > 0)
                    strcat(ipAddr, "; ");
                strcat(ipAddr, addressBuffer);
            }
            else {
                printf("too many ip\n");
                break;
            }
        }
        iter = iter->ifa_next;
    }
    freeifaddrs(ifAddrStruct);
    printf("The ips : %s\n", ipAddr);

    return 0;
}
```

● 使用 main 传参的方式选择要绑定的 IP 地址和端口号，server 发起连接

```
[root@vm-linux 03]# ./tcp_server3 172.18.12.230 2025
get a client,ip:172.18.12.231,port:7790
client:
client:
```

client 发起连接

```
[root@vm-linux d1650373]# ./tcp_client3 172.18.12.230 2025
send###
send###
send###
```

● tcp_client3 连接未绑定 IP，connect()函数拒绝连接

```
[root@vm-linux d1650373]# ./tcp_client3 192.168.80.230 2025
connect: Connection refused
```

4. 写一对 TCP Socket 测试程序，体会函数 read/recv & write/send 的区别

● 设置 server 的 read 函数一次读 20 字节，在 client 端用 write 函数向其一次写入超过 20 字节的内容

server 端开始连接

```
[root@vm-linux 04]# ./tcp_server4-1 172.18.12.230 2022
get a client,ip:172.18.12.231,port:4865
```

client 端写入

```
[root@vm-linux d1650373]# ./tcp_client4-1-1 172.18.12.230 2022
send###client is sending a message......
send###client is sending a message......
send###
```

server 端接收，每 20 个字节读一次，不满 20 字节不继续读

```
[root@vm-linux 04]# ./tcp_server4-1 172.18.12.230 2022
get a client,ip:172.18.12.231,port:4879
client:client is sending a client:message......
client:client is sending a client:message......
```

● 设置 server 的 read 函数一次读 20 字节，在 client 端每次写入 2 字节再延时 1 秒，再写 2 字节.....

client 端写入

```
[root@vm-linux d1650373]# ./tcp_client4-1-2 172.18.12.230 2022
send###12
send###23
send###34
send###
```

server 端接收，与 client 写入内容一致

```
[root@vm-linux 04]# ./tcp_server4-1 172.18.12.230 2022
get a client,ip:172.18.12.231,port:4865
client:12
client:23
client:34
```

●将 read 换成 recv 函数，write 换成 send 函数，在 client 端用 write 函数向其一次写入超过 20 字节的内容

client 端写入

```
[root@vm-linux d1650373]# ./tcp_client4-2-1 172.18.12.230 2024
send###client is sending a message......
send###client is sending a message.....
send###client is sending a message.....
send###
```

server 端接收，每 20 个字节读一次，不满 20 字节则等待下一次输入

```
[root@vm-linux 04]# ./tcp_server4-2 172.18.12.230 2024
get a client,ip:172.18.12.231,port:3393

client:client is sending a client:message......
clientclient: is sending a messagclient:e.....
client is senclient:ding a message.....
```

●将 read 换成 recv 函数，write 换成 send 函数，在 client 端每次写入 2 字节再延时 1 秒，再写 2 字节.....

client 端写入

```
[root@vm-linux d1650373]# ./tcp_client4-2-1 172.18.12.230 2025
send###ab
send###ab
send###ab
send###ab
send###ab
send###ab
send###ab
send###ab
send###ab
send###ab
send###ab
send###ab
send###ab
send###ab
send###ab
```

server 端接收，将回车记作一个字符，每读入 20 字节才重新进行接收

```
[root@vm-linux 04]# ./tcp_server4-2 172.18.12.230 2025
get a client,ip:172.18.12.231,port:7904
client:ab
ab
ab
ab
ab
ab
abclient:
ab
ab
ab
ab
ab
ab
```

● **read 函数与 recv 函数的区别：recv 函数可以对阻塞/非阻塞信息进行控制**

```
read(sockfd,buff,buff_size);
```

read 原则：数据在不超过设置的指定长度时有多少读多少，没有数据则会一直等待。使用 read 函数需要循环读取数据，因为一次 read 并不能保证已经读到了需要的数据长度，read 完一次需要判断独到的数据长度再确定是否读取下一次。

```
recv(sockfd,buff,buff_size,MSG_WAITALL/MSG_DONTWAIT);
```

可以看出，recv 函数相比于 read 函数多了一个 MSG_WAITALL（阻塞模式接收）/MSG_DONTWAIT（非阻塞模式接收）参数，对阻塞和非阻塞信息进行控制。
使用参数 WAITALL 会让 recv 读取到指定长度的数据

● **wirte/send 函数的区别与 read/recv 函数区别相同**

5. 写一对 TCP Socket 测试程序，对 TCP 收发缓冲区进行测试

● **在 server 的 read 函数前添加 getchar()，进入等待输入状态**

连接成功后，client 端不断向 server 端发送数据，发送到大约 356470 个字节后陷入阻塞

```
send###total byte : 356410
send###total byte : 356420
send###total byte : 356430
send###total byte : 356440
send###total byte : 356450
send###total byte : 356460
send###total byte : 356470
```

server 端接收数据

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaatotal byte : 16368
```

server 端 getchar（）后 client 端可继续写入

```
send###total byte : 486690
send###total byte : 486700
send###total byte : 486710
send###total byte : 486720
send###total byte : 486730
send###total byte : 486740
send###total byte : 486750
send###total byte : 486760
send###total byte : 486770
send###total byte : 486780
send###total byte : 486790
```

●打开新 server 终端，使用 netstat -t 观察（-t：仅显示 tcp 相关选项）

```
[root@vm-linux ~]# netstat -t
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State

tcp   242072      0 vm-linux:scrabble       172.18.12.231:5643      ESTABLISHED
tcp        0     96 vm-linux:ssh            192.168.80.1:53711      ESTABLISHED
tcp        0      0 vm-linux:ssh            192.168.80.1:53598      ESTABLISHED
[root@vm-linux ~]#
[root@vm-linux ~]# netstat -t
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State


tcp   240026      0 vm-linux:scrabble       172.18.12.231:5643      ESTABLISHED
tcp        0     96 vm-linux:ssh            192.168.80.1:53711      ESTABLISHED
tcp        0      0 vm-linux:ssh            192.168.80.1:53598      ESTABLISHED
```

Proto：连接方式 tcp
Recv-Q：收到字节数量
Send-Q：发送字节数量
Local Address：本机地址
Forigen Address：连接到的 client 端的 IP 地址
State：连接状态，ESTABLISHED 代表这是一个打开的连接

●打开新 client 终端，用 netstat -t 观察

```
[root@vm-linux ~]# netstat -t
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0     96 vm-linux:ssh            192.168.80.1:53710      ESTABLISHED
tcp        0      0 vm-linux:ssh            192.168.80.1:53599      ESTABLISHED
tcp        0 228352 vm-linux:5643           172.18.12.230:scrabble  ESTABLISHED
```

【注】server 端的 Recv-Q 字节数与 client 端的 Send-Q 字节数与打印字节数不相等？

●双方角色互换，server 写至阻塞为止，client 开始读，过程是一样的

server 端写入至阻塞

```
send###total byte : 359210
send###total byte : 359220
send###total byte : 359230
send###total byte : 359240
send###total byte : 359250
```

client 端接收

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaatotal byte
 : 19437
```

getchar（）后 server 端继续发送

```
send###total byte : 489530
send###total byte : 489540
send###total byte : 489550
send###total byte : 489560
send###total byte : 489570
send###
```

netstat -t

```
[root@vm-linux ~]# netstat -t
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address             Foreign Address          State
tcp   240119      0 vm-linux:7329             172.18.12.:shadowserver  ESTABLISHED
tcp        0     96 vm-linux:ssh              192.168.80.1:53710       ESTABLISHED
tcp        0      0 vm-linux:ssh              192.168.80.1:54259       ESTABLISHED
tcp        0      0 vm-linux:ssh              192.168.80.1:53599       ESTABLISHED
```

```
[root@vm-linux ~]# netstat -t
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address             Foreign Address          State
tcp        0 230016 vm-linux:shadowserver     172.18.12.231:7329       ESTABLISHED
tcp        0     96 vm-linux:ssh              192.168.80.1:53711       ESTABLISHED
tcp        0      0 vm-linux:ssh              192.168.80.1:53598       ESTABLISHED
```

●改变 TCP 收发缓冲区大小——setsockopt（）函数

setsockopt（）：获取或设置某个套接字关联的选项

```
//设置接收缓冲区大小
setsockopt(sock,SOL_SOCKET,SO_RCVBUF,(char*)&optVal,sizeof(int));
//设置发送缓冲区大小
setsockopt(sock,SOL_SOCKET,SO_SNDBUF,(const char*)&optVal,sizeof(int));
```

改变缓冲区大小后 client 写入至阻塞

```
send###total byte : 2230
send###total byte : 2240
send###total byte : 2250
send###total byte : 2260
send###total byte : 2270
send###total byte : 2280
send###total byte : 2290
send###total byte : 2300
send###total byte : 2310
send###
```

server 端接收

```
[root@vm-linux 05]# ./tcp_server5-3 172.18.12.230 2028
get a client,ip:172.18.12.231,port:3639

client:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaatotal byte : 1023
```

6. 写一对 TCP Socket 的测试程序，分别对 server 与 client 进行 read 和 write 死循环测试

●server 先 read 再 write 死循环，client 先 read 再 write 死循环

两边都在等待写入，进入 read 阻塞状态，无法传送数据

```
[root@vm-linux d1650373]# ./tcp_client6-1 172.18.12.230 2030 1000 1000
```

```
[root@vm-linux 06]# ./tcp_server6-1 172.18.12.230 2030 1000 1000
get a client,ip:172.18.12.231,port:5082
```

●server 先 write 再 read，client 先 write 再 read

1000 字节：读写大小一致，可以正常传输

client 端：

```
aaaaaaaaaatotal byte : 715000server
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaa^C
```

server 端：

```
aatotal byte : 579000client:aaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

1000 500 字节：读写大小一致，可以正常传输

client 端：

```
aatotal byte : 177010server:aaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaa^Z
```

server 端：

```
aaaaaaaaaatotal byte : 39231
aaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

1000 700 字节：按道理来说，server 端与 client 端的读写大小不一致，冗余的数据会堵住一边（700 字节的 client 端），但直至如下截图时，两端还是正常收发的（可能是代码的问题......）

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaatotal byte : 6613070serv
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaa[root@vm-linux d
```

```
total byte : 6641708client:aaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

设置 TCP 收发缓冲区大小后阻塞

```
total byte : 2442client:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

```
aaaaaaaaaaaaaaaaatotal byte : 2580server:aaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaa
```

- server 先 write 再 read，client 先 read 再 write

1000 字节：正常收发

1000 500 字节：正常收发

700 字节：按理说应该阻塞，但出现了和上一问相同的问题

● server 先 read 再 write，client 先 write 再 read

1000 字节：正常收发
1000 500 字节：正常收发
700 字节：按理说应该阻塞，但出现了和上上问相同的问题