socket介绍

1、所谓 socket（套接字），就是`对网络中不同主机上的应用进程之间进行双向通信的端点的抽象`。
2、一个套接字就是网络上进程通信的一端，提供了应用层进程利用网络协议交换数据的机制。从所处的地位来讲，套接字上联应用进程，下联网络协议栈，是应用程序通过网络协议进行通信的接口，是应用程序与网络协议根进行交互的接口。
3、`socket 可以看成是两个网络应用程序进行通信时，各自通信连接中的端点，这是一个逻辑上的概念`。它是网络环境中进程间通信的 API，也是可以被命名和寻址的通信端点，使用中的每一个套接字都有其类型和一个与之相连进程。通信时其中一个网络应用程序将要传输的一段信息写入它所在主机的 socket 中，该 socket 通过与网络接口卡（NIC）相连的传输介质将这段信息送到另外一台主机的 socket 中，使对方能够接收到这段信息。socket 是由 IP 地址和端口结合的，提供向应用层进程传送数据包的机制。
4、在 Linux 环境下，`socket用于表示进程间网络通信的特殊文件类型。本质为内核借助缓冲区形成的伪文件`。既然是文件，那么理所当然的，我们可以使用文件描述符引用套接字。与管道类似的，Linux 系统将其封装成文件的目的是为了统一接口，使得读写套接字和读写文件的操作一致。区别是管道主要应用于本地进程间通信，而套接字多应用于网络进程间数据的传递。
5、socket套接字通信分两部分：
  * 服务器端：被动接受连接，一般不会主动发起连接
  * 客户端：主动向服务器发起连接
socket是一套通信的接口，Linux 和 Windows 都有，但是有一些细微的差别。

字节序

字节序顾名思义是字节的顺序，大于一个字节类型的数据在内存中的存放顺序。如果数据小于等于一个字节不考虑这个问题。

简介

1、现代CPU的累加器一次能装载至少4字节，也就是一个整数。那么这 4字节在内存中排列的顺序将影响它被累加器装载成的整数的值，这就是字节序问题（类似于现代人书写从左往右，古人从右往左）。
2、在各种计算机体系结构中，对于字节、字等的存储机制有所不同，因而引发了计算机通信领域中一个很重要的问题，即通信双方交流的信息单元（比特、字节、字、双字等等）应该以什么样的顺序进行传送。如果打不成一致，通信将是失败的。
3、字节序分为大端字节序（Big-Endian） 和小端字节序（Little-Endian）。`大端字节序是指一个整数的最高位字节（23 ~ 31 bit）存储在内存的低地址处，低位字节（0 ~ 7 bit）存储在内存的高地址处；小端字节序则是指整数的高位字节存储在内存的高地址处，而低位字节则存储在内存的低地址处`。小端字节序符合我们的常规思维，很多计算机也是使用小端字节序。网络中统一使用大端字节序。

字节序转换函数

1、当格式化的数据在两台使用不同字节序的主机之间直接传递时，接收端必然错误的解释之。解决问题的方法是：`发送端总是把要发送的数据转换成大端字节序数据后再发送，而接收端知道对方传送过来的数据总是采用大端字节序，所以接收端可以根据自身采用的字节序决定是否对接收到的数据进行转换（小端机转换，大端机不转换）`。
2、网络字节顺序是 TCP/IP 中规定好的一种数据表示格式，它与具体的 CPU 类型、操作系统等无关，从而可以保证数据在不同主机之间传输时能够被正确解释，`网络字节顺序采用大端排序方式`。
3、BSD Socket提供了封装好的转换接口，方便程序员使用。包括从主机字节序到网络字节序的转换函数：htons、htonl；从网络字节序到主机字节序的转换函数：ntohs、ntohl。

```
字节序函数转换函数
```

```
#include <arpa/inet.h>
/*
函数名分解：
    h - host 主机，主机字节序
    to - 转换成什么
    n - network 网络字节序
    s - short unsigned short（2个字节）
    l - long unsigned int    （4个字节）
*/

//主机字节序 --> 网络字节序
uint32_t htonl(uint32_t hostlong);      //转ip地址
uint16_t htons(uint16_t hostshort);     //转端口port

//网络字节序 --> 主机字节序
uint32_t ntohl(uint32_t netlong);       //转ip地址
uint16_t ntohs(uint16_t netshort);      //转端口port
```

说明：为什么上面两个函数只说转地址和端口，而不转实际传输的数据?
a、端口号和地址需要字节序转换：是因为TCP/IP协议栈要求的，必须要转。
b、数据不需要字节序转换：并不是正真的不需要转化，是因为我们现在使用的都是PC机，它们的主机字节序都是一样的（小端的），所以即使我们的数据在网络传输过程中没有进行字节序转换，对方收到以后也是能够正确的存储的。假如接收的是大端的主机，那么它收到例如中文（两个字节的数据）时就会出错了。要保证两种主机都能正通信，那么数据在传输过程中也一定要进行字节序转换。（注：一个字节的数据（如单个字符）传输无需字节序转换）

使用：在socket编程中，比如，客户端需要连接服务器，那么就需要将服务器的地址和端口封装到socket地址中，此时就需要这两个数据进行字节序的转换，转到网络字节序。一般是转端口，ip地址还要转成网络能接受的数据类型，所以还有其他函数对其作处理（包括了转字节序的功能）。

转换函数示例

```
#include <stdio.h>
#include <arpa/inet.h>

int main() {

    // htons 转换端口
    unsigned short a = 0x0102;
    printf("a : %x\n", a);
    unsigned short b = htons(a);
    printf("b : %x\n", b);

    printf("========================\n");

    // htonl  转换IP
    char buf[4] = {192, 168, 1, 100};
    int num = *(int *)buf;
    int sum = htonl(num);
    unsigned char *p = (char *)&sum;

    printf("%d %d %d %d\n", *p, *(p+1), *(p+2), *(p+3));

    printf("========================\n");

    // ntohl
```

```
    unsigned char buf1[4] = {1, 1, 168, 192};
    int num1 = *(int *)buf1;
    int sum1 = ntohl(num1);
    unsigned char *p1 = (unsigned char *)&sum1;
    printf("%d %d %d %d\n", *p1, *(p1+1), *(p1+2), *(p1+3));

    return 0;
}
```

socket地址

简介

socket地址其实是一个结构体，封装端口号和IP等信息。在socket编程中就需要使用该socket地址。

通用socket地址

socket 网络编程接口中表示 socket 地址的是结构体 `sockaddr`，其定义如下：

```
#include <bits/socket.h>
struct sockaddr {
    sa_family_t sa_family;
    char sa_data[14];
};
typedef unsigned short int sa_family_t;
```

结构体成员介绍：
`sa_family` 成员是地址族类型（sa_family_t）的变量。地址族类型通常与协议族类型对应。常见的协议族（protocol family，也称 domain）和对应的地址族入下所示：

协议族	地址族	描述
PF_UNIX	AF_UNIX	UNIX本地域协议族
PF_INET	AF_INET	TCP/IPv4协议族
PF_INET6	AF_INET6	TCP/IPv6协议族

宏 PF* 和 *AF** 都定义在 bits/socket.h 头文件中，且后者与前者有完全相同的值，所以二者通常混用。
`sa_data` 成员用于存放 socket 地址值。但是，不同的协议族的地址值具有不同的含义和长度，如下所示：

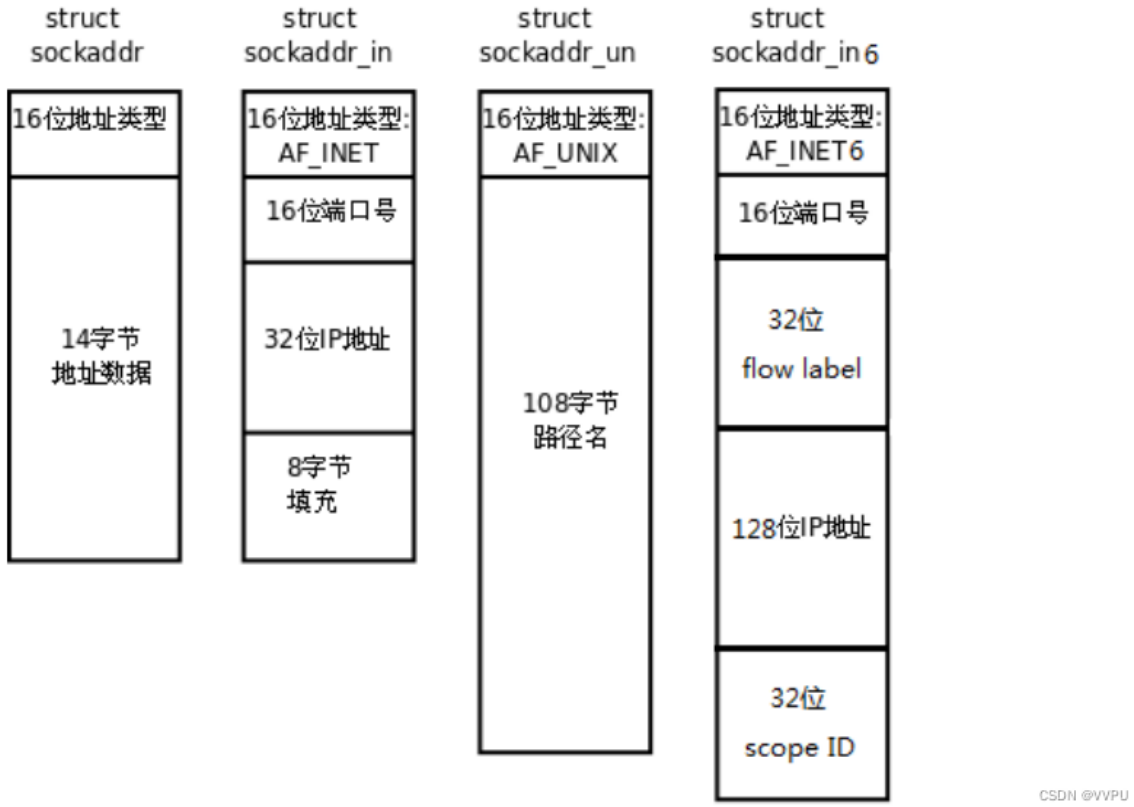
协议族	地址值含义和长度
PF_UNIX	文件的路径名，长度可达到108字节
PF_INET	16 bit 端口号和 32 bit IPv4 地址，共 6 字节
PF_INET6	16 bit 端口号，32 bit 流标识，128 bit IPv6 地址，32 bit 范围 ID，共 26 字节

由上表可知，`14 字节的 sa_data 根本无法容纳多数协议族的地址值`。因此，Linux 定义了下面这个新的通用的 socket 地址结构体，这个结构体不仅提供了足够大的空间用于存放地址值，而且是内存对齐的。

```
#include <bits/socket.h>
struct sockaddr_storage
{
    sa_family_t sa_family;
    unsigned long int __ss_align;
    char __ss_padding[ 128 - sizeof(__ss_align) ];
};
typedef unsigned short int sa_family_t;
```

专用socket地址

很多网络编程函数诞生早于 IPv4 协议，那时候都使用的是 struct sockaddr 结构体，为了向前兼容，现在sockaddr 退化成了（void *）的作用，传递一个地址给函数，至于这个函数是 sockaddr_in 还是 sockaddr_in6，由地址族确定，然后函数内部再强制类型转化为所需的地址类型。



1、UNIX 本地域协议族使用如下专用的 socket 地址结构体：

```
#include <sys/un.h>
struct sockaddr_un
{
    sa_family_t sin_family;
    char sun_path[108];
};
```

2、TCP/IP 协议族有 sockaddr_in 和 sockaddr_in6 两个专用的 socket 地址结构体，它们分别用于 IPv4 和IPv6：

```
#include <netinet/in.h>
struct sockaddr_in
{
    sa_family_t sin_family; /* __SOCKADDR_COMMON(sin_) */
    in_port_t sin_port;     /* Port number. */
    struct in_addr sin_addr; /* Internet address. */
```

```
    /* Pad to size of `struct sockaddr'. */
    unsigned char sin_zero[sizeof (struct sockaddr) - __SOCKADDR_COMMON_SIZE -
sizeof (in_port_t) - sizeof (struct in_addr)];
};
struct in_addr
{
    in_addr_t s_addr;   //uint32_t
};


struct sockaddr_in6
{
    sa_family_t sin6_family;
    in_port_t sin6_port; /* Transport layer port # */
    uint32_t sin6_flowinfo; /* IPv6 flow information */
    struct in6_addr sin6_addr; /* IPv6 address */
    uint32_t sin6_scope_id; /* IPv6 scope-id */
};

//别名
typedef unsigned short uint16_t;
typedef unsigned int uint32_t;
typedef uint16_t in_port_t;
typedef uint32_t in_addr_t;
#define __SOCKADDR_COMMON_SIZE (sizeof (unsigned short int))
```

注意：所有专用 socket 地址（以及 sockaddr_storage）类型的变量在实际使用时都需要`转化为通用 socket 地址类型 sockaddr`（强制转化即可），因为所有 socket 编程接口使用的地址参数类型都是 sockaddr。
通常，在设置socket地址具体数据的时候都是用专用地址（ipv4居多），作为参数传递进入socket函数的时候就强转为通用sockaddr类型。

IP地址转换

简介

通常，人们习惯用可读性好的字符串来表示 IP 地址，比如用点分十进制字符串表示 IPv4 地址，以及用十六进制字符串表示 IPv6 地址。但`编程中我们需要先把它们转化为整数（二进制数）方能使用`。而`记录日志时则相反，我们要把整数表示的 IP 地址转化为可读的字符串`。下面 3 个函数可用于用点分十进制字符串表示的 IPv4 地址和用网络字节序整数表示的 IPv4 地址之间的转换：

```
#include <arpa/inet.h>
in_addr_t inet_addr(const char *cp);
int inet_aton(const char *cp, struct in_addr *inp);
char *inet_ntoa(struct in_addr in);
```

下面这对更新的函数也能完成前面 3 个函数同样的功能，并且它们同时适用 IPv4 地址和 IPv6 地址，下面两函数更加常用：
会对其进行字节序转换。

```
#include <arpa/inet.h>
// p:点分十进制的IP字符串，n:表示network，网络字节序的整数
int inet_pton(int af, const char *src, void *dst);
    af:地址族：  AF_INET AF_INET6
    src:需要转换的点分十进制的IP字符串
    dst:转换后的结果保存在这个里面
// 将网络字节序的整数，转换成点分十进制的IP地址字符串
const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);
    af:地址族：  AF_INET AF_INET6
    src: 要转换的ip的整数的地址
    dst: 转换成IP地址字符串保存的地方
    size：第三个参数的大小（数组的大小）
    返回值：返回转换后的数据的地址（字符串），和 dst 是一样的
```

实例

```
int main()
{
    // int inet_pton(int af, const char *src, void *dst);
    //点分十进制 --> 网络字节序
    char *src = "192.168.43.221";

    void *dst = malloc(sizeof(void *));
    // unsigned short int dst = 0;
    int ret = inet_pton(AF_INET, src, dst);
    perror("ret = ");

    unsigned char *dstV = (unsigned char *)dst;
    printf("%d %d %d %d\n", dstV[0], dstV[1], dstV[2], dstV[3]);

    char dst2[16];
    inet_ntop(AF_INET, dst, dst2, 16); // 表示dst2的长度
    printf("%s\n", dst2);
}
```

TCP通信流程

简介

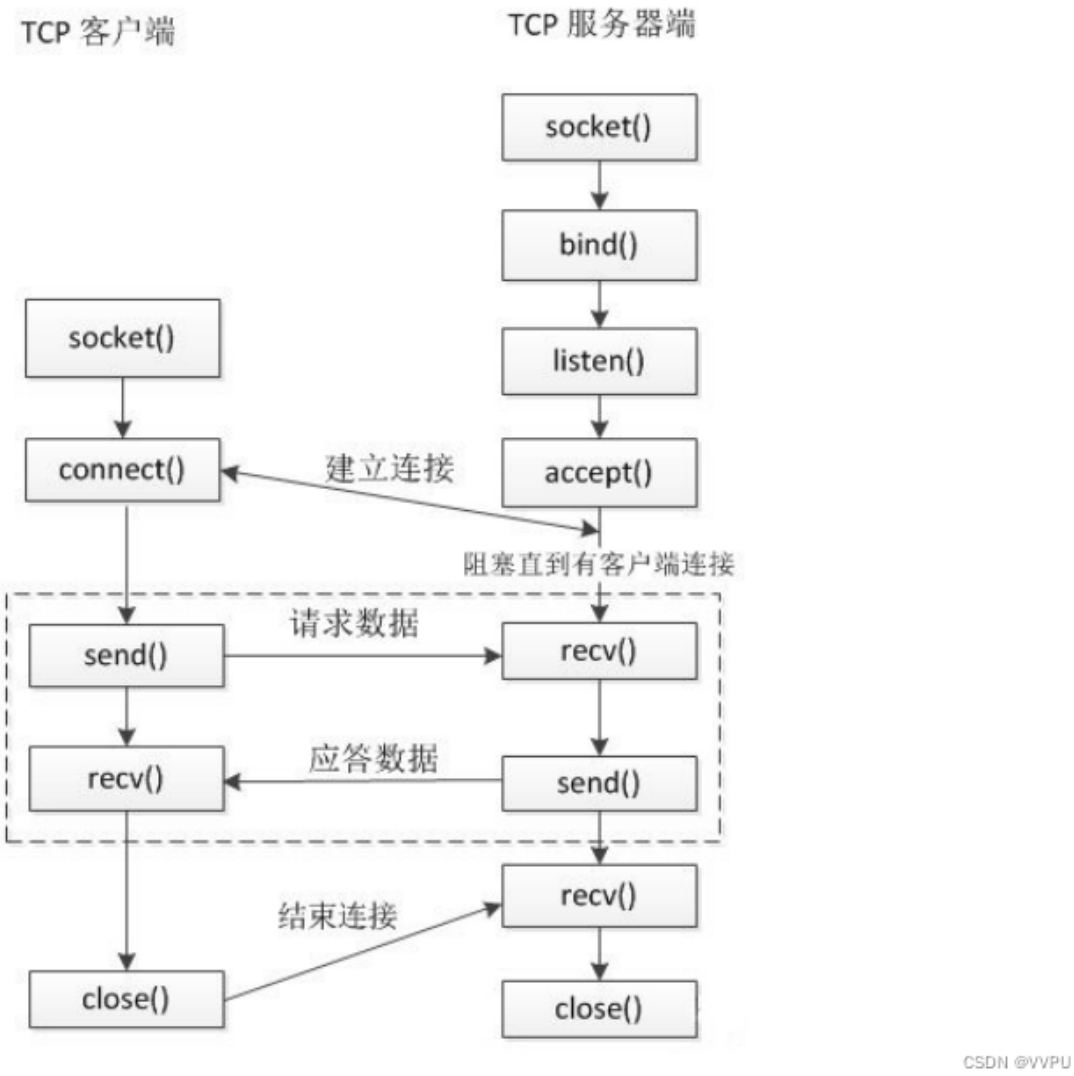
TCP 和 UDP -> 传输层的协议
UDP:用户数据报协议，面向无连接，可以单播，多播，广播， 面向数据报，不可靠
TCP:传输控制协议，面向连接的，可靠的，基于字节流，仅支持单播传输

	UDP	TCP
是否创建连接	无连接	面向连接
是否可靠	不可靠	可靠的
连接的对象个数	一对一、一对多、多对一、多对多	支持一对一
传输的方式	面向数据报	面向字节流
首部开销	8个字节	最少20个字节
适用场景	实时应用（视频会议，直播）	可靠性高的应用（文件传输）


```
// TCP 通信的流程
// 服务器端 （被动接受连接的角色）
1. 创建一个用于监听的套接字
    - 监听：监听有客户端的连接
```

    - 套接字：这个套接字其实就是一个文件描述符
2. 将这个监听文件描述符和本地的IP和端口绑定（IP和端口就是服务器的地址信息）
    - 客户端连接服务器的时候使用的就是这个IP和端口
3. 设置监听，监听的fd开始工作
4. 阻塞等待，当有客户端发起连接，解除阻塞，接受客户端的连接，会得到一个和客户端通信的套接字（fd）
5. 通信
    - 接收数据
    - 发送数据
6. 通信结束，断开连接

// 客户端
1. 创建一个用于通信的套接字（fd）
2. 连接服务器，需要指定连接的服务器的 IP 和 端口
3. 连接成功了，客户端可以直接和服务器通信
    - 接收数据
    - 发送数据
4. 通信结束，断开连接



套接字相关函数

```
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h> // 包含了这个头文件，上面两个就可以省略
```

```
int socket(int domain, int type, int protocol);
    - 功能：创建一个套接字
    - 参数：
        - domain: 协议族
            AF_INET : ipv4
            AF_INET6 : ipv6
            AF_UNIX, AF_LOCAL : 本地套接字通信（进程间通信）
        - type: 通信过程中使用的协议类型
            SOCK_STREAM : 流式协议
            SOCK_DGRAM : 报式协议
        - protocol : 具体的一个协议。一般写0
            - SOCK_STREAM : 流式协议默认使用 TCP
            - SOCK_DGRAM : 报式协议默认使用 UDP
    - 返回值：
    - 成功：返回文件描述符，操作的就是内核缓冲区。
    - 失败：-1，并且设置对应的错误编号
    -该函数相当有获取了一个文件描述符。之后的操作都依赖于这个文件描述符。

int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen); // socket命名
    - 功能：绑定，将fd 和本地的IP + 端口进行绑定。
    - 描述：
        当使用 socket(2) 创建套接字时，它存在于名称空间（地址族）中，但没有分配给它的地址。
        bind() 将 addr 指定的地址分配给文件描述符 sockfd 引用的套接字。
        addrlen 指定 addr 指向的地址结构体的大小（以字节为单位）。 传统上，此操作称为“为套接字分配名称”。
    - 参数：
        - sockfd : 通过socket函数得到的文件描述符
        - addr : 需要绑定的socket地址，这个地址封装了ip和端口号的信息
        - addrlen : 第二个参数结构体占的内存大小
    - 返回值：
        - 成功： 返回 0
        - 失败： 返回 -1， 并且设置对应的错误编号

int listen(int sockfd, int backlog); // 配置文件： /proc/sys/net/core/somaxconn
    - 功能：监听这个socket上的连接（将sockfd套接字指定为被动套接字）
    - 描述：
        listen() 将 sockfd 引用的套接字标记为被动套接字，即将使用 accept(2) 接受传入连接请求的套接字。
    - 参数：
        - sockfd : 通过socket()函数得到的文件描述符
        - backlog : 未连接的和已经连接的和的最大值 --> 系统设定了一个最大值，在/proc/sys/net/core/somaxconn有设定
                                使用者可以设置一个不大于系统设置的值
    - 返回值：
        - 成功： 返回 0
        - 失败： 返回 -1， 并且设置对应的错误编号

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
    - 功能：接收客户端连接，默认是一个阻塞的函数，阻塞等待客户端连接
    - 参数：
        - sockfd : 用于监听的文件描述符
        - addr : 传出参数，记录了连接成功后客户端的地址信息（ip，port）
        - addrlen : 指定第二个参数的对应的内存大小
    - 返回值：
        - 成功 ：用于通信的文件描述符，之后的通信就依赖于这个文件描述符
        - 失败 ： 返回 -1， 并且设置对应的错误编号
```

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
    - 功能： 客户端连接服务器
    - 参数：
        - sockfd : 用于通信的文件描述符
        - addr : 客户端要连接的服务器的地址信息
        - addrlen : 第二个参数的内存大小
    - 返回值：成功 0， 失败 -1，并且设置对应的错误编号

ssize_t write(int fd, const void *buf, size_t count); // 写数据

ssize_t read(int fd, void *buf, size_t count); // 读数据
```

实例1、客户端与服务端通信

客户端输入消息（用户通过键盘录入消息），服务端回应。

服务端代码

```
#include <stdio.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

// 根据用户端在键盘中输入的信息作为传输数据

int main()
{
    // 1、创建socket，返回文件描述符，是一个用于监听的套接字
    int sfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sfd == -1)
    {
        perror("socket");
        exit(-1); //失败直接结束进程
    }

    // 2、绑定ip和端口等数据（服务器端自身的特征数据）
    struct sockaddr_in addr; //先使用专用网络地址封装然后强转即可
    addr.sin_family = AF_INET;
    addr.sin_port = htons(8888);                                    //需要将主机字节序转换为网络字节序
    //inet_pton(AF_INET, "192.168.47.131", &(addr.sin_addr.s_addr)); //网络中需要将点分十进制的地址转换为网络中用的整数
    addr.sin_addr.s_addr = INADDR_ANY;    //一个电脑可能有多个网卡，如果使用这个宏，那么客户端连接任意一个网卡的ip地址都可以链接到服务端。这个宏相当于0
    int ret = bind(sfd, (struct sockaddr *)&addr, sizeof(addr));    //第二个参数需要强转成通用结构
    if (ret == -1)
    {
        perror("bind");
        exit(-1);
    }

    // 3、监听，处于等待客户端连接的阻塞状态
    ret = listen(sfd, 6);
    if (ret == -1)
    {
        perror("listen");
```

```
        exit(-1);
    }

    // 4、接受客户端连接
    struct sockaddr_in client_addr;
    int len = sizeof(client_addr);
    int client_fd = accept(sfd, (struct sockaddr *)&client_addr, &len);

    // 4.1 输出客户端的信息 ip和port （此时又需要将网络字节序转换为主机字节序）
    char client_ip[16];
    inet_ntop(AF_INET, &(client_addr.sin_addr.s_addr), client_ip, 16);
    uint16_t client_port = ntohs(client_addr.sin_port); //将网络字节序的端口转到本机字节序
    printf("%s  %d\n", client_ip, client_port);

    // 5、通信开始
    char recvBuf[1024] = {0};
    while (1)
    {
        //获取客户端数据
        int num = read(client_fd, recvBuf, sizeof(recvBuf));
        if (num == -1)
        {
            perror("read");
            exit(-1);
        }
        else if (num > 0)
        {
            printf("Server收到的消息 --> %s\n", recvBuf);
        }
        else if (num == 0)
        {
            //客户端断开连接
            printf("Client close...\n");
            break;
        }

        char *data = "接收到用户数据!";
        //向客户端发送数据
        write(client_fd, data, strlen(data));
    }

    //关闭文件描述符
    close(client_fd);
    close(sfd);

    return 0;
}
```

客户端代码

```
#include <stdio.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
```

```c
int main(int argc, char **argv)
{
    // 1、创建套接字
    int fd = socket(AF_INET, SOCK_STREAM, 0);
    if (fd == -1)
    {
        perror("socket");
        exit(-1);
    }

    //连接到服务器（这里封装的数据都是服务器的ip和端口）
    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    inet_pton(AF_INET, "192.168.47.131", &(addr.sin_addr.s_addr));
    addr.sin_port = htons(8888);
    int ret = connect(fd, (struct sockaddr *)&addr, sizeof(addr));
    if (ret == -1)
    {
        perror("connect");
        exit(-1);
    }

    // 3. 通信
    char recvBuf[1024] = {0};
    while (1)
    {
        char data[1024];
        scanf("%s", data);
        // 给客户端发送数据
        write(fd, data, sizeof(data));

        sleep(1);

        int len = read(fd, recvBuf, sizeof(recvBuf));
        if (len == -1)
        {
            perror("read");
            exit(-1);
        }
        else if (len > 0)
        {
            printf("recv server data : %s\n", recvBuf);
        }
        else if (len == 0)
        {
            // 表示服务器端断开连接
            printf("server closed...");
            break;
        }
    }

    // 关闭连接
    close(fd);

    return 0;
}
```

实例2、服务端多进程并发

服务器通过同进程技术实现多用户连接，传输数据。

```
#include <stdio.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

struct sockInfo
{
    int fd; // 通信的文件描述符
    struct sockaddr_in addr;
    pthread_t tid; // 线程号
};

struct sockInfo sockinfos[128];

void *working(void *arg)
{
    // 子线程和客户端通信   cfd 客户端的信息 线程号
    // 获取客户端的信息
    struct sockInfo *pinfo = (struct sockInfo *)arg;

    char cliIp[16];
    inet_ntop(AF_INET, &pinfo->addr.sin_addr.s_addr, cliIp, sizeof(cliIp));
    unsigned short cliPort = ntohs(pinfo->addr.sin_port);
    printf("client ip is : %s, prot is %d\n", cliIp, cliPort);

    // 接收客户端发来的数据
    char recvBuf[1024];
    while (1)
    {
        int len = read(pinfo->fd, &recvBuf, sizeof(recvBuf));

        if (len == -1)
        {
            perror("read");
            exit(-1);
        }
        else if (len > 0)
        {
            printf("recv client : %s\n", recvBuf);
        }
        else if (len == 0)
        {
            printf("client closed....\n");
            break;
        }
        write(pinfo->fd, recvBuf, strlen(recvBuf) + 1);
    }
    close(pinfo->fd);
    return NULL;
}
```

```c
int main()
{

    // 创建socket
    int lfd = socket(PF_INET, SOCK_STREAM, 0);
    if (lfd == -1)
    {
        perror("socket");
        exit(-1);
    }

    struct sockaddr_in saddr;
    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(9999);
    saddr.sin_addr.s_addr = INADDR_ANY;

    // 绑定
    int ret = bind(lfd, (struct sockaddr *)&saddr, sizeof(saddr));
    if (ret == -1)
    {
        perror("bind");
        exit(-1);
    }

    // 监听
    ret = listen(lfd, 128);
    if (ret == -1)
    {
        perror("listen");
        exit(-1);
    }

    // 初始化数据
    int max = sizeof(sockinfos) / sizeof(sockinfos[0]);
    for (int i = 0; i < max; i++)
    {
        bzero(&sockinfos[i], sizeof(sockinfos[i]));
        sockinfos[i].fd = -1;
        sockinfos[i].tid = -1;
    }

    // 循环等待客户端连接，一旦一个客户端连接进来，就创建一个子线程进行通信
    while (1)
    {
        struct sockaddr_in cliaddr;
        int len = sizeof(cliaddr);
        // 接受连接
        int cfd = accept(lfd, (struct sockaddr *)&cliaddr, &len);

        struct sockInfo *pinfo;
        for (int i = 0; i < max; i++)
        {
            // 从这个数组中找到一个可以用的sockInfo元素
            if (sockinfos[i].fd == -1)
            {
                pinfo = &sockinfos[i];
                break;
            }
```

```
            if (i == max - 1)
            {
                sleep(1);
                i--;
            }
        }

        pinfo->fd = cfd;
        memcpy(&pinfo->addr, &cliaddr, len);

        // 创建子线程
        pthread_create(&pinfo->tid, NULL, working, pinfo);

        pthread_detach(pinfo->tid);
    }

    close(lfd);
    return 0;
}
```

实例2、服务端多线程并发

```
#include <stdio.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

/*
    线程和进程不同，子线程通过一个函数去执行子进程的内在逻辑；
        从代码层面上，主线程中接收到的客户端的信息（socketaddr、data）和子线程的处理函数不在同一个作用域，所以需要将主线程接收到的客户端信息作为参数传递给子线程处理函数。
        从内存层面上，线程之间并不共享栈区的数据，而主线程接收到的客户端信息是栈区的，所以是不共享的，需要传递给子线程才能获取。
*/

//定义一个结构体，保存主线程接收到的客户端的相关信息，借助这个结构体传递给子线程的参数
struct sockInfo
{
    int fd;                   //通信的文件描述符
    struct sockaddr_in addr; //客户端的socket地址
    pthread_t tid;            //线程号
};

/*
    其次，如果结构体的数据在主线程的栈中，那么结束一次循环数据就丢失了
        如果是创建在堆中，那么将无法控制堆中的数据，那么在子线程的函数中就必须在最后删除堆上的数据，否则，将会有很多垃圾数据

    这里用一个全局的数组保存多个结构体数据。每次接收到一个新的客户端，就找一个可用的数组位置存放这些客户端数据
        每次子线程执行到最后，就把这块元素区域恢复到原来的状态，能够继续复用该区域
        如果超出了数组的大小，那么就丢弃该链接
*/
struct sockInfo sockInfos[3]; //元素个数设置为3，方便测试效果
```

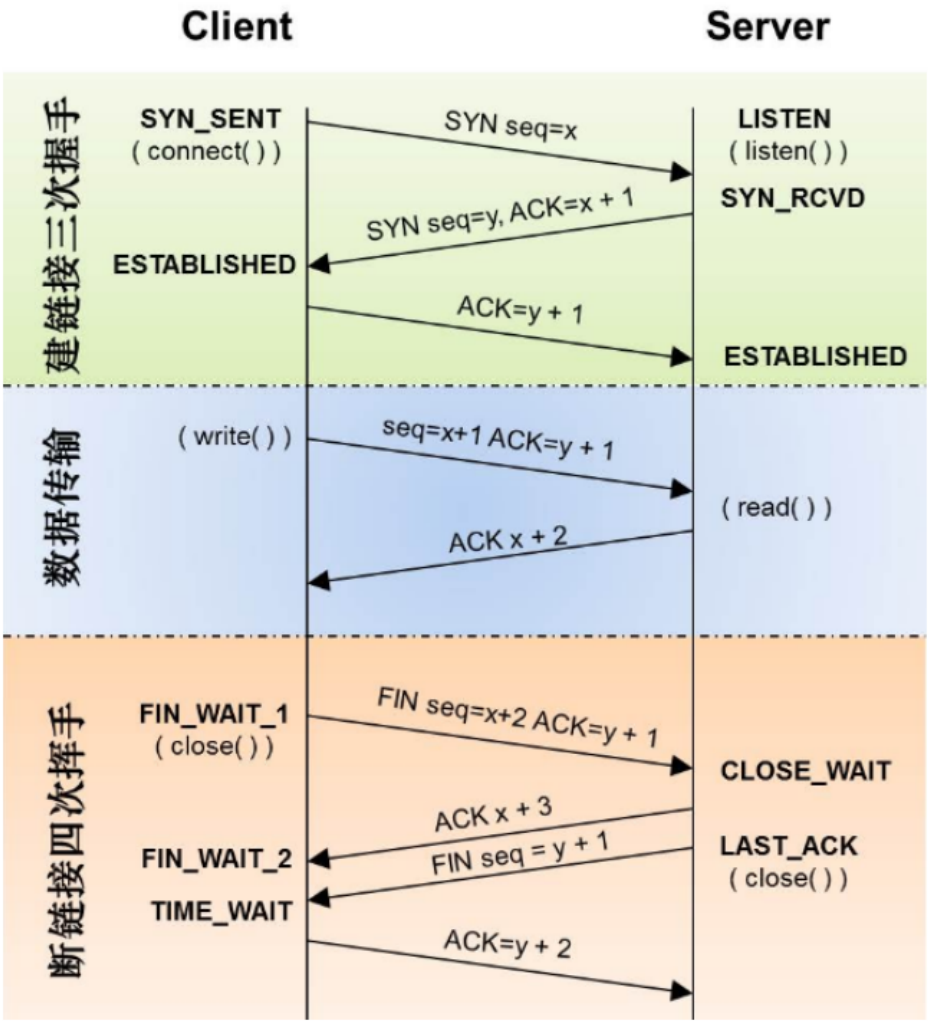
```c
//子线程逻辑
void *working(void *arg)
{
    // 子线程和客户端通信   cfd 客户端的信息 线程号
    // 获取客户端的信息
    struct sockInfo *pinfo = (struct sockInfo *)arg;

    char cliIp[16];
    inet_ntop(AF_INET, &pinfo->addr.sin_addr.s_addr, cliIp, sizeof(cliIp));
    unsigned short cliPort = ntohs(pinfo->addr.sin_port);
    printf("client ip is : %s, prot is %d has connected !!!\n", cliIp, cliPort);

    // 接收客户端发来的数据
    char recvBuf[1024];
    while (1)
    {
        int len = read(pinfo->fd, &recvBuf, sizeof(recvBuf));

        if (len == -1)
        {
            perror("read-server");
            exit(-1);
        }
        else if (len > 0)
        {
            printf("recv client : %s \n", recvBuf);
        }
        else if (len == 0)
        {
            printf("client closed....\n");
            break;
        }
        write(pinfo->fd, recvBuf, strlen(recvBuf) + 1);
    }
    close(pinfo->fd); //关闭通信socket文件描述符

    //重置需要判断的文件描述符即可，不能将内存子节清为0，否则会有错误
    pinfo->fd = -1;

    return NULL;
}

int main()
{
    //创建socket
    int lfd = socket(PF_INET, SOCK_STREAM, 0);
    if (lfd == -1)
    {
        perror("socket");
        exit(-1);
    }

    //绑定
    struct sockaddr_in saddr;
    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(9999);
    saddr.sin_addr.s_addr = INADDR_ANY;
```

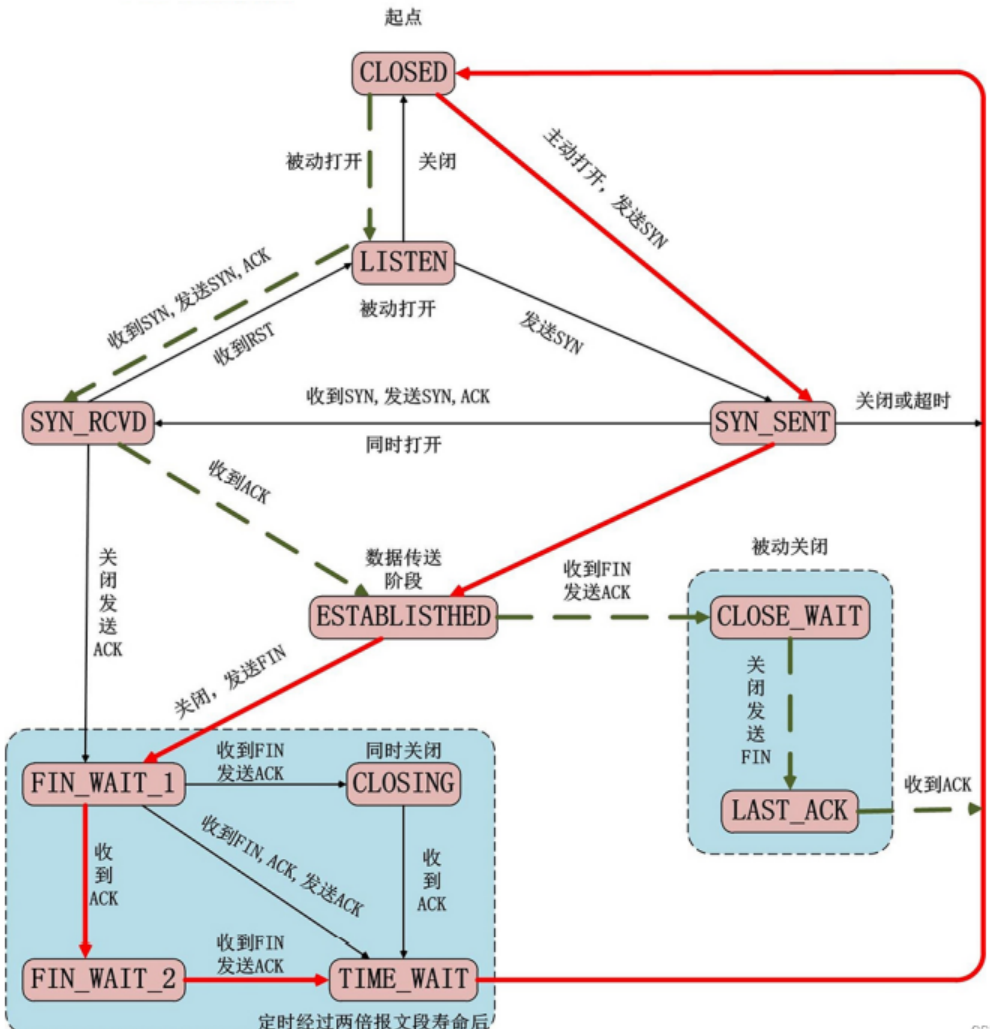
```
    int ret = bind(lfd, (struct sockaddr *)&saddr, sizeof(saddr));
    if (ret == 1)
    {
        perror("bind");
        exit(-1);
    }

    //监听
    ret = listen(lfd, 128);
    if (ret == -1)
    {
        perror("listen");
        exit(-1);
    }

    //初始化数据（为了能够根据初始化的值来判断数组的一块区域是否可用）
    int count = sizeof(sockInfos) / sizeof(sockInfos[0]); //得到数组大小
    for (int i = 0; i < count; i++)
    {
        // void bzero(void *, int n) == memset((void *)s, 0, size_tn); 将内存块的前
n个字节清零。
        bzero(&sockInfos[i], sizeof(sockInfos[i])); //将指定区域的数据设置为0
        sockInfos[i].fd = -1;
        sockInfos[i].tid = -1;
    }

    //开始循环等待客户端连接，一旦一个客户端连接进来，就创建一个子线程进行通信
    while (1)
    {
        struct sockaddr_in cliaddr; //客户端socket地址
        int len = sizeof(cliaddr);

        //接收连接
        int cfd = accept(lfd, (struct sockaddr *)&cliaddr, &len);

        struct sockInfo *pinfo; //将全局数组中的一块空间给这个指针变量
        int i = 0;
        for (; i < count; i++)
        {
            //遍历数组，找到一个可用的空间
            if (sockInfos[i].fd == -1)
            {
                pinfo = &sockInfos[i];
                break;
            }
        }
        if (i >= count)
            continue; //丢弃该客户端连接

        pinfo->fd = cfd;
        memcpy(&pinfo->addr, &cliaddr, len);
        //创建子线程
        pthread_create(&pinfo->tid, NULL, working, pinfo);

        //设置线程分离 自动回收子线程资源
        pthread_detach(pinfo->tid);
    }
```

```
    close(lfd);
    return 0;
}
```

TCP状态转换

简介


Client
Server
建链接三次握手
SYN_SENT
(connect( ))
SYN seq=x
LISTEN
(listen( ))
SYN_RCVD
SYN seq=y, ACK=x + 1
ESTABLISHED
ACK=y + 1
ESTABLISHED
数据传输
(write( ))
seq=x+1 ACK=y + 1
(read( ))
ACK x + 2
断链接四次挥手
FIN_WAIT_1
(close( ))
FIN seq=x+2 ACK=y + 1
CLOSE_WAIT
ACK x + 3
FIN_WAIT_2
FIN seq = y + 1
LAST_ACK
(close( ))
TIME_WAIT
ACK=y + 2


起点
CLOSED
被动打开
关闭
主动打开，发送SYN
LISTEN
被动打开
收到SYN, 发送SYN, ACK
收到RST
发送SYN
收到SYN, 发送SYN, ACK
关闭或超时
SYN_RCVD
SYN_SENT
同时打开
收到ACK
关闭发送ACK
数据传送阶段
收到FIN
发送ACK
被动关闭
ESTABLISTHED
CLOSE_WAIT
关闭，发送FIN
关闭发送FIN
收到FIN
发送ACK
同时关闭
FIN_WAIT_1
CLOSING
收到ACK
收到FIN, ACK, 发送ACK
收到ACK
收到FIN
发送ACK
FIN_WAIT_2
TIME_WAIT
定时经过两倍报文段寿命后

- 2MSL(Maximum Segment Lifetime)
  主动断开连接的一方，最后进入一个TIME_WAIT状态，这个状态会持续：2MSL。
  msl：官方建议2分钟，实际30秒。

> 当 TCP 连接主动关闭方接收到被动关闭方发送的 ACK和最终的 FIN后，连接的主动关闭方必须处于TIME_WAIT 状态并持续 2MSL 时间。
> 这样就能够让 TCP 连接的主动关闭方在它发送的 ACK 丢失的情况下重新发送最终的 ACK。
> 主动关闭方重新发送的最终 ACK 并不是因为被动关闭方重传了 ACK（它们并不消耗序列号，被动关闭方也不会重传），而是因为被动关闭方重传了它的 FIN。事实上，被动关闭方总是重传 FIN 直到它收到一个最终的 ACK。

- 半关闭

> 当 TCP 链接中 A 向 B 发送 FIN 请求关闭，另一端 B 回应 ACK 之后（A 端进入 FIN_WAIT_2状态），并没有立即发送 FIN 给 A，A 方处于半连接状态（半开关），此时 A 可以接收 B 发送的数据，但是 A 已经不能再向 B 发送数据。

shutdown函数

从程序的角度，可以使用 API 来控制实现半连接（半关闭）状态：

```
#include <sys/socket.h>

int shutdown(int sockfd, int how);
    sockfd: 需要关闭的socket的描述符
    how: 允许为shutdown操作选择以下几种方式:
    SHUT_RD(0): 关闭sockfd上的读功能，此选项将不允许sockfd进行读操作。该套接字不再接收数据，任何当前在套接字接受缓冲区的数据将被无声的丢弃掉。
    SHUT_WR(1): 关闭sockfd的写功能，此选项将不允许sockfd进行写操作。进程不能在对此套接字发出写操作。
    SHUT_RDWR(2):关闭sockfd的读写功能。相当于调用shutdown两次：首先是以SHUT_RD,然后以SHUT_WR。
```

使用 close 中止一个连接，但它只是减少描述符的引用计数，并不直接关闭连接，只有当描述符的引用计数为 0 时才关闭连接。shutdown 不考虑描述符的引用计数，直接关闭描述符。也可选择中止一个方向的连接，只中止读或只中止写。

注意

1、如果有多个进程共享一个套接字，close 每被调用一次，计数减 1 ，直到计数为 0 时，也就是所用进程都调用了 close，套接字将被释放。
2、在多进程中如果一个进程调用了 shutdown(sfd, SHUT_RDWR) 后，其它的进程将无法进行通信。但如果一个进程 close(sfd) 将不会影响到其它进程。

端口复用

简介

端口复用常见用途：
1、防止服务器重启时之前绑定的端口还未释放；
2、程序突然退出而系统没有释放端口。
如上，主动关闭的一方会进入一个状态TIME_WAIT，如果服务器主动断开了连接，那么最后会经过这个状态，此时会持续2MSL(1分钟)，此时的地址和端口号还是被占用着，如果要重启服务器，显示会出错。本节主要是解决服务器断开之后重启端口被占用一段时间无法重启的问题。

网络信息相关的命令

```
netstat
    -a 所有的socket
    -p 显示正在使用socket的程序的名称
    -n 直接使用ip地址，而不通过域名服务器
```

```
#include <sys/types.h>
#include <sys/socket.h>

//设置套接字属性（不仅仅能够设置端口复用）
int setsockopt(int sockfd, int level, int optname, const void * optval,
socklen_t optlen);
    参数:
        - sockfd ： 要操作的文件描述符
        - level ： 级别 - SOL_SOCKET（该宏是端口复用的级别）
        - optname ：选项的名称
            - SO_REUSEADDR
            - SO_REUSEPORT
        - optval ： 端口复用的值（整型）
            - 1 ： 可以复用
            - 0 ： 不可以复用
        - optlen ： optval参数的大小
注意：端口复用，设置的时机是在服务器绑定端口之前。
```

查看未端口复用的情况

如下代码，先将设置端口复用的两行代码注释掉，然后执行服务器端和客户端。执行过程中通过netstat命令查看socket连接。然后主动断开服务端，再查看状态，以及在断开之后的2MSL时间内，再次重启服务端；以及超出2MSL时间重启服务端。

1、仅有服务端开启

2、客户端开启

3、通信

4、服务端关闭

5、重启服务端（< 2MSL）

```
kiko@Hkiko:~/lesson/network$ ./server
bind: Address already in use
```


6、重启服务端（> 2MSL）

```
kiko@Hkiko:~/lesson/network$ ./server
```


端口复用后重启服务器端

当解开注释了端口复用的函数后，在服务端断开后小于2MSL时间内重启服务端。

```
kiko@Hkiko:~/lesson/network$ ./server
client's ip is 127.0.0.1, and port is 53128
^C
kiko@Hkiko:~/lesson/network$ ./server
```


代码

```c
#include <stdio.h>
#include <ctype.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]) {

    // 创建socket
    int lfd = socket(PF_INET, SOCK_STREAM, 0);

    if(lfd == -1) {
        perror("socket");
        return -1;
    }

    struct sockaddr_in saddr;
    saddr.sin_family = AF_INET;
    saddr.sin_addr.s_addr = INADDR_ANY;
    saddr.sin_port = htons(9999);

    //int optval = 1;
    //setsockopt(lfd, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval));

    int optval = 1;
    setsockopt(lfd, SOL_SOCKET, SO_REUSEPORT, &optval, sizeof(optval));

    // 绑定
    int ret = bind(lfd, (struct sockaddr *)&saddr, sizeof(saddr));
    if(ret == -1) {
        perror("bind");
        return -1;
    }

    // 监听
    ret = listen(lfd, 8);
    if(ret == -1) {
```

```
        perror("listen");
        return -1;
    }

    // 接收客户端连接
    struct sockaddr_in cliaddr;
    socklen_t len = sizeof(cliaddr);
    int cfd = accept(lfd, (struct sockaddr *)&cliaddr, &len);
    if(cfd == -1) {
        perror("accpet");
        return -1;
    }

    // 获取客户端信息
    char cliIp[16];
    inet_ntop(AF_INET, &cliaddr.sin_addr.s_addr, cliIp, sizeof(cliIp));
    unsigned short cliPort = ntohs(cliaddr.sin_port);

    // 输出客户端的信息
    printf("client's ip is %s, and port is %d\n", cliIp, cliPort );

    // 接收客户端发来的数据
    char recvBuf[1024] = {0};
    while(1) {
        int len = recv(cfd, recvBuf, sizeof(recvBuf), 0);
        if(len == -1) {
            perror("recv");
            return -1;
        } else if(len == 0) {
            printf("客户端已经断开连接...\n");
            break;
        } else if(len > 0) {
            printf("read buf = %s\n", recvBuf);
        }

        // 小写转大写
        for(int i = 0; i < len; ++i) {
            recvBuf[i] = toupper(recvBuf[i]);
        }

        printf("after buf = %s\n", recvBuf);

        // 大写字符串发给客户端
        ret = send(cfd, recvBuf, strlen(recvBuf) + 1, 0);
        if(ret == -1) {
            perror("send");
            return -1;
        }
    }

    close(cfd);
    close(lfd);

    return 0;
}
```

```
#include <stdio.h>
```

```c
#include <arpa/inet.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main() {

    // 创建socket
    int fd = socket(PF_INET, SOCK_STREAM, 0);
    if(fd == -1) {
        perror("socket");
        return -1;
    }

    struct sockaddr_in seraddr;
    inet_pton(AF_INET, "127.0.0.1", &seraddr.sin_addr.s_addr);
    seraddr.sin_family = AF_INET;
    seraddr.sin_port = htons(9999);

    // 连接服务器
    int ret = connect(fd, (struct sockaddr *)&seraddr, sizeof(seraddr));

    if(ret == -1){
        perror("connect");
        return -1;
    }

    while(1) {
        char sendBuf[1024] = {0};
        fgets(sendBuf, sizeof(sendBuf), stdin);

        write(fd, sendBuf, strlen(sendBuf) + 1);

        // 接收
        int len = read(fd, sendBuf, sizeof(sendBuf));
        if(len == -1) {
            perror("read");
            return -1;
        }else if(len > 0) {
            printf("read buf = %s\n", sendBuf);
        } else {
            printf("服务器已经断开连接...\n");
            break;
        }
    }

    close(fd);

    return 0;
}
```