第五章：网络编程

一、网络编程的基本概念

1.1 为什么引入网络编程

1> 回顾之前学习的进程间通信方式：

内核提供三种：有名管道、无名管道、信号

system V提供三种：消息队列、共享内存、信号量集

2> 以上通信方式，仅仅局限于同一主机之间多个进程间通信方式，不能实现跨主机的通信方式

3> 如果想要实现跨主机的通信方式，我们引入的套接字的通信方式，就是该门课程要讲述的内容

1.2 网络的发展历史

1> 第一阶段：APRAnet阶段----》冷战的产物

阿帕网，是Internet的最早雏形，注意，此时的网络**不能实现互联不同类型的计算机和不同类型的操作系统，并且没有纠错功能**

2> TCP\IP两个协议的阶段

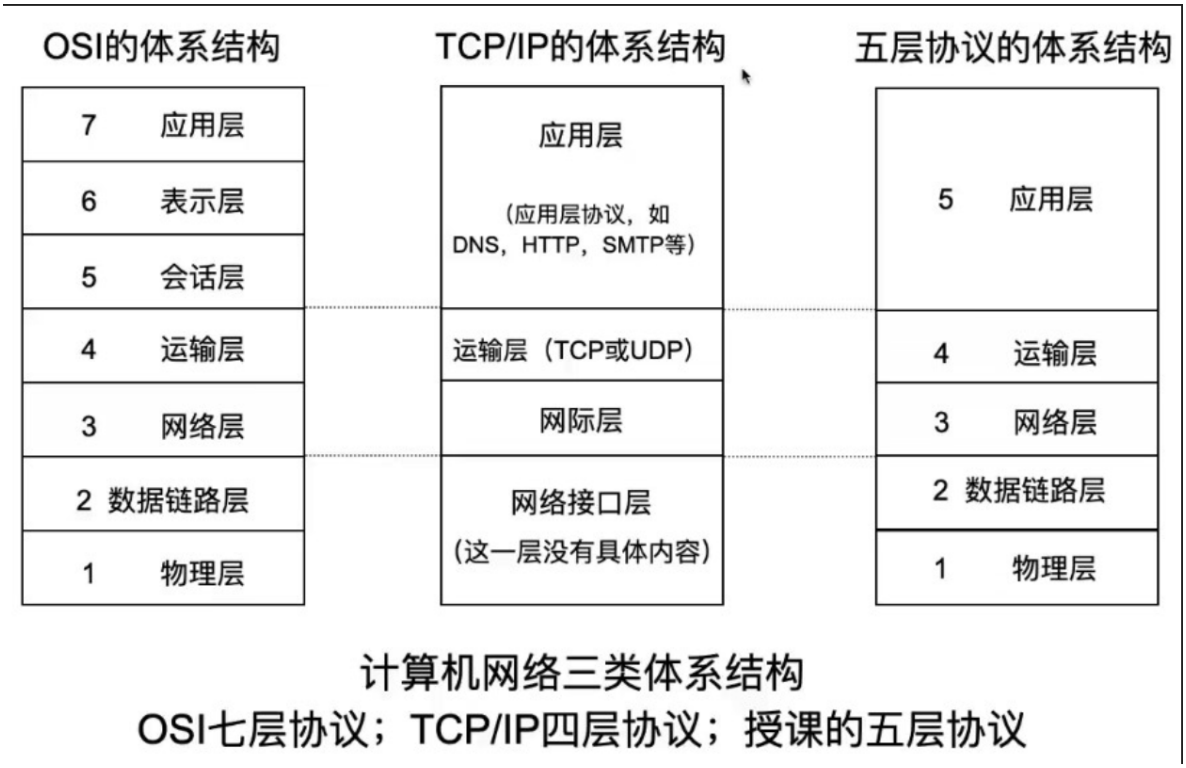
协议：在计算机网络中，要做到有条不紊的交换数据，需要遵循一些实现约定好的规则，这些规则明确规定了所交换数据的格式以及相关的同步问题。为了进行网络中数据交换而建立的规则、标准和约定统称为网络协议（protocol）

该阶段引入了两个协议（只有两个）：

TCP：传输控制协议，用来检测网络传输中的差错控制。

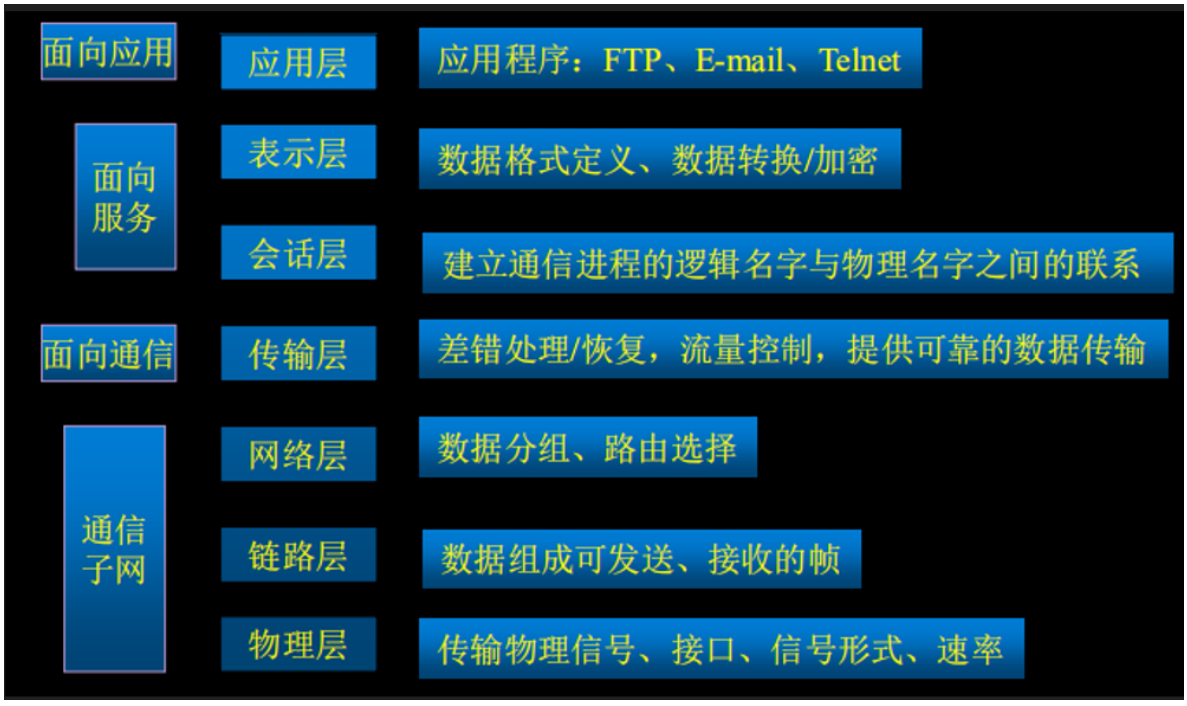
IP：网际协议，用来专门负责不同网络中进行互联的协议。

3> OSI（开放系统互联模型）的七层网络体系结构



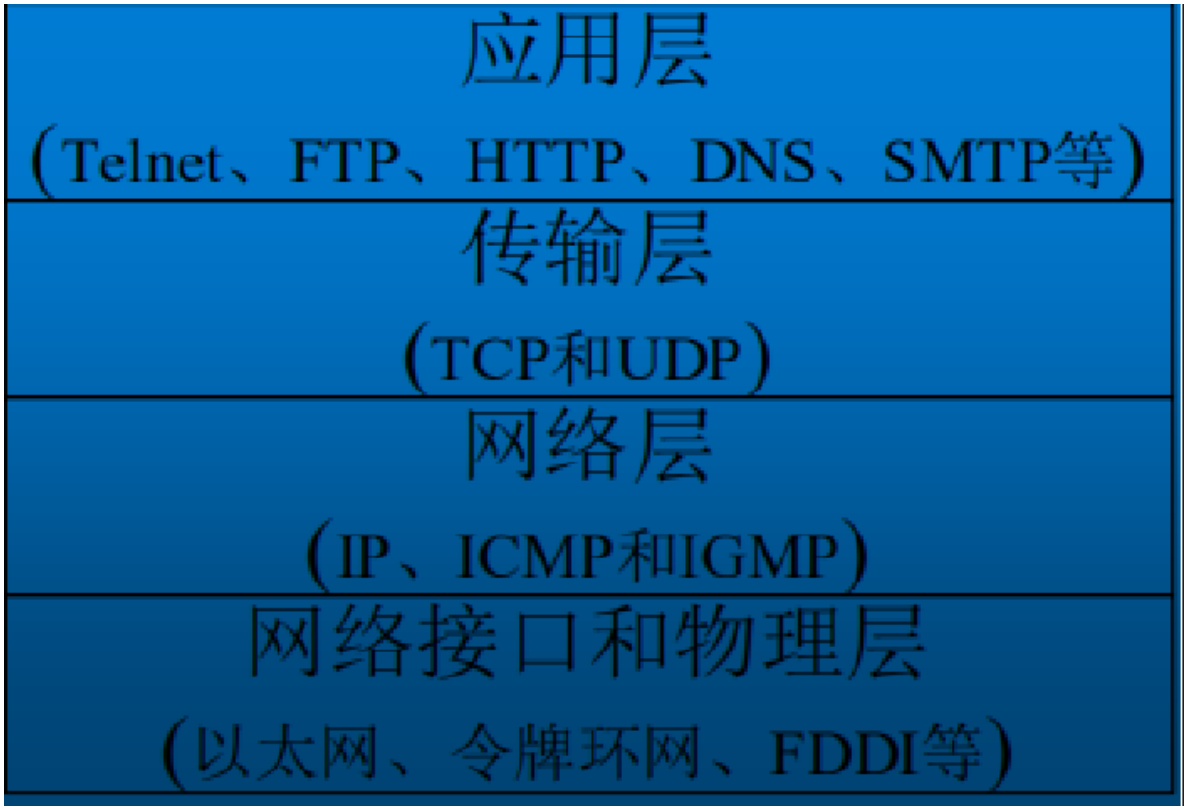
网络体系结构的概念
1> 每一层都有自己独立的分工，单每一层都不可获取
2> 通常将功能相近的协议组织在一起放在一层，称为协议栈，所以每一层中其实有多个协议
分层的好处：
1> 各层之间相互独立，每一层不需要知道下一层如何实现，而仅仅只需要知道该层通过层间接口锁提供的服务
2> 稳定、灵活性好。当每一层出现变化时，只要层间接口保持不变，则当前层的上层和下次不受影响
3> 易于实现和维护，只要知道哪一层的功能，直接对指定层进行维护即可
4> 促进标准化工作，每一层所提供的服务和技术都有精确的说明
5> 结构上是不可分割的，各层之间都采用的最合适的技术来实现

1、OSI（开发系统互联模型）是由ISO（国际标准化组织）提出的理想化模型，一共有七层：**物数网传会表应**



2、TCP\IP协议族的体系结构：**也是互联网事实上的工业标准**

一共提供的四层：应用层、传输层、网络层、链路层（网络接口层和物理）



3、虽然TCP\IP体系结构只有四层，但是做的事和OSI的七层体系结构是一样的

4、TCP\IP四层网络体系结构和OSI七层网络体系结构的对应关系如下

OSI中的层	功能	TCP/IP协议族
应用层	文件传输，电子邮件，文件服务，虚拟终端	TFTP，HTTP，SNMP，FTP，SMTP，DNS，Telnet
表示层	数据格式化，代码转换，数据加密	没有协议
会话层	解除或建立与别的节点的联系	没有协议
传输层	提供端对端的接口	TCP，UDP
网络层	为数据包选择路由	IP，ICMP，RIP，OSPF，BGP，IGMP
数据链路层	传输有地址的帧以及错误检测功能	SLIP，CSLIP，PPP，ARP，RARP，MTU
物理层	以二进制数据形式在物理媒体上传输数据	ISO2110，IEEE802.1,EEE802.2

4>　tcp\ip协议族中常见的协议

```
应用层：
HTTP(Hypertext Transfer Protocol)　　超文本传输协议
```

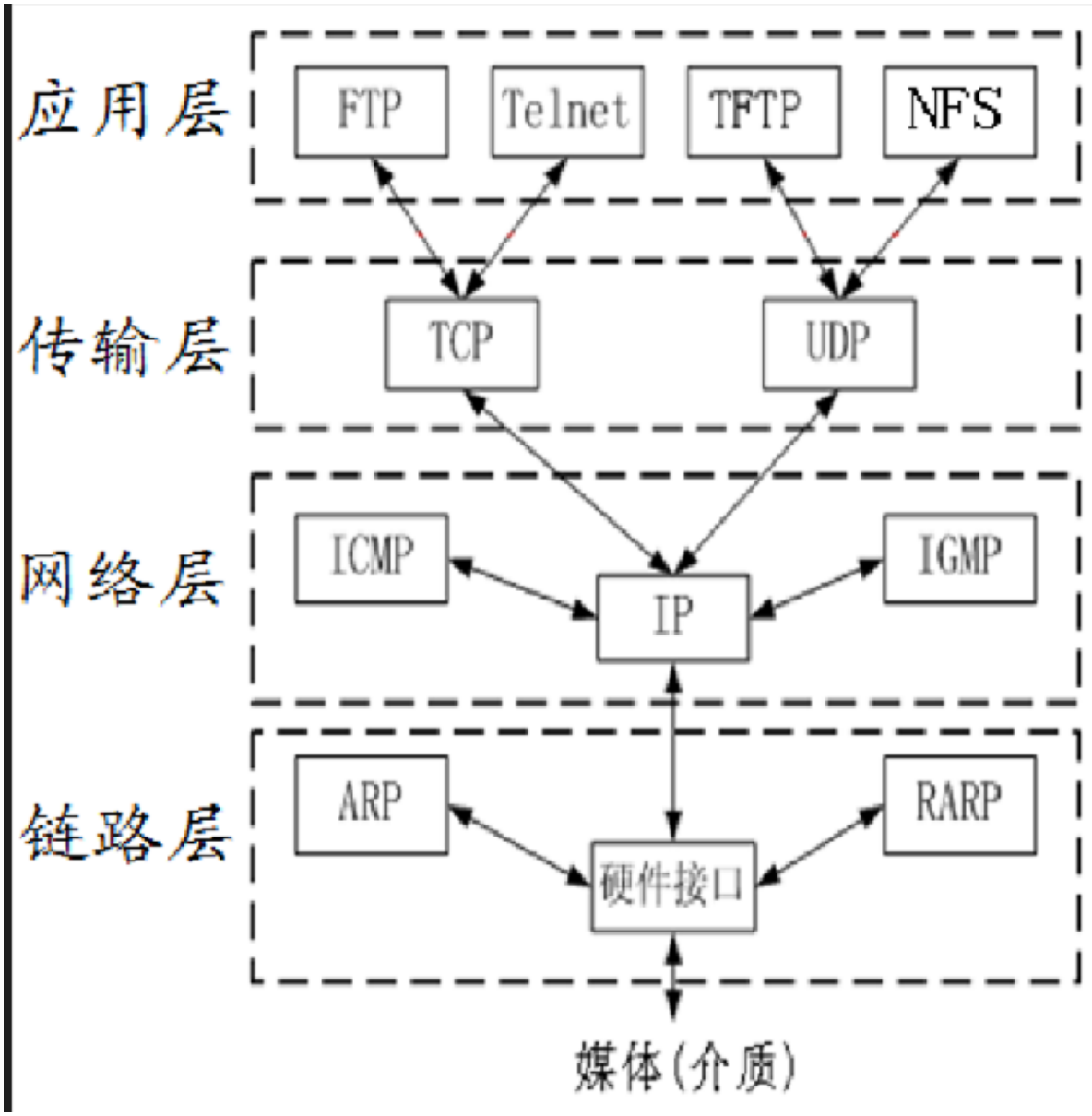
万维网的数据通信的基础
FTP(File Transfer Protocol) 文件传输协议
是用于在网络上进行文件传输的一套标准协议，使用TCP传输
TFTP(Trivial File Transfer Protocol) 简单文件传输协议
是用于在网络上进行文件传输的一套标准协议，使用UDP传输
SMTP(Simple Mail Transfer Protocol) 简单邮件传输协议
一种提供可靠且有效的电子邮件传输的协议

传输层：
TCP(Transport Control Protocol) 传输控制协议
是一种面向连接的、可靠的、基于字节流的传输层通信协议
UDP(User Datagram Protocol) 用户数据报协议
是一种无连接、不可靠、快速传输的传输层通信协议

网络层：
IP(Internetworking Protocol) 网际互连协议
是指能够在多个不同网络间实现信息传输的协议
ICMP(Internet Control Message Protocol) 互联网控制信息协议
用于在IP主机、路由器之间传递控制消息、ping命令使用的协议
IGMP(Internet Group Management Protocol) 互联网组管理
是一个组播协议，用于主机和组播路由器之间通信

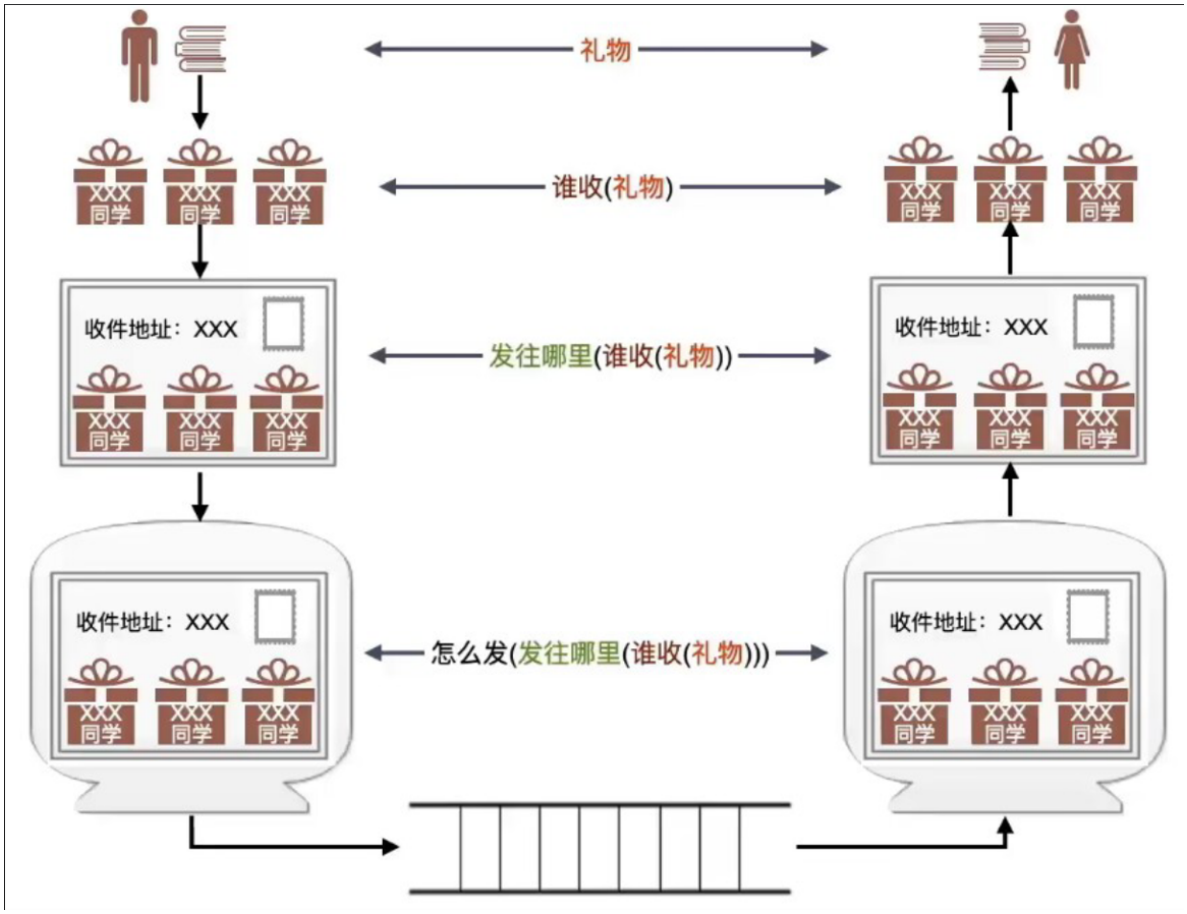
链路层：
ARP(Address Resolution Protocol) 地址解析协议
通过IP地址获取对方mac地址
RARP(Reverse Address Resolution Protocol) 逆向地址解析协议
通过mac地址获取ip地址

注意：每次使用的协议时由下层决定的，不能乱用

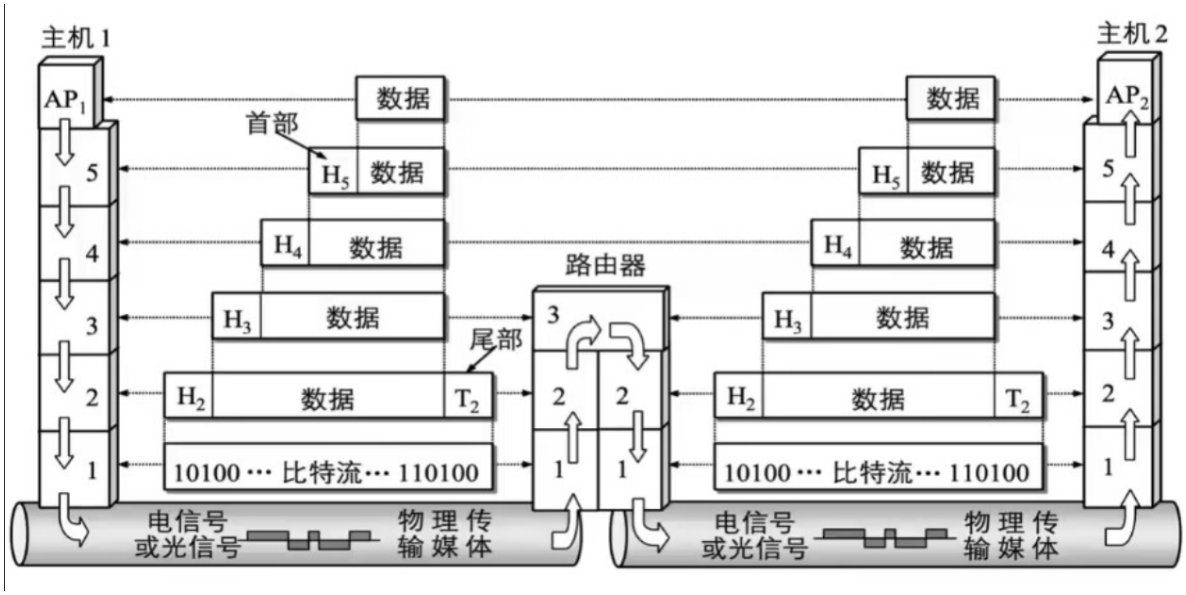


1.3 数据传输中封包和拆包过程

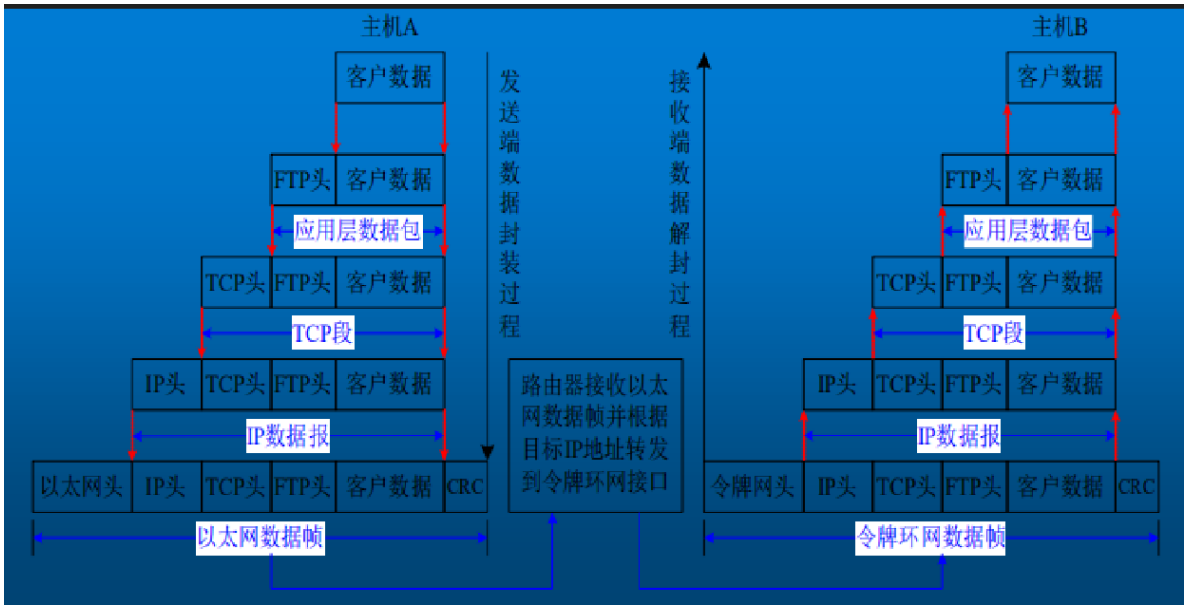
1> 案例引入

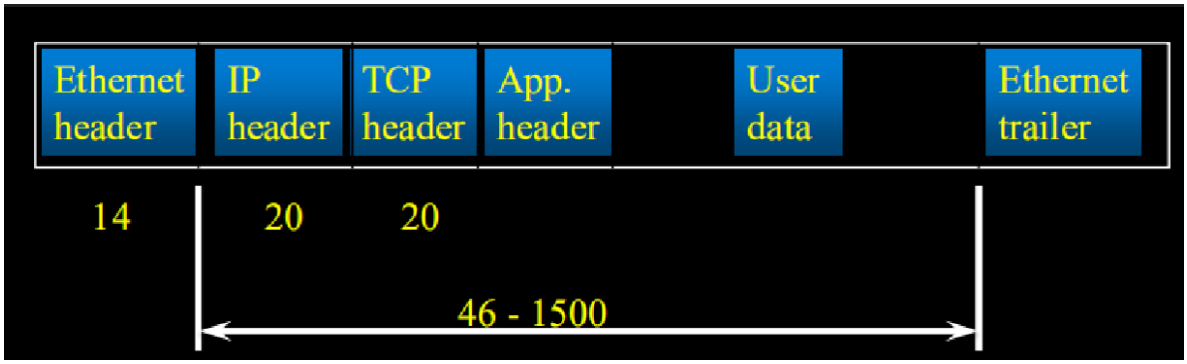


2> 网络通信中封包拆包过程的详解



3> 对等层（不同主机的同一层称为对等层）中数据包的名称





一帧数据的说明：

1、大小为：64--1518 （包含以太网首部14字节，以太网尾部4字节）

2、如果数据大于MTU(最大传输单元，linux中默认是1500)，需要分成两个或多个数据包进行传输

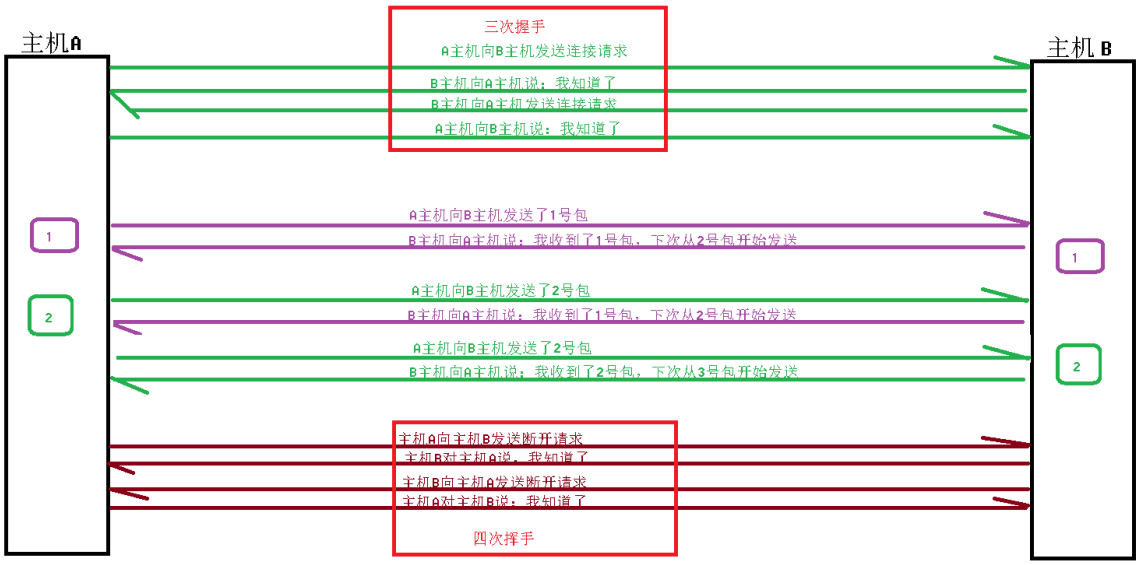
3、可以使用ifconfig查看mtu的值

1.4 TCP和UDP的区别

1> 两者都属于传输层的相关协议

2> TCP而言 -----> 稳定

1、TCP提供了面向连接的、可靠的数据传输服务
2、传输过程中，能够保障数据无误、数据无丢失、数据无重复、数据无失序
TCP通信中会给每个数据包编上编号，该编号称为序列号
每个序列号都需要应打包应答，如果没有应答，则会将上面的数据包重复传输
3、TCP通信中，数据传输效率较低，耗费资源较多
4、数据收发是不同步的
为了提高传输效率，tcp通信中会将多个较小的，发送时间间隔较短的数据包，沾成一个数据包进行发送，该现象称为沾包现象
5、TCP通信使用场景：对于传输质量要求较高的以及传输量较大的数据通信，在需要可靠传输信息的场合，一般使用TCP通信
例如：账户和密码登录注册、大型文件的下载



3> UDP通信 ------> 快速

1、提供面向无连接、不保证数据可靠传输、尽最大努力传输的协议
2、数据传输过程中，可能会出现数据丢失、重复、失序、乱序等现象
3、数据传输效率较高、实时性高
4、限制每次传输的数据大小，多出部分会直接被忽略删除
5、数据的收发是同步的，不会沾包
6、使用场景：发送小尺寸的，在收到数据后给出应答比较困难的情况下，采用udp通信
例如：广播、组播、通信软件的音视频传输

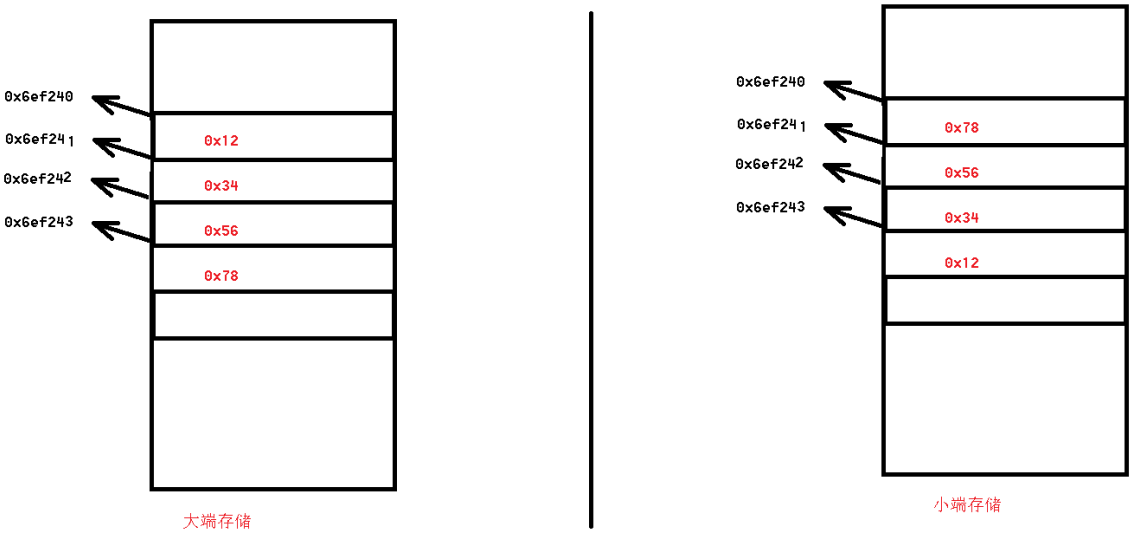
1.5 字节序的概念

1> 字节序：计算机在存储**多字节整数**时，根据主机的CPU处理架构不同，我们将主机分成大端存储的主机和小端存储的主机

大端存储：内存地址低位存储的是数据的高位

小端存储：内存地址的低位存储的是数据的低位

```
int num = 0x12345678
```



2> 验证当前主机是大端存储还是小端存储

使用指针来进行判断

```
#include <myhead.h>

int main(int argc, const char *argv[])
{
    //定义一个整形变量
    int num = 0x12345678;

    //定义一个字符类型的指针，指向整形变量的起始地址
    char *ptr = (char *)&num;

    //对ptr所指向的字节中的内容进行判断，如果是0x12则说明是大端存储
    //如果是0x78则说明是小端存储
    if(*ptr == 0x12)
    {
        cout<<"big endian"<<endl;
    }else if(*ptr == 0x78)
    {
        cout << "little endian"<<endl;
    }

    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

使用共用体判断主机的大小端

```
#include <myhead.h>

//定义一个共用体类型：多个成员共享一个成员的空间，共享的是所占内存空间最大的那个成员
union Info
{
    int num;       //四字节整数
    char ch;       //一字节
};

int main(int argc, const char *argv[])
{
    //定义一个共用体变量
    union Info temp;

    //给其整形成员赋值
    temp.num = 0x12345678;

    //判断其ch成员
    if(temp.ch == 0x12)
    {
        cout<<"big endian"<<endl;
    }else if(temp.ch == 0x78)
    {
        cout<<"little endian"<<endl;
    }


    std::cout << "Hello, World!" << std::endl;
    return 0;
```

```
}
```

3> 由于不同主机之间存储方式不同，可能会出现，小端存储的主机中的多字节整数，在网络传输过程中，明明没有出现任何问题，但是，由于大小端存储问题，导致，多字节整数传输出现错误。

基于此，我们引入的网络字节序的概念，规定网络字节序都是大端存储的。

无论发送端是大端存储还是小端存储，在传输多字节整数时，一律先转换为网络字节序。经由网络传输后，到达目的主机后，在转换为主机字节序即可

4> 系统给大家提供了一套有关网络字节序和主机字节序之间相互转换的函数

主机：host

网络：network

转换：to

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

uint32_t htonl(uint32_t hostlong);//将4字节整数主机字节序转换为网络字节序，参数是主机字节序，返回值是网络字节序

uint16_t htons(uint16_t hostshort);//将2字节整数主机字节序转换为网络字节序，参数是主机字节序，返回值是网络字节序

uint32_t ntohl(uint32_t netlong);//将4字节整数的网络字节序转换为主机字节序，参数是网络字节序，返回值是主机字节序

uint16_t ntohs(uint16_t netshort);//将2字节整数的网络字节序转换为主机字节序，参数是网络字节序，返回值是主机字节序
```

```
#include <myhead.h>

int main(int argc, const char *argv[])
{
    //定义一个4字节整数
    int num = 0x12345678;

    //调用函数，将该数据转换为网络字节序
    int res = htonl(num);

    printf("res = %#x\n", res);      //res = 0x78563412


    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

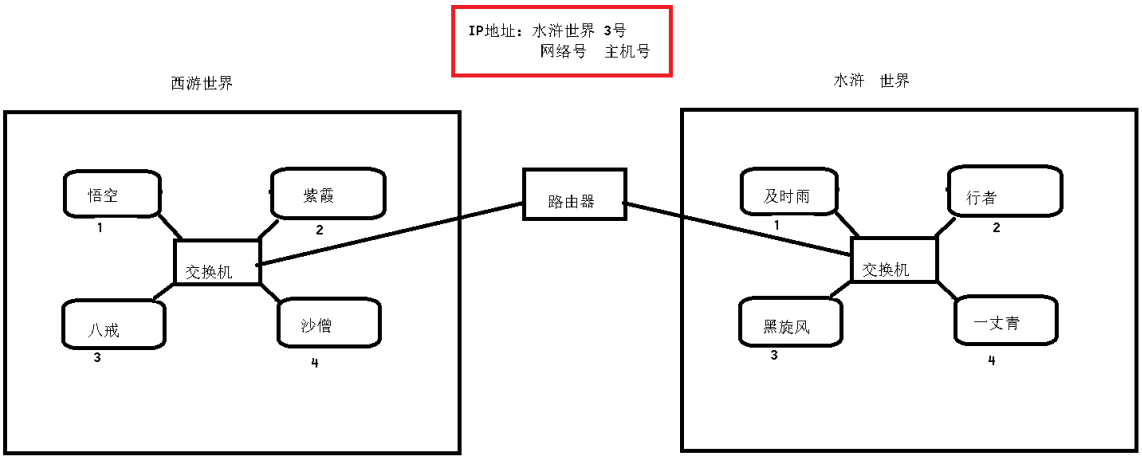
5> 何时使用网络字节序转换函数

1、**在进行多字节整数网络传输时，需要使用字节序转换函数**

2、在进行单字节整数传输时，不需要使用

3、在网络中传输字符串时，也不需要使用

1.6 ip地址



1> ip地址是主机在网络中的唯一标识，由两部分组成，分别是网络号和主机号。

网络号：确定计算机所从属的网络

主机号：标识该设备在该网络中的一个编号

2> 作用：在网络传输过程中，给网络传输载体必须添加的信息，指定源ip地址和目的ip地址，以便于找到目的主机

3> IP地址的分类

1、IPv4：是使用4字节无符号整数表示的一个ip地址，取值范围 [0, 2^32-1]
一共有四十多亿个，很明显不够，我们采用相关技术进行扩充
局域网扩充：为了解决ip地址不够用，让多个主机共享一个ip地址
WAN：wide area network （广域网）
LAN：local area network（局域网）

2、IPv6：是使用16字节无符号整数表示的一个ip地址，取值范围 【0， 2^128-1】

3、注意：IPv6是不兼容IPv4的

4> IP地址的划分：由于IP地址比较庞大，我们将其进行分组，一共分为5类网络，分别是A类、B类、C类、D类、E类网络

网络类型	取值范围	网络号个数	主机号个数	用途
A类网络	1.0.0.0 --- 127.255.255.255	2^7	2^24	已经保留不供给使用

B类网络：128.0.0.0--191.255.255.255	2^14	2^16	名地址网管中心
C类网络：192.0.0.0--223.255.255.255	2^21	2^8	家庭、校园、公司使用
D类网络：224.0.0.0--239.255.255.255			组播IP
E类网络：240.0.0.0--255.255.255.255			保留、实验室使用

5> 特殊的IP地址

```
1、网络号 + 全为0的主机号：表示该网络，不分配给任何主机使用，例如：192.168.10.0
2、网络号 + 全为1的主机号：表示当前网络的广播地址，也不分配给任何主机使用，例如：
192.168.10.255
3、网络号 + 主机号为1：默认表示网关，当然可以自己制定网关ip
4、127.0.0.0：本地环回ip，当没有网络时，用于测试当前主机的ip
5、0.0.0.0：表示当前局域网中的任意一个主机号
6、255.255.255.255：一般表示广播地址
```

6> 点分十进制

为了方便记忆，我们将ip地址的每一个字节单独计算出十进制数据，并用点进行分割，这种方式，称为点分十进制，在程序中使用的是字符串来存储的。但是，ip地址的本质是4字节无符号整数，在网络中进行传输时，需要使用的是4字节无符号整数，而不是点分十进制的字符串。此时，就需要引入关于点分十进制数据向4字节无符号整数转换的相关函数

地址：address

网络：network

转换：to

```
        #include <sys/socket.h>
        #include <netinet/in.h>
        #include <arpa/inet.h>

        in_addr_t inet_addr(const char *cp);//将点分十进制的ip地址转换为4字节无符号整数的网络字节序，参数时点分十进制数据，返回值时4字节无符号整数

        char *inet_ntoa(struct in_addr in);//将4字节无符号整数的网络字节序，转换为点分十进制的字符串
```

```
#include <myhead.h>

int main(int argc, const char *argv[])
{
    //地址指针记录ip地址
    const char *ip = "172.20.10.8";    //点分十进制

    unsigned int ip_net = inet_addr(ip);    //调用函数将点分十进制转换为4字节无符号整数

    // AC  14 A 8
    printf("ip_net = %#x\n", ip_net);     //0x80a14ac
```

```
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

1.7 端口号（port）

1> 作用：为了区分同一个主机之间的每个进程的，使用端口号来进行标识

概念：端口是是一个 2 字节的无符号整数表示的数字，取值范围 【0， 65535】

2> 为什么不使用进程号标识，而使用端口号

答：因为进程号是进程的唯一标识，当同一个应用程序，关闭再打开后，并不是同一个进程号了，但是是同一个应用程序

所以，端口号标识的是我们的应用程序，当一个应用程序关闭再打开后，端口号不变

3> 引入端口号后，网络通信的两个重要因素就集结完毕：ip 地址 + 端口号

ip地址可以在网络中，唯一确定对端的主机地址，通过端口号能够找到该主机中指定的对端应用程序

4> 端口号的分类

1、0~1023 ： 众所周知的 “VIP”端口号：被特殊的应用程序已经占用了的。

可以查看 / etc /services 中的文件内容，该文件中记录了特殊的端口号
TCP和UDP分别使用不同的一套标准
注意：有时使用这些特殊的端口号时，需要使用管理员权限

```
echo            7/tcp
echo            7/udp
ssh             22/tcp                          # The Secure Shell (SSH)
Protocol
ssh             22/udp                          # The Secure Shell (SSH)
Protocol
ftp             21/tcp
ftp             21/udp          fsp fspd
telnet          23/tcp
telnet          23/udp
tftp            69/tcp
tftp            69/udp
http            80/tcp          www www-http    # WorldWideWeb HTTP
http            80/udp          www www-http    # HyperText Transfer Protocol
http            80/sctp                         # HyperText Transfer Protocol
https           443/tcp                         # http protocol over TLS/SSL
https           443/udp                         # http protocol over TLS/SSL
https           443/sctp                        # http protocol over TLS/SSL
```

2、1024 ~ 49151：用户可分配的端口号

3、49152~65535：动态分配或系统自动分配的端口号

二、网络通信基础

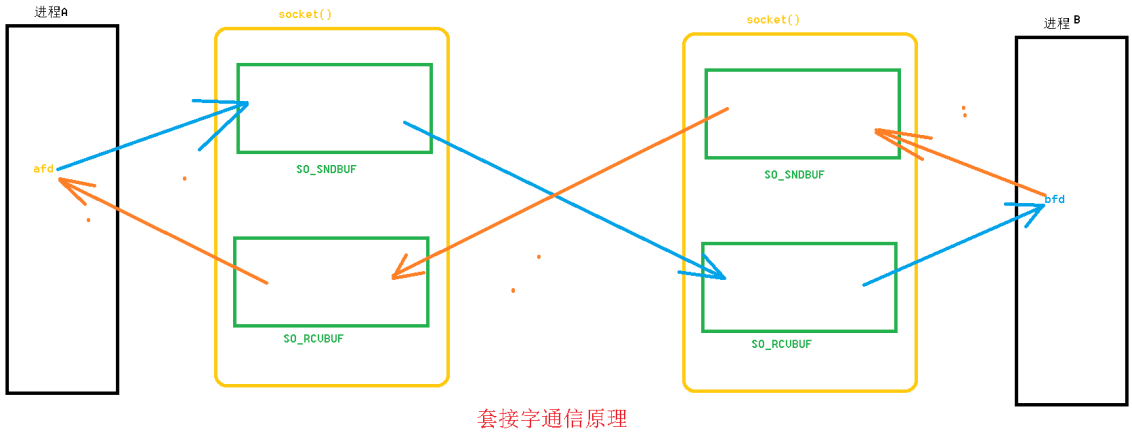
2.1 套接字（socket）的概念

1> 在网络通信过程中，需要创建一个信息的载体来进行数据的通信，这个载体我们可以称之为套接字

2> 我们可以调用函数：socket（），创建一个用于通信的套接字端点，并返回该端点对应的文件描述符

3> 在通信端点中，有两个缓冲区，分别对应发送缓冲区和接受缓冲区

4> 原理如下：



套接字通信原理

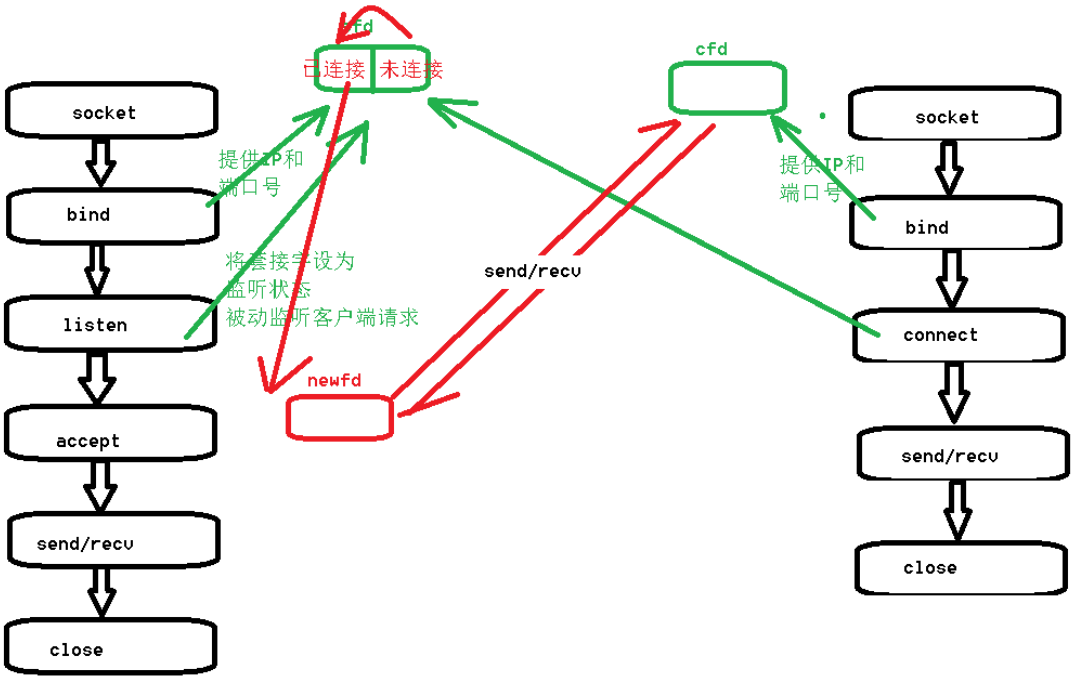
5> socket函数

```
        #include <sys/types.h>          /* See NOTES */
        #include <sys/socket.h>

        int socket(int domain, int type, int protocol);
        功能：为通信创建一个端点，并返回该端点对应的文件描述符，文件描述符的使用原则是最小未分配原则
        参数1：协议族，常用的协议族如下
         Name                  Purpose                                 Man page
        AF_UNIX, AF_LOCAL     本地通信，同一主机的多进程通信            具体内容查看 man 7 unix
        AF_INET               提供IPv4的相关通信方式                    具体内容查看 man 7 ip
        AF_INET6              提供IPv6的相关通信方式                    具体内容查看 man 7 ipv6

        参数2：通信类型，指定通信语义，常用的通信类型如下
                SOCK_STREAM       支持TCP面向连接的通信协议

                SOCK_DGRAM        支持UDP面向无连接的通信协议

        参数3：通信协议，当参数2中明确指定特定协议时，参数3可以设置为0，但是有多个协议共同使用时，需要用参数3指定当前套接字确定的协议
        返回值：成功返回创建的端点对应的文件描述符，失败返回-1并置位错误码
```

2.2 基于TCP面向连接的通信方式

在网络通信过程中，有两种通信方式，分别是基于BS模型的，即浏览器服务器模型（第六章讲解），和基于CS模型，即客户端服务器模型，该章节中，使用的是基于CS模型

1> 通信原理



服务器流程　　客户端流程

2> bind函数

```
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>

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

功能：为套接字分配名称，给套接字绑定ip地址和端口号

参数1：要被绑定的套接字文件描述符

参数2：通用地址信息结构体，对于不同的通信域而言，使用的实际结构体是不同的，该结构体的目的是为了强制类型转换，防止警告

通信域为：AF_INET而言，ipv4的通信方式

```
struct sockaddr_in {
    sa_family_t    sin_family; /* 地址族: AF_INET */
    in_port_t      sin_port;   /* 端口号的网络字节序 */
    struct in_addr sin_addr;   /* 网络地址 */
};

/* Internet address. */
struct in_addr {
    uint32_t       s_addr;     /* ip地址的网络字节序 */
};
```

通信域为：AF_UNIX而言，本地通信

```
struct sockaddr_un {
    sa_family_t sun_family;               /* 通信域：AF_UNIX */
    char        sun_path[UNIX_PATH_MAX];  /* 通信使用的文件 */
};
```

参数3：参数2的大小

返回值：成功返回0，失败返回-1并置位错误码

3> listen函数

```
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>

int listen(int sockfd, int backlog);
功能：将套接字设置成被动监听状态
参数1：套接字文件描述符
参数2：挂起队列能够增长的最大长度，一般为128
返回值：成功返回0，失败返回-1并置位错误码
```

4> accept函数

```
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
功能：阻塞等待客户端的连接请求，如果已连接队列中有客户端，则从连接队列中拿取第一个，并创建一个用于通信的套接字
参数1：服务器套接字文件描述符
参数2：通用地址信息结构体，用于接受已连接的客户端套接字地址信息的
参数3：接收参数2的大小
返回值：成功发那会一个新的用于通信的套接字文件描述符，失败返回-1并置位错误码
```

5> recv、send数据收发

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

ssize_t recv(int sockfd, void *buf, size_t len, int flags);

功能：从套接字中读取消息放入到buf中
参数1：通信的套接字文件描述符
参数2：要存放数据的起始地址
参数3：读取的数据的大小
参数4：读取标识位，是否阻塞读取
          0：表示阻塞等待
          MSG_DONTWAIT：非阻塞
 返回值：可以是大于0：表示成功读取的字节个数
          可以是等于0：表示对端已经下线（针对于TCP通信）
          -1：失败，并置位错误码

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

ssize_t send(int sockfd, const void *buf, size_t len, int flags);
功能：向套接字文件描述符中将buf这个容器中的内容写入
参数1：通信的套接字文件描述符
参数2：要发送的数据的起始地址
参数3：发送的数据的大小
参数4：读取标识位，是否阻塞读取
          0：表示阻塞等待
          MSG_DONTWAIT：非阻塞
 返回值：成功返回发送字节的个数
          -1：失败，并置位错误码
```

6> close关闭套接字

```
#include <unistd.h>

int close(int fd);
功能：关闭套接字文件描述符
参数：要关闭的套接字文件描述符
返回值：成功返回0，失败返回-1并置位错误码
```

7> connect连接函数

```
#include <sys/types.h>          /* See NOTES */
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
功能：将指定的套接字，连接到给定的地址上
参数1：要连接的套接字文件描述符
参数2：通用地址信息结构体
参数3：参数2的大小
返回值：成功返回0，失败返回-1并置位错误码
```

2.3 TCP服务器端和客户端实现

1> 服务器端代码实现

```c
#include <myhead.h>
#define SER_PORT 8888           //服务器端口号
#define SER_IP "192.168.11.53"      //服务器IP地址

int main(int argc, const char *argv[])
{
    //1、创建用于连接的套接字文件描述符
    int sfd = socket(AF_INET, SOCK_STREAM, 0);
    //参数1：AF_INET表示使用的是ipv4的通信协议
    //参数2：SOCK_STREAM表示使用的是tcp通信
    //参数3：由于参数2指定了协议，参数3填0即可
    if(sfd == -1)
    {
        perror("socket error");
        return -1;
    }
    printf("socket success sfd = %d\n", sfd);          //3

    //2、绑定ip地址和端口号
    //2.1 填充要绑定的ip地址和端口号结构体
    struct sockaddr_in sin;
    sin.sin_family = AF_INET;       //通信域
    sin.sin_port = htons(SER_PORT);    //端口号
    sin.sin_addr.s_addr = inet_addr(SER_IP);     //ip地址
```

```
    //2.2 绑定工作
    //参数1：要被绑定的套接字文件描述符
    //参数2：要绑定的地址信息结构体，需要进行强制类型转换，防止警告
    //参数3：参数2的大小
    if(bind(sfd, (struct sockaddr*)&sin, sizeof(sin)) ==-1)
    {
        perror("bind error");
        return -1;
    }
    printf("bind success\n");

    //3、启动监听
    //参数1：要启动监听的文件描述符
    //参数2：挂起队列的长度
    if(listen(sfd, 128) ==-1)
    {
        perror("listen error");
        return -1;
    }
    printf("listen success\n");

    //4、阻塞等待客户端的连接请求
    //定义变量，用于接受客户端地址信息结构体
    struct sockaddr_in cin;             //用于接收地址信息结构体的
    socklen_t socklen = sizeof(cin);        //用于接收地址信息的长度

    int newfd = accept(sfd, (struct sockaddr *)&cin, &socklen);
    //参数1：服务器套接字文件描述符
    //参数2：用于接收客户端地址信息结构体的容器，如果不接收，也可以填NULL
    //参数3：接收参数2的大小，如果参数2为NULL，则参数3也是NULL
    if(newfd == -1)
    {
        perror("accept error");
        return -1;
    }
    printf("[%s:%d]:已连接成功!!!!\n", inet_ntoa(cin.sin_addr),
 ntohs(cin.sin_port));

    //5、数据收发
    char rbuf[128] = "";        //数据容器
    while(1)
    {
        //清空容器中的内容
        bzero(rbuf, sizeof(rbuf));

        //从套接字中读取消息
        int res = recv(newfd, rbuf, sizeof(rbuf), 0);
        if(res == 0)
        {
            printf("对端已经下线\n");
            break;
        }
        printf("[%s:%d]:%s\n", inet_ntoa(cin.sin_addr),  ntohs(cin.sin_port),
rbuf);

        //对收到的数据处理一下，回给客户端
        strcat(rbuf, "*_*");
```

```
        //将消息发送给客户端
        if(send(newfd, rbuf, strlen(rbuf), 0) == -1)
        {
            perror("send error");
            return -1;
        }
        printf("发送成功\n");
    }


    //6、关闭套接字
    close(newfd);
    close(sfd);


    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

2> 客户端代码实现

```
#include <myhead.h>
#define SER_PORT 8888          //服务器端口号
#define SER_IP "192.168.11.53"     //服务器IP地址
#define CLI_PORT 9999           //客户端端口号
#define CLI_IP "192.168.11.53"     //客户端ip地址

int main(int argc, const char *argv[])
{
    //1、创建用于通信的客户端套接字文件描述符
    int cfd = socket(AF_INET, SOCK_STREAM, 0);
    if(cfd == -1)
    {
        perror("socket error");
        return -1;
    }
    printf("socket success cfd = %d\n", cfd);        //3

    //2、绑定ip地址和端口号(可选)
    //2.1 填充要绑定的地址信息结构体
    struct sockaddr_in cin;
    cin.sin_family = AF_INET;
    cin.sin_port = htons(CLI_PORT);
    cin.sin_addr.s_addr = inet_addr(CLI_IP);
    //2.2 绑定工作
    if(bind(cfd, (struct sockaddr*)&cin, sizeof(cin)) == -1)
    {
        perror("bind error");
        return -1;
    }
    printf("bind success\n");

    //3、连接服务器
    //3.1 填充要连接的服务器地址信息结构体
    struct sockaddr_in sin;
    sin.sin_family = AF_INET;     //通信域
    sin.sin_port = htons(SER_PORT); //端口号
```

```
    sin.sin_addr.s_addr = inet_addr(SER_IP);  //服务器ip地址
    //3.2 连接工作
    if(connect(cfd, (struct sockaddr*)&sin, sizeof(sin)) == -1)
    {
        perror("connect error");
        return -1;
    }
    printf("连接服务器成功\n");

    //4、数据收发
    char wbuf[128] = "";
    while(1)
    {
        //清空容器
        bzero(wbuf, sizeof(wbuf));

        //从终端获取数据
        fgets(wbuf, sizeof(wbuf), stdin);
        wbuf[strlen(wbuf)-1] = 0;          //将换行改成 '\0'

        //将数据发送给服务器
        if(send(cfd, wbuf, sizeof(wbuf), 0)==-1)
        {
            perror("send error");
            return -1;
        }

        //接受服务器发送过来的消息
        if(recv(cfd, wbuf, sizeof(wbuf), 0)==0)
        {
            printf("对端已经下线\n");
            break;
        }

        printf("收到服务器消息为：%s\n", wbuf);

    }

    //5、关闭套接字
    close(cfd);


    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

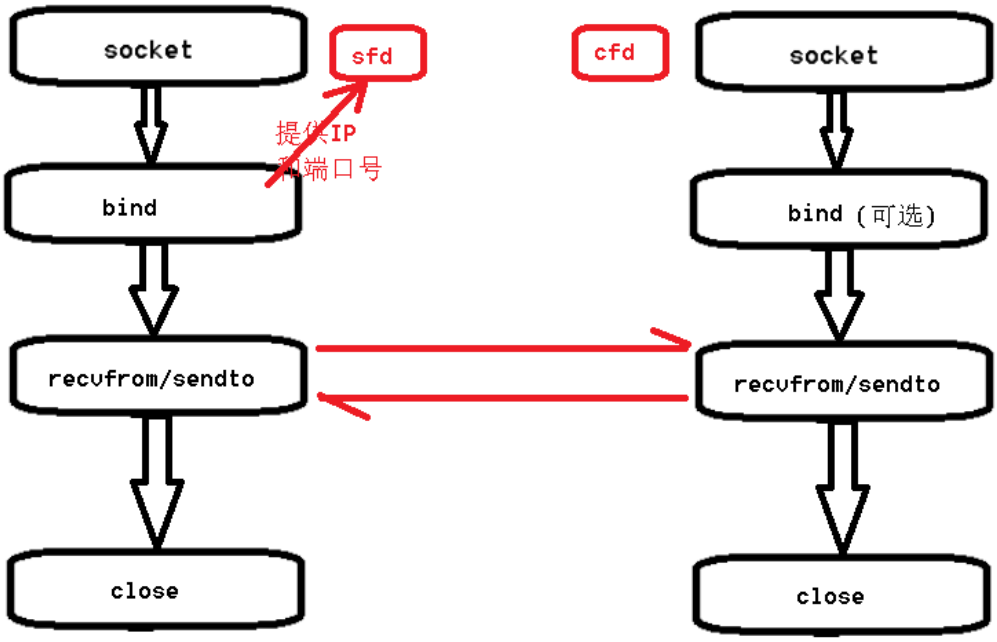
2.4 基于UDP面向无连接的通信方式

1> udp通信是面向无连接的，不可靠的，尽最大努力传输的通信方式

传输过程中，可能会出现数据的丢失、重复、失序、乱序等现象

创建通信套接字是，使用的传输层名称为：SOCK_DGRAM

2> 通信模型



服务器端　　客户端

3>　相关函数

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,struct sockaddr
*src_addr, socklen_t *addrlen);
功能：从套接字中读取消息放入到buf中，并接受对端的地址信息结构体
参数1：套接字文件描述符
参数2：存放数据的容器起始地址
参数3：读取的数据大小
参数4：是否阻塞，0表示阻塞 ，  MSG_DONTWAIT表示非阻塞
参数5：接收对端地址信息结构体的容器
参数6：参数5的大小
返回值：成功返回读取字节的个数，失败返回-1并置位错误码


ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,const struct
sockaddr *dest_addr, socklen_t addrlen);
功能：向套接字中发送消息，并且指定对方的地址信息结构体
参数1：套接字文件描述符
参数2：要发送的数据的起始地址
参数3：要发送的数据大小
参数4：是否阻塞，0表示阻塞 ，  MSG_DONTWAIT表示非阻塞
参数5：要发送的对端地址信息结构体
参数6：参数5的大小
返回值：成功返回发送的字节的个数，失败返回-1并置位错误码
```

2.5 UDP服务器端和客户端实现

1>　UDP服务器端程序代码实现

```
#include <myhead.h>
#define SER_PORT 8888          //服务器端口号
#define SER_IP "192.168.11.53"     //服务器IP地址
```

```c
int main(int argc, const char *argv[])
{
    //1、创建用于通信的套接字文件描述符
    int sfd = socket(AF_INET, SOCK_DGRAM, 0);
    //SOCK_DGRAM表示基于udp通信方式
    if(sfd == -1)
    {
        perror("socket error");
        return -1;
    }
    printf("socket success sfd = %d\n", sfd);       //3


    //2、绑定ip地址和端口号
        //2.1 填充要绑定的ip地址和端口号结构体
    struct sockaddr_in sin;
    sin.sin_family = AF_INET;       //通信域
    sin.sin_port = htons(SER_PORT);   //端口号
    sin.sin_addr.s_addr = inet_addr(SER_IP);    //ip地址

    //2.2 绑定工作
    //参数1：要被绑定的套接字文件描述符
    //参数2：要绑定的地址信息结构体，需要进行强制类型转换，防止警告
    //参数3：参数2的大小
    if(bind(sfd, (struct sockaddr*)&sin, sizeof(sin)) ==-1)
    {
        perror("bind error");
        return -1;
    }
    printf("bind success\n");

    //3、数据收发
    char rbuf[128] = "";
    //定义容器接收对端的地址信息结构体
    struct sockaddr_in cin;
    socklen_t socklen = sizeof(cin);

    while(1)
    {
        //清空容器
        bzero(rbuf, sizeof(rbuf));

        //从客户端中读取消息
        if(recvfrom(sfd, rbuf, sizeof(rbuf), 0, (struct sockaddr*)&cin,
&socklen)==-1)
        {
            perror("recvfrom error");
            return -1;
        }

        printf("[%s:%d]:%s\n", inet_ntoa(cin.sin_addr), ntohs(cin.sin_port),
rbuf);

        //加个笑脸发给客户端
        strcat(rbuf, "*_*");

        //将数据发送给客户端
        sendto(sfd, rbuf, strlen(rbuf), 0, (struct sockaddr*)&cin, sizeof(cin));
```

```
        printf("发送成功\n");

    }

    //4、关闭套接字
    close(sfd);

    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

2> UDP客户端程序实现

```
#include <myhead.h>
#define SER_PORT 8888           //服务器端口号
#define SER_IP "192.168.11.53"      //服务器IP地址
#define CLI_PORT 9999     //客户端端口号
#define CLI_IP "192.168.11.53"    //客户端IP地址

int main(int argc, const char *argv[])
{
    //1、创建用于通信的客户端套接字文件描述符
    int cfd = socket(AF_INET, SOCK_DGRAM, 0);
    if(cfd == -1)
    {
        perror("socket error");
        return -1;
    }
    printf("socket success cfd = %d\n", cfd);       //3

    //2、绑定ip地址和端口号（可选）
    //2.1 填充要绑定的地址信息结构体
    struct sockaddr_in cin;
    cin.sin_family = AF_INET;
    cin.sin_port = htons(CLI_PORT);
    cin.sin_addr.s_addr = inet_addr(CLI_IP);
    //2.2 绑定工作
    if(bind(cfd, (struct sockaddr*)&cin, sizeof(cin)) == -1)
    {
        perror("bind error");
        return -1;
    }
    printf("bind success\n");



    //3、数据收发
    char wbuf[128] = "";
    //填充服务器的地址信息结构体
    struct sockaddr_in sin;
    sin.sin_family = AF_INET;       //通信域
    sin.sin_port = htons(SER_PORT);   //端口号
    sin.sin_addr.s_addr = inet_addr(SER_IP);   //ip地址

    while(1)
    {
        //清空容器
```

```
        bzero(wbuf, sizeof(wbuf));

        //从终端读取数据
        fgets(wbuf, sizeof(wbuf), stdin);
        wbuf[strlen(wbuf)-1] = 0;

        //将数据发送给服务器
        sendto(cfd, wbuf, strlen(wbuf), 0, (struct sockaddr*)&sin, sizeof(sin));

        //接收服务器发来的数据
        recvfrom(cfd, wbuf, sizeof(wbuf), 0, NULL, NULL);

        printf("服务器发来的消息为：%s\n", wbuf);

    }

    //4、关闭套接字
    close(cfd);


    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

三、TCP并发服务器

3.1 并发服务器的引入

对于tcp通信方式而言，目前的通信方式，只能实现一个服务器端对应一个客户端，不能实现一个服务器对应多个客户端，为了完成该操作，我们需要引入并发操作

1> 能够实现一个服务器端对应多个客户端的操作：循环服务器

```
#include <myhead.h>
#define SER_PORT 8888          //服务器端口号
#define SER_IP "192.168.31.49"      //服务器IP地址

int main(int argc, const char *argv[])
{
    //1、创建用于连接的套接字文件描述符
    int sfd = socket(AF_INET, SOCK_STREAM, 0);
    //参数1：AF_INET表示使用的是ipv4的通信协议
    //参数2：SOCK_STREAM表示使用的是tcp通信
    //参数3：由于参数2指定了协议，参数3填0即可
    if(sfd == -1)
    {
        perror("socket error");
        return -1;
    }
    printf("socket success sfd = %d\n", sfd);          //3

    //2、绑定ip地址和端口号
    //2.1 填充要绑定的ip地址和端口号结构体
    struct sockaddr_in sin;
```

```
    sin.sin_family = AF_INET;        //通信域
    sin.sin_port = htons(SER_PORT);   //端口号
    sin.sin_addr.s_addr = inet_addr(SER_IP);    //ip地址

    //2.2 绑定工作
    //参数1：要被绑定的套接字文件描述符
    //参数2：要绑定的地址信息结构体，需要进行强制类型转换，防止警告
    //参数3：参数2的大小
    if(bind(sfd, (struct sockaddr*)&sin, sizeof(sin)) ==-1)
    {
        perror("bind error");
        return -1;
    }
    printf("bind success\n");

    //3、启动监听
    //参数1：要启动监听的文件描述符
    //参数2：挂起队列的长度
    if(listen(sfd, 128) ==-1)
    {
        perror("listen error");
        return -1;
    }
    printf("listen success\n");

    //4、阻塞等待客户端的连接请求
    //定义变量，用于接受客户端地址信息结构体
    struct sockaddr_in cin;              //用于接收地址信息结构体的
    socklen_t socklen = sizeof(cin);         //用于接收地址信息的长度

    while(1)
    {
        int newfd = accept(sfd, (struct sockaddr *)&cin, &socklen);
        //参数1：服务器套接字文件描述符
        //参数2：用于接收客户端地址信息结构体的容器，如果不接收，也可以填NULL
        //参数3：接收参数2的大小，如果参数2为NULL，则参数3也是NULL
        if(newfd == -1)
        {
            perror("accept error");
            return -1;
        }
        printf("[%s:%d]:已连接成功，newfd = %d!!!!\n", inet_ntoa(cin.sin_addr),
ntohs(cin.sin_port), newfd);

        //5、数据收发
        char rbuf[128] = "";        //数据容器
        while(1)
        {
            //清空容器中的内容
            bzero(rbuf, sizeof(rbuf));

            //从套接字中读取消息
            int res = recv(newfd, rbuf, sizeof(rbuf), 0);
            if(res == 0)
            {
                printf("对端已经下线\n");
                break;
            }
```

```
            printf("[%s:%d]:%s\n", inet_ntoa(cin.sin_addr),
 ntohs(cin.sin_port), rbuf);

            //对收到的数据处理一下，回给客户端
            strcat(rbuf, "*_*");

            //将消息发送给客户端
            if(send(newfd, rbuf, strlen(rbuf), 0) == -1)
            {
                perror("send error");
                return -1;
            }
            printf("发送成功\n");
        }


        //6、关闭套接字
        close(newfd);

    }
    close(sfd);


    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

2> 循环服务器的模型

```
sfd = socket();    //创建用于链接的套接字文件描述符
bind();             //绑定ip地址和端口号
listen();            //启动被动监听状态

while(1)
{
        newfd = accept();       //阻塞接受客户端连接请求

        //跟客户端进行数据收发
        recv();
        send();
        close(newfd);
}
close(sfd);
```

3> 通过上述案例，我们可以发现，accept函数和recv函数都是阻塞函数，但是，我们想让这两个阻塞任务同时执行，互不相干扰，此时，我们就要引入并发机制：多进程、多线程、IO多路复用

3.2 多进程实现并发服务器

1> 原理：主进程可以用于完成对客户端的连接请求，子进程可以完成对客户端通信操作

2> 代码实现

```
```

```c
#include <myhead.h>
#define SER_PORT 8888          //服务器端口号
#define SER_IP "192.168.31.49"      //服务器IP地址


//自定义信号处理函数
void handler(int signo)
{
    if(signo == SIGCHLD)
    {
        while(waitpid(-1, NULL, WNOHANG) > 0);        //循环以非阻塞的形式回收僵尸进程
    }
}


/********************主程序****************************** */
int main(int argc, const char *argv[])
{

    //将子进程发送的SIGCHLD信号连接到自定义信号处理函数中
    if(signal(SIGCHLD, handler) == SIG_ERR)
    {
        perror("signal error");
        return -1;
    }

    //1、创建用于连接的套接字文件描述符
    int sfd = socket(AF_INET, SOCK_STREAM, 0);
    //参数1：AF_INET表示使用的是ipv4的通信协议
    //参数2：SOCK_STREAM表示使用的是tcp通信
    //参数3：由于参数2指定了协议，参数3填0即可
    if(sfd == -1)
    {
        perror("socket error");
        return -1;
    }
    printf("socket success sfd = %d\n", sfd);          //3

    //2、绑定ip地址和端口号
    //2.1 填充要绑定的ip地址和端口号结构体
    struct sockaddr_in sin;
    sin.sin_family = AF_INET;        //通信域
    sin.sin_port = htons(SER_PORT);   //端口号
    sin.sin_addr.s_addr = inet_addr(SER_IP);    //ip地址

    //2.2 绑定工作
    //参数1：要被绑定的套接字文件描述符
    //参数2：要绑定的地址信息结构体，需要进行强制类型转换，防止警告
    //参数3：参数2的大小
    if(bind(sfd, (struct sockaddr*)&sin, sizeof(sin)) ==-1)
    {
        perror("bind error");
        return -1;
    }
    printf("bind success\n");

    //3、启动监听
    //参数1：要启动监听的文件描述符
```

```
    //参数2：挂起队列的长度
    if(listen(sfd, 128) ==-1)
    {
        perror("listen error");
        return -1;
    }
    printf("listen success\n");

    //4、阻塞等待客户端的连接请求
    //定义变量，用于接受客户端地址信息结构体
    struct sockaddr_in cin;              //用于接收地址信息结构体的
    socklen_t socklen = sizeof(cin);         //用于接收地址信息的长度

    while(1)
    {
        int newfd = accept(sfd, (struct sockaddr *)&cin, &socklen);
        //参数1：服务器套接字文件描述符
        //参数2：用于接收客户端地址信息结构体的容器，如果不接收，也可以填NULL
        //参数3：接收参数2的大小，如果参数2为NULL，则参数3也是NULL
        if(newfd == -1)
        {
            perror("accept error");
            return -1;
        }
        printf("[%s:%d]:已连接成功，newfd = %d!!!!\n", inet_ntoa(cin.sin_addr),
ntohs(cin.sin_port), newfd);

        //创建子进程，用于跟客户端进行通信
        pid_t pid = fork();
        if(pid > 0)
        {
            //父进程内容
            //wait(NULL);       //?    不可以使用wait，因为该函数时阻塞函数
            //waitpid(-1, NULL, WNOHANG);    //以非阻塞形式回收？
            //关闭newfd因为，父进程不对newfd进行操作
            close(newfd);

        }else if(pid == 0)
        {
            //关闭sfd，因为子进程主要专注于跟客户端的通信，主要使用newfd
            close(sfd);

            //5、数据收发
            char rbuf[128] = "";        //数据容器
            while(1)
            {
                //清空容器中的内容
                bzero(rbuf, sizeof(rbuf));

                //从套接字中读取消息
                int res = recv(newfd, rbuf, sizeof(rbuf), 0);
                if(res == 0)
                {
                    printf("对端已经下线\n");
                    break;
                }
                printf("[%s:%d]:%s\n", inet_ntoa(cin.sin_addr),
ntohs(cin.sin_port), rbuf);
```

```
                //对收到的数据处理一下，回给客户端
                strcat(rbuf, "*_*");

                //将消息发送给客户端
                if(send(newfd, rbuf, strlen(rbuf), 0) == -1)
                {
                    perror("send error");
                    return -1;
                }
                printf("发送成功\n");
            }


            //6、关闭套接字
            close(newfd);

            //退出子进程
            exit(EXIT_SUCCESS);
        }else
        {
            perror("fork error");
            return -1;
        }


    }
    close(sfd);     //关闭监听


    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

3> 总结模型

```
void handler(int signo)
{
    while(waitpid() > 0);       //以非阻塞形式回收僵尸进程
}


signal(SIGCHLD, handler);        //将信号与信号处理函数连接起来


sfd = socket();    //创建用于链接的套接字文件描述符
bind();             //绑定ip地址和端口号
listen();            //启动被动监听状态

while(1)
{
        newfd = accept();       //阻塞接受客户端连接请求

        pid = fork();     //创建子进程用于跟客户端进行通信
        if(pid > 0)
        {
```

```
            //关闭newfd
            close(newfd);
        }else if(pid == 0)
        {
            //子进程
            close(sfd);
            //跟客户端进行数据收发
            recv();
            send();
            close(newfd);

            exit(EXIT_SUCCESS);     //退出子进程
        }

}
close(sfd);
```

3.3 多线程实现并发服务器

1> 原理：主线程用于接收客户端的连接请求， 分支线程用于跟客户端进行通信

2> 代码实现

```
#include <myhead.h>
#define SER_PORT 8888           //服务器端口号
#define SER_IP "192.168.31.49"      //服务器IP地址

//定义用于传送数据的结构体类型
struct Info
{
    int newfd;      //套接字文件描述符
    struct sockaddr_in cin;     //客户端套接字地址信息结构体
};


//分支线程，用户跟客户端进行通信
void *deal_cli_msg(void *arg)
{
    //解析传过来的数据
    int newfd = ((struct Info *)arg)->newfd;     //解析套接字文件描述符
    struct sockaddr_in cin = ((struct Info *)arg)->cin;

//5、数据收发
        char rbuf[128] = "";        //数据容器
        while(1)
        {
            //清空容器中的内容
            bzero(rbuf, sizeof(rbuf));

            //从套接字中读取消息
            int res = recv(newfd, rbuf, sizeof(rbuf), 0);
            if(res == 0)
            {
                printf("对端已经下线\n");
                break;
```

```
            }
            printf("[%s:%d]:%s\n", inet_ntoa(cin.sin_addr),
 ntohs(cin.sin_port), rbuf);

            //对收到的数据处理一下，回给客户端
            strcat(rbuf, "*_*");

            //将消息发送给客户端
            if(send(newfd, rbuf, strlen(rbuf), 0) == -1)
            {
                perror("send error");
                return NULL;
            }
            printf("发送成功\n");
        }


        //6、关闭套接字
        close(newfd);

        //退出分支线程
        pthread_exit(NULL);
}



/*******************************主程序********************************** */
int main(int argc, const char *argv[])
{
    //1、创建用于连接的套接字文件描述符
    int sfd = socket(AF_INET, SOCK_STREAM, 0);
    //参数1：AF_INET表示使用的是ipv4的通信协议
    //参数2：SOCK_STREAM表示使用的是tcp通信
    //参数3：由于参数2指定了协议，参数3填0即可
    if(sfd == -1)
    {
        perror("socket error");
        return -1;
    }
    printf("socket success sfd = %d\n", sfd);          //3

    //2、绑定ip地址和端口号
    //2.1 填充要绑定的ip地址和端口号结构体
    struct sockaddr_in sin;
    sin.sin_family = AF_INET;        //通信域
    sin.sin_port = htons(SER_PORT);   //端口号
    sin.sin_addr.s_addr = inet_addr(SER_IP);    //ip地址

    //2.2 绑定工作
    //参数1：要被绑定的套接字文件描述符
    //参数2：要绑定的地址信息结构体，需要进行强制类型转换，防止警告
    //参数3：参数2的大小
    if(bind(sfd, (struct sockaddr*)&sin, sizeof(sin)) ==-1)
    {
        perror("bind error");
        return -1;
    }
    printf("bind success\n");
```

```
    //3、启动监听
    //参数1：要启动监听的文件描述符
    //参数2：挂起队列的长度
    if(listen(sfd, 128) ==-1)
    {
        perror("listen error");
        return -1;
    }
    printf("listen success\n");

    //4、阻塞等待客户端的连接请求
    //定义变量，用于接受客户端地址信息结构体
    struct sockaddr_in cin;              //用于接收地址信息结构体的
    socklen_t socklen = sizeof(cin);         //用于接收地址信息的长度

    while(1)
    {

        //对于accept函数而言，当程序执行到该函数时，会给当前客户端预分配一个最小未使用的文件描述符
        //后期再有新的小的文件描述符，也不使用了，新的最小的，用于下一次客户端的选定
        int newfd = accept(sfd, (struct sockaddr *)&cin, &socklen);
        //参数1：服务器套接字文件描述符
        //参数2：用于接收客户端地址信息结构体的容器，如果不接收，也可以填NULL
        //参数3：接收参数2的大小，如果参数2为NULL，则参数3也是NULL
        if(newfd == -1)
        {
            perror("accept error");
            return -1;
        }
        printf("[%s:%d]:已连接成功，newfd = %d!!!!\n", inet_ntoa(cin.sin_addr), ntohs(cin.sin_port), newfd);

        //填充要传输给分支线程的数据结构体变量
        struct Info buf = {newfd, cin};

        //创建分支线程，用于处理客户端的操作
        pthread_t tid = -1;
        if(pthread_create(&tid, NULL, deal_cli_msg, &buf) != 0)
        {
            printf("pthread_create error\n");
            return -1;
        }

        //完成对分支线程资源的处理
        //pthread_join(tid, NULL);         //?  不能使用该函数回收，因为该函数时阻塞函数
        pthread_detach(tid);         //将线程设置成分离态，执行结束后，系统自动回收资源


    }
    close(sfd);


    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

3> 通信模型

```
//定义分支线程
void *deal_cli_msg(void *arg)
{
//跟客户端进行数据收发
        recv();
        send();
        close(newfd);
        //退出线程
        pthread_exit(NULL);
}


//主线程内容
sfd = socket();    //创建用于链接的套接字文件描述符
bind();             //绑定ip地址和端口号
listen();            //启动被动监听状态

while(1)
{
        newfd = accept();       //阻塞接受客户端连接请求

        pthread_create(&tid, NULL, deal_cli_msg, &buf);    //创建分支线程

        //将线程设置成分离态
        pthread_detach(tid);


}
close(sfd);
```

3.4 IO多路复用

1> 引入目的：当主机没有操作系统时，或者说程序不能使用多进程或多线程完成任务的并发操作时，我们可以引入IO多路复用的技术，完成多任务并发执行的操作

2> 事件和函数的关系：比如，scanf是一个阻塞函数，该函数完成的是输入事件

当函数先于事件发生时，函数会阻塞等待事件的到来

当函数后于事件发生时，函数就不会阻塞，直接执行

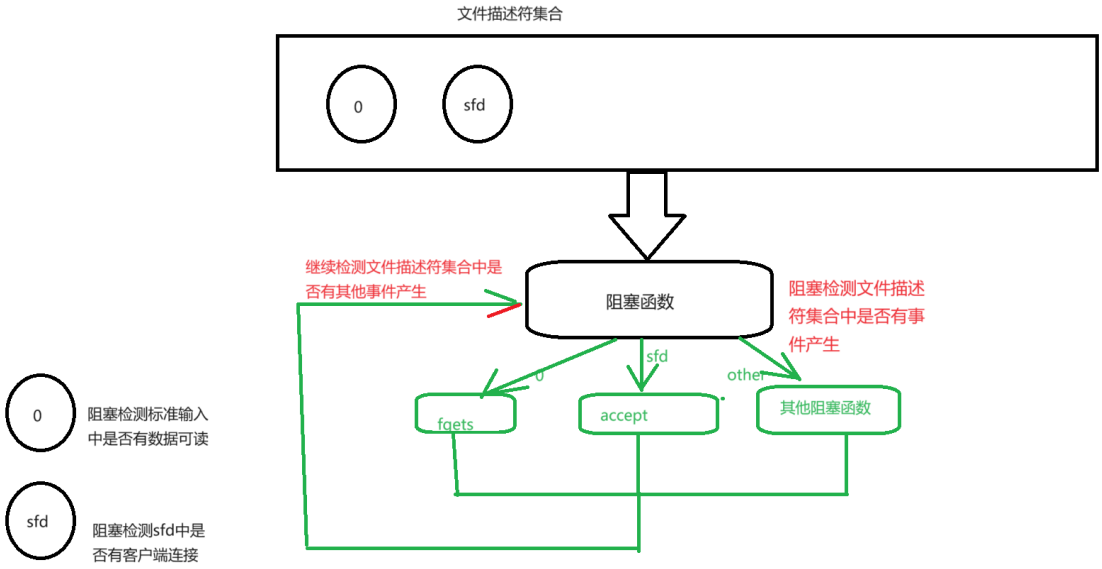
```
#include <myhead.h>

int main(int argc, const char *argv[])
{
    int num = 0, key;          //定义一个整型变量

    printf("请输入num的值：");
    scanf("%d", &num);             //提示并输入第一个值
    printf("num = %d\n", num);
```

```
    printf("请输入key的值：");
    scanf("%d", &key);              //提示并输入第二个值
    printf("num = %d\n", key);

    std::cout << "Hello, World!" << std::endl;
   return 0;
}
```

```
[zpp@MiWiFi-R4A-srv day4]$ ./a.out
请输入num的值：520
num = 520
请输入key的值：1314
num = 1314
Hello, World!
[zpp@MiWiFi-R4A-srv day4]$ ./a.out
请输入num的值：520 1314
num = 520
请输入key的值：num = 1314
Hello, World!
[zpp@MiWiFi-R4A-srv day4]$
```

从上述例子中的第二个输入可以看出，如果事件先于函数发生，当程序再执行到函数时，函数会直接运行，不会阻塞了

3> IO多路复用原理图



4> IO多路复用相关函数有两个：select、poll

5> select函数：如果man手册中没有select函数，执行指令 sudo yum install man-pages

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

        int select(int nfds, fd_set *readfds, fd_set *writefds,fd_set *exceptfds,
struct timeval *timeout);
        功能：阻塞等待文件描述符集合中是否有事件产生，如果有事件产生，则解除阻塞
        参数1：文件描述符集合中，最大的文件描述符 加1
```

参数2、参数3、参数4：分别表示读集合、写集合、异常处理集合的起始地址

由于对于写操作而言，我们也可以转换读操作，所以，只需要使用一个集合就行

对于不使用的集合而言，直接填NULL即可

参数5：超时时间，如果填NULL表示永久等待，如果想要设置时间，需要定义一个如下结构体类型的变量，并将地址传递进去

```
struct timeval {
    long    tv_sec;         /* 秒数 */
    long    tv_usec;        /* 微秒 */
};

and
struct timespec {
    long    tv_sec;         /* 秒数 */
    long    tv_nsec;        /* 纳秒 */
};
```

返回值：

>0:成功返回解除本次阻塞的文件描述符的个数

=0:表示设置的超时时间，时间已经到达，但是没有事件事件产生

=-1：表示失败，置位错误码

注意：当该函数解除阻塞时，文件描述符集合中，就只剩下本次触发事件的文件描述符，其余的文件描述符就被删除了

```
//专门针对于文件描述符集合提供的函数
void FD_CLR(int fd, fd_set *set);       //将fd文件描述符从容器set中删除
int  FD_ISSET(int fd, fd_set *set);     //判断fd文件描述符，是否存在于set容器中
void FD_SET(int fd, fd_set *set);      //将fd文件描述符，放入到set容器中
void FD_ZERO(fd_set *set);              //清空set容器
```

6> select函数的基本使用方式

```
#include <myhead.h>
#define SER_PORT 8888           //服务器端口号
#define SER_IP "192.168.31.49"      //服务器IP地址

int main(int argc, const char *argv[])
{
    //1、创建用于连接的套接字文件描述符
    int sfd = socket(AF_INET, SOCK_STREAM, 0);
    //参数1：AF_INET表示使用的是ipv4的通信协议
    //参数2：SOCK_STREAM表示使用的是tcp通信
    //参数3：由于参数2指定了协议，参数3填0即可
    if(sfd == -1)
    {
        perror("socket error");
        return -1;
    }
    printf("socket success sfd = %d\n", sfd);          //3

    //2、绑定ip地址和端口号
    //2.1 填充要绑定的ip地址和端口号结构体
    struct sockaddr_in sin;
    sin.sin_family = AF_INET;       //通信域
    sin.sin_port = htons(SER_PORT);   //端口号
    sin.sin_addr.s_addr = inet_addr(SER_IP);    //ip地址
```

```c
//2.2 绑定工作
//参数1：要被绑定的套接字文件描述符
//参数2：要绑定的地址信息结构体，需要进行强制类型转换，防止警告
//参数3：参数2的大小
if(bind(sfd, (struct sockaddr*)&sin, sizeof(sin)) ==-1)
{
    perror("bind error");
    return -1;
}
printf("bind success\n");

//3、启动监听
//参数1：要启动监听的文件描述符
//参数2：挂起队列的长度
if(listen(sfd, 128) ==-1)
{
    perror("listen error");
    return -1;
}
printf("listen success\n");

//4、阻塞等待客户端的连接请求
//定义变量，用于接受客户端地址信息结构体
struct sockaddr_in cin;              //用于接收地址信息结构体的
socklen_t socklen = sizeof(cin);         //用于接收地址信息的长度

//定义文件描述符集合
fd_set readfds, tempfds;        //读文件描述符集合

//将该文件描述符集合清空
FD_ZERO(&readfds);

//将0号文件描述符以及sfd文件描述符放入到集合中
FD_SET(0, &readfds);
FD_SET(sfd, &readfds);




while(1)
{
    //将reafds备份一份放入tempfds中
    tempfds = readfds;

    //调用阻塞函数，完成对文件描述符集合的管理工作
    int res = select(sfd+1, &tempfds, NULL, NULL, NULL);
    if(res == -1)
    {
        perror("select error");
        return -1;
    }else if(res == 0)
    {
        printf("time out !!!\n");
        return -1;
    }
```

```
        //程序执行至此，表示一定有其中至少一个文件描述符产生了事件，只需要判断哪个文件描述符还在集合中
        //就说明该文件描述符产生了事件

        //表示sfd文件描述符触发了事件
        if(FD_ISSET(sfd, &tempfds))
        {
            int newfd = accept(sfd, (struct sockaddr *)&cin, &socklen);
            //参数1：服务器套接字文件描述符
            //参数2：用于接收客户端地址信息结构体的容器，如果不接收，也可以填NULL
            //参数3：接收参数2的大小，如果参数2为NULL，则参数3也是NULL
            if(newfd == -1)
            {
                perror("accept error");
                return -1;
            }
            printf("[%s:%d]:已连接成功，newfd = %d!!!!\n", inet_ntoa(cin.sin_addr),  ntohs(cin.sin_port), newfd);


        }




        //判断0号文件描述符是否产生了事件
        if(FD_ISSET(0, &tempfds))
        {
            char wbuf[128] = "";     //字符数组
            fgets(wbuf, sizeof(wbuf), stdin);       //从终端读取数据,阻塞函数
            printf("触发了键盘输入事件：%s\n", wbuf);
        }


    }
    close(sfd);


    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

7> select 实现TCP并发服务器

```
#include <myhead.h>
#define SER_PORT 8888          // 服务器端口号
#define SER_IP "192.168.174.128" // 服务器IP地址

int main(int argc, const char *argv[])
{
    // 1、创建用于连接的套接字文件描述符
    int sfd = socket(AF_INET, SOCK_STREAM, 0);
    // 参数1：AF_INET表示使用的是ipv4的通信协议
    // 参数2：SOCK_STREAM表示使用的是tcp通信
    // 参数3：由于参数2指定了协议，参数3填0即可
    if (sfd == -1)
    {
```

```
        perror("socket error");
        return -1;
    }
    printf("socket success sfd = %d\n", sfd); // 3

    // 2、绑定ip地址和端口号
    // 2.1 填充要绑定的ip地址和端口号结构体
    struct sockaddr_in sin;
    sin.sin_family = AF_INET;                   // 通信域
    sin.sin_port = htons(SER_PORT);             // 端口号
    sin.sin_addr.s_addr = inet_addr(SER_IP); // ip地址

    // 2.2 绑定工作
    // 参数1：要被绑定的套接字文件描述符
    // 参数2：要绑定的地址信息结构体，需要进行强制类型转换，防止警告
    // 参数3：参数2的大小
    if (bind(sfd, (struct sockaddr *)&sin, sizeof(sin)) == -1)
    {
        perror("bind error");
        return -1;
    }
    printf("bind success\n");

    // 3、启动监听
    // 参数1：要启动监听的文件描述符
    // 参数2：挂起队列的长度
    if (listen(sfd, 128) == -1)
    {
        perror("listen error");
        return -1;
    }
    printf("listen success\n");

    // 4、阻塞等待客户端的连接请求
    // 定义变量，用于接受客户端地址信息结构体
    struct sockaddr_in cin;           // 用于接收地址信息结构体的
    socklen_t socklen = sizeof(cin); // 用于接收地址信息的长度

    // 定义文件描述符集合
    fd_set readfds, tempfds; // 读文件描述符集合

    // 将该文件描述符集合清空
    FD_ZERO(&readfds);

    // 将0号文件描述符以及sfd文件描述符放入到集合中
    FD_SET(0, &readfds);
    FD_SET(sfd, &readfds);

    // 定义一个变量，用于存储容器中的最大文件描述符
    int maxfd = sfd;

    int newfd = -1; // 接收客户端连接请求后，创建的通信套接字文件描述符
    //定义一个地址信息结构体数组来存储客户端对应的地址信息
    struct sockaddr_in cin_arr[1024];

    while (1)
    {
        // 将reafds备份一份放入tempfds中
```

```
        tempfds = readfds;

        // 调用阻塞函数，完成对文件描述符集合的管理工作
        int res = select(maxfd + 1, &tempfds, NULL, NULL, NULL);
        if (res == -1)
        {
            perror("select error");
            return -1;
        }
        else if (res == 0)
        {
            printf("time out !!!\n");
            return -1;
        }

        // 程序执行至此，表示一定有其中至少一个文件描述符产生了事件，只需要判断哪个文件描述符
还在集合中
        // 就说明该文件描述符产生了事件

        // 表示sfd文件描述符触发了事件
        if (FD_ISSET(sfd, &tempfds))
        {
            newfd = accept(sfd, (struct sockaddr *)&cin, &socklen);
            // 参数1：服务器套接字文件描述符
            // 参数2：用于接收客户端地址信息结构体的容器，如果不接收，也可以填NULL
            // 参数3：接收参数2的大小，如果参数2为NULL，则参数3也是NULL
            if (newfd == -1)
            {
                perror("accept error");
                return -1;
            }
            printf("[%s:%d]:已连接成功，newfd = %d!!!!\n",
inet_ntoa(cin.sin_addr), ntohs(cin.sin_port), newfd);

            //将该客户端对应的套接字地址信息结构体放入数组对应的位置上
            cin_arr[newfd] = cin;           //newfd文件描述符对应的地址信息结构体未
cin_arr[newfd]

            // 将当前的newfd放入到检测文件描述符集合中，以便于检测使用
            FD_SET(newfd, &readfds); // 加入到tempfds中
            // 更新maxfd，如何更新？
            if (maxfd < newfd) // 判断最新的文件描述符是否比当前容器中最大的文件描述符大
            {
                maxfd = newfd;
            }
        }

        // 判断0号文件描述符是否产生了事件
        if (FD_ISSET(0, &tempfds))
        {
            char wbuf[128] = "";               // 字符数组
            fgets(wbuf, sizeof(wbuf), stdin); // 从终端读取数据,阻塞函数
            printf("触发了键盘输入事件：%s\n", wbuf);
            //能不能将输入的数据，全部发送给所有客户端
            for(int i=4; i<=maxfd; i++)
            {
                send(i, wbuf, strlen(wbuf), 0);      //将数据发送给所有客户端
            }
```

```
        }

        // 判断是否是newfd产生了事件
        // 循环将所有客户端文件描述符遍历一遍，如果还存在于tempfds中的客户端，表示有数据接收过来
        for (int i = 4; i <= maxfd; i++)
        {
            if (FD_ISSET(i, &tempfds))
            {
                // 5、数据收发
                char rbuf[128] = ""; // 数据容器

                // 清空容器中的内容
                bzero(rbuf, sizeof(rbuf));

                // 从套接字中读取消息
                int res = recv(i, rbuf, sizeof(rbuf), 0);
                if (res == 0)
                {
                    printf("对端已经下线\n");
                    // 将文件描述符进行关闭
                    close(i);
                    //需要将该文件描述符从readfds中删除
                    FD_CLR(i, &readfds);

                    // 更新maxfd
                    for(int k=maxfd; k>=0; k--)
                    {
                        if(FD_ISSET(k, &readfds))
                        {
                            maxfd = k;
                            break;        //结束向下进行的循环
                        }
                    }

                    continue; // 本次循环结束，继续下一次的select的阻塞
                }
                printf("[%s:%d]:%s\n", inet_ntoa(cin_arr[i].sin_addr), ntohs(cin_arr[i].sin_port), rbuf);

                // 对收到的数据处理一下，回给客户端
                strcat(rbuf, "*_*");

                // 将消息发送给客户端
                if (send(i, rbuf, strlen(rbuf), 0) == -1)
                {
                    perror("send error");
                    return -1;
                }
                printf("发送成功\n");
            }
        }
    }
    close(sfd);

    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

8> select实现TCP并发服务器的模型

```
sfd = socket();         //创建用于连接的套接字文件描述符
bind();                 //绑定ip和端口号
listen();                //监听

fd_set  readfds, tempfds;    //定义文件描述符集合
FD_ZERO();         //清空容器
FD_SET();          //将文件描述符放入容器

maxfd = sfd;          //记录最大的文件描述符

while(1)
{
    tempfds = readfds;       //备份一份容器
    select(maxfd, &readfds, NULL, NULL, NULL);      //阻塞等待集合中是否有事件产生

    //判断相关文件描述符是否在集合中
    if(FD_ISSET(sfd, &tempfds))
    {
        newfd = accept();       //接收客户端请求
        FD_SET(newfd, &readfds);     //将新文件描述符放入集合
        //更新maxfd
    }

    //判断是否是客户端发来数据
    for(i=4; i<=maxfd; i++)
    {
        send();
        recv();
        close(i);      //退出客户端
        FD_CLR(i, &readfds);
        //更新maxfd
    }

}

//关闭监听
close(sfd);
```

9> 使用poll实现IO多路复用

```
        #include <poll.h>

        int poll(struct pollfd *fds, nfds_t nfds, int timeout);
        功能：阻塞等待文件描述符集合中是否有事件产生，如果有，则解除阻塞，返回本次触发事件的文件描述符个数
        参数1：文件描述符集合容器的起始地址，是一个结构体数组，结构体类型如下
             struct pollfd {
                int   fd;         /* 文件描述符 */
                short events;     /* 要等待的事件：由用户填写 */
                short revents;    /* 实际发生的事件 ：调用函数结束后，内核会自动设置*/
```

```
            };
            关于事件对应的位：
            POLLIN:读事件
            POLLOUT：写事件

        参数2：集合中文件描述符的个数
        参数3：超时时间，负数表示永久等待，0表示非阻塞
        返回值：
            >0:表示触发本次解除阻塞事件的文件描述符的个数
            =0:表示超时
            =-1:出错，置位错误码
```

10> poll的使用实例：poll完成TCP客户端中发送数据和读取数据的并发

```c
#include <myhead.h>
#define SER_PORT 8888             // 服务器端口号
#define SER_IP "192.168.174.128" // 服务器IP地址
#define CLI_PORT 9999             // 客户端端口号
#define CLI_IP "192.168.174.128" // 客户端ip地址

int main(int argc, const char *argv[])
{
    // 1、创建用于通信的客户端套接字文件描述符
    int cfd = socket(AF_INET, SOCK_STREAM, 0);
    if (cfd == -1)
    {
        perror("socket error");
        return -1;
    }
    printf("socket success cfd = %d\n", cfd); // 3

    // 2、绑定ip地址和端口号(可选)
    // 2.1 填充要绑定的地址信息结构体
    struct sockaddr_in cin;
    cin.sin_family = AF_INET;
    cin.sin_port = htons(CLI_PORT);
    cin.sin_addr.s_addr = inet_addr(CLI_IP);
    // 2.2 绑定工作
    if (bind(cfd, (struct sockaddr *)&cin, sizeof(cin)) == -1)
    {
        perror("bind error");
        return -1;
    }
    printf("bind success\n");

    // 3、连接服务器
    // 3.1 填充要连接的服务器地址信息结构体
    struct sockaddr_in sin;
    sin.sin_family = AF_INET;                 // 通信域
    sin.sin_port = htons(SER_PORT);           // 端口号
    sin.sin_addr.s_addr = inet_addr(SER_IP); // 服务器ip地址
    // 3.2 连接工作
    if (connect(cfd, (struct sockaddr *)&sin, sizeof(sin)) == -1)
    {
```

```
        perror("connect error");
        return -1;
    }
    printf("连接服务器成功\n");

    // 使用poll完成终端输入和套接字接收数据的并发执行
    struct pollfd pfds[2]; // pfds[0]   pfds[1]
    // 分别给数组中两个文件描述符成员赋值
    pfds[0].fd = 0;           // 表示检测0号
    pfds[0].events = POLLIN; // 表示检测的是读事件

    pfds[1].fd = cfd;         // 检测cfd文件描述符
    pfds[1].events = POLLIN; // 检测读事件

    // 4、数据收发
    char wbuf[128] = "";
    while (1)
    {
        int res = poll(pfds, 2, -1);
        // 功能：阻塞等待文件描述符集合中是否有事件产生
        // 参数1：文件描述符集合起始地址
        // 参数2：文件描述符个数
        // 参数3：表示永久等待
        if (res == -1)
        {
            perror("poll error");
            return -1;
        }

        // 程序执行至此，表示文件描述符容器中，有事件产生
        // 表示0号文件描述符的事件
        if (pfds[0].revents == POLLIN)
        {
            // 清空容器
            bzero(wbuf, sizeof(wbuf));

            // 从终端获取数据
            fgets(wbuf, sizeof(wbuf), stdin); // 0
            wbuf[strlen(wbuf) - 1] = 0;       // 将换行改成 '\0'

            // 将数据发送给服务器
            if (send(cfd, wbuf, sizeof(wbuf), 0) == -1)
            {
                perror("send error");
                return -1;
            }
        }

        //表示有客户端发来消息
        if (pfds[1].revents == POLLIN)
        {
            // 接受服务器发送过来的消息
            if (recv(cfd, wbuf, sizeof(wbuf), 0) == 0) // cfd
            {
                printf("对端已经下线\n");
                break;
            }
```

```
            printf("收到服务器消息为：%s\n", wbuf);
        }
    }

    // 5、关闭套接字
    close(cfd);

    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

11> 总结select和poll的区别

```
1、select管理三个文件描述符集合，poll只管理一个，但是，可以操作很多事件
2、select管理的文件描述符有上限，一般是1024个，而poll管理文件描述符没有这个限制
3、对于效率而言，poll的效率比select的略高
```

3.5 IO多路复用之epoll

1> epoll全称为eventpoll，是内核实现IO多路复用的一种实现方式。在其中一个或多个事件的到满足时，可以解除阻塞。

epoll是select和poll的升级版，相比于select和poll而言，epoll改进了工作方式，效率更高。

```
1、sellect和poll都是基于线性结构进行检测集合，而epoll是基于树形结构（红黑树）完成管理检测集合的
2、select和poll检测时，随着集合的增大，效率会越来越低。epoll使用的是函数回调机制，效率较高。处理文件描述符的效率也不会随着文件秒数的增大而降低。
3、select和poll在工作过程中，不断的在内核空间与用户空间频繁拷贝文件描述符的数据。epoll在注册新的文件描述符或者修改文件描述符时，只需进行一次，能够有效减少数据在用户空间和内核空间之间的切换
4、和poll一样，epoll没有最大文件描述符的限制，仅仅收到程序能够打开的最大文件描述符数量限制
5、对于select和poll而言，需要对返回的文件描述符集合进行判断后才知道时哪些文件描述符就绪了，而epoll可以直接得到已经就绪的文件描述符，无需再次检测
6、当多路复用比较频发进行、IO流量频繁的时候，一般不使用select和poll，使用epoll比较合适
7、epoll只适用于linux平台，不能跨平台操作
```

2> epoll相关API函数：epoll提供了三个函数，分别处理不同的操作

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

        int epoll_create(int size);
        功能：创建一个epoll实例，并返回该实例的句柄，是一个文件描述符
        参数1：epoll实例中能够容纳的最大节点个数，自从linux 2.6.8版本后，size可以忽略，但是必须要是一个大于0的数字
        返回值：成功返回控制epoll实例的文件描述符，失败返回-1并置位错误码


            #include <sys/epoll.h>

        int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
        功能：完成对epoll实例的控制
        参数1：通过epoll_create创建的epoll实例文件描述符
        参数2：op表示要进行的操作
```

EPOLL_CTL_ADD：向epoll树上添加新的要检测的文件描述符
EPOLL_CTL_MOD：改变epoll树上的文件描述符检测的事件
EPOLL_CTL_DEL：删除epoll树上的要检测的文件描述符,此时参数3可以省略填NULL
参数3：要检测的文件描述符
参数4：要检测的事件，是一个结构体变量地址，属于输入变量

```
typedef union epoll_data {
    void        *ptr;         //提供的解释性数据
    int          fd;         //文件描述符（常用）
    uint32_t     u32;
    uint64_t     u64;
} epoll_data_t;

struct epoll_event {
    uint32_t     events;      /* 要检测的事件 */
    epoll_data_t data;        /* 用户有效数据，是一个共用体 */
};
```

要检测的事件：
EPOLLIN：读事件
EPOLLOUT：写事件
EPOLLERR：异常事件
EPOLLET：表示设置epoll的模式为边沿触发模式

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

int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
```

功能：阻塞检测epoll实例中是否有文件描述符准备就绪，如果准备就绪了，就解除阻塞
参数1：epoll实例对于的文件描述符
参数2：文件描述符集合，当有文件描述符产生事件时，将所有产生事件的文件描述符，放入到该集合中
参数3：参数2的大小
参数4：超时时间，以毫秒为单位的超时时间，如果填-1表示永久阻塞
返回值：>0:表示解除本次操作的触发的文件描述符个数
=0：表示超时，但是没有文件描述符产生事件
=-1：失败，置位错误码

3> epoll实现TCP并发服务器的模型

```
1、sfd = socket();          //创建用于连接的文件描述符
2、bind（）；                //绑定ip地址和端口号
3、listen（）；              //将套接字设置成被动监听状态
4、创建一个epoll实例，并将用于连接的文件描述符放入到epoll树中
    struct epoll_event ev;
    ev.events = EPOLLIN;       //检测读事件
    ev.data.fd = sfd;          //要检测的文件描述符
    int ret = epoll_ctl(epfd, EPOLL_CTL_ADD, sfd, &ev);

5、检测文件描述符是否就绪（有客户端发来连接请求）
    int num = epoll_wait(epfd, evs, size, -1);

6、如果检测到文件描述符就绪，则建立连接，将newfd放入到集合中
    newfd = accept();         //接收客户端连接请求
    struct epoll_event ev;
    ev.event = EPOLLIN;
    ev.data.fd = newfd;
```

```
    int ret = epoll_ctl(epfd, EPOLL_CTL_ADD, newfd, &ev);

7、如果检测的是客户端文件描述符，则进行通信，如果客户端下线，则将文件描述符进行删除
    int len = recv();
    if(len == 0)
    {
        epoll_ctl(epfd, EPOLL_CTL_DEL, newfd, NULL);    //将newfd从集合中删除
        close(newfd);
    }else if(len > 0)
    {
        send();
    }
```

4> epoll实现一个TCP数据处理服务器

```
#include <myhead.h>
#define SER_PORT 8888        // 服务器端口号
#define SER_IP "172.20.10.8" // 服务器IP地址

int main(int argc, const char *argv[])
{
    // 1、创建用于连接的套接字文件描述符
    int sfd = socket(AF_INET, SOCK_STREAM, 0);
    // 参数1：AF_INET表示使用的是ipv4的通信协议
    // 参数2：SOCK_STREAM表示使用的是tcp通信
    // 参数3：由于参数2指定了协议，参数3填0即可
    if (sfd == -1)
    {
        perror("socket error");
        return -1;
    }
    printf("socket success sfd = %d\n", sfd); // 3

    // 2、绑定ip地址和端口号
    // 2.1 填充要绑定的ip地址和端口号结构体
    struct sockaddr_in sin;
    sin.sin_family = AF_INET;                // 通信域
    sin.sin_port = htons(SER_PORT);          // 端口号
    sin.sin_addr.s_addr = inet_addr(SER_IP); // ip地址

    // 2.2 绑定工作
    // 参数1：要被绑定的套接字文件描述符
    // 参数2：要绑定的地址信息结构体，需要进行强制类型转换，防止警告
    // 参数3：参数2的大小
    if (bind(sfd, (struct sockaddr *)&sin, sizeof(sin)) == -1)
    {
        perror("bind error");
        return -1;
    }
    printf("bind success\n");

    // 3、启动监听
    // 参数1：要启动监听的文件描述符
    // 参数2：挂起队列的长度
    if (listen(sfd, 128) == -1)
    {
        perror("listen error");
```

```
        return -1;
    }
    printf("listen success\n");

    // 4、阻塞等待客户端的连接请求
    // 定义变量，用于接受客户端地址信息结构体
    struct sockaddr_in cin;          // 用于接收地址信息结构体的
    socklen_t socklen = sizeof(cin); // 用于接收地址信息的长度

    // 创建epoll实例，用于检测文件描述符
    int epfd = epoll_create(1);
    if (epfd == -1)
    {
        perror("epoll_create error");
        return -1;
    }

    // 将sfd放入到检测集合中
    struct epoll_event ev;
    ev.events = EPOLLIN; // 要检测的是读事件
    ev.data.fd = sfd;    // 要检测的文件描述符信息
    epoll_ctl(epfd, EPOLL_CTL_ADD, sfd, &ev);
    // 功能：将sfd放入到检测集合中
    // 参数1：epoll实例的文件描述符
    // 参数2：epoll操作，表示要添加文件描述符
    // 参数3：要检测的文件描述符的值
    // 参数4：要检测的事件

    // 定义接收返回的事件集合
    struct epoll_event evs[1024];
    int size = sizeof(evs) / sizeof(evs[0]); // 数组的大小

    while (1)
    {
        // 阻塞检测文件描述符集合中是否有事件产生
        int num = epoll_wait(epfd, evs, size, -1);
        // 参数1：epoll实例的文件描述符
        // 参数2：返回触发事件的文件事件集合
        // 参数3：集合的大小
        // 参数4：是否阻塞
        printf("num = %d\n", num); // 输出本次触发的文件描述符个数

        // 循环遍历集合，判断是哪个文件描述符就绪
        for (int i = 0; i < num; i++)
        {
            int fd = evs[i].data.fd; // 获取本次解除阻塞的文件描述符

            // 判断是否为sfd文件描述符就绪
            if (fd == sfd)
            {
                // 说明有新的客户端发来连接请求
                int newfd = accept(sfd, (struct sockaddr *)&cin, &socklen);
                // 参数1：服务器套接字文件描述符
                // 参数2：用于接收客户端地址信息结构体的容器，如果不接收，也可以填NULL
                // 参数3：接收参数2的大小，如果参数2为NULL，则参数3也是NULL
                if (newfd == -1)
                {
                    perror("accept error");
```

```
                    return -1;
                }
                printf("[%s:%d]:已连接成功，newfd = %d!!!!\n",
inet_ntoa(cin.sin_addr), ntohs(cin.sin_port), newfd);

                // 将客户端文件描述符放入到epoll检测集合中
                struct epoll_event ev;
                ev.events = EPOLLIN; // 要检测的是读事件
                ev.data.fd = newfd;  // 要检测的文件描述符信息
                epoll_ctl(epfd, EPOLL_CTL_ADD, newfd, &ev);
            }
            else
            {
                // 表示客户端文件描述符就绪，也就是说客户端有数据发来
                // 5、数据收发
                char rbuf[128] = ""; // 数据容器

                // 清空容器中的内容
                bzero(rbuf, sizeof(rbuf));

                // 从套接字中读取消息
                int res = recv(fd, rbuf, sizeof(rbuf), 0);
                if (res == 0)
                {
                    printf("对端已经下线\n");
                    //将客户端从epoll树中删除
                    epoll_ctl(epfd, EPOLL_CTL_DEL, fd, NULL);

                    //关闭套接字
                    close(fd);

                    break;
                }
                printf("收到数据:%s\n",  rbuf);

                // 对收到的数据处理一下，回给客户端
                strcat(rbuf, "*_*");

                // 将消息发送给客户端
                if (send(fd, rbuf, strlen(rbuf), 0) == -1)
                {
                    perror("send error");
                    return -1;
                }
                printf("发送成功\n");

            }
        }
    }
    close(sfd);          //关闭监听
    close(epfd);    //关闭epoll实例

    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

5> epoll的工作模式:epoll的工作模式有两种，分别是水平触发和边沿触发

1、水平模式：
简称LT模式（level trigered）,是默认的一种初始模式，并且该模式支持阻塞和非阻塞的形式
在这种模式下，当文件描述符准备就绪后，内核会通知使用者哪些文件描述符就绪了。
使用者可以对就绪的文件描述符进行操作。
如果使用者不做任何操作或者没有全部处理完该文件描述符的信息，内核会继续通知使用者处理数据

特点：
对于读事件：如果该文件描述符中的缓冲区中的数据没有被读取完毕，则内核会继续解除阻塞，让用户继续处理，直到缓冲区中没有数据可以处理为止。
后面的解除阻塞，是自动完成的，无需用户进行对文件描述符的后续操作。
对于写事件：检测文件描述符缓冲区是否可以，如果可用则解除阻塞，一般写文件描述符的缓冲区都是可以的，一般不对写文件描述符进行检测

2、边沿模式
简称ET模式（edge trigered）,需要手动设置该模式，并且该模式一般支持非阻塞形式
在种模式下，当文件描述符准备就绪后，内核会通知使用者哪些文件描述符就绪了，但是仅仅只通知一次
使用者可以对就绪的文件描述符进行操作
如果使用者不做任何操作，或者没有全部处理该文件描述符的信息，内核不会通知使用者再次处理数据，知道下一次该文件描述符的事件产生

特点：对于读事件：如果文件描述符中的缓冲区数据没有读取完毕，则内核也不会再次解除阻塞，直到下一次的该文件描述符事件产生，但是下一次的文件描述符的事件，读取的是上一次没有读取完毕的内容
对于写事件：检测文件描述符缓冲区是否可以，如果可用则解除阻塞，一般写文件描述符的缓冲区都是可以的，一般不对写文件描述符进行检测
边沿触发模式的处理效率会更高一些，要求用户必须一次性处理文件描述符中的数据

3、如何设置边沿触发：在将文件描述符放入到epoll树中时，需要加一个属性
```
struct epoll_event ev;
ev.event = EPOLLIN|EPOLLEV;
```

验证水平模式

```
#include <myhead.h>
#define SER_PORT 8888        // 服务器端口号
#define SER_IP "172.20.10.8" // 服务器IP地址

int main(int argc, const char *argv[])
{
    // 1、创建用于连接的套接字文件描述符
    int sfd = socket(AF_INET, SOCK_STREAM, 0);
    // 参数1：AF_INET表示使用的是ipv4的通信协议
    // 参数2：SOCK_STREAM表示使用的是tcp通信
    // 参数3：由于参数2指定了协议，参数3填0即可
    if (sfd == -1)
    {
        perror("socket error");
        return -1;
    }
    printf("socket success sfd = %d\n", sfd); // 3

    // 2、绑定ip地址和端口号
    // 2.1 填充要绑定的ip地址和端口号结构体
```

```c
    struct sockaddr_in sin;
    sin.sin_family = AF_INET;                   // 通信域
    sin.sin_port = htons(SER_PORT);             // 端口号
    sin.sin_addr.s_addr = inet_addr(SER_IP); // ip地址

    // 2.2 绑定工作
    // 参数1：要被绑定的套接字文件描述符
    // 参数2：要绑定的地址信息结构体，需要进行强制类型转换，防止警告
    // 参数3：参数2的大小
    if (bind(sfd, (struct sockaddr *)&sin, sizeof(sin)) == -1)
    {
        perror("bind error");
        return -1;
    }
    printf("bind success\n");

    // 3、启动监听
    // 参数1：要启动监听的文件描述符
    // 参数2：挂起队列的长度
    if (listen(sfd, 128) == -1)
    {
        perror("listen error");
        return -1;
    }
    printf("listen success\n");

    // 4、阻塞等待客户端的连接请求
    // 定义变量，用于接受客户端地址信息结构体
    struct sockaddr_in cin;           // 用于接收地址信息结构体的
    socklen_t socklen = sizeof(cin); // 用于接收地址信息的长度

    // 创建epoll实例，用于检测文件描述符
    int epfd = epoll_create(1);
    if (epfd == -1)
    {
        perror("epoll_create error");
        return -1;
    }

    // 将sfd放入到检测集合中
    struct epoll_event ev;
    ev.events = EPOLLIN; // 要检测的是读事件
    ev.data.fd = sfd;    // 要检测的文件描述符信息
    epoll_ctl(epfd, EPOLL_CTL_ADD, sfd, &ev);
    // 功能：将sfd放入到检测集合中
    // 参数1：epoll实例的文件描述符
    // 参数2：epoll操作，表示要添加文件描述符
    // 参数3：要检测的文件描述符的值
    // 参数4：要检测的事件

    // 定义接收返回的事件集合
    struct epoll_event evs[1024];
    int size = sizeof(evs) / sizeof(evs[0]); // 数组的大小

    while (1)
    {
        // 阻塞检测文件描述符集合中是否有事件产生
        int num = epoll_wait(epfd, evs, size, -1);
```

```
        // 参数1：epoll实例的文件描述符
        // 参数2：返回触发事件的文件事件集合
        // 参数3：集合的大小
        // 参数4：是否阻塞
        printf("num = %d\n", num); // 输出本次触发的文件描述符个数

        // 循环遍历集合，判断是哪个文件描述符就绪
        for (int i = 0; i < num; i++)
        {
            int fd = evs[i].data.fd; // 获取本次解除阻塞的文件描述符

            // 判断是否为sfd文件描述符就绪
            if (fd == sfd)
            {
                // 说明有新的客户端发来连接请求
                int newfd = accept(sfd, (struct sockaddr *)&cin, &socklen);
                // 参数1：服务器套接字文件描述符
                // 参数2：用于接收客户端地址信息结构体的容器，如果不接收，也可以填NULL
                // 参数3：接收参数2的大小，如果参数2为NULL，则参数3也是NULL
                if (newfd == -1)
                {
                    perror("accept error");
                    return -1;
                }
                printf("[%s:%d]:已连接成功，newfd = %d!!!!\n",
inet_ntoa(cin.sin_addr), ntohs(cin.sin_port), newfd);

                // 将客户端文件描述符放入到epoll检测集合中
                struct epoll_event ev;
                ev.events = EPOLLIN; // 要检测的是读事件
                ev.data.fd = newfd;  // 要检测的文件描述符信息
                epoll_ctl(epfd, EPOLL_CTL_ADD, newfd, &ev);
            }
            else
            {
                // 表示客户端文件描述符就绪，也就是说客户端有数据发来
                // 5、数据收发
                char rbuf[5] = ""; // 数据容器

                // 清空容器中的内容
                bzero(rbuf, sizeof(rbuf));

                // 从套接字中读取消息
                int res = recv(fd, rbuf, sizeof(rbuf)-1, 0);
                if (res == 0)
                {
                    printf("对端已经下线\n");
                    //将客户端从epoll树中删除
                    epoll_ctl(epfd, EPOLL_CTL_DEL, fd, NULL);

                    //关闭套接字
                    close(fd);

                    break;
                }
                printf("收到数据:%s\n",  rbuf);

                // 对收到的数据处理一下，回给客户端
```

```
                strcat(rbuf, "*_*");

                // 将消息发送给客户端
                if (send(fd, rbuf, strlen(rbuf), 0) == -1)
                {
                    perror("send error");
                    return -1;
                }
                printf("发送成功\n");

            }
        }
    }
    close(sfd);          //关闭监听
    close(epfd);     //关闭epoll实例

    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

```
socket success sfd = 3
bind success
listen success
num = 1
[172.20.10.7:31867]:已连接成功, newfd = 5!!!!
num = 1
收到数据: hel
发送成功
num = 1
收到数据:lowo
发送成功
num = 1
收到数据:rldn
发送成功
num = 1
收到数据:ihao
发送成功
num = 1
收到数据:xing
```

如果缓冲区中数据没有处理结束
会再次通知使用者处理数据

边沿触发的测试

```
#include <myhead.h>
#define SER_PORT 8888          // 服务器端口号
#define SER_IP "172.20.10.8" // 服务器IP地址

int main(int argc, const char *argv[])
{
    // 1、创建用于连接的套接字文件描述符
    int sfd = socket(AF_INET, SOCK_STREAM, 0);
    // 参数1：AF_INET表示使用的是ipv4的通信协议
    // 参数2：SOCK_STREAM表示使用的是tcp通信
    // 参数3：由于参数2指定了协议，参数3填0即可
    if (sfd == -1)
    {
```

```
        perror("socket error");
        return -1;
    }
    printf("socket success sfd = %d\n", sfd); // 3

    // 2、绑定ip地址和端口号
    // 2.1 填充要绑定的ip地址和端口号结构体
    struct sockaddr_in sin;
    sin.sin_family = AF_INET;                   // 通信域
    sin.sin_port = htons(SER_PORT);             // 端口号
    sin.sin_addr.s_addr = inet_addr(SER_IP); // ip地址

    // 2.2 绑定工作
    // 参数1：要被绑定的套接字文件描述符
    // 参数2：要绑定的地址信息结构体，需要进行强制类型转换，防止警告
    // 参数3：参数2的大小
    if (bind(sfd, (struct sockaddr *)&sin, sizeof(sin)) == -1)
    {
        perror("bind error");
        return -1;
    }
    printf("bind success\n");

    // 3、启动监听
    // 参数1：要启动监听的文件描述符
    // 参数2：挂起队列的长度
    if (listen(sfd, 128) == -1)
    {
        perror("listen error");
        return -1;
    }
    printf("listen success\n");

    // 4、阻塞等待客户端的连接请求
    // 定义变量，用于接受客户端地址信息结构体
    struct sockaddr_in cin;           // 用于接收地址信息结构体的
    socklen_t socklen = sizeof(cin); // 用于接收地址信息的长度

    // 创建epoll实例，用于检测文件描述符
    int epfd = epoll_create(1);
    if (epfd == -1)
    {
        perror("epoll_create error");
        return -1;
    }

    // 将sfd放入到检测集合中
    struct epoll_event ev;
    ev.events = EPOLLIN | EPOLLET; // 要检测的是读事件    EPOLLET表示该文件描述符检测是使用边沿触发模式
    ev.data.fd = sfd;     // 要检测的文件描述符信息
    epoll_ctl(epfd, EPOLL_CTL_ADD, sfd, &ev);
    // 功能：将sfd放入到检测集合中
    // 参数1：epoll实例的文件描述符
    // 参数2：epoll操作，表示要添加文件描述符
    // 参数3：要检测的文件描述符的值
    // 参数4：要检测的事件
```

```
    // 定义接收返回的事件集合
    struct epoll_event evs[1024];
    int size = sizeof(evs) / sizeof(evs[0]); // 数组的大小

    while (1)
    {
        // 阻塞检测文件描述符集合中是否有事件产生
        int num = epoll_wait(epfd, evs, size, -1);
        // 参数1：epoll实例的文件描述符
        // 参数2：返回触发事件的文件事件集合
        // 参数3：集合的大小
        // 参数4：是否阻塞
        printf("num = %d\n", num); // 输出本次触发的文件描述符个数

        // 循环遍历集合，判断是哪个文件描述符就绪
        for (int i = 0; i < num; i++)
        {
            int fd = evs[i].data.fd; // 获取本次解除阻塞的文件描述符

            // 判断是否为sfd文件描述符就绪
            if (fd == sfd)
            {
                // 说明有新的客户端发来连接请求
                int newfd = accept(sfd, (struct sockaddr *)&cin, &socklen);
                // 参数1：服务器套接字文件描述符
                // 参数2：用于接收客户端地址信息结构体的容器，如果不接收，也可以填NULL
                // 参数3：接收参数2的大小，如果参数2为NULL，则参数3也是NULL
                if (newfd == -1)
                {
                    perror("accept error");
                    return -1;
                }
                printf("[%s:%d]:已连接成功，newfd = %d!!!!\n",
inet_ntoa(cin.sin_addr), ntohs(cin.sin_port), newfd);

                // 将客户端文件描述符放入到epoll检测集合中
                struct epoll_event ev;
                ev.events = EPOLLIN | EPOLLET; // 要检测的是读事件
                ev.data.fd = newfd;  // 要检测的文件描述符信息
                epoll_ctl(epfd, EPOLL_CTL_ADD, newfd, &ev);
            }
            else
            {
                // 表示客户端文件描述符就绪，也就是说客户端有数据发来
                // 5、数据收发
                char rbuf[5] = ""; // 数据容器

                // 清空容器中的内容
                bzero(rbuf, sizeof(rbuf));

                // 从套接字中读取消息
                int res = recv(fd, rbuf, sizeof(rbuf)-1, 0);
                if (res == 0)
                {
                    printf("对端已经下线\n");
                    //将客户端从epoll树中删除
                    epoll_ctl(epfd, EPOLL_CTL_DEL, fd, NULL);
```

```
                    //关闭套接字
                    close(fd);

                    break;
                }
                printf("收到数据:%s\n",  rbuf);

                // 对收到的数据处理一下，回给客户端
                strcat(rbuf, "*_*");

                // 将消息发送给客户端
                if (send(fd, rbuf, strlen(rbuf), 0) == -1)
                {
                    perror("send error");
                    return -1;
                }
                printf("发送成功\n");

            }
        }
    }
    close(sfd);          //关闭监听
    close(epfd);    //关闭epoll实例

    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

```
问题    输出    调试控制台    终端    端口

listen success
num = 1
[172.20.10.7:31892]:已连接成功，newfd = 5!!!!
num = 1
收到数据:hell
发送成功
```

只通知使用者，处理数据一次
要再次处理数据的话，需要下一次的事件产生

四、组播和广播

4.1 网络属性

1> 网络套接字在不同层中，有不同的设置，分别有应用层（套接字层）、传输层、网络层、以太网层

2> 对套接字属性进行设置，可以使用setsockopt和getsockopt函数完成

```
        #include <sys/types.h>          /* See NOTES */
        #include <sys/socket.h>

        int getsockopt(int sockfd, int level, int optname,void *optval, socklen_t
*optlen);
        int setsockopt(int sockfd, int level, int optname,const void *optval,
socklen_t optlen);
        功能：设置或者获取套接字文件描述符的属性
        参数1：套接字文件描述符
        参数2：表示操作的套接字层
              SOL_SOCKET:表示应用层或者套接字层，通过man 7 socket进行查找
                      该层中常用的属性：SO_REUSEADDR   地址快速重用
                                        SO_BROADCAST   允许广播
```

```
                                        SO_RCVTIMEO and SO_SNDTIMEO:发送或接收超时时间
            IPPROTO_TCP:表示传输层的基于TCP的协议
            IPPROTO_UDP:表示传输层中基于UDP的协议
            IPPROTO_IP:表示网络层
                    该层常用的属性：IP_ADD_MEMBERSHIP，加入多播组
        参数3:对应层中属性的名称
        参数4：参数3属性的值，一般为int类型，其他类型，会给出
        参数5：参数4的大小
        返回值：成功返回0，失败返回-1并置位错误码
```

3> 验证网络属性

```c
// 1、创建用于连接的套接字文件描述符
  int sfd = socket(AF_INET, SOCK_STREAM, 0);
  // 参数1：AF_INET表示使用的是ipv4的通信协议
  // 参数2：SOCK_STREAM表示使用的是tcp通信
  // 参数3：由于参数2指定了协议，参数3填0即可
  if (sfd == -1)
  {
      perror("socket error");
      return -1;
  }
  printf("socket success sfd = %d\n", sfd); // 3


  //测试套接字地址默认是否能快速重用
  char reuse = -1;       //接收获取下来的数据
  socklen_t reuselen ;        //接收大小
  if(getsockopt(sfd, SOL_SOCKET, SO_REUSEADDR, &reuse, &reuselen)==-1)
  {
      perror("getscokopt error");
      return -1;
  }
  printf("reuse = %d\n", reuse);      //0表示默认不可用    1表示默认可用

  //设置端口号快速重用
  int reu = 1;
  if(setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR, &reu, sizeof(reu)) == -1)
  {
      perror("setsockopt error");
      return -1;
  }


  //验证
  if(getsockopt(sfd, SOL_SOCKET, SO_REUSEADDR, &reuse, &reuselen)==-1)
  {
      perror("getscokopt error");
      return -1;
  }
  printf("reuse = %d\n", reuse);      //    1表示默认可用
```

4.2 广播

1> 主机之间是一对多的通信方式，网络对其中的每一台主机都会将消息进行转发

2> 在当前网络下的所有主机都会收到该消息，无论是否愿意收到消息

3> 基于无连接的通信方式完成，UDP通信

4> 广播分为发送端和接收端

5> 对于发送端而言，需要设置允许广播，并且向广播地址发送数据

6> 广播地址：网络号 + 全为1的主机号，可以通过指令 ifconfig查看

7> 广播消息不能穿过路由器到骨干路由上

4.2.1 广播发送到流程 ---> 类似于UDP的客户端

```
1、socket：创建用于发送数据的套接字文件描述符
2、setsockopt：设置套接字属性，允许广播  SOL_SOCKET层  中 SO_BROADCAST
3、bind：可以绑定也可以不绑定
4、数据收发，对端套接字地址信息结构体
    IP：广播地址
    PORT：与接收端保持一致
5、sendto/recvfrom：数据收发
6、close：关闭套接字
```

```c
#include <myhead.h>

int main(int argc, const char *argv[])
{
    //1、创建用于通信的客户端套接字文件描述符
    int sndfd = socket(AF_INET, SOCK_DGRAM, 0);
    if(sndfd == -1)
    {
        perror("socket error");
        return -1;
    }
    printf("socket success cfd = %d\n", sndfd);       //3

    //设置允许广播
    int broad = 1;      //要设置的值
    if(setsockopt(sndfd, SOL_SOCKET, SO_BROADCAST, &broad, sizeof(broad)) ==-1)
    {
        perror("setsockopt error");
        return -1;
    }
    printf("成功设置广播\n");


    //2、可以绑定也可以不绑定

    //3、数据收发
    char wbuf[128] = "";
    //填充接收端的地址信息结构体
    struct sockaddr_in rin;
    rin.sin_family = AF_INET;       //通信域
    rin.sin_port = htons(8888);    //端口号
    rin.sin_addr.s_addr = inet_addr("192.168.174.255");    //广播ip地址
```

```
    while(1)
    {
        //清空容器
        bzero(wbuf, sizeof(wbuf));

        //从终端读取数据
        fgets(wbuf, sizeof(wbuf), stdin);
        wbuf[strlen(wbuf)-1] = 0;

        //将数据发送给服务器
        sendto(sndfd, wbuf, strlen(wbuf), 0, (struct sockaddr*)&rin,
sizeof(rin));

        printf("发送成功\n");

    }

    //4、关闭套接字
    close(sndfd);


    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

4.2.2 广播的接收端流程 ---> 类似于UDP的服务器端

```
1、socket：创建用于接收数据的套接字文件描述符
2、bind：必须绑定
    ip:广播地址
    PORT：与发送端一致

4、sendto/recvfrom：数据收发
5、close：关闭套接字
```

```
#include <myhead.h>


int main(int argc, const char *argv[])
{
    //1、创建用于通信的套接字文件描述符
    int recvfd = socket(AF_INET, SOCK_DGRAM, 0);
    //SOCK_DGRAM表示基于udp通信方式
    if(recvfd == -1)
    {
        perror("socket error");
        return -1;
    }
    printf("socket success sfd = %d\n", recvfd);        //3


    //2、绑定ip地址和端口号
        //2.1 填充要绑定的ip地址和端口号结构体
```

```
    struct sockaddr_in rin;
    rin.sin_family = AF_INET;        //通信域
    rin.sin_port = htons(8888);    //端口号
    rin.sin_addr.s_addr = inet_addr("192.168.174.255");     //广播ip地址

    //2.2 绑定工作
    //参数1：要被绑定的套接字文件描述符
    //参数2：要绑定的地址信息结构体，需要进行强制类型转换，防止警告
    //参数3：参数2的大小
    if(bind(recvfd, (struct sockaddr*)&rin, sizeof(rin)) ==-1)
    {
        perror("bind error");
        return -1;
    }
    printf("bind success\n");

    //3、数据收发
    char rbuf[128] = "";
    //定义容器接收对端的地址信息结构体
    struct sockaddr_in cin;
    socklen_t socklen = sizeof(cin);

    while(1)
    {
        //清空容器
        bzero(rbuf, sizeof(rbuf));

        //从客户端中读取消息
        if(recvfrom(recvfd, rbuf, sizeof(rbuf), 0, (struct sockaddr*)&cin,
&socklen)==-1)
        {
            perror("recvfrom error");
            return -1;
        }

        printf("[%s:%d]:%s\n", inet_ntoa(cin.sin_addr), ntohs(cin.sin_port),
rbuf);

    }

    //4、关闭套接字
    close(recvfd);

    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

4.3 组播

1> 广播是给同一个网络下的所有主机发送消息，会占用大量的网络带宽，影响正常网络通信，造成网络拥塞

2> 组播也是实现一对多的通信机制，也就是说，加入同一个多播组的成员都能收到组播消息

3> 组播也是使用UDP实现的，发送者发送的消息，无论接收者愿不愿意，都会收到消息

4>　组播地址：D类网络地址 （224.0.0.0 --- 239.255.255.255）

4.3.1 组播的发送端流程 ---> 类似于UDP客户端

```
1、socket:创建用于消息通信的套接字文件描述符
2、bind：非必须
3、填充要发送的地址信息结构体
    IP：D类网络
    PORT：与接收端保持一致
4、sendto：发送数据
5、close:关闭套接字
```

```
#include <myhead.h>


int main(int argc, const char *argv[])
{
    //1、创建用于通信的客户端套接字文件描述符
    int sendfd = socket(AF_INET, SOCK_DGRAM, 0);
    if(sendfd == -1)
    {
        perror("socket error");
        return -1;
    }
    printf("socket success cfd = %d\n", sendfd);        //3

    //2、绑定ip地址和端口号（可选）


    //3、数据收发
    char wbuf[128] = "";
    //填充接收端的地址信息结构体
    struct sockaddr_in rin;
    rin.sin_family = AF_INET;       //通信域
    rin.sin_port = htons(8888);    //端口号
    rin.sin_addr.s_addr = inet_addr("224.1.2.3");    //组播ip地址

    while(1)
    {
        //清空容器
        bzero(wbuf, sizeof(wbuf));

        //从终端读取数据
        fgets(wbuf, sizeof(wbuf), stdin);
        wbuf[strlen(wbuf)-1] = 0;

        //将数据发送给服务器r
        sendto(sendfd, wbuf, strlen(wbuf), 0, (struct sockaddr*)&rin,
sizeof(rin));

    }

    //4、关闭套接字
    close(sendfd);


    std::cout << "Hello, World!" << std::endl;
```

```
    return 0;
}
```

4.3.2 组播的接收端流程 ---> 类似于UDP服务器端

```
1、socket：创建用于通信的套接字文件描述符
2、setsocketopt：设置套接字加入多播组    在IPPROTO_IP网络层   IP_ADD_MEMBERSHIP属性
        属性值为结构体类型
        struct ip_mreqn {
                        struct in_addr imr_multiaddr; /* 组播地址的网络字节序 */
                        struct in_addr imr_address;   /* 接收端的主机地址 */
                        int            imr_ifindex;   /* 网卡设备编号   一般为2   可以通
过指令 ip ad 查看 */
                    };
3、bind：绑定地址信息结构体
    IP：组播地址
    PORT：与发送端保持一致
4、recv、recvfrom、read：接收数据
5、close:关闭套接字
```

```c
#include <myhead.h>


int main(int argc, const char *argv[])
{
    //1、创建用于通信的套接字文件描述符
    int recvfd = socket(AF_INET, SOCK_DGRAM, 0);
    //SOCK_DGRAM表示基于udp通信方式
    if(recvfd == -1)
    {
        perror("socket error");
        return -1;
    }
    printf("socket success sfd = %d\n", recvfd);       //3

    //将该套接字加入多播组
    struct ip_mreqn imr;     //多播组属性结构体
    imr.imr_multiaddr.s_addr = inet_addr("224.1.2.3");    //组播地址
    imr.imr_address.s_addr = inet_addr("192.168.174.128");   //本机ip地址
    imr.imr_ifindex = 2;          //网卡编号
    //调用设置网络属性函数
    if(setsockopt(recvfd, IPPROTO_IP, IP_ADD_MEMBERSHIP, &imr, sizeof(imr))
==-1)
    {
        perror("setsockopt error");
        return -1;
    }
    printf("成功加入多播组\n");



    //2、绑定ip地址和端口号
        //2.1 填充要绑定的ip地址和端口号结构体
    struct sockaddr_in rin;
```

```
    rin.sin_family = AF_INET;        //通信域
    rin.sin_port = htons(8888);   //端口号
    rin.sin_addr.s_addr = inet_addr("224.1.2.3");    //组播ip地址

    //2.2 绑定工作
    //参数1：要被绑定的套接字文件描述符
    //参数2：要绑定的地址信息结构体，需要进行强制类型转换，防止警告
    //参数3：参数2的大小
    if(bind(recvfd, (struct sockaddr*)&rin, sizeof(rin)) ==-1)
    {
        perror("bind error");
        return -1;
    }
    printf("bind success\n");

    //3、数据收发
    char rbuf[128] = "";

    while(1)
    {
        //清空容器
        bzero(rbuf, sizeof(rbuf));

        //从客户端中读取消息
        if(recvfrom(recvfd, rbuf, sizeof(rbuf), 0, NULL, NULL)==-1)
        {
            perror("recvfrom error");
            return -1;
        }

        printf("[读取的消息为]:%s\n",  rbuf);

    }

    //4、关闭套接字
    close(recvfd);

    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

五、域套接字

5.1 域套接字相关内容

1>　域套接字就是原始的套接字通信方式，跟IPC通信方式一样，属于同一主机之间的多个进程间通信方式

2>　由于不需要进行跨主机通信了，也就不需要使用ip地址和端口号了

3>　域套接字通信载体为套接字文件

　　linux中文件类型：bcd-l**s**p　其中 s类型的文件就是套接字文件

4>　通信本质：使用内核空间完成数据的传输

5> 域套接字通信也分为流式域套接字和报式域套接字

6> 跟跨主机网络通信区别的相关函数

```
        #include <sys/types.h>          /* See NOTES */
        #include <sys/socket.h>

        int socket(int domain, int type, int protocol);
        功能：为通信创建一个端点，并返回该端点对应的文件描述符，文件描述符的使用原则是最小未分配原则
        参数1：协议族，常用的协议族如下
         Name                  Purpose                                 Man page
        AF_UNIX, AF_LOCAL   本地通信，同一主机的多进程通信          具体内容查看 man 7 unix
        AF_INET             提供IPv4的相关通信方式                  具体内容查看 man 7 ip
        AF_INET6            提供IPv6的相关通信方式                  具体内容查看 man 7 ipv6

        参数2：通信类型，指定通信语义，常用的通信类型如下
                SOCK_STREAM      支持TCP面向连接的通信协议

                SOCK_DGRAM       支持UDP面向无连接的通信协议

        参数3：通信协议，当参数2中明确指定特定协议时，参数3可以设置为0，但是有多个协议共同使用时，需要用参数3指定当前套接字确定的协议
        返回值：成功返回创建的端点对应的文件描述符，失败返回-1并置位错误码



        #include <sys/types.h>          /* See NOTES */
        #include <sys/socket.h>

        int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
        功能：为套接字分配名称，给套接字绑定ip地址和端口号
        参数1：要被绑定的套接字文件描述符
        参数2：通用地址信息结构体，对于不同的通信域而言，使用的实际结构体是不同的，该结构体的目的是为了强制类型转换，防止警告
              通信域为：AF_UNIX而言，本地通信
              struct sockaddr_un {
                 sa_family_t sun_family;               /* 通信域：AF_UNIX */
                 char        sun_path[UNIX_PATH_MAX];  /* 通信使用的文件 */     要求套接字文件不存在
             };
          参数3：参数2的大小
          返回值：成功返回0，失败返回-1并置位错误码
```

7> 如何判断一个文件是否存在于文件系统

```
#include <unistd.h>

int access(const char *pathname, int mode);
功能：判断给定的文件是否具有给的的权限
参数1：要被判断的文件描述符
参数2：要被判断的权限
      R_OK:读权限
      W_OK:写权限
      X_OK:执行权限
      F_OK:是否存在
 返回值：如果要被判断的权限都存在，则返回0，否则返回-1并置位错误码
```

8> 如何删除一个文件系统中的文件

```
#include <unistd.h>

int unlink(const char *path);
功能：删除指定的文件
参数：要删除的文件路径
返回值：成功删除返回0，失败返回-1并置位错误码
```

5.2 流式域套接字实现

1> 服务器端实现

```c
#include <myhead.h>

int main(int argc, const char *argv[])
{
    //1、创建用于连接的套接字文件描述符
    int sfd = socket(AF_UNIX, SOCK_STREAM, 0);
    //参数1：AF_UNIX表示使用的是ipv4的通信协议
    //参数2：SOCK_STREAM表示使用的是tcp通信
    //参数3：由于参数2指定了协议，参数3填0即可
    if(sfd == -1)
    {
        perror("socket error");
        return -1;
    }
    printf("socket success sfd = %d\n", sfd);          //3


    //判断套接字文件是否存在
    if(access("./unix", F_OK) == 0)
    {
        //说明文件已经存在，需要将其进行删除操作
        if(unlink("./unix") == -1)
        {
            perror("unlink error");
            return -1;
        }
    }

    //2、绑定套接字文件描述符
```

```c
    //2.1 填充要绑定的ip地址和端口号结构体
    struct sockaddr_un sun;
    sun.sun_family = AF_UNIX;        //通信域
    //sun.sun_path = "./unix";       //?不可以，因为字符串赋值需要使用 strcpy
    strcpy(sun.sun_path , "./unix");


    //2.2 绑定工作
    //参数1：要被绑定的套接字文件描述符
    //参数2：要绑定的地址信息结构体，需要进行强制类型转换，防止警告
    //参数3：参数2的大小
    if(bind(sfd, (struct sockaddr*)&sun, sizeof(sun)) ==-1)
    {
        perror("bind error");
        return -1;
    }
    printf("bind success\n");



    //3、启动监听
    //参数1：要启动监听的文件描述符
    //参数2：挂起队列的长度
    if(listen(sfd, 128) ==-1)
    {
        perror("listen error");
        return -1;
    }
    printf("listen success\n");

    //4、阻塞等待客户端的连接请求
    //定义变量，用于接受客户端地址信息结构体
    struct sockaddr_un cun;              //用于接收地址信息结构体的
    socklen_t socklen = sizeof(cun);         //用于接收地址信息的长度

    int newfd = accept(sfd, (struct sockaddr *)&cun, &socklen);
    //参数1：服务器套接字文件描述符
    //参数2：用于接收客户端地址信息结构体的容器，如果不接收，也可以填NULL
    //参数3：接收参数2的大小，如果参数2为NULL，则参数3也是NULL
    if(newfd == -1)
    {
        perror("accept error");
        return -1;
    }
    printf("[%s]:已连接成功!!!!\n", cun.sun_path);     //输出套接字文件

    //5、数据收发
    char rbuf[128] = "";        //数据容器
    while(1)
    {
        //清空容器中的内容
        bzero(rbuf, sizeof(rbuf));

        //从套接字中读取消息
        int res = recv(newfd, rbuf, sizeof(rbuf), 0);
        if(res == 0)
        {
            printf("对端已经下线\n");
```

```
            break;
        }
        printf("[%s]:%s\n", cun.sun_path, rbuf);

        //对收到的数据处理一下，回给客户端
        strcat(rbuf, "*_*");

        //将消息发送给客户端
        if(send(newfd, rbuf, strlen(rbuf), 0) == -1)
        {
            perror("send error");
            return -1;
        }
        printf("发送成功\n");
    }


    //6、关闭套接字
    close(newfd);
    close(sfd);



    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

2> 客户端实现

```
#include <myhead.h>

int main(int argc, const char *argv[])
{
    //1、创建用于通信的客户端套接字文件描述符
    int cfd = socket(AF_UNIX, SOCK_STREAM, 0);
    if(cfd == -1)
    {
        perror("socket error");
        return -1;
    }
    printf("socket success cfd = %d\n", cfd);        //3

    //2、绑定ip地址和端口号(可选)     如果没有绑定，系统也不会自动绑定，会给的一个随机的文件

     //判断套接字文件是否存在
    if(access("./linux", F_OK) == 0)
    {
        //说明文件已经存在，需要将其进行删除操作
        if(unlink("./linux") == -1)
        {
            perror("unlink error");
            return -1;
        }
    }

    //2.1 填充要绑定的地址信息结构体
```

```c
    struct sockaddr_un cun;
    cun.sun_family = AF_UNIX;
    strcpy(cun.sun_path, "./linux");

    //2.2 绑定工作
    if(bind(cfd, (struct sockaddr*)&cun, sizeof(cun)) == -1)
    {
        perror("bind error");
        return -1;
    }
    printf("bind success\n");


    //3、连接服务器
    //3.1 填充要连接的服务器地址信息结构体
    struct sockaddr_un sun;
    sun.sun_family = AF_UNIX;      //通信域
    strcpy(sun.sun_path, "./unix");    //服务器套接字文件

    //3.2 连接工作
    if(connect(cfd, (struct sockaddr*)&sun, sizeof(sun)) == -1)
    {
        perror("connect error");
        return -1;
    }
    printf("连接服务器成功\n");


    //4、数据收发
    char wbuf[128] = "";
    while(1)
    {
        //清空容器
        bzero(wbuf, sizeof(wbuf));

        //从终端获取数据
        fgets(wbuf, sizeof(wbuf), stdin);       //0
        wbuf[strlen(wbuf)-1] = 0;          //将换行改成 '\0'

        //将数据发送给服务器
        if(send(cfd, wbuf, sizeof(wbuf), 0)==-1)
        {
            perror("send error");
            return -1;
        }




        //接受服务器发送过来的消息
        if(recv(cfd, wbuf, sizeof(wbuf), 0)==0)
        {
            printf("对端已经下线\n");
            break;
        }

        printf("收到服务器消息为：%s\n", wbuf);
```

```
    }

    //5、关闭套接字
    close(cfd);


    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

3> 注意：

```
1、跟网络通信比较，不需要使用ip地址和端口号了，需要使用的是套接字文件
2、与网络通信比较，注意有两个不同：通信域为AF_UNIX，地址信息结构体为 struct scokaddr_un
3、绑定时用到的套接字文件，必须是不存在的
4、对于客户端而言，如果没有绑定地址信息结构体，系统不会自动帮其绑定一个随机文件的，文件名是一个乱码
```

5.3 报式域套接字实现

1> 服务器端实现

```
#include <myhead.h>


int main(int argc, const char *argv[])
{
    //1、创建用于通信的套接字文件描述符
    int sfd = socket(AF_UNIX, SOCK_DGRAM, 0);
    //SOCK_DGRAM表示基于udp通信方式
    if(sfd == -1)
    {
        perror("socket error");
        return -1;
    }
    printf("socket success sfd = %d\n", sfd);       //3


    //2、绑定套接字文件
        //判断套接字文件是否存在
    if(access("./unix", F_OK) == 0)
    {
        //说明文件已经存在，需要将其进行删除操作
        if(unlink("./unix") == -1)
        {
            perror("unlink error");
            return -1;
        }
    }

        //2.1 填充要绑定的ip地址和端口号结构体
    struct sockaddr_un sun;
    sun.sun_family = AF_UNIX;       //通信域
```

```
    strcpy(sun.sun_path, "./unix");   //套接字文件


    //2.2 绑定工作
    //参数1：要被绑定的套接字文件描述符
    //参数2：要绑定的地址信息结构体，需要进行强制类型转换，防止警告
    //参数3：参数2的大小
    if(bind(sfd, (struct sockaddr*)&sun, sizeof(sun)) ==-1)
    {
        perror("bind error");
        return -1;
    }
    printf("bind success\n");

    //3、数据收发
    char rbuf[128] = "";
    //定义容器接收对端的地址信息结构体
    struct sockaddr_un cun;
    socklen_t socklen = sizeof(cun);

    while(1)
    {
        //清空容器
        bzero(rbuf, sizeof(rbuf));

        //从客户端中读取消息
        if(recvfrom(sfd, rbuf, sizeof(rbuf), 0, (struct sockaddr*)&cun,
&socklen)==-1)
        {
            perror("recvfrom error");
            return -1;
        }

        printf("[%s]:%s\n", cun.sun_path, rbuf);

        //加个笑脸发给客户端
        strcat(rbuf, "*_*");

        //将数据发送给客户端
        sendto(sfd, rbuf, strlen(rbuf), 0, (struct sockaddr*)&cun, sizeof(cun));
        printf("发送成功\n");

    }

    //4、关闭套接字
    close(sfd);

    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

2> 客户端端实现

```
#include <myhead.h>
```

```c
int main(int argc, const char *argv[])
{
    //1、创建用于通信的客户端套接字文件描述符
    int cfd = socket(AF_UNIX, SOCK_DGRAM, 0);
    if(cfd == -1)
    {
        perror("socket error");
        return -1;
    }
    printf("socket success cfd = %d\n", cfd);       //3

    //2、绑定ip地址和端口号（可选）
         //判断套接字文件是否存在
    if(access("./linux", F_OK) == 0)
    {
        //说明文件已经存在，需要将其进行删除操作
        if(unlink("./linux") == -1)
        {
            perror("unlink error");
            return -1;
        }
    }

    //2.1 填充要绑定的地址信息结构体
    struct sockaddr_un cun;
    cun.sun_family = AF_UNIX;
    strcpy(cun.sun_path, "./linux");     //绑定套接字文件


    //2.2 绑定工作
    if(bind(cfd, (struct sockaddr*)&cun, sizeof(cun)) == -1)
    {
        perror("bind error");
        return -1;
    }
    printf("bind success\n");



    //3、数据收发
    char wbuf[128] = "";
    //填充服务器的地址信息结构体
    struct sockaddr_un sun;
    sun.sun_family = AF_UNIX;       //通信域
    strcpy(sun.sun_path, "./unix");   //套接字文件

    while(1)
    {
        //清空容器
        bzero(wbuf, sizeof(wbuf));

        //从终端读取数据
        fgets(wbuf, sizeof(wbuf), stdin);
        wbuf[strlen(wbuf)-1] = 0;

        //将数据发送给服务器
        sendto(cfd, wbuf, strlen(wbuf), 0, (struct sockaddr*)&sun, sizeof(sun));
```

```
        //接收服务器发来的数据
        recvfrom(cfd, wbuf, sizeof(wbuf), 0, NULL, NULL);

        printf("服务器发来的消息为：%s\n", wbuf);

    }

    //4、关闭套接字
    close(cfd);


    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

练习

按照给的的网络通信模型，将所有通信方式进行实现一遍