1. 字节序

- 小端存储：主机字节序，内存低位地址存储数据低位字节，内存高位地址存储数据高位字节。
- 大端存储：网络字节序，内存低位地址存储数据高位字节，内存高位地址存储数据低位字节。
- 字节序转换函数

```
//网络通信的时候，IP和端口都要使用大端模式（网络字节序）
//默认使用的IP和端口在本地是小端存储，通信时需要转换
#include <arpa/inet.h>
//主机字节序转网络字节序
uint32_t htonl(uint32_t hostlong);//IP地址
uint16_t htons(uint16_t hostshort);//端口
//网络字节序转主机字节序
uint32_t ntohl(uint32_t netlong);//IP地址
uint16_t ntohs(uint16_t netshort);//端口
```

2. ip地址转换

```
//IP地址 字符串 <-> 字节
#include <arpa/inet.h>
//主机字节序 字符串地址->网络字节序 整形地址
int inet_pton(int af, const char *src, void *dst);
参数:
    - af：地址族协议
        - AF_INET：ipv4
        - AF_INET6：ipv6
返回值:
    成功：0
    失败：-1
//网络字节序 整形地址->主机字节序 字符串地址
const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);
参数:
    - size：参数dst指向内存的大小
返回值:
    失败：NULL
    成功：dst指向的内存
```

3. 套接字

服务器端通信流程

```
/*
    在服务器端有两类文件描述符
        1.监听的
            - 检测有没有新的客户端连接服务器
            - 服务器端有一个就够了
        2.通信的
            - 负责与建立连接的客户端通信
            - 和多少客户端建立连接，就有多少通信文件描述符
*/

//1.创建一个用于监听的套接字，就是一个文件描述符
//类似于管道中的文件描述符，对应的是内核中的内存，通过文件描述符可以读写内核中的内存数据
```

```
int lfd = socket();

//2.让监听的文件描述符和本地的ip:端口进行绑定，绑定后就可以检测有没有客户端连接请求
bind();

//3.给绑定成功的套接字设置监听
listen();

//4.等待并接受客户端的连接，得到一个用于通信的文件描述符
int cfd = accept();

//5.使用通信文件描述符与客户端通信
//接收数据
read();
recv();
//发送数据
write();
send();

//6.断开连接，关闭文件描述符
//关闭通信文件描述符：不能通信了
//关闭监听文件描述符：不能检测客户端连接
close(cfd);
close(lfd);
```

客户端通信流程

```
//1.创建用于通信的文件描述符
int fd = socket();

//2.使用得到的通信文件描述符连接服务器，通过服务器绑定的ip:端口连接
connect();

//3.连接之后，通信
//接收数据
read();
recv();
//发送数据
write();
send();
//4.断开和服务器的连接
close();
```

套接字中的文件描述符

- 文件描述符对应内核中的两块内存
- 一个读缓冲区
- 一个写缓冲区
- 通信文件描述符与监听文件描述符的内存结构相同
- 监听文件描述符
    - 读缓冲区

        - 客户端连接服务器，向服务器发送连接请求，请求数据进入服务器端监听文件描述符的读缓冲区中
        - 只要这个缓冲区中有数据就意味着有新客户端连接
- 通信文件描述符
    - 读缓冲区
        - 保存对端发送过来的数据，通过调用读函数将数据从内核中读出来

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

    - 写缓冲区
        - 调用发送数据的函数，要发送的数据被写入到套接字对应的写缓冲区中

```
ssize_t write(int fd, const void *buf, size_t count);
```

        - 内核检测到写缓冲区中有数据，会将数据发送到网络的另外一端

函数

```
//#include <arpa/inet.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
int listen(int sockfd, int backlog);
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
```

端口复用

TIME_WAIT状态进程会有2MSL的等待时常，此时进程没有退出，绑定的端口不会被释放

```
int setsockopt(int sockfd, int level, int optname,
               const void *optval, socklen_t optlen);
参数:
    - level：SOL_SOCKET
    - optname：SO_RESUEPORT
    - optval:enable a boolean option, or zero if the option is to be disabled
    - optlen:sizeof(optval)
```

4. IO多路复用

进行套接字通信的时候有一些阻塞函数：accept, read/recv, write/send

- 需要不停的检测新的客户端连接；需要不停的调用accept，需要占用一个单独的线程/进程进行检测
- 和客户端连接建立成功了，通信
    - 发送数据：write/read，如果缓冲区写满，阻塞 -> 需要一个单独的线程/进程处理

- 接收数据：read/recv，对方不给当前终端发送数据，当前终端阻塞 -> 需要单独线程处理数据接收

IO多路复用就是调用一个系统函数委托内核帮我们检测程序中一系列文件描述符的状态，内核检测完毕之后，会给用户一个反馈，用户通过内核的反馈就知道那些文件描述符有变化，有针对性的对这些文件描述符进程状态处理

4.1 select

1. 在你开始监测这些描述符时，你先将这些文件描述符全部置为0
2. 当你需要监测的描述符置为1
3. 使用select函数监听置为1的文件描述符是否有数据到来
4. 当状态为1的文件描述符有数据到来时，此时你的状态仍然为1，但是其他状态为1的文件描述如果没有数据到来，那么此时会将这些文件描述符置为0
5. 当select函数返回后，可能有一个或者多个文件描述符为1，那么你怎么知道是哪个文件描述符准备好了呢？其实select并不会告诉你说，我哪个文件描述符准备好了，他只会告诉你他的那些bit为位哪些是0，哪些是1。也就是说你需要自己用逻辑去判断你要的那个文件描是否准备好了

函数

```
#include <sys/select.h>
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);
参数：
    - nfds：是一个整数值，是指集合中所有文件描述符的范围，即所有文件描述符的最大值加1
    - readfds:readfs是一个容器，里面可以容纳多个文件描述符，把需要监视的描述符放入这个集合中，当有文件描述符可读时，select就会返回一个大于0的值，表示有文件可读
        - 两种情况：
            - 判断有没有新连接
            - 判断有没有通信数据
        - 传入传出参数
            - 传入的是委托内核检测的文件描述符集合
            - 传出的是检测到的满足条件的文件描述符
    - writefds:和readfs类似，表示有一个可写的文件描述符集合，当有文件可写时，select就会返回一个大于0的值，表示有文件可写
        - 一般情况下，文件描述符的写缓冲区都是可写的，很少用
        - 不检测指定为NULL
    - exceptfds:同上面两个参数的意图，用来监视文件错误异常文件
        - 一般情况下很少用
        - 不检测指定为NULL
    - timeout：当将timeout置为NULL时，表明此时select是阻塞的；当将tineout设置为timeout->tv_sec = 0，timeout->tv_usec = 0时，表明这个函数为非阻塞；
返回值：
        >0：检测完成之后，满足条件的文件描述符的总个数
        =0：没有检测到满足条件的文件描述符，超时时间到了强制函数返回
        -1：函数调用失败

//管理set中的fd（添加，删除，判断，清空）
//FD_SETSIZE=1024
void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

逻辑代码

```
int main()
{
    //1.创建监听的套接字
    int lfd = socket();
    //2.绑定
    bind();
    //3.给绑定成功的套接字设置监听
    listen();
    //4.设置要检测的文件描述符集合
    fd_set reads, tmp;
    FD_ZERO(&reads);
    FD_SET(lfd,&reads);
    //5.使用select检测集合中文件描述符的状态
    int nfds = lfd;
    while(1)
    {
        //reads，传入传出参数，传入要委托内核检测的集合，传出缓冲出有数据的文件描述符集合
        tmp = reads;
        int num = select(nfds+1, &tmp, NULL, NULL, NULL);
        for(int i = lfd; i<=nfds; ++i)
        {
            //有没有新连接
            if(FD_ISSET(lfd,&tmp))
            {
                //建立连接
                int cfd = accept(lfd, NULL, NULL);
                //将cfd添加到检测集合中，下次调用就可以检测到了
                FD_SET(cfd,&reads);
            }
            //有没有通信数据
            else
            {
                //除了lfd，都是通信文件描述符，说明有客户端连接
                if(FD_ISSET(i,&tmp))
                {
                    int len = read(i, buf, sizeof(buf));
                    if(len == 0)
                    {
                        //客户端断开连接，需要从reads中删除
                        FD_CLR(i, &reads);
                        close(i);
                    }
                }
            }
        }
    }
}
```

服务器代码

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

int main(int argc, char const *argv[])
{
    int lfd = socket(AF_INET, SOCK_STREAM, 0);
    if (lfd == -1)
    {
        perror("socket");
        exit(0);
    }

    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    addr.sin_port = htons(8989);

    int ret = bind(lfd, (struct sockaddr *)&addr, sizeof(addr));
    if (ret == -1)
    {
        perror("bind");
        exit(0);
    }

    ret = listen(lfd, 128);
    if (ret == -1)
    {
        perror("listen");
        exit(0);
    }

    fd_set reads, tmp;
    FD_ZERO(&reads);
    FD_SET(lfd, &reads);
    int nfds = lfd;

    while (1)
    {
        tmp = reads;
        int num = select(nfds + 1, &tmp, NULL, NULL, NULL);
        printf("num = %d\n", num);
        for (int i = 0; i <= nfds; i++)
        {
            if (i == lfd && FD_ISSET(lfd, &tmp))
            {
                int cfd = accept(lfd, NULL, NULL);
                FD_SET(cfd, &reads);
                nfds = nfds < cfd ? cfd : nfds;
            }
```

```
            else
            {
                if (FD_ISSET(i, &tmp))
                {
                    char buf[1024];
                    memset(buf, 0, sizeof(buf));
                    int len = recv(i, buf, sizeof(buf), 0);
                    if (len == 0)
                    {
                        printf("客户端断开连接...\n");
                        FD_CLR(i, &reads);
                        close(i);
                    }
                    else if (len > 0)
                    {
                        printf("recv data: %s\n", buf);
                        send(i, buf, len, 0);
                    }
                    else
                    {
                        perror("recv");
                        break;
                    }
                }
            }
        }
    }

    close(lfd);

    return 0;
}
```

4.2 poll

- 是从select到epoll的一个过渡
- 不能跨平台，只能在Linux上使用
- 特点
  - 可以检测超过1024数量的文件描述符限制，和硬件有关
  - 和select一样是线性的
  - 使用方式更加直观

4.3 epoll

- 只支持Linux
- 连接数跟内存有关
- 树状(`红黑树`)模型，检测效率高
- 委托用户检测的文件描述符集合用户跟内核使用的同一块内存，没有数据拷贝
  - 使用了共享内存
- 返回值
  - 有多少文件描述符发生变化

- 可以精确知道到底是哪个文件描述符发生变化

函数

```
#include <sys/epoll.h>
//创建一个epoll模型
int epoll_create(int size);
参数：
    - size：the size argument is ignored, but must be greater than zero.
返回值：
    - 成功：文件描述符，可以理解为红黑树根节点，通过这个文件描述符访问实例
    - 失败：-1

typedef union epoll_data {
   void        *ptr;
   int          fd; //常用的一个成员
   uint32_t     u32;
   uint64_t     u64;
} epoll_data_t;

struct epoll_event {
   uint32_t     events;      /* Epoll events */
   epoll_data_t data;        /* User data variable */
};
成员：
    - events：
        - EPOLLIN：表示对应的文件描述符可以读（包括对端SOCKET正常关闭）；
        - EPOLLOUT：表示对应的文件描述符可以写；
        - EPOLLPRI：表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）；
        - EPOLLERR：表示对应的文件描述符发生错误；
        - EPOLLHUP：表示对应的文件描述符被挂断；
        - EPOLLET：将epoll设置为边缘触发(Edge Triggered)模式（默认为水平触发），这是相对于水平触发(Level Triggered)来说的。
        - EPOLLONESHOT：  只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket的话，需要再次把这个socket加入到EPOLL队列里
    - data：要检测的fd

//对epoll树节点操作函数
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
参数：
    - epfd：epoll_create返回值
    - op：
        - EPOLL_CTL_ADD：添加节点
        - EPOLL_CTL_MOD：修改节点，比如检测 读事件 改为 写事件
        - EPOLL_CTL_DEL：删除节点
    - fd：op要操作的文件描述符
    - event：
        - op添加：设置要检测的文件描述符的事件
        - op修改：修改对应的文件描述符的事件
        - op删除：NULL

//委托内核检测epoll上的文件描述符状态，如果没有变化，该函数一直阻塞
//有满足条件的文件描述符被检测到，函数返回
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
参数：
    - epfd：epoll_create返回值
```

- events：结构体数组，传出参数，记录了当前这轮检测epoll模型中状态有变化的文件描述符信息
- maxevents：本次可以返回的最大事件数目，通常与events数组的大小是相等的
- timeout：超时时长
  - -1：阻塞到有文件描述符发生变化
  - 0：调用之后立即返回
  - >0：检测到事件发生时最多等待的时间（单位为毫秒）

返回值：

- 成功：变化的文件描述符的数量
- 失败：-1

逻辑代码

```
int main()
{
    //1.创建监听的套接字
    int lfd = socket();
    //2.绑定
    bind();
    //3.给绑定成功的套接字设置监听
    listen();
    //4.创建epoll模型
    int epfd = epoll_create();
    //5.将需要检测的文件描述符添加到epoll中
    struct epoll_event ev;
    ev.events = EPOLLIN;
    ev.data.fd = lfd;
    epoll_ctl(epfd, epoll_ctl_add, lfd, &ev);
    //6.开始检测
    struct epoll_event events[1024];
    while(1)
    {
        int num = epoll_wait(epfd, events, 1024, -1);
        //处理num个有状态变化的文件描述符
        for(int i=0; i<=num; ++1)
        {
            //更严谨的判断，如果不是读事件就忽略
            if(events[i].events & EPOLLOUT)
            {
                continue;
            }
            int curfd = events[i].data.fd;
            if(curfd = lfd)
            {
                //连接
                int cfd = accept(lfd, NULL, NULL);
                ev.data.fd = cfd;
                ev.events = EPOLLIN;
                epoll_ctl(epfd, epoll_ctl_add, cfd, &ev);
            }
            else
            {
                //通信
                int len = read(i, buf, sizeof(buf));
                if(len == 0)
                {
```

```
                    //客户端断开连接，需要从reads中删除
                    epoll_ctl(epfd, epoll_ctl_del, curfd, NULL);
                    close(curfd);
                }
                else if(len > 0)
                {
                    send();
                }
                else
                {
                    perror();
                }
            }
        }
    }
}
```

服务器代码

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

int main(int argc, char const *argv[])
{
    int lfd = socket(AF_INET, SOCK_STREAM, 0);
    if (lfd == -1)
    {
        perror("socket");
        exit(0);
    }

    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    addr.sin_port = htons(8989);

    int ret = bind(lfd, (struct sockaddr *)&addr, sizeof(addr));
    if (ret == -1)
    {
        perror("bind");
        exit(0);
    }

    ret = listen(lfd, 128);
    if (ret == -1)
    {
        perror("listen");
        exit(0);
    }

    int epfd = epoll_create(1);
```

```
    if (epfd == -1)
    {
        perror("epoll create error");
        exit(0);
    }

    struct epoll_event ev;
    ev.events = EPOLLIN;
    ev.data.fd = lfd;
    ret = epoll_ctl(epfd, EPOLL_CTL_ADD, lfd, &ev);
    if (ret == -1)
    {
        perror("epoll_ctl error");
        exit(0);
    }

    struct epoll_event evs[1024];
    while (1)
    {
        int num = epoll_wait(epfd, evs, 1024, -1);
        printf("num = %d\n", num);
        for (int i = 0; i < num; i++)
        {
            int curfd = evs[i].data.fd;
            if (curfd == lfd)
            {
                int cfd = accept(lfd, NULL, NULL);
                ev.data.fd = cfd;
                ev.events = EPOLLIN;
                epoll_ctl(epfd, EPOLL_CTL_ADD, cfd, &ev);
            }
            else
            {
                char buf[1024];
                memset(buf, 0, sizeof(buf));
                int len = recv(curfd, buf, sizeof(buf), 0);
                if (len == 0)
                {
                    printf("客户端断开连接...\n");
                    epoll_ctl(epfd, EPOLL_CTL_DEL, curfd, NULL);
                    close(curfd);
                }
                else if (len > 0)
                {
                    printf("recv data: %s\n", buf);
                    send(curfd, buf, len, 0);
                }
                else
                {
                    perror("recv");
                    break;
                }
            }
        }
    }
```

```
    close(lfd);

    return 0;
}
```

epoll工作模式

- 水平触发(level-triggered) `默认`
  - 当文件描述符关联的内核读缓冲区非空，有数据可以读取，就一直发出可读信号进行通知
  - 当文件描述符关联的内核写缓冲区不满，有空间可以写入，就一直发出可写信号进行通知
  - 支持阻塞和非阻塞的套接字
- 边缘触发(edge-triggered)
  - 当文件描述符关联的内核读缓冲区由空转化为非空的时候，则发出可读信号进行通知，即空的接收缓冲区刚接收到数据时触发读事件
  - 当文件描述符关联的内核写缓冲区由满转化为不满的时候，则发出可写信号进行通知，即满的缓冲区刚空出空间时触发读事件
  - 仅支持非阻塞的套接字
  - ```
    struct epoll_event ev;
    ev.events = EPOLLIN | EPOLLET;//设置边缘模式
    ev.data.fd = lfd;
    ret = epoll_ctl(epfd, EPOLL_CTL_ADD, lfd, &ev);
    ```
- *边缘触发仅触发一次，水平触发会一直触发*

> `libevent` 采用水平触发， `nginx` 采用边沿触发

解决边缘触发弊端

1. 循环读取数据
2. 设置数据接收动作为非阻塞 -> 修改文件描述符为非阻塞

   ```
   int flag = fcntl(cfd, F_GETFL);
   flag |= O_NOBLOCK;
   fcntl(cfd, F_SETFL, flag);
   ```
3. 读取结束判断：read返回EAGAIN或者EWOULDBLOCK

5. UDP

5.1 UDP通信

udp：面向无连接的，不安全的，报式传输协议；跟tcp一样是传输层协议

- 无连接：udp通信的时候不需要connect，只需要指定ip:port
- 不安全：会丢包，没有数据校验机制，丢了就没有了，只有全丢跟不丢
- 报式：发送端发送多少数据，接收端接受多少
   ```

服务器端通信流程

```
//1.创建通信套接字
int cfd = socket(AF_INET, SOCK_DGRAM, 0);
//2.通信套接字和本地的ip:port绑定
//绑定目的：程序启动之后不主动发送数据，先接收数据，就需要绑定端口，不绑定就自动绑定
struct scokaddr_in addr;
bind(cfd, (struct sockaddr)&addr, sizeof(addr));
//3.通信
//接收
recvfrom();
//发送
sendto();
//4.关闭通信的文件描述符
close();
```

客户端通信流程

```
//1.创建通信套接字
int cfd = socket(AF_INET, SOCK_DGRAM, 0);
//2.通信套接字和本地的ip:port绑定
//绑定目的：程序启动之后不主动发送数据，先接收数据，就需要绑定端口
struct scokaddr_in addr;
bind(cfd, (struct sockaddr)&addr, sizeof(addr));
//3.通信
//接收
recvfrom();
//发送
sendto();
//4.关闭通信的文件描述符
close();
```

函数

```
#include <sys/select.h>
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);
参数：
    - sockfd：通信的文件描述符
    - buf：指向一块有效内存地址，存储接收数据
    - len：buf指向的内存大小
    - flags：使用默认属性0
    - src_addr：传出参数，发送方地址结构体，网络字节序(大端存储)
    - addrlen：传入传出参数，结构体大小；src_addr=NULL也指定为NULL
返回值：
    >0：接收的字节数
    -1：失败
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
参数：
    - sockfd：通信的文件描述符
    - buf：指向一块有效内存地址，存储发送数据
    - len：buf指向的内存中待发送长度
    - flags：使用默认属性0
```

- dest_addr：传入参数，接收方地址结构体，网络字节序(大端存储)
- addrlen：传入参数，dest_addr结构体大小

返回值：

>0：发送的字节数

-1：失败

服务器代码

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

int main(int argc, char const *argv[])
{
    int fd = socket(AF_INET, SOCK_DGRAM, 0);
    if (fd == -1)
    {
        perror("socket");
        exit(0);
    }

    struct sockaddr_in addr, client;
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    addr.sin_port = htons(8989);
    int ret = bind(fd, (struct sockaddr *)&addr, sizeof(addr));
    if (ret == -1)
    {
        perror("bind");
        exit(0);
    }

    char ip[24];
    char buf[1024];
    int clilen = sizeof(client);
    while (1)
    {
        int len = recvfrom(fd, buf, sizeof(buf), 0, (struct sockaddr *)&client,
&clilen);
        if (len == -1)
        {
            break;
        }
        printf("client ip:%s, port:%d\n",
               inet_ntop(AF_INET, &client.sin_addr.s_addr, ip, sizeof(ip)),
               ntohs(client.sin_port));
        printf("client say: %s\n", buf);

        sendto(fd, buf, strlen(buf) + 1, 0, (struct sockaddr *)&client, clilen);
    }

    close(fd);
```

```
    return 0;
}
```

客户端代码

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

int main(int argc, char const *argv[])
{
    int fd = socket(AF_INET, SOCK_DGRAM, 0);
    if (fd == -1)
    {
        perror("socket");
        exit(0);
    }

    struct sockaddr_in server;
    server.sin_family = AF_INET;
    inet_pton(AF_INET, "127.0.0.1", &server.sin_addr.s_addr);
    server.sin_port = htons(8989);
    int serverlen = sizeof(server);

    int num = 0;
    char buf[1024];
    char ip[24];
    while (1)
    {
        sprintf(buf, "hello world!,%d...", num++);
        sendto(fd, buf, strlen(buf) + 1, 0, (struct sockaddr *)&server,
serverlen);
        int len = recvfrom(fd, buf, sizeof(buf), 0, NULL, NULL);
        if (len == -1)
        {
            break;
        }
        printf("server say: %s\n", buf);
        sleep(1);
    }

    close(fd);

    return 0;
}
```

5.2 广播

向子网中多台计算机发送消息，并且子网中所有计算机都可以接收到发送方发送的消息，每个广播消息都包含一个特殊的ip地址，这个ip中子网内主机标志部分的二进制全部为1

- 广播是1对N，如果是1对1，则地址是实际ip地址
- 广播需要一个特殊的地址：192.168.x.255

- 只能在局域网使用，广域网不支持
- 只要在局域网访问内，并且终端绑定了广播端口，那么就能收到广播消息，无法拒绝

> 特点：
>
> - 广播发送端的开销很小，只是使用了广播地址，数据就可以发送到多个接收端上
> - 只能局域网使用
> - 发送广播的一端必须要设置广播属性，广播的消息发送到广播地址上

通信流程

- 数据发送端 -> 只有一个

```
//1.创建通信套接字
int fd = socket(af_inet, sock_dgram, 0);
//2.因为是主动发送数据，因此不需要手动绑定端口
//  需要设置广播属性，setsockopt()函数
setsockopt();
//3.初始化接收端信息
//  广播地址：192.168.x.255
//  port：接收广播端绑定的端口
sendto();
//4.关闭套接字
close();
```

- 数据接收端 -> 有N个

```
//1.创建通信套接字
int fd = socket(af_inet, sock_dgram, 0);
//2.被动接收数据，需要手动绑定端口
bind();
//3.接收数据
recvfrom();
//4.关闭套接字
close();
```

- 广播属性设置

```
#include <sys/socket.h>
int setsockopt(int sockfd, int level, int optname,
               const void *optval, socklen_t optlen);
参数：
    - level：SOL_SOCKET
    - optname：SO_BROADCAST
    - optval:enable a boolean option, or zero if the option is to be disabled
    - optlen:sizeof(optval)
```

广播代码

```
//服务器端
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

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

int main(int argc, char const *argv[])
{
    int fd = socket(AF_INET, SOCK_DGRAM, 0);
    if (fd == -1)
    {
        perror("socket");
        exit(0);
    }

    int opt = 1;
    setsockopt(fd, SOL_SOCKET, SO_BROADCAST, &opt, sizeof(opt));

    struct sockaddr_in client;
    client.sin_family = AF_INET;
    inet_pton(AF_INET, "192.168.164.255", &client.sin_addr.s_addr);
    client.sin_port = htons(8989);

    int num = 0;
    char buf[1024];
    while (1)
    {
        sprintf(buf, "广播数据%d:不要回答！！！", num++);
        sendto(fd, buf, strlen(buf) + 1, 0, (struct sockaddr *)&client,
sizeof(client));
        printf("广播数据：%s\n", buf);
        sleep(1);
    }

    close(fd);

    return 0;
}
//客户端
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>

int main(int argc, char const *argv[])
{
    int fd = socket(AF_INET, SOCK_DGRAM, 0);
    if (fd == -1)
    {
        perror("socket");
        exit(0);
    }

    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    addr.sin_port = htons(8989);
    int ret = bind(fd, (struct sockaddr *)&addr, sizeof(addr));
```

```
    if (ret == -1)
    {
        perror("bind");
        exit(0);
    }

    char buf[1024];
    while (1)
    {
        memset(buf, 0, 1024);
        int len = recvfrom(fd, buf, sizeof(buf), 0, NULL, NULL);
        if (len == -1)
        {
            break;
        }
        printf("广播端 say: %s\n", buf);
        sleep(1);
    }

    close(fd);

    return 0;
}
```

5.3 组播（多播）

5.4 本地套接字

本地套接字是进程间通信的一种实现方式，不需要使用ip:port，需要使用套接字文件，套接字文件中不存储数据，数据在内核中的某块内存中存储，套接字文件关联着内核中的内存，通过文件描述符进行访问内存读写

- 基于tcp实现 -> 推荐
- 基于udp实现

> - 管道：有名管道，匿名管道 -> 简单，推荐使用
> - 内存映射区：不阻塞
> - 信号：携带信息量少，并且信号优先级太高，打乱程序的执行顺序 -> 不推荐
> - 本地套接字：基于网络套接字实现
> - 共享内存：效率很高，但是不阻塞

服务器端通信流程

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

//1.创建监听的套接字
int lfd = socket(AF_UNIX, SOCK_STREAM, 0);
参数:
    - domain: AF_UNIX/AF_LOCAL
```

```
    - type：SOCK_STREAM/SOCK_DGRAM
    - 0
//2.监听套接字和本地的套接字文件绑定
struct sockaddr_un addr;
bind(lfd, (struct sockaddr *)&addr, sizeof(addr));
//3.设置监听
listen();
//4.等待并接收连接
accept(lfd, (struct sockaddr *)&cliaddr, &len;
//5.通信
//接收
recv();
//发送
send();
//6.断开连接
close();
```

客户端通信流程

```
//1.创建通信的套接字
int cfd = socket(AF_UNIX, SOCK_STREAM, 0);
//2.监听套接字和本地的套接字文件绑定
struct sockaddr_un addr;
bind(lfd, (struct sockaddr *)&addr, sizeof(addr));
//3.连接服务器进程
struct sockaddr_un servaddr;
connect(cfd, (struct sockaddr *)&servaddr, sizeof(servaddr));
//4.通信
//接收
recv();
//发送
send();
//5.断开连接
close();
```

实现代码

```
//服务器
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#include <sys/un.h>

int main(int argc, char const *argv[])
{
    int lfd = socket(AF_LOCAL, SOCK_STREAM, 0);
    if (lfd == -1)
    {
        perror("socket");
        exit(0);
    }

    struct sockaddr_un addr;
```

```
    addr.sun_family = AF_LOCAL;
    strcpy(addr.sun_path, "./server.sock");
    int ret = bind(lfd, (struct sockaddr *)&addr, sizeof(addr));
    if (ret == -1)
    {
        perror("bind");
        exit(0);
    }

    ret = listen(lfd, 128);

    struct sockaddr_un cliaddr;
    int clilen = sizeof(cliaddr);
    int cfd = accept(lfd, (struct sockaddr *)&cliaddr, &clilen);
    if (cfd == -1)
    {
        perror("accept");
        exit(0);
    }
    printf("客户端本地套接字路径：%s\n", cliaddr.sun_path);

    while (1)
    {
        char buf[1024];
        memset(buf, 0, 1024);
        int len = recv(cfd, buf, 1024, 0);
        if (len == 0)
        {
            printf("客户端断开连接...\n");
            break;
        }
        else if (len > 0)
        {
            printf("client say: %s\n", buf);
            send(cfd, buf, sizeof(buf), 0);
        }
        else
        {
            perror("recv");
            break;
        }
        sleep(1);
    }

    close(cfd);
    close(lfd);
    return 0;
}
//客户端
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <arpa/inet.h>
#include <sys/un.h>
```

```c
int main(int argc, char const *argv[])
{
    int cfd = socket(AF_LOCAL, SOCK_STREAM, 0);
    if (cfd == -1)
    {
        perror("socket");
        exit(0);
    }

    struct sockaddr_un addr;
    addr.sun_family = AF_LOCAL;
    strcpy(addr.sun_path, "./client.sock");
    int ret = bind(cfd, (struct sockaddr *)&addr, sizeof(addr));
    if (ret == -1)
    {
        perror("bind");
        exit(0);
    }

    struct sockaddr_un servaddr;
    servaddr.sun_family = AF_LOCAL;
    strcpy(servaddr.sun_path, "./server.sock");
    connect(cfd, (struct sockaddr *)&servaddr, sizeof(servaddr));
    printf("服务器本地套接字路径：%s\n", servaddr.sun_path);

    int num = 0;
    while (1)
    {
        char buf[1024];
        sprintf(buf, "本地套接字通信：%d", num++);
        send(cfd, buf, strlen(buf) + 1, 0);

        int len = recv(cfd, buf, 1024, 0);
        if (len == 0)
        {
            printf("服务器断开连接...\n");
            break;
        }
        else if (len > 0)
        {
            printf("server say: %s\n", buf);
        }
        else
        {
            perror("recv");
            break;
        }
        sleep(1);
    }

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

6. libevent

- 事件驱动，高性能；
- 轻量级，专注于网络；
- 跨平台，支持 Windows、Linux、Mac Os等；
- 支持多种 I/O多路复用技术， epoll、poll、dev/poll、select 和kqueue 等；
- 支持 I/O，定时器和信号等事件；