1.TCP协议简介

TCP是TCP/IP体系中非常复杂的一个协议，它有以下特点：

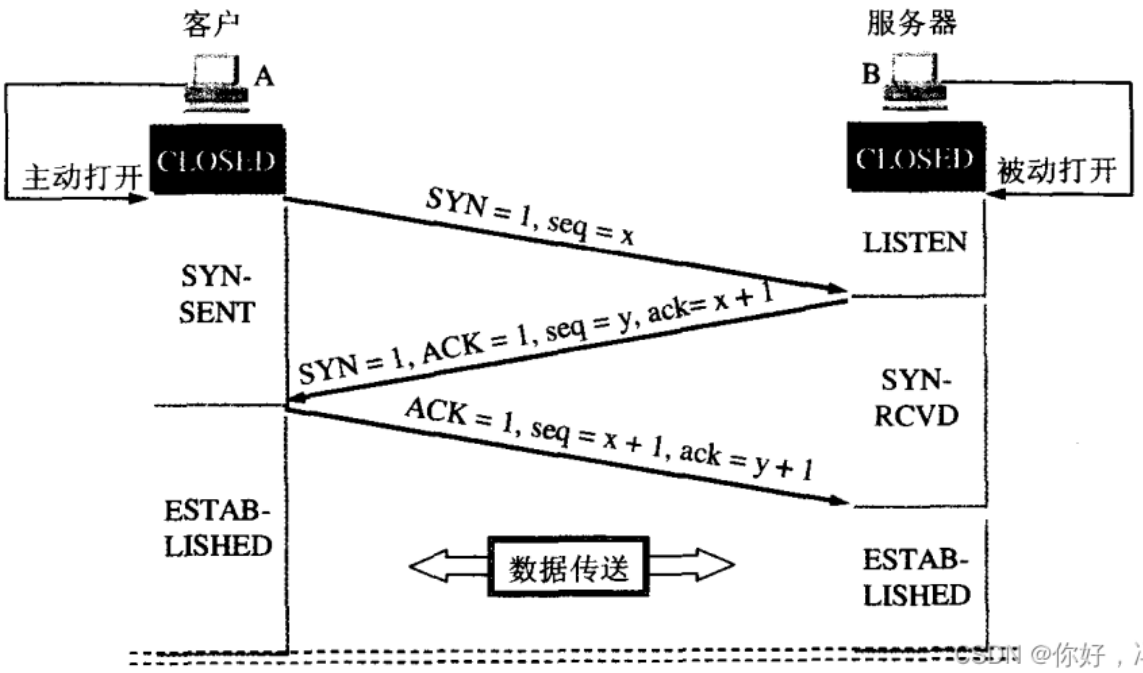
- TCP是面向连接的运输层协议。这就是说，应用程序在使用TCP协议之前，必须先建立TCP连接。在传送数据完毕后，必须释放已经建立的TCP连接。这就是说，应用进程之间的通信好像在“打电话”:通话前要先拨号建立连接，通话结束后要挂机释放连接。
- 每一条TCP连接只能有两个端点， 每一 条TCP连接只能是点对点的
- TCP提供可靠交付的服务。也就是说，通过TCP连接传送的数据，无差错、不丢失、不重复、并且按序到达。
- TCP提供全双工通信。TCP允许通信双方的应用进程在任何时候都能发送数据。TCP连接的两端都设有发送缓存和接收缓存，用来临时存放双向通信的数据。在发送时，应用程序在把数据传送给TCP的缓存后，就可以做自己的事，而TCP在合适的时候把数据发送出去。在接收时，TCP把收到的数据放入缓存，.上层的应用进程在合适的时候读取缓存中的数据。
- 面向字节流。TCP中的“流"指的是流入到进程或从进程流出的字节序列。面向字节流” 的含义是:虽然应用程序和TCP的交互是一次一个数据块(大小不等)，但TCP把应用程序交下来的数据看成仅仅是一连串的无结构的字节流。也就是说TCP不保证接收方应用程序所收到的数据块和发送方应用程序所发出的数据块具有对应大小的关系。

总的来说：TCP(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

1.1.TCP的连接建立（三次握手）

三次握手其实就是指建立一个TCP连接时，需要客户端和服务器总共发送3个包。进行三次握手的主要作用就是为了确认双方的接收能力和发送能力是否正常、指定自己的初始化序列号为后面的可靠性传送做准备。实质上其实就是连接服务器指定端口，建立TCP连接，并同步连接双方的序列号和确认号，交换TCP窗口大小信息。

下面来看看三次握手的流程图：

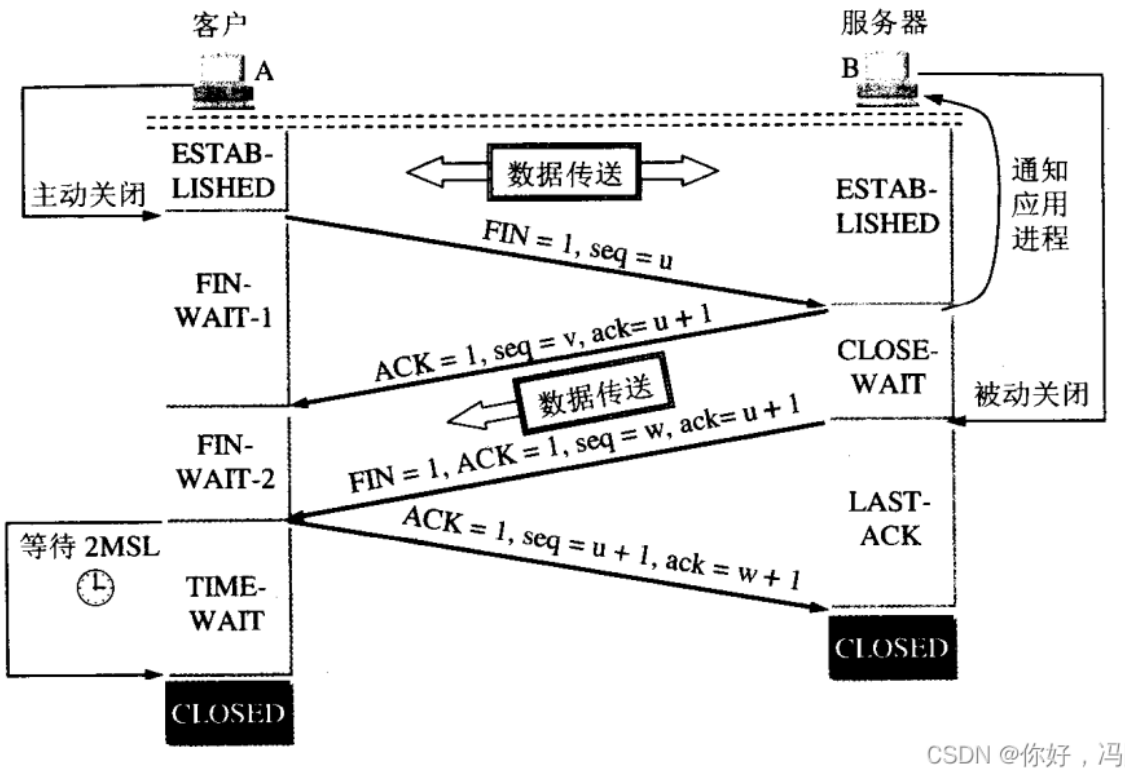


- 第一次握手：A的TCP客户进程也是首先创建传输控制模块TCB，然后向B发出连接请求报文段，这时首部中的同步位SYN = 1， 同时选择-一个初始序号seq = x。TCP规定，SYN报文段(即SYN = 1的报文段)不能携带数据，但要消耗掉一个序号。这时，TCP客户进程进入SYN- SENT(同步已发送)状态。
- 第二次握手：B收到连接请求报文段后，如同意建立连接，则向A发送确认。在确认报文段中应把SYN位和ACK位都置1，确认号是ack=x + 1，同时也为自己选择一个初始序号seq= y。请注意，这个报文段也不能携带数据，但同样要消耗掉一个序号。 这时TCP服务器进程进入SYN-RCVD (同步收到)状态。
- 第三次握手：TCP客户进程收到B的确认后，还要向B给出确认。确认报文段的ACK置1，确认号ack=y+1，而自己的序号seq=x+ 1。TCP的标准规定，ACK报文段可以携带数据。但如果不携带数据则不消耗序号，在这种情况下，下一个数据报文段的序号仍是seq=x+ 1。这时，TCP连接已经建立，A进入ESTABLISHED (已建立连接)状态。当B收到A的确认后，也进入ESTABLISHED状态。

1.2.TCP连接释放（四次挥手）

- 四次挥手即终止TCP连接，就是指断开一个TCP连接时，需要客户端和服务端总共发送4个包以确认连接的断开。在socket编程中，这一过程由客户端或服务端任一方执行close来触发。
- 由于TCP连接是全双工的，因此，每个方向都必须要单独进行关闭，这一原则是当一方完成数据发送任务后，发送一个FIN来终止这一方向的连接，收到一个FIN只是意味着这一方向上没有数据流动了，即不会再收到数据了，但是在这个TCP连接上仍然能够发送数据，直到这一方向也发送了FIN。首先进行关闭的一方将执行主动关闭，而另一方则执行被动关闭。

下面来看看四次挥手的流程图：



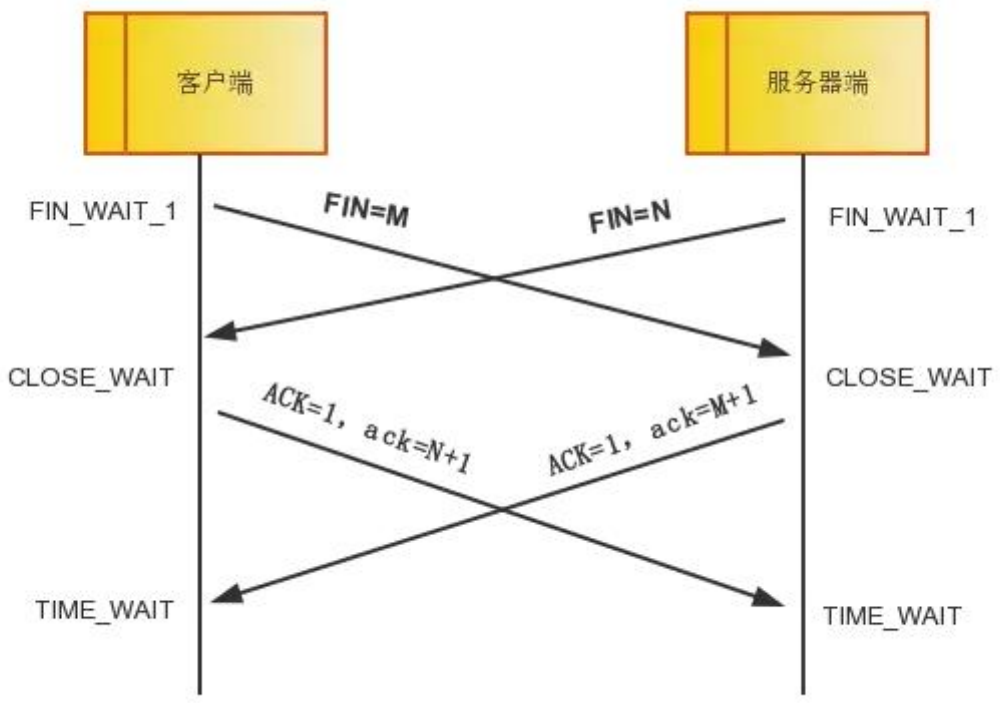
- 第一次挥手：A的应用进程先向其TCP发出连接释放报文段，并停止再发送数据，主动关闭TCP连接。A把连接释放报文段首部的FIN置1，其序号seq=u,它等于前面已传送过的数据的最后一个字节的序号加1。这时A进入FIN-

WAIT-1 (终止等待1)状态，等待B的确认。请注意，TCP规定，FIN报文段即使不携带数据，它也消耗掉一个序号。

- 第二次挥手：B收到连接释放报文段后即发出确认，确认号是ack=u+ 1，而这个报文段自己的序号是v，等于B前面已传送过的数据的最后一个字节的序号加1。然后B就进入CLOSE-WAIT (关闭等待)状态。TCP服务器进程这时应通知高层应用进程，因而从A到B这个方向的连接就释放了，这时的TCP连接处于半关闭状态，即A已经没有数据要发送了，但B若发送数据，A仍要接收。也就是说，从B到A这个方向的连接并未关闭。这个状态可能会持续一些时间。A收到来自B的确认后，就进入FIN-WAIT-2 (终止等待2)状态，等待B发出的连接释放报文段。
- 第三次挥手：若B已经没有要向A发送的数据，其应用进程就通知TCP释放连接。这时B发出的连接释放报文段必须使FIN = 1。现假定B的序号为w (在半关闭状态B可能又发送了一些数据)。B还必须重复上次已发送过的确认号ack= u + 1。这时B就进入LAST-ACK (最后确认)状态，等待A的确认。
- 第四次挥手：A在收到B的连接释放报文段后，必须对此发出确认。在确认报文段中把ACK置1,确认号ack=w + 1，而自己的序号是seq=u+ 1 (根据TCP标准，前面发送过的FIN报文段要消耗一个序号)。然后进入到TIME-WAIT (时间等待)状态。请注意，现在TCP连接还没有释放掉。必须经过时间等待计时器设置的时间2MSL后，A才进入到CL OSED状态，才能开始建立下一个新的连接。当A撤销相应的传输控制块TCB后，就结束了这次的TCP连接。

1.3.上面是一方主动关闭，另一方被动关闭的情况，实际中还会出现同时发起主动关闭的情况

具体流程如下图（同时挥手）：



2.UDP协议简介

UDP (User Datagram Protocol) 是在 OSI 七层模型中的传输层上的一种协议。它和 TCP 类似是用来传输数据的，但是 UDP 更加简单、高效、灵活，适用于对数据传输速度要求较高，但对可靠性要求不高的场景，例如游戏、音频、视频等实时通讯场景。UDP 的工作原理和应用场景都有很大区别于 TCP，本文将详细介绍 UDP 协议的基本原理、特点、应用场景、优缺点以及使用实例。

UDP 协议最主要的特点如下：

- 不可靠性：UDP 传输的数据并不会进行校验和确认，也不会重复发送，无法保证数据的可靠性。如果某个数据包在传输过程中丢失或损坏，接收方将无法得到这个数据包。
- 无序性：UDP 协议是无序的，发送的数据可能会经过不同的路径到达目标地址，因此接收方可能无法按照发送顺序对数据进行组装。
- 基于数据报文：UDP把应用层提交给它的数据报文，添加上UDP首部后传输，每个UDP数据报的大小不能超过64KB（含首部）。
- 无连接：UDP协议在传输数据时不需要建立连接，收发双方都不需要了解对方的情况，只需要按照协议格式互相传输数据即可。
- 快速：因为UDP没有确认机制，不需要等待对方的回应就可以发送下一个数据报，所以UDP传输速度比TCP快。
- 轻量级：UDP协议的头部非常简单，只有8个字节，相较于TCP协议，UDP协议的数据头较小，需要的传输数据大小更小。
- 支持多播和广播：UDP协议支持一对多的数据传输方式，可以将数据报发送给指定的多个主机，从而减少网络负载。

UDP 协议的数据传输流程比 TCP 简单得多，只需要两个步骤：

1. 发送数据：向目标地址和端口号发送需要传输的数据。
2. 接收数据：接收从源地址和源端口号发来的数据包。

2.1.UDP协议之组播和广播

UDP协议支持的另一个重要特性是组播（Multicast）和广播（Broadcast）。在UDP中，广播和组播都是用于将数据同时传输给多个接收方的方式。

- 广播（Broadcast）是指将数据一次性发送给网络中的所有主机，然后所有的主机都会收到这个数据包。广播地址通常是网络中的一个特定地址，例如255.255.255.255是IPv4网络中的广播地址。通常情况下，广播只用于特定的场景，例如DHCP服务器在向客户端分配IP地址时使用广播。
- 组播（Multicast）是指将数据只发送给特定的一组主机，相比广播，它能够更加优雅地在网络上进行数据分发。组播通常需要进行满足一些条件的专门设置，以便于网络设备能够正确地对组播数据进行管理和转发。组播地址通常是在一个特定的IP地址范围内，例如224.0.0.0到239.255.255.255之间的地址都是用于组播。组播通常用于实时视频和音频流等多媒体数据的传输，可以降低网络传输的负载，提高传输效率。

3.C语言实现TCP

3.1.函数接口

3.1.1.创建套接字

函数原型：

```
int socket(int domain, int type, int protocol);
```

参数说明：

```
domain：协议族
      AF_UNIX, AF_LOCAL   本地通信
      AF_INET             ipv4
      AF_INET6            ipv6
type：套接字类型
      SOCK_STREAM:流式套接字
      SOCK_DGRAM：数据报套接字
protocol：协议 - 填0 自动匹配底层 ，根据type系统默认自动帮助匹配对应协议
           传输层：IPPROTO_TCP、IPPROTO_UDP、IPPROTO_ICMP
           网络层：htons(ETH_P_IP|ETH_P_ARP|ETH_P_ALL)
```

返回值：

```
成功：文件描述符
失败：-1
```

3.1.2.绑定IP和端口

函数原型：

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

参数说明：

```
sockfd：文件描述符
```

```
addr：通用结构体，根据socket第一个参数选择的通信方式最终确定这需要真正填充传递的结构体是那个类型。强转后传参数。
addrlen：填充的结构体的大小
```

返回值：

```
成功：0
失败：-1
```

通用结构体：相当于预留一个空间

```
struct sockaddr {
    sa_family_t sa_family;
    char sa_data[14];
}
```

ipv4的结构体：

```
 struct sockaddr_in {
    sa_family_t    sin_family;  //协议族AF_INET
    in_port_t      sin_port;  //端口
    struct in_addr sin_addr;
};
 struct in_addr {
    uint32_t       s_addr;   //IP地址
};
```

本地址通信结构体：

```
  struct sockaddr_un {
    sa_family_t sun_family;  //AF_UNIX
    char        sun_path[108]; //在本地创建的套接字文件的路径及名字
};
```

ipv6通信结构体：

```
struct sockaddr_in6 {
    sa_family_t     sin6_family;
    in_port_t       sin6_port;
    uint32_t        sin6_flowinfo;
    struct in6_addr sin6_addr;
    uint32_t        sin6_scope_id;
};
struct in6_addr {
    unsigned char s6_addr[16];
};
```

3.1.3.监听，将主动套接字变为被动套接字

函数原型：

```
int listen(int sockfd, int backlog);
```

参数说明：

```
 sockfd：套接字
 backlog：同时响应客户端请求链接的最大个数，不能写0，不同平台可同时链接的数不同，一般写6-8个
          (队列1：保存正在连接)
          (队列2，连接上的客户端)
```

返回值：

```
成功：0
失败：-1
```

3.1.4.阻塞等待客户端的连接请求

函数原型：

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

参数说明：

```
Sockfd ： 套接字
addr：链接客户端的ip和端口号
      如果不需要关心具体是哪一个客户端，那么可以填NULL;
addrlen：结构体的大小
         如果不需要关心具体是哪一个客户端，那么可以填NULL;
```

返回值：

```
成功：文件描述符; //用于通信
失败：-1
```

3.1.5.接收数据

函数原型：

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

参数说明：

```
sockfd：acceptfd
buf：接受数据的缓冲区
len：接收数据的缓冲区大小
flags：一般填0，相当于read()函数
MSG_DONTWAIT：非阻塞
```

返回值：

```
< 0  失败出错  更新errno
==0  表示客户端退出
```

```
> 0  成功接收的字节个数
```

3.1.6.用于连接服务器

函数原型：

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

参数说明：

```
sockfd：socket函数的返回值
addr：填充的结构体是服务器端的；
addrlen：结构体的大小
```

返回值：

```
成功：0
失败：-1
```

3.1.7.发送数据

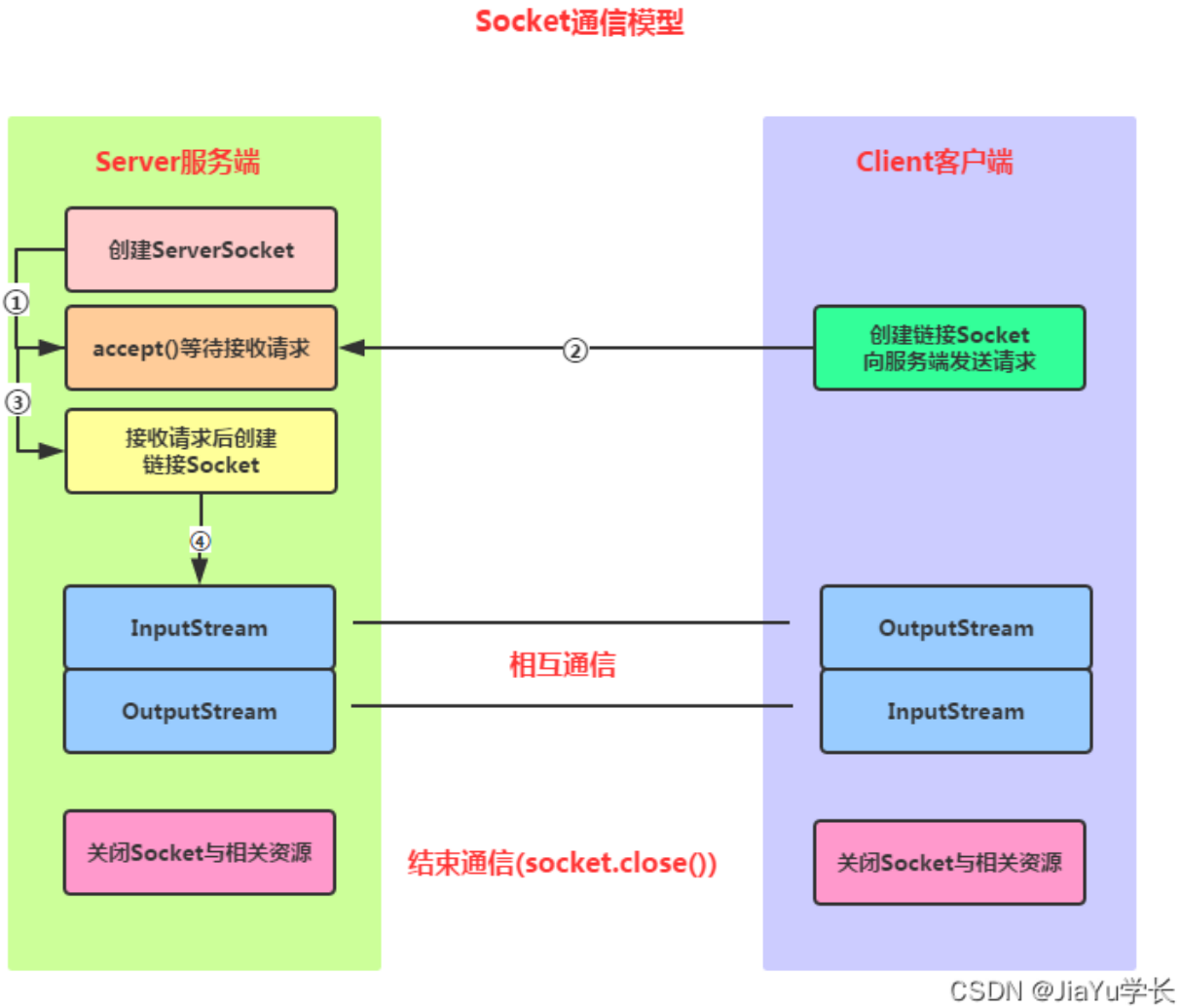
函数原型：

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

参数说明：

```
sockfd:socket函数的返回值
buf：发送内容存放的地址
len：发送内存的长度
flags：如果填0，相当于write();
```

3.2.示例



服务端：

```
#include <stdio.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <stdlib.h>
 #include <sys/wait.h>
 #include <string.h>

int main(int argc, char const *argv[])
```

```
{
    if (argc != 2) {
        printf("please input %s <port>\n", argv[0]);
        return -1;
    }
    int sockfd, acceptfd;
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0){
        perror("socket err.");
        return -1;
    }
    struct sockaddr_in serveraddr, caddr;
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(atoi(argv[1]));
    // serveraddr.sin_addr.s_addr=inet_addr(argv[1]);

    //自动获取ip
    //serveraddr.sin_addr.s_addr=htonl(INADDR_ANY);
    // serveraddr.sin_addr.s_addr=INADDR_ANY; //0.0.0.0
    serveraddr.sin_addr.s_addr = inet_addr("0.0.0.0");

         socklen_t len = sizeof(caddr);

    if (bind(sockfd, (struct sockaddr *)&serveraddr, sizeof(serveraddr)) < 0) {
        perror("bind err.");
        return -1;
    }

    if (listen(sockfd, 5) < 0) {
        perror("listen err.");
        return -1;
    }
    while (1) {
        acceptfd = accept(sockfd, (struct sockaddr *)&caddr, &len);
        if (acceptfd < 0) {
            perror("accept err.");
            return -1;
        }
        printf("client:ip=%s  port=%d\n",
        inet_ntoa(caddr.sin_addr), ntohs(caddr.sin_port));
```

```
54
55          char buf[128];
56          pid_t pid = fork();
57          if (pid < 0) {
58              perror("fork err.");
59              return -1;
60          }
61          else if (pid == 0){ //发送

62              int sendbyte;
63              while (1) {
64                  fgets(buf, sizeof(buf), stdin);
65                  if (buf[strlen(buf) - 1] == '\n')
66                      buf[strlen(buf) - 1] = '\0';
67                  sendbyte = send(acceptfd, buf, sizeof(buf), 0);
68
                        if (sendbyte < 0){
69                      perror("send error.");
70                      return -1;
71                  }
72              }
73          }
74          else{
75
                    int recvbyte;
76              while (1) {
77                  recvbyte = recv(acceptfd, buf, sizeof(buf), 0);
78                  if (recvbyte < 0) {
79                      perror("recv err.");
80                      return -1;
81                  }
82                  else if (recvbyte == 0) {
83                      printf("client exit.\n");
84                      kill(pid,SIGKILL);
85                      wait(NULL);
86                      break;
87                  }
88                  else {
89                      printf("buf:%s\n", buf);
90                  }
```

```
            }
            close(acceptfd);
        }
    }
    close(sockfd);
    return 0;
}
```

客户端：

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <sys/wait.h>

 int main(int argc, char const *argv[])
{
    if (argc != 3) {
        printf("please input %s <ip> <port>\n", argv[0]);
        return -1;
    }
    //1.创建套接子
    int sockfd;
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0) {
        perror("socket error.");
        return -1;
    }
    printf("sockfd=%d\n", sockfd);
    //填充ipv4的通信结构体  服务器的ip和端口
    struct sockaddr_in serveraddr;
```

```
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(atoi(argv[2]));
    serveraddr.sin_addr.s_addr = inet_addr(argv[1]);

    //2.请求链接服务器
    if (connect(sockfd, (struct sockaddr *)&serveraddr, sizeof(serveraddr)) < 0) {
        perror("connect error.");
        return -1;
    }

    //5.循环发送消息  通信
    char buf[128];
    pid_t pid = fork();
    if (pid < 0) {
        perror("fork err.");
        return -1;
    }
    else if (pid == 0) {
        int recvbyte;
        while (1) {
            recvbyte = recv(sockfd, buf, sizeof(buf), 0);
            if (recvbyte < 0) {
                perror("recv err.");
                return -1;
            }
            printf("buf:%s\n", buf);
        }
    }
    else {
        int sendbyte;
        while (1) {
            fgets(buf, sizeof(buf), stdin);
            if (buf[strlen(buf) - 1] == '\n')
                buf[strlen(buf) - 1] = '\0';
            sendbyte = send(sockfd, buf, sizeof(buf), 0);
            if (sendbyte < 0) {
                perror("send error.");
                return -1;
            }
            if(strncmp(buf,"quit",4)==0) {
```

```
                    kill(pid,SIGKILL);
                    wait(NULL);
                    exit(-1);
                }
            }
        }
        close(sockfd);
        return 0;
}
```

4.C语言实现UDP

4.1.函数接口

4.1.1.接收数据

```
ssize_t recvfrom(int sockfd,void*buf,size_t len,int flags,struct sockaddr *
src_addr,socklen_t * addrlen);
```

参数说明：

```
sockfd：套接字描述符
buf:接收缓存区的首地址
len：接收缓存区的大小
flags：0//调用方式标志位
src_addr:发送端的网络信息结构体的指针
addrlen：发送端的网络信息结构体的大小的指针
```

返回值：

```
成功：接收的字节个数
失败：-1
0:客户端退出
```

4.1.2.发送数据

```
ssize_t sendto(int sockfd,constvoid*buf,size_t len,int flags,const struct sockaddr*
dest_addr,socklen_t addrlen);
```

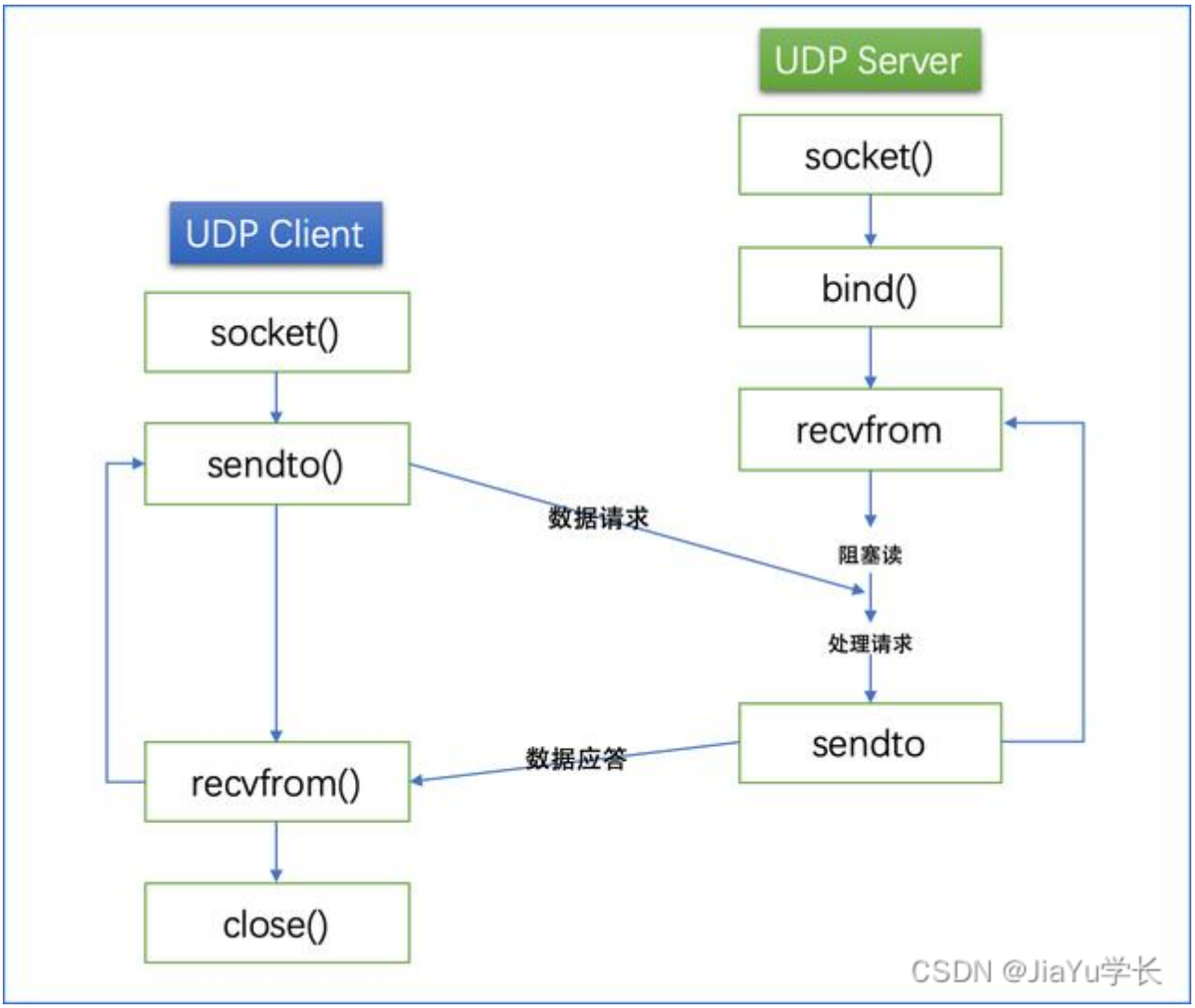
参数说明：

```
sockfd：套接字描述符
buf:发送缓存区的首地址
len：发送缓存区的大小
flags：0
src_addr:接收端的网络信息结构体的指针
addrlen：接收端的网络信息结构体的大小
```

返回值：

```
成功：发送的字节个数
失败：-1
```

4.2.示例



服务端：

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

#define MAX_BUFFER_SIZE 1024
#define SERVER_PORT 8888

int main() {
    int sockfd;
    struct sockaddr_in server_addr, client_addr;
    char buffer[MAX_BUFFER_SIZE];

    // 创建UDP套接字
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
```

```
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    memset(&server_addr, 0, sizeof(server_addr));
    memset(&client_addr, 0, sizeof(client_addr));

    // 设置服务器地址和端口
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = INADDR_ANY;
    server_addr.sin_port = htons(SERVER_PORT);

    // 绑定服务器地址和端口
    if (bind(sockfd, (const struct sockaddr*)&server_addr, sizeof(server_addr)) < 0) {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }

    printf("Server running on port %d...\n", SERVER_PORT);

    while (1) {
        // 接收来自客户端的数据
        memset(buffer, 0, sizeof(buffer));
        socklen_t client_len = sizeof(client_addr);
        ssize_t message_size = recvfrom(sockfd, buffer, sizeof(buffer) - 1, 0, (struct sockaddr*)&client_addr, &client_len);
        if (message_size < 0) {
            perror("recvfrom failed");
            exit(EXIT_FAILURE);
        }

        // 打印客户端发送的数据
        printf("Client message: %s\n", buffer);

        // 向客户端发送响应
        if (sendto(sockfd, buffer, message_size, 0, (struct sockaddr*)&client_addr, sizeof(client_addr)) < 0) {
            perror("sendto failed");
            exit(EXIT_FAILURE);
        }
```

```
    }

    // 关闭套接字
    close(sockfd);

    return 0;
}
```

客户端：

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

#define MAX_BUFFER_SIZE 1024
#define SERVER_IP "127.0.0.1"
#define SERVER_PORT 8888

int main() {
    int sockfd;
    struct sockaddr_in server_addr;
    char buffer[MAX_BUFFER_SIZE];

    // 创建UDP套接字
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    memset(&server_addr, 0, sizeof(server_addr));

    // 设置服务器地址和端口
    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.s_addr = inet_addr(SERVER_IP);
    server_addr.sin_port = htons(SERVER_PORT);

```

```
    // 从标准输入读取数据
    printf("Enter message: ");
    fgets(buffer, MAX_BUFFER_SIZE, stdin);

    // 发送数据到服务器
    sendto(sockfd, buffer, strlen(buffer), 0, (struct sockaddr*)&server_addr, sizeof(server_addr));

    // 等待接收服务器的响应
    memset(buffer, 0, sizeof(buffer));
    recvfrom(sockfd, buffer, sizeof(buffer), 0, NULL, NULL);

    // 打印服务器的响应
    printf("Server response: %s\n", buffer);

    // 关闭套接字
    close(sockfd);

    return 0;
}
```

5.Qt实现TCP

5.1.客户端

```
#include "clientwidget.h"
#include "ui_clientwidget.h"

ClientWidget::ClientWidget(QWidget *parent)
    : QWidget(parent)
    , ui(new Ui::ClientWidget)
{
    ui->setupUi(this);
    tcpsocket=nullptr;
    setWindowTitle("客户端");

    tcpsocket=new QTcpSocket(this);
```

```
    connect(tcpsocket,&QTcpSocket::connected,[=](){
        ui->textEditRead->setText("服务器连接成功! ");
    });

    connect(tcpsocket,&QTcpSocket::readyRead,[=](){
       //获取通信套接字的内容
        QString str=tcpsocket->readAll();
        //在显示编辑区域显示
       ui->textEditRead->append("服务器端: "+str);//不用settext 这样会覆盖之前的消息
    });

}

ClientWidget::~ClientWidget()
{
    delete ui;
}


void ClientWidget::on_buttonconnect_clicked()
{
    if(nullptr==ui->lineEditIP || nullptr==ui->lineEditPort)
        return ;
    //获取IP地址和端口号
    QString IP=ui->lineEditIP->text();
    quint16 Port=ui->lineEditPort->text().toInt();

    //与服务器连接
    tcpsocket->connectToHost(IP,Port);
}

void ClientWidget::on_buttonsend_clicked()
{
    if(nullptr==tcpsocket)//连接失败则不发送
        return;

    //获取发送的信息
    QString str=ui->textEditWrite->toPlainText();

    //将信息写入到通信套接字
```

```
    tcpsocket->write(str.toUtf8().data());

    //将自己的信息显示在聊天窗口
    ui->textEditRead->append("客服端: "+str);//不用settext 这样会覆盖之前的消息
}


void ClientWidget::on_buttonclose_clicked()
{
    if(nullptr==tcpsocket)
        return;
    tcpsocket->disconnectFromHost();//断开与服务器的连接
    tcpsocket->close();//关闭通信套接字
}
```

5.2.服务端

```
#include "serverwidget.h"
#include "ui_serverwidget.h"

serverWidget::serverWidget(QWidget *parent)
    : QWidget(parent)
    , ui(new Ui::serverWidget)
{
    ui->setupUi(this);
    tcpserver=nullptr;
    tcpsocket=nullptr;
    //创建监听套接字
    tcpserver=new QTcpServer(this);//指定父对象 回收空间

    //bind+listen
    tcpserver->listen(QHostAddress::Any,8888);//绑定当前网卡所有的ip 绑定端口 也就是设置服务器地址和端口号

    //服务器建立连接
    connect(tcpserver,&QTcpServer::newConnection,[=](){
        //取出连接好的套接字
        tcpsocket=tcpserver->nextPendingConnection();
```

```
21
22          //获得通信套接字的控制信息
23          QString ip=tcpsocket->peerAddress().toString();//获取连接的 ip地址
24          quint16 port=tcpsocket->peerPort();//获取连接的 端口号
25          QString temp=QString("[%1:%2] 客服端连接成功").arg(ip).arg(port);
26          //显示连接成功
27          ui->textEditRead->setText(temp);
28
29          //接收信息   必须放到连接中的槽函数 不然tcpsocket就是一个野指针
30          connect(tcpsocket,&QTcpSocket::readyRead,[=](){
31              //从通信套接字中取出内容
32              QString str=tcpsocket->readAll();
33               //在编辑区域显示
34              ui->textEditRead->append("客户端: "+str);//不用settext 这样会覆盖之前的消息
35          });
36      });
37
38  }
39
40  serverWidget::~serverWidget()
41  {
42      delete ui;
43  }
44
45
46  void serverWidget::on_buttonsend_clicked()
47  {
48      if(tcpsocket==nullptr){
49          return ;
50      }
51      //获取编辑区域的内容
52      QString str=ui->textEditWrite->toPlainText();
53
54      //写入通信套接字 协议栈自动发送
55      tcpsocket->write(str.toUtf8().data());
56
57      //在编辑区域显示
58     ui->textEditRead->append("服务器端: "+str);//不用settext 这样会覆盖之前的消息
59  }
```

```
60
61 void serverWidget::on_buttonclose_clicked()
62 {
63     //通信套接字主动与服务端断开连接
64     tcpsocket->disconnectFromHost();//结束聊天
65
66     //关闭 通信套接字
67     tcpsocket->close();
68
69     tcpsocket=nullptr;
70 }
```

6.Qt实现UDP

6.1.客户端

```
1  #include "udpclient.h"
2
3  UDPClient::UDPClient()
4  {
5  //    mType = 0;//Unicast
6  //    mType = 1;//Broadcast
7      mType = 2;//Multicast
8      InitSocket();
9      InitTimer();
10
11 }
12
13 void UDPClient::InitSocket()
14 {
15     mUdpSocket = new QUdpSocket;//初始化socket
16     mGroupAddress.setAddress("239.2.2.222");//设置组播地址
17     mUdpSocket->bind(6666);//绑定端口号
18     if(mType == 2)
19     {
20         //组播的数据的生存期，数据报没跨1个路由就会减1.表示多播数据报只能在同一路由下的局域网内传播
```

```
        mUdpSocket->setSocketOption(QAbstractSocket::MulticastTtlOption,1);
    }
}

void UDPClient::InitTimer()
{
    mTimer = new QTimer;//初始化定时器
    connect(mTimer,&QTimer::timeout,this,[=]
    {
        SendData();
    });
    mTimer->start(1000);//每隔一秒发送一次数据
}

void UDPClient::SendData()
{
    QByteArray _data = "hello";
    if(mType == 0)//单播
    {
        QHostAddress _peerHostAddress = QHostAddress("10.0.0.177");//对位服务器IP
        quint16 _port = 6666;//对位服务器端口
        if(-1 !=mUdpSocket->writeDatagram(_data.data(),_data.size(),_peerHostAddress,_port))
        {
            qDebug()<< "Unicast ==> Send data : "<< _data<<endl;
        }
        mUdpSocket->flush();
    }
    else if(mType == 1)//广播
    {
        quint16 _port = 6666;//广播端口
        if(-1 !=mUdpSocket->writeDatagram(_data.data(),QHostAddress::Broadcast,_port))
        {
            qDebug()<< "Broadcast ==> Send data : "<< _data<<endl;
        }
        mUdpSocket->flush();
    }
    else if(mType == 2)//组播
    {
        quint16 _port = 8888;//组播端口
```

```
        if(-1 != mUdpSocket->writeDatagram(_data.data(),mGroupAddress,_port))
        {
            qDebug()<< "Multicast ==> Send data : "<< _data<<endl;
        }
        mUdpSocket->flush();
    }
    else
    {
        qDebug()<< "mType is error! "<<endl;
        return;
    }
}
```

6.2.服务端

```
#include "udpserver.h"
UDPServer::UDPServer()
{
//    mType = 0;//Unicast
//    mType = 1;//Broadcast
    mType = 2;//Multicast
    InitSocket();

}

void UDPServer::InitSocket()
{
    //初始化socket，设置组播地址
    mUdpSocket = new QUdpSocket;
    mGroupAdress.setAddress("239.2.2.222");
    if(mType == 0 || mType == 1)
    {
        //绑定本地IP和端口号
        mUdpSocket->bind(6666);
    }
    else if(mType == 2)
    {
```

```
        if(mUdpSocket->bind(QHostAddress::AnyIPv4,8888,QUdpSocket::ShareAddress))
        {
            //加入组播地址
            mUdpSocket->joinMulticastGroup(mGroupAdress);
            qDebug()<<("Join Multicast Adrress [")<<mGroupAdress.toString()
                   <<("] Successful!")<<endl;
        }
    }
    else
    {
        qDebug()<< "mType is error! "<<endl;
        return;
    }

    connect(mUdpSocket,&QUdpSocket::readyRead,this,[=]{
        ReadPendingDataframs();
    });
}



void UDPServer::ReadPendingDataframs()
{
    QByteArray _data;
    _data.resize(mUdpSocket->pendingDatagramSize());
    if(mType == 0)//Unicast
    {
        QHostAddress *_peerHostAddress = new QHostAddress("10.0.0.32");
        quint16 _port = 6666;
        while(mUdpSocket->hasPendingDatagrams())
        {
            mUdpSocket->readDatagram(_data.data(),_data.size(),_peerHostAddress,&_port);//接收指定IP和端口的udp报文
            qDebug()<<"Unicast ==> Receive data : "<<QString::fromLatin1(_data)<<endl;
        }
    }
    else if(mType == 1)//Broadcast
    {
        QHostAddress _peerHostAddress;
        quint16 _port;
```

```
60          while(mUdpSocket->hasPendingDatagrams())
61          {
62              mUdpSocket->readDatagram(_data.data(),_data.size(),&_peerHostAddress,&_port);//接收同一子网的udp报文
63              qDebug()<<"Broadcast ==> Receive data : "<<QString::fromLatin1(_data)<<endl;
64          }
65      }
66      else if(mType == 2)//Multicast
67      {
68          QHostAddress _peerHostAddress;
69          quint16 _port;
70          while(mUdpSocket->hasPendingDatagrams())
71          {
72              mUdpSocket->readDatagram(_data.data(),_data.size(),&_peerHostAddress,&_port);//接收同组的udp报文
73              qDebug()<<"Multicast ==> Receive data : "<<QString::fromLatin1(_data)<<endl;
74          }
75      }
76      else
77      {
78          qDebug()<< "mType is error! "<<endl;
79          return;
80      }
81  }
```