计算机网络实验一

学号：2213034

姓名：辛杰

目录

计算机网络实验一

- 目录
- 一、协议设计
  - 1、消息类型
  - 2、语法
  - 3、语义
  - 4、时序
- 二、各模块功能
  - A、客户端功能
    - 1.创建Socket和设置服务器地址
    - 2.初始化网络库
    - 3.判断套接字是否连接到指定的服务器地址
    - 4.创建线程
      - 1）接收消息的线程
      - 2）发送消息的线程
    - 5.关闭线程和套接字，释放网络资源
  - B、服务端功能
    - 1.创建Socket和设置服务器地址
    - 2.初始化网络库
    - 3.绑定套接字
    - 4.开启监听
    - 5. 接受客户端的连接请求
    - 6.创建线程
      - 1）处理客户端连接的线程
      - 2）消息发送的线程
      - 3）消息接收的线程
- 三、程序展示
  - 1、开始界面
  - 2、多人聊天
  - 3、日志输出
  - 4、退出
- 四、实验中遇到的问题
- 五、心得体会

一、协议设计

1、消息类型

> 1. **姓名消息**：用户首次连接时发送的姓名。
> 2. **聊天消息**：用户在聊天室中发送的消息。
> 3. **退出消息**：用户发送的退出指令（`/quit`）。

2、语法

> 1. **姓名消息**：直接发送用户的姓名字符串。
> 2. **聊天消息**：格式为 `"用户名:消息内容"`，其中用户名和消息内容之间用冒号分隔。
> 3. **退出消息**：发送特殊字符串 `"/quit"` 来告知服务器用户想要退出聊天。

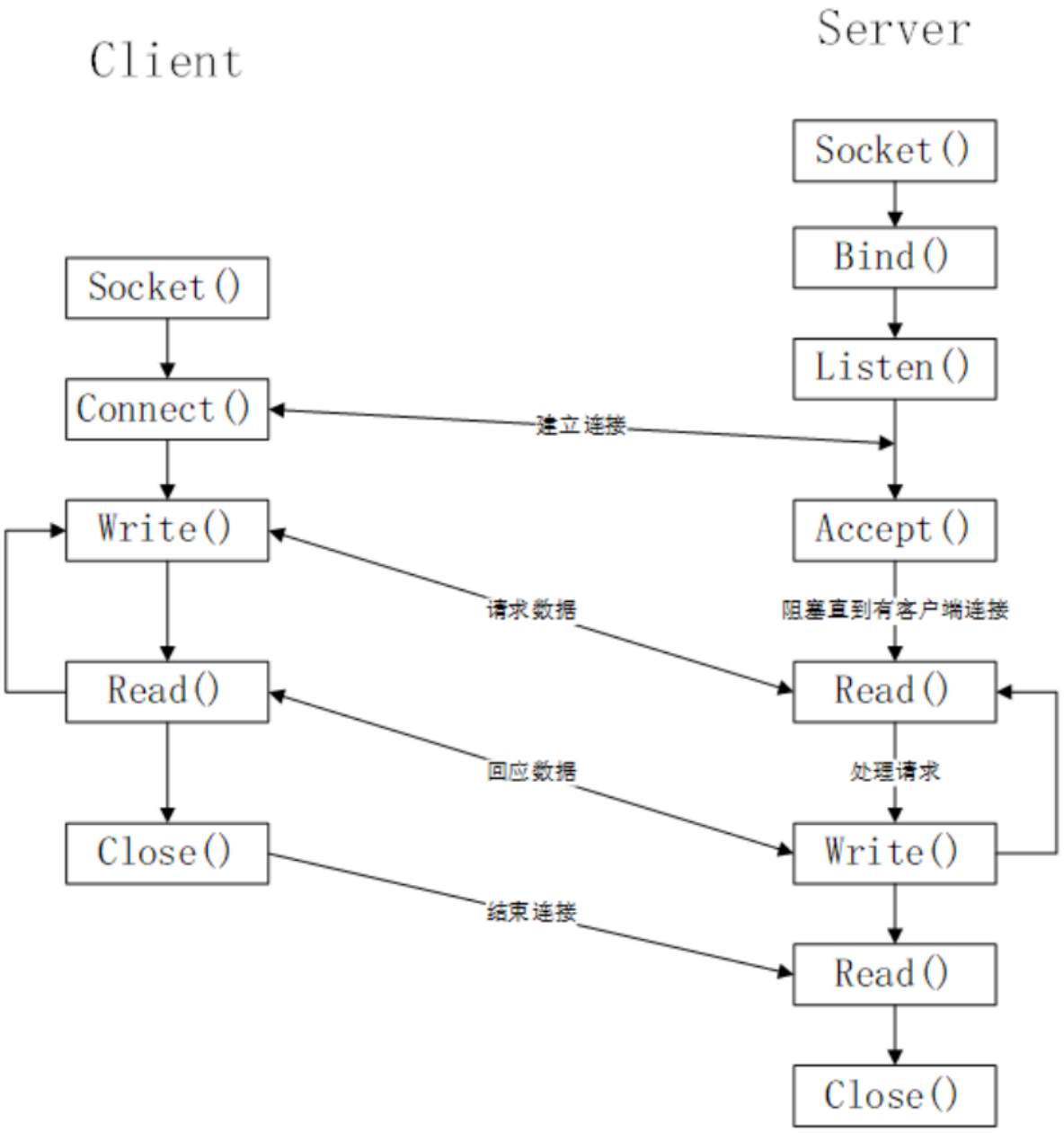
3、语义

> 1. **姓名消息**：告知服务器用户的姓名，以便在聊天室中识别用户。
> 2. **聊天消息**：传递用户想要分享的信息给其他在线用户。
> 3. **退出消息**：通知服务器用户退出，结束聊天会话。

4、时序

> 1. **初始化**：程序首先初始化Winsock，然后连接到服务器。
> 2. **发送姓名**：用户输入姓名后，程序将姓名发送给服务器。
> 3. **接收消息**：程序在一个独立的线程中循环接收服务器发送的消息，并更新聊天列表。
> 4. **发送消息**：用户输入的消息在一个独立的线程中被发送到服务器，同时更新本地聊天列表。
> 5. **退出**：用户输入 `/quit` 后，程序关闭Socket连接，并退出程序。

二、各模块功能



A、客户端功能

1.创建Socket和设置服务器地址

```
    SOCKET s_server;
    SOCKADDR_IN server_addr;

    server_addr.sin_family = AF_INET;
    server_addr.sin_addr.S_un.S_addr = inet_addr("127.0.0.1");
    server_addr.sin_port = htons(5010);
    s_server = socket(AF_INET, SOCK_STREAM, 0);
```

`SOCKADDR_IN`结构体的定义：

```
struct sockaddr_in {
    short sin_family; // 地址族，对于IPv4，通常是AF_INET
    unsigned short sin_port; // 端口号，使用网络字节序
    struct in_addr sin_addr; // IPv4地址，使用网络字节序
    char sin_zero[8]; // 保留，必须填充为0
};
```

socket() 函数：

> - `AF_INET`：指定地址族为 `AF_INET`，即IPv4地址。
> - `SOCK_STREAM`：指定套接字类型为 `SOCK_STREAM`，这代表一个面向连接的、可靠的、基于字节流的通信服务。这种类型的套接字通常用于TCP连接。
> - `0`：在Windows中，这个参数是协议参数，设置为 `0` 表示使用默认协议。对于IPv4和TCP，通常使用 `IPPROTO_TCP` 作为协议参数。但在Windows的Winsock中，`socket()` 函数可以自动推断协议，因此可以设置为 `0`。

2.初始化网络库

```
void initialization() {
    WORD w_req = MAKEWORD(2, 2);
    WSADATA wsadata;
    int err;
    err = WSAStartup(w_req, &wsadata);
    if (err != 0) {
        cout << "Winsock 初始化失败" << endl;
    }
}
```

w_req = MAKEWORD(2, 2)创建了一个表示版本号 `2.2` 的 `WORD` 值

`WSADATA` 是一个结构体，它用于存储有关 Winsock 实现的信息

作用：确保了程序能够使用 Winsock 提供的网络功能

3.判断套接字是否连接到指定的服务器地址

```
    connect(s_server, (SOCKADDR *)&server_addr, sizeof(SOCKADDR));
```

`connect` 函数的原型：

```
int connect(
  SOCKET s,                 // 套接字描述符
  const struct sockaddr *name,  // 服务器地址结构体的指针
  int namelen               // 服务器地址结构体的大小
);
```

如果 `connect` 调用成功，它会返回0；如果失败，它会返回 `SOCKET_ERROR`，并且可以通过调用 `WSAGetLastError` 来获取具体的错误代码。

4.创建线程

```
    HANDLE hThread = CreateThread(NULL, NULL, newMessage, LPVOID(s_server), 0, NULL);
    HANDLE hThread2 = CreateThread(NULL, NULL, sentMessage, LPVOID(s_server), 0, NULL);
```

CreateThread 函数参数

```
HANDLE CreateThread(
  LPCVOID   lpThreadAttributes,
  SIZE_T    dwStackSize,
  LPTHREAD_START_ROUTINE   lpStartAddress,
  LPVOID    lpParameter,
  DWORD     dwCreationFlags,
  LPDWORD   lpThreadId
);
```

> - **lpThreadAttributes**：指向一个安全属性对象，用于控制线程的安全性。如果设置为 `NULL`，则线程继承调用者的安全性。在这个例子中，它被设置为 `NULL`。
> - **dwStackSize**：为新线程指定堆栈大小。如果设置为 `NULL` 或 `0`，则使用默认的堆栈大小。在这个例子中，它被设置为 `NULL`，意味着使用默认的堆栈大小。
> - **lpStartAddress**：指向线程函数的指针，这是新线程执行的函数。在这个例子中，它是 `newMessage` 函数。
> - **lpParameter**：传递给线程函数的参数。在这个例子中，它是 `s_server` 的值，类型转换为 `LPVOID`。这个参数允许将socket的句柄传递给线程函数。
> - **dwCreationFlags**：控制线程的创建和启动行为。如果设置为 `0`，则线程立即运行。在这个例子中，它被设置为 `0`。
> - **lpThreadId**：如果需要，可以接收新线程的标识符。如果设置为 `NULL`，则不返回线程标识符。在这个例子中，它被设置为 `NULL`。

1）接收消息的线程

线程函数newMessage

作用：在一个独立的线程中接收从客户端socket接收到的消息

```
DWORD WINAPI newMessage(LPVOID lparam)
{
    SOCKET ClientSocket = (SOCKET)(LPVOID)lparam;
    while (online)
    {
        Sleep(10);
        char recv_buf[200];
        Sleep(100);
        recv(ClientSocket, recv_buf, 200, 0);
        cout << recv_buf << endl;
        string str(recv_buf);
        messList[mess] = str;
        ++mess;
    }
```

```
    return 0;
}
```

其中recv（）函数原型

作用：从连接的远程socket接收传入的数据，并将其存储在指定的缓冲区内

```
int recv(
    SOCKET s,           // 套接字描述符
    char* buf,          // 指向接收缓冲区的指针
    int len,            // 缓冲区的长度
    int flags           // 控制操作的可选标志
);
```

- `ClientSocket`：这是一个 `SOCKET` 类型的参数，它是一个标识符，用来指定从哪个 socket 接收数据。这个 socket 必须已经通过 `connect` 或 `accept` 函数与远程 socket 建立了连接。
- `recv_buf`：这是一个字符数组，用来存储从 socket 接收到的数据。
- `200`：这个参数指定了 `recv_buf` 缓冲区的最大长度，也就是你想要接收的最大字节数。实际接收到的数据量可能小于这个值，这取决于 socket 上可用的数据量。
- `0`：这是 `flags` 参数，用来修改 `recv` 函数的行为。传递 `0` 表示使用默认行为，即普通的接收操作，没有特殊的标志。。

2）发送消息的线程

线程函数sentMessage

作用：从控制台读取用户输入，然后将输入的消息发送到服务器

```
DWORD WINAPI sentMessage(LPVOID lparam)
{
    SOCKET ClientSocket = (SOCKET)(LPVOID)lparam;
    bool first = true;
    while (online)
    {
        if (first)
        {
            cout << "输入你的姓名";
            first = false;
        }
        char sent_buf[100];
        cin.getline(sent_buf,100);
        clsScreen();
        strcpy_s(name, sent_buf);
        string o(sent_buf);
        if (o == quit && !first)
            exit(0);
        send(ClientSocket, sent_buf, 100, 0);
    }
    online = false;
    return 0;
}
```

其中 clsScreen()函数

作用：在控制台应用程序中清除屏幕，并重新打印之前接收到的所有消息

```
void clsScreen()
{
    Sleep(100);
    system("cls");
    for (int i = 0; i < mess; ++i)
        cout << messList[i].c_str() << endl;
}
```

send()函数原型

作用：通过指定的套接字 `s` 发送数据，并将数据发送到与该套接字连接的远程地址

```
int send(
  SOCKET s,           // 套接字描述符
  const char* buf,    // 指向要发送数据缓冲区的指针
  int len,            // 缓冲区的长度
  int flags           // 指定发送操作的标志
);
```

5.关闭线程和套接字，释放网络资源

```
    WaitForSingleObject(hThread, INFINITE);
    WaitForSingleObject(hThread2, INFINITE);
    CloseHandle(hThread);
    CloseHandle(hThread2);
    closesocket(s_server);
    WSACleanup();
```

B、服务端功能

1.创建Socket和设置服务器地址

2.初始化网络库

3.绑定套接字

```
    bind(s_server, (SOCKADDR *)&server_addr, sizeof(SOCKADDR));
```

bind函数原型

作用：将套接字 `s_server` 与 `server_addr` 指定的地址和端口号关联起来

```
int bind(
        SOCKET s,
        const struct sockaddr *name,
        int namelen
);
```

4.开启监听

```
    listen(s_server, SOMAXCONN);
```

listen函数原型：

```
int listen(
    SOCKET s,
    int backlog //套接字排队的最大挂起连接的数量
);
```

`SOMAXCONN` 是一个常用的宏，定义在头文件 `<winsock2.h>` 中，它表示建议的最大挂起连接数

5. 接受客户端的连接请求

```
        SOCKET sockConn = accept(s_server, (SOCKADDR*)&accept_addr, &len);
```

accept函数

作用：会等待客户端的连接请求。当一个连接请求到达时，`accept` 函数会被调用来接受这个连接，并创建一个新的套接字用于与客户端通信。

6.创建线程

1）处理客户端连接的线程

```
HANDLE hThread = CreateThread(NULL, NULL, newConnect, LPVOID(sockConn), 0, NULL)

DWORD WINAPI newConnect(LPVOID lparam)
{
    SOCKET ClientSocket = (SOCKET)(LPVOID)lparam;
    HANDLE hThread = CreateThread(NULL, NULL, newMessage, LPVOID(ClientSocket), 0, NULL);
    HANDLE hThread2 = CreateThread(NULL, NULL, acceptMessage, LPVOID(ClientSocket), 0, NULL);
    WaitForSingleObject(hThread, INFINITE);
    WaitForSingleObject(hThread2, INFINITE);
    CloseHandle(hThread);
    CloseHandle(hThread2);
    return 0;
}
```

其中，又创建两个新线程，`hThread` 和 `hThread2`，分别用于处理客户端的消息发送（`newMessage`）和消息接收（`acceptMessage`）。

2）消息发送的线程

其中newMessage

```
DWORD WINAPI newMessage(LPVOID lparam)//用于检查消息队列是否有新的消息，发送给用户
{
    SOCKET ClientSocket = (SOCKET)(LPVOID)lparam;
    int fsn = 0;
```

```
    string name("unname");
    while (1)
    {
        Sleep(10);
        if (socketToname.find(ClientSocket) != socketToname.end())
            name = socketToname.find(ClientSocket)->second;
        if (fsn != messageNum)
        {
            for (; fsn< messageNum; ++fsn)
            {
                if (strcmp(M[fsn].from, name.c_str()) == 0)
                    continue;
                char sen[200];
                char cn[10] = ":";
                strcpy_s(sen, M[fsn].from);
                strcat_s(sen, cn);
                strcat_s(sen, M[fsn].message);
                send(ClientSocket, sen, 200, 0);
            }
        }
    }
}
```

3）消息接收的线程

acceptMessage

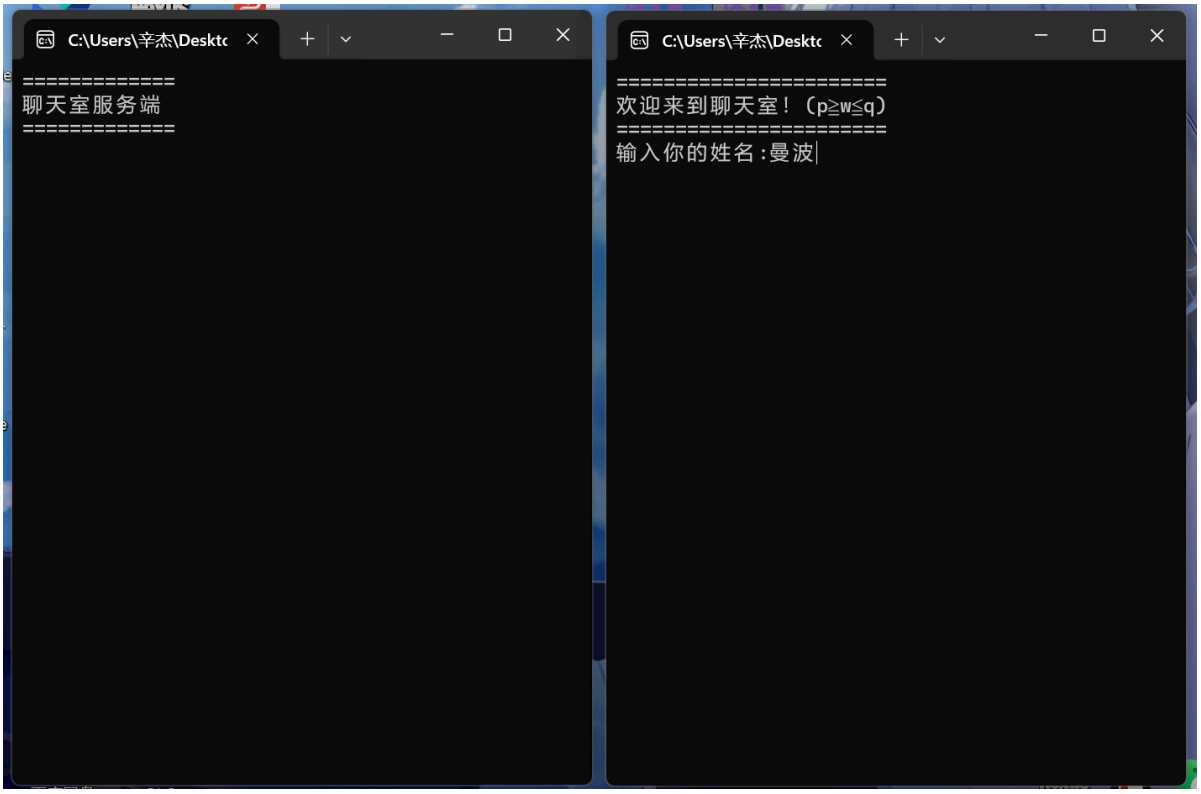
```
DWORD WINAPI acceptMessage(LPVOID lparam)//用于检查该用户是否发来了新的消息，添加到消息队列
{
    SOCKET ClientSocket = (SOCKET)(LPVOID)lparam;
    bool first=true;
    char name[100]="unname";
    while (1)
    {
        Sleep(10);
        char recv_buf[100];
        int ret =recv(ClientSocket, recv_buf, 100, 0);
        if (first)
        {
            strcpy_s(name, recv_buf);
            first = false;
            string str(name);
            socketToname.insert(pair<SOCKET, string>(ClientSocket, str));
            cout << recv_buf << " 进入聊天室" << endl << endl;
            continue;
        }
        else
        {
            if (ret == SOCKET_ERROR || ret == 0)
            {
                cout << name << " 退出了聊天室" << endl << endl;
                break;
            }
            cout << name << ":" << recv_buf << endl<<endl;
```

```
            strcpy_s(M[messageNum].message, recv_buf);
            strcpy_s(M[messageNum].from, name);
            messageNum++;
        }
    }
    return 0;
}
```

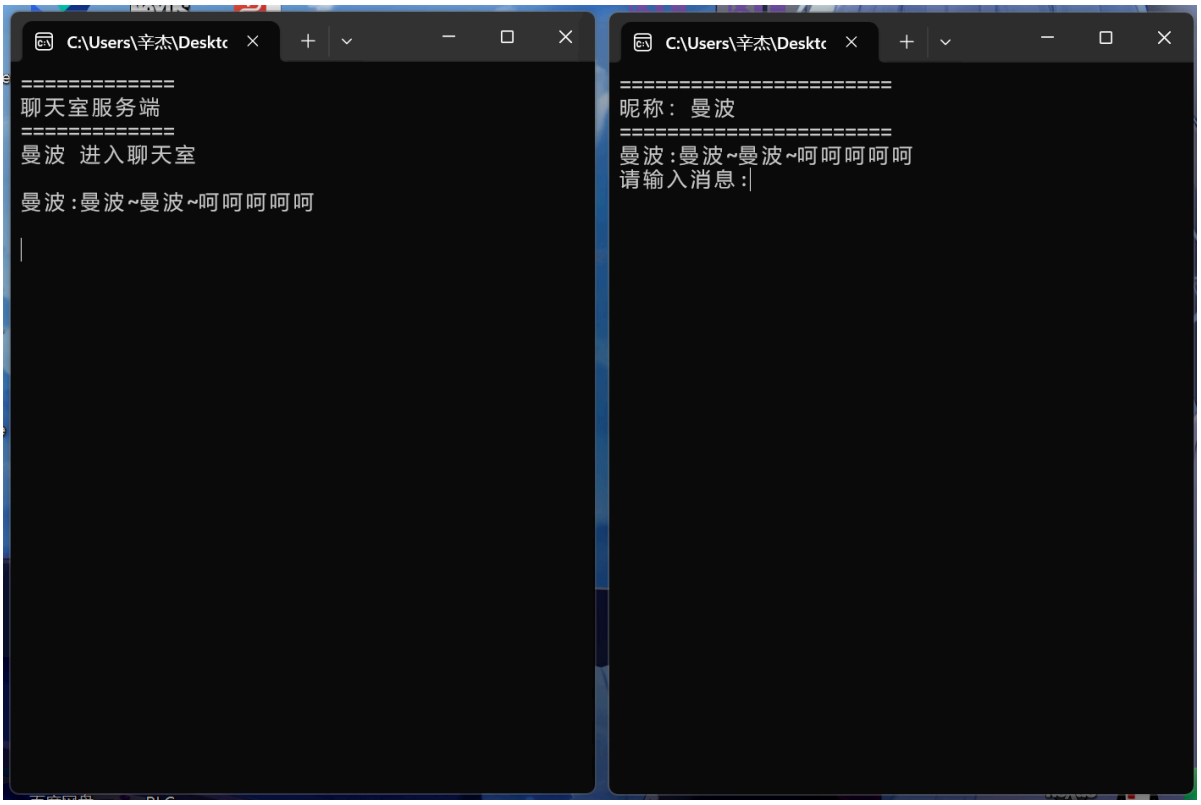
三、程序展示

1、开始界面

先打开服务端，再打开客户端，直接会连上，客户端输入名字

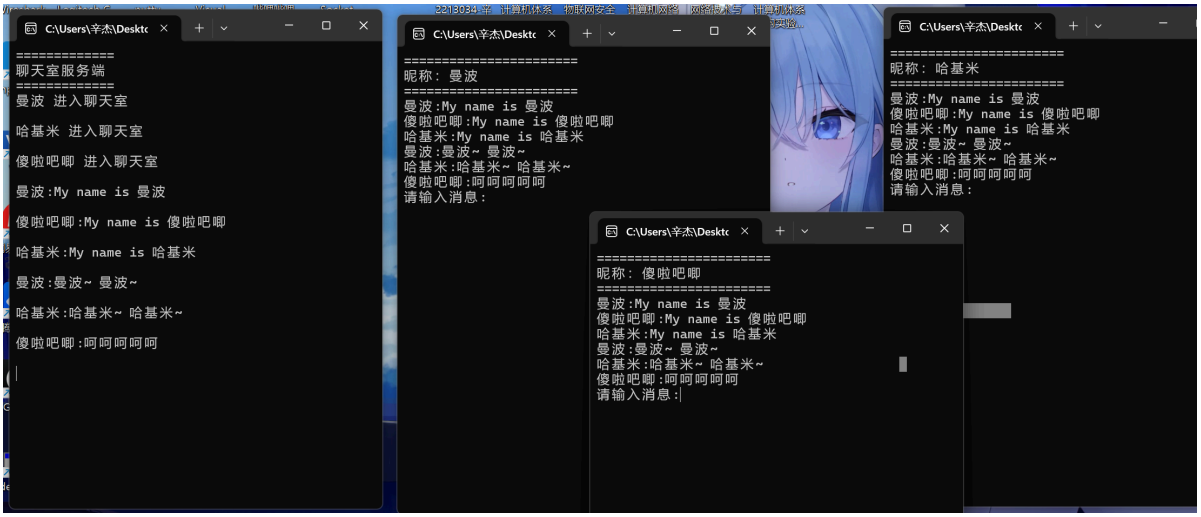


即可输入消息



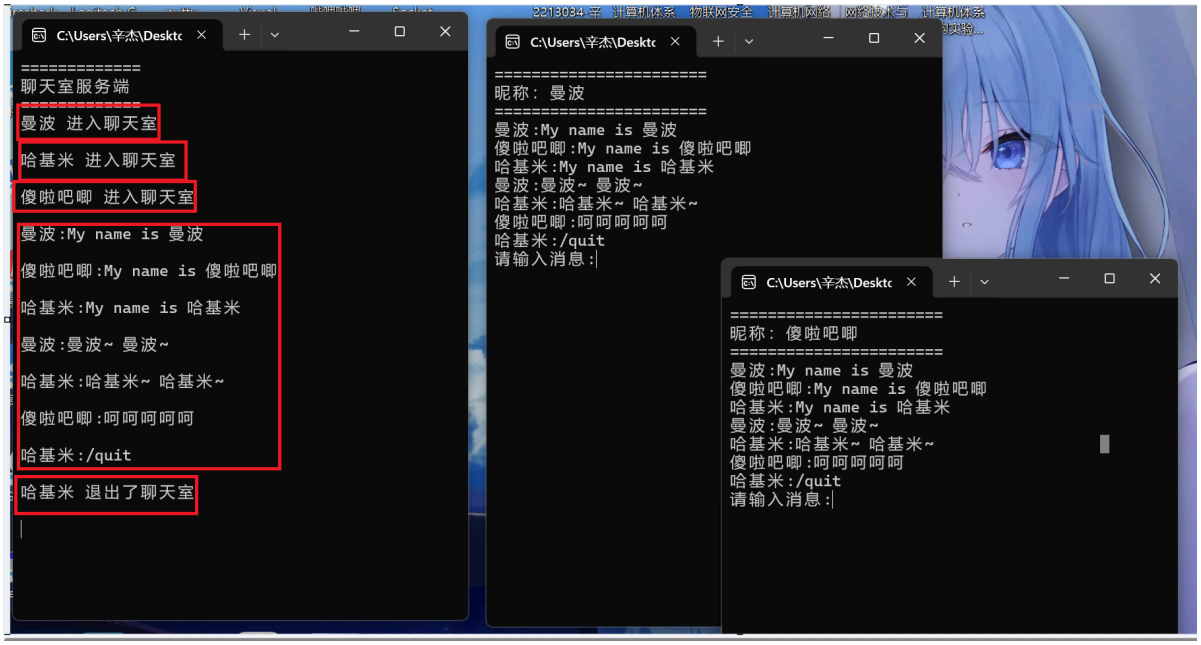
2、多人聊天

以三个人聊天为例子，如图所示



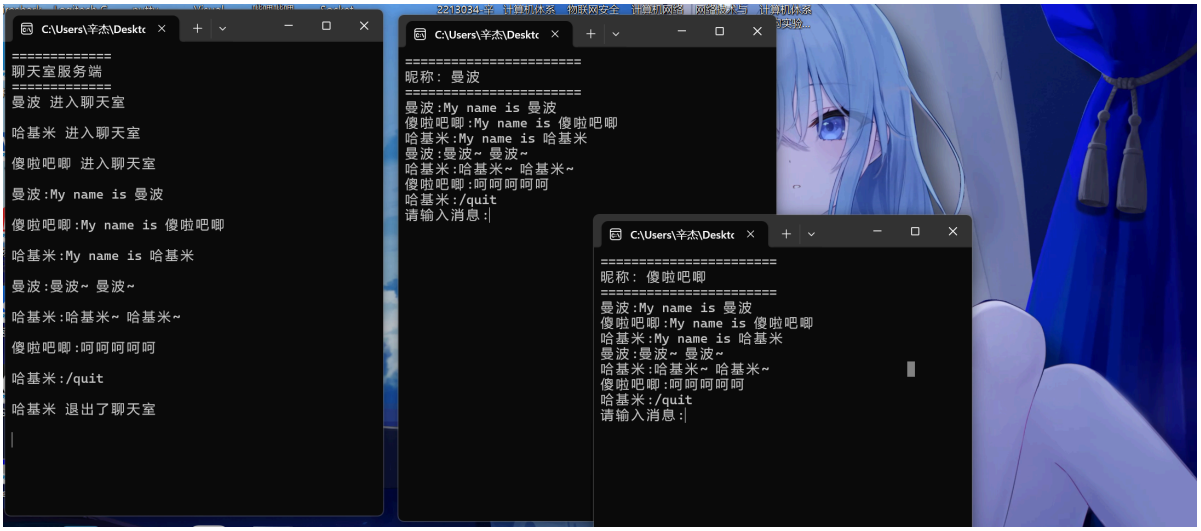
3、日志输出

服务端会记录客户端的进出和消息记录



4、退出

输入/quit会关闭程序且退出聊天室



四、实验中遇到的问题

在编写基于Socket的聊天程序时，我遇到了处理用户输入和显示消息的同步问题。由于程序需要同时监听网络事件和用户输入，我最初使用了一个线程来处理网络通信，另一个线程来处理用户输入。然而，当用户输入消息并发送给其他客户端时，消息的显示和网络通信之间出现了同步问题，导致消息有时会显示延迟或乱序。为了解决这个问题，我引入了一个线程安全的队列来管理消息的发送和接收，确保消息按照正确的顺序显示。

五、心得体会

通过实现基于Socket的聊天程序，我深入理解了网络通信原理，掌握了多线程编程技巧，并提高了问题解决能力。