计算机网络lab1

利用流式套接字编写聊天程序

学院：网络空间安全学院

专业：物联网工程

学号：2213523

姓名：姜卓含

一、实验目的

(1) 设计聊天协议，并给出聊天协议的完整说明。

(2) 利用C或C++语言，使用基本的Socket函数进行程序编写，不允许使用CSocket等封装后的类。

(3) 程序应有基本的对话界面，但可以不是图形界面。程序应有正常的退出方式。

(4) 完成的程序应能支持英文和中文聊天。

(5) 采用多线程，支持多人聊天。

(6) 编写的程序应结构清晰，具有较好的可读性。

(7) 在实验中观察是否有数据包的丢失，提交程序源码、可执行代码和实验报告。

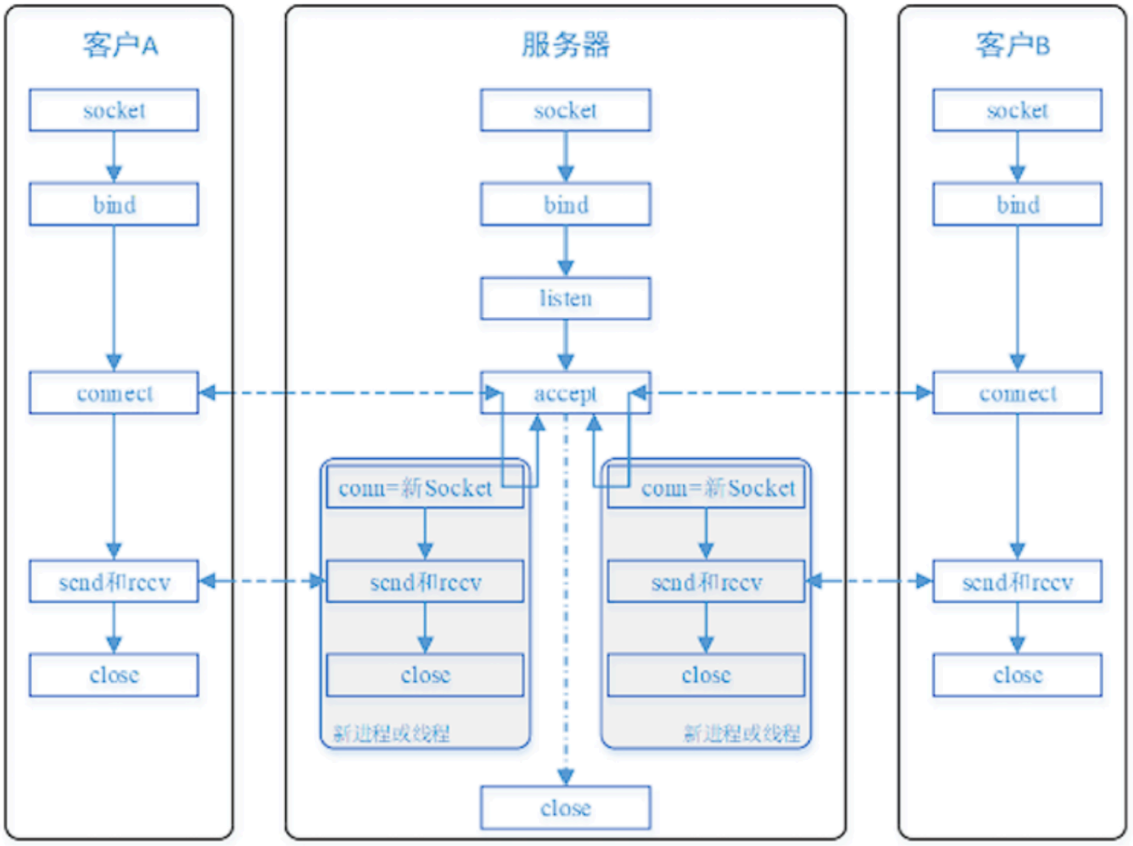
二、实验原理

本实验设计并实现了一个基于 Client/Server 架构的多人聊天室系统。系统利用 Windows Sockets API 即 Winsock 进行底层网络通信，并采用多线程技术来支持多人聊天。

2.1 总体逻辑

选择 C++ 语言，并直接使用 Winsock 2.0 提供的 `socket`, `bind`, `listen`, `accept`, `connect`, `send`, `recv` 等基本 Socket 函数。

使用 TCP 流式套接字，可以确保所有消息可靠、按序到达。



采用多线程设计，服务器能并发处理多个客户端，客户端能同时发送和接收消息。

自定义设计一套完整的聊天协议，包含时间标签。支持多用户注册并带查重、公聊、私聊、展示在线列表和正常退出。

2.2 多线程框架设计

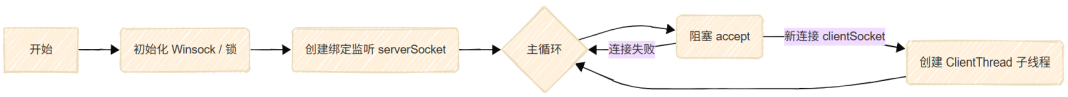
为了实现多人聊天，服务器端要同时处理多个客户端，客户端要同时收发消息，本在服务器和客户端均采用了多线程模型。

2.2.1 服务器端

服务器端采用每客户端一线程模型，由一个主线程和N个子线程构成：

1.**主线程**

- **核心:** `main()` 函数。
- **功能:** 负责服务器的初始化（`WSAStartup`，`socket`，`bind`，`listen`）。
- **流程:** 在一个 `while(true)` 循环中阻塞在 `accept()`。每当有新客户端连接，立即 `CreateThread()` 创建一个客户端服务的子线程，并将新 `clientSocket` 的控制权移交，自己则立刻返回继续 `accept()`。



2.**子线程**

- **核心:** `ClientThread()` 函数。
- **功能:** 一对一服务一个客户端 `clientSocket` 的完整生命周期。

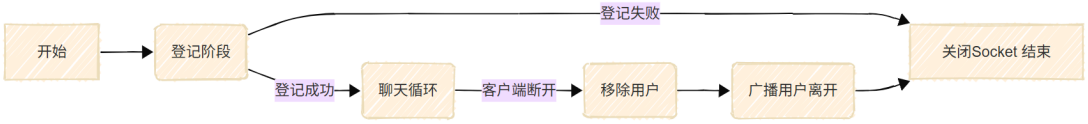
- **流程:**

  (1) **登记:** 阻塞 `recv()` 等待 `JOIN:` 消息，用户名需要唯一。

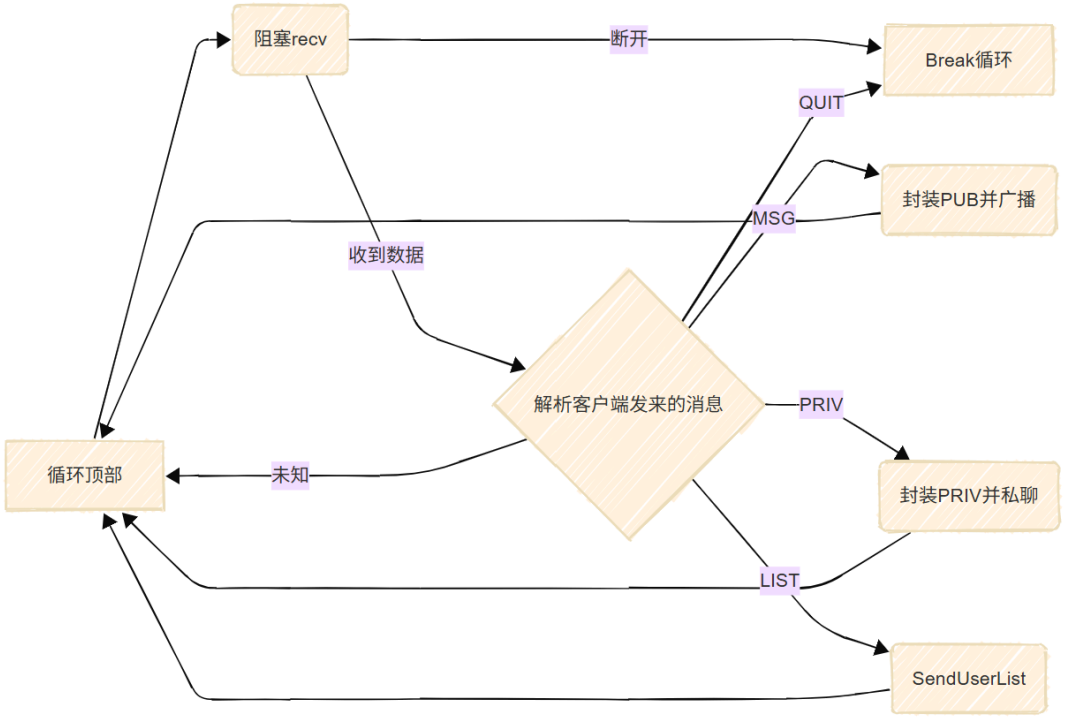
  (2) **解析:** 解析协议 (`MSG:`, `PRIV:`, `LIST:`, `QUIT:`)。

  (3) **转发:** 调用消息处理模块，将消息广播或私发。

  (4) **结束:** 当客户端退出时，将其从 `g_clients` 中移除，并 `closesocket()`。



聊天循环的流程图：



3.**共享数据模块**

- **核心:** `std::map<SOCKET, std.string> g_clients` 和 `CRITICAL_SECTION g_cs`。
- **功能:** 采用 map 存储所有在线用户的 `Socket` 和 `username`。
- **维护线程安全:** 通过锁维护线程安全，任何对 `g_clients` 的并发读写操作都必须在 `EnterCriticalSection` 和 `LeaveCriticalSection` 之间完成，保护临界区防止出现问题。

4.**消息和工具函数**

- **核心:** `BroadcastMessage()`, `SendPrivateMessage()`, `SendUserList()`, `getCurrentTimestamp()`。
- **功能:** 广播、私聊、获取用户列表和获得当前时间的函数。

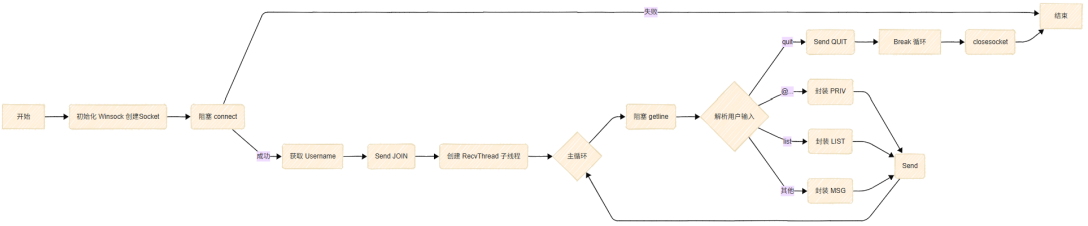
2.2.2 客户端

相比服务器端，客户端的实现比较简单，客户端采用双线程模型，主线程负责处理客户端的输入和发送，子线程负责接收和显示消息：

1.**主线程**

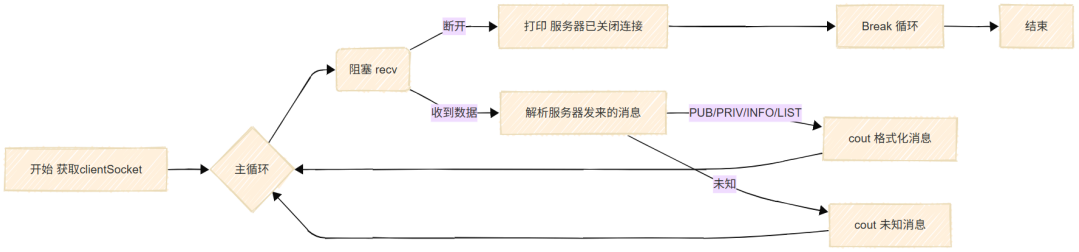
- **核心:** `main()` 函数。
- **功能:** 作为打字员，负责处理所有的键盘输入和发送。
- **流程:** 在 `connect()` 和 `JOIN:` 成功后，`CreateThread()` 启动接收模块。主线程进入 `while(true)` 循环，阻塞在 `getline(cin, inputLine)`，等待用户输入，然后将其封装为协议并 `send()` 出去。

  主线程用户输入的流程图：



2.**子线程**

- **核心:** `RecvThread()` 函数。
- **功能:** 作为接线员，负责所有的消息接收和显示。
- **流程:** 在 `while(true)` 循环中，阻塞在 `recv()`，等待服务器发送消息。一旦收到，它负责解析协议并输出到屏幕上。



2.3 自定义聊天协议

传输层实验本系统设计了一套基于ASCII文本的、以冒号 (`:`) 为核心分隔符的自定义聊天协议，所有消息均以 `\0` 字符串结束符作为消息边界，兼容 UTF-8/GBK 中文，可以实现中英文聊天。

2.3.1 消息类型和语法

1.**客户端->服务器消息**

客户端会将用户输入的消息封装成下面的格式并发给服务器：

消息类型	语法	功能说明
注册	`JOIN:<Username>`	时序：必须是第一条消息，客户端请求以 `<Username>` 身份登记。
公聊	`MSG:<Message>`	客户端请求向所有在线用户 `<Message>`。
私聊	`PRIV:<TargetUser>:<Message>`	客户端请求向 `<TargetUser>` 单独发送 `<Message>`。
列表	`LIST:`	客户端请求获取当前所有在线用户的列表。
退出	`QUIT:`	客户端主动通知服务器自己即将下线。

2.**服务器->客户端消息**

每接到一个客户端的新连接，服务器就会为客户端创建一个专属的子线程，负责将服务器的消息封装成下面的格式并发给客户端：

消息类型	语法	功能说明
公聊	`PUB:<Timestamp>[FromUser]<Message>`	(含时间标签) 服务器转发的公共聊天消息。
私聊	`PRIV:<Timestamp>[FromUser]<Message>`	(含时间标签) 服务器转发的私人聊天消息。
通知	`INFO:<InfoMessage>`	服务器向客户端发送的系统消息。
列表	`LIST:<User1>,<User2>,...`	服务器响应 `LIST:` 请求，返回以逗号分隔的在线用户列表。

2.3.2 消息语义

定义接收方如何相应特定消息。

1.**服务器端 `ClientThread`**

- 当收到 `JOIN:<Username>`：

  客户端请求登记。

  先上锁 `EnterCriticalSection`，之后查重遍历 `g_clients`，检查 `<Username>` 是否已存在。如果重复，必须 `LeaveCriticalSection`，然后 `send("INFO:用户名已被占用...")`，最后 `closesocket()` 并终止此线程。如果不重复，则将将 `(Socket, Username)` 存入 `map`，之后 `LeaveCriticalSection`，最后广播消息并单独欢迎用户。

- 在用户加入后进入聊天部分，当收到 `MSG:<Message>`：

  客户端请求公聊。

  解析公聊，并调用 `getCurrentTimestamp()` 获取 `[HH:MM:SS]` 格式的字符串。将消息封装成服务器传给客户端的消息语法，并调用 `BroadcastMessage(fullMsg, clientSocket)` 发送给除客户端自己外的所有人。

- 当收到 `PRIV:<TargetUser>:<Message>`：

客户端请求私聊。

解析私聊，调用 `getCurrentTimestamp()` 获取 `[HH:MM:SS]` 格式的字符串，将消息封装成服务器传给客户端的消息语法，调用 `SendPrivateMessage(fullMsg, targetUser, username)` 发送给私聊的对象。

- 当收到 `QUIT:`：

  客户端主动下线。

  在服务器端打印出客户单主动退出的信息，之后先上锁，将客户端从 `map` 中移除 `g_clients.erase(clientSocket)`，之后解锁，广播客户端离开聊天室，最后调用 `closesocket` 终止线程。

- 当收到 `LIST:`

  客户端请求用户列表。

  调用 `SendUserList(clientSocket)` 打印出在线用户列表发给客户端。

2.**客户端 `RecvThread`**

客户端解析服务器发来的信息并打印出来，相比服务器来说很简单：

- 当收到 `PUB:<...>`：

  收到公聊消息， `cout << " (公聊) " << command.substr(4)`，打印协议头后的内容。

- 当收到 `PRIV:<...>`：

  收到一条私聊消息。

- 当收到 `INFO:<...>`：

  收到一条系统通知（欢迎、加入、离开、或错误信息）。

- 当收到 `LIST:<...>`：

  收到服务器响应的在线用户列表。

2.3.3 时间标签

聊天信息带有时间标签，在服务器端生成，服务器在 `ClientThread` 中，每当收到 `MSG:` 或 `PRIV:` 请求，先调用 `getCurrentTimestamp()`。并将返回的 `[HH:MM:SS]` 字符串封装进 `PUB:` 和 `PRIV:` 的语法中。客户端的 `RecvThread` 通过 `substr()` 原样打印，从而实现了时间标签的显示。

三、实验具体内容

核心功能实现，整体代码较长，下面有省略。

3.1 server_1.cpp

使用 `std::map` 存储全局用户列表，并使用 `CRITICAL_SECTION` 锁来保护它，防止多线程并发访问时崩溃。

```
//全局变量
std::map<SOCKET, std::string> g_clients;//采用map存储所有连接的客户端
CRITICAL_SECTION g_cs; //锁 保护临界区
```

`getCurrentTimestamp()` 函数用于获取当前时间并格式化，在后面公聊和私聊部分用到。

`BroadcastMessage()` 函数用于广播消息，即公聊，遍历所有用户，将消息发送给除自身外的所有用户，注意使用锁保护临界区。

```
//广播函数
void BroadcastMessage(const std::string& message, SOCKET skipSocket)
{
    EnterCriticalSection(&g_cs);//加锁
    for (auto& pair : g_clients)//遍历 pair.first是SOCKET pair.second是用户名
    {
        if (pair.first != skipSocket)//不发给发送者自身
        {
            send(pair.first, message.c_str(), message.length() + 1, 0);
        }
    }
    LeaveCriticalSection(&g_cs); //解锁
}
```

`SendPrivateMessage()`函数用于私聊，使用锁保护临界区，遍历所有用户，如果找到私聊对象则发送，否则输出私聊失败找不到用户，并发送`INFO:`格式的通知发给发送者，找不到用户私聊失败。

```
//私聊函数
void SendPrivateMessage(const std::string& message, const std::string&
targetUser, const std::string& fromUser)
{
    EnterCriticalSection(&g_cs); //加锁

    SOCKET targetSocket = INVALID_SOCKET;
    for (auto& pair : g_clients)
    {
        if (pair.second == targetUser)
        {
            targetSocket = pair.first;
            break;
        }
    }
    if (targetSocket != INVALID_SOCKET)//如果找到则发送
    {
        send(targetSocket, message.c_str(), message.length() + 1, 0);
    }
    else
    {
        cout << "  (私聊失败: " << fromUser << " 找不到 " << targetUser << ")"
<< endl;
        SOCKET fromSocket = INVALID_SOCKET;
        for (auto& pair : g_clients)//找发送者
        {
            if (pair.second == fromUser)
            {
                fromSocket = pair.first;
                break;
            }
        }
        if (fromSocket != INVALID_SOCKET)
        {
            std::string failMsg = "INFO:私聊失败, 找不到用户: " + targetUser;
            send(fromSocket, failMsg.c_str(), failMsg.length() + 1, 0);//
把"找不到用户"的消息发回给发送者
```

```
        }
    }
    LeaveCriticalSection(&g_cs); //解锁
}
```

`SendUserList()` 函数用于获取用户列表，使用锁保护临界区，遍历map将所有用户名(pair.second)拼起来，并发送给请求者。

```
//获取用户列表函数
void SendUserList(SOCKET clientSocket)
{
    EnterCriticalSection(&g_cs); //加锁
    std::string userList = "LIST:";
    for (auto& pair : g_clients)//遍历map，把所有用户名(pair.second)拼起来
    {
        userList += pair.second + ",";
    }
    if (!g_clients.empty())
    {
        userList.pop_back(); //去掉最后一个多余的','
    }
    LeaveCriticalSection(&g_cs); //解锁
    send(clientSocket, userList.c_str(), userList.length() + 1, 0);//发送给请求者
}
```

`ClientThread` 子线程，每个客户端一个子线程，用于解析客户端发来的消息以及将消息封装好后发给客户端：

- 先进行登记，线程启动后，立刻 `recv()`，等待客户端发送的第一条消息应，收到以 `JOIN:` 开头的消息，若格式不正确则会终止线程，如果格式正确，会用 `username = recvBuf + 5` 提取出用户名。

  ```
  DWORD WINAPI ClientThread(LPVOID lpParameter)
  {
      SOCKET clientSocket = (SOCKET)lpParameter;
      std::string username; //线程对应的用户名
      char recvBuf[1024];//客户端发送的第一条消息
      int recvResult = recv(clientSocket, recvBuf, sizeof(recvBuf), 0);
      ...
  ```

- 用户名查重以及添加用户，使用锁保护临界区，遍历当前所有在线用户检查用户名是否已存在。如果发现重复则解锁，并 `send("INFO:用户名已被占用...")` 通知客户端失败并 `closesocket()` 终止这个线程，如果不重复则将新用户加入列表，之后解锁，并广播新用户加入以及向新客户端发送欢迎加入的信息。

```
...
    else
    {
        g_clients[clientSocket] = username; //将(Socket, Username)存入map
        LeaveCriticalSection(&g_cs); //解锁
    }
    cout << ">>> " << username << " (Socket: " << clientSocket << ") 加入了聊天室。" << endl;
    std::string joinMsg = "INFO:" + username + " 加入了聊天室。";
    BroadcastMessage(joinMsg, clientSocket); //广播给别人"xxx 加入"的消息
    send(clientSocket, "INFO:欢迎加入！\0", 18, 0); //单独欢迎自己
```

- 聊天循环，首先清空缓冲区并阻塞在 `recv()`，等待客户端发送新消息，共有`MSG:`，`PRIV:`，`LIST:`，`QUIT:`四种类型，判断消息类型并进行封装，调用前面的函数进行输出和发送。
- 循环结束后进行退出工作，先上锁，从全局列表中移除该用户，之后解锁，并广播用户离开的消息，关闭这个 `socket`，线程结束。

```
  do//聊天部分
    {
        memset(recvBuf, 0, sizeof(recvBuf)); //清空缓冲区
        recvResult = recv(clientSocket, recvBuf, sizeof(recvBuf), 0);
...
...
    EnterCriticalSection(&g_cs);
    g_clients.erase(clientSocket); //从map中移除
    LeaveCriticalSection(&g_cs);

    //广播xxx离开
    std::string leftMsg = "INFO:" + username + " 离开了聊天室。";
    BroadcastMessage(leftMsg, clientSocket); //广播给剩下的人

    closesocket(clientSocket);
    cout << ">>> 线程 " << clientSocket << " 服务结束。" << endl;
    return 0;
}
```

`main`函数，主线程，用于设置服务器和分发客户端

- 初始化锁，启动和加载 Windows 的网络库

```
InitializeCriticalSection(&g_cs);//初始化锁
WSADATA wsadata;
WSAStartup(MAKEWORD(2, 2), &wsadata);
```

- 创建socket，使用IPv4 TCP流式套接字

```
SOCKET serverSocket = socket(AF_INET, SOCK_STREAM, 0);//IPv4 TCP流式套接字
```

- 准备地址，选择设置要监听的端口号为12345，并转换成网络字节序，采用 `INADDR_ANY` 监听本机所有ip

```
SOCKADDR_IN serverAddr;
serverAddr.sin_family = AF_INET;
serverAddr.sin_port = htons(12345);//转换为网络字节序
serverAddr.sin_addr.S_un.S_addr = INADDR_ANY;//监听本机所有ip
```

- bind绑定，将总机`serverSocket`和`serverAddr`端口号绑定

```
bind(serverSocket, (SOCKADDR*)&serverAddr, sizeof(serverAddr));
```

- listen监听，使`serverSocket`正式进入监听状态，最多允许 5 个新连接排队等待

```
listen(serverSocket, 5);//等待队列长度5
```

- 进入循环，阻塞等待`accept`，当有客户端`connect()`时接触阻塞，主线程接到新连接，并创建和这个客户端一对一的子线程，再次回到循环开始等待下个客户端连接。

```
while (true)
{
    SOCKET clientSocket = accept(serverSocket, NULL, NULL);
    if (clientSocket == INVALID_SOCKET)
    {
        cout << "Accept失败: " << WSAGetLastError() << endl;
        continue;
    }
    cout << "主线程: 接到一个新连接!  ID: " << clientSocket << endl;
    CreateThread(NULL, 0, ClientThread, (LPVOID)clientSocket, 0, NULL);

}
```

- 最后关闭socket并释放锁，不过因为服务器会一直阻塞在前面，这里的代码不会执行。

3.2 client_1.cpp

`RecvThread`子线程，循环接收并解析服务器发来的消息，阻塞在`recv()`等待服务器的消息，解析消息，共有`PUB:`，`PRIV:`，`INFO:`，`LIST:`四种类型，剥掉协议头并打印出来。

```
DWORD WINAPI RecvThread(LPVOID lpParameter)//客户端的接收消息线程
{
    SOCKET clientSocket = (SOCKET)lpParameter;//强制转换为SOCKET类型
    char recvBuf[1024];//缓冲区 保存消息
    int recvResult;//收到消息字节数

    while (true)
    {
        recvResult = recv(clientSocket, recvBuf, sizeof(recvBuf), 0);//recv阻塞函数
        if (recvResult > 0)//成功收到消息
        {
            recvBuf[recvResult] = '\0';//加一个\0表示字符串结束
            std::string command(recvBuf);//拷贝到string字符串对象
            //解析服务器发来的协议
            if (command.find("PUB:") == 0)//如果command是PUB开头 是公聊
            {
```

```
                cout << "  (公聊) " << command.substr(4) << endl;//从第4个索引位置开始输出
            }
...
...
        }
        else if (recvResult == 0)
        {
            cout << ">>> 服务器已关闭连接。" << endl;
            break;
        }
        else
        {
            cout << ">>> 接收失败，连接中断: " << WSAGetLastError() << endl;
            break;
        }
    }
    return 0;
}
```

`main` 主线程，负责启动、链接、登录与发送。

- 和server一样先初始化网络库并创建socket，阻塞在`connect()`连接服务器

```
if (connect(clientSocket, (SOCKADDR*)&serverAddr, sizeof(serverAddr)) == SOCKET_ERROR)
{
    cout << "连接失败: " << WSAGetLastError() << endl;
    closesocket(clientSocket);
    WSACleanup();
    return 1;
}
cout << "连接服务器成功，请开始聊天" << endl;
```

- 输入用户名，并封装好发送给服务器完成登记

```
std::string username;
cout << "请输入您的用户名: ";
getline(cin, username);

std::string joinMsg = "JOIN:" + username;
send(clientSocket, joinMsg.c_str(), joinMsg.length() + 1, 0);
```

- 创建接收消息的线程，接收消息的线程和发送消息的线程同时运行

```
CreateThread(NULL, 0, RecvThread, (LPVOID)clientSocket, 0, NULL);//接收消息的线程
```

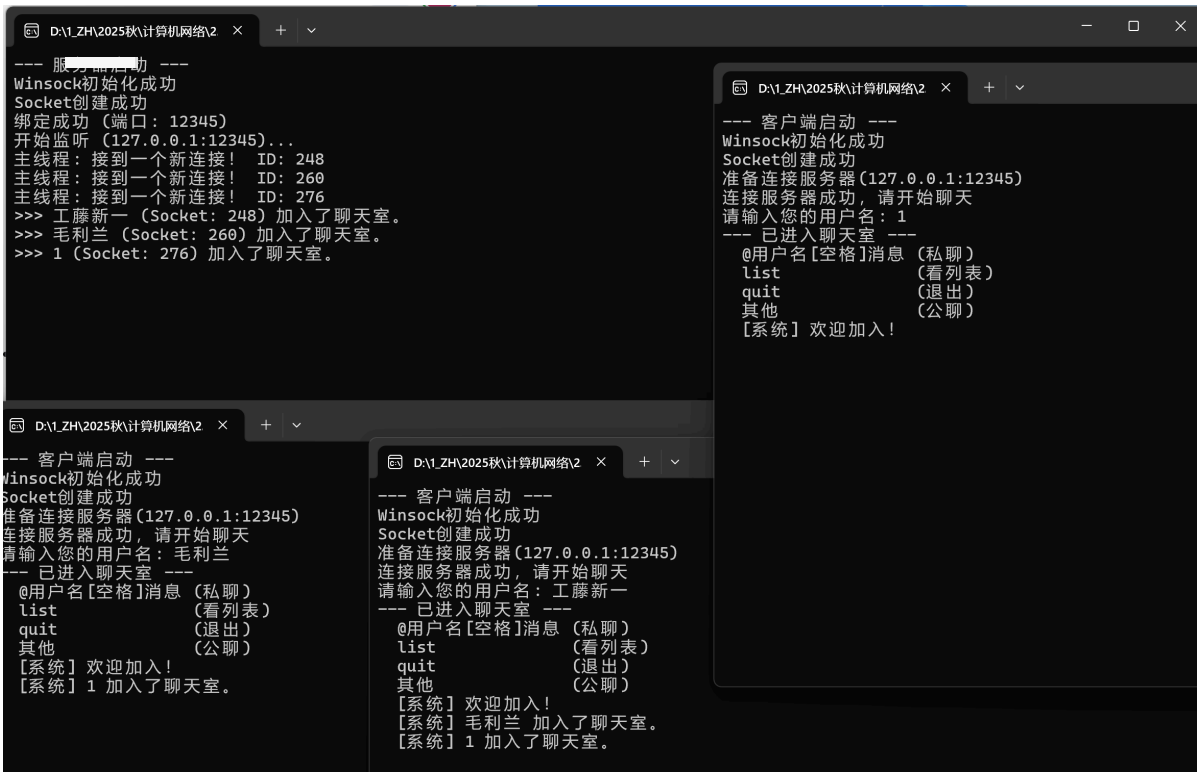
- 发送消息，等待用户输入并判断类型，封装好发给服务器，私聊需要判断格式是否正确。
- 当用户输入quit，退出循环并`closesocket(clientSocket)`

```
std::string inputLine;//用户输入
while (true)
```

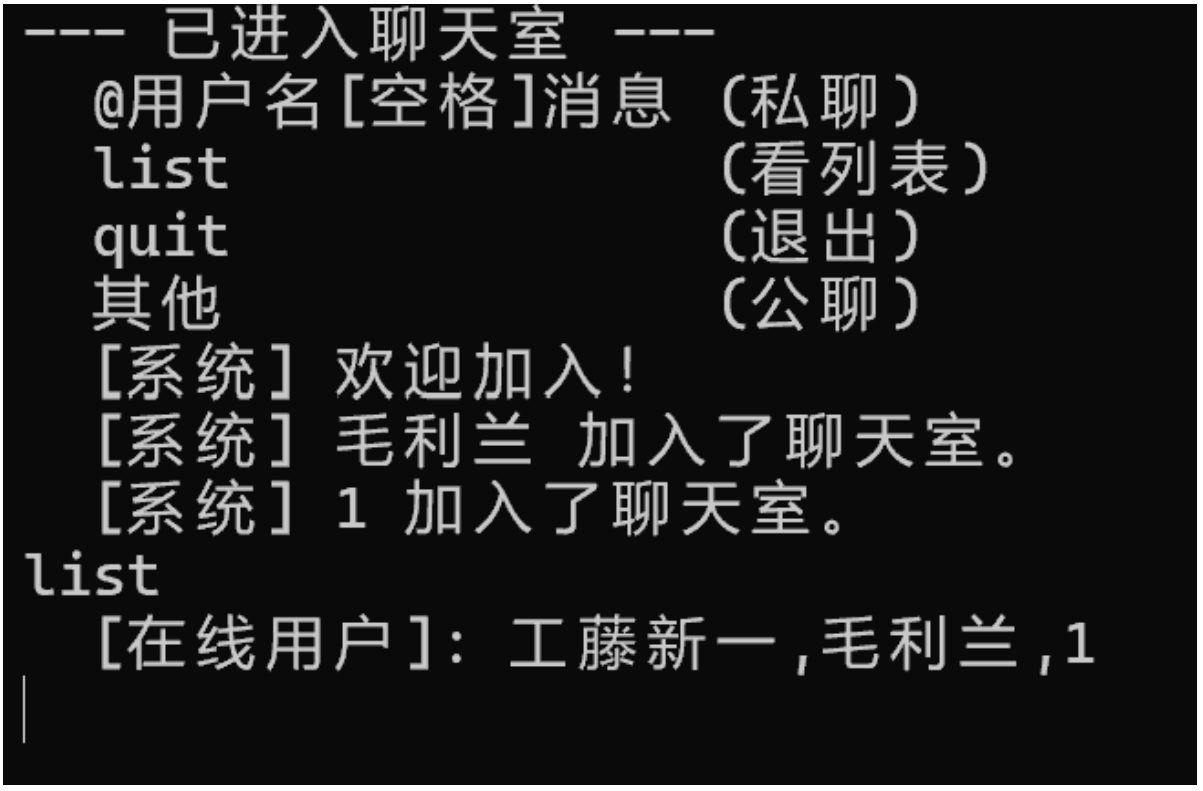
```
{
    getline(cin, inputLine);//阻塞在这里等待用户输入
    if (inputLine.empty()) continue;
    std::string sendBuf; //发给服务器的协议字符串

    if (inputLine == "quit")//主动退出
    {
        sendBuf = "QUIT:";
        send(clientSocket, sendBuf.c_str(), sendBuf.length() + 1, 0);
        cout << ">>> 你已断开连接" << endl;
        break;
    }
    else if (inputLine == "list")//用户列表
    {
        sendBuf = "LIST:";
    }
    else if (inputLine[0] == '@')//检测是否私聊
    {
        size_t separatorPos = inputLine.find(' ');//查找第一个空格
        if (separatorPos != std::string::npos)//如果找到
        {
            std::string targetUser = inputLine.substr(1, separatorPos - 1);//私聊对象
            std::string msg = inputLine.substr(separatorPos + 1);//私聊消息
            sendBuf = "PRIV:" + targetUser + ":" + msg;//私聊协议
        }
        else
        {
            cout << "  [系统] 私聊格式错误，应为: @用户名[空格]消息" << endl;
            continue; //本次输入无效 不发送
        }
    }
    else//默认为公聊
    {
        sendBuf = "MSG:" + inputLine;
    }
    //统一发送
    int sendResult = send(clientSocket, sendBuf.c_str(), sendBuf.length() + 1, 0);
    if (sendResult == SOCKET_ERROR)
    {
        cout << ">>> 发送失败: " << WSAGetLastError() << endl;
        break;
    }
}
closesocket(clientSocket);
WSACleanup();
system("pause");
return 0;
```

四、测试功能

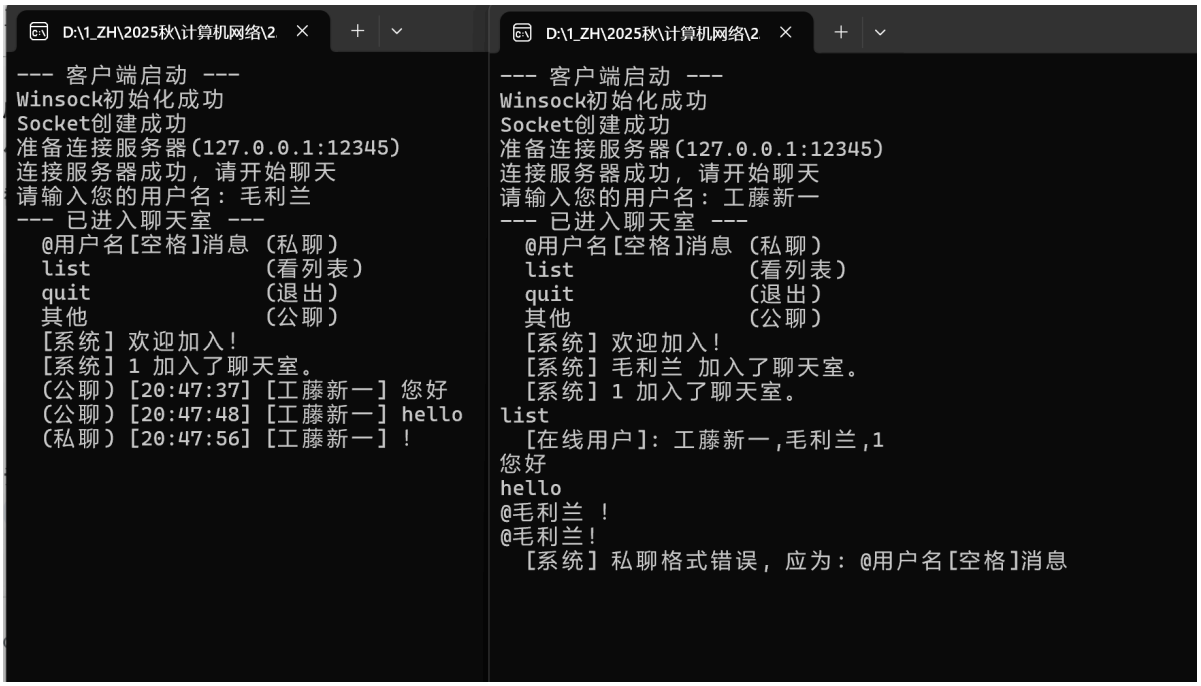
打开一个server和三个client，输入用户名，成功加入：



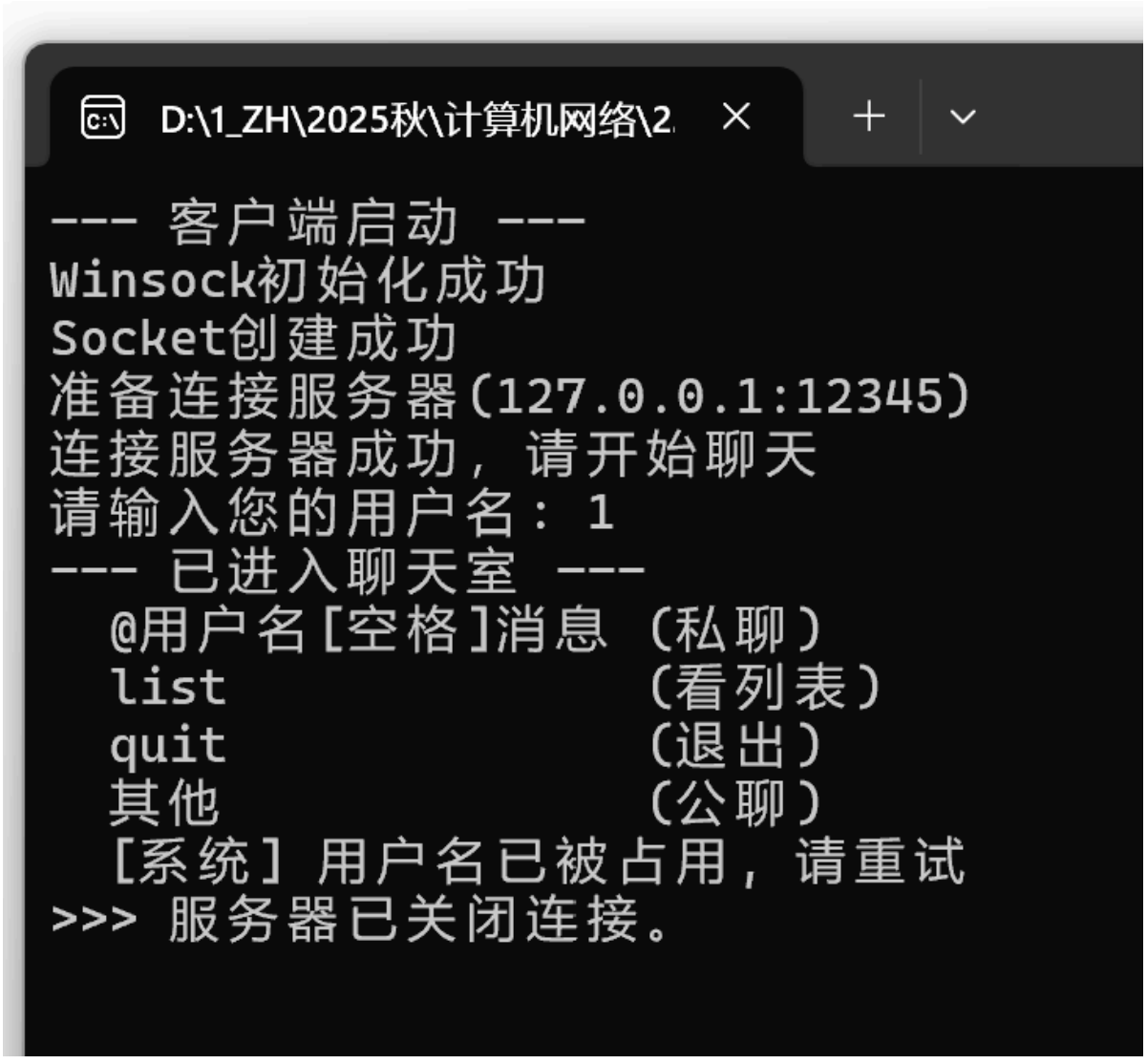
查看用户列表：



带时间戳的中英文公聊与私聊，私聊会检查格式是否正确：


D:\1_ZH\2025秋\计算机网络\2
--- 客户端启动 ---
Winsock初始化成功
Socket创建成功
准备连接服务器(127.0.0.1:12345)
连接服务器成功，请开始聊天
请输入您的用户名： 毛利兰
--- 已进入聊天室 ---
@用户名[空格]消息 (私聊)
list (看列表)
quit (退出)
其他 (公聊)
[系统] 欢迎加入！
[系统] 1 加入了聊天室。
(公聊) [20:47:37] [工藤新一] 您好
(公聊) [20:47:48] [工藤新一] hello
(私聊) [20:47:56] [工藤新一] ！
D:\1_ZH\2025秋\计算机网络\2
--- 客户端启动 ---
Winsock初始化成功
Socket创建成功
准备连接服务器(127.0.0.1:12345)
连接服务器成功，请开始聊天
请输入您的用户名： 工藤新一
--- 已进入聊天室 ---
@用户名[空格]消息 (私聊)
list (看列表)
quit (退出)
其他 (公聊)
[系统] 欢迎加入！
[系统] 毛利兰 加入了聊天室。
[系统] 1 加入了聊天室。
list
[在线用户]: 工藤新一,毛利兰,1
您好
hello
@毛利兰 ！
@毛利兰！
[系统] 私聊格式错误，应为： @用户名[空格]消息

用户名查重：


D:\1_ZH\2025秋\计算机网络\2
--- 客户端启动 ---
Winsock初始化成功
Socket创建成功
准备连接服务器(127.0.0.1:12345)
连接服务器成功，请开始聊天
请输入您的用户名： 1
--- 已进入聊天室 ---
@用户名[空格]消息 (私聊)
list (看列表)
quit (退出)
其他 (公聊)
[系统] 用户名已被占用，请重试
>>> 服务器已关闭连接。

安全退出：

五、总结

本次实验成功实现了一个客户端/服务器架构的多线程聊天室，理解了两种多线程并发模型：

1. 服务器 (1+N)： `main` 线程负责 `accept`， `ClientThread` 子线程一对一地处理 `recv`。
2. 客户端 (1+1)： `main` 线程负责 `getline` 发送消息， `RecvThread` 线程负责 `recv` 接收消息。

使用socket编程，封装了底层的 TCP/IP 协议。在服务器端，其标准流程为 `socket()` 创建、 `bind()` 绑定端口、 `listen()` 开启监听、 `accept()` 阻塞等待并接受连接；在客户端，流程为 `socket()` 创建、 `connect()` 发起连接。一旦连接建立，双方即可通过 `send()` 和 `recv()` 函数进行双向的、可靠的数据流传输。

需要维护线程安全，处理服务器 `g_clients` 的并发访问，使用 `CRITICAL_SECTION` 锁来保护所有的读和写操作，避免服务器崩溃。

设计了一套完整的消息协议，构建了一个功能完备逻辑清晰的聊天系统。

关于数据包丢失，因为实验使用的是 `SOCK_STREAM` ，没有观察到丢包，证明了 TCP 协议栈的超时重传机制保证了数据的可靠交付。