



南开大学  
Nankai University

南开大学

计算机学院和网络空间安全学院

《计算机网络》实验报告

---

Lab01: Socket 编程

---

姓名：陆皓喆

学号：2211044

专业：信息安全

指导教师：徐敬东、张建忠

2024 年 10 月 18 日

# 目录

|                                       |           |
|---------------------------------------|-----------|
| <b>1 实验要求</b>                         | <b>2</b>  |
| <b>2 github 仓库</b>                    | <b>2</b>  |
| <b>3 实验原理</b>                         | <b>2</b>  |
| 3.1 TCP/IP 协议 . . . . .               | 2         |
| 3.2 socket 编程 . . . . .               | 2         |
| 3.3 socket 常用函数 . . . . .             | 3         |
| <b>4 协议设计</b>                         | <b>3</b>  |
| 4.1 总体设计 . . . . .                    | 3         |
| 4.2 Server 端协议 . . . . .              | 4         |
| 4.3 Client 端协议 . . . . .              | 4         |
| 4.4 退出程序机制 . . . . .                  | 5         |
| <b>5 功能实现 &amp; 代码分析</b>              | <b>5</b>  |
| 5.1 My_Head_File.h . . . . .          | 5         |
| 5.2 client.cpp . . . . .              | 7         |
| 5.3 server.cpp . . . . .              | 11        |
| <b>6 结果展示</b>                         | <b>17</b> |
| 6.1 运行界面演示 . . . . .                  | 17        |
| 6.1.1 client 界面 (不开 server) . . . . . | 17        |
| 6.1.2 server 界面 . . . . .             | 18        |
| 6.1.3 client 界面 (开 server) . . . . .  | 18        |
| 6.2 多人聊天演示 . . . . .                  | 19        |
| 6.3 退出程序演示 . . . . .                  | 21        |
| 6.4 附加功能演示 . . . . .                  | 22        |
| <b>7 遇到的问题 &amp; 解决方案</b>             | <b>23</b> |
| <b>8 心得体会</b>                         | <b>24</b> |

## 1 实验要求

- 给出聊天协议的完整说明；
- 利用 C 或 C++ 语言，使用基本的 Socket 函数完成程序。不允许使用 CSocket 等封装后的类编写程序；
- 使用流式套接字、采用多线程（或多进程）方式完成程序；
- 程序应该有基本的对话界面，但可以不是图形界面。程序应该有正常的退出方式；
- 完成的程序应该支持多人聊天，支持英文和中文聊天；
- 编写的程序应该结构清晰，具有较好的可读性；
- 在实验中观察是否有数据丢失，提交可执行文件、程序源码和实验报告。

## 2 github 仓库

本次实验的有关代码和文件，都已经上传至我的个人 github 中。  
您可以通过访问[此链接](#)来查阅我的代码文件。

## 3 实验原理

### 3.1 TCP/IP 协议

TCP/IP 协议，包含了一系列构成互联网基础的网络协议，是 Internet 的核心协议。TCP/IP 协议包含了应用协议、传输协议等。

TCP/IP 体系中的应用层协议，主要包括 HTTP（超文本传输协议）、SMTP（简单邮件传送协议）、FTP（文件传输协议）、TELNET（远程登录协议）、SNMP（简单网络管理协议）；传输层协议，主要包括 TCP（传输控制协议）、UDP（用户数据报协议）。

在本实验中，TCP/IP 协议为本次实验的核心套件 socket 提供了传输途径，是本次实验重要的传输层协议。

### 3.2 socket 编程

Socket 编程基于客户端-服务器模型，其中客户端和服务端之间通过网络进行通信。在 Socket 编程中，客户端和服务端分别创建自己的 Socket 对象，并通过互相发送和接收数据来实现通信。

Socket 是应用层与 TCP/IP 协议族通信的中间软件抽象层。它把复杂的 TCP/IP 协议族隐藏在 Socket 后面，对用户来说只需要调用 Socket 规定的相关函数，让 Socket 去组织符合指定的协议数据然后进行通信。本次实验，我们的任务就是通过调用抽象的封装好的 socket 函数去完成我们的多人聊天程序。

### 3.3 socket 常用函数

socket 有很多常用的函数, 在课上我们学习了很多, 比如 WSAStartup、WSAData、SOCKADDR、WSACleanup、socket、bind、listen、connect、send、recvrecvfrom 等等。此处由于篇幅原因, 我仅列举几个自认为重要的函数的功能及分析。

- recvfrom

```
1 int recvfrom(  
2     socket s,  
3     char *buf,  
4     int len,  
5     int flags,  
6     sockaddr *from,  
7     int *fromlen  
8 );
```

该函数的作用是从特定的目的地接收数据。s 表示要接收数据的套接字描述符; buf 指向用于存储接收数据的缓冲区的指针; len 指的是缓冲区的长度; flags 表示接收标志, 通常可以设置为 0; from 指向 sockaddr 结构的指针, 用于存储发送端的地址信息; 指向 int 的指针, 用于存储 from 结构的长度。在调用 recvfrom 之前, \*fromlen 应该被设置为 sizeof(sockaddr\_in)。

如果函数执行成功, 那么就返回接收到的字节数; 如果没有执行成功, 就返回 SOCKET\_ERROR, 然后可以去调用 WSAGetLastError 来获取错误码。

- send

```
1 int send(  
2     SOCKET s,  
3     const char *buf,  
4     int len,  
5     int flags  
6 );
```

s 代表要发送数据的已连接套接字描述符; buf 指向包含要发送数据的缓冲区的指针; len 指要发送的数据的长度; flags 指发送标志, 通常被设置为 0。

如果函数成功, 返回发送的字节数; 如果失败, 即返回 SOCKET\_ERROR, 然后可以去调用 WSAGetLastError 来获取错误码。

## 4 协议设计

### 4.1 总体设计

本部分, 我主要就我在实验中设计的主要协议做一下解释和说明。

在本实验的设计中, 我们使用 TCP 传输协议, 选用流式套接字, 采用多线程方式进行编程。我们分别设计了两个程序 client.exe 和 server.exe, 分别实现了用户端盒客户端的通信传输要求, 在使用

时, 我们首先打开 server 可执行文件, 然后任意打开多个 client 可执行文件, 即可完成多人之间的通信, 并且在用户端和客户端均可以看到所有的通信消息。

为了保证通讯质量, 本次实验在 server 设置了最大缓冲区 BufSize。当发送的信息达到消息缓冲区的限制时, 超出部分无法发送。另外, 我们对输入名称的长度也同样做了限制, 如果超过了最大缓冲区, 我们就会直接输出 “the name is too long!”, 然后循环让用户进行姓名的输入, 直到长度符合要求。我们还编写了很多的封装式函数和报错函数, 方便我们调试程序和调用函数, 这部分我会在实验过程一节中介绍。

我们的消息传输内容在代码中已经做了规定, 总体的输出情况为: “<Luhaozhhe’s Chatting Room:: Server str\_time # Message>:sendbuf”。

< 是我们的消息开始符号, 然后输出我们的聊天室的名字, 后面可以输出 server 或者 client, 规定了为客户端还是用户端; str\_time 通过时间戳的计算, 输出了当前聊天的时间; message 代表了聊天的类型, 此处我分别设计了 message 和 notice。message 代表收到的消息的内容, notice 代表用户加入聊天室时的提示。最后, 则是输出存放在缓冲区内的消息。这样我们的消息就发送完毕了。

下面我分别就 server 的 send、server 的 recv、client 的 send 和 client 的 recv 来说明一下我设计的协议。

## 4.2 Server 端协议

我们在 server 中定义了不同的消息输出模式, 包括输出用户的消息、用户的进入与退出。

- **Message 类:** 表示正常向其他人发送消息, 并且输出在 server 自己的界面中;
- **Notice 类:** 如果有新的人进入了聊天室, 就输出 Welcome < XXX > join the ChatGroup!(XXX 为进入的用户名), 如果有人通过 QUIT 退出了聊天程序, 那我们就释放对应的资源, 同时同步更新服务器端的状态。

服务器端监听指定端口 8000, 等待客户端的连接请求。服务器端可以同时接受多个客户端的连接请求, 每个客户端连接后, 服务器为其创建一个独立的线程, 接收用户发送的消息, 并在两个端口分别输出。

每当服务器接收到一个新的用户线程, 对应生成的一个用户名, 首先, 服务器会判定该用户名的名字是否超过最大缓冲区的容量, 如果超过的话就需要重新输入; 如果没有超过的话, 就成功进入聊天室, 同时, server 接收到 client 的新用户名后, 会自动创建一个新的类对象 client, 初始化其 socket 和 Client\_Name, 然后同时对在线总人数做一下更新, 存储在 clientCount 中。

如果用户输入了 QUIT 退出, 在 server 端就可以监听到 log 的值发生了变化, 如果 log 为 0 的话, 清除对应 client 的内容, 更新 vector 容器的状态, 更新在线的总人数, 同时在服务器端和用户端同时输出用户离开聊天室的信息。

## 4.3 Client 端协议

客户端创建一个套接字, 然后连接到服务器的 IP 地址和端口号 (127.0.0.1 和 8000)。

首先进入聊天室, 需要输入用户名新建用户, 具体内容为: Please enter your name(No more than 255 words):XXX (“XXX” 代表着输入的用户名)。

新建用户成功后, 系统会提示用户 “Please enter:”, 我们需要输入对应的英文或中文, 来进行公开聊天。发送的信息首先通过 send 函数传递到 server 服务器端, 再由服务器端来进行广播发送的工作。

如果用户输入“QUIT”，那么我们的 Is\_Quit 函数就会检测到用户退出了程序，首先 client 端会关闭该进程；同时 server 端也会做相应的更新，输出“left the ChatGroup!”的信息，同时更新总人数。

#### 4.4 退出程序机制

程序可以通过用户在聊天框内输入 QUIT 来进行退出聊天室，结束对应的进程；同时，服务器端会检测客户端的连接状态，如果客户端主动关闭连接，服务器会在控制台上显示客户端退出的消息，并关闭相应的套接字，释放资源。同时，总人数也会做相应的更新。

## 5 功能实现 & 代码分析

本部分我主要就我编写的程序分别进行分析，由于篇幅原因，此处我只挑选核心代码进行详细的分析。该项目主要由两个工程文件构成，分别为 client 和 server，同时编写了一个头文件，供 client 和 server 的主文件使用。

### 5.1 My\_Head\_File.h

该头文件，我对后续编程需要使用的类和函数分别做了定义，下面做一下分析。

#### Get\_Last\_Error\_Details 函数

```
1 string Get_Last_Error_Details() {
2     int error_code = WSAGetLastError();
3
4     char errorMsg[256] = { 0 };
5     FormatMessageA(
6         FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_IGNORE_INSERTS,
7         NULL,
8         error_code,
9         MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
10        errorMsg,
11        sizeof(errorMsg) - 1,
12        NULL
13    );
14
15    return string("Error code is: ") + to_string(error_code) + ", and the Details:
16    ↪  " + errorMsg;
17 }
```

该函数用于获取错误代码，并使用 FormatMessage 输出详细日志信息便于调试。这样我们在编写程序调试时，就可以知道具体出现的问题了。用户使用连接时，若没有完成连接，也可以知道为什么会连接失败。

### Get\_Random\_Name 函数

```
1 string Get_Random_Name() { //如果用户输入换行, 则自动分配随机用户名
2     srand(time(0));
3     int name_length = rand() % 5 + 1;
4     const string charpool =
5         ↪ "0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";
6     string temp_name;
7     for (int i = 0; i < name_length; i++) {
8         int Index_Of_Name = rand() % charpool.length();
9         temp_name = temp_name + charpool[Index_Of_Name];
10    }
11    return temp_name;
12 }
```

该函数用于随机生成用户名。如果在 client 端口, 用户没有输入对应的名称就按下了换行键, 那么我们不能给他一个空的名字, 所以我们随机为其分配一个用户名, 使用 rand() 函数来完成用户名的随机分配。

### Is\_Quit 函数

```
1 //判断用户是否自愿退出
2 bool Is_Quit(char* text) {
3     int len = 0;
4     while (text[len] != '\0') {
5         len++;
6     }
7     if (len == 4 && text[0] == 'Q' && text[1] == 'U' && text[2] == 'I' && text[3]
8         ↪ == 'T') {
9         return true;
10    }
11    else {
12        return false;
13    }
14 }
```

该函数用于判定用户是否自行退出。具体方法就是, 通过匹配用户输入的信息, 如果输入了 QUIT(均为大写), 那么我们就判定该用户想要退出程序, 于是该函数对应输出 true。

### 定义用户类

```
1 class Client {
2     private:
```

```
3     SOCKET Client_Socket;
4     string Client_Name;
5
6 public:
7     Client() {
8
9     }
10    SOCKET getClientSocket() const {
11        return Client_Socket;
12    }
13    void setClientSocket(SOCKET socket_name) {
14        Client_Socket = socket_name;
15    }
16    string getClientName() {
17        return Client_Name;
18    }
19    void setClientName(string temp) {
20        if (temp == "\\0") //若输入进来的是串尾符，则没有正常输入名字
21            Client_Name = Get_Random_Name();
22        else
23            Client_Name = temp;
24    }
25};
```

该部分为 **client** 类的定义，我们申明了一个 **client** 成员类，后续需要使用的时候，直接使用该类进行新建成员即可。该类含有：私有变量，分别为对应的 **socket** 和用户的名字；公有类为：初始化函数、获取用户的 **socket** 函数、设置用户的 **socket** 函数、获取用户的名字、设置用户的名字。其中，我在 **setClientName** 函数中添加了条件，如果用户输入回车的话，说明用户没有选择输入用户名，我们就帮他随机生成一个用户名即可。此处调用了前面提到的 **Get\_Random\_Name** 函数。

## 5.2 client.cpp

该程序主要完成了 **client** 端的构建，核心函数为 **Recv\_Message\_Thread**，**main** 函数完成了程序主要功能的构建，下面一一进行分析。

### Recv\_Message\_Thread 函数

```
1 DWORD WINAPI Recv_Message_Thread(LPVOID lpParamter) {
2     SOCKET recv_temp = (SOCKET)lpParamter;
3     int log;
4
5     bool recv_flag = true;
6     while (recv_flag == true || (log != SOCKET_ERROR && log != 0)) {
```



```
7     recv_flag = false;
8     memset(recvbuf, 0, sizeof(recvbuf));
9     log = recv(recv_temp, recvbuf, 255, 0);
10    if (log == SOCKET_ERROR) {
11        if (quit_flag) {
12            return 0;
13        }
14        else {
15            cout << endl;
16            cout << "Server unexpectedly closed. Press ENTER to reconnect..."
17                << endl;
18            reconnect = true;
19            return 0;
20        }
21    }
22    if (string(recvbuf) == "Please enter:")
23        cout << string(recvbuf);
24    else
25        cout << string(recvbuf) << endl;
26    }
27 }
```

这个函数主要完成了接收消息的功能。SOCKET `recv_temp` 是一个套接字，用于网络通信。它被转换为 SOCKET 类型，从 `lpParameter` 参数中接收。然后我们初始化日志内容，设置循环的 bool 值，然后进行 do while 循环，直到 SOCKET 错误为止。

我们首先清空缓冲区 `recvbuf` 上的内存，然后判定 `log` 的值是否为 SOCKET\_ERROR，如果是的话，再判定 `quit_flag` 的值，如果程序是用户主动退出的，那么就退出程序；如果不是自己退出的，那么就让用户自己尝试 reconnect，也就是我们将 `reconnect` 的值设置为 true。在循环中，我们如果接收到 "Please enter:"，就照常输出不换行，如果收到别的，就需要进行换行。

**main 函数部分，我们大概介绍一下总流程。**首先，输出欢迎语句，然后进入 do while 循环，只要 `reconnect` 的 bool 值为 true 就可以进行重连。然后前面是一堆配置初始环境的代码：

### 初始化 Winsock 库

```
1  if (WSAStartup(MAKEWORD(2, 0), &Wsa_Data) != 0) {
2      cout << "Something Wrong!Failed to Initialize the Environment!" << endl;
3      cout << Get_Last_Error_Details() << endl;
4      cout << "You Should Try Again!" << endl;
5      return 0;
6  }
7  else {
8      cout << "Congratulations!Successfully Initialized the Environment!" << endl;
```

```
}

```

该部分主要完成了对 Winsock 库的初始化工作, MAKEWORD(2, 0) 表示我们初始化的版本为 2.0; Wsa\_Data 是指向 WSADATA 结构的指针, 该结构用于接收 Winsock 实现的详细信息; 如果 WSASStartup 返回非零值, 表示初始化失败。在这种情况下, 代码会输出错误信息, 调用 Get\_Last\_Error\_Details 函数进行报错输出。如果 WSASStartup 的值为 0 的话, 说明我们的初始化成功, 进入 else 语句, 输入正确信息。

### 创建套接字

```
1 Client_Socket = socket(AF_INET, SOCK_STREAM, 0);
2
3 if (Client_Socket == INVALID_SOCKET) {
4     cout << "Something Wrong!Failed to Create the Socket!" << endl;
5     cout << Get_Last_Error_Details() << endl;
6     cout << "You Should Try Again!" << endl;
7     WSACleanup();
8     return 0;
9 }
10 else {
11     cout << "Congratulations!Successfully Initialized the Socket!" << endl;
12 }
```

首先, 我们创建了一个新的套接字, AF\_INET 表示地址族为 IPv4, SOCK\_STREAM 表示套接字类型为流式套接字 (TCP), 参数 0 表示使用默认协议。然后进入 if 循环, 如果 Client\_Socket 的值是 INVALID\_SOCKET, 那么就说明创建失败, 我们输出一系列的报错信息, 然后调用 WSACleanup 进行网络资源的清除, 结束程序; 如果创建成功, 则输出成功配置信息。

### 配置 ip 地址

```
1 Client_Addr.sin_family = AF_INET;
2 Client_Addr.sin_port = htons(8000);
3 inet_pton(AF_INET, "127.0.0.1", &(Client_Addr.sin_addr));
```

该处的作用为, 配置地址族为 AF\_INET, 表示使用 IPv4 地址; 然后设置端口号为 8000, htons 函数用于将主机字节序的无符号短整型转换为网络字节序 (大端序), 这是网络通信中的标准格式。最后将点分十进制的 IP 地址字符串转换为网络字节序的二进制形式, 并存储在 Client\_Addr.sin\_addr 中, 这样我们的 ip 地址就配置完毕了!

### 连接服务器

```
1 if (connect(Client_Socket, (SOCKADDR*)&Client_Addr, sizeof(Client_Addr)) ==
   ↪ SOCKET_ERROR) {
2     cout << "Oops!Fail to Connect the Server!" << endl;
```

```
3     cout << Get_Last_Error_Details << endl;
4     cout << "Wait for 5 Seconds and Please Try Again!" << endl;
5     WSACleanup();
6     Sleep(5000); //等待 5 秒, 重新尝试
7     continue;
8 }
9 else {
10     cout << "Congratulations! Successfully Connect the Server!" << endl;
11     reconnect = false; //修改连接状态的 bool 值, 防止再一次进入循环
12 }
```

此处的作用为, 使用 connect 函数连接到指定的服务器地址 (Client\_Addr)。如果连接失败, connect 将返回 SOCKET\_ERROR, if 函数中, 如果遇到连接失败, 那么就输出报错信息, 同时程序睡眠 5 秒, 再进行下一次连接尝试; 如果连接上了, 首先将 reconnect 的值变成 false, 防止再次连接, 然后输出成功信息。

### 配置用户名

```
1  memset(sendbuf, 0, sizeof(sendbuf));
2  cout << "Please Enter your Name(No more than 255 words):" << endl;
3
4  LLL: bool create_condition = true;
5
6  while (create_condition) { //判定创建用户是否成功
7      getline(cin, Client_Name);
8      if (Client_Name.length() > 255) {
9          cout << "the name is too long!" << endl;
10         goto LLL;
11     }
12     create_condition = false;
13 }
```

该段代码的功能为, 首先清空对应的缓冲区, 准备接收用户输入的姓名。然后使用一个 goto 语句, 来完成对用户名称进行接收的操作。如果用户输入的名称长度超过 255, 那么就提示名字过长, 要求重新输入, 只有当 create\_condition 的值为 false 的时候, 才会退出 goto 循环。

### 通过循环传输消息

```
1  send(Client_Socket, Client_Name.data(), Client_Name.length(), 0);
2  HANDLE ThreadHandle = CreateThread(NULL, 0, Recv_Message_Thread,
   ↪ (LPVOID)Client_Socket, 0, NULL);
3  CloseHandle(ThreadHandle);
4  int log = 0;
5  bool log_send_condition = true;
```

```
6 while (log_send_condition == true || (log != SOCKET_ERROR && log != 0)) {
7     log_send_condition = false;
8     memset(sendbuf, 0, sizeof(sendbuf));
9     cin.getline(sendbuf, 255);
10    if (Is_Quit(sendbuf) == true) {
11        quit_flag = true;
12        shutdown(Client_Socket, 0x02);
13        closesocket(Client_Socket);
14        WSACleanup();
15        return 0;
16    }
17    log = send(Client_Socket, sendbuf, 255, 0);
18 }
```

这段代码首先将 Client\_Socket 的信息都进行了发送,创建了对应的句柄,包含了对应的线程,然后关闭的句柄的线程。之后,进入 while 循环,清空缓冲区开始接收消息,这边我们还需要考虑用户主动退出的情况,于是设计 if 函数,如果用户输入的 sendbuf 中的值为 QUIT,那么我们就直接强制关闭该进程,同时清空该进程中所有的网络数据,同时退出程序。如果正常输入的话,我们就记录对应的 log 为 send 函数传递的那些值。

最后,我们一直接收对应传输的消息,直到进程中断,即 reconnect 的值变为 false。这样我们就完成了对应的传输任务,最后关闭套接字,同时清空网络缓存即可。

综上所述,客户端的设计其实非常简单,我们首先加载 Winsock2 环境,然后进行套接字创建,接着设定客户端地址和端口,最后连接服务器端,完成环境的配置。然后我们就要开始传输消息了。由于 client 端没有需要不断连接的需求,只要一次连接成功即可,因此 main 线程用作主线程不断地接收命令行的输入并发送给服务器端,使用 do...while 循环进行接收;但是还需要一个线程不断来接收来自服务器端中转的消息,因此,我们还需要额外一个线程作为接收信息的线程。

### 5.3 server.cpp

该程序主要完成了 server 端的构建,核心函数为 Send\_Message\_Thread、Recv\_Message\_Thread、main 函数完成了主要功能的构建,下面进行详细的分析。

#### Broad\_Cast\_Message 函数

```
1 //用于广播我们的信息
2 void Broad_Cast_Message(const string& message) {
3     lock_guard<mutex> lock(Clients_Mutex);
4     for (const auto& client : Clients) {
5         send(client.getClientSocket(), message.c_str(), message.size(), 0);
6     }
7 }
```

这个函数接受一个 const string& 类型的参数 message,表示要广播的消息。然后我们使用了 lock\_guard 来自动管理一个互斥锁 (mutex) 的生命周期。Clients\_Mutex 是一个互斥锁对象,用于保护对 Clients

容器的访问，确保了线程安全。最后，我们遍历所有的用户，完成我们信息的广播操作，这样所有人都可以收到消息了。

### Send\_Prompt 函数

```
1 //用于传输我们的提示符
2 void Send_Prompt() {
3     lock_guard<mutex> lock(Clients_Mutex);
4     for (const auto& client : Clients) {
5         send(client.getClientSocket(), "Please enter:", 13, 0);
6     }
7 }
8 }
```

这个函数的用处很明显，就是为了给我们的用户都发送输入提示符 “Please enter:”，前面的部分和上个函数用处是一样的，此处不再说明。通过该函数，所有的用户都会收到对应的信息。

### Log\_And\_Broadcast 函数

```
1 //用于记录日志和广播
2 void Log_And_Broadcast(const string& message) {
3     cout << message << endl;
4     Broad_Cast_Message(message);
5     cout << "Current online clients: " << clientCount << endl;
6 }
```

该函数嵌套了前面的 Broad\_Cast\_Message 函数，作用是首先对于输入的 message 在服务器端进行输出，然后调用 Broad\_Cast\_Message 函数，向每一个 client 都发送对应的消息，最后在服务器端同步输出当前在线的人数 clientCount。

### Get\_Formatted\_Time 函数

```
1 //用于获取当前的时间
2 string Get_Formatted_Time() {
3     time_t t;
4     char str_time[26];
5     time(&t);
6     ctime_s(str_time, sizeof str_time, &t);
7     str_time[strlen(str_time) - 1] = '\\0';
8     return string(str_time);
9 }
10 }
```

这个函数用于输出当前的时间戳，在后续的 main 函数中，我们会经常使用该函数来完成当前时间的读取。

### Send\_Message\_Thread 函数

```
1  DWORD WINAPI Send_Message_Thread(LPVOID lpParameter) {
2      SOCKET send_temp = (SOCKET)lpParameter;
3      int log = 0; // 初始化 log 变量
4      bool Bool_Condition = true;
5      while (Bool_Condition = true || (log != SOCKET_ERROR && log != 0)) {
6
7          Bool_Condition = false;
8          memset(sendbuf, 0, sizeof(sendbuf));
9          cin.getline(sendbuf, bufsize - 1);
10
11         if (Is_Quit(sendbuf)) {
12             closesocket(Server_Socket);
13             WSACleanup();
14             exit(0);
15         }
16         string str_time = Get_Formatted_Time();
17         str = "<Luhaozhhe's Chatting Room::Server @ " + str_time + " # Message>:"
18             ↵ " + string(sendbuf);
19         cout << str << endl;
20         Broad_Cast_Message(str);
21
22         Send_Prompt();
23     }
24     return 0;
25 }
```

该函数完成了消息的发送操作。首先，lpParameter 被转换为 SOCKET 类型，表示要用于发送数据的套接字；然后初始化 log 变量和 bool 值，然后进入 while 循环，清空缓冲区，读取用户的输入值。然后首先进行第一次判定，如果输入的存在缓冲区的值为 QUIT 的话，我们的 Is\_Quit 函数就会返回 true，程序执行退出，清空网络数据，如果不是 QUIT，说明接收到的是正常的信息，我们首先读取系统当前的时间，然后将消息进行打包输出，附带有前缀、当前时间、消息类型为 messege，最后是读取到的缓冲区内的字符值，也就是我们发送过去的值。

然后，我们先将 str 输出在 server 端，调用 Broad\_Cast\_Message 函数和 Send\_Prompt 函数来完成消息的广播操作以及完成在用户端的“Please enter:”的提示符输出。这样我们的 Send\_Message\_Thread 的功能就都实现了。

### Recv\_Message\_Thread 函数

```
1  //接收消息的线程函数
2  DWORD WINAPI Recv_Message_Thread(LPVOID lpParameter) {
3      SOCKET recv_temp = (SOCKET)lpParameter;
```

```
4
5     memset(recvbuf, 0, sizeof(recvbuf));
6     int log = recv(recv_temp, recvbuf, bufsize - 1, 0);
7     string username;
8     Client c;
9     if (recvbuf[0] == '\0') {
10         c.setClientSocket(recv_temp);
11         c.setClientName("\0");
12     }
13     else {
14         c.setClientSocket(recv_temp);
15         c.setClientName(string(recvbuf));
16     }
17
18     {
19         lock_guard<mutex> lock(Clients_Mutex);
20         Clients.push_back(c);
21         clientCount++; // 增加客户端计数
22     }
23     username = c.getClientName();
24
25     string str_time = Get_Formatted_Time();
26     str = "<Luhaozhhe's Chatting Room::Server @ " + str_time + " # Notice>:"
27     ↪ " Welcome <" + username + "> join the ChatGroup!";
28     Log_And_Broadcast(str);
29
30     bool recv_flag = true;
31
32     while (recv_flag = true || (log != SOCKET_ERROR && log != 0)) {
33         recv_flag = false;
34         memset(recvbuf, 0, sizeof(recvbuf));
35         Send_Prompt();
36         log = recv(recv_temp, recvbuf, bufsize - 1, 0);
37
38         if (log == 0) {
39             {
40                 lock_guard<mutex> lock(Clients_Mutex);
41                 clientCount--; // 减少客户端计数
42                 Clients.erase(remove_if(Clients.begin(), Clients.end(),
43                     ↪ [recv_temp](const Client& client) { return
44                     ↪ client.getClientSocket() == recv_temp; }), Clients.end());
45             }
46         }
47     }
```



```
43         string str_time = Get_Formatted_Time();
44         str = "<Luhaozhhe's Chatting Room::Server @ " + str_time + " #
           ↳ Notice>: <" + username + "> has left the ChatGroup!";
45         Log_And_Broadcast(str);
46         break;
47     }
48
49     string str_time = Get_Formatted_Time();
50     string msg = string(recvbuf);
51
52     str = "<Luhaozhhe's Chatting Room::" + username + " @ " + str_time + " #
           ↳ Message>: " + msg;
53     Log_And_Broadcast(str);
54 }
55
56 closesocket(recv_temp); // 关闭客户端套接字
57 return 0;
58 }
```

**该函数完成了消息线程的接收工作。**首先接收到一个对应的套接字，我们定义为 `recv_temp`。然后我们清空缓冲区，用自带的 `recv` 函数（前面已经介绍过了功能）读取对应的 `log` 记录。

然后我们开始定义我们的用户。首先新开一个 `Client` 类的用户 `c`，对应的用户名为 `username`。然后判断读取到的第一位是否为换行键，如果是的话就随机分配一个名字；如果不是的话，就读取缓冲区内的内容作为用户名。然后通过互斥锁将 `c` 用户 `push` 到 `vector` 容器中，同时完成在线人数的更新，也就是加一操作。然后，将 `c` 的用户名赋值给 `username`。然后，定义 `str` 的内容为欢迎加入聊天组，通过之前定义的 `Log_And_Broadcast` 函数来完成广播操作。

后面相当于和前面的是一样的，就是通过 `do while` 循环来接收各个用户传递的信息，并且通过 `Log_And_Broadcast` 函数进行广播与输出。需要注意的是，此处我们还做了 `client` 用户数量的更新，也就是说，每当读取到一个用户的 `log` 值为 0，也就是退出程序的时候，我们在操作互斥锁的时候，同时对在线人数做一次更新，这样就可以保证我们在线人数统计的正确性。

### main 函数

```
1  if (bind(Server_Socket, (SOCKADDR*)&Server_Addr, sizeof(Server_Addr)) ==
    ↳ SOCKET_ERROR) {
2      cerr << "Oops! Failed to Bind the Socket!" << endl;
3      cerr << Get_Last_Error_Details() << endl;
4      cerr << "You Should Try Again!" << endl;
5      WSACleanup();
6      return 0;
7  }
8  cout << "Congratulations! Successfully Bind the Socket!" << endl;
```



由于 main 函数中与 client 程序重复的较多, 所以我只挑不同的核心部分进行分析。

首先就是对于 bind 函数的使用了。Server\_Socket 是之前创建的套接字, Server\_Addr 是一个 sockaddr\_in 结构体 (在 IPv4 中), 它包含了服务器的地址和端口信息。bind 函数尝试将套接字绑定到由 Server\_Addr 指定的网络地址和端口上。如果 bind 函数返回 SOCKET\_ERROR, 表示绑定操作失败, 输出错误信息; 如果绑定成功, 那就输出成功的消息。

```
1  if (listen(Server_Socket, bufsize - 1) != 0) {
2      cerr << "Fail to Listen For the Connections!" << endl;
3      cerr << Get_Last_Error_Details() << endl;
4      cerr << "You Should Try Again!" << endl;
5      WSACleanup();
6      return 0;
7  }
8  cout << "Congratulations! Successfully Start Listening!" << endl;
```

该段程序完成了套接字的监听, 说明服务器准备接受来自客户端的连接请求。Server\_Socket 是之前创建并绑定到特定地址和端口的套接字。listen 函数使套接字进入监听状态, 准备接受连接请求。5 是一个参数, 表示服务器队列中可以容纳的最大挂起连接数。也就是说, 我们同时支持 255 个用户进行在线聊天。

```
1  CloseHandle(CreateThread(NULL, 0, Send_Message_Thread, (LPVOID)Server_Socket, 0,
   ↪  NULL));
2  while (1) {
3      sockaddr_in addrClient;
4      int addr_client_len = sizeof(addrClient);
5      SOCKET Socket_Information = accept(Server_Socket, (sockaddr*)&addrClient,
   ↪  &addr_client_len);
6      if (Socket_Information == INVALID_SOCKET) {
7          cerr << "Oops! Failed to connect a client" << endl;
8          cerr << Get_Last_Error_Details() << endl;
9      }
10     CloseHandle(CreateThread(NULL, 0, Recv_Message_Thread,
   ↪  (LPVOID)Socket_Information, 0, NULL));
11 }
```

首先, 我们使用 CreateThread 函数创建一个新线程, 该线程将执行 Send\_Message\_Thread 函数。然后, 我们将服务器套接字 Server\_Socket 作为参数传递给 Send\_Message\_Thread 函数。CloseHandle 调用用于关闭新创建的线程句柄。

然后, 我们通过 While 循环不停接收其它用户的 accept 请求, 并输出相关日志。每当有个新用户进来, 通过 CreateThread 创建为该用户准备的个人线程 RecvMessageThread 用来接收对应的消息。

如果连接失败, 就输出失败的消息, 如果成功连接, 那么完成传输后, 关闭对应的句柄即可。

最后, 跟上一个文件一样, 我们还是在最后释放出我们的网络数据和所有句柄, 程序结束。

综上所述,服务器端的设计思想为:main 线程用来不断接收新的用户的连接请求;Send\_Message\_Thread 线程用来发送数据;Recv\_Message\_Thread 用来为某个用户接收数据。在连接开始前,输出我们的欢迎信息,然后开始加载我们的 winsock2 环境。接着进行套接字的初始化,使用 AF\_INET 即 IPv4 协议,SOCK\_STREAM 即用 TCP 协议的流式套接字。我们同样输出类似的日志;然后,我们为套接字绑定 IP 地址和端口,使用 bind 函数绑定,设置端口 8000,设置 IPv4 地址为 127.0.0.1 即本机的 localhost。接着启用我们的 listen 函数进行监听,不能超出一定的用户数量,不然就监听不到了。最后,创建发送信息的线程,CreateThread 后关闭线程句柄,重定向为 Send\_Message\_Thread,并绑定套接字 Server\_Socket。

接收到连接后,我们通过 While 循环,来不停地接收其它用户的请求,并输出相关日志。每当有新用户进来的时候,就通过 CreateThread 函数来创建为该用户准备的个人线程 Recv\_Message\_Thread 用来接收对应的消息。这就是有关 server 端的总体设计。

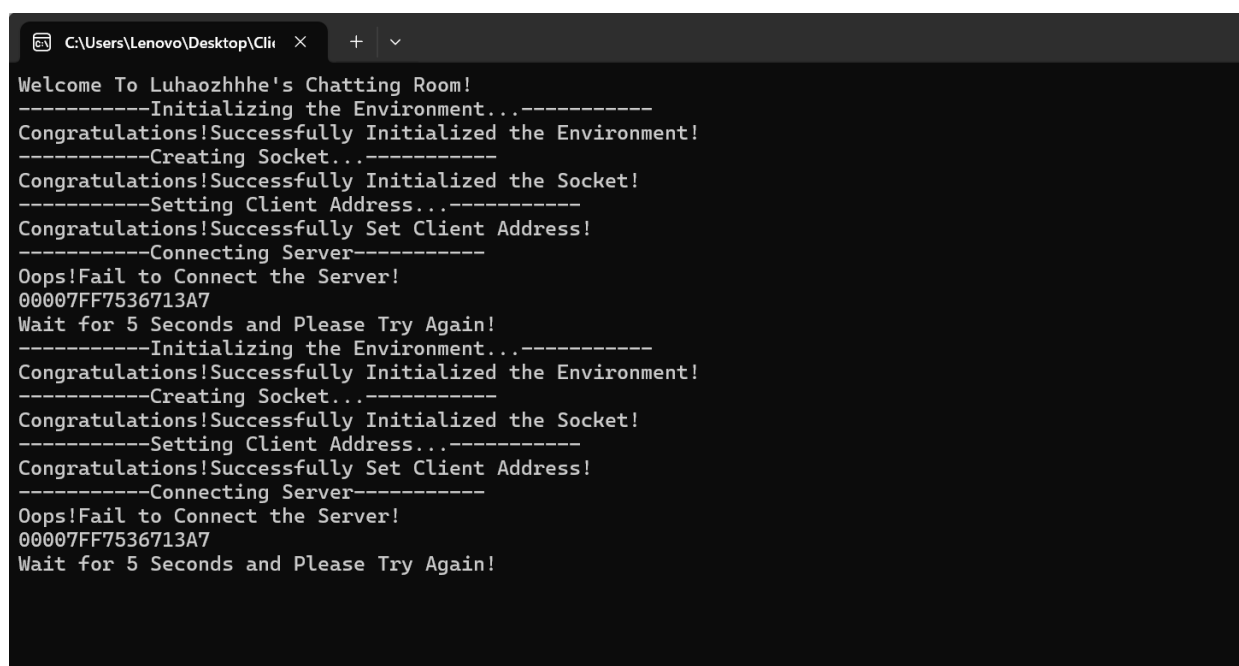
## 6 结果展示

下面,我就对我设计的可执行程序的主要界面、基本功能以及进阶功能进行演示。

### 6.1 运行界面演示

#### 6.1.1 client 界面 (不开 server)

我们首先运行 client 界面,但是不使用服务器,如图6.1所示。



```

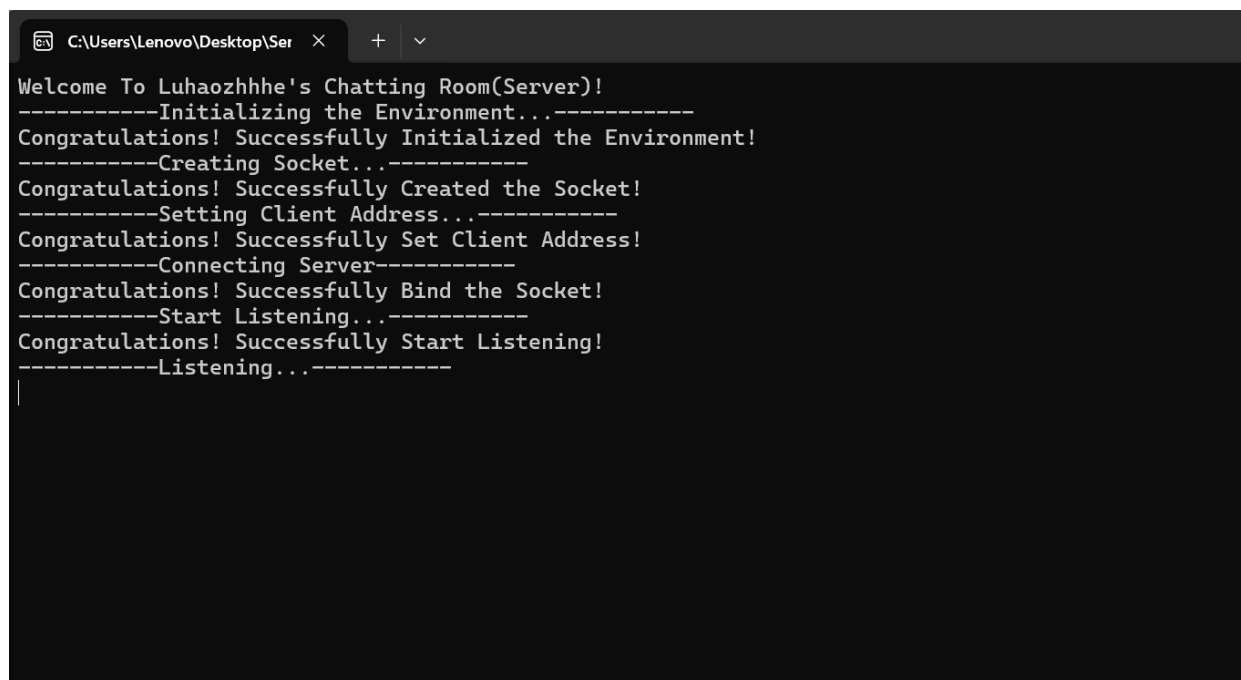
C:\Users\Lenovo\Desktop\Cli
Welcome To Luhaozhhe's Chatting Room!
-----Initializing the Environment...-----
Congratulations!Successfully Initialized the Environment!
-----Creating Socket...-----
Congratulations!Successfully Initialized the Socket!
-----Setting Client Address...-----
Congratulations!Successfully Set Client Address!
-----Connecting Server-----
Oops!Fail to Connect the Server!
00007FF7536713A7
Wait for 5 Seconds and Please Try Again!
-----Initializing the Environment...-----
Congratulations!Successfully Initialized the Environment!
-----Creating Socket...-----
Congratulations!Successfully Initialized the Socket!
-----Setting Client Address...-----
Congratulations!Successfully Set Client Address!
-----Connecting Server-----
Oops!Fail to Connect the Server!
00007FF7536713A7
Wait for 5 Seconds and Please Try Again!
```

图 6.1: client 界面 (不开 server)

我们发现,在没有 server 的情况下,用户是无法进行连接的,所以程序调用了设计的 Get\_Last\_Error\_Details 来输出详细的错误信息,系统提示,等待五秒后再进行重连。

### 6.1.2 server 界面

然后我们打开 server 界面，发现成功输出了日志等信息，如图6.2所示。



```
C:\Users\Lenovo\Desktop\Ser x + v
Welcome To Luhaozhhe's Chatting Room(Server)!
-----Initializing the Environment...-----
Congratulations! Successfully Initialized the Environment!
-----Creating Socket...-----
Congratulations! Successfully Created the Socket!
-----Setting Client Address...-----
Congratulations! Successfully Set Client Address!
-----Connecting Server-----
Congratulations! Successfully Bind the Socket!
-----Start Listening...-----
Congratulations! Successfully Start Listening!
-----Listening...-----
|
```

图 6.2: server 界面

说明 server 端的运行是正常的!

### 6.1.3 client 界面 (开 server)

然后我们先运行 server 端，然后再运行 client 端，会发现 client 端成功识别到了 server 的 ip 端口，并且系统提示我们让我们输入用户名，如图6.3所示。

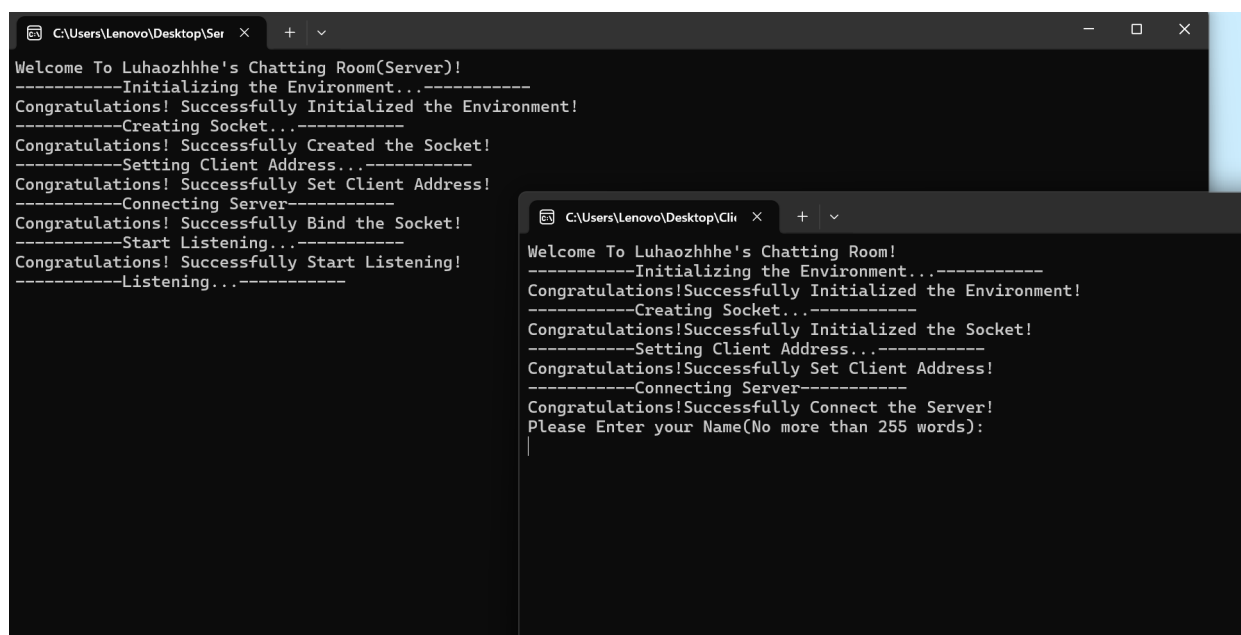


图 6.3: client 界面 (开 server)

我们发现，在 server 开启的情况下，client 端也可以正常的运行了！

## 6.2 多人聊天演示

下面，我们对本次作业的主要部分进行一个演示。由于我们设置的监听最大个数为 255 个，所以我们本次演示仅开启一个 server 端和三个 client 端。首先完成对应的初始化，如图6.4所示。

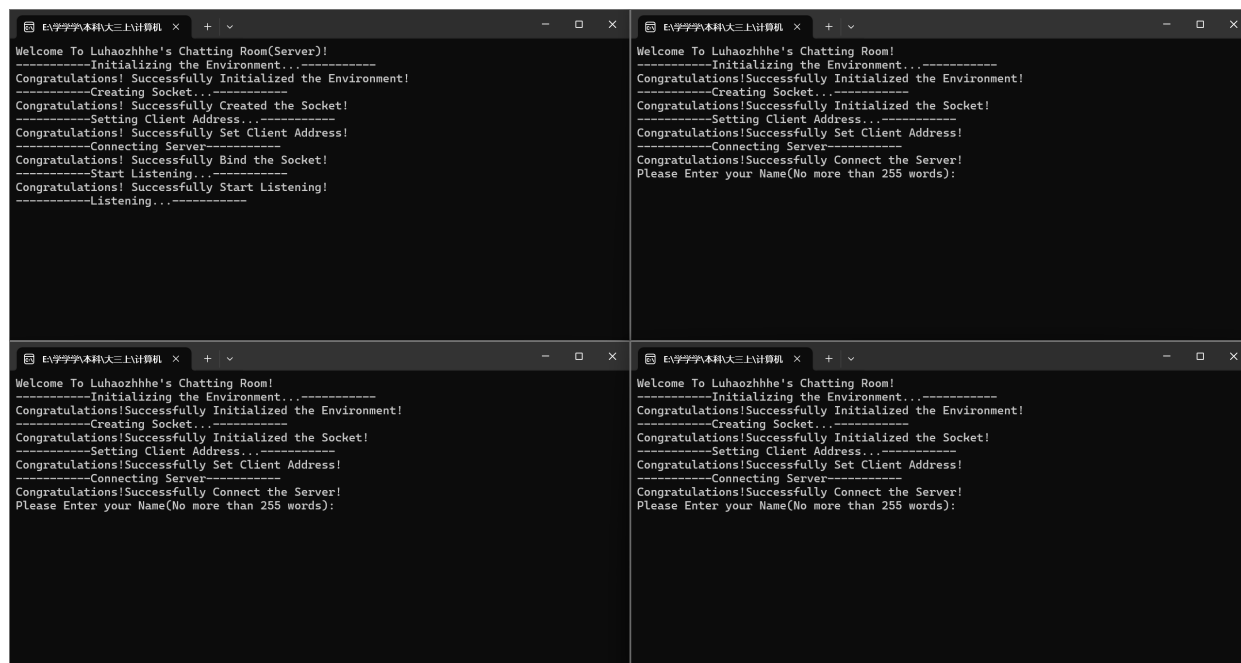


图 6.4: 页面初始化

首先，我们分别输入三个用户的用户名 (如图6.5所示)。

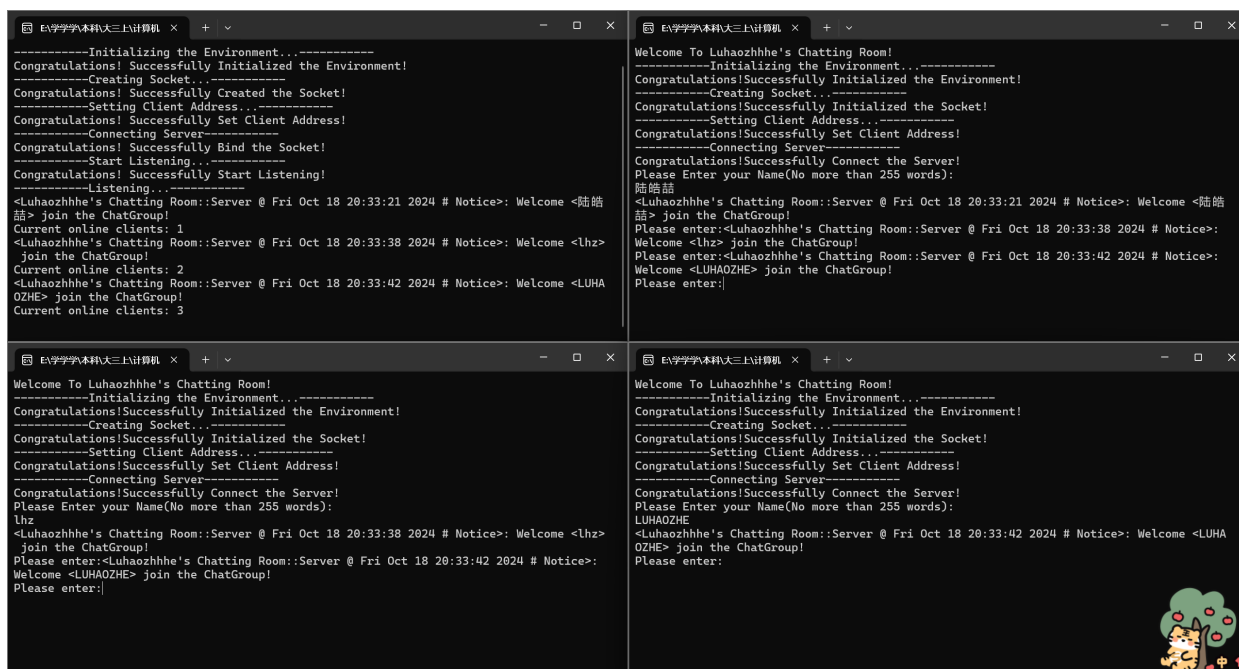


图 6.5: 输入用户名信息

我们发现，英文的大写、小写以及中文的输入均可以正常输入。在导入三个用户后，server 端的用户在线总人数变成了 3，符合我们的实际情况。

然后我们开始进行通信。我们分别在各个用户界面输入中文和英文，发现均可以正常传输，而且信息可以同步到所有人的页面中，包括 server 端，如图 6.6 所示。

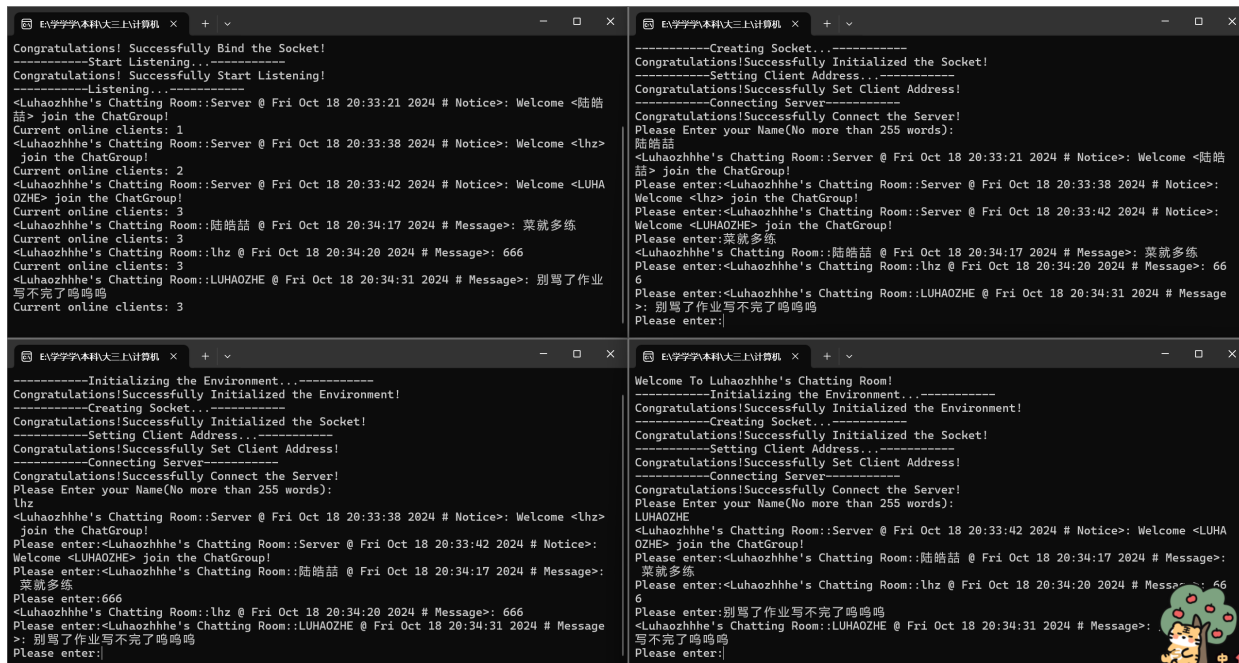


图 6.6: 三方通信

### 6.3 退出程序演示

所有的输入环节都已经验证完了。我们现在开始通过设计的 QUIT 来退出程序。首先，退出一个账号，发现该进程关闭，同时所有界面都会显示，该用户退出聊天；server 端的总人数也会减少 1，如图6.7所示。

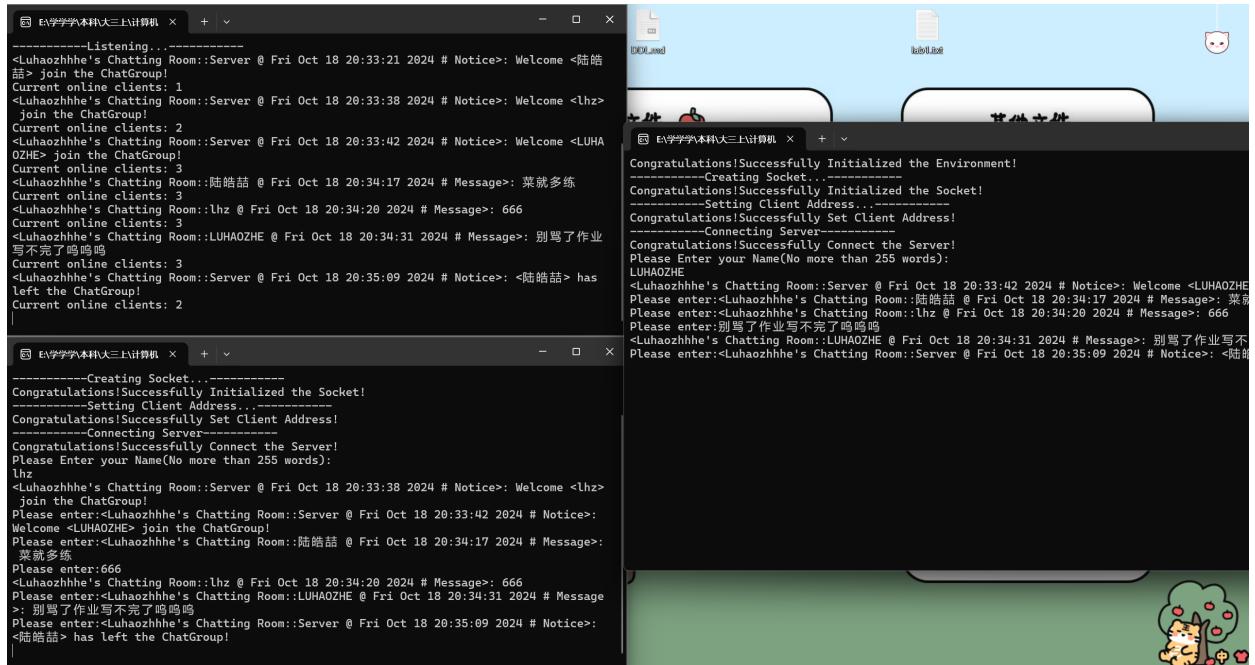


图 6.7: 一方退出

然后我们按照一样的步骤进行第二个用户的退出。和前面所说的一样，总用户减少到 1，同时关闭了第二个用户的进程，如图6.8所示。

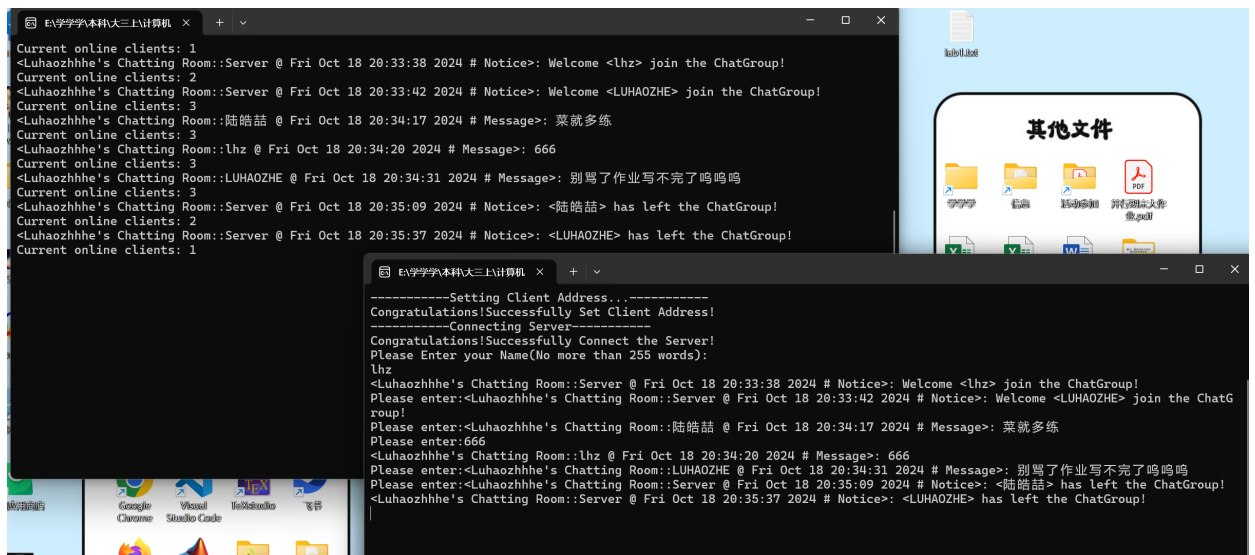


图 6.8: 双方退出

然后进行最后一位用户的退出，和前面的操作是一样的，如图6.9所示。可以发现总人数减少为 0。

```
Current online clients: 1
<Luhaozhhe's Chatting Room::Server @ Fri Oct 18 20:33:38 2024 # Notice>: Welcome <lhz> join the ChatGroup!
Current online clients: 2
<Luhaozhhe's Chatting Room::Server @ Fri Oct 18 20:33:42 2024 # Notice>: Welcome <LUHAOZHE> join the ChatGroup!
Current online clients: 3
<Luhaozhhe's Chatting Room::陆皓喆 @ Fri Oct 18 20:34:17 2024 # Message>: 菜就多练
Current online clients: 3
<Luhaozhhe's Chatting Room::lhz @ Fri Oct 18 20:34:20 2024 # Message>: 666
Current online clients: 3
<Luhaozhhe's Chatting Room::LUHAOZHE @ Fri Oct 18 20:34:31 2024 # Message>: 别骂了作业写不完了呜呜呜
Current online clients: 3
<Luhaozhhe's Chatting Room::Server @ Fri Oct 18 20:35:09 2024 # Notice>: <陆皓喆> has left the ChatGroup!
Current online clients: 2
<Luhaozhhe's Chatting Room::Server @ Fri Oct 18 20:35:37 2024 # Notice>: <LUHAOZHE> has left the ChatGroup!
Current online clients: 1
<Luhaozhhe's Chatting Room::Server @ Fri Oct 18 20:36:26 2024 # Notice>: <lhz> has left the ChatGroup!
Current online clients: 0
```

图 6.9: 全部退出

## 6.4 附加功能演示

我们下面演示一下我们设计的附加功能。

首先是随机用户名的生成。我们取用户名时，在客户端输入换行符，可以看到系统直接帮我们生成了一个随机的名字，如图6.10所示。

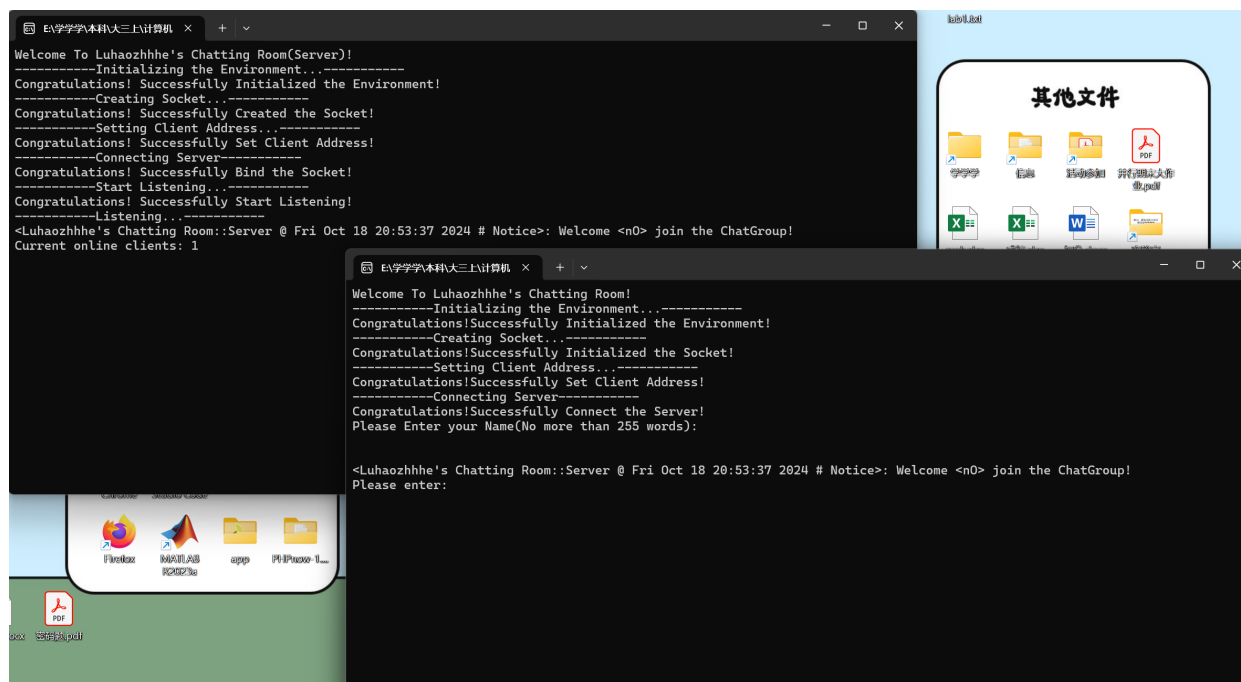


图 6.10: 用户名随机生成

该功能验证完毕！



下面我们进行 client 断连后重连的测试。首先我们打开 server 和 client，正常输入用户名。然后我们关闭 server 的进程，发现 client 端发生异常，系统提示按下 enter 尝试重连。我们再打开一个 server 端，在 client 端按下回车，发现成功重连！（如图6.11所示）

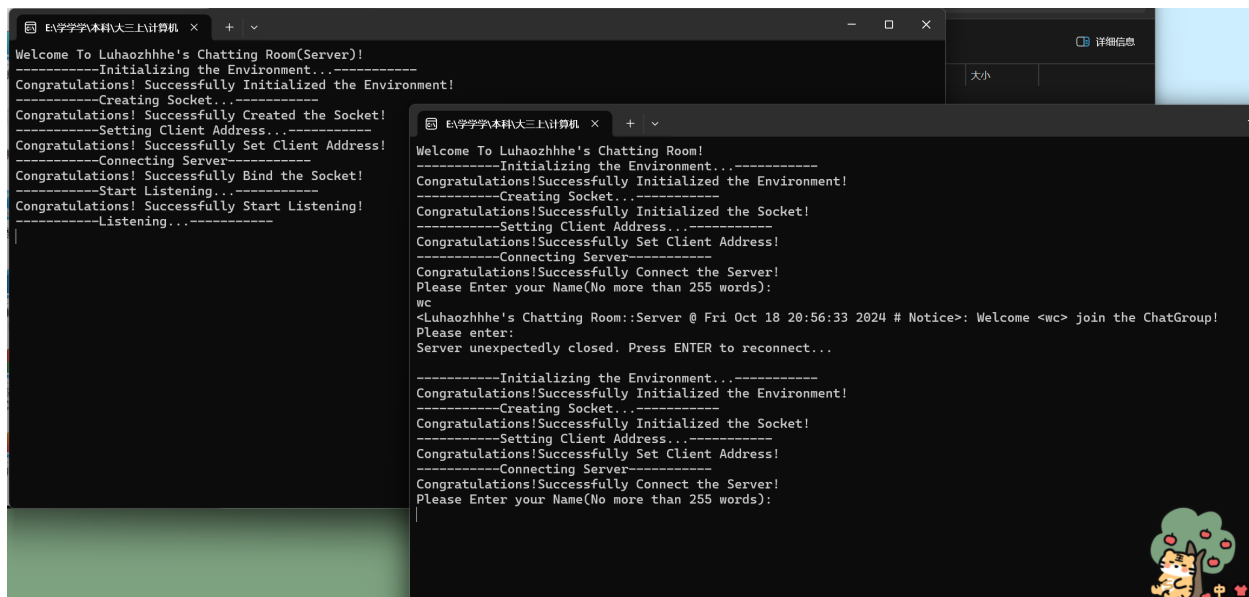


图 6.11: 断连后重连

可以看到，首先，客户端出现了 server 关闭的通知，并让我们按下 enter 键进行重连。enter 后，成功完成了连接操作，验证成功！

综上所述，我们的所有功能都验证完毕了!!! 实验圆满收官!!!

## 7 遇到的问题 & 解决方案

在这一部分，我就简单说一说我在编程过程中遇到的问题和挑战。

首先第一个就是对于用户数量的控制。刚开始我并没有使用动态容器加上互斥锁进行控制，发现会出很多问题，包括用户数量溢出，用户退出后不能更新状态等等。后来，我使用了 vector 容器就解决了这个问题。

第二个问题就是，对于客户端在线数量的统计，刚开始我以为只需要对用户的数量进行加减即可，但是出现了很多错误。后续发现，原因是因为我们在创建用户的时候已经对容器做了操作，所以我们在清除对应的用户时，不仅仅要把数量减一，还需要将 vector 中的用户数据也一起清除了。

我展示一下对应的代码：

```
1  if (log == 0) {
2      {
3          lock_guard<mutex> lock(Clients_Mutex);
4          clientCount--; // 减少客户端计数
5          Clients.erase(remove_if(Clients.begin(), Clients.end(), [recv_temp](const
           ↪ Client& client) { return client.getClientSocket() == recv_temp; })),
           ↪ Clients.end());
6      }
```



```
7     string str_time = Get_Formatted_Time();
8     str = "<Luhaozhhe's Chatting Room::Server @ " + str_time + " # Notice>: <" +
        ↵ username + "> has left the ChatGroup!";
9     Log_And_Broadcast(str);
10    break;
11 }
```

我们在对 clientCount 做减一操作的时候，还需要使用 Clients 的 erase 函数去完成对应字段的清除工作，此处调试了很久，最后才通过了对应的检测。

还有很多问题，由于时间关系，我都没有去完成实现，我在此处做一些说明：

- 首先，我只完成了对用户输入用户名的长度的检测，而没有完成对用户输入字段的溢出检测；
- 我没有实现用户动态进出中的数量限制，也就是说，vector 容器可能会溢出，但是我的监听数量不会溢出，因为我设置了对应的上限值；
- 本来还想做一个通过 ls 来查看当前在线的所有用户的名称和聊天记录，但是由于时间关系，实在来不及做了，但是大致已经构建出了对应的思路；
- 还有一个 bug，就是程序可以通过 QUIT 来进行退出，但是如果我直接关闭终端，server 界面就会变成死循环地输出日志，此处暂时还没解决。

综上所述，该程序还是会存在不少的漏洞，有些漏洞虽然不是致命的，但是会影响观感。但是总体来说，我很好地完成了所有基础要求，并做了一定的扩展和延伸。自认为本次实验的完成还是较好的。

## 8 心得体会

第一次使用 c++ 中的 socket 进行编程，我在刚开始也遇到了很多的困难。本人的代码能力不算很强，所以这个实验也是边写边改，很多时候都会出现，加了一个功能，导致出现十个 bug 这样的情况。但是我最后还是成功地完成了该项目，实现了从 server 端到 client 端的通信框架。

总的来说，本次实验让我大致掌握了 socket 的编程方法，熟练掌握了各类 socket 函数的使用，并且自己在程序中也编写了很多的函数块来帮助我们完成实验。在用户数量的实现上，我采用了动态容器和互斥锁来保证我们的通信可靠性，在用户名检测时，通过使用栈溢出保护，来限制输入的字符串长度。

综上所述，本次实验让我受益匪浅。我会继续持续学习有关 socket 方面的编程技巧，再对我的实验内容进行进一步的改进。感谢老师课上的教授以及助教学长的指导!!!