



南开大学
Nankai University

南 开 大 学

计 算 机 学 院

计算机网络实验报告

实验一：利用流式套接字编写聊天程序

查科言

年级：2023 级

专业：计算机科学与技术

指导教师：徐敬东 张建忠

2025 年 11 月 22 日

目录

一、 实验要求	3
二、 gitHub 仓库	3
三、 实验原理	3
(一) TCP/IP 协议	3
(二) Socket 编程	3
(三) Socket 常用函数	4
1. 初始化与清理 (Windows 特有)	4
2. 套接字创建与配置	4
3. 连接与数据传输	5
四、 协议设计	7
(一) 总体设计	7
1. 帧结构定义	7
2. 消息交互流程	7
3. 核心保障机制	8
(二) Server.cpp 核心流程	8
1. 客户端接入流程	8
2. 消息转发流程	8
3. 客户端退出流程	8
(三) Client 端协议	9
1. 服务器连接流程	9
2. 消息发送流程	9
3. 消息接收流程	9
4. 客户端退出流程	9
(四) 退出程序机制	9
1. 客户端正常退出	9
2. 客户端异常退出	10
3. 服务器退出	10
五、 功能实现 & 代码分析	10
(一) Sever.cpp	10
1. 网络初始化与清理函数	10
2. 数据可靠收发函数	11
3. 协议帧处理函数	12
4. JSON 处理函数	12
5. 客户端管理与广播函数	14
6. 主函数 (程序入口)	15
(二) Client.cpp	17
1. 字符编码转换函数	17
2. 网络数据收发的底层支撑函数	18
3. 协议帧的封装与解析函数	18
4. JSON 处理工具函数	19

5.	连接管理函数	20
6.	界面布局函数	22
7.	程序入口函数	22
六、 结果展示		23
(一)	只有 client 运行	23
(二)	运行 sever 和 client	24
1.	单人聊天	24
2.	多人聊天	25
3.	离开聊天区	26
七、 心得体会		27

一、 实验要求

- 设计聊天协议，并给出聊天协议的完整说明
- 利用 C 或 C++ 语言，使用基本的 Socket 函数进行程序编写，不允许使用 CSocket 等封装后的类
- 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
- 完成的程序应能支持英文和中文聊天
- 采用多线程，支持多人聊天
- 编写的程序应结构清晰，具有较好的可读性
- 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、 gitHub 仓库

本次实验的完整代码（客户端 client.cpp、服务器端 server.cpp）、编译脚本、可执行文件及本实验报告已上传至个人 GitHub 仓库，可通过以下链接访问，便于查阅与复用：

<https://github.com/Ke-yyan/Computer-Internet-/tree/main/Lab1>

三、 实验原理

（一） TCP/IP 协议

TCP/IP 协议族是互联网通信的核心协议集合，包含应用层、传输层、网络层和数据链路层四层结构。本实验的 Socket 编程主要依赖以下两层协议：

- **传输层 (TCP)**：选择 TCP（传输控制协议），具备**面向连接、可靠传输、字节流服务**三大特性。通过“三次握手”建立连接、“四次挥手”释放连接，确保聊天消息**不丢失、不重复、按序到达**；通过滑动窗口实现流量/拥塞控制，为**多人聊天**的稳定消息传输提供保障，规避 UDP 可能出现的丢包问题。
- **应用层 (自定义协议)**：采用 **4 字节长度前缀 + UTF-8 JSON** 的聊天协议，统一封装 type（如 join、msg、quit、sys）、from（昵称）、text（消息内容）等字段，便于客户端/服务器按一致规则解析与处理。

综上，TCP/IP 为 Socket 提供跨网络的端到端传输能力，客户端与服务器基于“IP 地址 + 端口”建立连接，这是远程多人聊天的基础。

（二） Socket 编程

我们在课上已经知道，Socket（套接字）是应用层与 TCP/IP 协议族交互的抽象，屏蔽底层协议如 TCP 握手、挥手、重传等，向应用暴露简洁 API，就可以完成网络数据收发。我们在实验中采用 C/S（客户端-服务器）模型，核心流程如下：

1. **服务器端流程**: `socket` 创建监听套接字 → `bind` 绑定 IP/端口 → `listen` 进入监听 → `accept` 接受连接并为每个客户端创建独立线程 (`handle_client`) → 线程内用 `recv_frame` 接收消息, 解析后调用 `broadcast` 广播给所有在线客户端 → 断开时 `closesock` 关闭套接字并释放资源。
2. **客户端流程**: `WSAStartup` 初始化 WinSock → `socket` 创建套接字 → `connect` 连接服务器 → 读取 GUI 中的昵称并发送 `join` → 创建接收线程 (`RxLoop`) 持续 `recv_frame` 获取广播 → 用户输入后用 `send_frame` 发送消息 → 退出时 `DoDisconnect` 关闭套接字并清理。

归纳而言, 我们可以知道 Socket 本质上是“网络通信的文件描述符”。双方基于该描述符完成跨进程、跨设备的数据交互, 是聊天功能的核心载体。

(三) Socket 常用函数

在课上我们学习了一系列的 Socket 的常用函数, 比如: `WSAStartup`, `WSACleanup`, `bind`, `listen`, `connect`, `accept`, `sendto`, `send`, `recvfrom`, `closesocket`, 我接下来对重要的一些函数进行分析和讲解。

1. 初始化与清理 (Windows 特有)

WSAStartup `WSAStartup` 是 Windows Socket 编程的初始化函数, 其核心功能是初始化 Socket DLL 并协商使用的 Socket 版本, 在调用其他任何 Socket 函数前必须先执行该函数。

它的参数中, `wVersionRequested` 指定调用者希望使用的最高 Socket 版本, 而 `lpWSAData` 作为存储可用 Socket 详细信息结构指针, 会返回推荐使用的版本号 `wVersion` 和系统支持的最高版本号 `wHighVersion`。

该函数成功调用时返回 0, 失败则返回错误代码, 且需注意调用成功后, 在不再使用 Socket 资源时必须通过 `WSACleanup` 释放资源。

```
1 int WINAPI WSAStartup(WORD wVersionRequested, LPWSADATA lpWSADATA);
```

WSACleanup `WSACleanup` 的功能明确, 用于结束 Socket 的使用并释放 Socket DLL 所占用的资源, 是与 `WSAStartup` 配套的收尾函数。

其返回值为 0 时表示调用成功, 若调用失败, 可通过 `WSAGetLastError` 函数获取具体的错误信息, 以排查问题。

```
1 int WINAPI WSACleanup();
```

2. 套接字创建与配置

socket `socket` 函数用于创建一个 Socket 实例, 并将其绑定到特定的传输层服务。

它的三个关键参数分别对应地址类型、服务类型和协议: `af` 指定地址族, 如 `AF_INET` 对应 IPv4, `AF_INET6` 对应 IPv6; `type` 定义服务类型, 例如 `SOCK_STREAM` 代表流式服务 (对应 TCP)、`SOCK_DGRAM` 代表数据报服务 (对应 UDP); `protocol` 指定具体协议, 如 `IPPROTO_TCP` 和 `IPPROTO_UDP` 分别对应 TCP 和 UDP 协议, 若设为 0 则由系统根据前两个参数自动选择协议。

函数调用成功时返回 Socket 描述符, 失败则返回 `INVALID_SOCKET`, 此时可通过 `WSAGetLastError` 获取错误详情。

```
1 SOCKET WSAAPI socket(  
2     int af,  
3     int type,  
4     int protocol);
```

bind bind 函数的作用是将本地地址（包含 IP 地址和端口号）与指定的 Socket 绑定，建立 Socket 和本地网络地址的关联。

其参数中，s 为目标 Socket 的描述符；addr 是存储本地地址信息的结构，若将其中的 IP 地址设为 INADDR_ANY (IPv4) 或 in6addr_any (IPv6)，则由系统自动分配可用的本地 IP 地址；namelen 用于指定地址结构的长度。

该函数返回 0 表示绑定成功，返回 SOCKET_ERROR 表示失败，错误信息可通过 WSAGetLastError 获取。

```
1 int bind(  
2     SOCKET s,  
3     const socketaddr *addr,  
4     int namelen);
```

listen listen 是仅用于 TCP 服务端的函数，功能是使指定的 Socket 进入监听状态，持续监测是否有远程客户端的连接请求。

它的两个参数中，s 是需要设置为监听状态的 Socket 描述符，backlog 则指定连接等待队列的最大长度，即最多能同时处于等待状态的连接请求数量。

函数调用成功返回 0，失败返回 SOCKET_ERROR，错误详情可通过 WSAGetLastError 查询。

```
1 int WSAAPI listen(  
2     SOCKET S,  
3     int backlog);
```

3. 连接与数据传输

connect connect 是 TCP 客户端专用函数，用于向指定的远程 Socket 发起连接请求，以建立客户端与服务端之间的 TCP 连接。

其参数中，s 为客户端创建的 Socket 描述符；name 存储目标服务端的地址信息，包含服务端的 IP 地址和端口号；namelen 为地址结构的长度。函数返回 0 说明连接建立成功，返回 SOCKET_ERROR 表示连接失败，可通过 WSAGetLastError 获取具体错误原因。

```
1 int WSAAPI connect(  
2     SOCKET S,  
3     const sockaddr *addr,  
4     int namelen);
```

accept accept 是 TCP 服务端用于接受连接的函数，专门处理处于监听状态的 Socket 的连接等待队列中的连接请求。

其参数中，s 是处于监听状态的 Socket 描述符；addr 用于存储发起连接的远程客户端的地址信息；addrlen 则用于存储地址结构的长度。

该函数调用成功时，会返回一个新的 Socket 描述符，后续与该客户端的通信均通过此新描述符进行；失败则返回 `INVALID_SOCKET`，错误信息可通过 `WSAGetLastError` 获取。需要注意的是，该函数通常处于阻塞状态，会一直等待直到有新的连接请求到来。

```
1 SOCKET WINAPI accept(  
2     SOCKET s,  
3     sockaddr *addr,  
4     int *addrlen);
```

send `send` 主要用于 TCP 等流式传输场景，用于向已建立连接的远程 Socket 发送数据。

其参数包括：`s` 为已连接的 Socket 描述符；`buf` 是存储待发送数据的缓冲区；`len` 为缓冲区中待发送数据的长度；`flags` 用于指定调用的处理方式，一般情况下设为 0。

函数成功调用时返回实际发送的字节数，由于 TCP 的流式特性，实际发送字节数可能小于待发送长度，需注意处理；调用失败则返回 `SOCKET_ERROR`，错误信息可通过 `WSAGetLastError` 获取。

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

sendto `sendto` 主要用于 UDP 等数据报式传输的场景，功能是向指定的目标地址发送数据。

其参数较为丰富：`s` 为用于发送数据的 Socket 描述符；`buf` 是存储待发送数据的缓冲区；`len` 指定缓冲区中待发送数据的长度；`flags` 用于设置调用的处理方式，通常设为 0 即可；`to` 存储目标 Socket 的地址信息；`tolen` 为目标地址结构的长度。

函数调用成功时返回实际发送的字节数，失败则返回 `SOCKET_ERROR`，可通过 `WSAGetLastError` 获取错误详情。

```
1 int WINAPI sendto(  
2     SOCKET s,  
3     const char *buf,  
4     int len,  
5     int flags,  
6     const sockaddr *to,  
7     int tolen);
```

recvfrom `recvfrom` 主要用于 UDP 等数据报式传输场景，用于从指定的源地址接收数据。

其参数中，`s` 为用于接收数据的 Socket 描述符；`buf` 是存储接收数据的缓冲区；`len` 指定缓冲区的长度；`flags` 用于设置调用处理方式，通常为 0；`from` 用于存储发送数据的源 Socket 地址信息；`fromlen` 为源地址结构的长度。

函数调用成功时返回实际接收到的字节数，失败则返回 `SOCKET_ERROR`，可通过 `WSAGetLastError` 获取错误详情，该函数也常处于阻塞状态，等待数据接收。

```
1 int WINAPI recvfrom(  
2     SOCKET s,  
3     char *buf,
```

```
4     int len ,
5     int flags ,
6     sockaddr *from ,
7     int *fromlen);
```

为解决 TCP 粘/拆包并简化上层逻辑，代码封装了如下工具：

- **send_all/recv_n**: 保证“写满/读满”指定长度。由于 send/recv 可能只处理部分字节，这两个函数通过循环调用直至完成或出错，避免数据截断。
- **send_frame/recv_frame**: 实现“4B 长度前缀 + 负载”分帧。前缀用网络字节序(htonl/ntohl)，先发/收 4 字节长度，再发/收 JSON 负载，从根源上解决粘/拆包。
- **broadcast** (服务器): 遍历在线客户端，逐个用 send_frame 广播；若发送失败，说明连接失效，立即关闭并从在线表剔除，保持系统健壮。

四、 协议设计

(一) 总体设计

本实验的协议设计核心目标是解决 TCP 粘包问题与统一客户端-服务器消息格式，采用“4 字节长度前缀 + UTF-8 JSON”的分帧结构，基于 TCP 传输协议、流式套接字实现，采用多线程方式处理并发，具体设计如下：

1. 帧结构定义

TCP 为字节流协议，若直接发送 JSON 数据可能出现“粘包”（多个消息合并）或“拆包”（单个消息拆分），因此设计固定帧结构如下，确保接收端能准确截取完整消息：

[长度前缀(4字节)][JSON数据(n字节)]

长度前缀：无符号 32 位整数 (uint32_t)，采用网络字节序（大端序）存储，通过 htonl 函数将主机字节序转换为网络字节序，接收端通过 ntohl 转换回主机字节序，该字段表示后续 JSON 数据的字节数；

JSON 数据：UTF-8 编码的字符串，封装消息的核心字段（type 消息类型、from 发送者昵称、text 消息内容），具备可读性与扩展性，支持中文与特殊字符（通过 json_escape 函数转义）；

约束条件：单条消息最大长度限制为 1MB (1u<<20 字节)，避免超大消息占用过多网络资源或导致内存溢出，recv_frame 函数中会校验该条件，超过则返回失败。

2. 消息交互流程

- **启动服务器**：服务器绑定 5000 端口（默认，可通过命令行参数指定），进入监听状态，等待客户端连接；
- **客户端连接**：客户端输入服务器 IP（默认 127.0.0.1）、端口（默认 5000）与昵称，点击“加入”按钮，与服务器建立 TCP 连接，发送 join 类型消息；
- **消息广播**：服务器接收客户端消息后，解析消息类型，若为 msg 普通消息则广播给所有在线客户端，若为 join 或者 quit 则生成 sys 系统通知并广播；

- **客户端接收：**客户端独立线程（RxLoop）持续接收服务器消息，解析后更新图形界面的聊天日志区；
- **退出流程：**客户端点击“离开”或输入 /quit，发送 quit 类型消息，服务器广播退出通知并清理资源。

3. 核心保障机制

线程安全：服务器端通过 mutex 互斥锁（g_mu）保护共享的客户端列表（g_clients），避免多线程并发修改导致的容器异常；

编码兼容：客户端通过 utf8_to_wide/wide_to_utf8 函数实现 UTF-8 与 Win32 宽字符（UTF-16）的转换，支持中文消息无乱码；

错误处理：网络操作（如 connect、send_frame、recv_frame）失败时，客户端提示错误信息（如“连接被关闭”），服务器自动剔除失效客户端，确保系统稳定性。

（二） Server.cpp 核心流程

服务器端核心功能为管理客户端连接与消息广播，围绕“接入-转发-退出”三阶段设计，确保多客户端并发时的稳定性与正确性。

1. 客户端接入流程

服务器监听 5000 端口，通过 accept 函数阻塞等待客户端连接。每接收到一个连接请求，便创建 handle_client 线程单独处理该客户端，主线程则继续等待新连接。

客户端连接后，会发送 join 类型的 JSON 消息（如 {"type":"join","from":"user1"}），服务器通过 recv_frame 接收并解析该消息。若消息中 type 字段非 join 或 from 字段为空，服务器会自动为客户端分配默认昵称“guest”。

之后，服务器加互斥锁（lock_guard<mutex>）保护客户端列表，将该客户端的套接字与昵称存入 vector<ClientInfo> 容器完成注册。最后，生成 sys 类型的系统通知（如“user1 进入聊天区”），通过 broadcast 函数广播给所有在线客户端，同时在服务器控制台打印该通知。

2. 消息转发流程

当客户端发送 msg 类型的消息（如 {"type":"msg","from":"user1","text":" 大家好"}）时，服务器线程通过 recv_frame 接收消息。

调用 get_field 函数提取 type、from、text 字段后，校验消息合法性：若为 msg 类型需确保文本非空，若为 quit 类型则触发客户端退出流程。

校验通过后，broadcast 函数遍历客户端列表，通过 send_frame 向所有在线客户端转发消息。若发送失败（表示客户端连接失效），则关闭对应套接字并从列表中移除，确保客户端列表的准确性。同时，服务器控制台会同步打印消息内容与发送者信息，便于管理员查看。

3. 客户端退出流程

服务器检测到客户端退出的场景分为两种：主动退出和异常断开。

主动退出时，客户端发送 quit 消息，handle_client 函数解析后跳出消息循环；异常断开时，recv_frame 接收失败，触发退出逻辑。

退出流程中，服务器先加互斥锁保护共享资源，从 `vector<ClientInfo>` 容器中删除该客户端信息；随后生成 `sys` 类型的通知（如“user1 离开聊天区”）并广播给所有在线客户端；最后关闭该客户端对应的套接字，释放线程资源，避免内存泄漏。

（三） Client 端协议

客户端基于 Win32 API 实现图形界面，协议流程与服务器端对齐，核心涵盖连接服务器、收发消息及退出机制，确保确保与服务器端交互的兼容性。

1. 服务器连接流程

客户端启动后，首先初始化 Winsock 库并创建 TCP 套接字，若初始化或创建失败，将通过 Win32 弹窗提示错误。

用户在图形界面输入服务器 IP（默认 127.0.0.1）、端口（默认 5000）与昵称，点击“加入”按钮触发 `DoConnect` 函数：校验昵称非空，解析 IP 和端口，若失败则进行相应提示；

调用 `connect` 建立连接，成功后发送 `join` 类型 JSON 消息；

启动 `RxLoop` 接收线程，并更新界面状态——将按钮文本改为“离开”，禁用 IP、端口和昵称输入框，启用发送按钮。

2. 消息发送流程

用户在消息输入框输入内容后，点击“发送”按钮或按 Enter 键，触发 `DoSend` 函数。

该函数先校验消息非空，若输入内容为 `/quit` 则触发退出流程；随后将输入内容转换为 UTF-8 编码，封装为 `msg` 类型 JSON 消息，通过 `send_frame` 发送给服务器；若发送失败则断开连接，发送成功则清空输入框，等待下一次输入。

3. 消息接收流程

独立的接收线程 `RxLoop` 会循环接收消息，直至连接关闭。

线程调用 `recv_frame` 接收服务器发送的消息，若接收失败，会通过 `PostMessageW` 通知 UI 线程“连接被关闭”；解析消息时，根据 `type` 字段处理——`msg` 类型消息以“发送者：内容”格式显示，`sys` 类型消息直接显示文本内容；

最后通过 `PostMessageW` 通知 UI 线程更新聊天日志，确保 UI 操作在主线程执行，避免跨线程操作异常。

4. 客户端退出流程

客户端退出分为主动退出和异常断开两种场景：主动退出时，客户端发送 `quit` 消息，`handle_client` 函数解析后跳出消息循环；

异常断开时，`recv_frame` 返回失败触发退出逻辑。退出过程中，会加互斥锁保护共享资源，从客户端列表中删除该客户端信息，生成并广播“离开”通知，同时关闭套接字以释放线程资源。

（四） 退出程序机制

1. 客户端正常退出

客户端正常退出的触发方式是点击“离开”按钮或输入 `/quit`，调用 `DoDisconnect`；我们标记连接关闭，发送 `quit` 消息，关闭套接字并等待接收线程结束，恢复界面状态。

2. 客户端异常退出

在异常情况客户端退出时，如服务器关闭或网络中断，recv_frame失败；我们采用 UI 线程提示“连接被关闭”，自动调用DoDisconnect释放资源。

3. 服务器退出

- 正常退出：关闭命令行窗口（可扩展quit指令）；
- 异常退出：函数失败时关闭监听套接字，释放 Winsock 资源；
- 客户端同步：检测到连接关闭后触发异常退出流程。

五、 功能实现 & 代码分析

（一） Sever.cpp

1. 网络初始化与清理函数

网络初始化与清理函数是服务器进行网络通信的基础，负责适配不同操作系统的网络环境：

```
1 // 初始化网络环境（跨平台适配）
2 static void net_init() {
3     #ifdef _WIN32
4         WSADATA wsa;
5         WSAStartup(MAKEWORD(2, 2), &wsa); // Windows 需初始化 Winsock 2.2
6     #else
7         // Linux/macOS 无需额外初始化
8     #endif
9 }
10
11 // 清理网络资源
12 static void net_cleanup() {
13     #ifdef _WIN32
14         WSACleanup(); // Windows 释放 Winsock 资源
15     #else
16         // Linux/macOS 无需额外操作
17     #endif
18 }
19
20 // 跨平台关闭套接字
21 static void closesock(socket_t s) {
22     #ifdef _WIN32
23         closesocket(s);
24     #else
25         close(s);
26     #endif
27 }
28
29 // 获取网络错误码
```

```

30 static int last_net_err() {
31 #ifdef _WIN32
32     return WSAGetLastError();
33 #else
34     return errno;
35 #endif
36 }

```

功能说明： - net_init() 在 Windows 系统中通过 WSAStartup 初始化网络库，为套接字操作提供支持；类 Unix 系统因原生支持 POSIX 接口，无需额外初始化。

- net_cleanup() 对应 Windows 平台的 WSACleanup 函数，确保程序退出时释放网络资源，避免泄漏。

- closesock() 和 last_net_err() 分别封装了不同系统下的套接字关闭和错误码获取接口，实现代码跨平台兼容。

2. 数据可靠收发函数

为解决 TCP 协议“字节流无边界”特性导致的分包/粘包问题，实现了底层可靠收发函数：

send_all 函数

```

1 // 确保发送全部字节（应对 TCP 分包）
2 static bool send_all(socket_t s, const char* buf, size_t len) {
3     size_t sent = 0;
4     while (sent < len) {
5         int n = send(s, buf + (int)sent, (int)(len - sent), 0);
6         if (n <= 0) return false; // 发送失败（连接关闭或错误）
7         sent += n;
8     }
9     return true;
10 }

```

send_all(socket_t s, const char* buf, size_t len) 函数是确保数据完整发送的核心支撑。由于 TCP 协议是流式传输，单次 send 调用可能无法发送全部数据（如缓冲区满），因此该函数通过循环调用 send 实现“可靠发送”：每次发送剩余未发送的数据，直到所有 len 字节的数据都被成功发送

若中途 send 返回值小于等于 0（表示发送失败或连接关闭），函数返回 false；若所有数据发送完成，返回 true。这一函数解决了 TCP 分包问题，保证应用层数据的完整性。

recv_n 函数

```

1 // 确保接收固定长度字节（应对 TCP 粘包）
2 static bool recv_n(socket_t s, char* buf, size_t len) {
3     size_t r = 0;
4     while (r < len) {
5         int n = recv(s, buf + (int)r, (int)(len - r), 0);
6         if (n <= 0) return false; // 接收失败
7         r += n;
8     }
9     return true;
10 }

```

对应的 `recv_n(socket_t s, char* buf, size_t len)` 函数则负责可靠接收固定长度的数据。与发送类似, TCP 的 `recv` 调用可能只返回部分数据, 因此该函数通过循环调用 `recv`, 累计接收数据直到达到 `len` 字节。

若中途 `recv` 返回值小于等于 0 (表示接收失败或连接关闭), 返回 `false`; 否则返回 `true`。它解决了 TCP 粘包问题, 确保应用层能准确获取预期长度的数据。

3. 协议帧处理函数

我们基于“4 字节长度前缀 + UTF-8 JSON”的自定义协议, 实现消息帧的封装与解析: `send_frame(socket_t s, const string& payload)` 实现了自定义协议帧的发送。

send_frame 函数

```
1 // 发送帧: 前缀 (网络字节序长度) + payload (JSON)
2 static bool send_frame(socket_t s, const string& payload) {
3     uint32_t L = htonl((uint32_t)payload.size()); // 转换为网络字节序
4     return send_all(s, (char*)&L, 4) && send_all(s, payload.data(), payload.
5         size());
6 }
```

该协议采用“4 字节长度前缀 + UTF-8 JSON”的格式: 首先计算 `payload` (JSON 字符串) 的长度, 通过 `htonl` 函数将长度转换为网络字节序 (大端序), 作为 4 字节前缀发送; 随后调用 `send_all` 发送 `payload` 内容。

这一设计通过长度前缀明确消息边界, 使接收方能够准确拆分连续的字节流。

recv_frame 函数

```
1 // 接收帧: 先读长度前缀, 再读对应长度的 payload
2 static bool recv_frame(socket_t s, string& out) {
3     uint32_t Lnet = 0;
4     if (!recv_n(s, (char*)&Lnet, 4)) return false; // 读取 4 字节长度
5     uint32_t L = ntohl(Lnet); // 转换为主机字节序
6     if (L > (1u << 20)) return false; // 限制最大 1MB, 防止恶意数据
7     out.assign(L, '\0');
8     return recv_n(s, out.data(), L); // 读取 payload
9 }
```

`recv_frame(socket_t s, string& out)` 则对应协议帧的接收过程。它先通过 `recv_n` 读取 4 字节长度前缀, 再用 `ntohl` 将网络字节序转换为本地字节序, 得到 `payload` 的实际长度。为防止恶意数据攻击, 函数会校验长度是否超过 1MB ($1 \ll 20$ 字节), 若超限则返回失败; 否则通过 `recv_n` 读取对应长度的 `payload`, 存入 `out` 中。这一过程确保了接收方能够正确解析出发送方的完整 JSON 消息。

4. JSON 处理函数

JSON 处理函数是协议消息结构化的关键。

json_escape 函数

```
1 // JSON 特殊字符转义
2 static string json_escape(const string& s) {
3     string o; o.reserve(s.size() + 8);
4     for (unsigned char c : s) {
```

```

5         if (c == '\\\' || c == '\') { o.push_back('\\'); o.push_back((char)c);
6             }
7         else if (c == '\n') { o += "\\n"; }
8         else o.push_back((char)c);
9     }
10    return o;
11}

```

json_escape(const string s) 负责对字符串进行 JSON 特殊字符转义，例如在
和” 前添加转义符
，将换行符
n 转换为
n，其他字符则直接保留。这一处理保证了生成的 JSON 字符串格式合法，避免特殊字符破坏
JSON 结构。

make_json 函数

```

1 // 构建 JSON 消息 (type/from/text 三字段)
2 static string make_json(const string& type, const string& from, const string&
3     text) {
4     return string("{\"type\": \"\" + json_escape(type) + "\", \"from\": \"\" +
5         json_escape(from) + "\", \"text\": \"\" + json_escape(text) + \"\"}");
6 }

```

make_json(const string& type, const string& from, const string& text) 函数基于转义后的
字段构建标准 JSON 消息，格式为”type”:...”,”from”:...”,”text”:...””，其中 type 表示消息类型
(如 join、msg、sys)，from 表示发送者，text 表示消息内容。

所有字段均经过 json_escape 处理，确保 JSON 格式的正确性。

get_field 函数

```

1 // 从 JSON 中提取指定字段
2 static string get_field(const string& j, const char* key) {
3     string pat = string("\"") + key + "\"";
4     size_t p = j.find(pat);
5     if (p == string::npos) return "";
6     p += pat.size();
7     string v;
8     for (size_t i = p; i < j.size(); ++i) {
9         char c = j[i];
10        if (c == '\\\' && i + 1 < j.size()) { v.push_back(j[i+1]); ++i; }
11        else if (c == '\') break;
12        else v.push_back(c);
13    }
14    return v;
15}

```

get_field(const string& j, const char* key) 则用于从 JSON 字符串中提取指定字段的值，是一
种简易的 JSON 解析实现。

它通过查找”key”:” 的模式定位字段值的起始位置，然后从该位置开始提取字符，直到遇到
下一个” (字段结束标志)，同时处理转义字符

(如
” 会被解析为”)。这一函数无需依赖完整的 JSON 库，轻量且适配项目的协议需求。

5. 客户端管理与广播函数

实现客户端连接管理与消息广播，是聊天室核心功能：

客户端信息结构体

```
1 // 客户端信息结构体
2 struct ClientInfo { socket_t sock; string nick; };
3 static vector<ClientInfo> g_clients; // 在线客户端列表
4 static mutex g_mu; // 保护客户端列表的互斥锁
```

broadcast 函数

```
1 // 广播消息到所有在线客户端
2 static void broadcast(const string& json) {
3     lock_guard<mutex> lk(g_mu); // 加锁确保线程安全
4     for (auto it = g_clients.begin(); it != g_clients.end(); ) {
5         if (!send_frame(it->sock, json)) { // 发送失败 (连接失效)
6             closesock(it->sock);
7             it = g_clients.erase(it); // 移除无效客户端
8         } else {
9             ++it;
10        }
11    }
12 }
```

broadcast(const string& json) 函数实现消息的全局广播，是聊天室服务器的核心功能之一。

它首先通过 lock_guard<mutex> 加锁 g_mu(全局互斥锁),确保对全局客户端列表 g_clients 的线程安全访问。

随后遍历 g_clients 中的每个客户端，调用 send_frame 发送 json 消息；若发送失败（表示客户端连接已失效），则调用 closesock 关闭该客户端的套接字，并从列表中删除，保证客户端列表的准确性。

这一过程确保了所有在线客户端都能收到广播消息，同时自动清理无效连接。

handle_client 函数

```
1 // 处理单个客户端连接的线程函数
2 static void handle_client(socket_t cs) {
3     // 接收客户端第一条消息 (join 类型)
4     string first;
5     if (!recv_frame(cs, first)) { closesock(cs); return; }
6
7     // 解析昵称，默认为 guest
8     string type = get_field(first, "type");
9     string nick = get_field(first, "from");
10    if (type != "join" || nick.empty()) nick = "guest";
11
12    // 注册客户端并广播加入通知
13    { lock_guard<mutex> lk(g_mu); g_clients.push_back({cs, nick}); }
```

```

14 broadcast(make_json("sys", "server", nick + "进入聊天区"));
15 printf("[sys]_%s_进入聊天区\n", nick.c_str());
16
17 // 消息循环：处理后续消息
18 string frame;
19 while (g_running) {
20     if (!recv_frame(cs, frame)) break; // 接收失败（断开连接）
21     string t = get_field(frame, "type");
22     string txt = get_field(frame, "text");
23     if (t == "msg") { // 转发普通消息
24         broadcast(make_json("msg", nick, txt));
25         printf("[%s]_%s\n", nick.c_str(), txt.c_str());
26     } else if (t == "quit") { // 处理主动退出
27         break;
28     }
29 }
30
31 // 清理客户端并广播离开通知
32 { lock_guard<mutex> lk(g_mu);
33     for (auto it = g_clients.begin(); it != g_clients.end(); ++it) {
34         if (it->sock == cs) { g_clients.erase(it); break; }
35     }
36 }
37 broadcast(make_json("sys", "server", nick + "离开聊天区"));
38 printf("[sys]_%s_离开聊天区\n", nick.c_str());
39 closesock(cs);
40 }

```

handle_client(socket_t cs) 是处理单个客户端连接的线程函数，每个客户端连接会对应一个独立的该线程。

函数首先通过 recv_frame 接收客户端的首条消息，解析其中的 type 和 from 字段：若 type 不是 join（客户端加入消息）或 from 为空（未指定昵称），则自动为客户端分配默认昵称 guest。

接着，函数加锁将客户端信息（套接字 cs 和昵称）存入 g_clients 列表，完成在线注册，并通过 broadcast 发送 sys 类型的“进入聊天区”通知，同时在服务器控制台打印该信息。

之后，函数进入消息循环：持续通过 recv_frame 接收客户端消息，解析 type 字段——若为 msg 类型（普通消息），则调用 broadcast 转发；若为 quit 类型（退出消息）或接收失败（如客户端异常断开），则退出循环。

最后，函数从 g_clients 中移除该客户端，广播“离开聊天区”通知，关闭套接字，释放资源。

6. 主函数（程序入口）

```

1 static bool g_running = true; // 服务器运行状态
2
3 int main(int argc, char** argv) {
4     int port = (argc >= 2) ? atoi(argv[1]) : 5000; // 默认端口 5000
5     net_init(); // 初始化网络环境
6
7     // 创建 TCP 监听套接字

```

```
8     socket_t ls = socket(AF_INET, SOCK_STREAM, 0);
9     if (ls == INVALID_SOCKET) {
10         fprintf(stderr, "socket() failed\n");
11         return 1;
12     }
13
14     // 允许端口复用 (避免重启时端口占用)
15     int yes = 1;
16     setsockopt(ls, SOL_SOCKET, SO_REUSEADDR, (char*)&yes, sizeof(yes));
17
18     // 绑定地址与端口
19     sockaddr_in addr{};
20     addr.sin_family = AF_INET;
21     addr.sin_addr.s_addr = htonl(INADDR_ANY); // 监听所有网卡
22     addr.sin_port = htons((uint16_t)port);
23     if (bind(ls, (sockaddr*)&addr, sizeof(addr)) == SOCKET_ERROR) {
24         fprintf(stderr, "bind() failed: %d\n", last_net_err());
25         closesock(ls);
26         net_cleanup();
27         return 1;
28     }
29
30     // 启动监听 (最大等待连接数 16)
31     if (listen(ls, 16) == SOCKET_ERROR) {
32         fprintf(stderr, "listen() failed\n");
33         closesock(ls);
34         net_cleanup();
35         return 1;
36     }
37     printf("server listening on 0.0.0.0:%d\n", port);
38
39     // 循环接受客户端连接
40     while (g_running) {
41         sockaddr_in cli{};
42         socklen_t cl = sizeof(cli);
43         socket_t cs = accept(ls, (sockaddr*)&cli, &cl); // 阻塞等待连接
44         if (cs == INVALID_SOCKET) {
45             fprintf(stderr, "accept() failed\n");
46             break;
47         }
48         thread(handle_client, cs).detach(); // 创建线程处理客户端
49     }
50
51     // 退出时清理资源
52     closesock(ls);
53     net_cleanup();
54     return 0;
55 }
```

main 函数是服务器的入口，负责初始化并启动服务。它首先解析命令行参数获取端口（默认 5000），调用 `net_init` 初始化网络环境。随后创建 TCP 监听套接字（`SOCK_STREAM` 类型），并设置 `SO_REUSEADDR` 选项（允许端口复用，避免服务器重启时端口被占用）。接着将套接字绑定到本机所有网卡（`INADDR_ANY`）的指定端口，调用 `listen` 启动监听（最大等待连接数为 16），并在控制台输出“监听中”信息。之后进入主循环：通过 `accept` 阻塞等待客户端连接，每接收到一个连接，便创建 `handle_client` 线程处理该客户端，并通过 `detach` 让线程独立运行，主线程则继续等待新连接。当 `g_running`（全局运行状态标记）为 `false` 时，循环退出，关闭监听套接字，调用 `net_cleanup` 清理资源，程序结束。

整体而言，这些函数相互配合，实现了一个支持多客户端并发连接、消息广播、加入 / 离开通知的简易 TCP 聊天室服务器，我们通过自定义协议帧解决了 TCP 的粘包 / 分包问题，通过多线程和互斥锁保证了并发处理的安全性。

（二） Client.cpp

我们的代码实现了一个基于 Win32 API 和 Winsock 的简易聊天室客户端程序，采用 C++17 标准编写，可在 Windows 系统下运行。程序开头通过宏定义指定了 Unicode 编码和精简的 Win32 头文件，随后包含了网络通信（`winsock2.h`、`ws2tcpip.h`）、窗口界面（`windows.h`）以及多线程、原子变量等必要的头文件，并链接了 `Ws2_32.lib` 库以支持网络功能。

1. 字符编码转换函数

字符编码转换是程序的基础功能之一，`utf8_to_wide` 和 `wide_to_utf8` 函数分别实现了 UTF-8 与宽字符（UTF-16）之间的转换，这是因为网络传输通常使用 UTF-8 编码，而 Win32 API 的界面函数多使用宽字符，两者的转换确保了文本在显示和传输时的一致性，避免乱码。

```
1 // UTF-8转宽字符（用于将网络接收的UTF-8数据显示到Win32界面）
2 static wstring utf8_to_wide(const string& s) {
3     if (s.empty()) return L"";
4     int n = MultiByteToWideChar(CP_UTF8, 0, s.c_str(), (int)s.size(), nullptr,
5     , 0);
6     wstring w(n, L'\0');
7     MultiByteToWideChar(CP_UTF8, 0, s.c_str(), (int)s.size(), &w[0], n);
8     return w;
9 }
10 // 宽字符转UTF-8（用于将界面输入的宽字符转换为网络传输格式）
11 static string wide_to_utf8(const wstring& w) {
12     if (w.empty()) return "";
13     int n = WideCharToMultiByte(CP_UTF8, 0, w.c_str(), (int)w.size(), nullptr,
14     , 0, nullptr, nullptr);
15     string s(n, '\0');
16     WideCharToMultiByte(CP_UTF8, 0, w.c_str(), (int)w.size(), &s[0], n,
17     nullptr, nullptr);
18     return s;
19 }
```

2. 网络数据收发的底层支撑函数

`send_all` 和 `recv_n` 是 TCP 数据可靠传输的基础，确保完整发送/接收指定长度的字节流，是 `send_frame` 和 `recv_frame` 的核心依赖：

```

1 // 确保发送全部字节（应对TCP分包）
2 static bool send_all(SOCKET s, const char* buf, size_t len) {
3     size_t sent = 0;
4     while (sent < len) {
5         int n = send(s, buf + (int)sent, (int)(len - sent), 0);
6         if (n <= 0) return false;
7         sent += n;
8     }
9     return true;
10 }
11
12 // 确保接收固定长度字节（应对TCP粘包）
13 static bool recv_n(SOCKET s, char* buf, size_t len) {
14     size_t r = 0;
15     while (r < len) {
16         int n = recv(s, buf + (int)r, (int)(len - r), 0);
17         if (n <= 0) return false;
18         r += n;
19     }
20     return true;
21 }

```

作用：解决 TCP 协议的“字节流无边界”特性，保证应用层数据的完整性（例如发送 4 字节长度前缀时，需确保这 4 字节被完整接收）。

3. 协议帧的封装与解析函数

`send_frame` 和 `recv_frame` 实现“4 字节长度前缀 + UTF-8 JSON”的自定义协议格式，是客户端与服务器通信的核心协议载体：

```

1 // 发送帧：前缀（网络字节序长度）+ payload (JSON)
2 static bool send_frame(SOCKET s, const string& payload) {
3     uint32_t L = htonl((uint32_t)payload.size()); // 转换为网络字节序
4     return send_all(s, (char*)&L, 4) && send_all(s, payload.data(), payload.
5         size());
6 }
7
8 // 接收帧：先读长度前缀，再读对应长度的payload
9 static bool recv_frame(SOCKET s, string& out) {
10     uint32_t Lnet = 0;
11     if (!recv_n(s, (char*)&Lnet, 4)) return false; // 读取4字节长度
12     uint32_t L = ntohl(Lnet); // 转换为主机字节序
13     if (L > (1u << 20)) return false; // 限制最大1MB，防止恶意数据
14     out.assign(L, '\0');
15     return recv_n(s, out.data(), L); // 读取payload

```

15 }

作用：定义客户端与服务器之间的“消息边界”，确保 JSON 数据被完整解析（例如服务器广播的消息需通过此函数正确拆分）。

4. JSON 处理工具函数

为了规范消息格式，我们的程序实现了简易的 JSON 处理功能。json_escape 函数用于对特殊字符（如引号、反斜杠、换行符）进行转义，避免破坏 JSON 格式；make_json 函数根据消息类型、发送者和内容构建 JSON 字符串；get_field 函数则从 JSON 字符串中提取指定字段的值，实现了简单的解析功能，这些函数共同保证了客户端与服务器之间消息的结构化传输。

```

1 // 对JSON特殊字符转义（如”、\、换行）
2 static string json_escape(const string& s) {
3     string o; o.reserve(s.size() + 8);
4     for (unsigned char c : s) {
5         if (c == '\\' || c == '"') { o.push_back('\\'); o.push_back((char)c);
6             }
7         else if (c == '\n') { o += "\\n"; }
8         else o.push_back((char)c);
9     }
10    return o;
11}
12
13 // 构建JSON格式消息（type/from/text三字组）
14 static string make_json(const string& type, const string& from, const string&
15     text) {
16     return string("{\"type\":\"" + json_escape(type) + "\",\"from\":\"" +
17         json_escape(from) + "\",\"text\":\"" + json_escape(text) + "\"}");
18 }
19
20 // 从JSON中提取指定字段的值（简易解析，适配协议需求）
21 static string get_field(const string& j, const char* key) {
22     string pat = string("\"") + key + "\"";
23     size_t p = j.find(pat);
24     if (p == string::npos) return "";
25     p += pat.size();
26     // 跳过可能的空格和引号
27     while (p < j.size() && (j[p] == ' ' || j[p] == '"')) p++;
28     string v;
29     for (size_t i = p; i < j.size(); ++i) {
30         char c = j[i];
31         if (c == '\\') { if (i + 1 < j.size()) { v.push_back(j[i + 1]); ++i;
32             } }
33         else if (c == '"' || c == ',' || c == '}') { break; }
34         else v.push_back(c);
35     }
36     return v;
37 }

```

作用：确保消息按协议约定的 JSON 格式封装（如 join/msg/quit 类型），并正确解析服务器返回的结构化数据（如提取其他用户的消息内容）。

5. 连接管理函数

连接服务器函数 DoConnect

```

1 static void DoConnect(HWND hWnd){
2     if(g_connected) return;
3     wchar_t ipw[128], portw[16]; GetWindowTextW(g_hIp, ipw, 128);
4     GetWindowTextW(g_hPort, portw, 16);
5     int nlen=GetWindowTextLengthW(g_hNick); wstring nickw(nlen,L'\0');
6     GetWindowTextW(g_hNick,&nickw[0],nlen+1);
7     wstring nickw_trim = Trim(nickw);
8     if(nickw_trim.empty()){
9         MessageBoxW(hWnd, L"请输入昵称后再加入。", L"提示", MB_OK|
10             MB_ICONINFORMATION);
11         SetFocus(g_hNick); SendMessageW(g_hNick, EM_SETSEL, 0, -1); return;
12     }
13     string host = wide_to_utf8(ipw); int port = _wtoi(portw);
14     if(host.empty() || port<=0){ AppendLog(g_hLog, L"请输入正确的 IP 与端口");
15         return; }
16     addrinfo hints{}; hints.ai_family=AF_INET; hints.ai_socktype=SOCK_STREAM;
17     addrinfo* res=nullptr; char portStr[16]; sprintf(portStr, "%d", port);
18     if(getaddrinfo(host.c_str(), portStr, &hints, &res)!=0){ AppendLog(g_hLog,
19         L"解析地址失败"); return; }
20     g_sock = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
21     if(g_sock==INVALID_SOCKET){ freeaddrinfo(res); AppendLog(g_hLog,L"socket 创建失败"); return; }
22     if(connect(g_sock, res->ai_addr, (int)res->ai_addrlen)==SOCKET_ERROR){
23         freeaddrinfo(res); closesocket(g_sock); g_sock=INVALID_SOCKET;
24         AppendLog(g_hLog,L"连接失败"); return;
25     }
26     freeaddrinfo(res);
27
28     g_connected=true; g_exit=false;
29     string nick = wide_to_utf8(nickw_trim);
30     send_frame(g_sock, make_json("join", nick, "")); // 入场广播
31
32     g_rxThread = thread(RxLoop, hWnd);
33     SetWindowTextW(g_hBtnConn, L"离开"); EnableWindow(g_hSend, TRUE);
34     EnableAddrBar(FALSE);
35     AppendLog(g_hLog, L"你已加入聊天区");
36 }

```

DoConnect 函数负责处理与服务器的连接过程。

首先检查是否已连接，若已连接则返回。接着获取用户输入的服务器 IP、端口和昵称信息，并对昵称进行 trim 处理。如果处理后的昵称为空，会弹出提示框要求用户输入昵称，并将焦点定位到昵称输入框。若 IP 为空或端口无效（小于等于 0），则在日志中提示输入正确的 IP 和端

口。

之后，通过 `getaddrinfo` 函数解析服务器地址信息，创建套接字并尝试连接服务器，若过程中出现解析失败、套接字创建失败或连接失败等情况，都会在日志中给出相应提示并释放已分配的资源。连接成功后，将连接状态设为 `true`，退出标志设为 `false`，发送类型为“join”的加入消息告知服务器，启动接收线程 `RxLoop`，更新界面控件状态（将“加入”按钮改为“离开”，启用发送按钮，禁用地址栏相关控件），并在日志中添加“你已加入聊天区”的提示。

发送聊天信息函数 `DoSend`

```
1 static void DoSend() {
2     if (!g_connected) return;
3     int len = GetWindowTextLengthW(g_hInput); if (len <= 0) return;
4     wstring w(len, L'\0'); GetWindowTextW(g_hInput, &w[0], len + 1);
5     int nlen = GetWindowTextLengthW(g_hNick); wstring nw(nlen, L'\0');
6     GetWindowTextW(g_hNick, &nw[0], nlen + 1);
7     string nick = wide_to_utf8(nw.empty() ? L"guest" : nw);
8     string text = wide_to_utf8(w);
9     if (text == "/quit") { DoDisconnect(); return; }
10    if (!send_frame(g_sock, make_json("msg", nick, text))) { AppendLog(g_hLog, L
11        "发送失败"); DoDisconnect(); return; }
12    SetWindowTextW(g_hInput, L"");
13 }
```

`DoSend` 函数用于发送聊天消息。

首先检查是否处于连接状态，若未连接则返回。获取输入框中的消息内容，若内容长度为 0 则直接返回。接着获取用户昵称（为空时默认“guest”），将昵称和消息内容从宽字符转换为 UTF-8 编码。

如果消息内容是“/quit”，则触发断开连接操作。否则，调用 `send_frame` 函数向服务器发送类型为“msg”的消息，若发送失败，在日志中提示并断开连接。发送成功后，清空输入框内容。

断开连接函数 `DoDistconnect`

```
1 static void DoDistconnect() {
2     if (!g_connected) return;
3     g_connected = false;
4     g_exit = true;
5     // 发送quit消息告知服务器
6     int nlen = GetWindowTextLengthW(g_hNick);
7     wstring w(nlen, L'\0');
8     GetWindowTextW(g_hNick, &w[0], nlen + 1);
9     string nick = wide_to_utf8(w.empty() ? L"guest" : w);
10    send_frame(g_sock, make_json("quit", nick, ""));
11    // 关闭套接字并清理线程
12    shutdown(g_sock, SD_BOTH);
13    closesocket(g_sock);
14    g_sock = INVALID_SOCKET;
15    if (g_rxThread.joinable()) g_rxThread.join();
16    // 恢复界面状态
17    SetWindowTextW(g_hBtnConn, L"加入");
18    EnableWindow(g_hSend, FALSE);
19    EnableAddrBar(TRUE);
20 }
```

```

20 AppendLog(g_hLog, L"你已离开聊天区");
21 }

```

DoDisconnect 函数主要用于处理客户端断开与服务器连接的操作。

它首先检查当前是否处于连接状态，如果未连接则直接返回。若已连接，会先将连接状态标记为 false，退出标志设为 true，然后获取用户昵称（若为空则默认使用“guest”），并向服务器发送一条类型为“quit”的退出消息，告知服务器当前用户已离开。接着，通过 shutdown 关闭套接字的发送和接收功能，再调用 closesocket 关闭套接字并将其重置为无效状态。

之后，检查接收线程是否可连接，若可连接则等待线程结束，确保资源正确释放。最后，更新界面控件状态，将“离开”按钮改回“加入”，禁用发送按钮，启用地址栏相关控件，并在聊天日志中添加“你已离开聊天区”的提示信息。

6. 界面布局函数

```

1 static void DoLayout(HWND hWnd){
2     RECT rc; GetClientRect(hWnd,&rc); int W=rc.right-rc.left, H=rc.bottom-rc.
        top;
3     int pad=10,row=28,y=pad,x=pad;
4
5     // 标签 + 输入框
6     MoveWindow(g_hLblIp, x, y+5, 70, row, TRUE); x+=70+6;
7     MoveWindow(g_hIp, x, y, 220, row, TRUE); x+=220+10;
8
9     MoveWindow(g_hLblPort,x, y+5, 40, row, TRUE); x+=40+6;
10    MoveWindow(g_hPort, x, y, 80, row, TRUE); x+=80+10;
11
12    MoveWindow(g_hLblNick,x, y+5, 40, row, TRUE); x+=40+6;
13    MoveWindow(g_hNick, x, y, 140, row, TRUE); x+=140+10;
14
15    MoveWindow(g_hBtnConn,x, y, 90, row, TRUE);
16
17    y+=row+pad; x=pad;
18    MoveWindow(g_hLog, x, y, W-2*pad, H-y-row-2*pad, TRUE);
19    MoveWindow(g_hInput,x, H-row-pad, W-2*pad-90-pad, row, TRUE);
20    MoveWindow(g_hSend, W-pad-90, H-row-pad, 90, row, TRUE);
21 }

```

DoLayout 函数负责调整窗口中控件的布局，确保在窗口大小改变时控件能合理排列。

它首先获取窗口客户区的矩形区域，计算出客户区的宽度 (W) 和高度 (H)，并定义了边距 (pad) 和控件行高 (row)。

然后从窗口顶部开始，依次设置各个标签（服务器、端口、昵称）和对应的输入框以及“加入 / 离开”按钮的位置和大小，通过调整 x 坐标依次排列这些控件。

接着计算聊天日志区域 (g_hLog) 的位置，使其位于顶部控件下方，并占据合适的高度。最后设置输入框 (g_hInput) 和发送按钮 (g_hSend) 的位置，使其位于窗口底部，输入框占据大部分宽度，发送按钮在其右侧，整体布局保持一致的边距和间距，确保界面美观且合理。

7. 程序入口函数

```

1 int APIENTRY wWinMain(HINSTANCE hInst,HINSTANCE,LPWSTR,int nShow){
2     WSADATA w; WSASStartup(MAKEWORD(2,2),&w);
3     WNDCLASSW wc{}; wc.lpfnWndProc=WndProc; wc.hInstance=hInst; wc.
        lpzClassName=kClassName;
4     wc.hCursor=LoadCursor(nullptr,IDC_ARROW); wc.hbrBackground=(HBRUSH)(
        COLOR_WINDOW+1); RegisterClassW(&wc);
5     HWND hWnd=CreateWindowW(kClassName,L"简易聊天室",WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT,CW_USEDEFAULT,920,620,nullptr,nullptr,hInst,nullptr);
6     ShowWindow(hWnd,nShow);
7     MSG msg; while(GetMessageW(&msg,nullptr,0,0)){
8         if(msg.message==WM_KEYDOWN && msg.wParam==VK_RETURN && GetFocus()==
            g_hInput){ DoSend(); continue; }
9         TranslateMessage(&msg); DispatchMessageW(&msg);
10    }
11    WSACleanup(); return 0;
12 }

```

程序入口 `wWinMain` 函数首先初始化 Winsock 库，注册窗口类并创建主窗口，然后进入消息循环，处理用户输入和系统消息，其中特别处理了输入框中的回车键，使其能触发发送消息操作，最后在程序退出时清理 Winsock 资源。

整体来看，我们的程序通过多线程实现了网络通信与界面响应的分离，采用结构化的消息格式保证了数据传输的可靠性，提供了简洁的用户界面，实现了聊天室客户端的基本功能。

六、 结果展示

我们程序的运行需要在 MinGW 的终端中输入以下命令进行编译：

```

1 # 服务器
2 g++ -std=c++17 server.cpp -lws2_32 -o server.exe
3 # 客户端
4 g++ -std=c++17 client_win32.cpp -lws2_32 -luser32 -lgdi32 -municode -mwindows
    -o client.exe

```

我们写了一个简单的 Makefile 文件，并且写了一个简单的自动化脚本进行运行。之后，我们就可以通过先按 `Ctrl+Shift+B` 执行 Build (make)。再打开 Terminal → Run Task... 选择 Run server 或 Run client (GUI)。或者也可以按 `Ctrl+Shift+P` 输入 Tasks: Run Task，再选对应任务。

接下来，我们对几种不同的情况进行测试。

(一) 只有 client 运行

当我们不运行服务器端，只运行客户端时，如下：

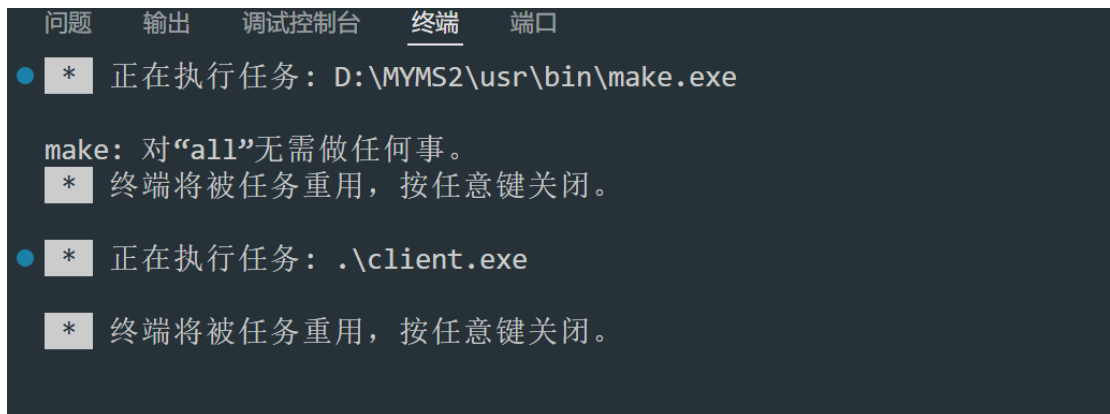


图 1: 只运行 client.exe

我们可以看到, 输入昵称后显示连接失败



图 2: client 连接失败

(二) 运行 sever 和 client

我们先运行 server.exe, 可以看到在监听 5000 端口。



图 3: Enter Caption

1. 单人聊天

我们运行 client.exe, 输入昵称, 发现加入了聊天区, 在输入框中输入, 没有问题。



图 4: 运行单个客户端

2. 多人聊天

我们运行多个 client.exe, 打开多个聊天框, 分别是 user1,2,3,4, 我们设置的最大监听是 255 个, 这里只开 4 个演示一下。

我们可以看到, 先来的用户可以看到后进来的用户的信息。

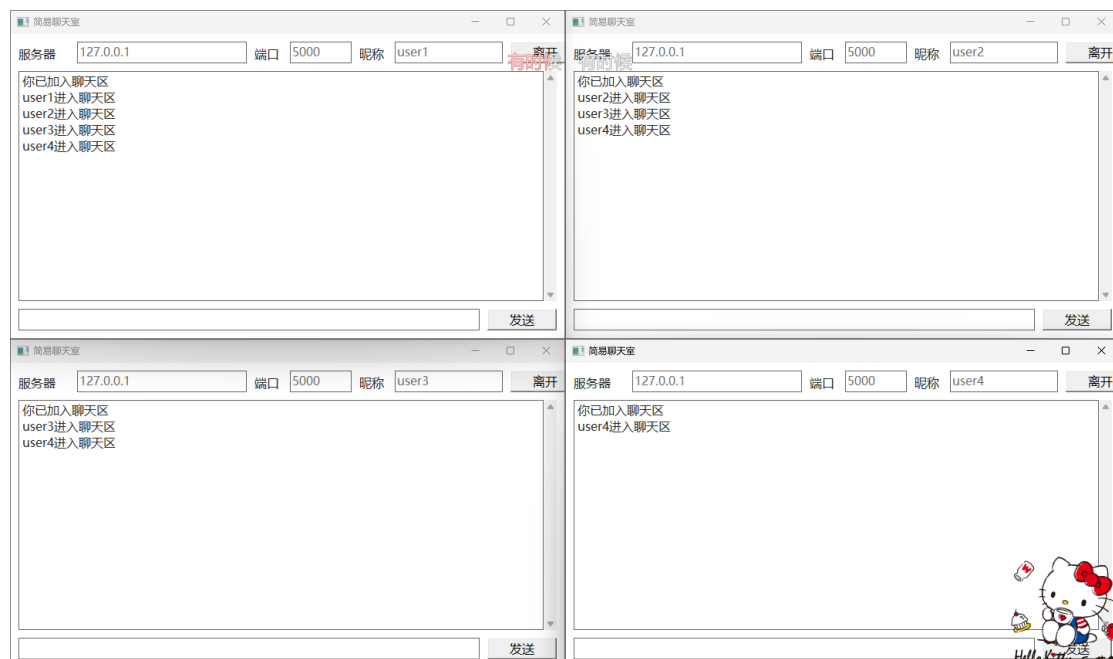


图 5: 多个客户端聊天

4 个客户端进行聊天彼此都可以收到信息, 并且支持中文和英文, 功能无误。

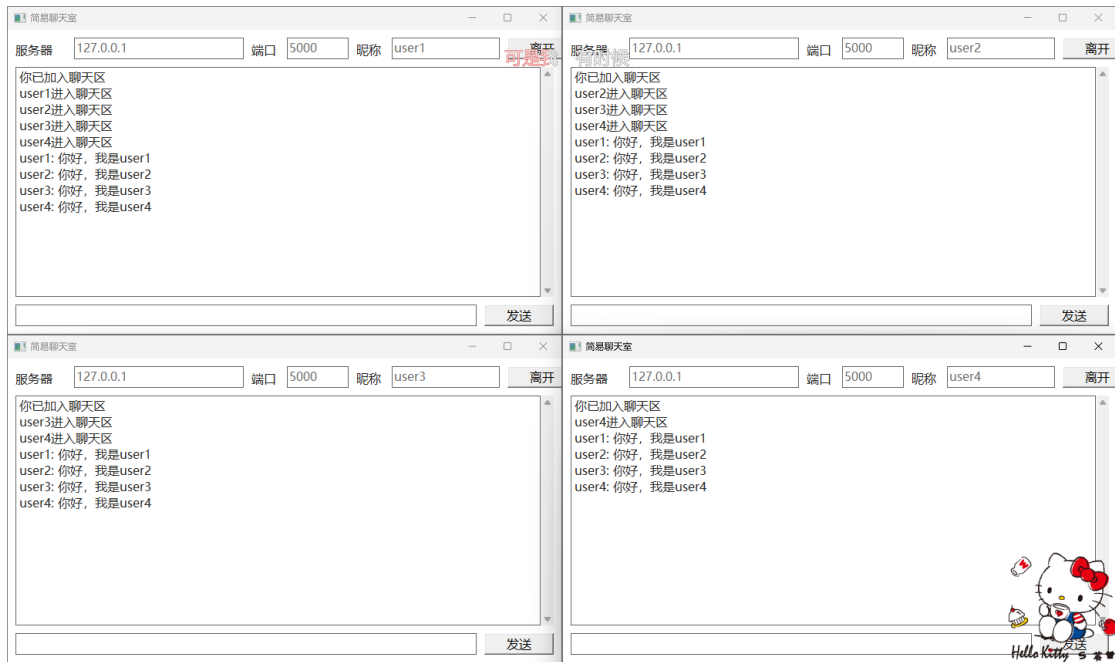


图 6: 多个客户端聊天

3. 离开聊天区

我们点击右上角的“离开”，即可离开聊天区，但是没有退出界面，可以再重新加入，点击右上角的“×”号，即可以关闭聊天区。

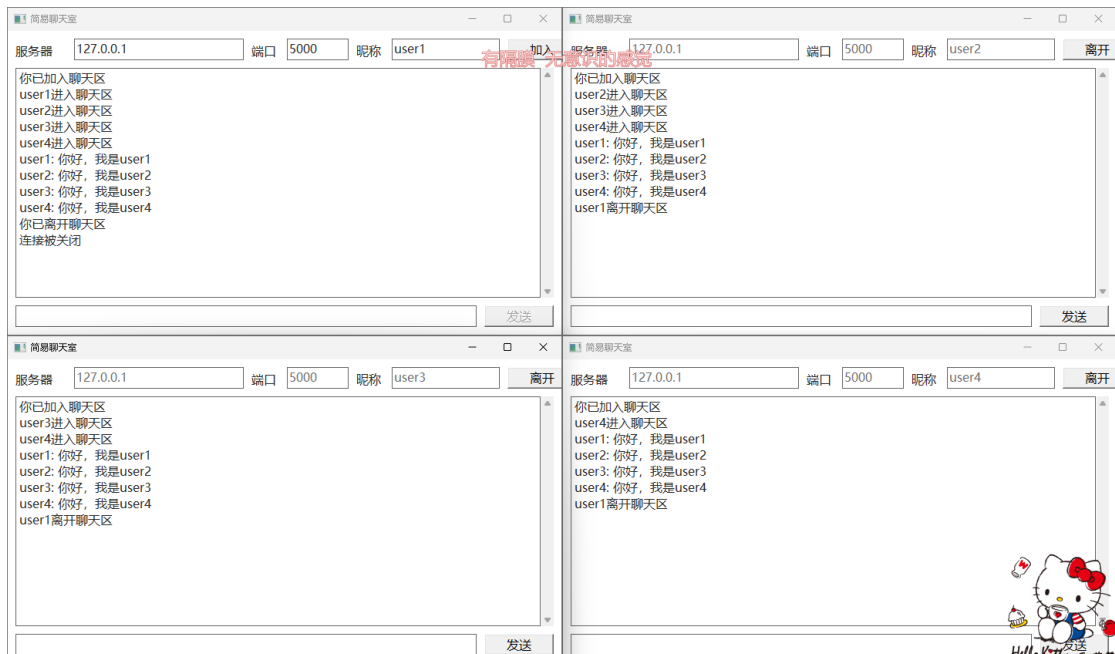


图 7: 离开聊天区

七、 心得体会

本次利用流式套接字编写聊天程序的实验，是对计算机网络理论知识与编程实践的一次深度结合，过程中既有挑战带来的困惑，也有突破后的收获，让我对网络编程有了更直观且深刻的理解。

实验初期，最大的困难在于解决 TCP 协议的粘包与拆包问题。由于 TCP 是面向字节流的协议，多次发送的消息可能被合并传输，单次发送的长消息也可能被拆分，直接导致接收方无法准确解析消息边界。最初尝试通过换行符作为分隔符，但在测试中发现，当消息内容本身包含换行时会误判，且中文消息的编码转换可能干扰分隔符识别。最终参考协议设计思路，采用“4 字节长度前缀 + UTF-8 JSON”的帧结构，通过 `htonl/ntohl` 函数处理字节序转换，确保接收方先读取长度再获取完整消息，彻底解决了这一问题，也让我深刻体会到“协议设计是网络通信的基石”——清晰的协议规则能从根源上规避数据解析的混乱。

多线程并发处理是另一个挑战。服务器需要同时应对多个客户端的连接与消息转发，初期未加锁保护客户端列表时，频繁出现线程安全问题：多个线程同时修改列表导致迭代器失效，或客户端退出时未及时清理资源引发程序崩溃。通过引入 `std::mutex` 互斥锁，对客户端列表的添加、删除、遍历操作进行同步控制，确保了多线程环境下的数据一致性。这让我认识到，并发编程中“共享资源的保护”至关重要，合理的同步机制是程序稳定性的保障。

客户端的中文显示问题也曾让我困扰许久。Win32 API 默认使用宽字符 (UTF-16)，而网络传输需采用 UTF-8 编码，两者转换时若处理不当会导致中文乱码。通过 `MultiByteToWideChar` 和 `WideCharToMultiByte` 函数实现编码转换，并在 JSON 序列化时对特殊字符进行转义，最终实现了中英文消息的正常收发。这一过程让我明白了编码兼容的细节：不同系统、不同接口对字符编码的要求存在差异，编程时需针对性处理，才能保证文本信息的完整性。

此外，调试网络程序的过程也充满挑战。由于网络操作涉及客户端、服务器两端，且多线程环境下故障难以复现，初期常遇到“消息发送后无响应”“客户端突然断开连接”等问题。通过在关键步骤打印日志（如服务器控制台输出客户端加入/离开信息、消息转发记录），结合 Wireshark 抓包分析数据帧结构，逐步定位到错误原因——可能是 `send_frame` 函数返回失败未处理，或 `recv_frame` 校验长度时逻辑疏漏。这让我掌握了网络程序的调试方法：日志记录与抓包分析相结合，能高效追踪数据流向，定位问题节点。

通过本次实验，我不仅掌握了 Socket 编程的核心流程（创建套接字、绑定、监听、连接、收发数据），理解了 TCP 协议的可靠传输机制，更体会到“分层设计”的思想：将程序拆分为网络初始化、数据收发、协议解析、UI 交互等模块，每个模块专注于单一功能，既便于代码维护，也能降低调试难度。同时，跨平台兼容（如服务器端对 Windows 和类 Unix 系统的适配）与异常处理（如网络中断后的资源清理）的实践，让我认识到健壮的程序需要考虑各种边界情况，细节处理直接影响用户体验。

总之，这次实验让我深刻感受到理论与实践的差距——书本上的 TCP 三次握手、套接字函数定义，只有在实际编程中才能理解其背后的设计逻辑与应用场景。未来在编写网络程序时，我会更加注重协议的清晰性、并发的安全性和异常的鲁棒性，将本次收获融入到更复杂的系统开发中。