

电子科技大学

计算机专业类课程

实验报告

课程名称：计算机网络编程

学 院：计算科学与工程学院

学院专业：计算机科学与技术

学 号：2023080901012

学生姓名：古文斌

指导教师：向渝老师

日 期：2025年11月29日

电子科技大学

实验报告

学生姓名：古文斌 学号：2023080901012 指导教师：向渝老师

实验地点：电子科技大学清水河 校区主楼 A2-412

一、实验室名称：电子科技大学清水河校区主楼 A2-412

二、实验项目名称：基于 TCP 协议的简单聊天软件的实现

三、实验原理

(一) TCP 协议原理

TCP (Transmission Control Protocol, 传输控制协议) 是一种面向连接的、可靠的、基于字节流的传输层通信协议。TCP 协议在传输数据之前需要建立连接，通过三次握手 (Three-Way Handshake) 确保双方都准备好进行数据传输。在数据传输过程中，TCP 通过序列号、确认应答、超时重传、流量控制和拥塞控制等机制保证数据的可靠传输。

TCP 协议的主要特点包括：

- 面向连接：通信前必须先建立连接，通信结束后需要释放连接
- 可靠传输：通过确认机制、序列号和重传机制保证数据可靠到达
- 全双工通信：允许数据在两个方向上同时传输
- 面向字节流：TCP 把应用程序交付的数据看成一连串无结构的字节流

(二) Socket 编程接口

Socket (套接字) 是网络通信的端点，是应用程序与网络协议栈交互的接口。在 Linux/Unix 系统中，Socket 编程遵循 BSD Socket API 规范，主要涉及以下系统调用：

- `socket()`: 创建一个套接字描述符
- `bind()`: 将套接字绑定到特定的 IP 地址和端口
- `listen()`: 使套接字进入监听状态，准备接受连接
- `accept()`: 接受客户端的连接请求，返回新的套接字
- `connect()`: 客户端发起与服务器的连接
- `send()/recv()`: 发送和接收数据
- `close()`: 关闭套接字连接

(三) 多线程并发编程

在聊天服务器的实现中，需要同时处理多个客户端的连接和消息转发。使用 POSIX 线程 (pthread) 库实现多线程并发处理是常用的解决方案。每当有新的客户端连接时，服务器创建一个新的线程来处理该客户端的消息收发，主线程继续监听新的连接请求。

线程同步是多线程编程中的关键问题。当多个线程需要访问共享资源（如客户端列表）时，必须使用互斥锁 (mutex) 来保证数据的一致性，防止竞态条件的发生。

(四) 客户端/服务器通信模型

TCP 客户端/服务器通信的基本流程如下：

- 1) 服务器调用 `socket()` 创建套接字
- 2) 服务器调用 `bind()` 绑定地址和端口

- 3) 服务器调用 `listen()` 进入监听状态
- 4) 服务器调用 `accept()` 等待客户端连接
- 5) 客户端调用 `socket()` 创建套接字
- 6) 客户端调用 `connect()` 连接服务器
- 7) 双方通过 `send()/recv()` 进行数据交换
- 8) 通信结束后双方调用 `close()` 关闭连接

四、实验目的

- 1) 掌握服务端软件的工作原理
- 2) 掌握针对字符流的 TCP 客户/服务端软件的编程步骤和编程实现
- 3) 实现两个客户端之间的字符流消息传送

五、实验内容

本实验包括以下主要任务：

(一) TCP 聊天服务器的实现

- 1) 创建 TCP 套接字并绑定到指定端口（8888 端口）
- 2) 监听客户端连接请求，支持最多 10 个客户端同时在线
- 3) 使用多线程处理多个客户端的并发连接
- 4) 实现消息广播功能，将消息转发给其他所有在线客户端
- 5) 实现私聊功能，支持向指定用户发送消息
- 6) 管理客户端的上线、下线通知
- 7) 使用互斥锁保证线程安全

(二) TCP 聊天客户端的实现

- 1) 连接到指定的服务器地址和端口
- 2) 使用多线程分离消息发送和接收功能
- 3) 支持命令行参数配置服务器地址和端口
- 4) 实现用户命令处理：修改昵称、查看在线用户、私聊、退出等
- 5) 实现实时消息显示

六、实验器材(设备、元器件)

硬件环境

- 计算机：个人 PC
- 网络环境：局域网/本地回环接口

软件环境

- 操作系统：Linux
- 编译器：GCC
- 开发工具：VS Code
- 第三方库/框架：POSIX Sockets API、POSIX Threads（`pthread`）
- 版本控制：Git
- 构建工具：Make

七、实验步骤

(一) 问题描述

7.1.1. TCP 聊天服务器

设计并实现一个 TCP 聊天服务器，该程序能够：

- 在指定端口监听客户端连接
- 同时处理多个客户端的连接和消息
- 接收客户端消息并广播给其他所有客户端
- 支持私聊功能，将消息只发送给指定用户
- 管理用户昵称和在线状态
- 通知所有客户端用户的上线和下线

7.1.2. TCP 聊天客户端

设计并实现一个 TCP 聊天客户端，该程序能够：

- 连接到指定的服务器
- 发送消息到服务器
- 接收并显示来自服务器的消息
- 支持修改昵称、查看在线用户、私聊等命令
- 优雅地处理退出操作

(二) 算法分析与概要设计

7.2.1. 服务器算法设计

输入：客户端的连接请求和消息数据

输出：转发给目标客户端的消息

算法流程：

- 1) 步骤 1：初始化服务器套接字，绑定端口并开始监听
- 2) 步骤 2：主循环接受客户端连接，为每个连接创建处理线程
- 3) 步骤 3：客户端处理线程接收消息，根据消息类型（命令或普通消息）进行处理
- 4) 步骤 4：普通消息广播给所有其他客户端，私聊消息发送给指定用户
- 5) 步骤 5：客户端断开时，清理资源并通知其他用户

7.2.2. 客户端算法设计

输入：用户键盘输入和服务器返回的消息

输出：发送给服务器的消息和屏幕显示

算法流程：

- 1) 步骤 1：连接到服务器
- 2) 步骤 2：创建接收消息的后台线程
- 3) 步骤 3：主线程循环读取用户输入并发送给服务器
- 4) 步骤 4：接收线程实时显示服务器返回的消息
- 5) 步骤 5：用户输入退出命令时关闭连接

(三) 核心算法的详细设计与实现

7.3.1. 客户端信息管理

客户端信息结构体设计：

代码 1: 客户端信息结构体

```

1 // 客户端信息结构体
2 typedef struct {
3     int sockfd;           // socket文件描述符
4     struct sockaddr_in addr; // 客户端地址
5     char name[32];        // 客户端昵称
6     int active;          // 是否活跃
7 } ClientInfo;
8
9 // 全局变量
10 ClientInfo clients[MAX_CLIENTS];
11 pthread_mutex_t clients_mutex = PTHREAD_MUTEX_INITIALIZER;

```

7.3.2. 服务器主循环与连接处理

接受客户端连接并创建处理线程:

代码 2: 服务器主循环接受连接

```

1 // 主循环: 接受客户端连接
2 while (server_running) {
3     client_fd = accept(server_fd, (struct sockaddr *)&client_addr, &
4         client_len);
5     if (client_fd < 0) {
6         if (server_running) perror("接受连接失败");
7         continue;
8     }
9     // 查找空闲槽位并添加客户端
10    pthread_mutex_lock(&clients_mutex);
11    int idx = -1;
12    for (int i = 0; i < MAX_CLIENTS; i++) {
13        if (clients[i].active == 0) { idx = i; break; }
14    }
15    // 添加新客户端并创建处理线程
16    clients[idx].sockfd = client_fd;
17    clients[idx].active = 1;
18    pthread_mutex_unlock(&clients_mutex);
19
20    int *client_idx = malloc(sizeof(int));
21    *client_idx = idx;
22    pthread_create(&tid, NULL, handle_client, client_idx);
23    pthread_detach(tid);
}

```

7.3.3. 消息广播功能

将消息转发给所有在线客户端:

代码 3: 消息广播函数

```

1 // 广播消息给所有客户端 (除了发送者)
2 void broadcast_message(const char *message, int sender_idx) {
3     pthread_mutex_lock(&clients_mutex);
4     for (int i = 0; i < MAX_CLIENTS; i++) {
5         if (clients[i].active && i != sender_idx) {

```

```

6     if (send(clients[i].sockfd, message, strlen(message), 0) < 0) {
7         clients[i].active = 0; // 发送失败，标记为不活跃
8     }
9 }
10}
11pthread_mutex_unlock(&clients_mutex);
12}

```

7.3.4. 私聊功能实现

向指定用户发送私聊消息：

代码 4: 私聊消息发送函数

```

1 // 发送私聊消息给指定用户
2 void send_private_message(const char *message, const char *target_name,
3                           int sender_idx) {
4     int found = 0;
5     pthread_mutex_lock(&clients_mutex);
6     for (int i = 0; i < MAX_CLIENTS; i++) {
7         if (clients[i].active && strcmp(clients[i].name, target_name) == 0)
8             {
9                 send(clients[i].sockfd, message, strlen(message), 0);
10                found = 1;
11                break;
12            }
13    }
14    pthread_mutex_unlock(&clients_mutex);
15
16    if (!found) {
17        char error_msg[128];
18        snprintf(error_msg, sizeof(error_msg),
19                  "[系统] 用户 '%s' 不在线或不存在\n", target_name);
20        send(clients[sender_idx].sockfd, error_msg, strlen(error_msg), 0);
21    }
}

```

7.3.5. 客户端消息接收线程

后台线程接收服务器消息：

代码 5: 客户端消息接收线程

```

1 // 接收消息的线程函数
2 void *receive_messages(void *arg) {
3     char buffer[BUFFER_SIZE];
4     int bytes_received;
5
6     while (client_running) {
7         memset(buffer, 0, sizeof(buffer));
8         bytes_received = recv(sockfd, buffer, sizeof(buffer) - 1, 0);
9
10    if (bytes_received <= 0) {
11        if (client_running) {
12            printf("\n与服务器的连接已断开\n");
}

```

```

13     client_running = 0;
14 }
15 break;
16 }
17 printf("%s", buffer);
18 fflush(stdout);
19 }
20 return NULL;
21 }

```

注：以上仅展示核心代码片段，完整源代码请参见附录一。

八、实验数据及结果分析

(一) TCP 聊天服务器功能验证

8.1.1. 服务器启动与端口监听

服务器程序成功启动后，在 8888 端口监听客户端连接。服务器显示启动信息，包括监听端口号和最大连接数等配置信息。

```

PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE
airprofly@airprofly:/mnt/e/codeFiles/netWorkProgram/expr5_1$ make
gcc -Wall -Wextra -g -o server server.c -pthread
airprofly@airprofly:/mnt/e/codeFiles/netWorkProgram/expr5_1$ ./server
=====
TCP聊天服务器已启动
监听端口: 8888
最大连接数: 10
按 Ctrl+C 关闭服务器
=====

[+] 新客户端连接: 127.0.0.1:55714 (分配为 用户1)
当前在线人数: 1
[+] 新客户端连接: 127.0.0.1:55722 (分配为 用户2)
当前在线人数: 2
[+] 新客户端连接: 127.0.0.1:42398 (分配为 用户3)
当前在线人数: 3
[消息] 用户3: hello 这是来自用户3的消息
[私聊] 用户3 -> 用户2: 这用发给用户2的私聊消息
[私聊] 用户2 -> 用户1: 这里是用户2发给用户1的私聊消息
|
```

图 1: 服务器启动与客户端连接过程

结果分析：从图1可以看出，服务器成功启动并在 8888 端口监听。当客户端连接时，服务器显示客户端的 IP 地址、端口号，并为其分配用户昵称。同时显示当前在线人数，说明服务器的连接管理功能正常工作。

8.1.2. 多客户端连接与消息广播

验证服务器能够同时处理多个客户端的连接，并将消息广播给其他所有在线客户端。

结果分析：从图8.1.2、1、4,3可以看出，当用户 3 发送消息时，服务器成功将消息转发给其他所有在线客户端(用户 1 和用户 2)。消息格式包含发送者的昵称，使接收者能够识别消息来源。这验证了实验要求中“实现两个客户端之间的字符流消息传送”的功能。

(二) TCP 聊天客户端功能验证

8.2.1. 客户端连接与消息收发

验证客户端能够成功连接服务器，并进行消息的发送和接收。

```
○ airprofly@airprofly:/mnt/e/codeFiles/networkProgram/expr5_1$ ./client
=====
TCP聊天客户端
正在连接 127.0.0.1:8888 ...
=====
连接成功！

欢迎来到聊天室！你的昵称是：用户1
命令列表：
/quit      - 退出聊天室
/name <昵称> - 修改昵称
/list      - 查看在线用户
/msg <用户> <消息> - 私聊指定用户
直接输入消息则广播给所有人
[系统] 用户2 加入了聊天室
[系统] 用户3 加入了聊天室
[用户3]: hello 这是来自用户3的消息
[私聊][用户2 -> 你]: 这里是用户2发给用户1的私聊消息
[
```

图 2: 多客户端消息广播验证

```
PROBLEMS    OUTPUT    TERMINAL  1    DEBUG CONSOLE

○ airprofly@airprofly:/mnt/e/codeFiles/networkProgram/expr5_1$ ./client
=====
TCP聊天客户端
正在连接 127.0.0.1:8888 ...
=====
连接成功！

欢迎来到聊天室！你的昵称是：用户2
命令列表：
/quit      - 退出聊天室
/name <昵称> - 修改昵称
/list      - 查看在线用户
/msg <用户> <消息> - 私聊指定用户
直接输入消息则广播给所有人
[系统] 用户3 加入了聊天室
[用户3]: hello 这是来自用户3的消息
[私聊][用户3 -> 你]: 这用发给用户2的私聊消息
[msg 这里使用用户 2发给用户1的私聊消息
[系统] 用法：/msg <用户名> <消息>
/msg 用户1 这里是用户2发给用户1的私聊消息
[私聊][你 -> 用户1]: 这里是用户2发给用户1的私聊消息
[
```

图 3: 客户端消息收发功能

结果分析：从图3可以看出，客户端成功连接到服务器后收到欢迎消息和命令列表提示。用户可以直接输入消息进行广播，也可以使用各种命令（如/name 修改昵称、/list 查看在线用户等）。当其他用户加入或离开聊天室时，客户端能够实时收到系统通知。

8.2.2. 私聊功能验证

验证客户端之间的私聊功能，确保私聊消息只发送给指定用户。



The screenshot shows a terminal window with the following text output:

```
PROBLEMS OUTPUT TERMINAL PORTS 1 DEBUG CONSOLE
airprofly@airprofly:/mnt/e/codeFiles/networkProgram/expr5_1$ ./client
=====
TCP聊天客户端
正在连接 127.0.0.1:8888 ...
=====
连接成功！

欢迎来到聊天室！你的昵称是： 用户3
命令列表：
/quit      - 退出聊天室
/name <昵称> - 修改昵称
/list      - 查看在线用户
/msg <用户> <消息> - 私聊指定用户
直接输入消息则广播给所有人
hello 这是来自用户3的广播 消息
/msg 用户2 这是用用户3发给用户2的私聊消息
[私聊][你 -> 用户2]: 这用发给用户2的私聊消息
```

图 4: 私聊功能验证

结果分析：从图4可以看出，使用/msg 命令可以向指定用户发送私聊消息。发送者收到确认信息”[私聊][你 -> 用户名]: 消息内容”，接收者收到”[私聊][发送者 -> 你]: 消息内容”。从图8.1.2、3、4可以看出，私聊消息(图4中用户3发给用户2的消息)不会被广播给其他用户(用户1)，保证了通信的私密性。同时，服务器端也记录了私聊信息的日志，便于管理和调试。

九、实验结论

通过本次实验，达成了全部实验目标：

- 1) **掌握服务端软件的工作原理：**通过实现 TCP 聊天服务器，深入理解了服务端程序的工作流程，包括套接字创建、地址绑定、端口监听、连接接受、多线程并发处理等关键环节。服务器采用多线程模型，每个客户端连接由独立线程处理，主线程持续监听新连接，实现了高效的并发处理能力。
- 2) **掌握 TCP 客户/服务端软件的编程步骤和实现：**完整实现了 TCP 服务器和客户端程序，掌握了 socket()、bind()、listen()、accept()、connect()、send()、recv() 等核心 API 的使用方法。理解了 TCP 面向连接、可靠传输的特性在实际编程中的体现，以及字节流传输的处理方式。
- 3) **实现两个客户端之间的字符流消息传送：**成功实现了通过服务器转发的客户端间消息传送功能。不仅实现了消息广播（一对多通信），还实现了私聊功能（一对一通信）。消息能够实时、可靠地在客户端之间传递，验证了整个通信链路的正确性。

实验结果表明，基于 TCP 协议的聊天软件能够提供稳定可靠的消息传输服务。服务器端的多线程设计保证了良好的并发处理能力，互斥锁的使用确保了线程安全。客户端的双线程设计（主线程发送、子线程接收）实现了消息收发的并行处理，提升了用户体验。

十、总结及心得体会

通过本次实验，我对 TCP 网络编程有了深入的理解，同时在多线程编程、网络协议应用等方面都有了显著提升。以下是我的主要收获和心得体会：

- 1) 理解了 **TCP/IP 协议栈的工作机制**：通过实际编程实践，深刻理解了 TCP 协议面向连接、可靠传输的特性。三次握手建立连接、四次挥手释放连接的过程不再是抽象的概念，而是在程序运行中真实发生的网络交互。
- 2) 掌握了 **Socket 编程的核心技术**：熟练使用了 POSIX Socket API 进行网络编程，理解了服务器端和客户端在编程模式上的差异。服务器需要 bind+listen+accept 的被动等待模式，而客户端采用 connect 的主动连接模式。
- 3) 深入理解了 **多线程并发编程**：为了支持多客户端并发，学习并实践了 pthread 多线程编程。理解了线程创建、线程分离、线程同步（互斥锁）等关键技术，以及多线程编程中常见的竞态条件问题及其解决方法。
- 4) 提升了 **程序设计和调试能力**：在实现过程中遇到了各种问题，如消息边界处理、连接异常处理、资源泄漏等。通过逐步调试和优化，提升了发现问题、分析问题、解决问题的能力。
- 5) 理解了 **网络程序的健壮性设计**：学会了处理各种异常情况，如客户端突然断开、网络传输错误、服务器资源耗尽等。通过信号处理、错误检查、资源清理等机制，提高了程序的健壮性和稳定性。
- 6) 培养了 **工程化开发思维**：使用 Makefile 进行项目构建，使用 Git 进行版本控制，编写 README 文档说明项目使用方法，这些工程化实践为今后的软件开发工作打下了基础。

总之，本次实验不仅让我掌握了 TCP 网络编程的核心技术，更重要的是培养了系统化思考和工程化开发的能力。这些经验和技能将为今后学习分布式系统、网络安全等高级课程奠定坚实的基础。

十一、对本实验过程及方法、手段的改进建议及展望

虽然本次实验成功实现了基于 TCP 协议的简单聊天软件，但在功能完善性、性能优化、安全性等方面仍有提升空间。针对当前实现的不足之处，提出以下改进建议和未来展望：

- 1) **用户认证机制**：当前实现中用户无需认证即可加入聊天室，可以添加用户注册、登录功能，使用用户名和密码进行身份验证，防止未授权访问。
- 2) **消息持久化**：添加聊天记录保存功能，将聊天消息存储到文件或数据库中。用户重新登录时可以查看历史消息，也便于进行消息审计和分析。
- 3) **文件传输功能**：扩展程序功能，支持客户端之间传输文件。需要设计文件传输协议，处理大文件分块传输、断点续传等问题。
- 4) **加密通信**：使用 TLS/SSL 对通信内容进行加密，保护用户隐私和数据安全。可以使用 OpenSSL 库实现安全的加密通信。
- 5) **性能优化**：当前每个客户端使用一个独立线程处理，在客户端数量较多时可能导致资源消耗过大。可以使用线程池或者 epoll/select 等 I/O 多路复用技术来优化性能。
- 6) **图形用户界面**：当前程序使用命令行界面，可以使用 Qt、GTK 等 GUI 框架开发图形界面，提升用户体验。
- 7) **群组功能**：实现聊天群组功能，用户可以创建群组、加入群组，在群组内进行多人聊天。
- 8) **消息格式优化**：设计更完善的消息协议，支持消息类型标识、消息 ID、时间戳等

元信息，便于消息管理和扩展。

以上改进方向涵盖了功能扩展、性能优化、安全加固等多个层面。在后续学习中，我将逐步尝试实现这些改进，不断提升自己的网络编程能力和系统设计水平。

报告评分：
指导教师签字：

附录一 详细代码

GitHub 代码仓库地址: <https://github.com/airprofly/networkProgram.git>

表 1: 代码文件说明表

文件名	所属部分	说明
code/server.c	服务器	TCP 聊天服务器源代码
code/client.c	客户端	TCP 聊天客户端源代码

(一) TCP 聊天服务器

代码 6: server.c

```
1  /**
2   * TCP 聊天服务器
3   * 功能: 接受多个客户端连接, 转发消息给其他客户端
4   */
5
6 #include <arpa/inet.h>
7 #include <netinet/in.h>
8 #include <pthread.h>
9 #include <signal.h>
10 #include <stdio.h>
11 #include <stdlib.h>
12 #include <string.h>
13 #include <sys/socket.h>
14 #include <sys/types.h>
15 #include <unistd.h>
16
17 #define MAX_CLIENTS 10
18 #define BUFFER_SIZE 1024
19 #define PORT 8888
20
21 // 客户端信息结构体
22 typedef struct {
23     int sockfd;
24     struct sockaddr_in addr;
25     char name[32];
26     int active;
27 } ClientInfo;
28
29 // 全局变量
30 ClientInfo clients[MAX_CLIENTS];
31 pthread_mutex_t clients_mutex = PTHREAD_MUTEX_INITIALIZER;
32 int server_running = 1;
33
34 // 函数声明
35 void *handle_client(void *arg);
36 void broadcast_message(const char *message, int sender_idx);
37 void send_private_message(const char *message, const char *target_name,
38                           int sender_idx);
39 void remove_client(int idx);
```

```
40 int get_client_count();
41 void signal_handler(int sig);
42
43 int main() {
44     int server_fd, client_fd;
45     struct sockaddr_in server_addr, client_addr;
46     socklen_t client_len = sizeof(client_addr);
47     pthread_t tid;
48
49     // 设置信号处理
50     signal(SIGINT, signal_handler);
51     signal(SIGPIPE, SIG_IGN); // 忽略SIGPIPE信号
52
53     // 初始化客户端数组
54     for (int i = 0; i < MAX_CLIENTS; i++) {
55         clients[i].sockfd = -1;
56         clients[i].active = 0;
57     }
58
59     // 创建socket
60     server_fd = socket(AF_INET, SOCK_STREAM, 0);
61     if (server_fd < 0) {
62         perror("创建socket失败");
63         exit(EXIT_FAILURE);
64     }
65
66     // 设置socket选项，允许地址重用
67     int opt = 1;
68     if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt))
69         ) < 0) {
70         perror("设置socket选项失败");
71         close(server_fd);
72         exit(EXIT_FAILURE);
73     }
74
75     // 配置服务器地址
76     memset(&server_addr, 0, sizeof(server_addr));
77     server_addr.sin_family = AF_INET;
78     server_addr.sin_addr.s_addr = INADDR_ANY;
79     server_addr.sin_port = htons(PORT);
80
81     // 绑定
82     if (bind(server_fd, (struct sockaddr *)&server_addr, sizeof(
83         server_addr)) <
84         0) {
85         perror("绑定失败");
86         close(server_fd);
87         exit(EXIT_FAILURE);
88     }
89
90     // 监听
91     if (listen(server_fd, MAX_CLIENTS) < 0) {
92         perror("监听失败");
93     }
```

```

91     close(server_fd);
92     exit(EXIT_FAILURE);
93 }
94
95 printf("=====\\n");
96 printf("    TCP 聊天服务器已启动\\n");
97 printf("    监听端口: %d\\n", PORT);
98 printf("    最大连接数: %d\\n", MAX_CLIENTS);
99 printf("    按 Ctrl+C 关闭服务器\\n");
100 printf("=====\\n\\n");
101
102 // 主循环: 接受客户端连接
103 while (server_running) {
104     client_fd = accept(server_fd, (struct sockaddr *)&client_addr, &
105                         client_len);
106     if (client_fd < 0) {
107         if (server_running) {
108             perror("接受连接失败");
109         }
110         continue;
111     }
112
113     // 检查是否达到最大连接数
114     pthread_mutex_lock(&clients_mutex);
115     int idx = -1;
116     for (int i = 0; i < MAX_CLIENTS; i++) {
117         if (clients[i].active == 0) {
118             idx = i;
119             break;
120         }
121     }
122     if (idx == -1) {
123         pthread_mutex_unlock(&clients_mutex);
124         printf("连接已满, 拒绝新连接: %s:%d\\n", inet_ntoa(client_addr.
125                                         sin_addr),
126                                         ntohs(client_addr.sin_port));
127         const char *msg = "服务器已满, 请稍后再试。\\n";
128         send(client_fd, msg, strlen(msg), 0);
129         close(client_fd);
130         continue;
131     }
132
133     // 添加新客户端
134     clients[idx].sockfd = client_fd;
135     clients[idx].addr = client_addr;
136     clients[idx].active = 1;
137     snprintf(clients[idx].name, sizeof(clients[idx].name), "用户%d",
138              idx + 1);
139     pthread_mutex_unlock(&clients_mutex);
140
141     printf("[+] 新客户端连接: %s:%d (分配为 %s)\\n",
142           inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port))

```

```

        ,
clients[idx].name);
printf("    当前在线人数: %d\n", get_client_count());
143
144 // 创建线程处理客户端
145 int *client_idx = malloc(sizeof(int));
146 *client_idx = idx;
147 if (pthread_create(&tid, NULL, handle_client, client_idx) != 0) {
148     perror("创建线程失败");
149     remove_client(idx);
150     free(client_idx);
151     continue;
152 }
153 pthread_detach(tid);
154 }

155 // 清理
156 close(server_fd);
157 printf("\n服务器已关闭\n");
158 return 0;
159 }

160 }

161 // 处理客户端消息的线程函数
162 void *handle_client(void *arg) {
163     int idx = *((int *)arg);
164     free(arg);

165     char buffer[BUFFER_SIZE];
166     char message[BUFFER_SIZE + 64];
167     int bytes_received;

168     // 发送欢迎消息
169     snprintf(buffer, sizeof(buffer),
170             "欢迎来到聊天室！你的昵称是: %s\n"
171             "命令列表:\n"
172             "  /quit      - 退出聊天室\n"
173             "  /name <昵称> - 修改昵称\n"
174             "  /list      - 查看在线用户\n"
175             "  /msg <用户> <消息> - 私聊指定用户\n"
176             "直接输入消息则广播给所有人\n",
177             clients[idx].name);
178     send(clients[idx].sockfd, buffer, strlen(buffer), 0);

179     // 通知其他用户
180     snprintf(message, sizeof(message), "[系统] %s 加入了聊天室\n",
181             clients[idx].name);
182     broadcast_message(message, idx);

183     // 接收并转发消息
184     while (server_running && clients[idx].active) {
185         memset(buffer, 0, sizeof(buffer));
186         bytes_received = recv(clients[idx].sockfd, buffer, sizeof(buffer) -
187             1, 0);

```

```

192
193     if (bytes_received <= 0) {
194         // 客户端断开连接
195         break;
196     }
197
198     // 移除末尾的换行符
199     buffer[strcspn(buffer, "\r\n")] = '\0';
200
201     if (strlen(buffer) == 0) {
202         continue;
203     }
204
205     // 处理命令
206     if (strncmp(buffer, "/quit", 5) == 0) {
207         break;
208     } else if (strncmp(buffer, "/name ", 6) == 0) {
209         // 修改昵称
210         char old_name[32];
211         pthread_mutex_lock(&clients_mutex);
212         strncpy(old_name, clients[idx].name, sizeof(old_name) - 1);
213         strncpy(clients[idx].name, buffer + 6, sizeof(clients[idx].name)
214             - 1);
215         clients[idx].name[sizeof(clients[idx].name) - 1] = '\0';
216         pthread_mutex_unlock(&clients_mutex);
217
218         sprintf(message, sizeof(message), "[系统] %s 改名为 %s\n",
219                 old_name,
220                 clients[idx].name);
221         broadcast_message(message, -1);
222         printf("[*] %s 改名为 %s\n", old_name, clients[idx].name);
223     } else if (strncmp(buffer, "/list", 5) == 0) {
224         // 显示在线用户
225         char list_msg[BUFFER_SIZE] = "[在线用户列表]\n";
226         pthread_mutex_lock(&clients_mutex);
227         for (int i = 0; i < MAX_CLIENTS; i++) {
228             if (clients[i].active) {
229                 char user_info[64];
230                 sprintf(user_info, sizeof(user_info), " - %s%s\n", clients[
231                     i].name,
232                     (i == idx) ? "(你)" : ""));
233                 strncat(list_msg, user_info, sizeof(list_msg) - strlen(
234                     list_msg) - 1);
235             }
236         }
237         pthread_mutex_unlock(&clients_mutex);
238         send(clients[idx].sockfd, list_msg, strlen(list_msg), 0);
239     } else if (strncmp(buffer, "/msg ", 5) == 0) {
240         // 私聊功能： /msg <用户名> <消息>
241         char *cmd_content = buffer + 5;
242         char target_name[32] = {0};
243         char *space_pos = strchr(cmd_content, ' ');

```

```

241     if (space_pos == NULL) {
242         const char *usage = "[系统] 用法: /msg <用户名> <消息>\n";
243         send(clients[idx].sockfd, usage, strlen(usage), 0);
244     } else {
245         // 提取目标用户名
246         size_t name_len = space_pos - cmd_content;
247         if (name_len >= sizeof(target_name)) {
248             name_len = sizeof(target_name) - 1;
249         }
250         strncpy(target_name, cmd_content, name_len);
251         target_name[name_len] = '\0';
252
253         // 提取消息内容
254         char *private_msg = space_pos + 1;
255
256         if (strlen(private_msg) == 0) {
257             const char *empty_msg = "[系统] 消息内容不能为空\n";
258             send(clients[idx].sockfd, empty_msg, strlen(empty_msg), 0);
259         } else {
260             // 构建私聊消息
261             snprintf(message, sizeof(message), "[私聊][%s -> 你]: %s\n",
262                     clients[idx].name, private_msg);
263             send_private_message(message, target_name, idx);
264
265             // 给发送者确认
266             snprintf(message, sizeof(message), "[私聊][你 -> %s]: %s\n",
267                     target_name, private_msg);
268             send(clients[idx].sockfd, message, strlen(message), 0);
269
270             printf("[私聊] %s -> %s: %s\n", clients[idx].name,
271                   target_name,
272                   private_msg);
273         }
274     } else if (strncmp(buffer, "/msg", 4) == 0) {
275         // 用户输入了 /msg 但格式不对 (如 /msg 用户2)
276         const char *usage =
277             "[系统] 私聊格式错误! 正确用法: /msg 用户名 消息内容\n";
278         "[系统] 注意: /msg 后面必须有空格\n";
279         "[系统] 示例: /msg 用户2 你好\n";
280         send(clients[idx].sockfd, usage, strlen(usage), 0);
281     } else {
282         // 广播普通消息
283         snprintf(message, sizeof(message), "[%s]: %s\n", clients[idx].name,
284                 buffer);
285         broadcast_message(message, idx);
286         printf("[消息] %s: %s\n", clients[idx].name, buffer);
287     }
288 }
289
290 // 客户端断开连接
291 char name_copy[32];

```

```

292     strncpy(name_copy, clients[idx].name, sizeof(name_copy) - 1);
293     name_copy[sizeof(name_copy) - 1] = '\0';
294
295     remove_client(idx);
296
297     printf("[-] %s 已断开连接\n", name_copy);
298     printf("    当前在线人数: %d\n", get_client_count());
299
300     snprintf(message, sizeof(message), "[系统] %s 离开了聊天室\n",
301             name_copy);
302     broadcast_message(message, -1);
303
304     return NULL;
305 }
306
307 // 广播消息给所有客户端（除了发送者）
308 void broadcast_message(const char *message, int sender_idx) {
309     pthread_mutex_lock(&clients_mutex);
310     for (int i = 0; i < MAX_CLIENTS; i++) {
311         if (clients[i].active && i != sender_idx) {
312             if (send(clients[i].sockfd, message, strlen(message), 0) < 0) {
313                 // 发送失败，标记为不活跃
314                 clients[i].active = 0;
315             }
316         }
317     }
318     pthread_mutex_unlock(&clients_mutex);
319 }
320
321 // 发送私聊消息给指定用户
322 void send_private_message(const char *message, const char *target_name,
323                           int sender_idx) {
324     int found = 0;
325     pthread_mutex_lock(&clients_mutex);
326     for (int i = 0; i < MAX_CLIENTS; i++) {
327         if (clients[i].active && strcmp(clients[i].name, target_name) == 0)
328         {
329             if (send(clients[i].sockfd, message, strlen(message), 0) < 0) {
330                 clients[i].active = 0;
331             }
332             found = 1;
333             break;
334         }
335     }
336     pthread_mutex_unlock(&clients_mutex);
337
338     // 如果目标用户不存在，通知发送者
339     if (!found) {
340         char error_msg[128];
341         snprintf(error_msg, sizeof(error_msg), "[系统] 用户 '%s' 不在线或不
342             存在\n",
343             target_name);
344         send(clients[sender_idx].sockfd, error_msg, strlen(error_msg), 0);

```

```

342     }
343 }
344
345 // 移除客户端
346 void remove_client(int idx) {
347     pthread_mutex_lock(&clients_mutex);
348     if (clients[idx].active) {
349         close(clients[idx].sockfd);
350         clients[idx].sockfd = -1;
351         clients[idx].active = 0;
352         memset(clients[idx].name, 0, sizeof(clients[idx].name));
353     }
354     pthread_mutex_unlock(&clients_mutex);
355 }
356
357 // 获取当前在线客户端数量
358 int get_client_count() {
359     int count = 0;
360     pthread_mutex_lock(&clients_mutex);
361     for (int i = 0; i < MAX_CLIENTS; i++) {
362         if (clients[i].active) {
363             count++;
364         }
365     }
366     pthread_mutex_unlock(&clients_mutex);
367     return count;
368 }
369
370 // 信号处理函数
371 void signal_handler(int sig) {
372     (void)sig; // 抑制未使用参数警告
373     printf("\n接收到关闭信号，正在关闭服务器...\n");
374     server_running = 0;
375
376     // 关闭所有客户端连接
377     pthread_mutex_lock(&clients_mutex);
378     for (int i = 0; i < MAX_CLIENTS; i++) {
379         if (clients[i].active) {
380             const char *msg = "[系统] 服务器关闭\n";
381             send(clients[i].sockfd, msg, strlen(msg), 0);
382             close(clients[i].sockfd);
383             clients[i].active = 0;
384         }
385     }
386     pthread_mutex_unlock(&clients_mutex);
387 }

```

(二) TCP 聊天客户端

代码 7: client.c

```

1 /**
2  * TCP 聊天客户端

```

```

3  * 功能：连接服务器，发送和接收聊天消息
4  */
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <string.h>
8 #include <unistd.h>
9 #include <sys/socket.h>
10 #include <sys/types.h>
11 #include <netinet/in.h>
12 #include <arpa/inet.h>
13 #include <pthread.h>
14 #include <signal.h>
15
16 #define BUFFER_SIZE 1024
17 #define DEFAULT_PORT 8888
18 #define DEFAULT_SERVER "127.0.0.1"
19
20 // 全局变量
21 int client_running = 1;
22 int sockfd = -1;
23
24 // 函数声明
25 void *receive_messages(void *arg);
26 void signal_handler(int sig);
27 void print_usage(const char *program);
28
29 int main(int argc, char *argv[])
30 {
31     struct sockaddr_in server_addr;
32     pthread_t recv_thread;
33     char buffer[BUFFER_SIZE];
34     char *server_ip = DEFAULT_SERVER;
35     int port = DEFAULT_PORT;
36
37     // 解析命令行参数
38     if (argc >= 2)
39     {
40         server_ip = argv[1];
41     }
42     if (argc >= 3)
43     {
44         port = atoi(argv[2]);
45         if (port <= 0 || port > 65535)
46         {
47             fprintf(stderr, "无效的端口号: %s\n", argv[2]);
48             exit(EXIT_FAILURE);
49         }
50     }
51
52     // 设置信号处理
53     signal(SIGINT, signal_handler);
54     signal(SIGPIPE, SIG_IGN);
55 }
```

```

56 // 创建 socket
57 sockfd = socket(AF_INET, SOCK_STREAM, 0);
58 if (sockfd < 0)
59 {
60     perror("创建 socket 失败");
61     exit(EXIT_FAILURE);
62 }
63
64 // 配置服务器地址
65 memset(&server_addr, 0, sizeof(server_addr));
66 server_addr.sin_family = AF_INET;
67 server_addr.sin_port = htons(port);
68
69 if (inet_pton(AF_INET, server_ip, &server_addr.sin_addr) <= 0)
70 {
71     fprintf(stderr, "无效的服务器地址: %s\n", server_ip);
72     close(sockfd);
73     exit(EXIT_FAILURE);
74 }
75
76 printf("=====\\n");
77 printf("    TCP 聊天客户端\\n");
78 printf("    正在连接 %s:%d ...\\n", server_ip, port);
79 printf("=====\\n");
80
81 // 连接服务器
82 if (connect(sockfd, (struct sockaddr *)&server_addr, sizeof(
83     server_addr)) < 0)
84 {
85     perror("连接服务器失败");
86     close(sockfd);
87     exit(EXIT_FAILURE);
88 }
89
90 printf("连接成功! \\n\\n");
91
92 // 创建接收消息的线程
93 if (pthread_create(&recv_thread, NULL, receive_messages, NULL) !=
94     0)
95 {
96     perror("创建接收线程失败");
97     close(sockfd);
98     exit(EXIT_FAILURE);
99 }
100
101 // 主循环: 发送消息
102 while (client_running)
103 {
104     memset(buffer, 0, sizeof(buffer));
105
106     // 读取用户输入
107     if (fgets(buffer, sizeof(buffer), stdin) == NULL)
108     {

```

```

107         break;
108     }
109
110     // 检查是否退出
111     if (strncmp(buffer, "/quit", 5) == 0)
112     {
113         printf("正在退出...\n");
114         send(sockfd, buffer, strlen(buffer), 0);
115         break;
116     }
117
118     // 发送消息
119     if (strlen(buffer) > 0)
120     {
121         if (send(sockfd, buffer, strlen(buffer), 0) < 0)
122         {
123             if (client_running)
124             {
125                 perror("发送消息失败");
126             }
127             break;
128         }
129     }
130
131     // 清理
132     client_running = 0;
133     close(sockfd);
134     sockfd = -1;
135
136     // 等待接收线程结束
137     pthread_cancel(recv_thread);
138     pthread_join(recv_thread, NULL);
139
140     printf("\n已断开连接\n");
141     return 0;
142 }
143
144
145 // 接收消息的线程函数
146 void *receive_messages(void *arg)
147 {
148     (void)arg; // 抑制未使用参数警告
149     char buffer[BUFFER_SIZE];
150     int bytes_received;
151
152     while (client_running)
153     {
154         memset(buffer, 0, sizeof(buffer));
155         bytes_received = recv(sockfd, buffer, sizeof(buffer) - 1, 0);
156
157         if (bytes_received <= 0)
158         {
159             if (client_running)

```

```

160         {
161             printf("\n与服务器的连接已断开\n");
162             client_running = 0;
163         }
164         break;
165     }
166
167     // 显示收到的消息
168     printf("%s", buffer);
169     fflush(stdout);
170 }
171
172 return NULL;
173 }
174
175 // 信号处理函数
176 void signal_handler(int sig)
177 {
178     (void)sig; // 抑制未使用参数警告
179     printf("\n接收到退出信号\n");
180     client_running = 0;
181     if (sockfd >= 0)
182     {
183         const char *quit_msg = "/quit\n";
184         send(sockfd, quit_msg, strlen(quit_msg), 0);
185         close(sockfd);
186         sockfd = -1;
187     }
188     exit(0);
189 }
190
191 // 打印使用说明
192 void print_usage(const char *program)
193 {
194     printf("使用方法: %s [服务器IP] [端口]\n", program);
195     printf("  服务器IP: 服务器的IP地址 (默认: %s)\n", DEFAULT_SERVER);
196     printf("  端口:      服务器的端口号 (默认: %d)\n", DEFAULT_PORT);
197     printf("\n示例:\n");
198     printf("  %s          # 连接本地服务器\n", program);
199     printf("  %s 192.168.1.100    # 连接指定IP\n", program);
200     printf("  %s 192.168.1.100 9999 # 连接指定IP和端口\n", program);
201 }

```