

# 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：针对 ECHO 服务的 TCP 客户软件及并发服务器的实现

## 三、实验原理

### (一) TCP 协议与 Socket 编程

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 协议通过三次握手建立连接、四次挥手断开连接的机制，确保数据在网络中的可靠传输。在 TCP 通信模型中，服务器端需要先创建套接字并绑定到指定端口进行监听，等待客户端的连接请求；客户端则创建套接字后主动发起连接请求。

Socket (套接字) 是网络通信的基本操作单元，是支持 TCP/IP 协议的网络通信的基本操作接口。它是网络通信过程中端点的抽象表示，包含进行网络通信必需的五种信息：连接使用的协议、本地主机的 IP 地址、本地进程的协议端口、远程主机的 IP 地址以及远程进程的协议端口。

### (二) ECHO 协议

ECHO 协议 (RFC 862) 是一种简单的网络协议，标准口号为 7。其工作原理非常简单：服务器接收客户端发送的任何数据，然后将这些数据原样返回给客户端。ECHO 协议主要用于网络连通性测试和调试，可以验证网络链路是否正常工作。

### (三) TCP 客户端工作原理

TCP 客户端程序的基本工作流程如下：首先调用 `socket()` 函数创建一个套接字；然后使用 `connect()` 函数向服务器发起连接请求，此过程中会执行 TCP 三次握手；连接建立后，使用 `send()` 和 `recv()` 函数进行数据的发送和接收；最后调用 `close()` 函数关闭套接字，释放资源。

### (四) 并发服务器原理

并发服务器是能够同时处理多个客户端请求的服务器程序。与迭代服务器（一次只能处理一个客户端）不同，并发服务器可以同时为多个客户端提供服务，大大提高了服务器的吞吐量和响应能力。

在 Unix/Linux 系统中，实现并发服务器有多种方式：多进程模型、多线程模型和 I/O 多路复用模型等。本实验采用多进程模型，使用 `fork()` 系统调用为每个客户端连接创建一个子进程来处理请求。

`fork()` 系统调用会创建一个与父进程几乎完全相同的子进程。调用 `fork()` 后，父进程和子进程将从 `fork()` 调用处继续执行，但 `fork()` 在父进程中返回子进程的 PID，在子进程中返回 0。通过这个返回值可以区分父子进程，从而让它们执行不同的任务。

为避免僵尸进程 (Zombie Process)，需要在父进程中处理 `SIGCHLD` 信号，使用 `waitpid()` 回收已终止的子进程资源。

## 四、实验目的

本实验包含两部分，实验目的分别如下：

### 实验一：针对 ECHO 服务的 TCP 客户软件的实现

- 1) 掌握客户端软件的工作原理
- 2) 掌握针对 ECHO 服务的 TCP 客户端软件的编程实现方法

### 实验二：并发的面向连接的 ECHO 服务器

- 1) 掌握服务端软件的工作原理
- 2) 掌握并发的面向连接的 ECHO 服务器软件的编程实现方法

## 五、实验内容

本实验包含两部分内容，实验内容分别如下：

### (一) 针对 ECHO 服务的 TCP 客户软件的实现

- 1) 使用指定编程工具编写针对 ECHO 服务的 TCP 客户端软件的源代码
- 2) 使用指定编程工具对代码进行编译和调试，生成执行程序
- 3) 在 PC 机上执行程序，检验程序访问 ECHO 服务的功能

### (二) 并发的面向连接的 ECHO 服务器软件的实现

- 1) 使用指定编程工具编写并发的面向连接的 ECHO 服务器软件的源代码
- 2) 使用指定编程工具对代码进行编译和调试，生成执行程序
- 3) 在 PC 机上执行程序，检验程序能提供并发的 ECHO 服务的功能

## 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：Ubuntu 22.04 LTS / Linux
- 编译器：GCC 11.4
- 开发工具：VS Code、vim
- 调试工具：GDB
- 版本控制：Git
- 构建工具：Make

## 七、实验步骤

### (一) 问题描述

#### 7.1.1. TCP ECHO 客户端

设计并实现一个 TCP ECHO 客户端程序，该程序能够：

- 连接到指定 IP 地址和端口的 ECHO 服务器
- 接收用户从键盘输入的文本信息
- 将文本信息通过 TCP 连接发送给服务器

- 接收服务器回显的数据并显示
- 验证发送和接收的数据一致性

### 7.1.2. 并发 ECHO 服务器

设计并实现一个并发的 TCP ECHO 服务器程序，该程序能够：

- 监听指定端口，接受多个客户端的连接请求
- 使用多进程模型同时处理多个客户端
- 对每个客户端实现 ECHO 功能：接收数据并原样返回
- 正确处理客户端断开连接的情况
- 避免僵尸进程的产生

## (二) 算法分析与概要设计

### 7.2.1. 客户端算法设计

**输入：**命令行参数（服务器 IP、端口号）、用户输入的文本消息

**输出：**连接状态、发送字节数、回显数据、验证结果

**算法流程：**

- 1) 参数解析：获取服务器 IP 和端口号
- 2) Socket 创建：调用 `socket()` 创建 TCP 套接字
- 3) 地址配置：填充 `sockaddr_in` 结构体
- 4) 连接建立：调用 `connect()` 发起 TCP 三次握手
- 5) 数据交互：循环执行“读取 → 发送 → 接收 → 验证”
- 6) 资源释放：调用 `close()` 关闭套接字

### 7.2.2. 并发服务器算法设计

**输入：**监听端口号、客户端发送的数据

**输出：**服务器状态、客户端连接信息、回显数据

**算法流程：**

- 1) 初始化：设置信号处理、创建监听 Socket、绑定端口、开始监听
- 2) 主循环（父进程）：`accept()` 等待连接、`fork()` 创建子进程、关闭连接 Socket
- 3) 客户端服务（子进程）：关闭监听 Socket、`recv()` 接收、`send()` 回显、`exit()` 退出

## (三) 核心算法的详细设计与实现

### 7.3.1. 客户端核心代码

**Socket 创建与连接：**

代码 1：客户端 Socket 创建与连接

```

1  /* 创建 TCP Socket */
2  sock_fd = socket(AF_INET, SOCK_STREAM, 0);
3
4  /* 配置服务器地址 */
5  memset(&server_addr, 0, sizeof(server_addr));
6  server_addr.sin_family = AF_INET;
7  server_addr.sin_port = htons(port);
8  inet_pton(AF_INET, argv[1], &server_addr.sin_addr);
9
10 /* 连接服务器 */
11 connect(sock_fd, (struct sockaddr *)&server_addr, sizeof(server_addr));

```

数据发送与接收:

代码 2: 数据收发与验证

```
1 /* 发送数据 */
2 send_len = send(sock_fd, send_buffer, len, 0);
3
4 /* 接收回显 */
5 recv_len = recv(sock_fd, recv_buffer, BUFFER_SIZE - 1, 0);
6
7 /* 验证一致性 */
8 if (strcmp(send_buffer, recv_buffer) == 0) {
9     printf("[验证] 回显数据与发送数据一致\n");
10 }
```

### 7.3.2. 并发服务器核心代码

信号处理函数:

代码 3: SIGCHLD 信号处理

```
1 void sigchld_handler(int signo) {
2     (void)signo;
3     while (waitpid(-1, NULL, WNOHANG) > 0);
4 }
5
6 /* 设置信号处理 */
7 struct sigaction sa;
8 sa.sa_handler = sigchld_handler;
9 sigemptyset(&sa.sa_mask);
10 sa.sa_flags = SA_RESTART | SA_NOCLDSTOP;
11 sigaction(SIGCHLD, &sa, NULL);
```

fork() 创建子进程:

代码 4: 多进程并发处理

```
1 pid = fork();
2 if (pid == 0) {
3     // 子进程: 处理客户端
4     close(server_fd);
5     handle_client(client_fd, &client_addr);
6     exit(EXIT_SUCCESS);
7 } else {
8     // 父进程: 继续监听
9     close(client_fd);
10 }
```

客户端处理函数:

代码 5: ECHO 处理函数

```
1 void handle_client(int client_fd, struct sockaddr_in *client_addr) {
2     char buffer[BUFFER_SIZE];
3     ssize_t bytes_received;
```

```

5     while ((bytes_received = recv(client_fd, buffer, BUFFER_SIZE-1, 0)) 
6         > 0) {
7         send(client_fd, buffer, bytes_received, 0); // 回显
8     }
9 }
```

## 八、实验数据及结果分析

### (一) TCP 客户软件功能验证

本节验证「针对 ECHO 服务的 TCP 客户软件」的功能，检验程序访问 ECHO 服务的功能。

#### 8.1.1. 客户端程序编译

使用 GCC 编译器对 TCP 客户端源代码进行编译，生成可执行程序。编译命令如下：

代码 6: 编译 TCP 客户端

```

1 make client
2 # 或手动编译: gcc -Wall -o echo_client echo_client.c
```

#### 8.1.2. 服务器启动与监听

为测试客户端功能，首先启动 ECHO 服务器。服务器程序启动后，依次执行以下步骤：

- 1) 创建 TCP 监听 Socket
- 2) 设置 SO\_REUSEADDR 选项，允许快速重启
- 3) 绑定到指定端口（默认 7777）
- 4) 调用 listen() 开始监听，等待客户端连接

```

○ airprofly@airprofly:/mnt/e/codeFiles/networkProgram/expr1$ ./echo_server
=====
TCP ECHO 服务器
=====
[信息] Socket 创建成功
[信息] 已绑定到端口 7777
[信息] 服务器正在监听端口 7777 ...
[信息] 按 Ctrl+C 停止服务器
-----
[信息] 客户端已连接: 127.0.0.1:45026
[接收] 来自 127.0.0.1: Hello,ECHO Server! (18 字节)
[发送] 已回显 18 字节
[信息] 客户端 127.0.0.1 断开连接
[信息] 服务器正常退出

```

图 1: ECHO 服务器启动

**结果分析：**从图1可以看出，服务器成功创建 Socket，绑定到 7777 端口并进入监听状态。程序输出显示各步骤均执行成功，服务器已准备好接受客户端连接。

#### 8.1.3. 客户端访问 ECHO 服务功能验证

客户端程序执行流程：

- 1) 程序启动，解析命令行参数（服务器 IP 和端口）
- 2) 创建 TCP Socket，配置服务器地址结构

- 3) 调用 `connect()` 建立连接（触发 TCP 三次握手）
- 4) 进入交互循环：读取用户输入 → 发送数据 → 接收回显 → 验证一致性
- 5) 用户输入退出命令时关闭 Socket 释放资源

```

o airprofly@airprofly:/mnt/e/codeFiles/networkProgram/expr1$ ./echo_client 127.0.0.1 7777
=====
TCP ECHO 客户端
=====
目标服务器: 127.0.0.1:7777

[信息] Socket 创建成功
[信息] 正在连接服务器 127.0.0.1:7777 ...
[信息] 连接成功！

提示: 输入要发送的消息, 输入 'quit' 或 'exit' 退出程序

发送> Hello,ECHO Server!
[发送] 已发送 18 字节
[接收] 收到 18 字节: Hello,ECHO Server!
[验证] ✓ 回显数据与发送数据一致

发送> |

```

图 2: 客户端访问 ECHO 服务

**结果分析：**从图2可以看出，客户端成功连接到服务器 127.0.0.1:7777。用户输入”Hello, ECHO Server!”后，客户端发送了 19 字节数据，并收到服务器回显的 19 字节数据。程序自动验证发送和接收的数据完全一致，证明客户端访问 ECHO 服务的功能正常工作。

#### 8.1.4. 服务端响应验证

服务器接收到客户端连接后的处理流程：

- 1) `accept()` 返回新的连接 Socket 和客户端地址信息
- 2) 循环调用 `recv()` 接收客户端数据
- 3) 调用 `send()` 将数据原样返回（ECHO）
- 4) 客户端断开时检测到连接关闭，结束处理

```

PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE
airprofly@airprofly:/mnt/e/codeFiles/networkProgram/expr1$ ./echo_client 127.0.0.1 7777
=====
目标服务器: 127.0.0.1:7777

[信息] Socket 创建成功
[信息] 正在连接服务器 127.0.0.1:7777 ...
[信息] 连接成功！

提示: 输入要发送的消息, 输入 'quit' 或 'exit' 退出程序

发送> Hello,ECHO Server!
[发送] 已发送 18 字节
[接收] 收到 18 字节: Hello,ECHO Server!
[验证] ✓ 回显数据与发送数据一致

发送> quit
[信息] 用户请求退出

[信息] 正在关闭连接...
[信息] 程序结束

```

图 3: 服务端接收与回显日志

**结果分析：**从图3可以看出，服务器正确显示了客户端的连接信息（IP 地址和端口号），记录了接收到的数据内容和字节数，并成功完成了回显操作。当客户端断开连接时，服务器也正确检测到了连接关闭事件。

## (二) 并发 ECHO 服务器功能验证

本节验证「并发的面向连接的 ECHO 服务器软件」的功能，检验程序能提供并发的 ECHO 服务的功能。

### 8.2.1. 并发服务器程序编译

使用 GCC 编译器对并发服务器源代码进行编译：

代码 7: 编译并发服务器

```
1 make
2 # 或手动编译: gcc -Wall -o echo_server echo_server.c
```

### 8.2.2. 并发服务器启动

并发服务器启动时的初始化流程：

- 1) 设置 SIGCHLD 信号处理函数，用于回收子进程避免僵尸进程
- 2) 创建 TCP 监听 Socket
- 3) 设置 SO\_REUSEADDR 选项
- 4) 绑定到指定端口（默认 8888）
- 5) 调用 listen() 开始监听

```
make: warning: Clock skew detected. Your build may be incomplete.
o airprofly@airprofly:/mnt/e/codeFiles/networkProgram/expr2$ ./echo_server
套接字创建成功
绑定端口 8888 成功
服务器正在监听端口 8888...
等待客户端连接...

[主进程] 接受来自 127.0.0.1:42556 的连接
[主进程] 创建子进程 18290 处理客户端

[子进程 18290] 开始处理客户端 127.0.0.1:42556
[主进程] 接受来自 127.0.0.1:41910 的连接
[主进程] 创建子进程 18773 处理客户端

[子进程 18773] 开始处理客户端 127.0.0.1:41910
[子进程 18773] 收到数据: Hello,World
[子进程 18773] 已回显数据
[子进程 18290] 收到数据: Hello,World
[子进程 18290] 已回显数据
[子进程 18290] 客户端 127.0.0.1:42556 已断开连接
[子进程 18773] 客户端 127.0.0.1:41910 已断开连接
```

图 4: 并发 ECHO 服务器启动

**结果分析：**从图4可以看出，并发服务器成功完成初始化，绑定到 8888 端口并进入监听状态。与迭代式服务器不同，并发服务器在启动时已设置好信号处理机制，为后续的多进程并发服务做好准备。

### 8.2.3. 并发 ECHO 服务功能验证

为验证服务器能提供并发的 ECHO 服务功能，同时启动多个客户端进行测试。当有客户端连接时，服务器的并发处理流程：

- 1) 主进程调用 accept() 接受客户端连接
- 2) 主进程调用 fork() 创建子进程
- 3) 子进程：关闭监听 Socket，专注处理当前客户端的 ECHO 请求
- 4) 父进程：关闭连接 Socket，继续 accept() 等待新连接
- 5) 子进程完成服务后 exit() 退出，父进程通过 SIGCHLD 信号回收资源

```

PROBLEMS    OUTPUT    TERMINAL    PORTS    DEBUG CONSOLE

● airprofly@airprofly:/mnt/e/codeFiles/networkProgram/expr2$ ./echo_client
套接字创建成功
正在连接到服务器 127.0.0.1:8888...
连接成功!
输入要发送的内容 (输入 'quit' 退出) :

> Hello,World
服务器回显: Hello,World
> quit
正在断开连接...
连接已关闭
○ airprofly@airprofly:/mnt/e/codeFiles/networkProgram/expr2$ 

```

图 5: 多客户端同时连接测试

**结果分析:** 从图4和图5可以看出, 多个客户端能够同时连接到服务器。服务器为每个客户端创建了独立的子进程进行处理, 从输出的进程 ID 可以看出不同客户端由不同的子进程服务, 验证了服务器能提供并发的 ECHO 服务功能。

#### 8.2.4. 并发数据交互与进程管理验证

并发服务器同时处理多个客户端时的运行状态:

- 1) 每个子进程独立处理各自客户端的 recv() 和 send() 操作
- 2) 各客户端之间互不干扰, 数据隔离
- 3) 客户端断开时对应子进程正常退出
- 4) 父进程通过 waitpid() 回收已终止子进程的资源

```

● airprofly@airprofly:/mnt/e/codeFiles/networkProgram/expr2$ ./echo_client
套接字创建成功
正在连接到服务器 127.0.0.1:8888...
连接成功!
输入要发送的内容 (输入 'quit' 退出) :

> Hello,World
服务器回显: Hello,World
> quit
正在断开连接...
连接已关闭
○ airprofly@airprofly:/mnt/e/codeFiles/networkProgram/expr2$ 

```

图 6: 并发数据交互与子进程管理

**结果分析:** 从图6可以看出, 多个子进程分别处理各自客户端的 ECHO 请求, 每个子进程都能正确接收并回显数据。当客户端断开连接时, 子进程检测到连接关闭并正确退出。父进程通过 SIGCHLD 信号处理函数自动回收子进程资源, 系统中没有产生僵尸进程, 证明并发服务器的进程管理机制工作正常。

## 九、实验结论

通过本次实验, 达成了全部实验目标:

### 实验一: 针对 ECHO 服务的 TCP 客户软件的实现

- 1) 掌握客户端软件的工作原理: 通过实现 TCP ECHO 客户端, 深入理解了客户端程序的工作流程, 包括 Socket 创建、地址配置、连接建立、数据收发和资源释放

等核心步骤。实验中成功使用 `connect()` 建立 TCP 连接，使用 `send()/recv()` 进行数据交互，验证了客户端软件的工作原理。

- 2) 掌握针对 ECHO 服务的 TCP 客户端软件的编程实现方法：成功编写了完整的 TCP 客户端程序，实现了连接服务器、发送用户输入、接收回显数据、验证数据一致性等功能。程序能够正确访问 ECHO 服务，发送的数据与接收的回显数据完全一致。

## 实验二：并发的面向连接的 ECHO 服务器

- 1) 掌握服务端软件的工作原理：通过实现并发 ECHO 服务器，理解了服务端程序的工作机制，包括监听 Socket 与连接 Socket 的区别、`accept()` 阻塞等待连接、多进程并发处理模型等核心概念。
- 2) 掌握并发的面向连接的 ECHO 服务器软件的编程实现方法：成功使用 `fork()` 系统调用实现多进程并发服务器，能够同时处理多个客户端的连接请求。通过 `SIGCHLD` 信号处理机制避免了僵尸进程的产生，服务器运行稳定可靠。

实验结果表明，设计的 TCP ECHO 客户端和并发服务器程序功能完整、运行稳定，达到了预期目标。

## 十、总结及心得体会

通过本次实验，我对 TCP/IP 网络编程有了更加深入和系统的理解，同时在 Socket 编程、多进程并发处理等方面的能力得到了显著提升。以下是我的主要收获和心得体会：

- 1) 理论联系实际：将 TCP/IP 协议、Socket 编程等理论知识应用到实际编程中，加深了对网络通信原理的理解。通过亲手编写客户端和服务器程序，真正体会到了 TCP 三次握手、四次挥手等机制在程序中的体现
- 2) 客户端/服务器模型：深刻理解了 C/S 架构的工作方式，客户端主动连接、服务器被动监听的模式
- 3) 并发编程技能：掌握了使用 `fork()` 实现多进程并发服务器的方法，理解了父子进程的分工协作
- 4) 信号处理机制：认识到正确处理 `SIGCHLD` 信号对于避免僵尸进程的重要性
- 5) 字节序问题：深刻理解了网络字节序与主机字节序的区别，以及 `htonl()`、`inet_nton()` 等转换函数的重要性
- 6) 资源管理意识：认识到正确关闭 Socket、释放资源的重要性，避免资源泄漏
- 7) 调试能力：在实验过程中遇到连接失败、僵尸进程等问题，通过分析和调试提升了问题排查能力

总之，本次实验不仅让我掌握了 TCP Socket 编程和多进程并发服务器的核心技术，更重要的是培养了系统性思考网络编程问题的能力。这些经验和技能将为今后学习更高级的网络编程技术（如 I/O 多路复用、异步编程等）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 TCP ECHO 客户端和并发服务器的基本功能，但在性能优化、功能扩展、代码健壮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) 多线程模型：可以使用 `pthread` 库实现多线程服务器，减少进程创建的开销，提高并发处理效率
- 2) 线程池/进程池：预先创建一定数量的线程/进程，避免频繁创建销毁的开销，提升服务器响应速度

- 3) **I/O 多路复用**: 使用 `select()`/`poll()`/`epoll()` 实现高并发服务器，在单线程中处理大量并发连接
- 4) **超时机制**: 设置 Socket 超时选项 (`SO_RCVTIMEO`/`SO_SNDFTIMEO`)，避免程序在网络异常时无限阻塞
- 5) **IPv6 支持**: 使用 `AF_INET6` 协议族支持 IPv6，适应现代网络环境
- 6) **加密传输**: 结合 OpenSSL 等库实现 SSL/TLS 安全传输，保护数据隐私
- 7) **日志系统**: 添加完善的日志记录功能，包括日志级别、时间戳、文件输出等，便于问题追踪和性能分析
- 8) **配置文件支持**: 将端口号、缓冲区大小等参数配置化，提高程序的灵活性

以上改进方向涵盖了性能优化、功能扩展、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
expr1/echo_server.c	实验一	迭代式 ECHO 服务器代码
expr1/echo_client.c	实验一	ECHO 客户端代码
expr2/echo_server.c	实验二	并发 ECHO 服务器代码
expr2/echo_client.c	实验二	ECHO 客户端代码

### (一) 实验一：迭代式服务器与客户端

代码 8: echo\_server.c (迭代式服务器)

```
1  /**
2  * TCP ECHO 服务器程序
3  *
4  * 功能: 接收客户端发送的数据, 并将数据原样返回 (回显)
5  *
6  * 编译: gcc -o echo_server echo_server.c
7  * 运行: ./echo_server [端口号]
8  * 示例: ./echo_server 7777
9  */
10
11 #include <arpa/inet.h>
12 #include <errno.h>
13 #include <netinet/in.h>
14 #include <stdio.h>
15 #include <stdlib.h>
16 #include <string.h>
17 #include <sys/socket.h>
18 #include <sys/types.h>
19 #include <unistd.h>
20
21 /* 常量定义 */
22 #define BUFFER_SIZE 1024 /* 缓冲区大小 */
23 #define DEFAULT_PORT 7777 /* 默认监听端口 (使用非特权端口便于测试) */
24 #define BACKLOG 5 /* 连接队列长度 */
25
26 /**
27 * 打印使用说明
28 */
29 void print_usage(const char *program_name) {
30     printf("用法: %s [端口号]\n", program_name);
31     printf("示例: %s 7777\n", program_name);
32     printf("说明: 端口号默认为 %d\n", DEFAULT_PORT);
33 }
34
35 /**
36 * 处理客户端连接
37 */
```

```

38 void handle_client(int client_fd, struct sockaddr_in *client_addr) {
39     char buffer[BUFFER_SIZE];
40     ssize_t recv_len;
41     char client_ip[INET_ADDRSTRLEN];
42
43     /* 获取客户端 IP 地址 */
44     inet_ntop(AF_INET, &client_addr->sin_addr, client_ip, INET_ADDRSTRLEN
45             );
46     printf("[信息] 客户端已连接: %s:%d\n", client_ip,
47            ntohs(client_addr->sin_port));
48
49     /* 循环接收并回显数据 */
50     while ((recv_len = recv(client_fd, buffer, BUFFER_SIZE - 1, 0)) > 0)
51     {
52         buffer[recv_len] = '\0';
53         printf("[接收] 来自 %s: %s (%zd 字节)\n", client_ip, buffer,
54                recv_len);
55
56         /* 将数据原样返回给客户端 */
57         if (send(client_fd, buffer, recv_len, 0) < 0) {
58             perror("发送数据失败");
59             break;
60         }
61         printf("[发送] 已回显 %zd 字节\n", recv_len);
62     }
63
64     if (recv_len < 0) {
65         perror("接收数据失败");
66     } else {
67         printf("[信息] 客户端 %s 断开连接\n", client_ip);
68     }
69
70 /**
71 * 主函数
72 */
73 int main(int argc, char *argv[]) {
74     int server_fd, client_fd;          /* 服务器和客户端 Socket */
75     struct sockaddr_in server_addr;    /* 服务器地址 */
76     struct sockaddr_in client_addr;    /* 客户端地址 */
77     socklen_t client_len;             /* 客户端地址长度 */
78     int port;                         /* 监听端口 */
79     int opt = 1;                      /* Socket 选项值 */
80
81     /* 解析端口号 */
82     if (argc >= 2) {
83         port = atoi(argv[1]);
84         if (port <= 0 || port > 65535) {
85             fprintf(stderr, "错误: 无效的端口号 '%s'\n", argv[1]);
86             print_usage(argv[0]);
87             return EXIT_FAILURE;
88         }
89     } else {

```

```

88     port = DEFAULT_PORT;
89 }
90
91 printf("=====\\n");
92 printf("    TCP ECHO 服务器\\n");
93 printf("=====\\n");
94
95 /* 步骤1: 创建 TCP Socket */
96 server_fd = socket(AF_INET, SOCK_STREAM, 0);
97 if (server_fd < 0) {
98     perror("创建 Socket 失败");
99     return EXIT_FAILURE;
100}
101printf("[信息] Socket 创建成功\\n");
102
103/* 设置 SO_REUSEADDR 选项, 避免 "Address already in use" 错误 */
104if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt))
105    ) < 0) {
106    perror("设置 Socket 选项失败");
107    close(server_fd);
108    return EXIT_FAILURE;
109}
110
111/* 步骤2: 配置服务器地址结构 */
112memset(&server_addr, 0, sizeof(server_addr));
113server_addr.sin_family = AF_INET;
114server_addr.sin_addr.s_addr = INADDR_ANY; /* 监听所有网络接口 */
115server_addr.sin_port = htons(port);
116
117/* 步骤3: 绑定地址 */
118if (bind(server_fd, (struct sockaddr *)&server_addr, sizeof(
119    server_addr)) <
120    0) {
121    perror("绑定地址失败");
122    close(server_fd);
123    return EXIT_FAILURE;
124}
125printf("[信息] 已绑定到端口 %d\\n", port);
126
127/* 步骤4: 开始监听 */
128if (listen(server_fd, BACKLOG) < 0) {
129    perror("监听失败");
130    close(server_fd);
131    return EXIT_FAILURE;
132}
133printf("[信息] 服务器正在监听端口 %d ...\\n", port);
134printf("[信息] 按 Ctrl+C 停止服务器\\n");
135printf("-----\\n");
136
137/* 步骤5: 循环接受客户端连接 */
138while (1) {
    client_len = sizeof(client_addr);

```

```

139     /* 接受新连接 */
140     client_fd = accept(server_fd, (struct sockaddr *)&client_addr, &
141                         client_len);
142     if (client_fd < 0) {
143         perror("接受连接失败");
144         continue;
145     }
146     /* 处理客户端请求 */
147     handle_client(client_fd, &client_addr);
148
149     /* 关闭客户端连接 */
150     close(client_fd);
151 }
152
153 /* 关闭服务器 Socket (实际上不会执行到这里) */
154 close(server_fd);
155
156 return EXIT_SUCCESS;
157 }
```

代码 9: echo\_client.c (客户端)

```

1 /**
2 * TCP ECHO 客户端程序
3 *
4 * 功能: 连接到 ECHO 服务器, 发送用户输入的消息, 接收并显示服务器回显的
5 *       数据
6 *
7 * 编译: gcc -o echo_client echo_client.c
8 * 运行: ./echo_client <服务器IP> <端口号>
9 * 示例: ./echo_client 127.0.0.1 7
10 */
11 #include <arpa/inet.h>
12 #include <errno.h>
13 #include <netinet/in.h>
14 #include <stdio.h>
15 #include <stdlib.h>
16 #include <string.h>
17 #include <sys/socket.h>
18 #include <sys/types.h>
19 #include <unistd.h>
20
21 /* 常量定义 */
22 #define BUFFER_SIZE 1024 /* 缓冲区大小 */
23 #define DEFAULT_PORT 7 /* ECHO 服务默认端口 */
24
25 /**
26 * 打印使用说明
27 */
28 void print_usage(const char *program_name) {
29     printf("用法: %s <服务器IP> [端口号]\n", program_name);
```

```

30     printf("示例: %s 127.0.0.1 7\n", program_name);
31     printf("说明: 端口号默认为 7 (ECHO 服务标准端口)\n");
32 }
33
34 /**
35 * 主函数
36 */
37 int main(int argc, char *argv[]) {
38     int sock_fd;                      /* Socket 文件描述符 */
39     struct sockaddr_in server_addr;   /* 服务器地址结构 */
40     char send_buffer[BUFFER_SIZE];   /* 发送缓冲区 */
41     char recv_buffer[BUFFER_SIZE];   /* 接收缓冲区 */
42     int port;                        /* 服务器端口号 */
43     ssize_t send_len, recv_len;      /* 发送和接收的字节数 */
44
45     /* 步骤1: 参数检查 */
46     if (argc < 2) {
47         print_usage(argv[0]);
48         return EXIT_FAILURE;
49     }
50
51     /* 解析端口号, 如果未指定则使用默认端口 */
52     if (argc >= 3) {
53         port = atoi(argv[2]);
54         if (port <= 0 || port > 65535) {
55             fprintf(stderr, "错误: 无效的端口号 '%s', 端口范围应为 1-65535\n"
56                     ,
57                     argv[2]);
58             return EXIT_FAILURE;
59         }
60     } else {
61         port = DEFAULT_PORT;
62     }
63
64     printf("=====TCP ECHO 客户端\n");
65     printf("=====目标服务器: %s:%d\n\n", argv[1], port);
66
67     /* 步骤2: 创建 TCP Socket */
68     sock_fd = socket(AF_INET, SOCK_STREAM, 0);
69     if (sock_fd < 0) {
70         perror("创建 Socket 失败");
71         return EXIT_FAILURE;
72     }
73     printf("[信息] Socket 创建成功\n");
74
75     /* 步骤3: 配置服务器地址结构 */
76     memset(&server_addr, 0, sizeof(server_addr));
77     server_addr.sin_family = AF_INET;
78     server_addr.sin_port = htons(port);
79
80     /* 转换 IP 地址 */

```

```

82     if (inet_pton(AF_INET, argv[1], &server_addr.sin_addr) <= 0) {
83         fprintf(stderr, "错误：无效的 IP 地址 '%s'\n", argv[1]);
84         close(sock_fd);
85         return EXIT_FAILURE;
86     }
87
88     /* 步骤4：连接服务器 */
89     printf("[信息] 正在连接服务器 %s:%d ...\n", argv[1], port);
90     if (connect(sock_fd, (struct sockaddr *)&server_addr, sizeof(
91         server_addr)) <
92         0) {
93         perror("连接服务器失败");
94         close(sock_fd);
95         return EXIT_FAILURE;
96     }
97     printf("[信息] 连接成功！\n\n");
98
99     /* 步骤5：数据交互循环 */
100    printf("提示：输入要发送的消息，输入 'quit' 或 'exit' 退出程序\n");
101    printf("-----\n");
102
103    while (1) {
104        /* 显示提示符并读取用户输入 */
105        printf("\n发送> ");
106        fflush(stdout);
107
108        /* 读取一行输入 */
109        if (fgets(send_buffer, BUFFER_SIZE, stdin) == NULL) {
110            printf("\n[信息] 检测到输入结束，退出程序\n");
111            break;
112        }
113
114        /* 移除换行符 */
115        size_t len = strlen(send_buffer);
116        if (len > 0 && send_buffer[len - 1] == '\n') {
117            send_buffer[len - 1] = '\0';
118            len--;
119        }
120
121        /* 检查是否为空输入 */
122        if (len == 0) {
123            printf("[提示] 输入为空，请重新输入\n");
124            continue;
125        }
126
127        /* 检查退出命令 */
128        if (strcmp(send_buffer, "quit") == 0 || strcmp(send_buffer, "exit")
129            == 0) {
130            printf("[信息] 用户请求退出\n");
131            break;
132        }
133
134        /* 发送数据到服务器 */

```

```

133     send_len = send(sock_fd, send_buffer, len, 0);
134     if (send_len < 0) {
135         perror("发送数据失败");
136         break;
137     }
138     printf("[发送] 已发送 %zd 字节\n", send_len);
139
140     /* 接收服务器回显 */
141     memset(recv_buffer, 0, BUFFER_SIZE);
142     recv_len = recv(sock_fd, recv_buffer, BUFFER_SIZE - 1, 0);
143
144     if (recv_len < 0) {
145         perror("接收数据失败");
146         break;
147     } else if (recv_len == 0) {
148         printf("[信息] 服务器关闭了连接\n");
149         break;
150     }
151
152     /* 显示接收到的回显数据 */
153     recv_buffer[recv_len] = '\0';
154     printf("[接收] 收到 %zd 字节: %s\n", recv_len, recv_buffer);
155
156     /* 验证回显是否正确 */
157     if (strcmp(send_buffer, recv_buffer) == 0) {
158         printf("[验证] 回显数据与发送数据一致\n");
159     } else {
160         printf("[验证] 回显数据与发送数据不一致\n");
161     }
162 }
163
164 /* 步骤6: 关闭连接, 清理资源 */
165 printf("\n-----\n");
166 printf("[信息] 正在关闭连接...\n");
167 close(sock_fd);
168 printf("[信息] 程序结束\n");
169
170 return EXIT_SUCCESS;
171 }
```

## (二) 实验二: 并发服务器与客户端

代码 10: echo\_server.c (并发服务器)

```

1 /**
2 * 并发的面向连接的ECHO服务器
3 *
4 * 功能: 接收客户端发送的数据, 并将其原样返回(回显)
5 * 实现方式: 使用fork()创建子进程处理每个客户端连接, 实现并发服务
6 */
7
8 #include <arpa/inet.h>
9 #include <errno.h>
```

```

10 #include <netinet/in.h>
11 #include <signal.h>
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <string.h>
15 #include <sys/socket.h>
16 #include <sys/types.h>
17 #include <sys/wait.h>
18 #include <unistd.h>
19
20 #define PORT 8888          // 服务器监听端口
21 #define BUFFER_SIZE 1024 // 缓冲区大小
22 #define BACKLOG 10        // 最大等待连接队列长度
23
24 /**
25 * 信号处理函数：处理子进程终止信号，避免僵尸进程
26 */
27 void sigchld_handler(int signo) {
28     (void)signo; // 避免未使用参数警告
29     // 回收所有已终止的子进程
30     while (waitpid(-1, NULL, WNOHANG) > 0)
31         ;
32 }
33
34 /**
35 * 错误处理函数
36 */
37 void error_exit(const char *msg) {
38     perror(msg);
39     exit(EXIT_FAILURE);
40 }
41
42 /**
43 * 处理客户端连接的函数
44 * 实现ECHO功能：接收数据并原样返回
45 */
46 void handle_client(int client_fd, struct sockaddr_in *client_addr) {
47     char buffer[BUFFER_SIZE];
48     ssize_t bytes_received;
49     char client_ip[INET_ADDRSTRLEN];
50
51     // 获取客户端IP地址
52     inet_ntop(AF_INET, &(client_addr->sin_addr), client_ip,
53               INET_ADDRSTRLEN);
54     printf("[子进程 %d] 开始处理客户端 %s:%d\n", getpid(), client_ip,
55            ntohs(client_addr->sin_port));
56
57     // 循环接收并回显数据
58     while ((bytes_received = recv(client_fd, buffer, BUFFER_SIZE - 1, 0))
59             > 0) {
60         buffer[bytes_received] = '\0';
61         printf("[子进程 %d] 收到数据: %s", getpid(), buffer);

```

```

61     // 将数据原样发送回客户端 (ECHO)
62     if (send(client_fd, buffer, bytes_received, 0) < 0) {
63         perror("发送数据失败");
64         break;
65     }
66     printf("[子进程 %d] 已回显数据\n", getpid());
67 }
68
69     if (bytes_received < 0) {
70         perror("接收数据失败");
71     } else {
72         printf("[子进程 %d] 客户端 %s:%d 已断开连接\n", getpid(), client_ip
73             ,
74             ntohs(client_addr->sin_port));
75     }
76
77     close(client_fd);
78 }
79
80 int main(int argc, char *argv[]) {
81     int server_fd, client_fd;
82     struct sockaddr_in server_addr, client_addr;
83     socklen_t client_len;
84     pid_t pid;
85     int port = PORT;
86
87     // 可通过命令行参数指定端口
88     if (argc > 1) {
89         port = atoi(argv[1]);
90         if (port <= 0 || port > 65535) {
91             fprintf(stderr, "无效的端口号: %s\n", argv[1]);
92             exit(EXIT_FAILURE);
93         }
94     }
95
96     // 设置SIGCHLD信号处理，避免僵尸进程
97     struct sigaction sa;
98     sa.sa_handler = sigchld_handler;
99     sigemptyset(&sa.sa_mask);
100    sa.sa_flags = SA_RESTART | SA_NOCLDSTOP;
101    if (sigaction(SIGCHLD, &sa, NULL) < 0) {
102        error_exit("设置信号处理失败");
103    }
104
105    // 1. 创建TCP套接字
106    server_fd = socket(AF_INET, SOCK_STREAM, 0);
107    if (server_fd < 0) {
108        error_exit("创建套接字失败");
109    }
110    printf("套接字创建成功\n");
111
112    // 设置套接字选项，允许地址重用
113    int opt = 1;

```

```

113     if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR, &opt, sizeof(opt))
114         ) < 0) {
115     error_exit("设置套接字选项失败");
116 }
117 // 2. 绑定地址和端口
118 memset(&server_addr, 0, sizeof(server_addr));
119 server_addr.sin_family = AF_INET;
120 server_addr.sin_addr.s_addr = INADDR_ANY; // 监听所有网络接口
121 server_addr.sin_port = htons(port);
122
123 if (bind(server_fd, (struct sockaddr *)&server_addr, sizeof(
124     server_addr)) <
125     0) {
126     error_exit("绑定地址失败");
127 }
128 printf("绑定端口 %d 成功\n", port);
129
130 // 3. 开始监听
131 if (listen(server_fd, BACKLOG) < 0) {
132     error_exit("监听失败");
133 }
134 printf("服务器正在监听端口 %d...\n", port);
135 printf("等待客户端连接...\n\n");
136
137 // 4. 主循环: 接受连接并创建子进程处理
138 while (1) {
139     client_len = sizeof(client_addr);
140
141     // 接受客户端连接
142     client_fd = accept(server_fd, (struct sockaddr *)&client_addr, &
143         client_len);
144     if (client_fd < 0) {
145         if (errno == EINTR) {
146             // 被信号中断, 继续等待
147             continue;
148         }
149         perror("接受连接失败");
150         continue;
151     }
152     char client_ip[INET_ADDRSTRLEN];
153     inet_ntop(AF_INET, &(client_addr.sin_addr), client_ip,
154         INET_ADDRSTRLEN);
155     printf("[主进程] 接受来自 %s:%d 的连接\n", client_ip,
156         ntohs(client_addr.sin_port));
157
158     // 创建子进程处理客户端请求
159     pid = fork();
160     if (pid < 0) {
161         perror("创建子进程失败");
162         close(client_fd);
163         continue;

```

```

162     } else if (pid == 0) {
163         // 子进程
164         close(server_fd); // 子进程不需要监听套接字
165         handle_client(client_fd, &client_addr);
166         exit(EXIT_SUCCESS);
167     } else {
168         // 父进程
169         printf("[主进程] 创建子进程 %d 处理客户端\n\n", pid);
170         close(client_fd); // 父进程不需要客户端套接字
171     }
172 }
173
174 close(server_fd);
175 return 0;
176 }
```

代码 11: echo\_client.c (客户端)

```

1 /**
2 * ECHO 客户端程序
3 *
4 * 功能：连接到ECHO服务器，发送用户输入的数据，并接收服务器回显的数据
5 */
6
7 #include <arpa/inet.h>
8 #include <netinet/in.h>
9 #include <stdio.h>
10 #include <stdlib.h>
11 #include <string.h>
12 #include <sys/socket.h>
13 #include <sys/types.h>
14 #include <unistd.h>
15
16 #define DEFAULT_PORT 8888      // 默认服务器端口
17 #define DEFAULT_IP "127.0.0.1" // 默认服务器IP
18 #define BUFFER_SIZE 1024       // 缓冲区大小
19
20 /**
21 * 错误处理函数
22 */
23 void error_exit(const char *msg) {
24     perror(msg);
25     exit(EXIT_FAILURE);
26 }
27
28 int main(int argc, char *argv[]) {
29     int sock_fd;
30     struct sockaddr_in server_addr;
31     char send_buffer[BUFFER_SIZE];
32     char recv_buffer[BUFFER_SIZE];
33     ssize_t bytes_sent, bytes_received;
34
35     const char *server_ip = DEFAULT_IP;
```

```

36 int port = DEFAULT_PORT;
37
38 // 解析命令行参数
39 if (argc > 1) {
40     server_ip = argv[1];
41 }
42 if (argc > 2) {
43     port = atoi(argv[2]);
44     if (port <= 0 || port > 65535) {
45         fprintf(stderr, "无效的端口号: %s\n", argv[2]);
46         exit(EXIT_FAILURE);
47     }
48 }
49
50 // 1. 创建TCP套接字
51 sock_fd = socket(AF_INET, SOCK_STREAM, 0);
52 if (sock_fd < 0) {
53     error_exit("创建套接字失败");
54 }
55 printf("套接字创建成功\n");
56
57 // 2. 设置服务器地址
58 memset(&server_addr, 0, sizeof(server_addr));
59 server_addr.sin_family = AF_INET;
60 server_addr.sin_port = htons(port);
61
62 if (inet_pton(AF_INET, server_ip, &server_addr.sin_addr) <= 0) {
63     fprintf(stderr, "无效的IP地址: %s\n", server_ip);
64     close(sock_fd);
65     exit(EXIT_FAILURE);
66 }
67
68 // 3. 连接到服务器
69 printf("正在连接到服务器 %s:%d...\n", server_ip, port);
70 if (connect(sock_fd, (struct sockaddr *)&server_addr, sizeof(
71     server_addr)) <
72     0) {
73     error_exit("连接服务器失败");
74 }
75 printf("连接成功！\n");
76 printf("输入要发送的内容（输入 'quit' 退出）:\n\n");
77
78 // 4. 循环发送和接收数据
79 while (1) {
80     printf("> ");
81     fflush(stdout);
82
83     // 读取用户输入
84     if (fgets(send_buffer, BUFFER_SIZE, stdin) == NULL) {
85         printf("\n检测到EOF，退出...\n");
86         break;
87 }

```

```
88 // 检查是否退出
89 if (strncmp(send_buffer, "quit", 4) == 0) {
90     printf("正在断开连接...\n");
91     break;
92 }
93
94 // 发送数据到服务器
95 size_t msg_len = strlen(send_buffer);
96 bytes_sent = send(sock_fd, send_buffer, msg_len, 0);
97 if (bytes_sent < 0) {
98     perror("发送数据失败");
99     break;
100 }
101
102 // 接收服务器回显的数据
103 bytes_received = recv(sock_fd, recv_buffer, BUFFER_SIZE - 1, 0);
104 if (bytes_received < 0) {
105     perror("接收数据失败");
106     break;
107 } else if (bytes_received == 0) {
108     printf("服务器断开连接\n");
109     break;
110 }
111
112 recv_buffer[bytes_received] = '\0';
113 printf("服务器回显: %s", recv_buffer);
114 }
115
116 // 5. 关闭套接字
117 close(sock_fd);
118 printf("连接已关闭\n");
119
120 return 0;
121 }
```