计算机网络实验报告

Lab1 SOCKET编程

网络空间安全学院 物联网工程

2110951 梁晓储

代码已发布到github：https://github.com/WangshuXC/Computer_network

一、实验要求

（1）给出你聊天协议的完整说明。

（2）利用C或C++语言，使用基本的Socket函数完成程序。不允许使用CSocket等封装后的类编写程序。

（3）使用流式套接字、采用多线程（或多进程）方式完成程序。

（4）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。

（5）完成的程序应能支持多人聊天，支持英文和中文聊天。

（6）编写的程序应该结构清晰，具有较好的可读性。

（7）在实验中观察是否有数据的丢失，提交源码和实验报告。

二、协议说明

按照实验要求，基于实验目标，采取了以下的协议方案：

- 使用TCP传输协议，选用流式套接字，采用多线程方式
- 分别设计两个程序，一个作为服务器端，另一个作为客户端。使用时需首先启动`server.exe`，再启动若干个`client.exe`，`client`将消息发送到`server`，再由`server`将消息发送到全部的`client`，从而完成通信
- 为了保证通讯质量，程序在`server`设置了最大连接数`MaxClient`、最大缓冲区`BufSize`。当最大连接数上限时，无法再连接；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
- 服务器和客户端之间通过`send`和`recv`函数来发送和接收消息数据，这些数据的格式和大小都受到socket语法规则的约束
- 完成了日志输出功能，能够在控制台查看打印出的程序运行日志，方便调试

服务器端

- 服务器端自动检索本机ipv4并且用于绑定服务端ip，同时监听指定端口（在代码中为`6262`），等待客户端的连接请求
- 每个客户端连接后，服务器为其创建一个独立的线程（`ThreadFunction`函数），并使用一个新的套接字来处理该客户端的消息接收和分发
- 服务器端可以同时接受多个客户端的连接请求，最多支持`MaxClient - 1`个客户端同时连接，当达到最大连接数时，不再接入新的客户端（即不为它创建新线程），当有客户端连接或是退出时会将客户端id以及连接/退出时间打印到日志信息，同时刷新当前客户端连接数

- 服务器的每个线程接收它所负责客户端的消息，并将消息连同客户端id、发送人ipv4地址、发送消息时间戳等信息发送给其他客户端并打印到日志信息
- 服务器维护一个`Clients`数组，用于存储每个客户端的套接字，以便进行消息的收发

客户端

- 客户端创建一个套接字，询问用户服务端是否在本地，若是则自动获取本地ipv4地址并且绑定为服务端地址；若不是则让用户手动输入服务端ipv4地址，从而实现跨设备通信
- 客户端也创建一个独立的线程（`recvThread`函数）来接收服务器发送的消息，并将其显示在控制台上；该线程也让客户端能够同时发送消息和接受消息
- 客户端可以通过在控制台中输入文本消息（支持中文、英文、数字），然后将其发送给服务端，再由服务端分发给其他客户端

消息格式：

- 消息格式为`Id[ClientID] 在 年-月-日 时:分:秒 发送：\n "Message"`，其中`ClientID`是客户端的唯一标识（由套接字充当），`Message`是实际消息内容
- 客户端发送消息时，服务器接收到消息后会在服务器端控制台显示，并将消息转发给其他所有客户端，以便实现聊天功能
- 服务器端也会接收来自其他客户端的消息，并将其显示在服务器端控制台上

退出机制：

- 客户端可以输入`exit`来退出聊天程序，此时客户端会关闭连接，并在服务器端显示客户端退出的消息
- 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

三、功能实现与代码分析

（1）服务器端

该部分实现了一个多线程的聊天服务器，允许多个客户端连接并在客户端之间实现消息广播。通过创建线程处理每个客户端的通信，实现了同时处理多个客户端的连接请求和消息传递。

多线程通信

本次实验中，通过编写线程函数，借助宏定义，实现了有连接上限的多线程通信。

代码开头，实现了对连接数、客户端套接字数组等的定义

```
#define PORT 6262  //端口号
#define BufSize 1024  //缓冲区大小
#define MaxClient 5 //最大连接数

SOCKET Clients[MaxClient];    // 客户端socket数组
SOCKET Server;                // 服务器端socket
SOCKADDR_IN clientAddrs[MaxClient]; // 客户端地址数组
SOCKADDR_IN serverAddr;             // 定义服务器地址

int currentConnections = 0; // 当前连接的客户数
bool connectCondition[MaxClient] = {}; // 每一个连接的情况
```

编写了一个线程函数。每个客户端连接都会创建一个线程，函数负责处理该客户端的通信。

- 该函数首先通过传递给线程的参数 `lpParameter` 获取当前连接的 `Clients` 索引
- 使用 `recv` 函数接收客户端发送的消息，并根据协议的要求进行处理
- 如果接收成功，将消息格式化为特定格式，并通过 `send` 函数发送给其他连接的客户端，实现消息广播
- 如果客户端主动关闭连接，会通过 `recv` 返回值判断，并在服务端记录客户端退出的时间

```
DWORD WINAPI ThreadFunction(LPVOID lpParameter) // 线程函数
{
    int recvByt = 0;
    char recvMsg[msgSize]; // 接收缓冲区
    char sendMsg[msgSize]; // 发送缓冲区

    int num = (int)lpParameter; // 当前连接的索引

    // 发送连接成功的提示消息
    snprintf(sendMsg, sizeof(sendMsg), "你的id是 \x1b[32m%d\x1b[0m\n",
Clients[num]);
    send(Clients[num], sendMsg, strlen(sendMsg), 0);

    // 循环接收信息
    while (true)
    {
        Sleep(100); // 延时100ms
        recvByt = recv(Clients[num], recvMsg, sizeof(recvMsg), 0); // 接收信息

        // 获取客户端ip地址
        char clientIp[INET_ADDRSTRLEN] = "";
        inet_ntop(AF_INET, &(clientAddrs[num].sin_addr), clientIp,
INET_ADDRSTRLEN);
        if (recvByt > 0) // 接收成功
        {
            // 创建时间戳，记录当前通讯时间
            auto currentTime = chrono::system_clock::now();
            time_t timestamp = chrono::system_clock::to_time_t(currentTime);
            tm localTime;
            localtime_s(&localTime, &timestamp);
            char timeStr[50];
            strftime(timeStr, sizeof(timeStr), "%Y-%m-%d %H:%M:%S", &localTime);
// 格式化时间
```

```
            cout << "Client [" ANSI_COLOR_RED << Clients[num] << ANSI_RESET << "
" ANSI_COLOR_YELLOW << clientIp << ANSI_RESET "] 在 " ANSI_UNDERLINE << timeStr
<< ANSI_RESET " 发送: " ANSI_COLOR_YELLOW << endl << recvMsg << ANSI_RESET << endl
<< endl;
            sprintf_s(sendMsg, sizeof(sendMsg), "Id[" ANSI_COLOR_CYAN "%d"
ANSI_RESET "] 在 " ANSI_UNDERLINE "%s" ANSI_RESET " 发送:  \n" ANSI_COLOR_YELLOW
"%s" ANSI_RESET "\n ", Clients[num], timeStr, recvMsg); // 格式化发送信息

            for (int i = 0; i < MaxClient; i++) // 将消息同步到所有聊天窗口
            {
                if (connectCondition[i] == 1 && i != num)
                {
                    send(Clients[i], sendMsg, strlen(sendMsg), 0); // 发送信息
                }
            }
        }
        else // 接收失败
        {
            if (WSAGetLastError() == 10054) // 客户端主动关闭连接
            {
                // 创建时间戳，记录当前通讯时间
                auto currentTime = chrono::system_clock::now();
                time_t timestamp = chrono::system_clock::to_time_t(currentTime);
                tm localTime;
                localtime_s(&localTime, &timestamp);
                char timeStr[50];
                strftime(timeStr, sizeof(timeStr), "%Y-%m-%d %H:%M:%S",
&localTime); // 格式化时间

                cout << "Client [" ANSI_COLOR_RED << Clients[num] << ANSI_RESET "
" ANSI_COLOR_YELLOW << clientIp << ANSI_RESET "] 退出于 " ANSI_UNDERLINE <<
timeStr << ANSI_RESET << endl << endl;

                closesocket(Clients[num]);
                currentConnections--;
                connectCondition[num] = 0;
                cout << "当前客户端连接数量为: " ANSI_COLOR_CYAN <<
currentConnections << ANSI_RESET << endl << endl;
                return 0;
            }
            else
            {
                cout << "接收失败, Error:" << WSAGetLastError() << endl << endl;
                break;
            }
        }
    }
}
```

main函数

- 在 `main` 函数中，首先进行`WinSock2`库的初始化，检查初始化是否成功
- 自动获取本机ipv4地址，并将其储存进`ipAddr`中
- 创建服务器套接字 `Server`，并绑定服务器地址和端口
- 设置监听，以便接受客户端连接请求
- 进入循环，不断接受客户端的连接请求，如果连接数未达到最大连接数 `MaxClient`，则创建新的线程来处理客户端通信
- 在 `main` 函数中，也记录了客户端的连接时间和当前连接数，以及处理连接数达到最大值时的情况

```
int main()
{
    // 初始化WinSock库
    WSAData wsaData;
    if (WSAStartup(MAKEWORD(2, 1), &wsaData) == -1) {
        cout << "初始化 WinSock 出错,Error:" << WSAGetLastError;
    }
    else {
        cout << "初始化 WinSock 成功" << endl;
    }

    //获取本机ip地址
    char hostName[256];
    if (gethostname(hostName, sizeof(hostName)) == SOCKET_ERROR) {
        cout << "gethostname 失败" << endl;
        exit(EXIT_FAILURE);
    }
    struct addrinfo hints;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_INET;
    struct addrinfo* addrInfo = nullptr;
    if (getaddrinfo(hostName, nullptr, &hints, &addrInfo) != 0) {
        cout << "getaddrinfo 失败" << endl;
        exit(EXIT_FAILURE);
    }

    //存储本机ip
    char ipAddr[INET_ADDRSTRLEN];

    for (struct addrinfo* p = addrInfo; p != nullptr; p = p->ai_next) {
        struct sockaddr_in* ipv4 = reinterpret_cast<struct sockaddr_in*>(p-
>ai_addr);

        if (inet_ntop(AF_INET, &(ipv4->sin_addr), ipAddr, INET_ADDRSTRLEN) ==
nullptr) {
            cout << "inet_ntop 失败" << endl;
            exit(EXIT_FAILURE);
            continue;
        }
        else {
            cout << "本机IP地址 " ANSI_COLOR_MAGENTA << ipAddr << ANSI_RESET <<
endl;
```

```
        }
    }
    freeaddrinfo(addrInfo);


    Server = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    /*
    AF_INET：使用ipv4（AF是指定地址族的宏，Address Family），也可以使用PF_INET,在使用Socket API进行套接字编程时二者是等价的
    SOCK_STREAM：套接字类型，保证了数据的有序性，确保了数据的完整性和可靠性，以及提供了流式传输的特性，确保传输的数据按照发送顺序被接收
                    同时还是实验要求的一部分
    IPPROTO_TCP：使用TCP传输协议
    */

    if (Server == INVALID_SOCKET) // 错误处理
    {
        perror("创建 Socket 错误");
        exit(EXIT_FAILURE);
    }
    cout << "创建 Socket 成功" << endl;

    // 绑定服务器地址
    serverAddr.sin_family = AF_INET;   // 地址类型
    serverAddr.sin_port = htons(PORT); // 端口号

    if (inet_pton(AF_INET, ipAddr, &(serverAddr.sin_addr)) != 1)//将ServIp转换为二进制并且存储进servAddr.sin_addr
    {
        cout << "服务端地址绑定出错" << endl;
        exit(EXIT_FAILURE);
    }
    else {
        cout << "服务端地址 " ANSI_COLOR_MAGENTA << ipAddr << ANSI_RESET << " 绑定成功" << endl;
    }
    if (bind(Server, (LPSOCKADDR)&serverAddr, sizeof(serverAddr)) == SOCKET_ERROR) // 将服务器套接字与服务器地址和端口绑定
    {
        perror("套接字与端口绑定失败");
        exit(EXIT_FAILURE);
    }
    else
    {
        cout << "套接字与端口 " ANSI_COLOR_MAGENTA << PORT << ANSI_RESET " 绑定成功" << endl;
    }

    // 设置监听/等待队列
    if (listen(Server, MaxClient) != 0)
    {
        perror("设置监听失败");
        exit(EXIT_FAILURE);
    }
    else
    {
```

```
        cout << "设置监听成功" << endl;
    }

    cout << "服务端成功启动\n" << endl;

    // 循环接收客户端请求
    while (true)
    {
        if (currentConnections < MaxClient)
        {
            int num = isEmpty();
            int addrlen = sizeof(SOCKADDR);
            Clients[num] = accept(Server, (sockaddr*)&clientAddrs[num],
&addrlen); // 等待客户端请求

            // 获取客户端ip地址
            char clientIp[INET_ADDRSTRLEN] = "";
            inet_ntop(AF_INET, &(clientAddrs[num].sin_addr), clientIp,
INET_ADDRSTRLEN);

            if (Clients[num] == SOCKET_ERROR)
            {
                perror("客户端出错 \n");
                closesocket(Server);
                WSACleanup();
                exit(EXIT_FAILURE);
            }
            connectCondition[num] = 1;// 连接位置1表示占用
            currentConnections++; // 当前连接数加1

            // 创建时间戳，记录当前通讯时间
            auto currentTime = chrono::system_clock::now();
            time_t timestamp = chrono::system_clock::to_time_t(currentTime);
            tm localTime;
            localtime_s(&localTime, &timestamp);
            char timeStr[50];
            strftime(timeStr, sizeof(timeStr), "%Y-%m-%d %H:%M:%S", &localTime);
// 格式化时间

            cout << "Client [" ANSI_COLOR_RED << Clients[num] << ANSI_RESET " "
ANSI_COLOR_YELLOW << clientIp << ANSI_RESET "] 连接于 " ANSI_UNDERLINE << timeStr
<< ANSI_RESET << endl << endl;
            cout << "当前客户端连接数量为: " ANSI_COLOR_CYAN << currentConnections
<< ANSI_RESET << endl << endl;

            HANDLE Thread = CreateThread(NULL, 0,
(LPTHREAD_START_ROUTINE)ThreadFunction, (LPVOID)num, 0, NULL); // 创建线程

            if (Thread == NULL) // 线程创建失败
            {
                perror("线程创建失败\n");
                exit(EXIT_FAILURE);
            }
            else
            {
                CloseHandle(Thread);
```

```
            }
        }
        else
        {
            cout << "客户端数量已满" << endl << endl;
        }
    }

    closesocket(Server);
    WSACleanup();

    return 0;
}
```

（2）客户端

该部分实现了一个简单的客户端程序，用于连接到服务器并进行基本的消息通信。它使用多线程来同时接收和发送消息，允许用户在控制台上输入消息，并将消息发送到服务器。

基础部分与服务端端相同

线程函数

编写了一个接收信息的线程函数`recvThread`，用于接收从服务器发送过来的消息并显示在控制台上

- 使用`recv`函数来接收消息，然后将消息显示在控制台上
- 如果接收到的消息小于等于 0，表示连接已经断开，线程将退出

```
DWORD WINAPI recvThread() //接收消息线程
{
        while (true)
        {
                char msg[msgSize] = {};//接收消息缓冲区
                if (recv(Client, msg, sizeof(msg), 0) > 0)//参数：客户端套接字，要发送的缓冲区（信息），上一个参数的长度，标志
                {
                        cout << endl << msg << endl;
                }
                else if (recv(Client, msg, sizeof(msg), 0) < 0)
                {
                        cout << endl << "你已退出或是服务端断开" << endl;
                        break;
                }
        }
        Sleep(100);
        return 0;
}
```

main函数

- 在`main`函数中，首先进行`WinSock2`库的初始化，检查初始化是否成功
- 创建客户端套接字`Clients`，并连接到服务器
- 如果连接失败，程序会输出错误信息并退出

- 如果连接成功，程序会输出连接成功的信息
- `main` 函数进入一个循环，等待用户输入消息
  - 用户可以通过io设备输入消息，然后使用 `send` 函数将消息发送给服务器
  - 如果用户输入 "exit"，则退出循环，关闭套接字，清理WinSock2库并退出程序

```
int main()
{
	//初始化 WinSock 库
	WSADATA wsaData;
	if (WSAStartup(MAKEWORD(2, 1), &wsaData) == -1) {
		cout << "初始化 WinSock 出错,Error:" << WSAGetLastError;
	}
	else {
		cout << "初始化 WinSock 成功" << endl;
	}
	//创建一个客户端套接字
	Client = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
	/*
	AF_INET：使用ipv4（AF是指定地址族的宏，Address Family），也可以使用PF_INET,在使用Socket API进行套接字编程时二者是等价的
	SOCK_STREAM：套接字类型，保证了数据的有序性，确保了数据的完整性和可靠性，以及提供了流式传输的特性，确保传输的数据按照发送顺序被接收
                                        同时还是实验要求的一部分
	IPPROTO_TCP：使用TCP传输协议
	*/

	// 获取本机IP地址
	char hostName[256];
	if (gethostname(hostName, sizeof(hostName)) == SOCKET_ERROR) {
		cout << "gethostname 失败" << endl;
		exit(EXIT_FAILURE);
	}
	struct addrinfo hints;
	memset(&hints, 0, sizeof(hints));
	hints.ai_family = AF_INET;
	struct addrinfo* addrInfo = nullptr;
	if (getaddrinfo(hostName, nullptr, &hints, &addrInfo) != 0) {
		cout << "getaddrinfo 失败" << endl;
		exit(EXIT_FAILURE);
	}


	char localIp[INET_ADDRSTRLEN];
	char serverIp[INET_ADDRSTRLEN];

	for (struct addrinfo* p = addrInfo; p != nullptr; p = p->ai_next) {
		struct sockaddr_in* ipv4 = reinterpret_cast<struct sockaddr_in*>(p->ai_addr);

		if (inet_ntop(AF_INET, &(ipv4->sin_addr), localIp, INET_ADDRSTRLEN) == nullptr) {
			cout << "inet_ntop 失败" << endl;
			exit(EXIT_FAILURE);
		}
```

```
                else {
                        cout << "本机IP地址 " ANSI_COLOR_MAGENTA << localIp <<
ANSI_RESET << endl;
                }

                // 判断服务端是否在本地
                cout << "服务端是否在本地？(Y/N): ";
                char choice;
                cin >> choice;
                if (choice == 'Y' || choice == 'y') {
                        strcpy_s(serverIp, INET_ADDRSTRLEN, localIp);  // 将
ipAddr赋值给serverIp
                }
                else {
                        cout << "请输入服务端IP地址(输入N则选择默认IP）: ";
                        cin >> serverIp;
                        if (!strcmp(serverIp,"n")|| !strcmp(serverIp, "N")) {  //
判断用户是否选择默认ip
                                strcpy_s(serverIp, INET_ADDRSTRLEN, ServIp);
                        }
                }

        }
        freeaddrinfo(addrInfo);

        // 绑定服务器地址
        servAddr.sin_family = AF_INET;
        servAddr.sin_port = htons(PORT);

        if (inet_pton(AF_INET, serverIp, &(servAddr.sin_addr)) != 1) {
                cout << "服务端地址绑定出错" << endl;
                exit(EXIT_FAILURE);
        }
        else {
                cout << "服务端地址绑定成功";
                cout << ", 地址为 " ANSI_COLOR_MAGENTA << serverIp << ":" << PORT
<< ANSI_RESET << endl;
        }

        cout << "连接服务端中..." << endl;

        //向服务器发起请求
        if (connect(Client, (SOCKADDR*)&servAddr, sizeof(SOCKADDR)) ==
SOCKET_ERROR)
        {
                cout << "服务端连接失败，Error: " ANSI_COLOR_RED <<
WSAGetLastError() << ANSI_RESET << endl;
                if (WSAGetLastError() == 10061) {
                        cout << "服务端可能未开启\n";
                }
                exit(EXIT_FAILURE);
        }
        else
        {
                cout << "服务端连接成功\n" << endl;
        }
```

```
        //创建消息线程
        CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)recvThread, NULL, 0, 0);

        char msg[msgSize] = {};
        cout << "输入 '" ANSI_COLOR_RED << "exit" << ANSI_RESET "' 断开与服务端的连
接" << endl;


        //发送消息
        while (true)
        {
                cin.getline(msg, sizeof(msg));
                if (strcmp(msg, "exit") == 0) //输入exit断开
                {
                        break;
                }
                else if (strlen(msg) == 0)  // 当输入为空字符串时不发送消息
                {
                        continue;
                }
                send(Client, msg, sizeof(msg), 0);//向服务端发送消息
        }

        //关闭客户端套接字以及WinSock库
        closesocket(Client);
        WSACleanup();

        return 0;
}
```

(3) 可能遇到的问题

```
        // 获取本机IP地址
        char hostName[256];
        if (gethostname(hostName, sizeof(hostName)) == SOCKET_ERROR) {
                cout << "gethostname 失败" << endl;
                exit(EXIT_FAILURE);
        }
        struct addrinfo hints;
        memset(&hints, 0, sizeof(hints));
        hints.ai_family = AF_INET;
        struct addrinfo* addrInfo = nullptr;
        if (getaddrinfo(hostName, nullptr, &hints, &addrInfo) != 0) {
                cout << "getaddrinfo 失败" << endl;
                exit(EXIT_FAILURE);
        }


        char localIp[INET_ADDRSTRLEN];
        char serverIp[INET_ADDRSTRLEN];

        for (struct addrinfo* p = addrInfo; p != nullptr; p = p->ai_next) {
```

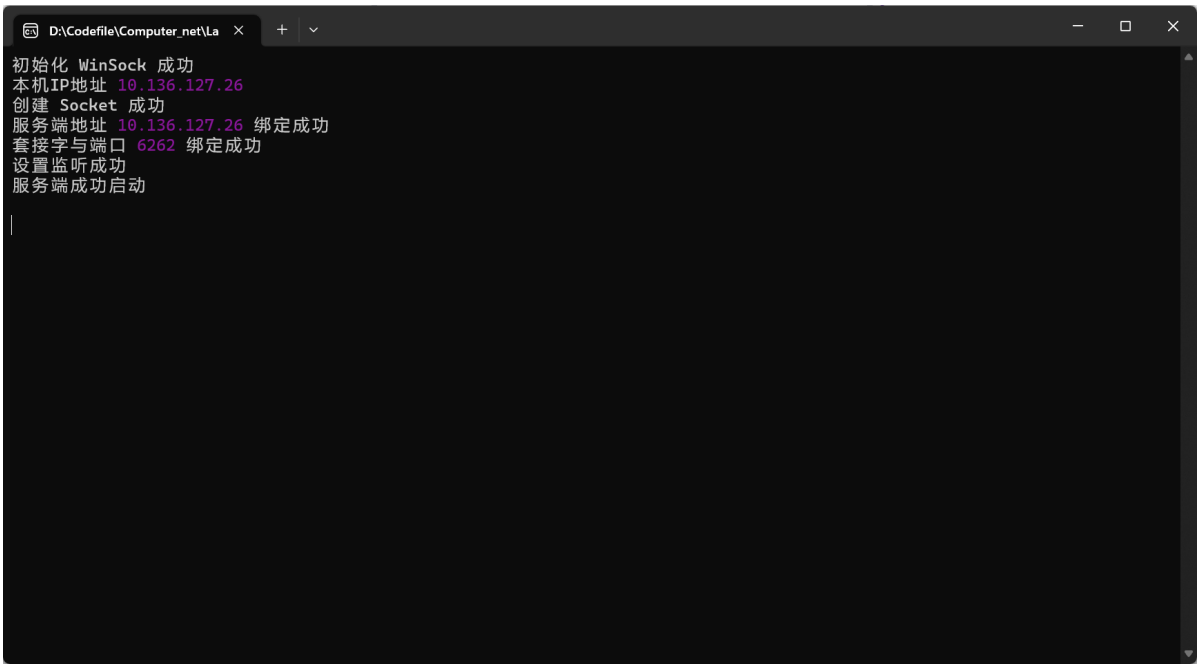
```
                struct sockaddr_in* ipv4 = reinterpret_cast<struct sockaddr_in*>
(p->ai_addr);

                if (inet_ntop(AF_INET, &(ipv4->sin_addr), localIp,
INET_ADDRSTRLEN) == nullptr) {
                        cout << "inet_ntop 失败" << endl;
                        exit(EXIT_FAILURE);
                }
                else {
                        cout << "本机IP地址 " ANSI_COLOR_MAGENTA << localIp <<
ANSI_RESET << endl;
                }

                // 判断服务端是否在本地
                cout << "服务端是否在本地？(Y/N): ";
                char choice;
                cin >> choice;
                if (choice == 'Y' || choice == 'y') {
                        strcpy_s(serverIp, INET_ADDRSTRLEN, localIp);  // 将
ipAddr赋值给serverIp
                }
                else {
                        cout << "请输入服务端IP地址(输入N则选择默认IP）: ";
                        cin >> serverIp;
                        if (!strcmp(serverIp,"n")|| !strcmp(serverIp, "N")) {  //
判断用户是否选择默认ip
                                strcpy_s(serverIp, INET_ADDRSTRLEN, ServIp);
                        }
                }

        }
        freeaddrinfo(addrInfo);
```

在程序中使用了通过addrinfo来查找sockaddr的方式来检索本机ip，但是校园网是由多个局域网组合而成，可能会出现一个设备被分发多个ipv4地址的情况，如果在其他网络环境下使用可以对获取到的本机ip进行筛选，不满足的则对该次循环continue。

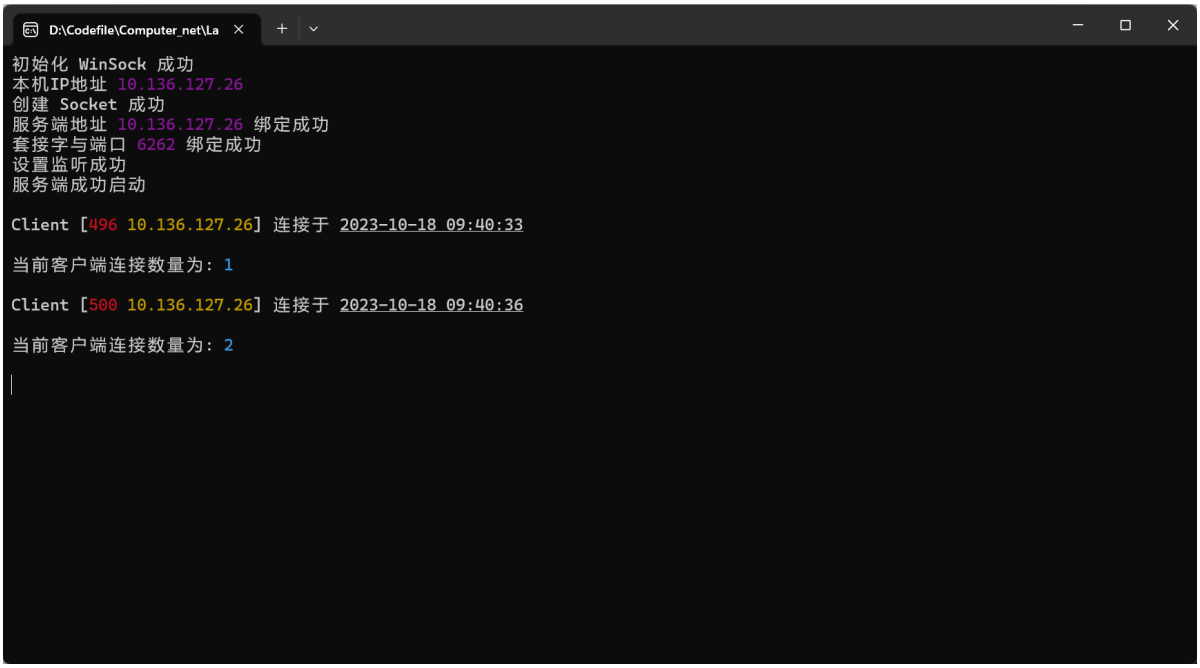
四、运行结果展示

服务端演示：

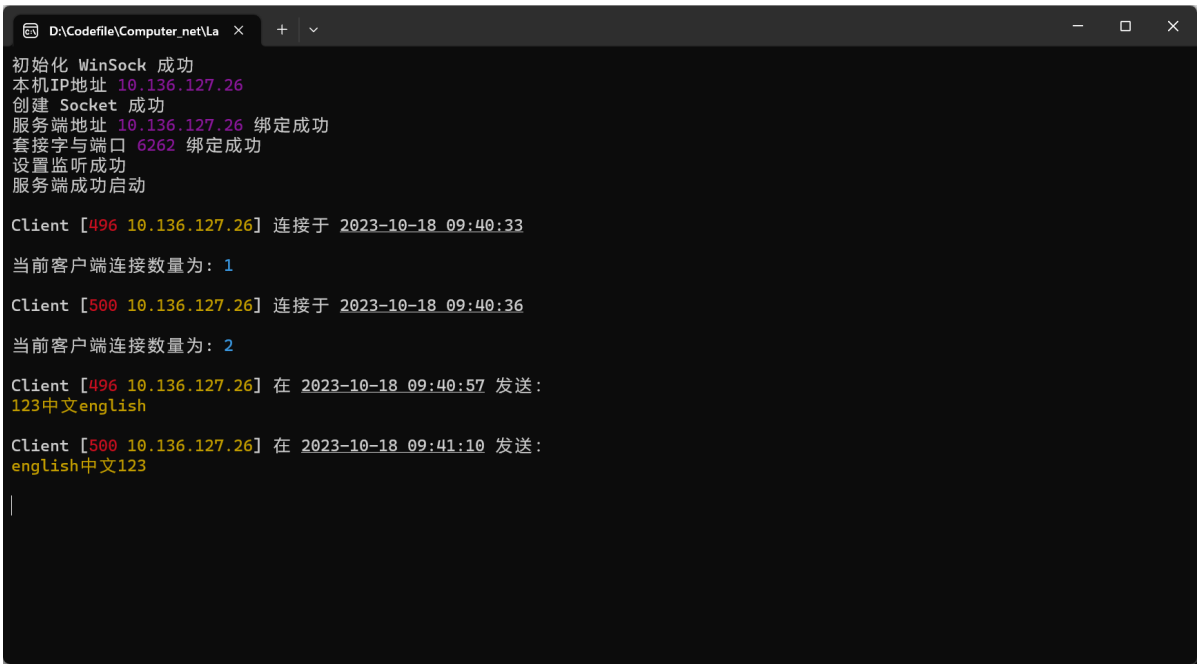
1. 初始化日志信息



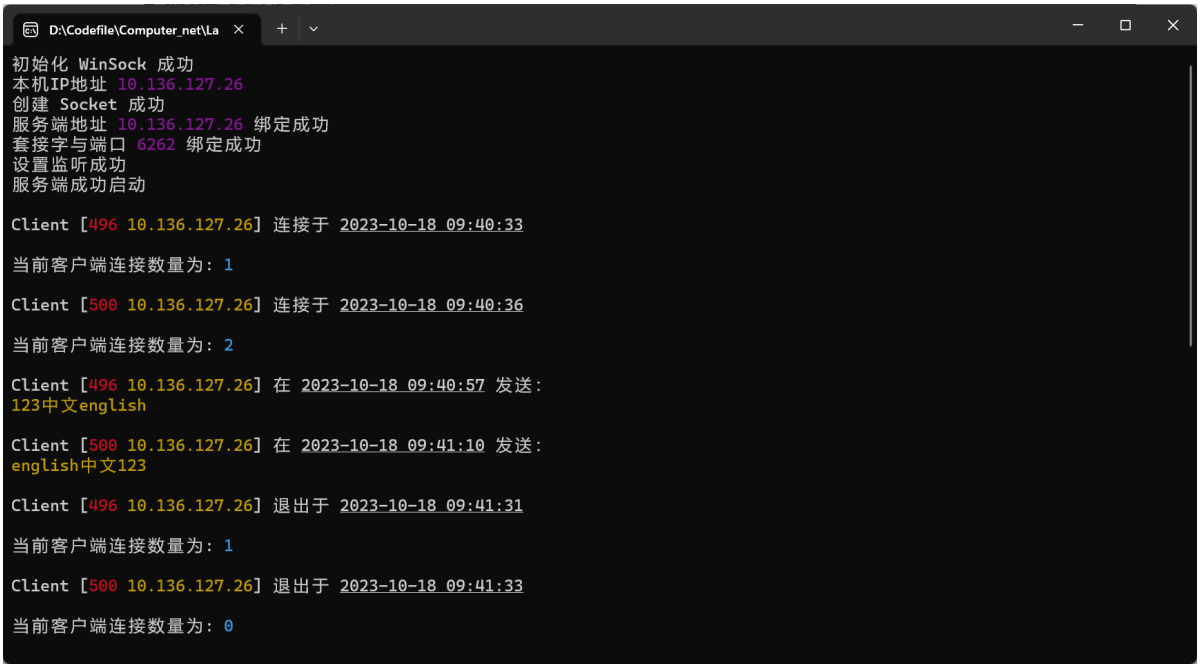
2. 客户端连接日志



3. 客户端消息发送日志

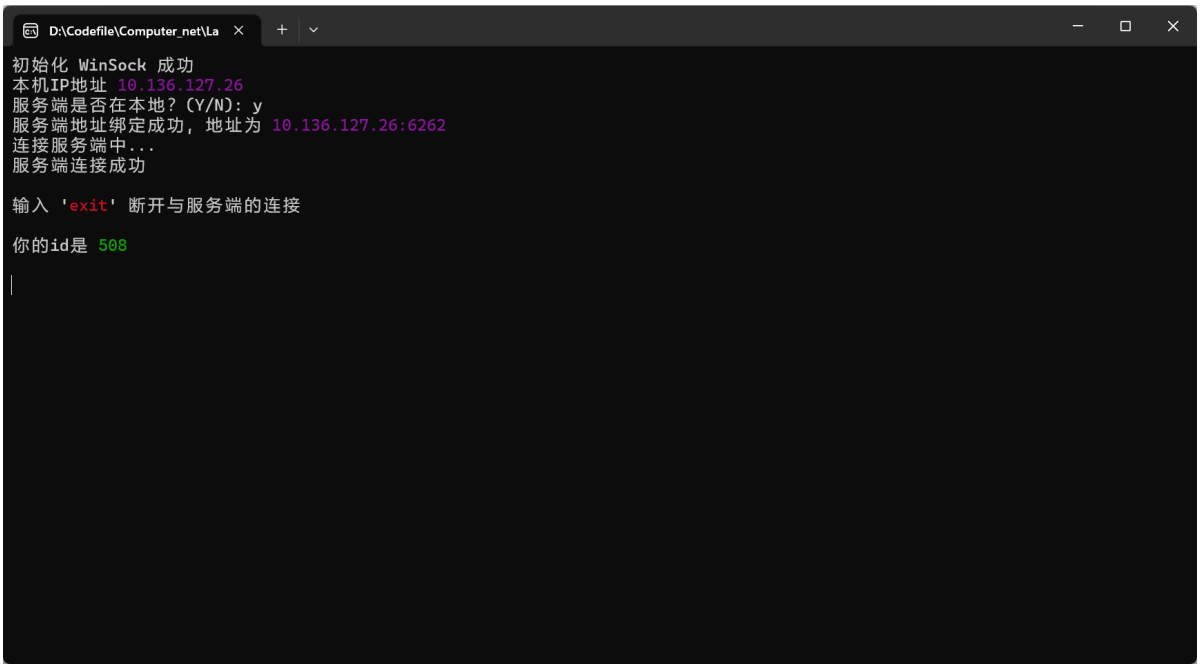


4. 客户端退出日志

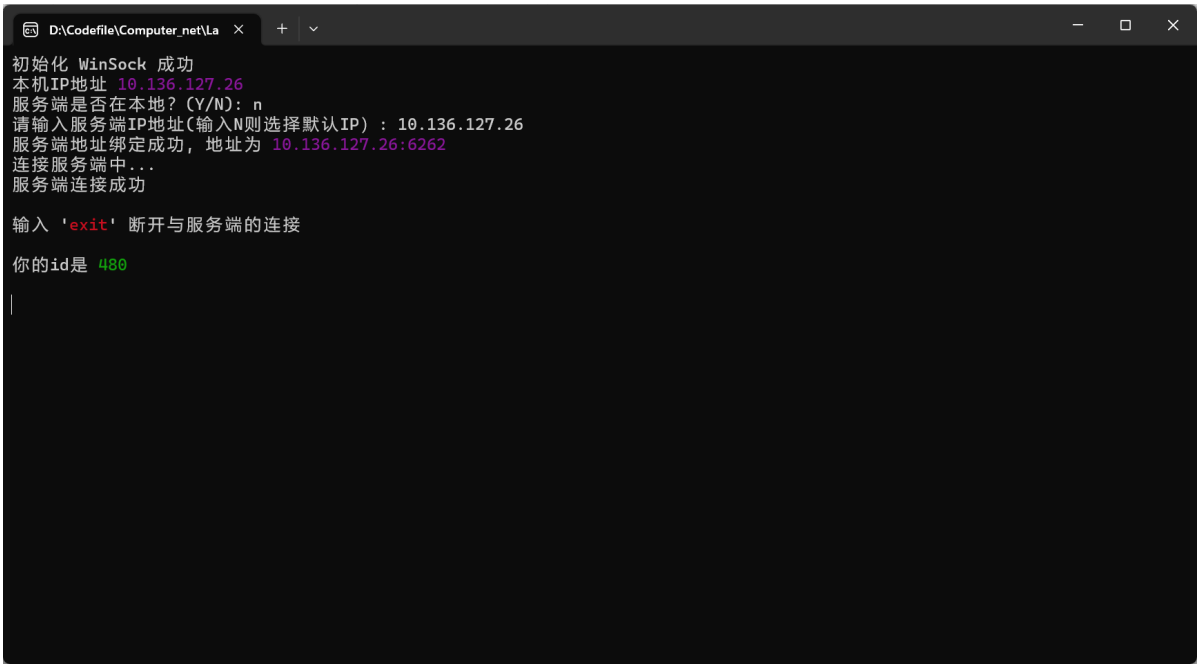


客户端演示：

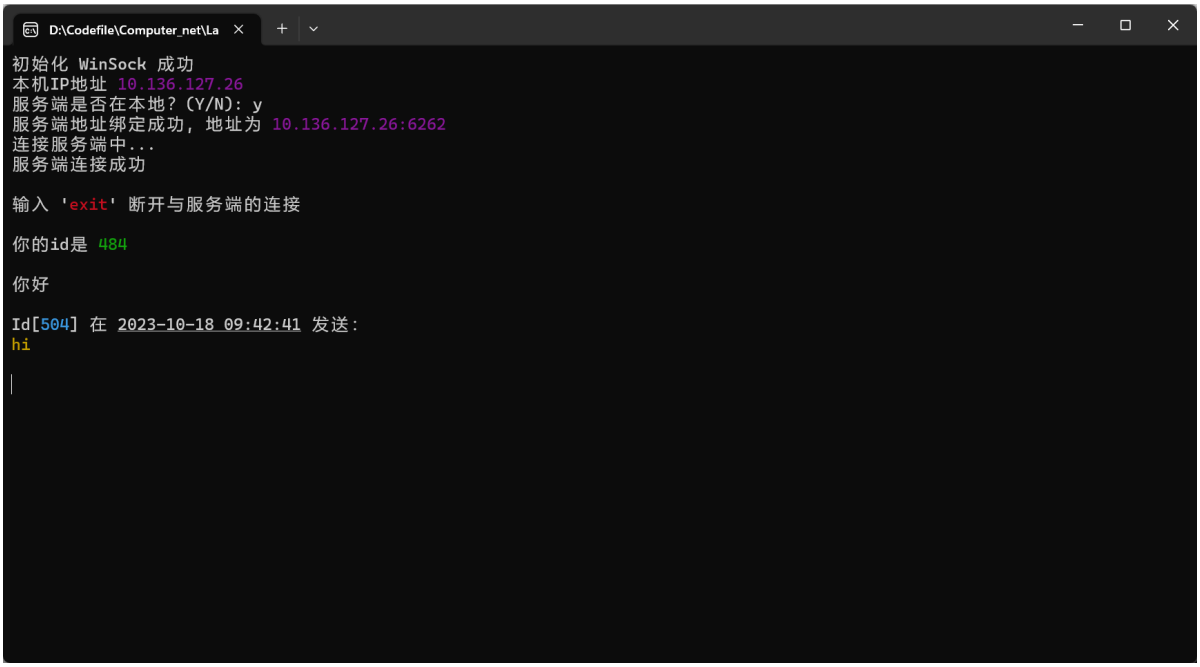
1. 客户端初始化日志（选择客户端在本地）



2. 客户端初始化日志（选择手动输入服务端ip）



3. 客户端发送消息以及接收消息


D:\Codefile\Computer_net\La
初始化 WinSock 成功
本机IP地址 10.136.127.26
服务端是否在本地？(Y/N)：y
服务端地址绑定成功，地址为 10.136.127.26:6262
连接服务端中...
服务端连接成功
输入 'exit' 断开与服务端的连接
你的id是 484
你好
Id[504] 在 2023-10-18 09:42:41 发送：
hi