计算机网络实验报告

实验一：使用Socket编程实现多人聊天室

姓名：谢雯菲 学号：2110803

实验要求

1. 给出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. 程序应该有基本的对话界面，但可以不是图形界面。程序应该有正常的退出方式；
5. 完成的程序应该支持多人聊天，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丢失，提交可执行文件、程序源码和实验报告。

实验原理

使用语言：`C++`
使用库：`Winsock2.h`

实验过程

聊天协议说明

传输层协议：`TCP`

`TCP（传输控制协议）`是一种面向连接的、可靠的`IETF`的`RFC 793`定义。`TCP`旨在适应支持多网络应用的分层协议层次结构。连接到不同但互连的计算机通信网络的主计算机中的成对进程之间依靠`TCP`提供可靠的通信服务。

- 面向连接：客户与服务器之间需要建立连接
- 可靠传输：可保证传递数据无差错
- 流量控制：发送数据不会超过接收端的容纳容量
- 拥塞控制：提供拥塞解决方案
- 不能提供：时延和带宽保证

套接字`Sockets`

`Socket`是一种操作系统提供的进程间通信机制。应用程序可以通过套接字接口，来使用网络套接字，以进行资料交换。

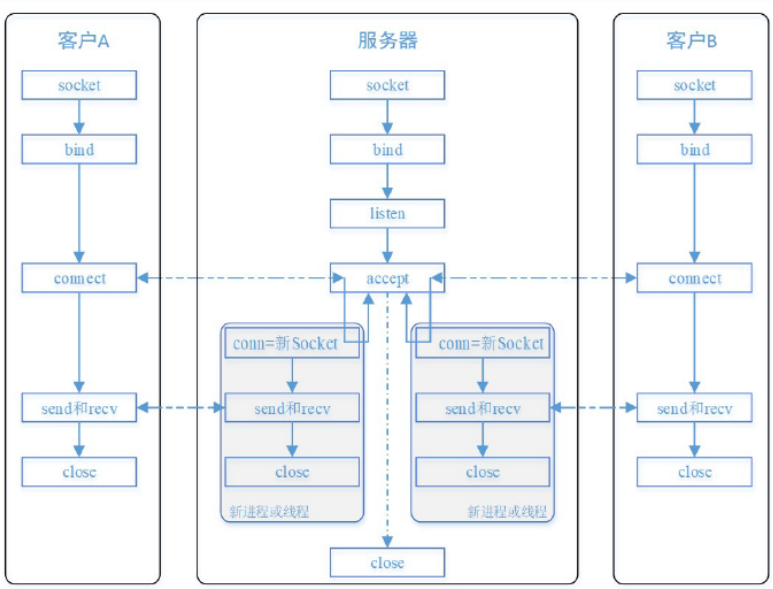
在套接字接口中，以`IP地址`及`通讯端口`组成`套接字地址(socket address)`。操作系统根据套接字地址，可以决定应该将资料送达特定的行程或线程。

在本次编程实验中，使用基于TCP协议的`流式套接字（stram socket）`，支持主机之间面向连接的、顺序的、可靠的、全双工字节流传输。

程序设计

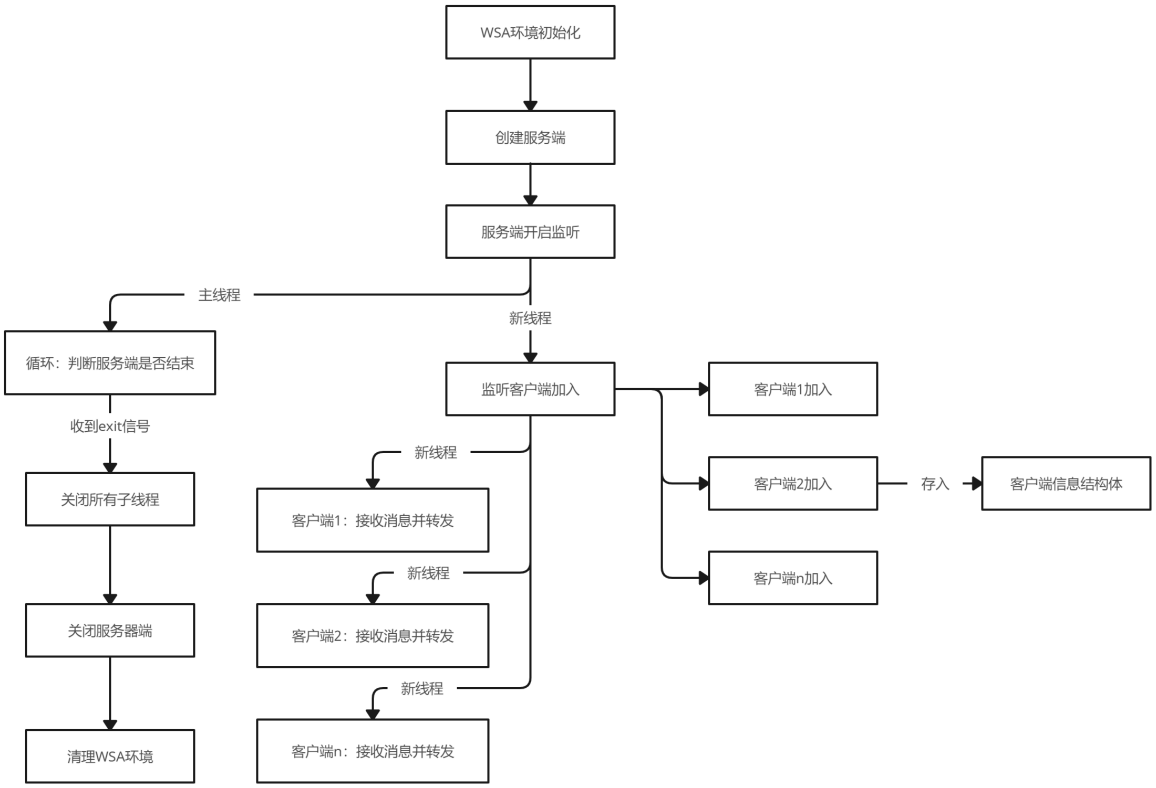
使用TCP服务的应用程序编写步骤

使用基于TCP服务的套接字编写数据传输协议的基本步骤如下图所示：



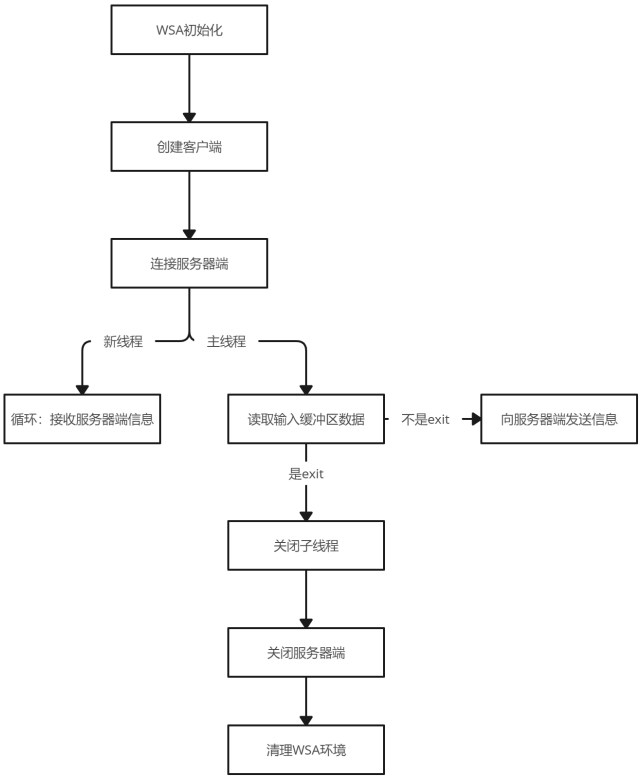
Server服务端设计思路

服务器端程序的设计思路如下图所示：



Client客户端设计思路

客户端程序的设计思路如下图所示：



代码详解

服务器端

初始化`Winsock`

调用`Winsock`函数(应用程序或DLL)的所有进程都必须在调用其他`Winsock`函数之前初始化`Windows`套接字DLL的使用。

创建名为`wsaData`的`WSADATA`对象，调用`WSAStartup`并将其值作为整数返回，并检查错误。`WSAStartup`的`MAKEWORD(2，2)`参数对系统上的`Winsock`版本2.2发出请求，并将传递的版本设置为调用方可以使用的最高版本的`Windows`套接字支持。如果成功，`WSAStartup`函数返回零。

初始化WSA环境部分的代码如下所示：

```
//初始化Winsock
//创建wsadata的对象
WSADATA wsaData;
//返回初始化结果
int wsaRe;
wsaRe = WSAStartup(MAKEWORD(2, 2), &wsaData);
if (wsaRe != 0) {
    cout << "! ! ! WSA环境初始化失败! ! ! " << endl;
    return 0;
}
```

创建服务器端套接字

`socket`函数创建并返回一个套接字，使用函数创建套接字过程如下：

```
//创建套接字
ser_sock = socket(
    AF_INET,            //使用IPv4地址
    SOCK_STREAM,        //使用流式套接字
    IPPROTO_TCP         //使用TCP协议
);
```

将套接字和IP、端口绑定

定义套接字的IP和端口

使用`sockaddr_in`结构体定义套接字的IP和端口，`sockaddr_in`结构体的结构如下所示：

```
struct sockaddr_in {
    short sin_family;           // 地址族，通常为 AF_INET
    unsigned short sin_port;    // 端口号
    struct in_addr sin_addr;    // IP地址
    char sin_zero[8];           // 用于填充，使结构体与 sockaddr 兼容
};
```

定义IP和端口部分的代码如下所示：

```
//定义套接字的IP和端口
memset(&ser_addr, 0, sizeof(sockaddr_in)); //清空addr内存部分的值
ser_addr.sin_family = AF_INET;
//使用htons将u_short将主句转换为TCP/IP网络字节顺序（大端）
ser_addr.sin_port = htons(8000);
//使用inet_addr函数将IPv4地址的字符串转换为s_addr结构
ser_addr.sin_addr.s_addr = inet_addr("127.0.0.1");
```

将套接字绑定到本地的IP和端口

使用`bind()`函数将套接字绑定到本地的IP和端口，`bind()`函数的参数如下：

```
int WSAAPI bind(
    [in] SOCKET         s,              //待绑定的套接字
    [in] const sockaddr* name,          //指向待分配的sockaddr结构的指针
    [in] int            namelen         //sockaddr内容的长度
);
```

如果未发生错误，绑定将返回零。否则，它将返回`SOCKET_ERROR`。

绑定部分的代码如下：

```
//将套接字绑定到本地IP和端口
if (bind(ser_sock, (SOCKADDR*)&ser_addr, saddr_len)) {
    //如果绑定发生错误
    cout << "套接字绑定发生错误："<< WSAGetLastError() << endl;
    return 0;
}
```

开启服务器监听

使用`listen()`函数开启服务端套接字的监听，`listen()`函数的输入参数如下所示：

```
int WSAAPI listen(
    [in] SOCKET s,          //未连接的套接字
    [in] int    backlog     //挂起连接队列的最大长度
);
```

如果未发生错误，将返回零。否则，将返回值`SOCKET_ERROR`。

开启监听代码如下所示：

```
//开启监听
if (listen(ser_sock, 20) != 0) {
    cout << "监听错误" << endl;
    return 0;
}
cout << ".....服务器端正在监听....." << endl;
```

创建新线程循环接收客户端的连接

创建一个新线程用来循环接收客户端的连接。使用`CreateThread()`函数创建线程，该函数参数要求如下：

```
HANDLE CreateThread(
    [in, optional]  LPSECURITY_ATTRIBUTES   lpThreadAttributes,
                    //决定该句柄是否可由子进程继承
    [in]            SIZE_T                  dwStackSize,
                    //堆栈初始大小，若为零，使用可执行文件的默认大小
    [in]            LPTHREAD_START_ROUTINE  lpStartAddress,
                    //指向线程的起始地址
    [in, optional]  __drv_aliasesMem LPVOID lpParameter,
                    //指向要传递给线程的变量的指针
    [in]            DWORD                   dwCreationFlags,
                    //创建线程的标志，为0时表示创建后立即运行
    [out, optional] LPDWORD                 lpThreadId
                    //指向接收线程标识符的指针，为NULL时不返回
);
```

`CreateThread()`函数会自动调用表示函数执行的回调函数，并会返回一个`HANDLE`句柄。

创建新线程的过程如下所示：

```
//接收客户端的连接的线程
HANDLE rthread = CreateThread(NULL, 0, ReceiveThread, (LPVOID)&cnum, 0,
NULL);
```

调用的回调函数如下所示：

```
//接收客户端线程
DWORD WINAPI ReceiveThread(LPVOID num) {
    //线程循环接收客户端的连接
    while (true) {
```

```
        client* c = &cli_inf[cnum];
        sockaddr_in addrClient;
        //返回与client通讯的socket
        c->cli_sock = accept(ser_sock, (SOCKADDR*)&addrClient, &saddr_len);
        c->cid = cnum;
        if (c->cli_sock != INVALID_SOCKET) {
            //如果返回的不是无效套接字
            //创建线程，接收该客户端的信息并转发
            HANDLE cThread = CreateThread(NULL, 0, ClientThread, (LPVOID)c->cid, 0, NULL);
            c->cthread = &cThread;
        }
        cnum++;
    }
    return 0;
}
```

在回调函数中，进行了以下操作：

1. 创建一个结构体存储客户端的信息：

```
//存储客户端信息的结构
int cnum = 0;           //客户端个数
struct client {
    int cid=-1;         //客户端编号
    SOCKET cli_sock;    //客户端套接字
    HANDLE* cthread;    //客户端线程
}cli_inf[20];
```

2. 使用`accept()`函数接收客户端的连接，`accept()`函数的参数如下所示：

```
SOCKET WSAAPI accept(
    [in]      SOCKET   s,       //服务端监听套接字
    [out]     sockaddr* addr,   //指向接收连接实体的地址的指针
    [in, out] int* addrlen      //指向接收addr结构长度的地址
);
```

3. 成功接收客户端的连接后，为每个客户端创建一个新的收发线程。在线程中，循环接收客户端的信息，并将信息转发给所有其他客户端。

每个客户端的收发线程

在为每个客户端创建的收发线程中，循环接收客户端的信息，并调用`bc()`函数将信息转发给所有其他客户端。

在循环中使用`recv()`函数不断接收客户端数据，`recv()`函数使用参数如下所示：

```
int WSAAPI recv(
    [in]  SOCKET s,     //连接的客户端套接字
    [out] char* buf,    //用于接收传入数据的缓冲区指针
    [in]  int    len,   //buf缓冲区长度
    [in]  int    flags  //标识符，在此设为0
);
```

编写一个`bc()`函数用于向客户端广播信息，在`bc()`函数中用`send()`函数向客户端发送信息，`send()`函数的使用参数如下所示：

```
int WSAAPI send(
    [in] SOCKET     s,      //连接的客户端套接字
    [in] const char* buf,   //指向要传输数据的缓冲区指针
    [in] int        len,    //buf缓冲区中数据的长度
    [in] int        flags   //指定调用方式，在此设为0
);
```

`bc()`函数代码如下：

```
//广播客户端函数
void bc(int cid, char* send_buf) {
    for (int i = 0; i < cnum; i++) {
        if (i == cid) {
            continue;
        }
        send(cli_inf[i].cli_sock, send_buf, 100, 0);
    }
    return;
}
```

接收客户端信息的线程的回调函数如下所示：

```
//接收客户端信息线程函数
DWORD WINAPI ClientThread(LPVOID cc) {
    //与客户端通讯，接收并发送信息
    int cid = (int)cc;
    cout << "--客户端" << cid << "加入聊天室--" << endl;
    //发送数据
    char send_buf[100] = { 0 };
    sprintf_s(send_buf,100, "客户端 %d 成功加入聊天室！\n", cid);
    bc(-1, send_buf);   //广播
    //循环接收客户端数据
    while (true) {
        char recv_buf[100] = { 0 };
        int ret = 100;  //表示recv()函数返回值
        ret = recv(cli_inf[cid].cli_sock, recv_buf, 100, 0);
        if (ret == SOCKET_ERROR) {
            cout << "！！！客户端" << cid << "接收错误："<<WSAGetLastError()
<<"！！！" << endl;
            break;
        }
        if (!strcmp(recv_buf,"exit")) {
            cout << "### 客户端" << cid << "已退出聊天室 ###" << endl;
            memset(send_buf, '0', sizeof(send_buf));    //清空发送缓冲区
            sprintf_s(send_buf,100, "### 客户端 %d 已退出聊天室！### \n", cid);
            bc(cid, send_buf);
            break;
        }
        memset(send_buf, '0', sizeof(send_buf));
        sprintf_s(send_buf,100, "客户端 %d : %s \n", cid, recv_buf);
        cout << send_buf;
```

```
        bc(cid, send_buf);
    }
    return 0;
}
```

关闭服务器端

服务器端读取指令，如果输入了“exit”指令，则表示退出服务器端，广播各客户端告知服务器端停止服务。

退出服务器端时需要关闭创建的所有线程，并且关闭服务器端套接字，清理winsock2的环境。

关闭服务器端部分代码如下：

```
//死循环判断主线程是否退出
while (true) {
    string command = "0";
    getline(cin,command);
    if (command == "exit") {
        //广播各客户端表示服务器端已停止服务
        char main_send[100] = { 0 };
        sprintf_s(main_send, 100, "exit");
        bc(-1, main_send);
        break;
    }
}
//退出服务器端指令
//关闭所有创建的线程
for (int i = 0; i < cnum; i++) {
    CloseHandle(*cli_inf[i].cthread);
}
CloseHandle(rthread);
//关闭监听套接字
closesocket(ser_sock);
//清理winsock2的环境
WSACleanup();
cout << "----服务器端已停止服务----" << endl;
```

客户端

连接服务器端

使用 `connect()` 函数将客户端连接到正在监听的服务器端套接字，如果未发生错误， 则连接返回零，函数参数结构如下：

```
int WSAAPI connect(
    [in] SOCKET         s,          //标识待连接客户端的套接字
    [in] const sockaddr* name,      //连接sockaddr结构的指针
    [in] int            namelen     //sockaddr指针的长度
);
```

代码如下：

```
//连接服务器端
if (connect(cli_sock, (SOCKADDR*)&cli_addr, saddr_len) == SOCKET_ERROR) {
    cout << "！！！连接服务器端失败：" << WSAGetLastError() << endl;
    return 0;
}
```

创建新线程接收服务器端信息

创建一个新线程循环接收服务器端的信息，如果接收到的信息是`exit`，则表示服务器端已停止服务：

```
//接收消息线程
DWORD WINAPI ReceiveThread(LPVOID cs) {
    //循环接收消息
    while (true) {
        char recv_buf[100] = { 0 };
        int ret = 100;
        ret= recv(cli_sock, recv_buf, 100, 0);
        if (ret == SOCKET_ERROR) {
            if (WSAGetLastError() != 10053) {
                cout << "！！！服务器端错误：" << WSAGetLastError() << "！！！"
<< endl;
            }
            return 0;
        }
        if (!strcmp(recv_buf,"exit")) {
            cout << "----服务器端已停止服务----" << endl;
            return 0;
        }
        if (ret == 0) {
            cout << "----服务器端已停止服务----" << endl;
            return 0;
        }
        cout << recv_buf << endl;
    }
    return 0;
}
```

主线程向服务器端传递信息

在主线程中循环读取输入缓冲区，向他人发送消息。如果输入的是`exit`，则退出聊天室；如果返回时显示服务器端已停止服务，则打印在窗口上：

```
//主线程循环发送消息
    while (true) {
        char send_buf[100] = { 0 };
        cin.getline(send_buf, 100);
        int ret = 100;
        ret = send(cli_sock, send_buf, 100, 0);
        if (!strcmp(send_buf, "exit")) {
            cout << "----客户端关闭----" << endl;
            break;
        }
        if (ret == -1) {
            if (WSAGetLastError() == 10054) {
```

```
13                  cout << "----服务器端已停止服务-----" << endl;
14              }
15          }
16      }
```

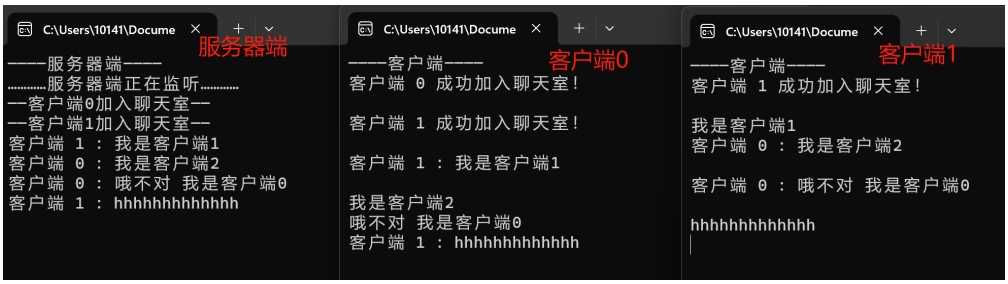
程序运行

开启服务

先运行服务器端程序，再运行客户端程序。打开多个客户端程序，以达到群聊目的。

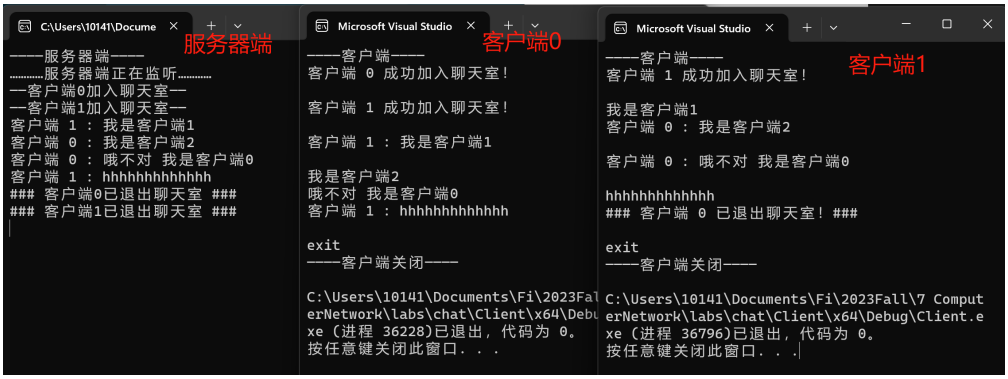
发送消息

客户端在群聊中发送消息。



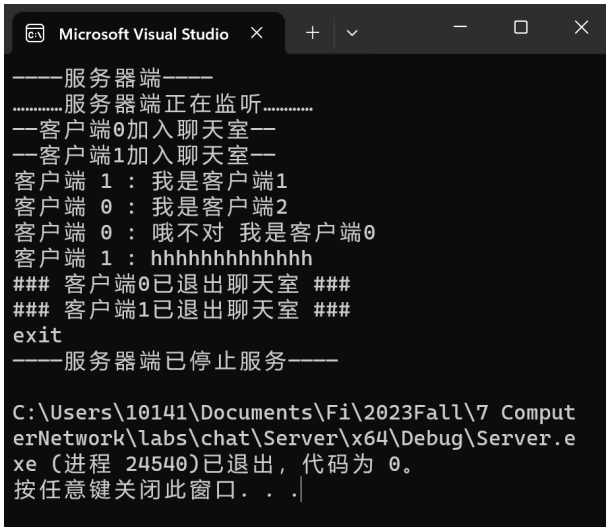
客户端退出群聊

客户端输入`exit`字段退出群聊。



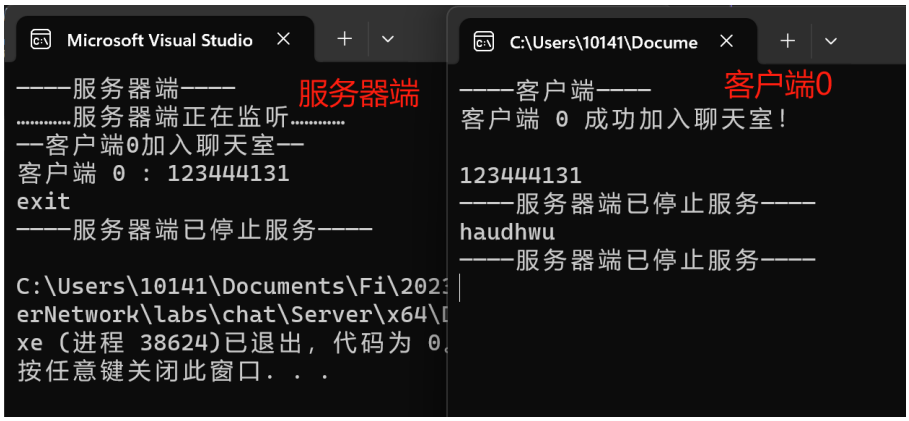
服务器端停止服务

服务器端输入`exit`字段退出群聊。



服务器端先退出服务

若服务器端先退出服务，客户端窗口会出现提示，且继续发送消息会显示“服务器端已停止服务”：



程序执行正确。

实验总结

通过本次实验，我对传输层和应用层协议有了进一步的理解。我初步掌握了利用 C++ 语言，使用基本的 Socket 流式套接字和多线程的方式编写聊天程序的方法。在实验过程中暂时没有观察到数据丢失的现象，后续可以进一步完善实验以实现群聊的更多功能。