



南開大學  
Nankai University

南 开 大 学

计 算 机 学 院

计算机网络实验报告

---

多人聊天协议设计

---

学号：1911437

姓名：刘浩通

年级：2019 级

专业：计算机科学与技术

2021 年 10 月 22 日

# 目录

一、 实验要求	1
二、 协议设计和程序设计	1
(一) 客户端设计 . . . . .	2
(二) 服务器设计 . . . . .	2
三、 程序实现	3
(一) 客户端程序实现 . . . . .	3
(二) 服务器端程序实现 . . . . .	7
四、 程序运行说明	16
五、 总结	18

## 一、 实验要求

1. 设计一个多人聊天协议，要求实现选择不同的用户进行分组聊天。
2. 设计多人聊天程序。
3. 在 Windows 系统下，利用 C/C++ 中的流式 Socket 对设计的程序进行实现。程序界面可以采用命令行方式，但需要给出使用方法。

本次实验我采用的是以控制台的形式来显示程序。在客户端使用普通的 TCP 连接来实现，在服务器端使用完成端口、重叠模型来实现。

## 二、 协议设计和程序设计

### 客户端设计

功能:

1. 数据发送
2. 数据接收

技术:

1. socket
2. 输入流和输出流
3. 多线程，客户端功能模块有两个线程

聊天:

1. 和其他客户端聊天
2. 和服务器聊天

### 服务器

功能:

1. 数据转发
2. 与用户建立连接
3. 和用户聊天

技术:

1. ServerSocket 服务器 socket，绑定完全端口
2. 每一个用户对应的 Sokcet 对象，使用重叠技术
3. 多线程处理任务，消息转发和服务器说话
4. 哈希表存储，用户 id 和用户 socket

数据转发:

1. 私聊前缀判断
2. 群聊所有人发送

## (一) 客户端设计

首先,介绍一下客户端的协议,客户端是使用 TCP 与服务器端连接,使用 connect 进行连接请求,使用 recv 和 send 函数进行消息的发送和接收。

为了防止发送消息后等待接收消息时,线程卡死,所以使用多线程进行设计,从主线程上另开辟一条进程进行消息的接收,主线程则进行消息的发送操作。相较于开辟两种线程,避免了主线程空转的浪费问题。

再然后,为了让服务器端便于消息的转发操作,在发送消息时使用“目的客户端:消息”的格式来进行书写。这样当服务器接收到客户端的请求后,可以提取出目的客户端的 id 号,来进行消息的转发。另外,当客户端向服务器端发送消息时,格式则为“:消息”,以便让服务器识别。

然后是服务器端的设计,这里使用了完全端口来进行设计,因为完成端口会充分利用 Windows 内核来进行 I/O 的调度,充分利用内核对象的调度,只使用少量的几个线程来处理客户端的所有通信,消除了无谓的线程上下文切换,最大限度的提高了网络通信的性能。

## (二) 服务器设计

服务器端首先会建立 ServerSocket,以便和客户端建立连接。在调用 bind 函数绑定端口时,会创建并绑定完全端口,完了后调用 listen 进行监听,等待客户端的连接。

这里服务器类 Server 中会有一个 Map 来存储 <id,socket>,也就是客户端 socket 及其对应的 id 值,这样便于维护服务器端与客户端的连接,并且易于实现消息的转发,毕竟要根据发来的消息来找到对应的目的客户端,这样的话会很方便。

随后创建与一定数目的线程进行处理事件。也就是连接客户端、转发消息的操作。

随后主线程的操作是实现服务器发送给客户端消息的功能。

然后具体说下完成端口在里面的操作。

完成端口的做法是这样的:事先开好几个线程,首先是为了避免了线程的上下文切换,因为线程想要执行的时候,总有 CPU 资源可用,然后让这几个线程等着,等到有用户请求来到的时候,就把这些请求都加入到一个公共消息队列中去,然后这几个开好的线程就排队逐一去从消息队列中取出消息并加以处理,这种方式实现了异步通信和负载均衡的问题,因为它提供了一种机制来使用几个线程“公平的”处理来自于多个客户端的输入/输出,并且线程如果没有操作的时候也会被系统挂起,不会占用 CPU 周期。

用完成端口遵循如下几个步骤:

(1) 调用 CreateIoCompletionPort() 函数创建一个完成端口,而且在一般情况下,我们需要且只需要建立这一个完成端口,把它的句柄保存好。

(2) 根据系统中有多少个处理器,就建立多少个工作者(为了醒目起见,下面直接说 Worker)线程,这几个线程是专门用来和客户端进行通信的,目前暂时没什么工作;

(3) 下面就是接收连入的 Socket 连接了,这里有两种实现方式:一是和别的编程模型一样,还需要启动一个独立的线程,专门用来 accept 客户端的连接请求;二是用性能更高更好的异步 AcceptEx() 请求。在这里使用的是异步 AcceptEx 函数。

(4) 每当有客户端连入的时候,我们就还是得调用 CreateIoCompletionPort() 函数,这里却不是新建完成端口了,而是把新连入的 Socket(也就是前面所谓的设备句柄),与目前的完成端口绑定在一起。

这里就是服务器绑定完全端口的操作,和服务器接收客户端请求连接绑定完全端口的操作。

然后再看建立的子线程的操作:

使用 GetQueuedCompletionStatus() 监控完成端口,子线程们在队列中监视完成端口的队列中是否有完成的网络操作。

一旦完成端口上出现了已完成的 I/O 请求，那么等待的线程会被立刻唤醒，然后继续执行后续的代码。比如说是与客户端建立连接，来自客户端的消息需要服务器转发处理，或者是客户端断线需要把对应客户端从服务器端的表中删去。

## 三、 程序实现

具体实现操作看注释

### (一) 客户端程序实现

client\_class.h

```
1 #pragma once
2 #include<iostream>
3 #include<WinSock2.h>
4 #include<thread>
5 #include<cstdlib>
6 #include<sstream>
7 #include<string>
8 #include<WS2tcpip.h>
9 using namespace std;
10 #pragma comment(lib, "ws2_32.lib")
11 class Client {
12 private:
13     bool is_Quit = false; //客户端是否退出
14     SOCKET Server = INVALID_SOCKET; //用于和服务端建立连接
15     bool start(); //初始化工作，建立连接
16     void Receive(); //接收消息
17     void Send(); //发送消息
18     void clean(); //断线后清理资源
19 public:
20     Client() = default; //构造函数
21     ~Client(); //析构函数
22     void run(); //运行接口，由main函数调用
23 };
```

client\_class.cpp

```
1 #include"client_class.h"
2
3 bool Client::start() { //初始化
4     cout << "客户端" << endl;
5     WSADATA wsadata;
6     if (0 != WSAStartup(MAKEWORD(2, 2), &wsadata)) { //打开网络库
7         cout << "初始化失败" << endl;
8         return false;
9     }
10    else{
11        cout << "初始化成功" << endl;
```

```
12     }
13     //校验版本号
14     if (HIBYTE(wsadata.wVersion) != 2 || LOBYTE(wsadata.wVersion) != 2){
15         cout << "版本号错误: " << WSAGetLastError() << endl;
16         return false;
17     }
18     //创建嵌套字
19     Server = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
20     if (INVALID_SOCKET == Server) {
21         cout << "嵌套字创建失败" << endl;
22         return false;
23     }
24     else
25     cout << "嵌套字创建成功" << endl;
26     // 初始化IP端口号和协议族信息
27     SOCKADDR_IN ServerAddr;
28     ServerAddr.sin_family = AF_INET;
29     ServerAddr.sin_port = htons(10086);
30     inet_pton(AF_INET, "127.0.0.1", & ServerAddr.sin_addr);
31     //与服务器进行连接
32     if (SOCKET_ERROR == connect(Server, (SOCKADDR*)&ServerAddr, sizeof(
33         SOCKADDR))) {
34         cout << "连接服务器端失败:" << WSAGetLastError() << endl;
35         return false;
36     }
37     else
38     cout << "连接服务器成功" << endl;
39     return true;
40 }
41 void Client::clean() { //客户端关闭后, 清理资源
42     if (Server)
43         closesocket(Server);
44     WSACleanup();
45 }
46 Client::~Client() {
47     clean();
48 }
49 //接收消息
50 void Client::Receive() {
51     char RecvBuf[MAXBYTE] = { 0 }; //接收消息的数组
52     while (!is_Quit){ //若是未断开连接状态
53         if (SOCKET_ERROR == recv(Server, RecvBuf, sizeof(RecvBuf), 0)
54             ) {
55             //如果消息接收失败
56             is_Quit = true;
57             cout << "客户端断联" << endl;
58             return;
59         }
```

```

58         }
59         else{
60             /*
61             * 这里是处理收到的消息
62             * 因为发送格式是 客户端id: 消息
63             * 所以收到消息时可根据": "前的id, 知道消息源
64             * 若是0, 则是服务器端来的消息
65             */
66             int id = 0;
67             string msg(RecvBuf);
68             auto position = msg.find(": ");
69             istringstream temp(msg.substr(0, position));
70             temp >> id;
71             if (0 == id) {
72                 cout << "服务器: " << RecvBuf + position + 1
73                     << endl;
74             }
75             else{
76                 cout << id << "号客户端: " << RecvBuf +
77                     position + 1 << endl;
78             }
79         }
80     }
81 //用于发送消息
82 void Client::Send() {
83     char SendBuf[MAXBYTE] = { 0 };//发送消息的数组
84     string msg;
85     while (!is_Quit) {
86         msg.clear();
87         /*
88         * 使用getline是因为消息间可以有空格
89         * 中间用空格分割, 防止cin遇到空格截断消息
90         */
91         getline(cin, msg);
92         if (msg == "quit") {
93             is_Quit = true;
94             return;
95         }
96         if (msg.empty())//若输入为空, 则重新输入
97             continue;
98         else if (msg == "quit"){//执行退出操作
99             is_Quit = true;
100             return;
101         }
102         else{//发送的消息中必须有": "符号, 否则为无效消息
103             auto position = msg.find(": ");
104             if (position != string::npos) {

```

```

104         strcpy_s(SendBuf, msg.c_str());
105         if (SOCKET_ERROR == send(Server, SendBuf,
106             sizeof(SendBuf), 0)) {
107             cout << "发送失败了,重发试试" << endl
108                 ;
109         }
110         else
111         {
112             cout << "发送成功" << endl;
113         }
114     }
115 }
116 void Client::run() { //运行函数
117     if (!start()) //首先是启动
118     return;
119     thread r(&Client::Receive, this); //创建一个线程用于接收消息
120     r.detach();
121     Send(); //主函数用于发送消息
122 }

```

server\_class.h

```

1  #pragma once
2  #include<iostream>
3  #include<string>
4  #include<WinSock2.h>
5  #include<MSWSock.h>
6  #include<mswsock.h>
7  #include<thread>
8  #include<sstream>
9  #include<WS2tcpip.h>
10 #include<list>
11 #include<vector>
12 #include<unordered_map>
13 using namespace std;
14 #pragma comment(lib, "ws2_32.lib")
15 #pragma comment(lib, "mswsock.lib")
16 struct SocketOverlapped{ //保存客户端嵌套字, 客户端id, 及重叠io
17     SOCKET sock;
18     int id = 0;
19     OVERLAPPED overlap;
20 };
21 class Server {
22 private:
23     bool is_Quit = false; //服务器是否退出
24     HANDLE completionPort = INVALID_HANDLE_VALUE; //完成端口
25     list<SocketOverlapped> sockOver; //存放所有的SocketOverlapped信息

```



```

26 unordered_map<int, decltype(sockOver.begin())> sockItems; //主要是方
    便根据id找socket, 不用遍历
27 char RecvBuf[MAXBYTE]; //接收数组
28 char SendBuf[MAXBYTE]; //发送数组
29 bool start(); //启动函数
30 void clean(); //资源清理函数
31 int Accept(); //接收来自客户端的连接
32 int Receive(int); //接收客户端消息
33 int Send(int); //将客户端消息转发出去
34 void Send_Msg(); //服务器也有向客户端说话的功能
35 friend DWORD WINAPI ThreadProcess(LPVOID); //回调函数
36 public:
37     Server();
38     Server(const Server&) = delete;
39     Server(Server&&) = default;
40     ~Server();
41     static Server& getInstance(); //可以获得实例
42     void run(); //启动函数
43 };

```

## (二) 服务器端程序实现

server\_class.cpp

```

1 #include "server_class.h"
2
3 Server::Server():RecvBuf{0},SendBuf{0}{}
4 Server::~Server() {
5     clean();
6 }
7 void Server::clean() { //资源清理函数
8     for (auto& tmpfile : sockOver) { //关闭连接, 关闭overlapped IO
9         closesocket(tmpfile.sock);
10        WSACloseEvent(tmpfile.overlap.hEvent);
11    }
12    //把完全端口关闭了
13    if (INVALID_HANDLE_VALUE != completionPort)
14        CloseHandle(completionPort);
15    WSACleanup();
16 }
17 bool Server::start() { //初始化服务器
18     cout << "服务器" << endl;
19     WSADATA wsadata; //打开网络库
20     if (0 != WSAStartup(MAKEWORD(2, 2), &wsadata)) {
21         cout << "网络初始化失败" << endl;
22         return false;
23     }
24     else

```

```
25     cout << "网络初始化成功" << endl;
26     // 校验版本号
27     if (HIBYTE(wsadata.wVersion) != 2 || LOBYTE(wsadata.wVersion) != 2) {
28         cout << "版本错误" << endl;
29         WSACleanup();
30         return false;
31     }
32     // 初始化嵌套字
33     SOCKET ServerSocket = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP,
34                                     nullptr, 0, WSA_FLAG_OVERLAPPED);
35     if (INVALID_SOCKET == ServerSocket) {
36         cout << "嵌套字初始化失败" << endl;
37         WSACleanup();
38         return false;
39     }
40     else
41         cout << "嵌套字初始化成功" << endl;
42     // 初始化IP端口号和协议族信息
43     SOCKADDR_IN ServerAddr;
44     ServerAddr.sin_family = AF_INET;
45     ServerAddr.sin_port = htons(10086);
46     inet_pton(AF_INET, "127.0.0.1", &ServerAddr.sin_addr);
47     // 绑定端口
48     if (SOCKET_ERROR == bind(ServerSocket, (SOCKADDR*)&ServerAddr, sizeof
49                             (SOCKADDR))) {
50         cout << "绑定失败" << endl;
51         closesocket(ServerSocket);
52         WSACleanup();
53         return false;
54     }
55     else
56         cout << "绑定成功" << endl;
57     // 调用函数来创建完成端口
58     completionPort = CreateIoCompletionPort(INVALID_HANDLE_VALUE, nullptr
59                                             , 0, 0);
60     if (0 == completionPort) {
61         cout << "完成端口创建失败" << endl;
62         closesocket(ServerSocket);
63         WSACleanup();
64         return false;
65     }
66     else
67         cout << "完成端口创建成功" << endl;
68     // 绑定完成端口
69     if (completionPort != CreateIoCompletionPort((HANDLE)ServerSocket,
70                                                 completionPort, 0, 0)) {
71         cout << "绑定完成端口失败了" << endl;
72         CloseHandle(completionPort);
```

```

69         closesocket(ServerSocket);
70         WSACleanup();
71         return false;
72     }
73     else
74     cout << "绑定完全端口成功" << endl;
75     //监听
76     if (SOCKET_ERROR == listen(ServerSocket, SOMAXCONN)) {
77         cout << "监听失败了" << endl;
78         closesocket(ServerSocket);
79         CloseHandle(completionPort);
80         WSACleanup();
81         return false;
82     }
83     else
84     cout << "监听成功" << endl;
85     /*把服务器socket作为sockOver列表的第一个
86     * 并且赋予id=0
87     * 初始化overlap
88     */
89     sockOver.emplace_back(SocketOverlapped{});
90     sockOver.begin()->sock = ServerSocket;
91     sockOver.begin()->id = 0;
92     sockOver.begin()->overlap.hEvent = WSACreateEvent();
93
94     sockItems[ServerSocket] = sockOver.begin();
95
96     cout << "服务器初始化完成" << endl;
97     return 1;
98 }
99 int Server::Accept() { //接收客户端连接请求
100     //使用的是WSASocket函数, 最后一个参数为WSA_FLAG_OVERLAPPED
101     //因为使用的是完全端口, 异步操作
102     SOCKET Client = WSASocket(AF_INET, SOCK_STREAM, IPPROTO_TCP, nullptr,
103                               0, WSA_FLAG_OVERLAPPED);
104     if (INVALID_SOCKET == Client) {
105         cout << "客户端嵌套字创建失败" << endl;
106         return WSAGetLastError();
107     }
108     //紧接着放入sockOver, 先来后到, 服务器是第一个
109     sockOver.emplace_back(SocketOverlapped{});
110     sockOver.back().sock = Client;
111     sockOver.back().id = int(Client);
112     sockOver.back().overlap.hEvent = WSACreateEvent();
113
114     sockItems[Client] = —sockOver.end();
115     DWORD count;
116     //AcceptEx函数, 异步执行, 使用效果要比Accept好, 尤其是完全端口

```

```

116     bool res = AcceptEx(sockOver.begin()->sock, Client, RecvBuf, 0,
117         sizeof(SOCKADDR_IN) + 16, sizeof(SOCKADDR_IN) + 16,
118         &count, &sockItems.at(Client)->overlap);
119
120     int error = WSAGetLastError();
121     if (0 == error || ERROR_IO_PENDING == error)
122         return 0;
123     else { // 若建立连接失败, 则将sockOver和sockItems中存放的对应数据抹去
124         closesocket(Client);
125         WSACloseEvent(sockItems.at(Client)->overlap.hEvent);
126         sockOver.erase(sockItems.at(Client));
127         sockItems.erase(Client);
128         return error;
129     }
130 }
131 int Server::Receive(int sock) { // 接收函数
132     /*
133     * 操作就是把接收到的信息放到RecvBuf中, 也就是保存信息的数组中
134     */
135     WSABUF wsabuf;
136     wsabuf.buf = RecvBuf;
137     wsabuf.len = sizeof(RecvBuf);
138     DWORD count = 0;
139     DWORD flag = 0;
140     bool res = WSARecv(sockItems.at(sock)->sock, &wsabuf, 1, &count, &
141         flag, &sockItems.at(sock)->overlap, nullptr);
142     int error = WSAGetLastError();
143     if (0 == error || WSA_IO_PENDING == error) {
144         return 0;
145     }
146     else
147         return error;
148 }
149 int Server::Send(int sock) { // 发送函数
150     WSABUF wsabuf;
151     wsabuf.buf = SendBuf;
152     wsabuf.len = sizeof(SendBuf);
153     DWORD count = 0;
154     DWORD flag = 0;
155     bool res = WSASend(sockItems.at(sock)->sock, &wsabuf, 1, &count, flag,
156         &sockItems.at(sock)->overlap, nullptr);
157     int error = WSAGetLastError();
158     if (0 == error || WSA_IO_PENDING == error) {
159         return 0;
160     }
161     else
162         return error;
163 }

```

```

161 void Server::Send_Msg() { //服务器的发送消息函数
162     //可以实现消息的单发和群发
163     string message;
164     getline(cin, message);
165     if (message.empty())
166         return;
167     auto position = message.find(":");
168     if (position != string::npos) {
169         //这里是获取要发送消息的id, 可以有多个id
170         //若为":消息"格式, 则是向所有建立连接的客户端发送
171         vector<int> IDs;
172         istringstream I(message.substr(0, position));
173         int tmp;
174         while (I >> tmp) {
175             IDs.push_back(tmp);
176         }
177         size_t i = 1;
178         SendBuf[0] = ':';
179         for (size_t j = position + 1; j < message.size(); ++i, ++j) {
180             SendBuf[i] = message.at(j);
181         }
182         SendBuf[i] = 0;
183         if (IDs.empty()) { //向所有客户端发送
184             for (const auto& tmp : sockOver) {
185                 Send(tmp.sock);
186             }
187         }
188         else {
189             for (auto tmp : IDs) {
190                 Send(tmp);
191             }
192         }
193         return;
194     }
195     if (message == "quit") {
196         is_Quit = true;
197         Sleep(1000);
198         return;
199     }
200 }
201
202 DWORD WINAPI ThreadProcess(LPVOID lptr) {
203     Server* server = static_cast<Server*>(lptr);
204     HANDLE Port = (HANDLE)server->completionPort;
205     DWORD numberofbytes = 0;
206     LPOVERLAPPED lpOverlapped;
207     ULONG_PTR sock = 0;
208     while (!server->is_Quit) { //服务器是否退出

```

```

209         bool res = GetQueuedCompletionStatus(Port, &numberofbytes, &
210             sock, &lpOverlapped, INFINITE);
211         //线程们排队等待消息
212         int thisSocket = static_cast<int>(sock); //保存来时间的客户端
213         if (!res) { //出错情况
214             int error = GetLastError();
215             if (64 == error) { //第一种是有客户端断开连接
216                 //这时要进行资源清理, 将断开连接的客户端消息
217                 //删去
218                 cout << "客户端 【" << server->sockItems.at(
219                     sock)->id << "】 下线了" << endl;
220                 closesocket(server->sockItems.at(thisSocket)
221                     ->sock);
222                 WSACloseEvent(server->sockItems.at(thisSocket)
223                     ->overlap.hEvent);
224                 server->sockOver.erase(server->sockItems.at(
225                     thisSocket));
226                 server->sockItems.erase(thisSocket);
227                 continue;
228             }
229             if (WSA_WAIT_TIMEOUT == error) {
230                 //时间超时, 等待了很久也没有事件发生
231                 continue;
232             }
233             cout << "线程获取消息失败" << endl;
234             continue;
235         }
236         else { //这里就是有事件发生
237             if (0 == sock) { // 是否是服务器socket
238                 //因为是当事件发生后GetQueuedCompletionStatus
239                 //才会响应
240                 //而run函数会先调用一个Accept函数才会建立线程
241                 //事件
242                 //将客户端绑定到完全端口上
243                 auto Client = (--server->sockOver.end())->
244                     sock;
245                 HANDLE tmp_port = CreateIoCompletionPort((
246                     HANDLE)Client, server->completionPort,
247                     Client, 0);
248                 if (tmp_port != server->completionPort) {
249                     cout << "绑定完成端口失败" << endl;
250                     closesocket(Client);
251                     server->sockOver.erase(server->
252                         sockItems.at(Client));
253                     server->sockItems.erase(Client);
254                     continue;
255                 }
256                 //绑定成功后, 则进入消息发送接收阶段

```

```

245         server->sockOver.back().id = int(Client);
246         sprintf_s(server->SendBuf, ":%d】 欢迎登陆",
247             server->sockItems.at(Client)->id);
248         server->Send(server->sockItems.at(Client)->
249             sock);
250         server->Receive(server->sockItems.at(Client)
251             ->sock);
252         server->Accept(); // 这段是继续等待连接
253         cout << "客户端 【" << server->sockItems.at(
254             Client)->id << "】 来辣" << endl;
255     }
256     else{
257         if (numberofbytes == 0) { // 正常退出
258             cout << "客户端 【" << server->
259                 sockItems.at(sock)->id << "】 下线
260                 " << endl;
261             // 资源清理，删去对应客户端信息
262             closesocket(server->sockItems.at(
263                 thisSocket)->sock);
264             WSACloseEvent(server->sockItems.at(
265                 thisSocket)->overlap.hEvent);
266             server->sockOver.erase(server->
267                 sockItems.at(thisSocket));
268             server->sockItems.at(thisSocket);
269         }
270         else{
271             if (server->RecvBuf[0]) {
272                 string str(server->RecvBuf);
273                 cout << str << endl;
274                 auto position = str.find(':');
275                 ;
276                 istringstream ss(str.substr
277                     (0, position));
278                 int tmp = 0;
279                 // 要接收此消息的id 目的id(
280                     socket), 是_tmp的值
281                 ss >> tmp;
282                 if (tmp) {
283                     sprintf_s(server->
284                         SendBuf, "%d:%s",
285                         server->
286                         sockItems.at(
287                             thisSocket)->id,
288                         server->RecvBuf +
289                             position + 1);
290                     cout << server->
291                         SendBuf << endl;

```

```
274         if (server->sockItems
275             .count(tmp)) {
276             server->Send(
277                 tmp);
278             server->
279                 RecvBuf
280                 [0] = 0;
281             cout << "消息
282                 转发：客
283                 户端 【"
284                 << server
285                 ->
286                 sockItems
287                 .at(
288                     thisSocket
289                 )->id <<
290                 "】到客户
291                 端 【" <<
292                 tmp << "
293                 】" <<
294                 endl;
295         }
296         else{
297             cout << "客户
298                 端 【" <<
299                 tmp << "
300                 】不存在"
301                 << endl;
302             sprintf_s(
303                 server->
304                 SendBuf,
305                 "：客户端
306                 【%d】不
307                 存在",
308                 tmp);
309             server->Send(
310                 server->
311                 sockItems
312                 .at(
313                     thisSocket
314                 )->id);
315         }
316     }
317 }
318 else{//如果是对服务器说话
319     cout << "客户端 【" <<
320         server->
321         sockItems.at(
```



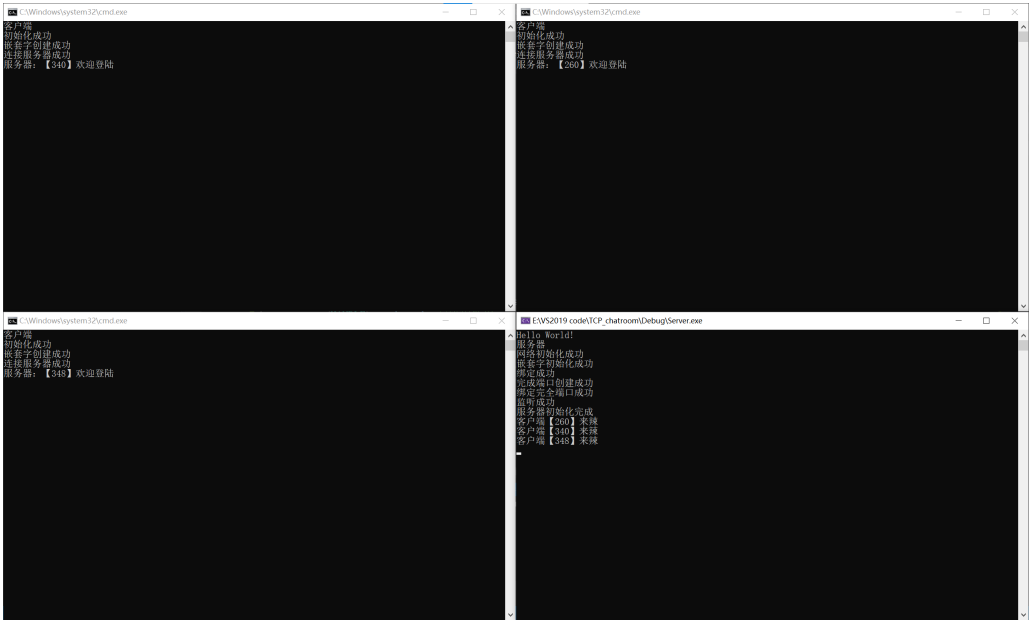
```

288                                     thisSocket)->sock
289                                     << "】对服务器说
290                                     " << server->
                                     RecvBuf +
                                     position + 1 <<
                                     endl;
291                                     }
292                                     server->RecvBuf[0] = 0;
293                                     server->Receive(server->
                                     sockItems.at(thisSocket)
                                     ->sock);
294                                     }
295                                     }
296                                     }
297                                     return 0;
298     }
299 }
300 Server& Server::getInstance() { // 返回实例
301     static Server server;
302     return server;
303 }
304
305 void Server::run() { // 运行函数
306     if (!start()) // 初始化
307         return;
308     if (Accept() != 0) { // 先建立一个连接
309         cout << "接受错误" << endl;
310         return;
311     }
312     // 开辟与电脑核心数相等数目的线程
313     SYSTEM_INFO si;
314     GetSystemInfo(&si);
315     int CPUCroeNumber = si.dwNumberOfProcessors;
316     vector<HANDLE> numthread; // 线程数组
317     for (int i = 0; i < CPUCroeNumber; i++) {
318         numthread.push_back(CreateThread(0, 0, ThreadProcess, this,
319                                         0, nullptr));
320     }
321     while (!is_Quit) { // 主函数也要会说话
322         Send_Msg();
323     }
324     for (int i = 0; i < CPUCroeNumber; i++)
325         CloseHandle(numthread.at(i));
326 }

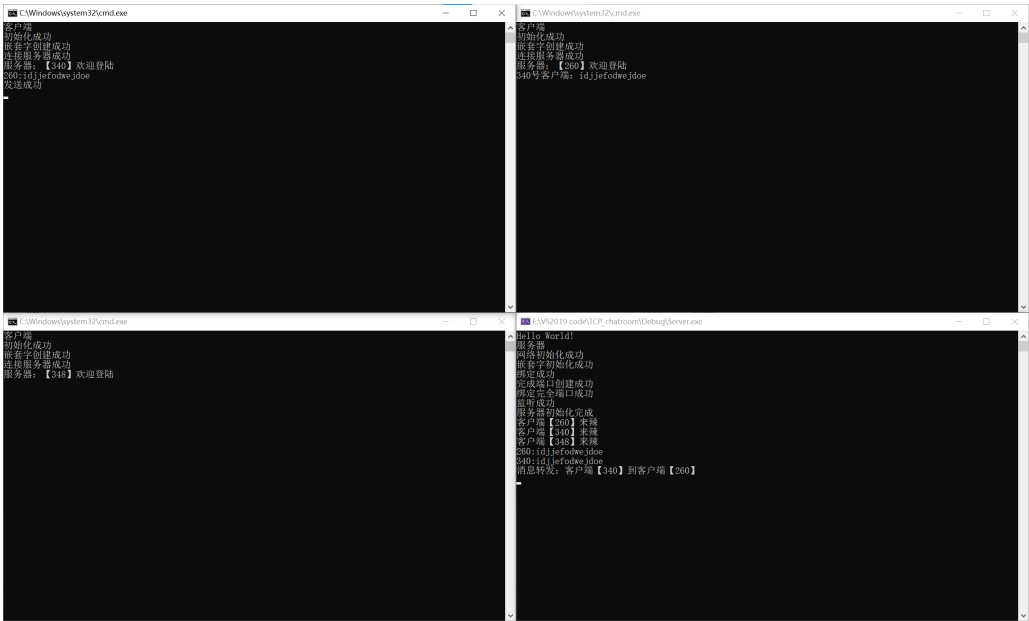
```

# 四、 程序运行说明

先建立服务器端和 3 个客户端，如下图所示，已经成功连接了。

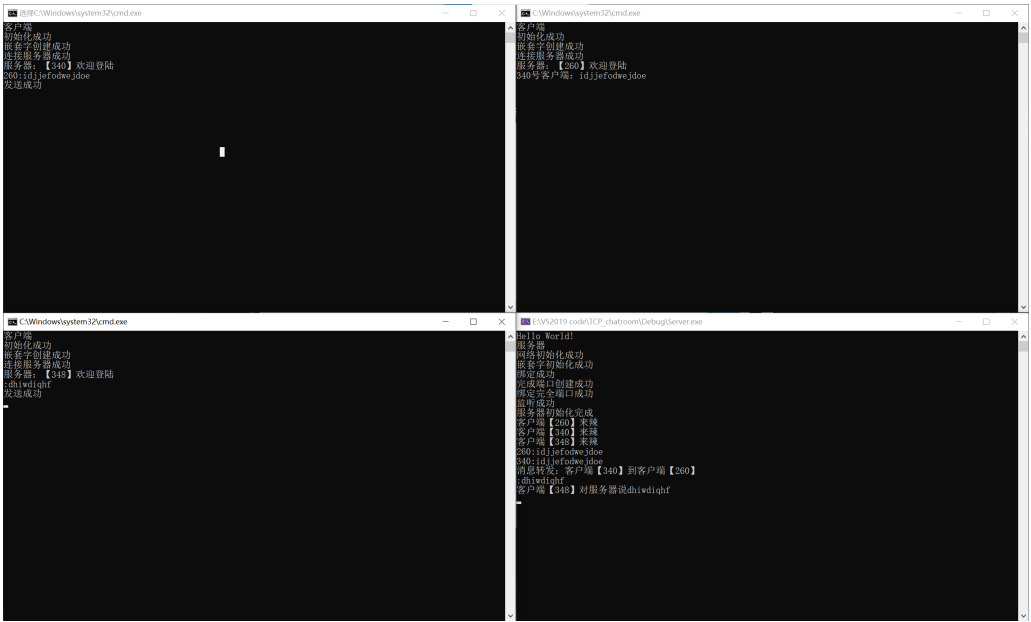


然后测试客户端之间的聊天功能，只实现了私聊：

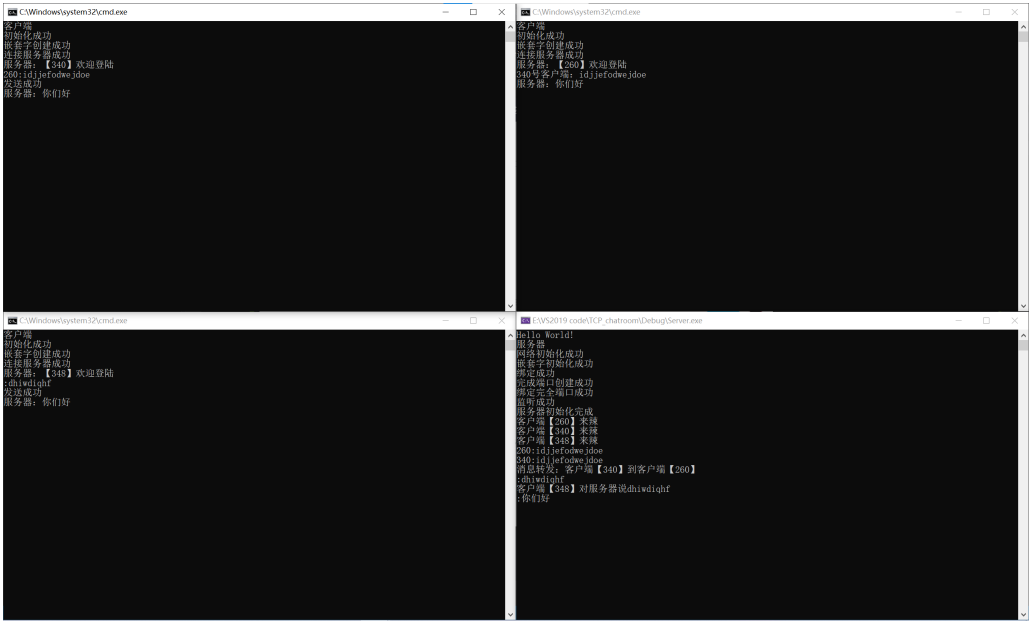


看到 340 向 260 成功发送了消息，并且 260 成功收到消息。

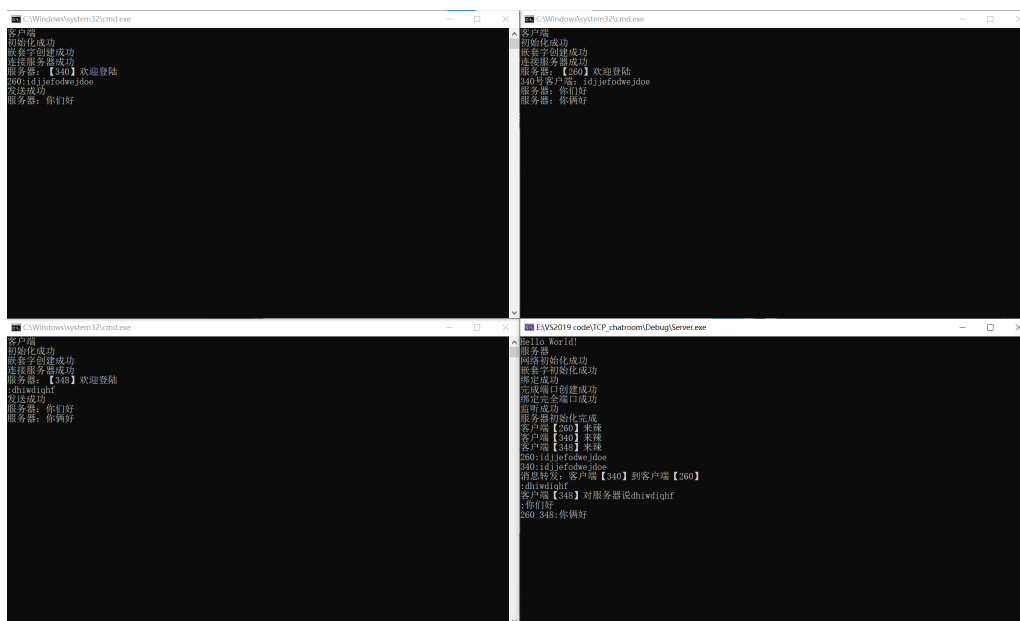
然后是客户端向服务器聊天：



348 向服务器发送消息，服务器成功收到消息。  
然后是服务器的群发功能，先是全局群发：



3 个已经连接的客户端都收到消息了。  
再测试服务器的局部说话：



```
客户端
初始化成功
嵌套字创建成功
连接服务器成功
服务器: 【260】 欢迎登陆
260: id: jeFodwejdoo
发送成功
服务器: 你们好

服务器
初始化成功
嵌套字创建成功
连接服务器成功
服务器: 【260】 欢迎登陆
340号客户端: id: jeFodwejdoo
服务器: 你们好
服务器: 你们好

Hello World!
服务器
初始化成功
嵌套字初始化成功
完成端口创建成功
绑定完成端口成功
服务器初始化完成
客户端【260】连接
客户端【340】未连接
客户端【348】未连接
260: id: jeFodwejdoo
340: id: jeFodwejdoo
消息转发: 客户端【260】到客户端【260】
:dhiwdighf
客户端【348】对服务器说dhiwdighf
你们好
260 348: 你们好
```

可以看到局部消息发送功能也实现了。

## 五、 总结

本次实现的是 TCP 协议的多人聊天协议。

服务器实现的功能是消息的群发和私发，与客户端建立连接以及客户端之间的消息转发功能。

客户端的功能就相比服务器较单一，实现与服务器的连接，消息的私发，以及消息的接收功能。

然后本次实验的难点就是实现服务器端的功能，因为服务器端会与多个客户端连接，如果每有一个客户端建立连接就开辟一个线程，会导致最终的资源浪费，毕竟每个线程不会一直保持活跃状态。然后就是用了完全端口，可以以一种异步的方式进行事件处理，一共有有限个线程，成队列处理事件，比如客户端的连接与断开，消息的转发，这样可以避免过量的资源浪费，并且还可以避免资源冲突。

还有一个实现难点，就是根据要发送对方的 id，来锁定对应的 socket 嵌套字，这里就是用了 map，实现 id(int) 和嵌套字 (socket) 的映射，根据 id 高效的找到对应的目的 socket。