实验三——高级聊天程序

SA20225085 朱志儒

实验目的

熟悉 Socket 编程

实验环境

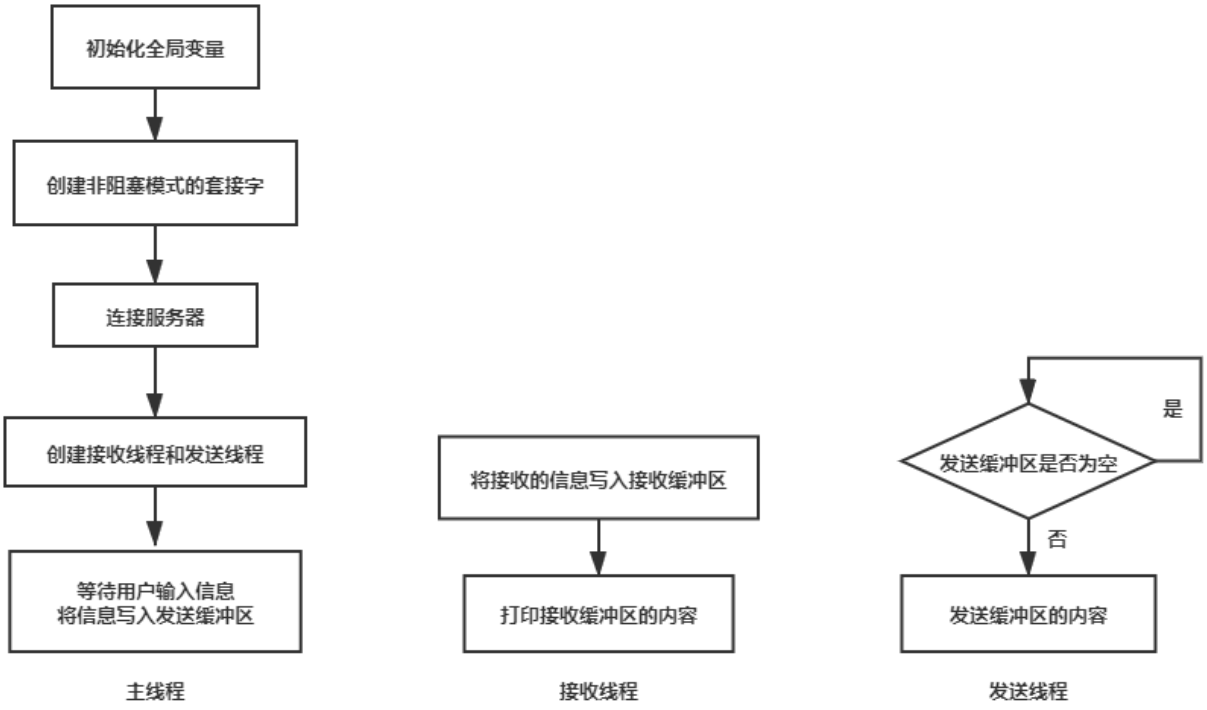
操作系统：Windows 10

编程环境：C++

实验内容

客户端

流程图：



主要变量：

```
SOCKET socketClient;
char sendBuffer[MAX_BUFF];
bool isConnect, isSend = false;
CRITICAL_SECTION criticalSection;
```

sockClient 表示客户端新建的 socket 连接。

sendBuffer 表示发送缓冲区，每当用户输入时，先将数据缓存在 sendBuffer，发送线程将读取 sendBuffer，然后将数据发送给服务器。

isConnect 表示客户端是否与服务器建立连接。

isSend 表示 sendBuffer 中是否有数据需要发送。

criticalSenction 是临界区标志位，每当用户有输入时，进入临界区，将 isSend 置为 true，将输入缓冲区的内容复制到 sendBuffer 中；每当 isSend 为 true，表示有数据需要发送时，发送线程进入临界区，将 sendBuffer 中的数据发送给服务器。

主要函数：

接收线程：接收服务器发送来的数据，并打印已接收的数据。

```
// 接收数据线程
DWORD __stdcall receiveDataThread(void* param) {
 int res;
 char receiveBuffer[MAX_BUFF];
 memset(receiveBuffer, 0, MAX_BUFF);
 while (isConnect) {
  res = recv(socketClient, receiveBuffer, MAX_BUFF, 0);
  if (res > 0) {
   cout << receiveBuffer << endl;
  }
  else if (0 == res) {
   isConnect = false;
   isSend = false;
   memset(receiveBuffer, 0, MAX_BUFF);
   cerr << "服务器关闭了连接！" << endl;
   return 0;
  }
  else if (SOCKET_ERROR == res) {
   if (WSAEWOULDBLOCK == WSAGetLastError()) {
    continue;
```

```
    }
    else {
     isConnect = false;
     cerr << "接收缓冲区不可用！" << endl;
     return 0;
    }
   }
  }
  return 0;
}
```

发送线程：依据 isSend 判断是否有数据需要发送，如果需要，则进入临界区，读取并发送 sendBuffer 中的数据到服务器。

```
// 发送数据线程
DWORD __stdcall SendDataThread(void* param) {
  while (isConnect) {
   if (isSend) {
    EnterCriticalSection(&criticalSection);
    while (true) {
     int res = send(socketClient, sendBuffer, MAX_BUFF, 0);
     if (SOCKET_ERROR == res) {
      if (WSAEWOULDBLOCK == WSAGetLastError()) {
       continue;
      }
      else {
       LeaveCriticalSection(&criticalSection);
       cerr << "发送缓冲区不可用！" << endl;
       return 0;
      }
     }
     isSend = false;
     break;
    }
    LeaveCriticalSection(&criticalSection);
   }
  }
  return 0;
}
```

创建非阻塞模式套接字：

```
  // 创建套接字
```

```
2.  int res;
3.  WSADATA wsData;
4.  res = WSAStartup(MAKEWORD(2, 2), &wsData); // 初始
    化 Windows Sockets Dll
5.  socketClient = socket(AF_INET, SOCK_STREAM, 0);
6.  if (INVALID_SOCKET == socketClient) {
7.   cerr << "套接字创建失败！" << endl;
8.   return -1;
9.  }
10. unsigned long ul = 1;
11. res = ioctlsocket(socketClient, FIONBIO, (unsigned long*)&ul);
12. if (SOCKET_ERROR == res) {
13.  cerr << "设置套接字非阻塞模式失败！" << endl;
14.  return -1;
15. }
```

连接服务器：

```
1.  // 连接服务器
2.  sockaddr_in serverAddress;
3.  serverAddress.sin_family = AF_INET;
4.  serverAddress.sin_port = htons(ServerPort);
5.  serverAddress.sin_addr.S_un.S_addr = inet_addr(ServerIP);
6.  while (true) {
7.   res = connect(socketClient, (sockaddr*)&serverAddress, sizeof(s
    erverAddress));
8.   if (0 == res) {
9.    break;
10.  }
11.  if (SOCKET_ERROR == res) {
12.   int errorCode = WSAGetLastError();
13.   if (WSAEWOULDBLOCK == errorCode || WSAEINVAL == errorCode) {
14.    continue;
15.   }
16.   else if (WSAEISCONN == errorCode) {
17.    break;
18.   }
19.   else {
20.    cerr << "连接服务器失败！" << endl;
21.    return -1;
22.   }
23.  }
24. }
25. cerr << "成功连接服务器！" << endl;
```

```
26. isConnect = true;
```

创建接收线程和发送线程：

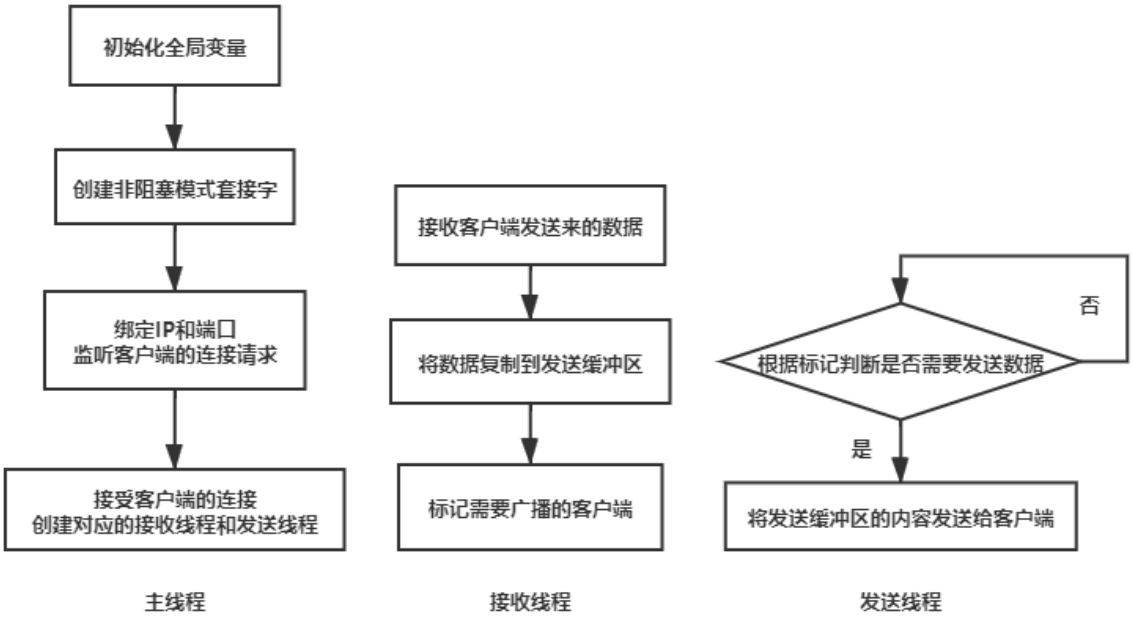
```
1.  // 创建接收和发送线程
2.  unsigned long theadID;
3.  HANDLE threadReceive = CreateThread(nullptr, 0, receiveDataThrea
    d, nullptr, 0, &theadID);
4.  if (nullptr == threadReceive) {
5.   cerr << "创建接收线程失败！" << endl;
6.   return -1;
7.  }
8.  HANDLE threadSend = CreateThread(nullptr, 0, SendDataThread, nul
    lptr, 0, &theadID);
9.  if (nullptr == threadSend) {
10.  cerr << "创建发送线程失败！" << endl;
11.  return -1;
12. }
```

等待用户输入：

```
1.  // 等待用户的输入
2.  char inputBuffer[MAX_BUFF];
3.  while (isConnect) {
4.   memset(inputBuffer, 0, MAX_BUFF);
5.   cin.getline(sendBuffer, MAX_BUFF);
6.   EnterCriticalSection(&criticalSection);
7.   memcpy(sendBuffer, inputBuffer, strlen(inputBuffer));
8.   LeaveCriticalSection(&criticalSection);
9.   isSend = true;
10.  cin.sync();
11. }
```

服务器

流程图：



主要变量：

```
1. char writeBuffer[MAX_BUFF];
2. SOCKET socketServer;
3. CRITICAL_SECTION criticalSection;
4. map<SOCKET, bool> clientMap;
```

socketServer 表示服务器用于监听和接收客户端连接请求的套接字。

clientMap 表示广播标记，每当有新的客户端连接服务器时，clientMap 新增一项 {连接该客户端的 socket：false}；每当服务器接收到客户端 A 发送来的消息时，遍历 clientMap 将除 A 之外的所有客户端对应的 value 置为 true，以标记这些客户端需要被广播消息。

writeBuffer 表示发送缓冲区，每当有客户端向服务器发送消息时，服务器接收这些消息并将其缓存到 writeBuffer。

criticalSection 表示临界区标志位，使用 criticalSection 以互斥访问 writerBuffer 和 clientMap。

主要函数：

接收线程：接收客户端 A 发送来的消息，打印该消息，进入临界区，将消息缓存到 writeBuffer，遍历 clientMap 将除 A 之外的所有客户端对应的 value 置为 true。

```
// 接收线程
DWORD __stdcall receiveDataThread(void* param) {
 Param* client = (Param*)param;
 int res;
 char buffer[MAX_BUFF];
 while (true) {
  memset(buffer, 0, MAX_BUFF);
  res = recv(client->socket, buffer, MAX_BUFF, 0);
  if (0 == res) {
   cerr << inet_ntoa(client->address.sin_addr) << ':' << ntohs(cl
ient->address.sin_port) << "\t 断开连接!
\t" << "error code: " << res << endl;
   break;
  }
  if (SOCKET_ERROR == res) {
   if (WSAEWOULDBLOCK == WSAGetLastError()) {
    continue;
   }
   else {
    cerr << inet_ntoa(client->address.sin_addr) << ':' << ntohs(c
lient->address.sin_port) << "\t 断开连接!
\t" << "error code: " << WSAGetLastError() << endl;
    break;
   }
  }
  if (res > 0) {
   cout << inet_ntoa(client->address.sin_addr) << ':' << ntohs(cl
ient->address.sin_port) << "\t" << buffer << endl;
   EnterCriticalSection(&criticalSection);
   memset(writeBuffer, 0, MAX_BUFF);
   string str = inet_ntoa(client->address.sin_addr);
   str += ":";
   str += to_string((unsigned int)ntohs(client->address.sin_port)
);
   str += "说:  ";
   str += buffer;
   memcpy(writeBuffer, str.c_str(), str.length());
   for (auto iter = clientMap.begin(); iter != clientMap.end(); +
+iter) {
```

```
    if (iter->first != client->socket) {
     iter->second = true;
    }
   }
   LeaveCriticalSection(&criticalSection);
   memset(buffer, 0, MAX_BUFF);
  }
 }
 return 0;
}
```

发送线程：进入临界区，依据 clientMap 判断是否需要发送数据，若需要，则将 writeBuffer 中的数据发送到客户端。

```
// 发送线程
DWORD __stdcall sendDataThread(void* param) {
 Param* client = (Param*)param;
 while (true) {
  EnterCriticalSection(&criticalSection);
  if (clientMap[client->socket]) {
   int res = send(client->socket, writeBuffer, strlen(writeBuffer
  ), 0);
   if (SOCKET_ERROR == res) {
    if (WSAEWOULDBLOCK == WSAGetLastError()) {
     continue;
    }
    else {
     cerr << inet_ntoa(client->address.sin_addr) << ':' << ntohs(
  client->address.sin_port) << "\t 断开连接！" << endl
      << "error code: " << WSAGetLastError() << endl;
     LeaveCriticalSection(&criticalSection);
     break;
    }
   }
   clientMap[client->socket] = false;
  }
  LeaveCriticalSection(&criticalSection);
 }
 return 0;
}
```

创建非阻塞模式套接字：

```
// 创建非阻塞模式套接字
int res;
WSADATA wsData;
res = WSAStartup(MAKEWORD(2, 2), &wsData);
socketServer = socket(AF_INET, SOCK_STREAM, 0);
if (INVALID_SOCKET == socketServer) {
 cerr << "套接字创建失败！" << endl;
 return -1;
}
unsigned long ul = 1;
res = ioctlsocket(socketServer, FIONBIO, (unsigned long*)&ul);
if (SOCKET_ERROR == res) {
 cerr << "设置套接字非阻塞模式失败！" << endl;
 return -1;
}
```

绑定 IP 和端口，监听客户端的连接请求：

```
// 绑定 IP 和端口，监听客户端的连接请求
sockaddr_in serverAddress;
serverAddress.sin_family = AF_INET;
serverAddress.sin_port = htons(ServerPort);
serverAddress.sin_addr.S_un.S_addr = INADDR_ANY;
res = bind(socketServer, (sockaddr*)&serverAddress, sizeof(serve
rAddress));
if (SOCKET_ERROR == res) {
 cerr << "套接字绑定失败！" << endl;
 return -1;
}
res = listen(socketServer, MAX_CONNECT);
if (SOCKET_ERROR == res) {
 cerr << "监听套接字失败！" << endl;
 return -1;
}
```

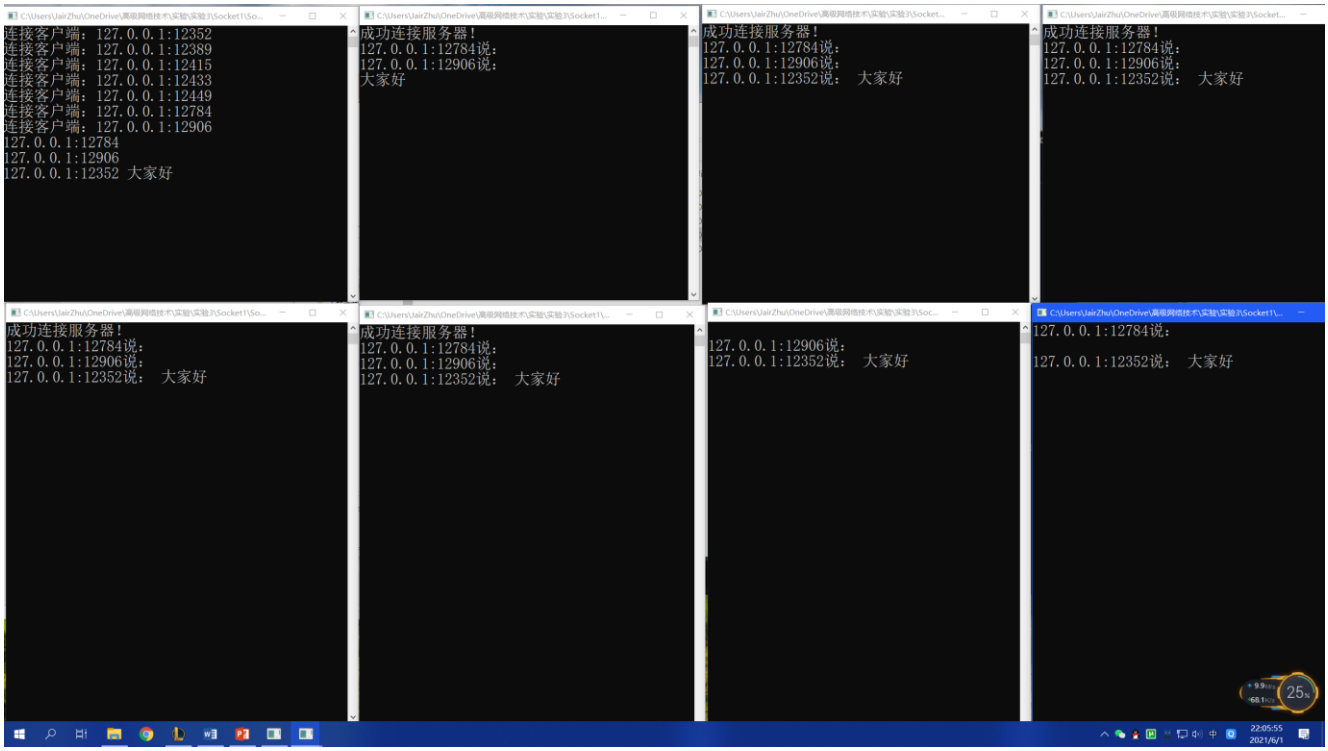
接受客户端连接：每接受一个客户端的连接请求时，建立两个与之对应的线程分别处理接收数据任务和发送数据任务。

```
// 接受客户端的连接
SOCKET socketAccept;
sockaddr_in clientAddress;
```

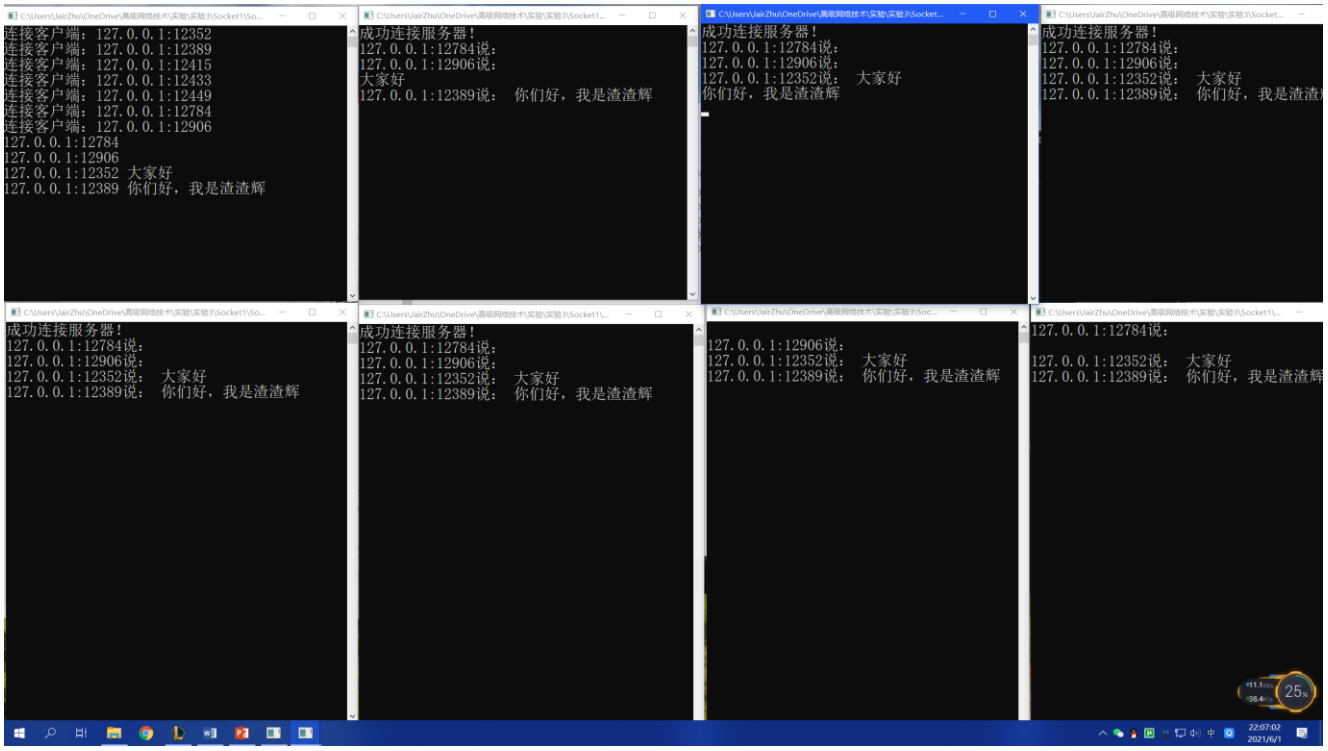
```
4.  while (true) {
5.   memset(&clientAddress, 0, sizeof(sockaddr_in));
6.   int sockaddr_in_len = sizeof(sockaddr_in);
7.   socketAccept = accept(socketServer, (sockaddr*)&clientAddress,
   &sockaddr_in_len);
8.   if (INVALID_SOCKET == socketAccept) {
9.    if (WSAEWOULDBLOCK == WSAGetLastError()) {
10.    Sleep(500);
11.    continue;
12.   }
13.   else {
14.    cerr << "接受客户端的连接失败！" << endl;
15.    cerr << "error code: " << WSAGetLastError() << endl;
16.    break;
17.   }
18.  }
19.  else {
20.   cout << "连接客户端：
   " << inet_ntoa(clientAddress.sin_addr) << ":" << ntohs(clientAddr
   ess.sin_port) << endl;
21.   unsigned long ul;
22.   clientMap[socketAccept] = false;
23.   HANDLE receiveThread = CreateThread(nullptr, 0, receiveDataThr
   ead, new Param(socketAccept, clientAddress), 0, &ul);
24.   if (nullptr == receiveThread) {
25.    cerr << "接收数据线程创建失败！" << endl;
26.    break;
27.   }
28.   HANDLE sendThread = CreateThread(nullptr, 0, sendDataThread, n
   ew Param(socketAccept, clientAddress), 0, &ul);
29.   if (nullptr == sendThread) {
30.    cerr << "发送数据线程创建失败！" << endl;
31.    break;
32.   }
33.  }
34. }
```

实验结果

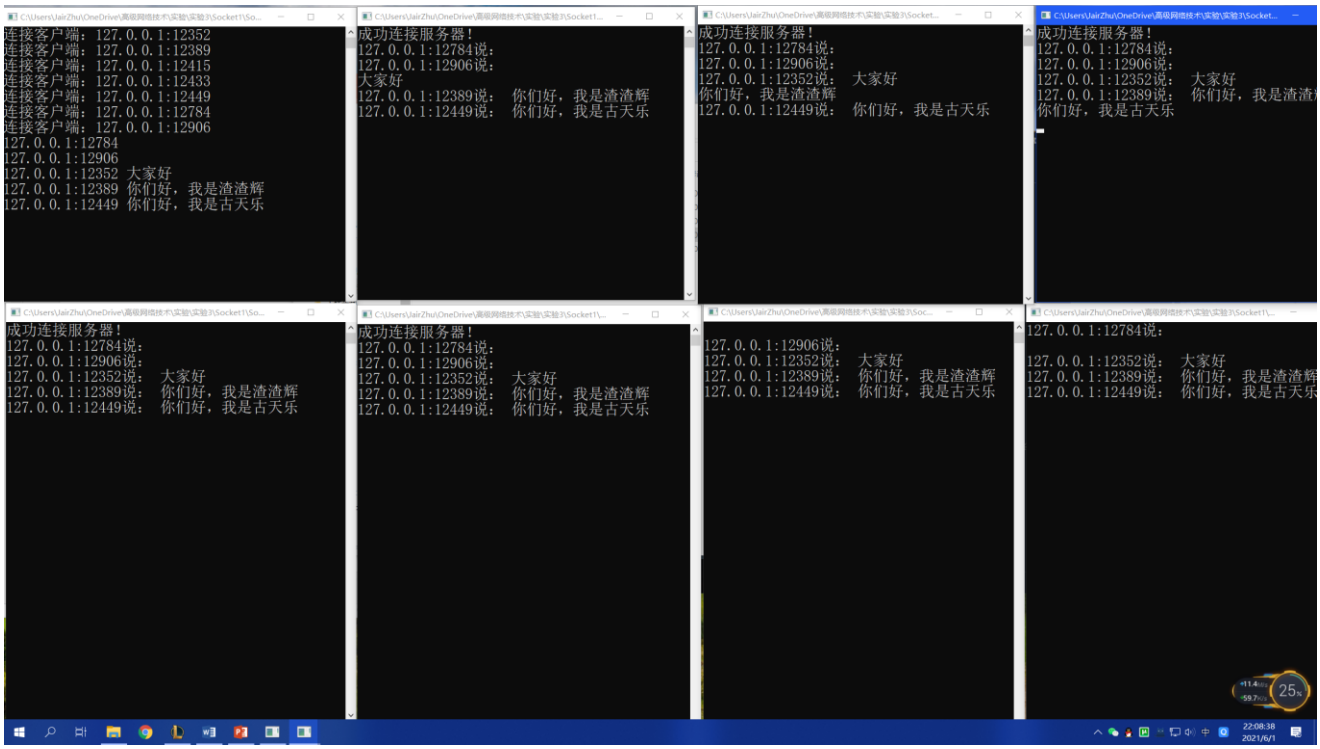
端口分别为 12352，12389，12415，12433，12449，12784，12906 的客户端分别连接服务器，进入聊天室，端口为 12352 的客户端向聊天室发送消息：



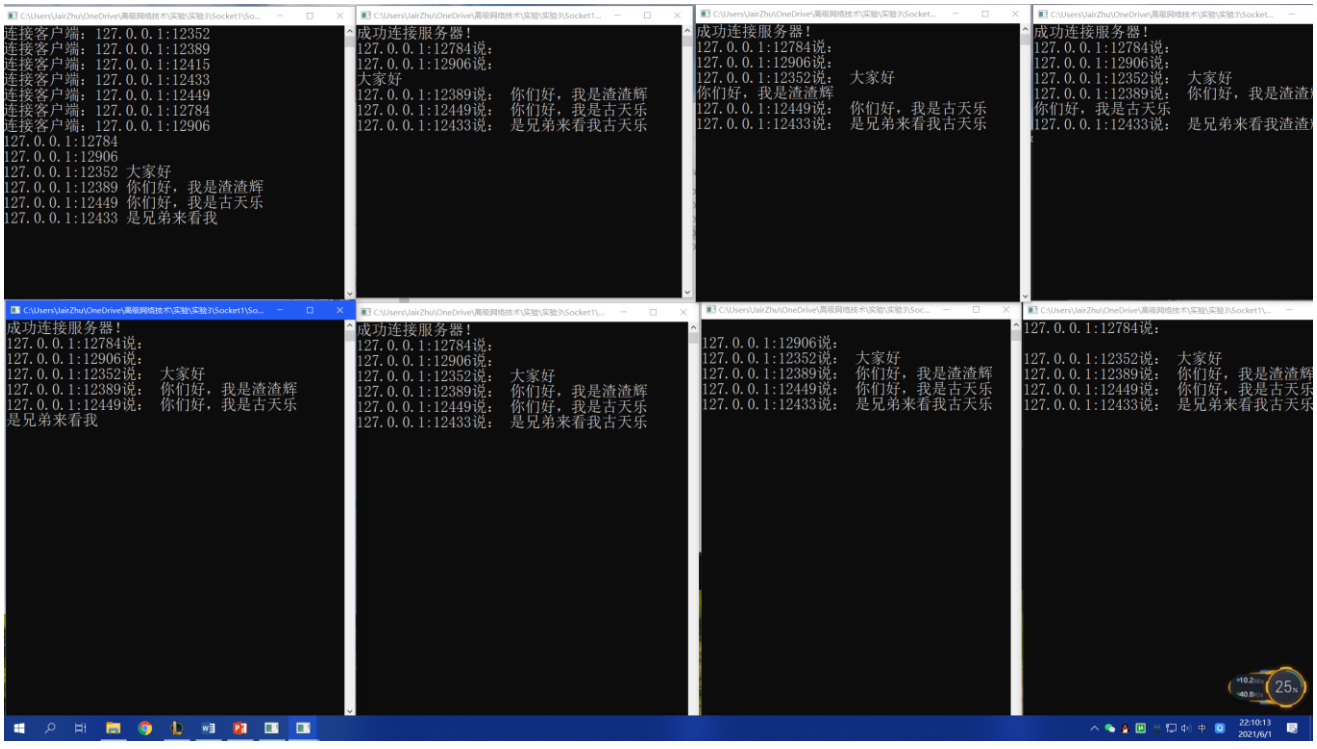
端口为 12389 的客户端向聊天室发送消息：



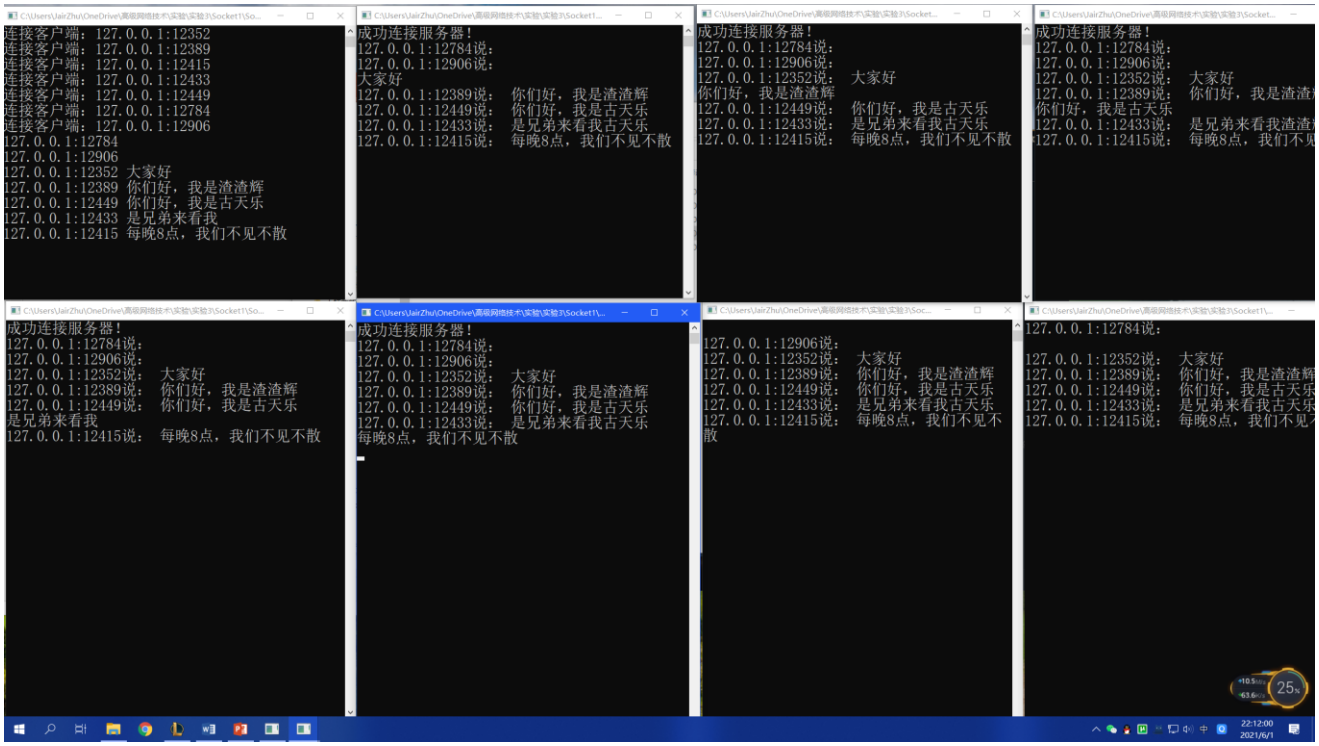
端口为 12449 的客户端向聊天室发送消息：



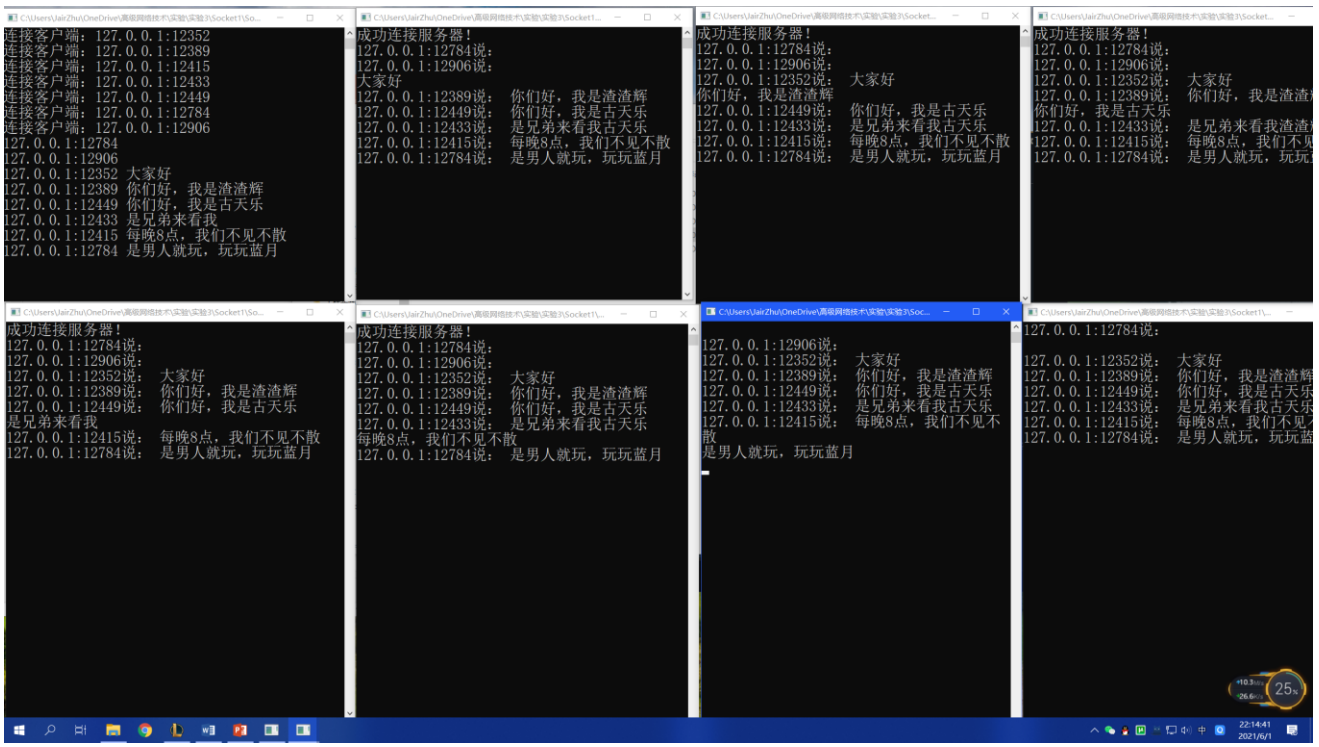
端口为 12433 的客户端向聊天室发送消息：



端口为 12415 的客户端向聊天室发送消息：



端口为 12784 的客户端向聊天室发送消息：



端口为 12906 的客户端向聊天室发送消息：