聊天服务器 实验报告

明确需求

实现一个多客户端的纯文本聊天服务器，能同时接受多个客户端的连接，并将任意一个客户端发送的文本向所有客户端（包括发送方）转发。 提交：

1. 源码；
2. 可执行的jar；
3. 实验报告。

总体架构

源代码架构示意图如下：

```
└─chat
    ├─client
    |      ChatPanel.java
    |      ClientApp.java
    |      ClientSocket.java
    |      LoginPanel.java
    |
    ├─common
    |      Message.java
    |      MessageType.java
    |
    └─server
        |  ServerApp.java
        |
        └─util
                ServerRoutine.java
```

`common`文件夹中存放POJO类`Message`，即客户端和服务端之间通信消息的封装，以及`MessageType`枚举类，用于客户端接收到`Message`时对其的解释方式。

`client`和`server`文件夹中，分别实现了客户端和服务端。其中：

- `ChatPanel, LoginPanel`为客户端的GUI视图模型，
- `ClientSocket`为客户端Socket程序逻辑，
- `ServerRoutine`为服务端程序使用的工具类。
- `ClientApp`和`ServerApp`分别为客户端和服务端的程序入口。服务端的Socket程序逻辑写到了`ServerApp`里面。

实验步骤

编写Message类

`Message`类预计将通过简单的`ObjectInputStream`和`ObjectOutputStream`串行化传递。在本程序中，只实现了两种Message类型，即纯文本和用户列表：

```
public enum MessageType {
    PLAIN_TEXT,  // 文本
    USER_LIST  // 聊天室内的用户列表
}
```

Message类如下所示。可见，`userList`也是通过文本方式（字符串）传递的。

```
public class Message implements Serializable {
    private MessageType mType;
    private String content;

    public MessageType getmType() {
        return mType;
    }

    public String getContent() {
        return content;
    }

    public Message(MessageType mType, String content) {
        this.mType = mType;
        this.content = content;
    }
}
```

编写服务端

`ServerApp`具有成员变量端口号、客户端连接列表`LinkedList<ClientConnection>`以及一个`ServerSocket`。在编写简单的构造函数：

```
public ServerApp(int port) {
    this.port = port;
    this.initSocket();  // this.socket = new ServerSocket(this.port);
}
```

之后，让其调用该`listen()`方法（省略了打印日志等的非核心代码）：

```
    public void listen() {
        while (true) {
            try {
                Socket clientSocket = this.socket.accept();

                ClientConnection clientConnection = new
ClientConnection(clientSocket);
                boolean readValid = clientConnection.readIndicator(); // 读取
Socket输入的用户名
                if (readValid == false) { // 读取不合法
                    continue;
                }
```

```
                this.clientList.add(clientConnection);
                new Thread(clientConnection).start();

                String clientIpPort = clientConnection.getClientIpPort();
                this.broadcastText("服务器",
String.format(USER_LOGIN_BROADCAST_PROMPT, clientConnection.userName));
                this.broadcastUserList();
            } catch (...) {
            }
        }
    }
```

其中，`clientConnection`为一实现了`Runnable`接口的内部类，负责管理连接服务端的某个客户端线程：

```
    private class ClientConnection implements Runnable {
        Socket clientSocket;
        String userName; // 该客户端的用户名
        String loginTime; // 该客户端的登录时间

        public ClientConnection(Socket clientSocket) {
            this.clientSocket = clientSocket;
        }

        public boolean readIndicator() {...}

        @Override
        public void run() {...}
    }
```

实现了其中的`readIndicator`方法，即可获取客户端在初次登陆时向服务端发送的用户名信息，之后便不再调用该方法。`run`方法内部为一死循环，调用`socketInput.readObject()`来获取该客户端发送的数据，然后再调用服务端提供的`broadcastText`，`broadcastUserList`方法，来向客户端列表广播该数据、以及用户列表信息。此外，一旦对`socketInput`的操作出现异常，我们便认为对端已下线，将该客户端移出客户端列表，并自然终止该线程。

编写多线程的同步方法

`broadcastText`方法，用于向客户端列表广播纯文本消息，其要求`clientList`在此过程中不改变。类似地，`broadcastUserList`向客户端列表广播当前服务端掌握的客户端列表。简单使用`synchronized`实现如下：

```
public synchronized void broadcastText(String senderName, String content) {
    final String curTime = new SimpleDateFormat("HH:mm:ss").format(new Date());
    final String textSent = String.format("[%s] %s: %s\n", curTime, senderName,
content);

    Message msg = new Message(MessageType.PLAIN_TEXT, textSent);
    this.clientList.forEach(e -> e.sendMsg(msg)); // 广播消息
}

public synchronized void broadcastUserList() {
    StringBuilder userListStr = new StringBuilder();
    this.clientList.forEach(e -> {
        userListStr.append(String.format("%s, %s\n", e.userName, e.loginTime));
```

```
        }); // 每行格式：用户名，登录时间\n
        Message msg = new Message(MessageType.USER_LIST, userListStr.toString());
        this.clientList.forEach(e -> e.sendMsg(msg));
    }
```

在`ClientConnection.run`中，如下调用这两个方法，即可保证客户端的数据同步：

```
    @Override
    public void run() {
        while (true) {
            // ObjectOutputStream socketOutput;
            try {
                ObjectInputStream socketInput = new
ObjectInputStream(clientSocket.getInputStream());
                Message message = (Message) socketInput.readObject();
                broadcastText(this.userName, message.getContent());
                broadcastUserList();
            } catch (ClassNotFoundException | IOException e) { // 目标已下线
                System.out.println(String.format(ERROR_WHILE_SENDING_MSG,
this.getClientIpPort()));
                removeClient(this.getClientIpPort());  // 不再建立连接
                broadcastText("服务器",
String.format(USER_LOGOUT_BROADCAST_PROMPT, this.userName));
                broadcastUserList();
                break;  // 线程结束
            }
        }
    }
```

编写客户端

编写客户端GUI

为提升用户友好度，客户端GUI使用Java Swing编写，包括登录窗口和聊天窗口两部分，由于不涉及网络编程核心部分，细节在此不表。

编写客户端Socket

在成功登录后，客户端Socket即得到创建。由`ClientApp`调用该`login`方法，根据返回值来确定Socket是否成功创建（若返回值为`false`，则Socket创建失败）。若Socket创建成功，还要用一个线程守候服务端发来的消息，并更新`chatPanel`。代码如下：

```
    public boolean login() {
        try {
            this.socket = new Socket(this.ip, this.port);
        } catch (IOException e) {
            return false;
        }

        try {
            ObjectOutputStream socketOut = new
ObjectOutputStream(this.socket.getOutputStream());
            socketOut.writeObject(this.userName); // send userName to server
(hello!)
        } catch (IOException e) {
            return false;
```

```
        }

        new Thread(() -> {
            while (true) {
                try {
                    ObjectInputStream socketIn = new
ObjectInputStream(socket.getInputStream());
                    Message serverMsg = (Message) socketIn.readObject();
                    String content = serverMsg.getContent();
                    switch (serverMsg.getmType()) {
                    case PLAIN_TEXT:
                        chatPanel.displayPlainText(content);
                        break;
                    case USER_LIST:
                        chatPanel.refreshUserList(content);
                        break;
                    }
                } catch (ClassNotFoundException | IOException e) {
                    chatPanel.disconnect();
                    break;  // Server down, no more effort
                }
            }
        }).start();

        this.chatPanel = new ChatPanel(this);
        return true;
    }
```

当用户希望发送消息时，由GUI绑定发送按钮至该`sendMsg`方法即可：

```
public void sendMsg(Message msg) {
    try {
        ObjectOutputStream socketOut = new
ObjectOutputStream(this.socket.getOutputStream());
        socketOut.writeObject(msg);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

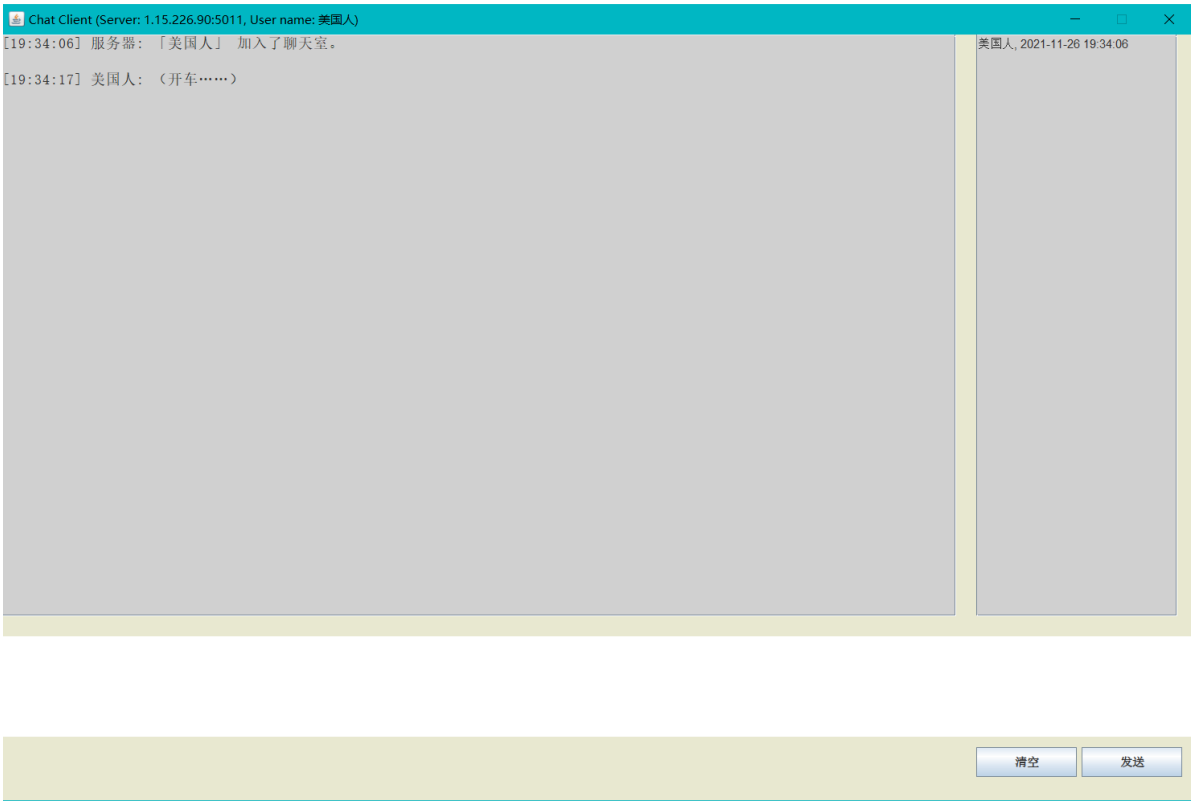
结果演示

声明：在所有结果演示中，服务端均运行在本人的一台腾讯云服务器上，端口号为5011。客户端均为本人连接校网的笔记本电脑，通过在同一台电脑上开启多个客户端程序来达到多客户端效果。

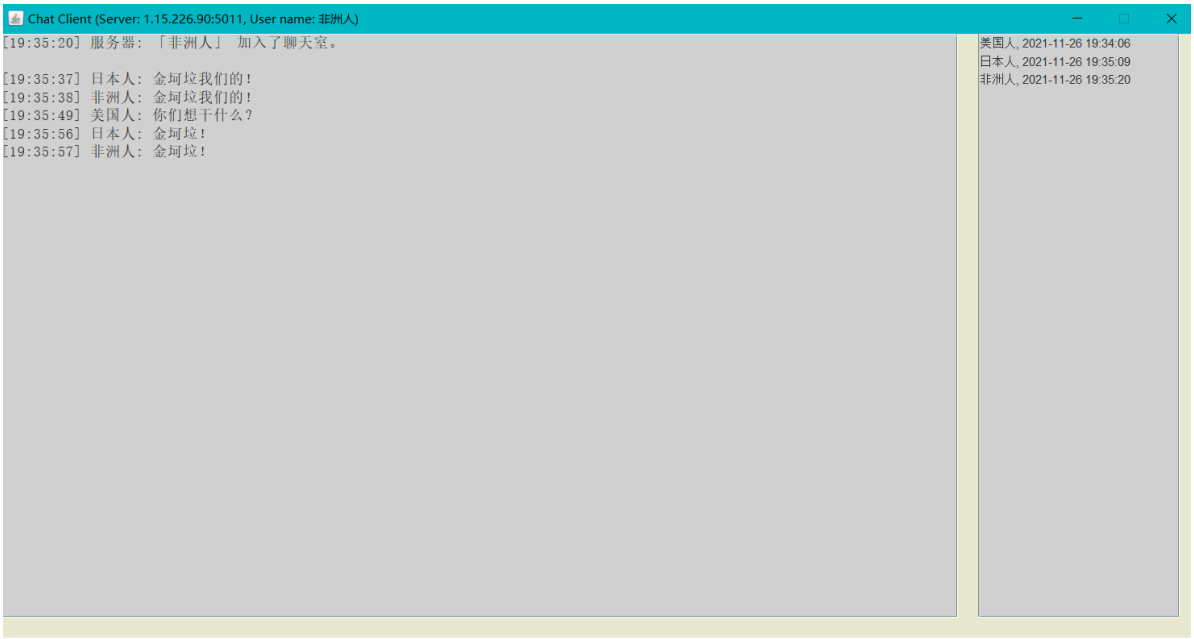
聊天演示

进行如下的聊天场景模拟演示。首先，打开客户端JAR程序，输入云服务器的公网IP地址，运行该聊天服务应用程序的端口号，并输入用户名，即可作为“美国人”登入聊天室。

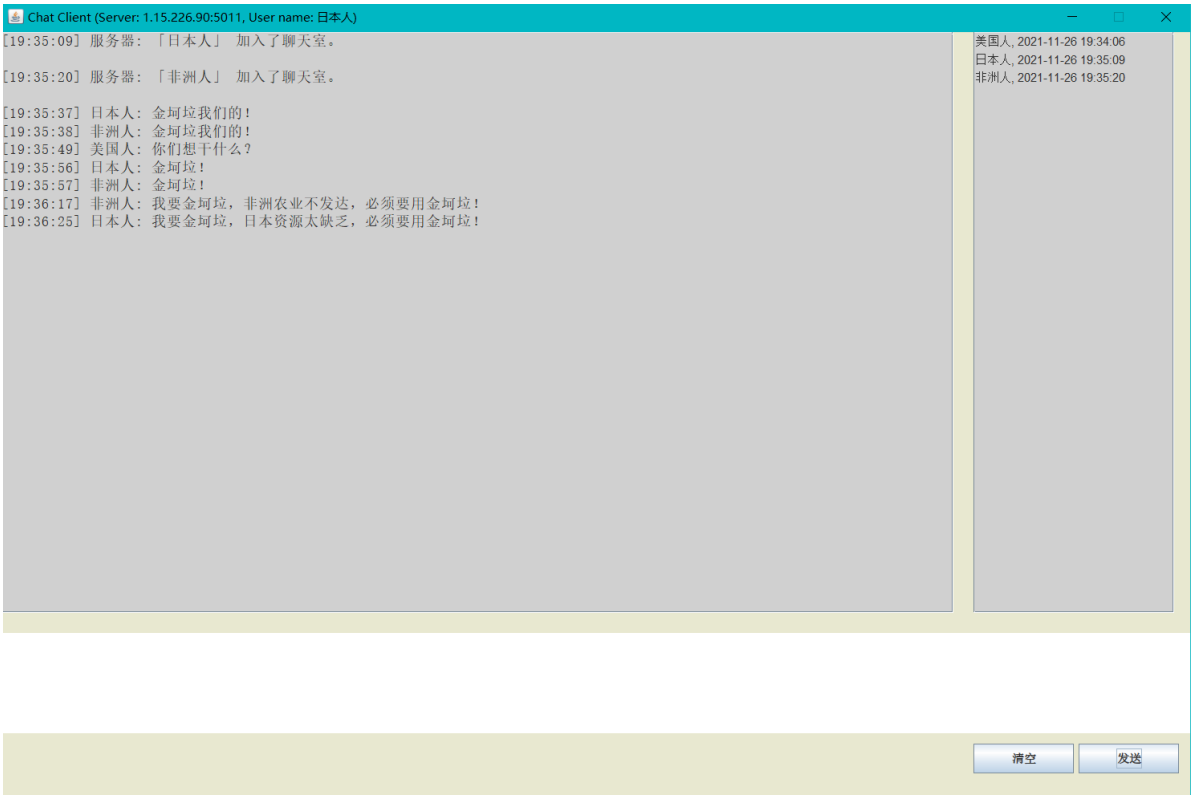
登录后，在下方文本框输入（开车……），并点击发送按钮，客户端GUI如下所示：



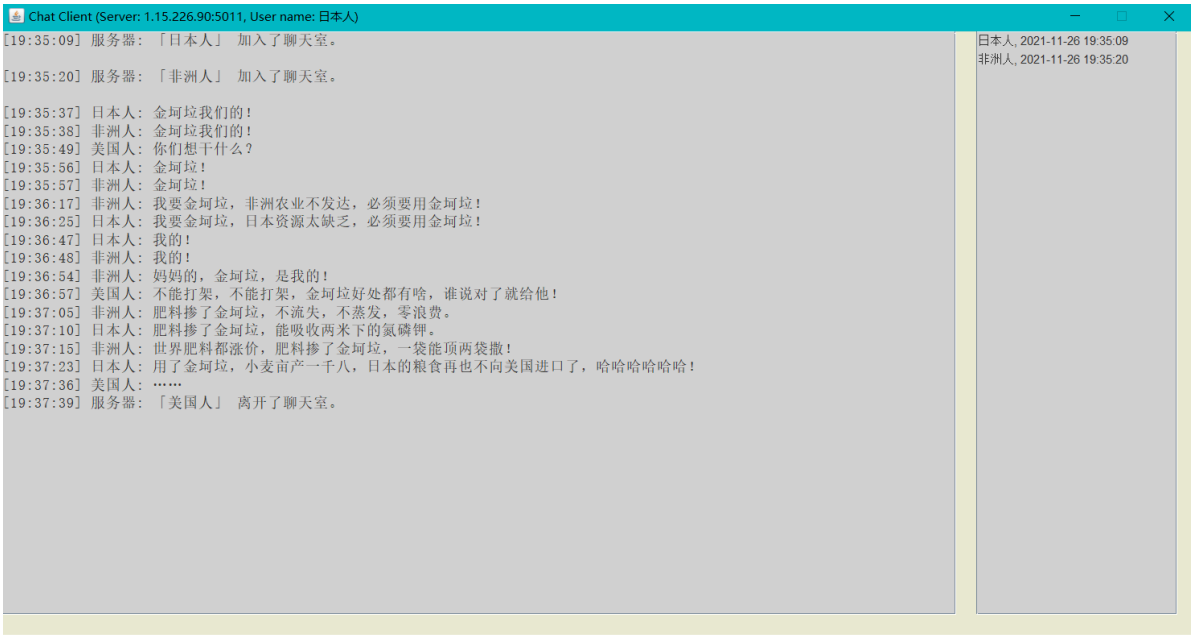
以同样的方式再开启两个客户端程序，分别作为“日本人”和“非洲人”登录，输入各自的聊天信息并发送。其中，“非洲人”在发送消息金坷垃！后，看到的客户端界面如下所示：



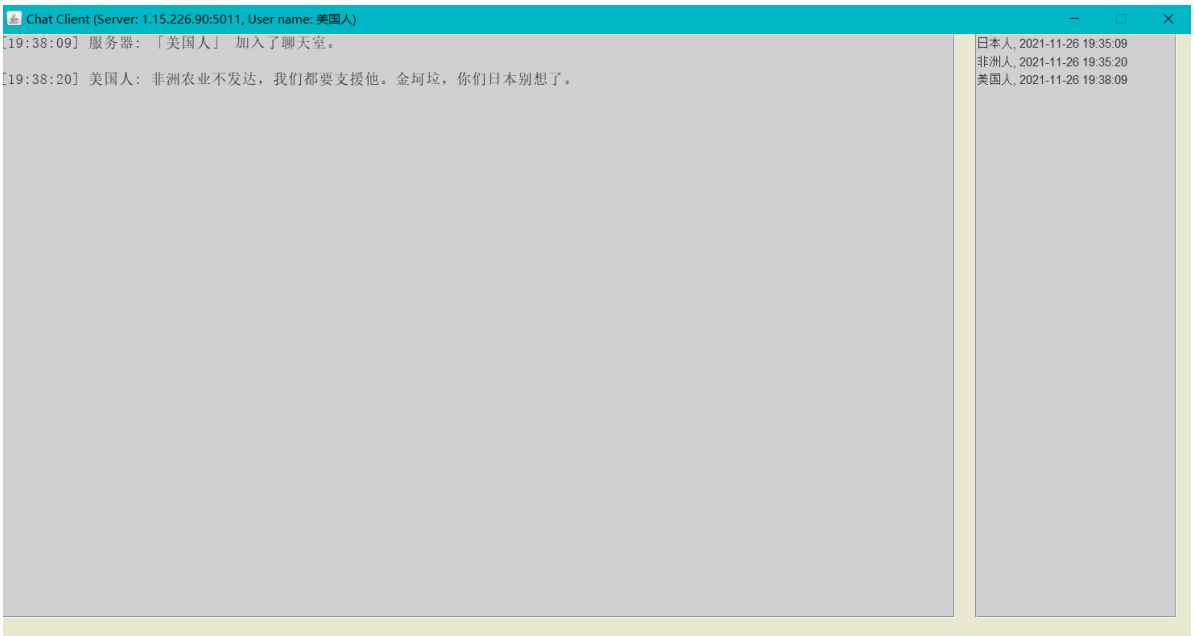
可见，当前聊天室的用户列表会在窗体右侧显示，且后登录的用户无法看到登陆之前聊天室中发送的信息。“日本人”在发送消息`我要金坷垃，日本资源太缺乏，必须要用金坷垃`后，看到的客户端界面如下所示：



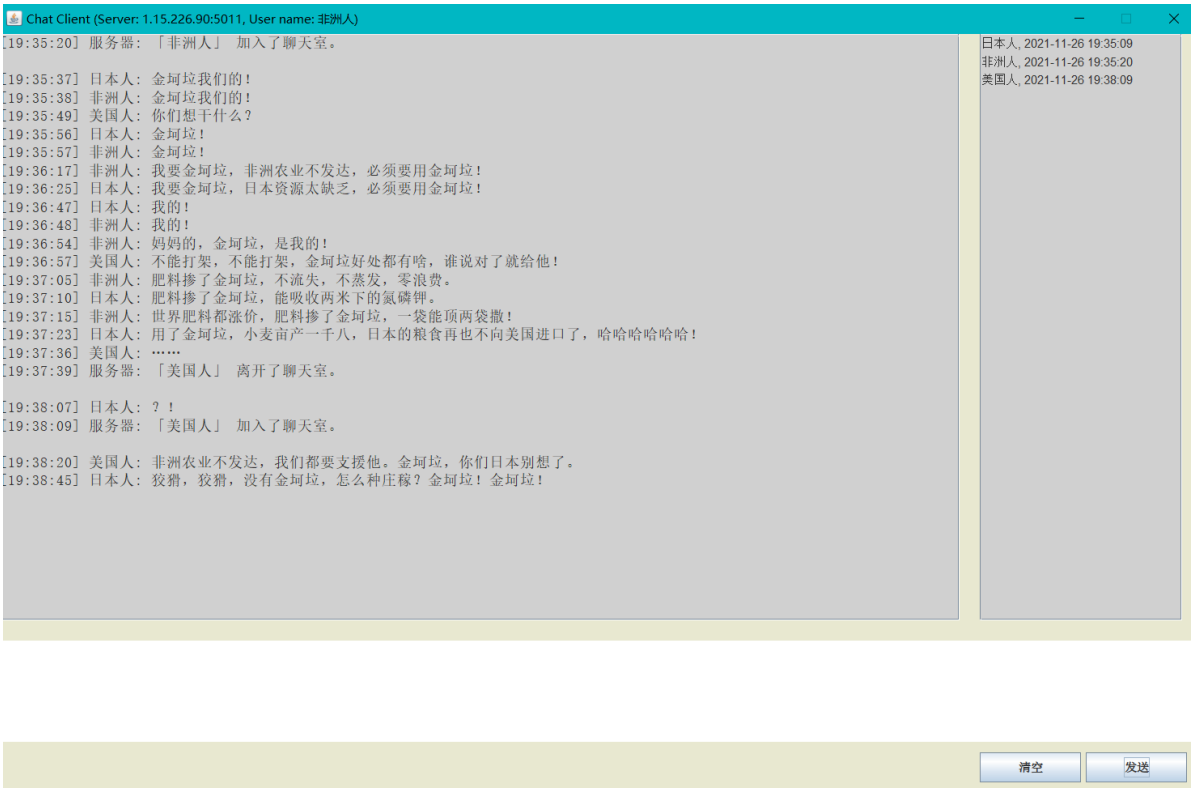
接下来，模拟美国人点击右上方关闭按钮后的登出效果。日本人和非洲人的客户端均受到了由服务器广播的“美国人离开聊天室”的消息，且用户列表得到刷新：



美国人重新登录，并发送一条消息：



最后，非洲人看到的客户端界面如下所示：

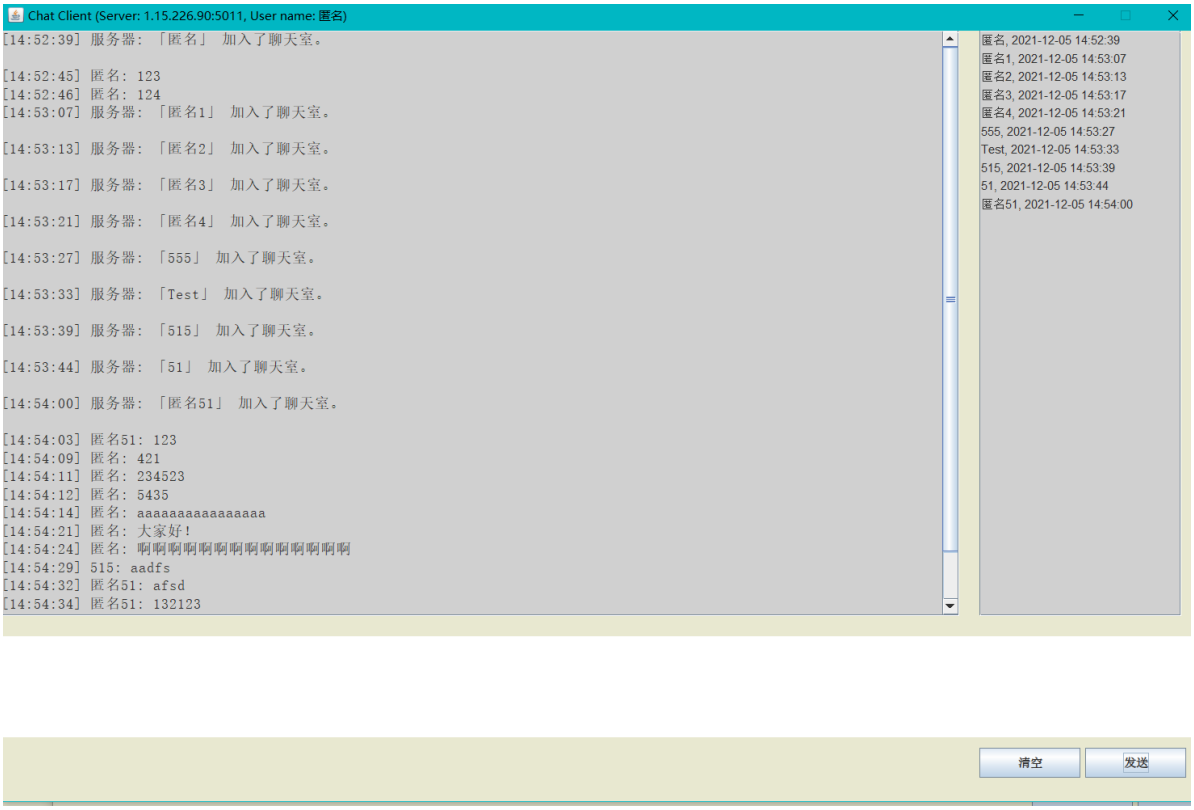


在Linux服务器上运行的服务端程序，一直以命令行的方式忠实地记录着用户的登录登出信息，如下所示（`匿名`与本次聊天演示无关）：

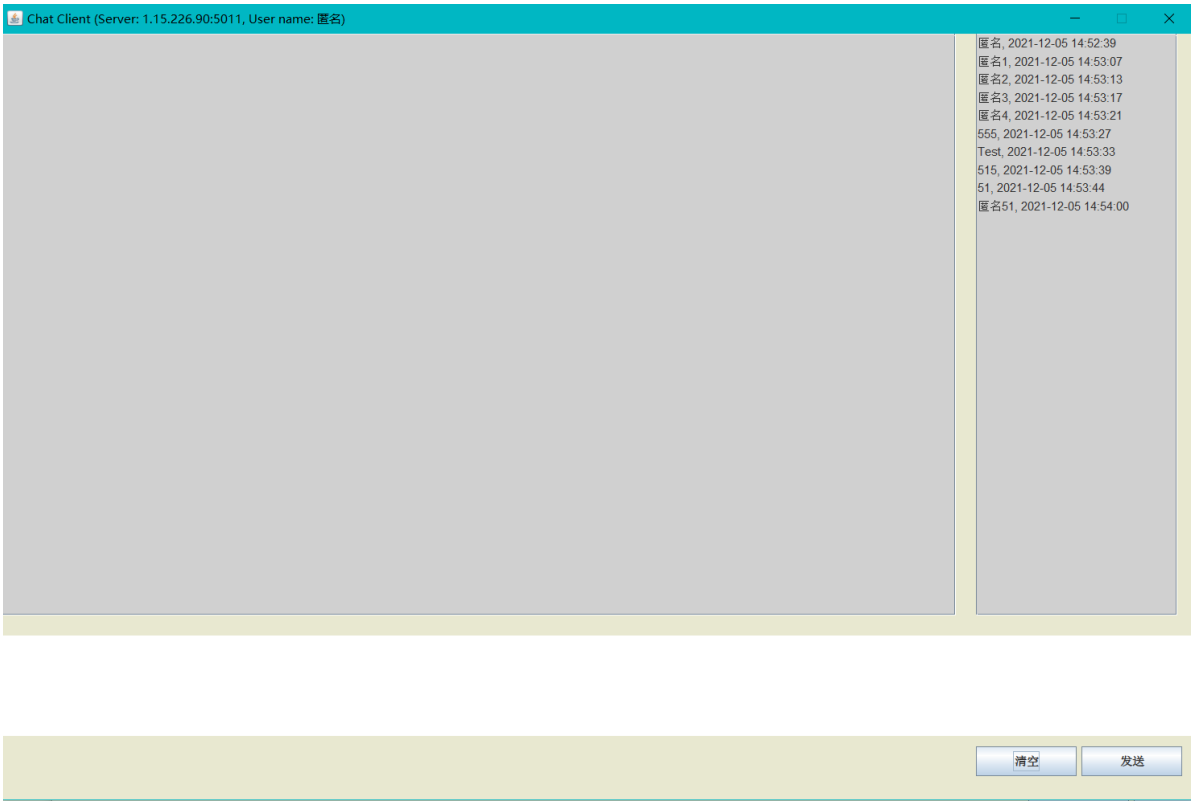
```
^Cubuntu@VM-0-5-ubuntu:~/java$ java -jar Server.jar
Server Socket is listening port 5011.
User 匿名 from /222.205.46.116:7472 has logged in.
Cannot send to client /222.205.46.116:7472. Target is offline.
User 美国人 from /222.205.46.116:7733 has logged in.
User 日本人 from /222.205.46.116:2943 has logged in.
User 非洲人 from /222.205.46.116:5380 has logged in.
Cannot send to client /222.205.46.116:7733. Target is offline.
User 美国人 from /222.205.46.116:6307 has logged in.
Cannot send to client /222.205.46.116:6307. Target is offline.
Cannot send to client /222.205.46.116:5380. Target is offline.
Cannot send to client /222.205.46.116:2943. Target is offline.
```

多用户场景测试

模拟一个多用户复杂聊天场景进行测试。



在按下清空按钮后，聊天信息被完全清空：



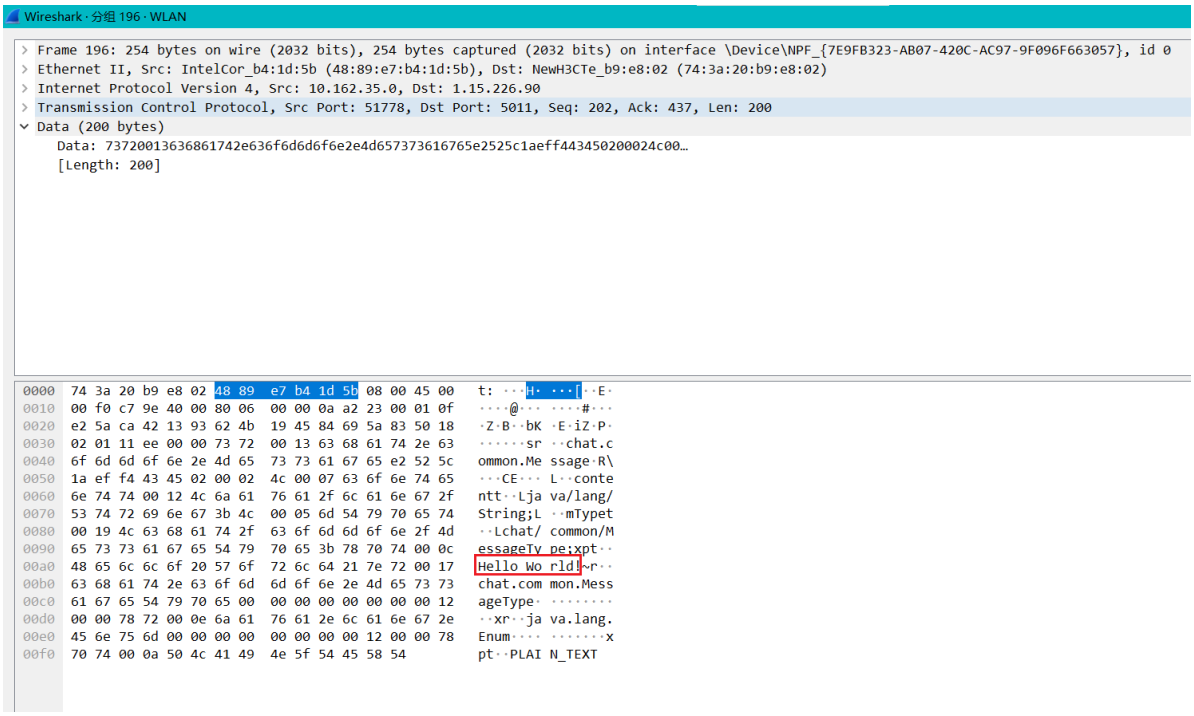
该测试下，服务端和客户端的表现均一切正常。

```
ubuntu@VM-0-5-ubuntu:~/java$ java -jar Server.jar
Please input the port number to be listened:
5011
Server Socket is listening port 5011.
User 匿名 from /222.205.46.116:2218 has logged in.
User 匿名1 from /222.205.46.116:6259 has logged in.
User 匿名2 from /222.205.46.116:6056 has logged in.
User 匿名3 from /222.205.46.116:2593 has logged in.
User 匿名4 from /222.205.46.116:2626 has logged in.
User 555 from /222.205.46.116:6148 has logged in.
User Test from /222.205.46.116:2673 has logged in.
User 515 from /222.205.46.116:2693 has logged in.
User 51 from /222.205.46.116:2720 has logged in.
User 匿名51 from /222.205.46.116:2835 has logged in.
Cannot send to client /222.205.46.116:2218. Target is offline.
Cannot send to client /222.205.46.116:2835. Target is offline.
Cannot send to client /222.205.46.116:2693. Target is offline.
Cannot send to client /222.205.46.116:6056. Target is offline.
Cannot send to client /222.205.46.116:2720. Target is offline.
Cannot send to client /222.205.46.116:2626. Target is offline.
Cannot send to client /222.205.46.116:6148. Target is offline.
Cannot send to client /222.205.46.116:2673. Target is offline.
Cannot send to client /222.205.46.116:2593. Target is offline.
```

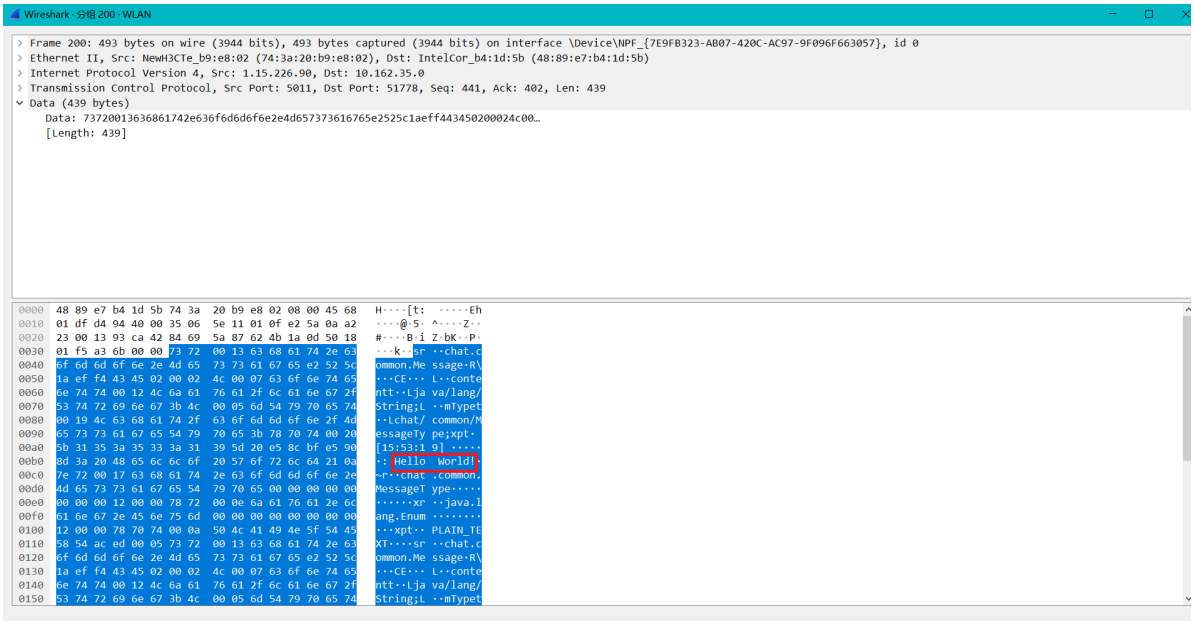
Wireshark抓包测试

在客户端运行Wireshark，并发送`Hello World!`消息的结果如下：

发送：



接收：



实际上，分析接收到的TCP包，可以发现，服务端把客户端列表也封装到了该TCP包中，其位置就在`Hello World`串行化信息之后。