

****

计算机网络课程设计

课程名称 计算机网络

题目名称 “从零开始了解化工”——基于Socket的网络聊天程序

学生学院 化学工程学院

专业班级 互联化工501&502班

学 号**2020141490193、2020141490041**

学生姓名 马骥飞、殷凯

指导教师 宋万忠

2022 年 6 月 19 日

**目录**

[“从零开始了解化工”——基于Socket的网络聊天程序 3](#_Toc29871)

[一、程序需求实现 3](#_Toc21710)

[二、软件开发工具及运行环境 3](#_Toc25919)

[1、软件开发工具： 3](#_Toc22257)

[2、运行环境： 3](#_Toc14462)

[三、程序开发的基础知识 4](#_Toc20699)

[1．学习Socket和TCP的基本原理和通信机制 4](#_Toc28132)

[2．功能设计和界面设计 5](#_Toc19676)

[四、总体设计 6](#_Toc5230)

[1.设计思路 6](#_Toc15417)

[2. 程序流程图 8](#_Toc1413)

[3.关键数据结构 9](#_Toc29452)

[4.关键性代码 13](#_Toc27620)

[五、测试 16](#_Toc9222)

[1.运行ServerMain和ClientMain 16](#_Toc7892)

[2. 开启服务器并登陆客户端 16](#_Toc25403)

[3.服务器群发消息 17](#_Toc23619)

[4. 客户端群发消息 17](#_Toc12391)

[5. 客户端发送私聊消息： 18](#_Toc116)

[6. 客户端下线 19](#_Toc19536)

## “从零开始了解化工”——基于Socket的网络聊天程序

## 一、程序需求实现

1.实现一个图形用户界面局域网内的消息系统，用NIO（非阻塞式IO）写的服务端客户端，外加jwt写的界面。

2.功能：建立一个局域网内的简单的聊天室。

2.1用户上线：服务器启动后，服务端监听上线的客户端的IP和端口号。客户端A启动与服务器通信，服务器将客户端A的信息，包括IP地址和登录时间等信息加载到一个列表中；

2.2发送消息和文件：与用户建立TCP连接，发送文件或消息。客户端发消息后(注：客户端在java编译器的终端输入消息)服务器能接收并转发消息给其他客户端。

3.用户界面：显示上线下线时间和聊天内容，并可实现清屏。

## 二、软件开发工具及运行环境

### 1、软件开发工具：

a、编程语言：Java

b、开发环境：Eclipse

c、JDK版本：1.8

d、操作系统：windows 10

### 2、运行环境：

与操作系统无关性，Windows、Linux、Mac OS X等安装了Java的运行环境JRE即可运行。

## 三、程序开发的基础知识

### 1．学习Socket和TCP的基本原理和通信机制

#### 1.1 TCP连接

TCP是面向连接的协议，TCP把连接作为最基本的抽象。每一条TCP连接唯一地被通信两端的两个端点所确定。那么，TCP连接的端点是什么呢？TCP连接的端点又叫套接字（socket），根据TCP协议的规定，端口号拼接到IP地址即构成了套接字。建立起一个TCP连接需要经过“三次握手”：

1）服务器进程准备好接受外来的连接，等待客户的连接请求，如有，则作出响应。

2）客户通过调用connect发起主动打开（active open），向服务器发出连接请求报文段，请求中的首部的同步位SYN = 1，同时选择一个初始序号seq = x。TCP规定，SYN报文段不能携带数据，则要消耗一个序号。

这时，TCP客户进入SYN-SEND（同步已发送）状态。

3）服务器收到客户端连接请求后，必须确认（ACK）客户的SYN报文段。在确认报文段中，把SYN和ACK位都置为1，确认号为ack = x + 1，同时也为自己选择一个初始序号seq = y。请注意，这个报文段也不能携带数据，但同样要消耗掉一个序号。

这时，TCP服务器进入SYN-RCVD（同步收到）状态。

4）客户在收到服务器的确认后，还要向服务器进程给出确认。确认报文段的ACK置1，确认号ack = y + 1，而自己的序号seq = x + 1。TCP规定，这个报文段可以携带数据，也可以不携带数据，如果不携带数据，下一个数据报文段的序号仍是seq = x + 1。

这时，TCP连接已经建立，客户进入ESTABLISHED（已建立连接）状态。

5）服务器收到客户的确认后，也进入ESTABLISHED状态。握手过程中传送的包里不包含数据，三次握手完毕后，客户端与服务器才正式开始传送数据。理想状态下，TCP连接一旦建立，在通信双方中的任何一方主动关闭连接之前，TCP 连接都将被一直保持下去。断开连接时服务器和客户端均可以主动发起断开TCP连接的请求，断开过程需要经过“四次握手”：

1）A的应用进程先发出释放连接报文段，并停止发送数据，主动关闭TCP连接。A把连接释放报文段首部FIN置1，其序号为seq = u。这时A进入FIN-WAIT-1（终止等待1）状态。

2）B收到连接释放报文段后即发出确认确认号为ack = u + 1，而自己的序号为seq = v。然后B就进入CLOSE-WAIT（关闭等待）状态。TCP服务器进程这时应通知高层应用进程，因而从A到B这个方向的连接就释放了，这时的TCP连接处于半关闭状态，即A已经没有数据要发送了，但B若发送数据，A仍接收。

3）A收到来自B的确认后，就进入FIN-WAIT-2（终止等待2）状态，等待B发出的连接释放报文段。

4）若B已经没有要向A发送的数据，其应用进程就通知TCP释放 连接。这时B发出的连接释放报文段FIN = 1，还必须重复上次已发送过的确认号ack = u + 1。假定B的序号为w（在半关闭期间B可能又发送了一些数据）。这时B就进入了LAST-ACK（最后确认）状态，等待A的确认。

5）A收到了的连接释放报文段后，必须对此发出确认。其确认号为ack = w + 1，而自己的序号为seq = u + 1。然后进入到TIME-WAIT（时间等待）状态。请注意，现在TCP连接还没有释放掉。必须经过时间等待计时器（TIME-WAIT timer）设置的时间 2MSL后，A才进入到CLOSED状态。时间MSL叫做最长报文段寿命（Maximum Segment Lifetime）。

6）B只要收到A发出的确认，就进入CLOSED状态。我们注意到，B结束TCP连接的时间要比A早一些。

#### 1.2 SOCKET原理

##### 1.2.1 套接字（socket）概念

套接字（socket）是通信的基石，是支持TCP/IP协议的网络通信的基本操作单元。它是网络通信过程中端点的抽象表示，包含进行网络通信必须的五种信息：连接使用的协议，本地主机的IP地址，本地进程的协议端口，远地主机的IP地址，远地进程的协议端口。

应用层通过传输层进行数据通信时，TCP会遇到同时为多个应用程序进程提供并发服务的问题。多个TCP连接或多个应用程序进程可能需要通过同一个 TCP协议端口传输数据。为了区别不同的应用程序进程和连接，许多计算机操作系统为应用程序与TCP／IP协议交互提供了套接字(Socket)接口。应用层可以和传输层通过Socket接口，区分来自不同应用程序进程或网络连接的通信，实现数据传输的并发服务。

##### 1.2.2 建立socket连接

建立Socket连接至少需要一对套接字，其中一个运行于客户端，称为ClientSocket ，另一个运行于服务器端，称为ServerSocket 。

套接字之间的连接过程分为三个步骤：服务器监听，客户端请求，连接确认。

服务器监听：服务器端套接字并不定位具体的客户端套接字，而是处于等待连接的状态，实时监控网络状态，等待客户端的连接请求

客户端请求：指客户端的套接字提出连接请求，要连接的目标是服务器端的套接字。为此，客户端的套接字必须首先描述它要连接的服务器的套接字，指出服务器端套接字的地址和端口号，然后就向服务器端套接字提出连接请求。

连接确认：当服务器端套接字监听到或者说接收到客户端套接字的连接请求时，就响应客户端套接字的请求，建立一个新的线程，把服务器端套接字的描述发给客户端，一旦客户端确认了此描述，双方就正式建立连接。而服务器端套接字继续处于监听状态，继续接收其他客户端套接字的连接请求。

#### 1.3 SOCKET连接与TCP连接

创建Socket连接时，可以指定使用的传输层协议，Socket可以支持不同的传输层协议（TCP或UDP），当使用TCP协议进行连接时，该Socket连接就是一个TCP连接。

### 2．功能设计和界面设计

#### 2.1 主要运用知识

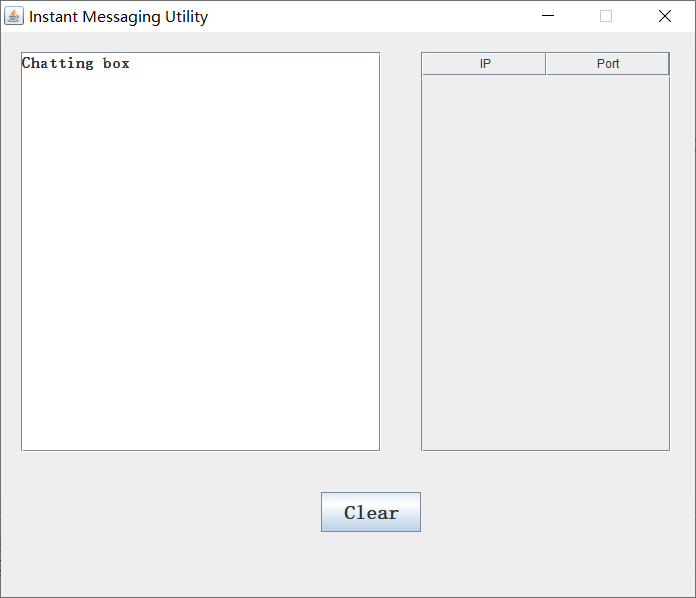
a. Java socket编程

b. Java GUI编程

c. Java继承和事件绑定

d. Java异常与捕获

#### 2.2 界面设计结果



## 四、总体设计

### 1.设计思路

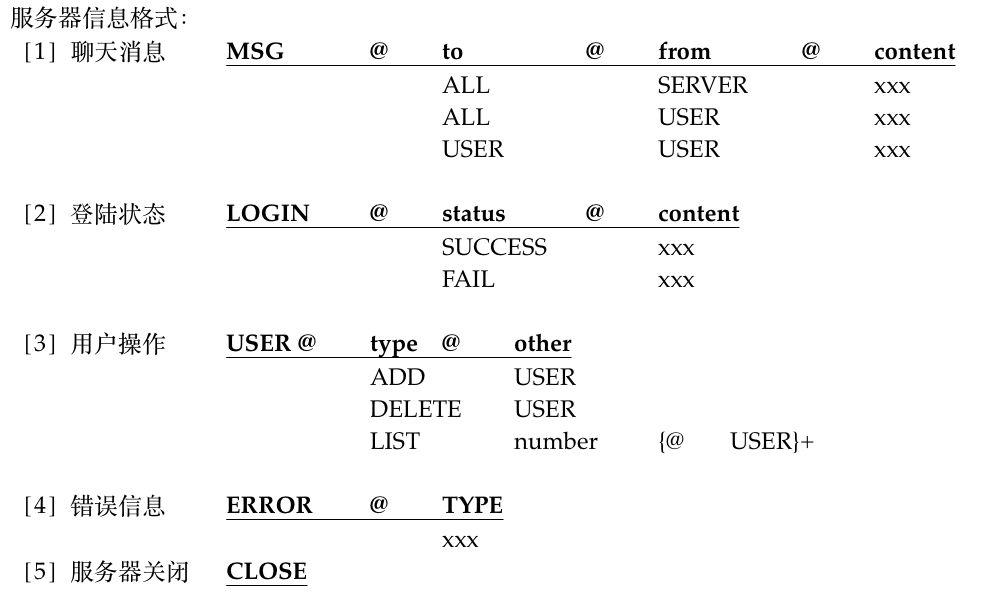
#### 1.1经典的TCP通信服务器客户端架构

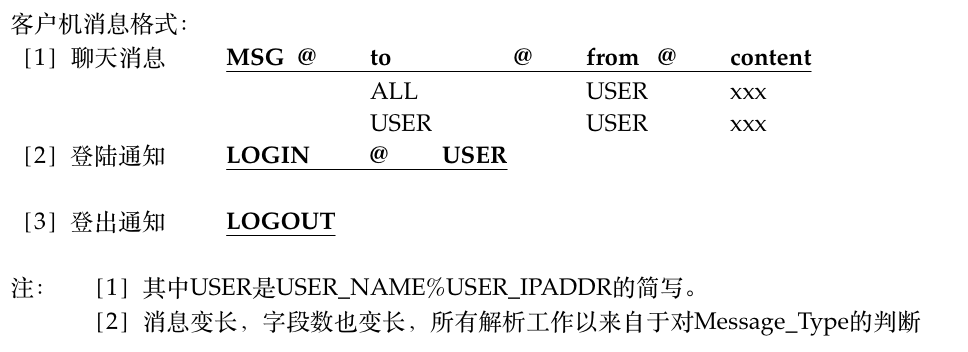
服务器有一个服务器等待用户连接的线程，该线程循环等待客户端的TCP连接请求。一旦用ServerSocket.accept()捕捉到了连接请求，就为该TCP连接分配一个客户服务线程，通过该消息传递线程服务器与客户端通信。服务器发送消息通过该客户服务线程的方法在主线程完成，而接收消息全部在客户服务线程中循环接收并处理。

客户机能发起一个向服务器的socket连接请求，一旦收到服务器成功响应连接请求，客户机便为这个socket分配一个消息接收线程，否则关闭该socket。和服务器任务分配类似，发送消息作为非常用方法在主线程中完成，而接收消息在消息接收线程中不停刷新并作相应处理。

#### 1.2统一ASCII码级文本传输协议

为了实现客户机对服务器命令的响应、服务器对客户机需求的解读以及客户机与客户机之间的消息传递，我为服务器和客户端之间通信定义了一组文本传输协议。协议属于变长文本传输协议，用@作为各字段分隔符，所有消息的首节一定是消息类型，方便解析。协议定义了以下按发送方分类的消息格式：



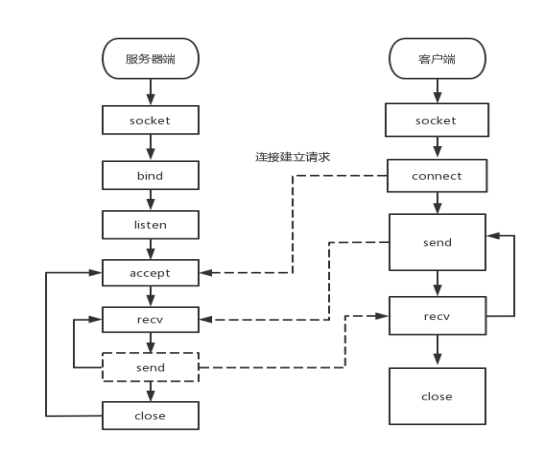


#### 1.3 MVC分层模式

Model-View-Controller是经典的应用程序开发设计模式，它讲究数据管理、界面显示和用户交互、程序维护管理分别封装在MVC三种类中，够成松耦合关系DefaultListModel模型类用于储存在线用户队列；将View单独放在一个包中，Controller监听用户操作事件，反映给Model类处理并在View中更新。

MVC的思想即是M和V之间不要直接产生联系，业务逻辑均封装在MC中，而V仅仅负责显示。本实验为V类绑定了各自的Listener监听用户操作，在C中完成业务逻辑处理，保存并更新User和DefaultListModel，最后再显示到UI界面上。但由于人手不足，暂未用到此模式，有待后续补充。

### 2. 程序流程图



### 3.关键数据结构

#### 3.1 UI类

##### （1）构造方法：

构造函数中包括初始化窗口的大小、聊天消息框、清空按钮等，用到了JScrollPane和JButton等。其中初始化了一个在线列表，暂存动态数据。

#### 3.2 GroupChatServer类

##### （1）关键字段

private Selector selector;

private ServerSocketChannel listenChannel;

##### （2）构造方法

构造函数得到了选择器，绑定了端口，也设置了非阻塞模式。

##### （3）listen方法

总思想是循环处理，通过遍历选择器（selector），确定是否有事件需要处理，得到selectKey集合。selector是可以同时监听多个SelectChannel的IO状况的多路复用器。若事件可接收，我们将这个信道相关信息注册到selector中。若通道发送可读事件，将读取事件中的信息。

##### （4）readData方法

通过选择器得到待读取消息的channel。除此之外还创建了一个buffer，将消息先写入到buffer中。若buffer测得有消息需要发送，我们将buffer中的数据转成字符串，并向其他客服端转发消息。若读取中出现异常，一概显示连接出错，将从注册的selector中删除这个key，关闭通道。

##### （5）sendInfoToOtherClients方法

转发消息。通过遍历所有注册遍历所有注册到 selector 上的 SocketChannel,并排除自己获得转发的channel。将消息写入buffer中，再将buffer写入通道中，实现转发。

#### 3.3 GroupChatClient类

##### （1）关键字段

private final String HOST = "127.0.0.1";//服务器的ip

private final int PORT = 6667;//服务器端口

private Selector selector;

private SocketChannel socketChannel;

private String username;

##### （2）构造方法

构造函数得到了选择器，绑定了端口，也设置了非阻塞模式，这里还设置了用户名username。

##### （3）Thread类

实现与服务器的交互，每3秒读取从服务器发送的数据，Run方法确定线程存活。

##### （4）sendInfo方法

消息格式为username和数据的集合，将数据写入buffer中，通过socketChannel转发到服务器。

##### （5）readInfo方法

该方法用于读取从服务器端回复的消息。首先监听可以使用的readChannel（在selector中已经注册），若该信道出现可读的信号，我们将消息读入到buffer中，转换成字符，和消息发送方的有关信息一同显示到客户端的UI中。

### 4.关键性代码

public class GroupChatServer {

    private Selector selector;

    private ServerSocketChannel listenChannel;

    private ClientFrame clientFrame;

    private static final int PORT = 6667;

    //构造器

    //初始化工作

    public GroupChatServer() {

        clientFrame = new ClientFrame();

        try {

            //得到选择器

            selector = Selector.open();

            //ServerSocketChannel

            listenChannel = ServerSocketChannel.open();

            //绑定端口

            listenChannel.socket().bind(new InetSocketAddress(PORT));

            //设置非阻塞模式

            listenChannel.configureBlocking(false);

            //将该 listenChannel 注册到 selector

            listenChannel.register(selector, SelectionKey.OP\_ACCEPT);

        } catch (IOException e) {

            e.printStackTrace();

        }

    }

    public void listen() {

        try {

            //循环处理

            while (true) {

                int count = selector.select();

                if (count > 0) { //有事件处理

                    // 遍历得到 selectionKey 集合

                    Iterator<SelectionKey> iterator = selector.selectedKeys().iterator();

                    while (iterator.hasNext()) {

                        //取出 selectionkey

                        SelectionKey key = iterator.next();

                        //监听到 accept

                        if (key.isAcceptable()) {

                            SocketChannel sc = listenChannel.accept();

                            sc.configureBlocking(false);

                            //将该 sc 注册到 seletor

                            sc.register(selector, SelectionKey.OP\_READ);

                            //提示

//                            System.out.println(sc.getRemoteAddress() + " 上线 ");

                            String remoteaddress = sc.getRemoteAddress().toString();

                            String[] split = remoteaddress.split(":");

                            clientFrame.setRowData(split[0],split[1]);

                        }

                        if (key.isReadable()) {//通道发送read事件，即通道是可读的状态

                            // 处理读(专门写方法..)

                            readData(key);

                        }

                        //当前的 key 删除，防止重复处理

                        iterator.remove();

                    }

                } else {

                    System.out.println("等待....");

                }

            }

        } catch (Exception e) {

            e.printStackTrace();

        } finally {

        }

    }

    //读取客户端消息

    public void readData(SelectionKey key) {

        SocketChannel channel = null;

        try {

            //得到 channel

            channel = (SocketChannel) key.channel();

            //创建 buffer

            ByteBuffer buffer = ByteBuffer.allocate(1024);

            int count = channel.read(buffer);

            //根据 count 的值做处理

            if (count > 0) {

                //把缓存区的数据转成字符串

                String msg = new String(buffer.array());

                //输出该消息

                System.out.println("from Client: " + msg);

                clientFrame.addinfo(msg);

                //向其它的客户端转发消息(去掉自己),专门写一个方法来处理

                sendInfoToOtherClients(msg, channel);

            }

        } catch (IOException e) {

            try {

                System.out.println(channel.getRemoteAddress() + "disconnect..");

                String remoteaddress = channel.getRemoteAddress().toString();

                String[] split = remoteaddress.split(":");

                clientFrame.delRowData(split[0],split[1]);

                //取消注册

                key.cancel();

                //关闭通道

                channel.close();

            } catch (IOException e2) {

                e2.printStackTrace();

            }

        }

    }

    //转发消息给其它客户(通道)

    private void sendInfoToOtherClients(String msg, SocketChannel self) throws IOException {

        System.out.println("服务器转发消息中...");

        //遍历所有注册到 selector 上的 SocketChannel,并排除 self

        for (SelectionKey key : selector.keys()) {

            //通过 key 取出对应的 SocketChannel

            Channel targetChannel = key.channel();

            //排除自己

            if (targetChannel instanceof SocketChannel && targetChannel != self) {

                //转型

                SocketChannel dest = (SocketChannel) targetChannel;

                //将 msg 存储到 buffer

                ByteBuffer buffer = ByteBuffer.wrap(msg.getBytes());

                //将 buffer 的数据写入通道

                dest.write(buffer);

            }

        }

    }

    public static void main(String[] args) {

        //创建服务器对象

        GroupChatServer groupChatServer = new GroupChatServer();

        groupChatServer.listen();

    }

}

public class GroupChatClient {

    private final String HOST = "127.0.0.1";//服务器的ip

    private final int PORT = 6667;//服务器端口

    private Selector selector;

    private SocketChannel socketChannel;

    private String username;

    //构造器,完成初始化工作

    public GroupChatClient() throws IOException {

        selector = Selector.open();

        //连接服务器

        socketChannel = SocketChannel.open(new InetSocketAddress(HOST, PORT));

        //设置非阻塞

        socketChannel.configureBlocking(false);

        //将 channel 注册到selector

        socketChannel.register(selector, SelectionKey.OP\_READ);

        //得到 username

        username = socketChannel.getLocalAddress().toString().substring(1);

        System.out.println(username + " is ok...");

    }

    //向服务器发送消息

    public void sendInfo(String info) {

        info = username + " Sending: " + info;

        try {

            socketChannel.write(ByteBuffer.wrap(info.getBytes()));

        } catch (IOException e) {

            e.printStackTrace();

        }

    }

    //读取从服务器端回复的消息

    public void readInfo() {

        try {

            int readChannels = selector.select();

            if (readChannels > 0) {//有可以用的通道

                Iterator<SelectionKey> iterator = selector.selectedKeys().iterator();

                while (iterator.hasNext()) {

                    SelectionKey key = iterator.next();

                    if (key.isReadable()) {

                        //得到相关的通道

                        SocketChannel sc = (SocketChannel) key.channel();

                        //得到一个 Buffer

                        ByteBuffer buffer = ByteBuffer.allocate(1024);

                        //读取

                        sc.read(buffer);

                        //把读到的缓冲区的数据转成字符串

                        String msg = new String(buffer.array());

                        System.out.println(msg.trim());

                    }

                }

                iterator.remove(); //删除当前的 selectionKey,防止重复操作

            } else {

                //System.out.println("没有可以用的通道...");

            }

        } catch (Exception e) {

            e.printStackTrace();

        }

    }

    public static void main(String[] args) throws Exception {

        //启动我们客户端

        GroupChatClient chatClient = new GroupChatClient();

        //启动一个线程,每个 3 秒，读取从服务器发送数据

        new Thread() {

            public void run() {

                while (true) {

                    chatClient.readInfo();

                    try {

                        Thread.currentThread().sleep(3000);

                    } catch (InterruptedException e) {

                        e.printStackTrace();

                    }

                }

            }

        }.start();

        //发送数据给服务器端

        Scanner scanner = new Scanner(System.in);

        while (scanner.hasNextLine()) {

            String s = scanner.nextLine();

            chatClient.sendInfo(s);

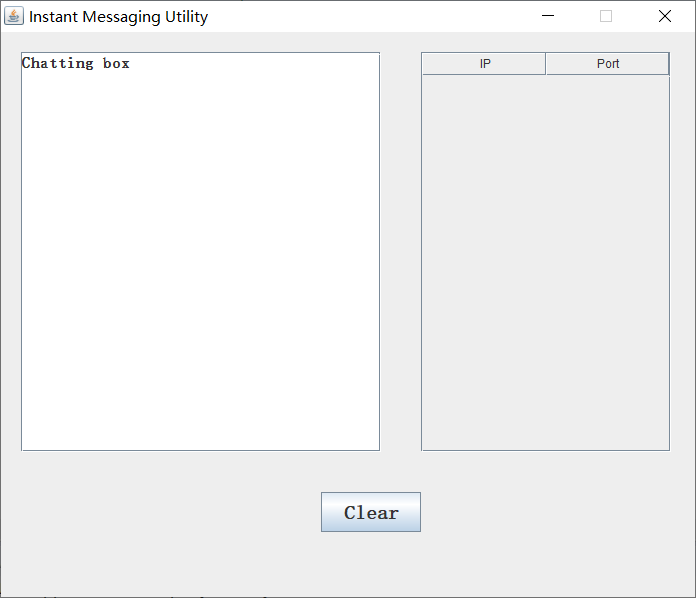
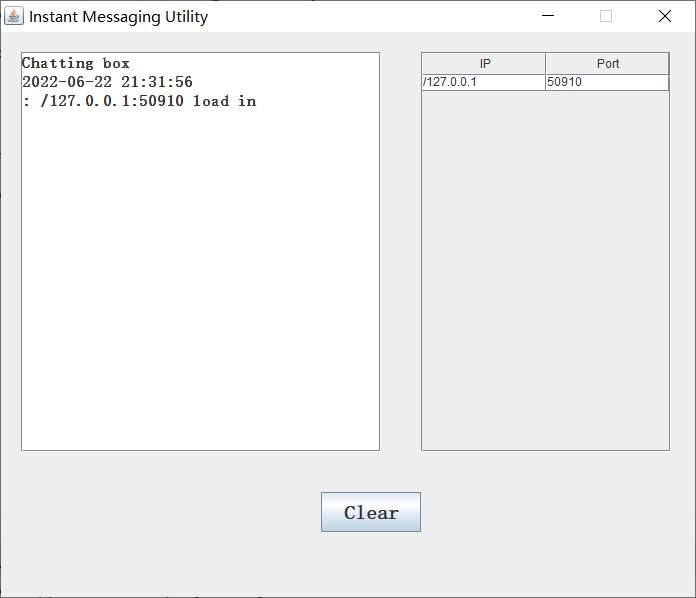
        }

    }

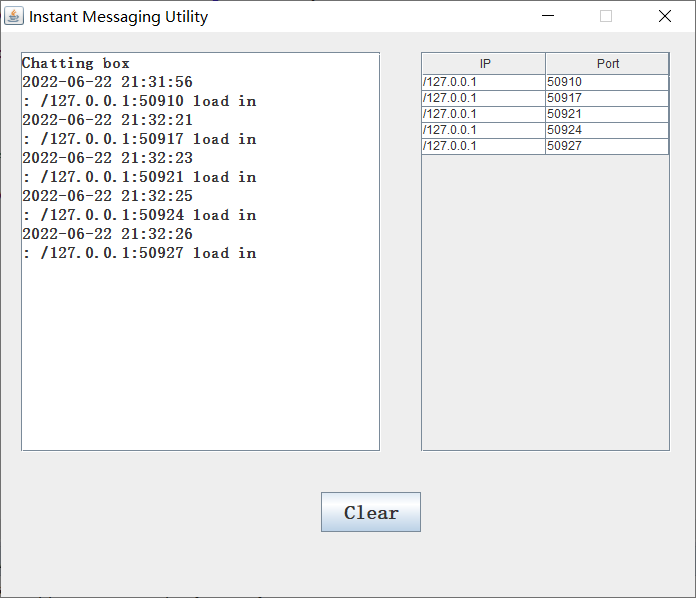
}

## 五、测试

### 1.运行GroupChatClient和GroupChatServer

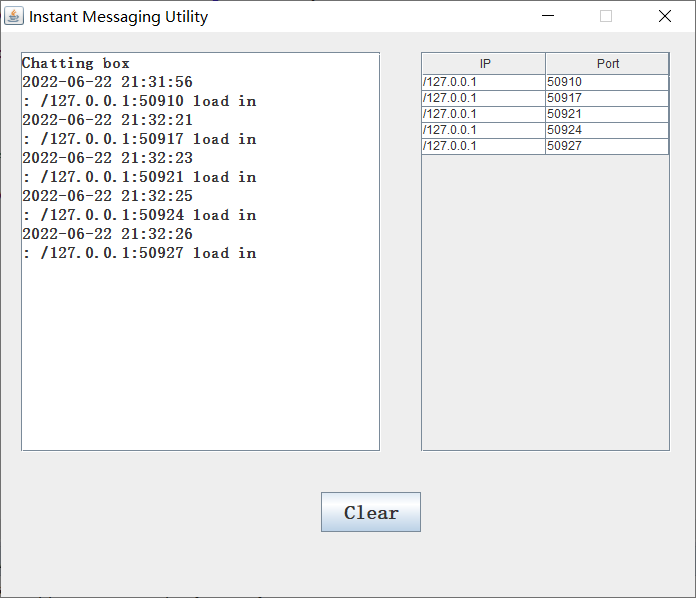
本程序支持单服务器多客户端，我分别以单服务器单客户和单服务器多客户演示。进程开启后界面如下：

服务器启动 单客户端



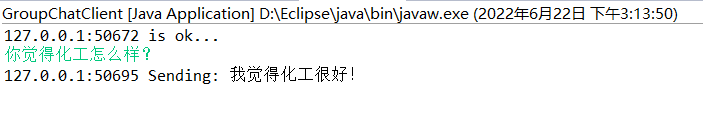
多客户端

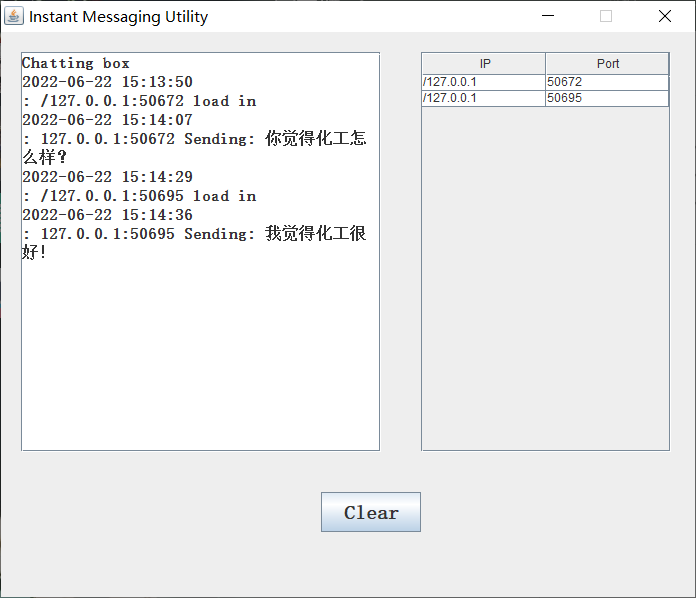
### 2.服务器群发消息



服务器群发消息显示目前在线客户，动态显示客户上线和下线。

### 3. 客户端群发消息

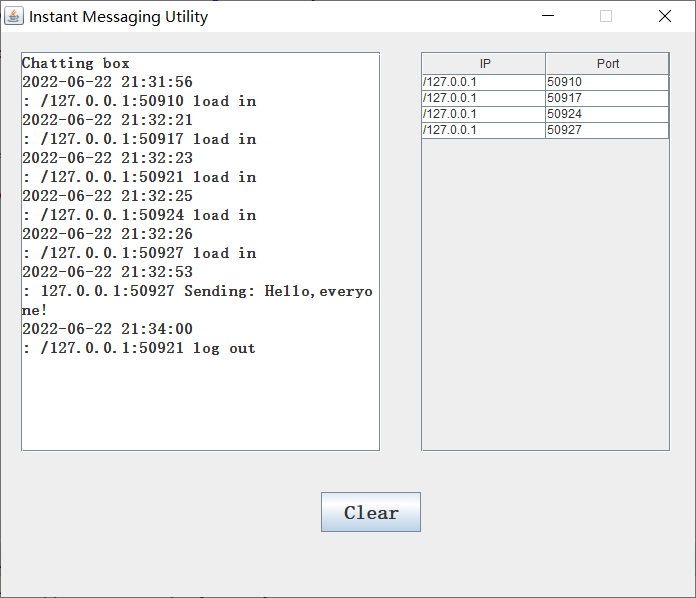




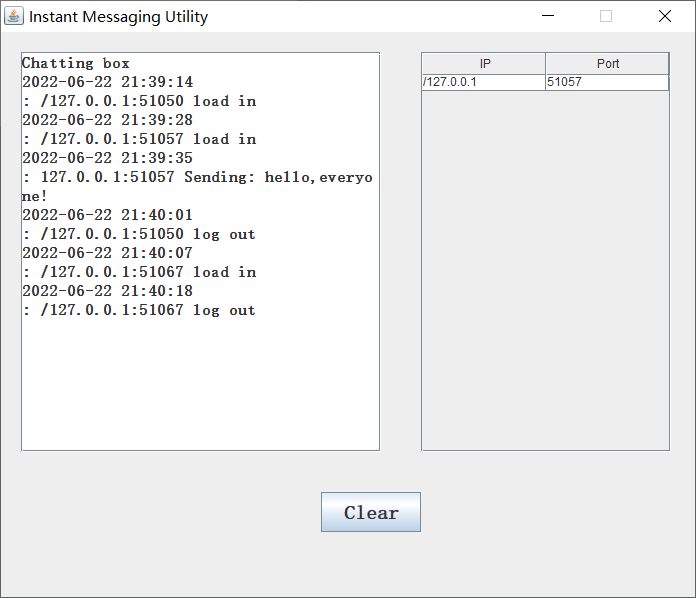
### 4. 客户端下线

客户端下线时所有在线用户收到其下线消息，在线列表中不再出现此用户。

客户3下线，通知服务器，服务器转发其下线消息：



多客户端下线：



### 5. 客户端清屏

在消息过多时执行清屏操作。

