

多客户端聊天服务器 实验报告

作者：鲍奕帆

学号：3180103499

日期：2020年12月19日

系统

macOS Catalina

版本 10.15.5

MacBook Pro (13-inch, 2020, Four Thunderbolt 3 ports)

处理器 2 GHz 四核Intel Core i5

内存 16 GB 3733 MHz LPDDR4

启动磁盘 Macintosh HD

图形卡 Intel Iris Plus Graphics 1536 MB

Java版本

```
evan@EvandeMacBook-Pro ~ % java --version
java 14.0.2 2020-07-14
Java(TM) SE Runtime Environment (build 14.0.2+12-46)
Java HotSpot(TM) 64-Bit Server VM (build 14.0.2+12-46, mixed mode, sharing)
```

项目要求

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。

功能介绍

本项目实现了多客户端的纯文本聊天服务器，能够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。同时也实现了客户端的GUI窗口，用于清晰的表示连接、输入、输出内容。

服务器运行情况

运行服务器程序，会出现入下窗口，要求输入端口号，默认端口号是8888，服务器的ip自然是本机ip。

```
evan@EvandeMacBook-Pro Java_lab2 % java Server
input server port you are going to use(0 for default port --- 8888):
```

如果输入前1024的端口号，则会输出"well know ports are not allowed, please try agian"。一般情况下，前1024端口大多被占用，因此这里自己实现的服务器不使用这些端口。

```
input server port you are going to use(0 for default port --- 8888):
80
well known ports are not allowed, please try again
```

在端口输入后服务器提示服务器启动，输出输入端口信息，并进入守听状态。

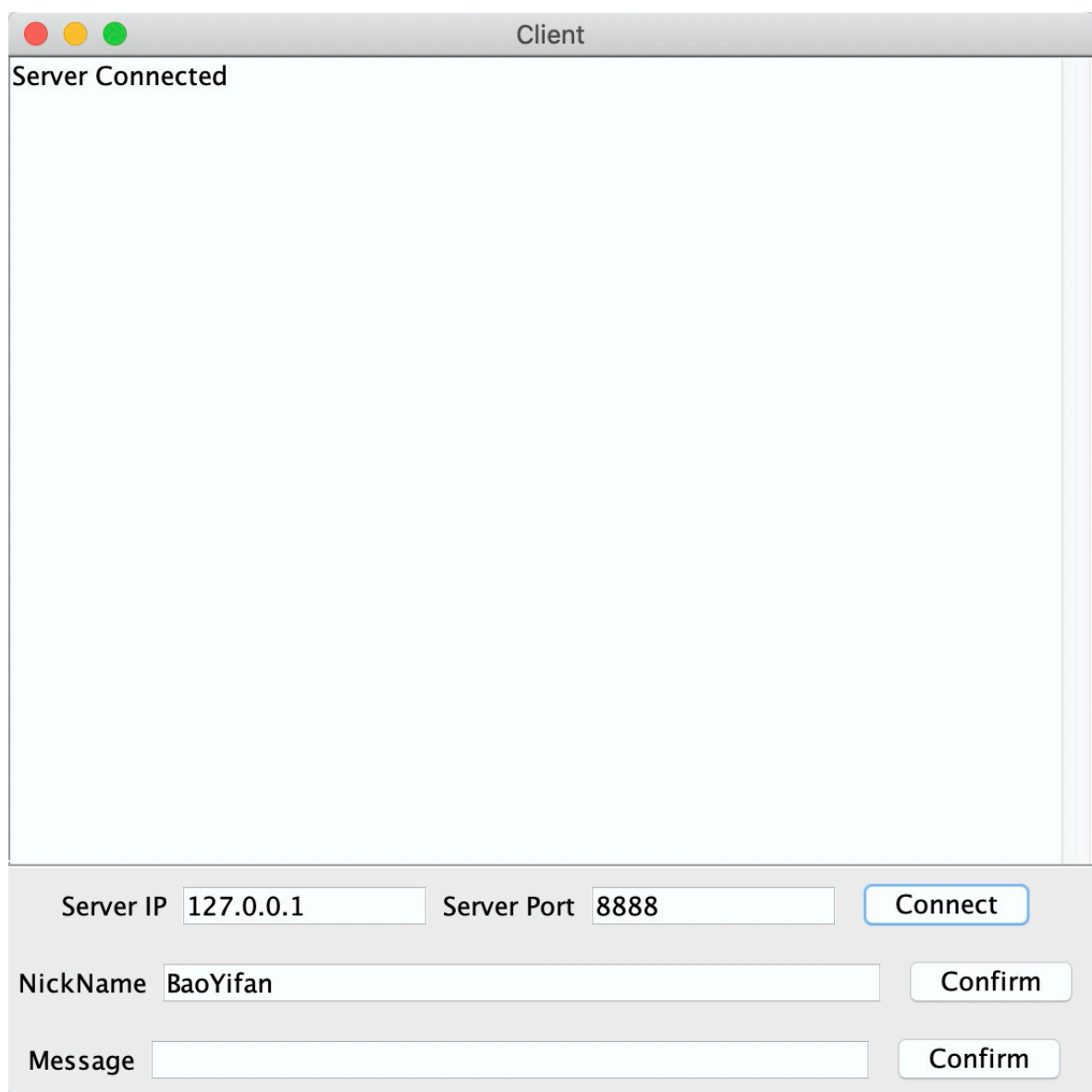
```
input server port you are going to use(0 for default port --- 8888):  
8888  
server port is: 8888.  
Starting server
```

当客户端连接的时候，服务器在标准输出中输出"one client is connected, this is the i th client "。本服务器只记录登陆的客户端的次序，但并不处理客户端离开的情况。如果要输出在线人数，可以通过ArrayList中元素的个数表示，并不困难，不过这里并没有实现。

```
one client is connected, this is the 1th client
```

客户端运行情况

客户端的基本界面



The screenshot shows a window titled "Client". The main area of the window is light blue and contains the text "Server Connected". Below this area is a form with three rows of input fields and buttons. The first row has "Server IP" with the value "127.0.0.1", "Server Port" with the value "8888", and a "Connect" button. The second row has "NickName" with the value "BaoYifan" and a "Confirm" button. The third row has "Message" with an empty text box and a "Confirm" button.

在输入Server IP与Server Port后，可以进行连接。连接成功后，会在上方的消息显示窗口显示Server Conneced表明连接成功。

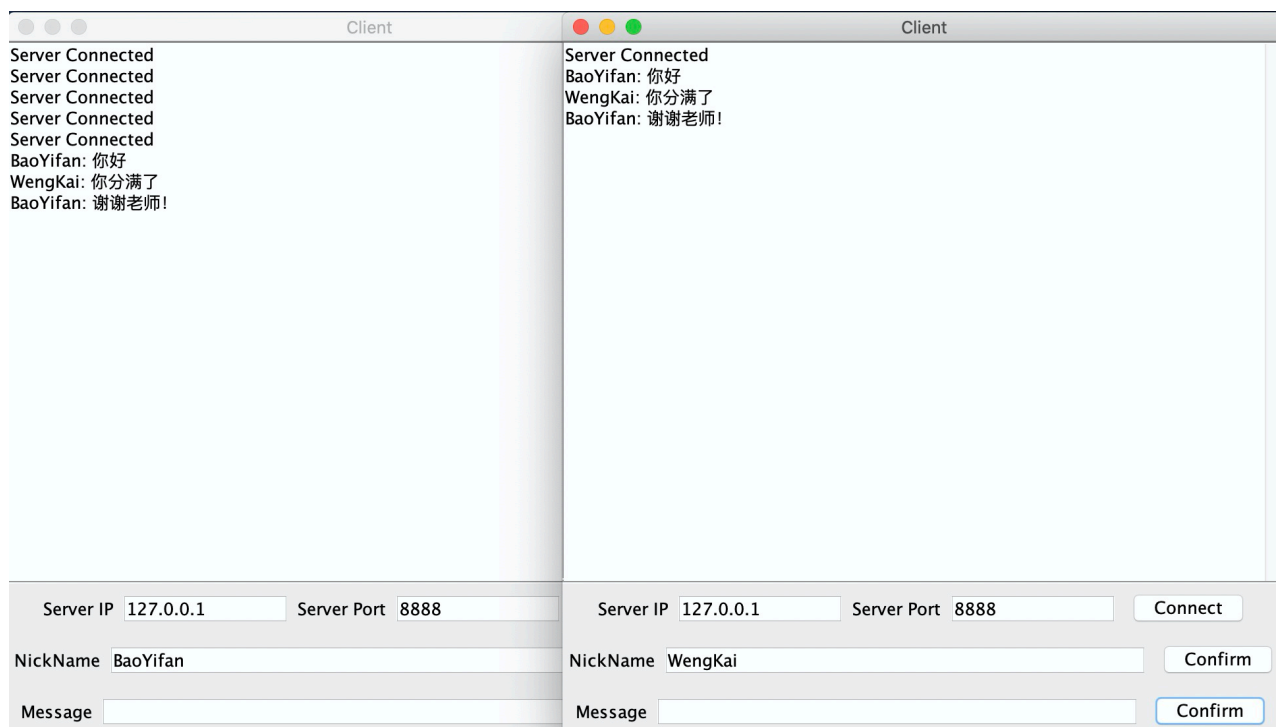
如果IP或者端口输入错误，则会在对话框中输出"Please input appropriate IP and PORT!"



如果与服务器连接失败，则会在对话框中输出"Cannot connect to server! Please verify your input"

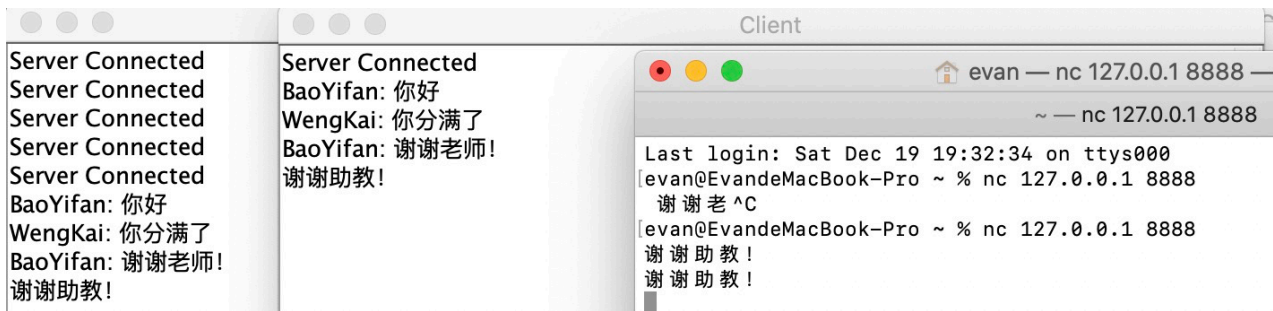


成功建立连接后，即可发送消息，同时可以设置自己的昵称。昵称并不唯一，可以随意设置。



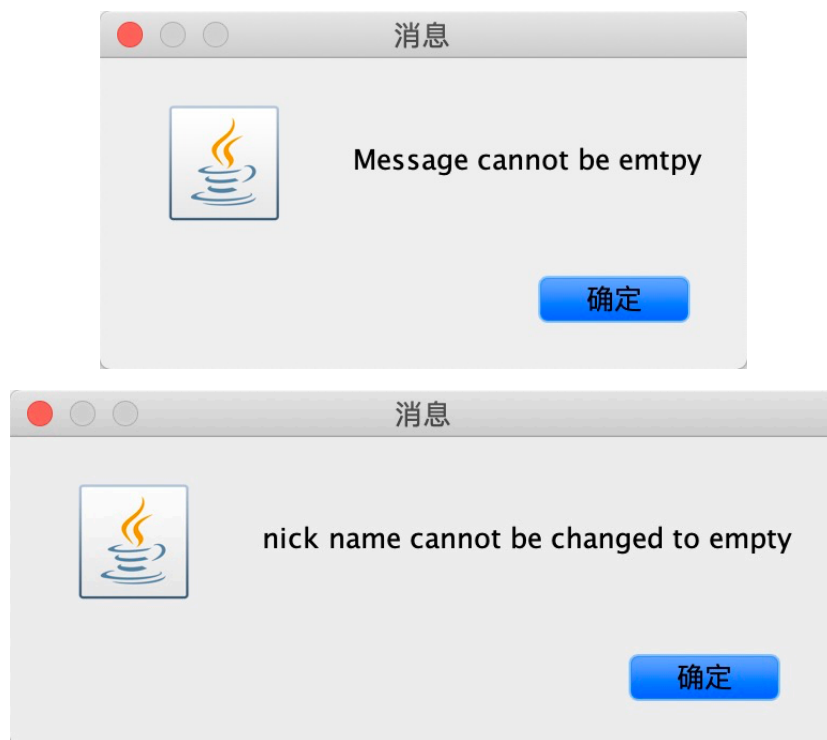
最终成功实现消息转发。

也可以通过Mac的nc命令连接服务器，并且发送数据。同样可以实现转发给所有客户端。



对于一些特殊情况

当输入为空的时候，无法发送，并在对话框中输出"Message cannot be empty"。如果要将昵称改为空，也是不允许的。不过通过命令行nc命令可以为空昵称。默认昵称为Client。



代码实现

服务端代码

采用bio实现。之所以没有采用nio，是因为老师课堂上最后没讲完，但后面也不讲了，自己学习成本比较高。在本实验完成后自己会重新将代码晚上，增加数据库功能以及其他的多功能聊天室。不过本代码满足了所有的基本要求，因此也已经足够了。

```
public class Server {
    private ArrayList<PrintWriter> clientPrintWriters = new
ArrayList<PrintWriter>(); //客户端输出流 用于转发所有数据
    private int PORTNUM = 8888; //端口号 默认为8888
    private int count = 0;
    public static void main(String[] args) {
        Server theServer = new Server();
        theServer.serverStart(); //开启服务器
    }
    public void serverStart() {
```

```

        System.out.println("input server port you are going to use(0 for
default port --- 8888):"); //输入端口号
        Scanner in = new Scanner(System.in);
        PORTNUM = in.nextInt(); //读入端口号
        while(PORTNUM <= 1024) {
            if(PORTNUM == 0) {
                PORTNUM = 8888;
                break;
            }
            System.out.println("well known ports are not allowed, please try
again"); //前面1024个端口基本上都被占用 不允许使用
            PORTNUM = in.nextInt();
        }
        System.out.println("server port is: " + PORTNUM + ".\nStarting
server");
        try {
            Thread serverThread = new Thread(new serverHandler()); //启动server
线程-用于接收请求 并创建新线程处理客户端请求
            serverThread.start();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        in.close();
    }

    // 等待客户端连接的服务端线程
    private class serverHandler implements Runnable {
        public void run() {
            ServerSocket serverSocket;
            try {
                serverSocket = new ServerSocket(PORTNUM);
                //持续监听 分发线程
                while(true) {
                    Socket clientSocket = serverSocket.accept(); // 阻塞监听连
接

                    count++;
                    System.out.println("one client is connected, this is the "
+ count + "th client");
                    PrintWriter writer = new
PrintWriter(clientSocket.getOutputStream());
                    clientPrintWriters.add(writer);
                    //启动客户端处理线程
                    Thread t = new Thread(new ClientHandler(clientSocket));
                    t.start();
                }
            } catch ( IOException e) {
                e.printStackTrace();
            }
        }
    }
}

```

```

    }
    // 客户端处理线程
    private class ClientHandler implements Runnable {
        BufferedReader bufferedReader;
        Socket theSocket;
        public ClientHandler(Socket socket) {
            theSocket = socket;
            try {
                bufferedReader = new BufferedReader(new
InputStreamReader(theSocket.getInputStream())); // 输入流
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        public void run() {
            String inputMessage;
            try {
                while ((inputMessage = bufferedReader.readLine()) != null) {
                    sendToEveryone(inputMessage); // 每一次读入消息后-调用外
部类的sendToEveryone方法-将所有数据转发到所有客户端
                }
            } catch (Exception ex) {
                ex.printStackTrace();
            }
        }
    }
    private void sendToEveryone(String message) {
        for(PrintWriter pw : clientPrintWriters) {
            pw.println(message);
            pw.flush();
        }
    }
}

```

总体思想如下

1. 读入端口，丢给serverHandler，建立ServerSocket
2. serverHandler持续监听连接请求，创建对应的ClientHandler线程并启动。
3. 将每一个client的socket输出流都加入到PrintWriter的容器中，用于全局广播
4. 在ClientHandle线程中读入client的消息，并通过内部类使用外部类的方法的方式调用外部方法 sendToEveryone()实现消息广播。

这里仅仅采用了老师上课讲的一种实现方式。课堂上对这个设计提及了多种方式。

下面针对四个步骤分析核心代码

- 1.读入端口，丢给serverHandler，建立ServerSocket

```

PORTNUM = in.nextInt();    //读入端口号
while(PORTNUM <= 1024) {
    if(PORTNUM == 0) {
        PORTNUM = 8888;
        break;
    }
}

```

while循环读入端口号，0表示默认8888；

```

Thread serverThread = new Thread(new serverHandler()); //启动server线程-用于接收请求 并创建新线程处理客户端请求
serverThread.start();

```

启动server线程。

2. serverHandler持续监听连接请求，创建对应的ClientHandler线程并启动。
3. 将每一个client的socket输出流都加入到PrintWriter的容器中，用于全局广播

```

while(true) {
    Socket clientSocket = serverSocket.accept();    // 阻塞监听连接
    count++;
    System.out.println("one client is connected, this is the " + count + "th client");
    PrintWriter writer = new PrintWriter(clientSocket.getOutputStream());
    clientPrintWriters.add(writer);
    //启动客户端处理线程
    Thread t = new Thread(new ClientHandler(clientSocket));
    t.start();
}

```

建立连接后输出相关信息。同时clientPrintWriters(是ArrayList的实例)添加对应的socket输出流(这里过滤为PrintWriter)

4. 在ClientHandle线程中读入client的消息，并通过内部类使用外部类的方法的方式调用外部方法sendToEveryone()实现消息广播

```

public void run() {
    String inputMessage;
    try {
        while ((inputMessage = bufferedReader.readLine()) != null) {
            sendToEveryone(inputMessage);    // 每一次读入消息后-调用外部类的sendToEveryone方法-将所有数据转发到所有客户端
        }
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}

```

```
private void sendToEveryone(String message) {
    for(PrintWriter pw : clientPrintWriters) {
        pw.println(message);
        pw.flush();
    }
}
```

由于已经由所有的客户端的PrintWriter，直接遍历即可输出。总体结构并不复杂。

客户端代码

客户端代码总体思想如下

1. 建立GUI窗口--在MiniCAD中已经有所学习
2. 添加每个按钮的事件响应。
3. 响应的重点为两个，一是建立连接按钮，二是发送消息按钮。对于建立连接按钮，需要在事件响应中根据输入的服务器ip和端口创建socket流，同时处理服务器发送来的消息--进行显示。对于发送消息按钮，直接通过流发送消息即可。

下面对具体的内容展开分析

1. 建立GUI窗口

```
private JFrame clientFrame = new JFrame(); //客户端主窗口
private JLabel labelIP = new JLabel("Server IP"); //服务器IP标识
private JLabel labelPort = new JLabel("Server Port"); //服务器端口标识
private JLabel labelMessage = new JLabel("Message"); //消息标识
private JLabel labelName = new JLabel("NickName"); //客户端名字标识
private JTextField textIP = new JTextField(10); //服务器IP输入框
private JTextField textPort = new JTextField(10); //服务器端口输入框
private JTextField textName = new JTextField(nickname, 30); //客户端名字输入框
private JTextField textMessage = new JTextField(30); //客户端消息输入框
private JButton buttonConnection = new JButton("Connect"); //确认ip和端口并建立连接按钮
private JButton buttonName = new JButton("Confirm"); //确认名字按钮
private JButton buttonMessage = new JButton("Confirm"); //确认消息按钮
private JPanel panelConnection = new JPanel(); //连接确认窗口容器
private JPanel panelSouth = new JPanel(); //布局南方窗口容器
private JPanel panelName = new JPanel(); //名字确认窗口容器
private JPanel panelMessage = new JPanel(); //消息发送窗口容器
private JTextArea messageArea = new JTextArea(); //客户端消息显示
private JScrollPane scroller = new JScrollPane(messageArea);
```

主要为如上变量。其布局可以很容易的根据最终的显示效果得到。

2. 添加每个按钮的事件响应

```
// 设置客户端名字
buttonName.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
```



```

String name = textName.getText();
if(name.equals(""))
    JOptionPane.showMessageDialog(clientFrame, "nick name cannot be changed to empty");
else nickname = name;
}
});
textName.addFocusListener(new FocusListener() {
    public void focusGained(FocusEvent e) {
    }
    public void focusLost(FocusEvent e) {
        String aText = GUI.textName.getText();
        if (!aText.equals("")) {
            nickname = aText;
        }
    }
});

```

第一个为名字设置。第二个是聚焦。剩余两个Listener在步骤3阐述。这里的基本实现较为简单，直接设置对应内容即可。需要了解的就是showMessageDialog这个API的使用

- 响应的重点为两个，一是建立连接按钮，二是发送消息按钮。对于建立连接按钮，需要在事件响应中根据输入的服务器ip和端口创建socket流，同时处理服务器发送来的消息--进行显示。对于发送消息按钮，直接通过流发送消息即可。

代码如下所示。对于客户端而言，最为关键的就是与服务器的连接。基本的实现就是根据读入的ip和端口号创建Socket对象。获取输入输出流，由于作业要求是文本，因此直接使用reader和writer。在连接成功后在消息面板输出"Server Conneced"并启动消息处理线程。

对于从客户端发送消息到服务器，由于已经有了writer，直接将textField中的字符输出即可。

对于接受服务器消息的线程，通过reader获取数据，并将结果转发。

```

buttonConnection.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String serverIP = textIP.getText(); //获取输入服务器ip
        String serverPort = textPort.getText(); //获取输入服务器端口
        if (serverIP.equals("") || serverPort.equals("")) {
            //如果输入为空-要求重新输入
            JOptionPane.showMessageDialog(clientFrame, "Please input appropriate IP and PORT!");
        }
        else {
            try {
                Socket clientSocket = new Socket(serverIP,
                    Integer.parseInt(serverPort)); //建立Socket连接
                reader = new BufferedReader(new
                    InputStreamReader(clientSocket.getInputStream())); //reader
                writer = new PrintWriter(clientSocket.getOutputStream()); //writer
                messageArea.append("Server Connected\n");
            }
            catch (Exception ex) {
                JOptionPane.showMessageDialog(clientFrame, "Error: " + ex.getMessage());
            }
        }
    }
});

```

```

        Thread readerThread = new Thread(new messageRecv()); //消息处理线程
        readerThread.start();
    } catch (Exception ex) {
        JOptionPane.showMessageDialog(clientFrame, "Cannot connect to
server!\nPlease verify your input");
    }
}
}
});

```

// 发送客户端消息到服务器

```

buttonMessage.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        String message = textMessage.getText();
        if (message.equals("")) {
            JOptionPane.showMessageDialog(clientFrame, "Message cannot be empty");
        }
        else {
            writer.println(nickname + ": " + message);
            writer.flush();
            textMessage.setText("");
        }
    }
});

```

// 接收服务器消息的线程

```

private class messageRecv implements Runnable {
    public void run() {
        String message;
        try {
            while ((message = reader.readLine()) != null) {
                GUI.messageArea.append(message + "\n");
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

