

Socket编程实践——网络聊天室实验报告

功能实现情况

功能	是否已完成
基础——服务端（创建聊天室、接受并广播消息...）	是
基础——客户端（连入聊天室、接受并展示消息...）	是
扩展——异常处理（粘包、异常断线的检测和处理）	是
扩展——并发调优与压力测试	是
扩展——文件传输	是
扩展——语音聊天室	是

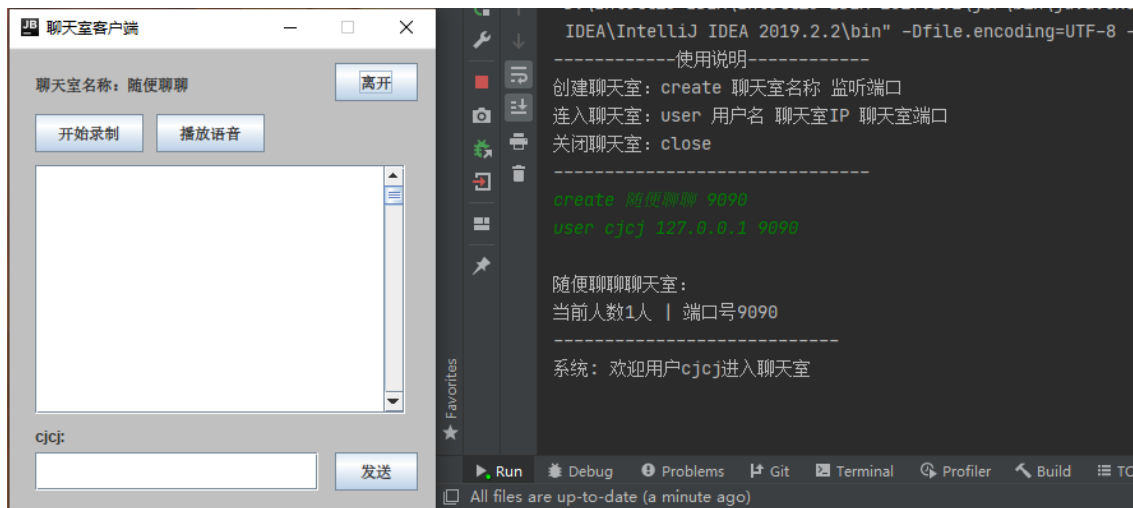
文件结构

```
|-- src
|   |-- Test.java // 可运行测试文件
|   |-- client
|   |   |-- AudioRecord.java // 录制声卡，保存PCM粗流音频
|   |   |-- Pcm2Wav.java // 将PCM粗流音频转为可播放的WAV格式
|   |   |-- AudioPlayer.java // 播放WAV格式音频
|   |   |-- Client.java // 客户端V1
|   |   |-- ClientV2.java // 客户端V2，使用nio进行并发测试
|   |   |-- ClientUI.java // 客户端swing图形界面
|   |-- example // 服务端文件存储根目录
|   |-- Server
|   |   |-- ChatRoom.java // 聊天室
|   |   |-- Server.java // 服务端V1
|   |   |-- MsgTask.java // 监控V1中每个socket连接的输入流
|   |   |-- ServerV2.java // 服务端V2，使用nio进行并发测试
|
4 directories, 11 files
```

基本功能展示

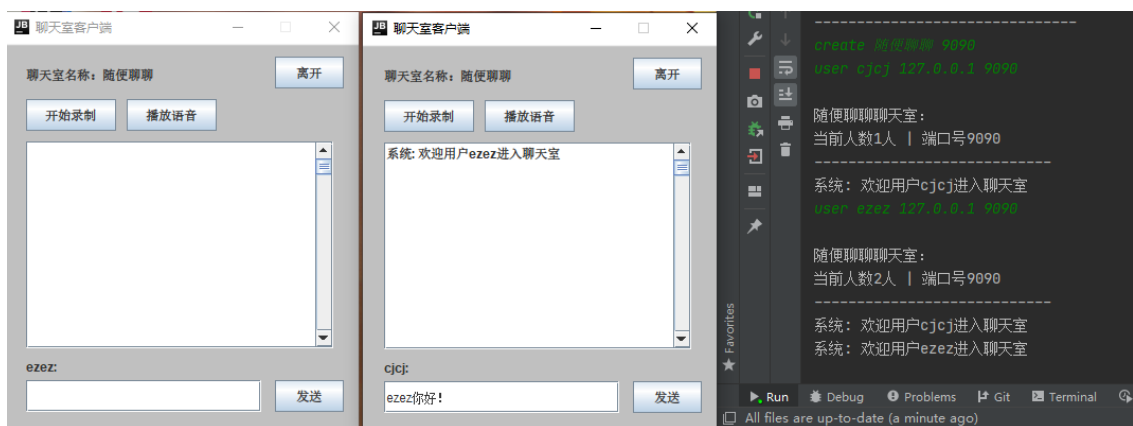
- 创建聊天室，连入第一位用户：

注：左侧的窗口是创建用户（user指令）后出现的



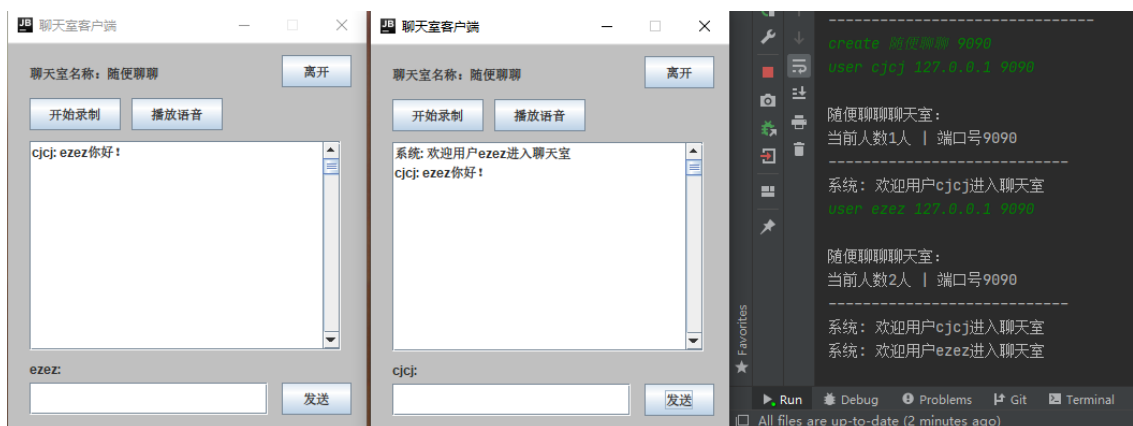
- 连入第二位用户:

注: 此时 cjcj 收到了系统消息, 准备输入文字打招呼



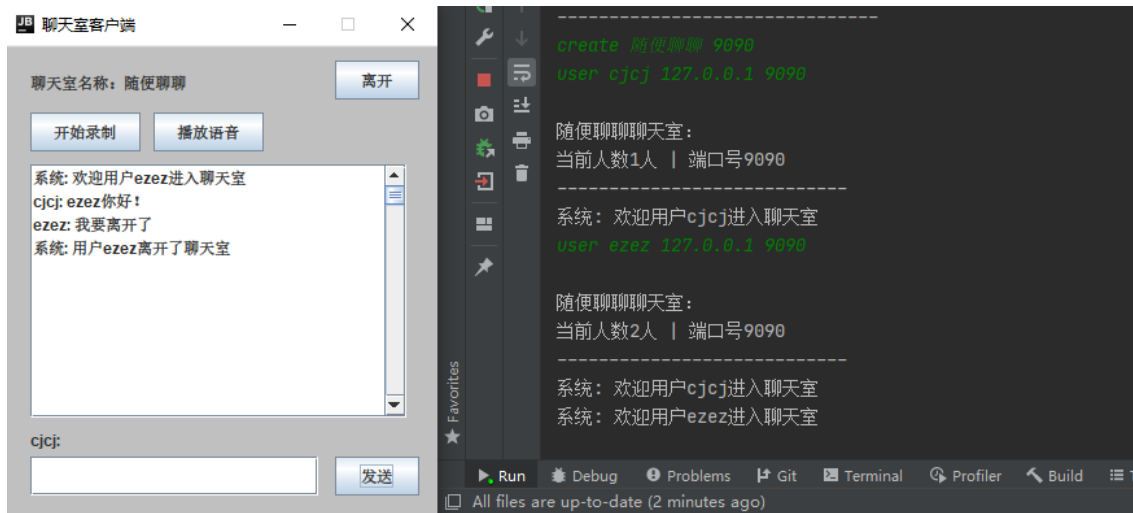
- 第一位用户发送文字消息:

注: cjcj 点击发送按钮, 发送了文字消息, 此时两位用户均能看到文字



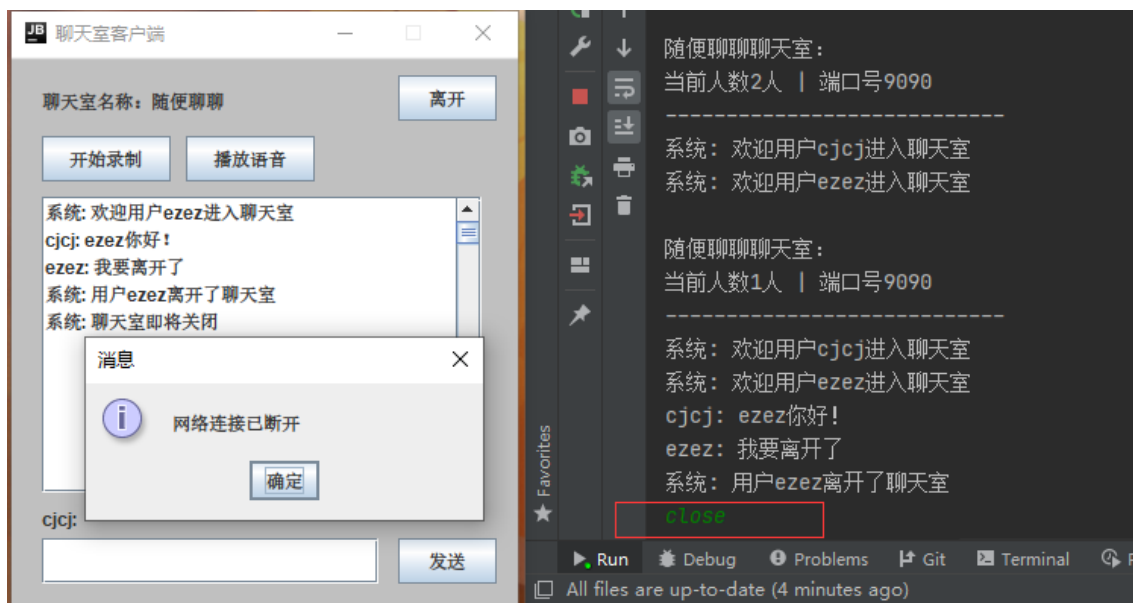
- 第二位用户离开聊天室:

注: ezez 点击离开按钮, 离开了聊天室, 此时 cjcj 可以看到来自系统的消息



- 关闭聊天室：

注：聊天室关闭后，客户端检测到socket关闭，弹出提示框



扩展功能

扩展一：异常处理

- 粘包检测和处理

粘包的主要原因：发送方每次写入数据 < 套接字（Socket）缓冲区大小；接收方读取套接字（Socket）缓冲区数据不够及时。

粘包问题演示：

```
// 服务端
static class ServSocket {
    // 缓冲区大小
    private static final int BYTE_LENGTH = 20;

    public static void create() throws IOException {
        ServerSocket serverSocket = new ServerSocket(9090); // 监听本地9090端口
        // 开新线程接受客户端连接，并持续读取数据
        new Thread(() -> {
            try {
                Socket clientSocket = serverSocket.accept();
```

```

        InputStream inputStream = clientSocket.getInputStream();
        while (true) {
            byte[] bytes = new byte[BYTE_LENGTH];
            int count = inputStream.read(bytes, 0, BYTE_LENGTH);
            if (count > 0) System.out.println("接收到客户端的信息是:" + new
String(bytes));
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}

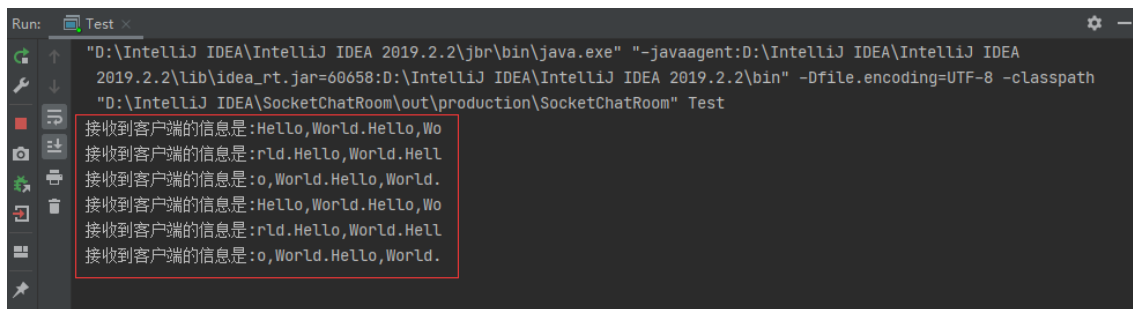
}).start();
}

}

// 客户端
static class ClientSocket {
    public static void connect() throws IOException {
        Socket socket = new Socket("127.0.0.1", 9090); // 连接本地9090端口
        final String message = "Hello,world."; // 待写入的数据
        OutputStream outputStream = socket.getOutputStream();
        for (int i = 0; i < 10; i++) // 连续发送十次数据
            outputStream.write(message.getBytes());
    }
}
}

```

执行结果：



```

Run: Test x
"D:\IntelliJ IDEA\IntelliJ IDEA 2019.2.2\jbr\bin\java.exe" "-javaagent:D:\IntelliJ IDEA\IntelliJ IDEA
2019.2.2\lib\idea_rt.jar=60658:D:\IntelliJ IDEA\IntelliJ IDEA 2019.2.2\bin" -Dfile.encoding=UTF-8 -classpath
"D:\IntelliJ IDEA\SocketChatRoom\out\production\SocketChatRoom" Test
接收到客户端的信息是:Hello,World.Hello,Wo
接收到客户端的信息是:rld.Hello,World.Hell
接收到客户端的信息是:o,World.Hello,World.
接收到客户端的信息是:Hello,World.Hello,Wo
接收到客户端的信息是:rld.Hello,World.Hell
接收到客户端的信息是:o,World.Hello,World.

```

改进措施：

1. 客户端关闭缓冲区，及时发送数据

```
if (!socket.getTcpNoDelay()) socket.setTcpNoDelay(true);
```

2. 服务器端使用 `BufferedReader` 读取数据，通过使用带缓冲区的输入字符流和输出字符流，在写入的时候加上 `\n` 来结尾，读取的时候使用 `readLine()` 按行来读取数据，这样就知道流的边界了

```

Socket socket = server.accept();
BufferedReader in = new BufferedReader(...);
String username = in.readLine();

```

- 异常断线检测和处理

1. 每个客户端单独开一个线程，定时发送心跳包，并检查 `tcp` 连接是否断开

```

new Thread(() -> {
    // 定义特殊字符串，服务端接收后不做处理
    final String heartbeat = "[usage for heartbeat packet]";
    while (true) {
        try {

```

```

        Thread.sleep(15 * 1000); // 每15s发送一次心跳
        out.write(heartbeat + "\r\n");
        out.flush();
        try {
            socket.sendUrgentData(0xFF); // 检查tcp连接是否有效
        } catch (IOException ex) {
            UI.showDialog("网络连接已断开");
            break;
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
}).start();

```

2. 服务端每次接受心跳包，刷新 `timestamp`，并将 `timeout` 清零。通过定时比较当前时间与 `timestamp` 的时间差，进行心跳包的检测，若相差大于20秒，则记一次超时

```

private long timestamp; // 上次接受数据的时间，用于心跳包检测
private int timeout; // 超时次数，到达3次则断开该socket连接

new Thread() -> {
    while (connected) {
        try {
            Thread.sleep(20 * 1000);
            if (new Date().getTime() - timestamp > 20 * 1000) {
                if (timeout == 2) disconnect();
                else timeout++;
            }
        } catch (InterruptedException | IOException e) {
            if (!(e instanceof SocketException)) e.printStackTrace();
        }
    }
}
}).start();

```

扩展二：并发调优与压力测试

使用传统 `SocketServer` + `Socket` + 非阻塞IO的方式：

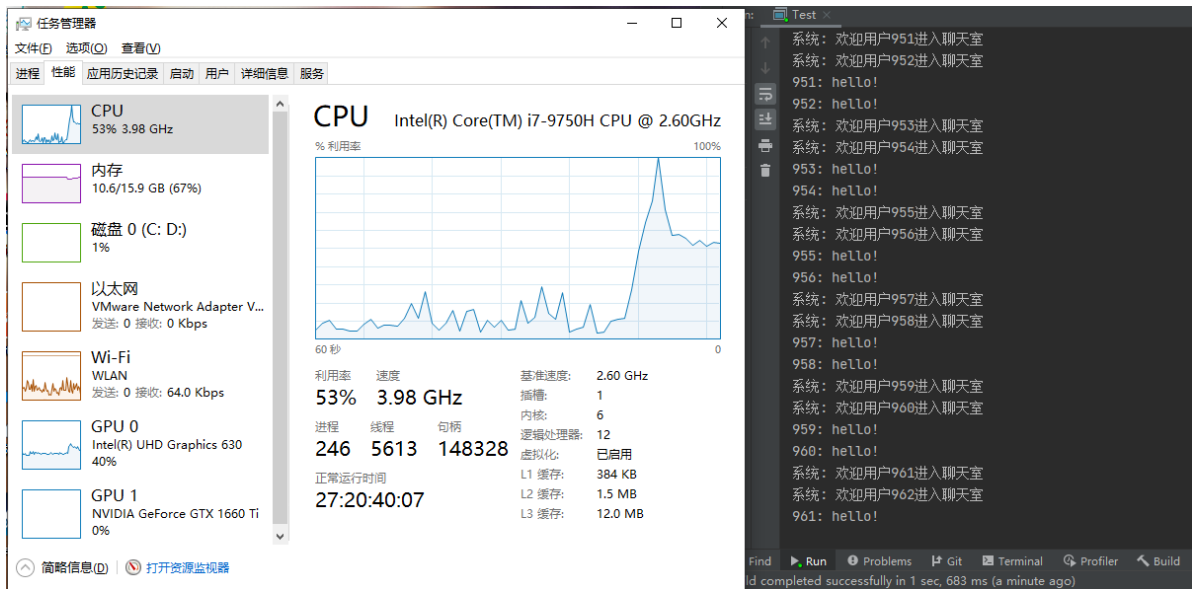
```

// 建立1000个和服务端间的socket连接，并记录总用时
public static void main(String[] args) {
    ChatRoom.create("测试", 9090);
    long start = new Date().getTime();
    for (int i = 0; i < 1000; i++) {
        ClientUI clientUI = new ClientUI();
        clientUI.createClient(String.valueOf(i), "127.0.0.1", 9090);
        clientUI.client.speak("hello!");
    }
    long end = new Date().getTime();
    System.out.println("-----");
    System.out.println("总处理时间: " + (end - start) + "ms");
}

```

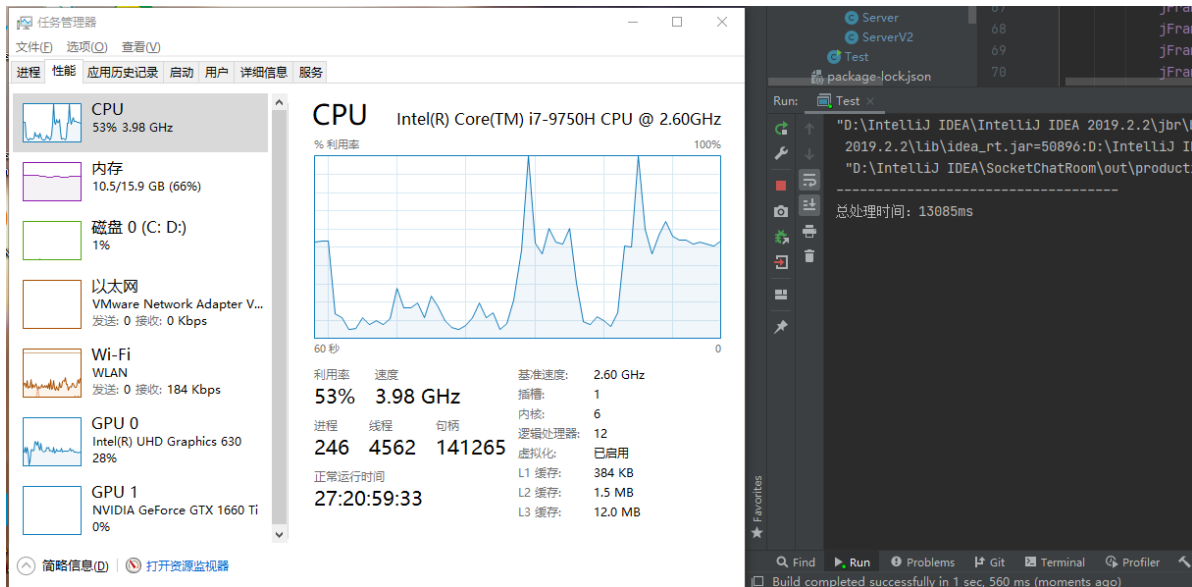
最高峰状态下的系统消耗：

经多次测试，在1000个客户端连入聊天室并分别发送消息的情况下，CPU占用提高了45%左右，内存占用了600M，增加了2100个线程使用（非阻塞IO、心跳包检测等），上述测试代码总用时约为34000ms。



使用同步非阻塞IO模型Nio的方式：（代码见ServerV2.java, ClientV2.java）

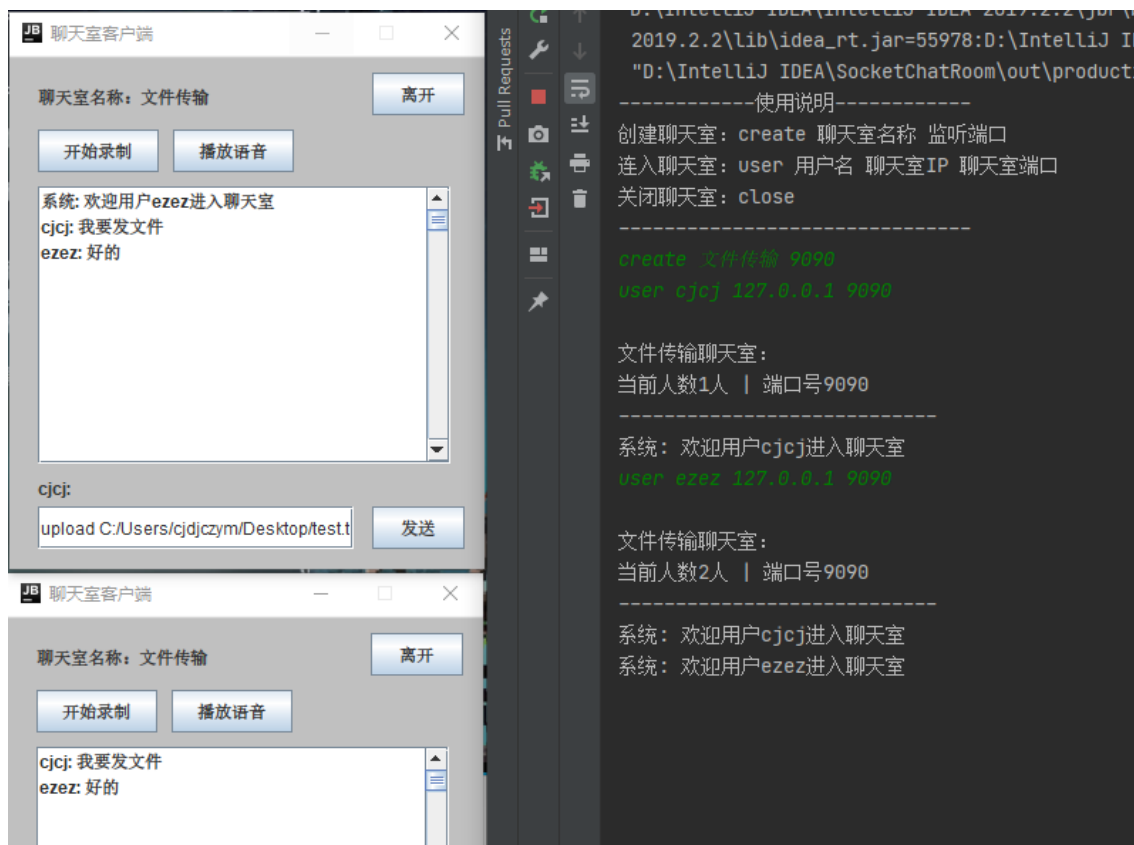
经多次测试，在1000个客户端连入聊天室并分别发送消息的情况下，CPU占用提高了45%左右，内存占用了500M，增加了1000个线程使用，上述测试代码总用时约为**13000ms**。



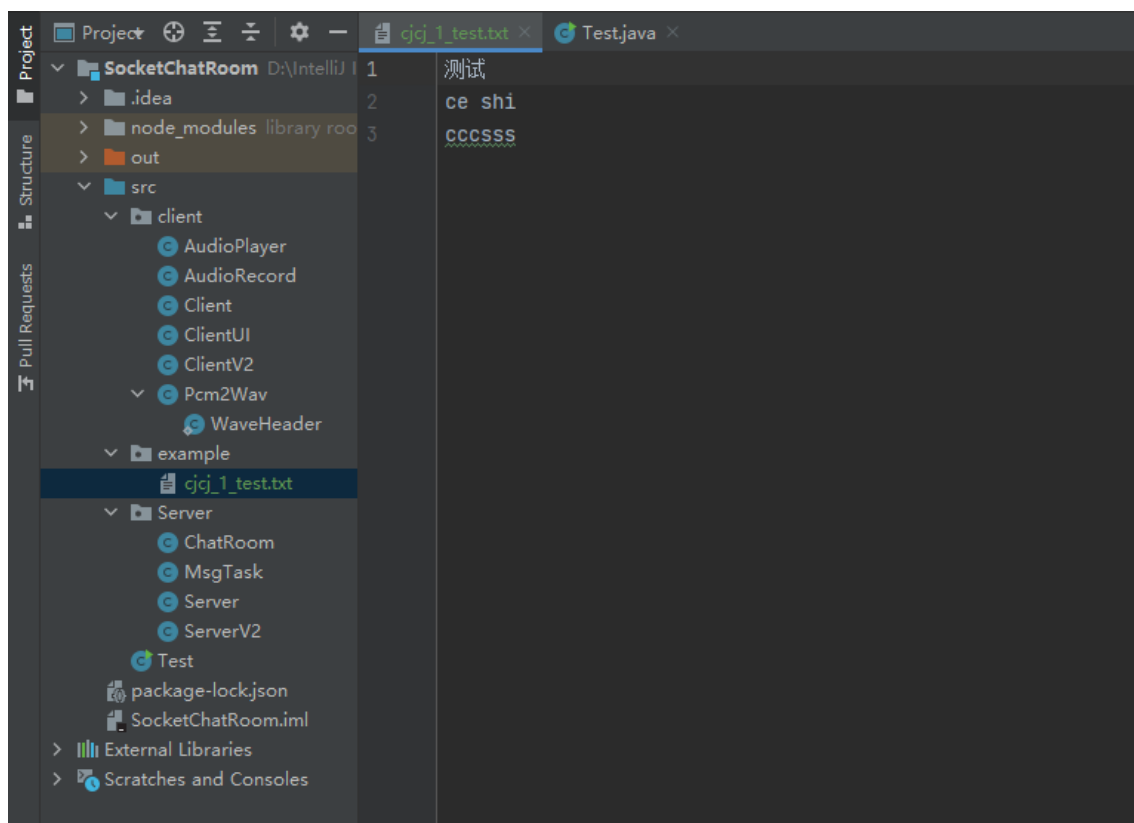
综上所述，采用Nio等非阻塞IO模型可以很好的节省线程、内存的使用，同时加快处理速度；但是Nio等模型在传输文件等场景时也有着其缺陷。因此我们要根据实际需求来进行选择。

扩展三：文件传输

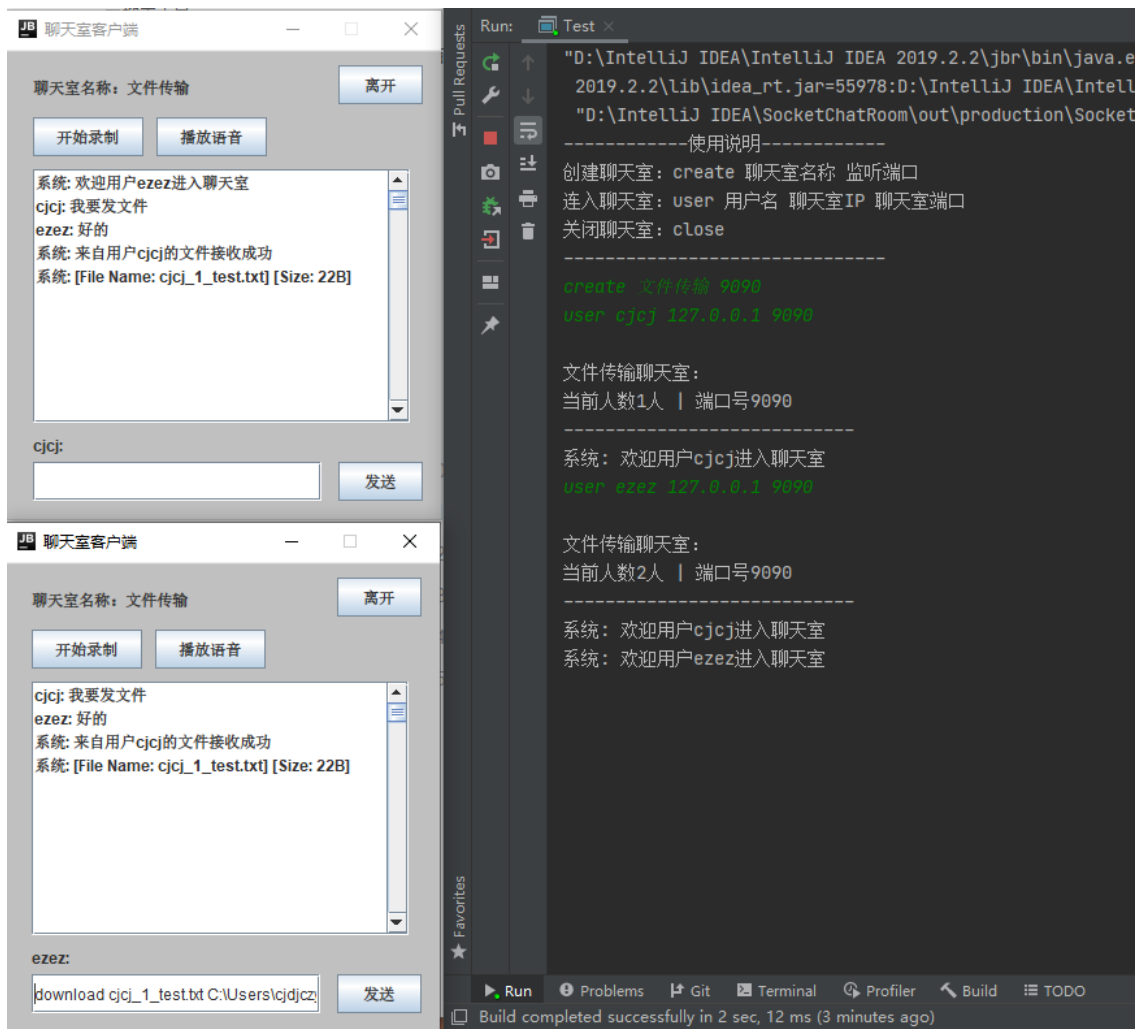
- 用户 `cjcj` 和 `ezez` 进入聊天室后，`cjcj` 使用 `upload` 指令向服务器上传本地（桌面）文件



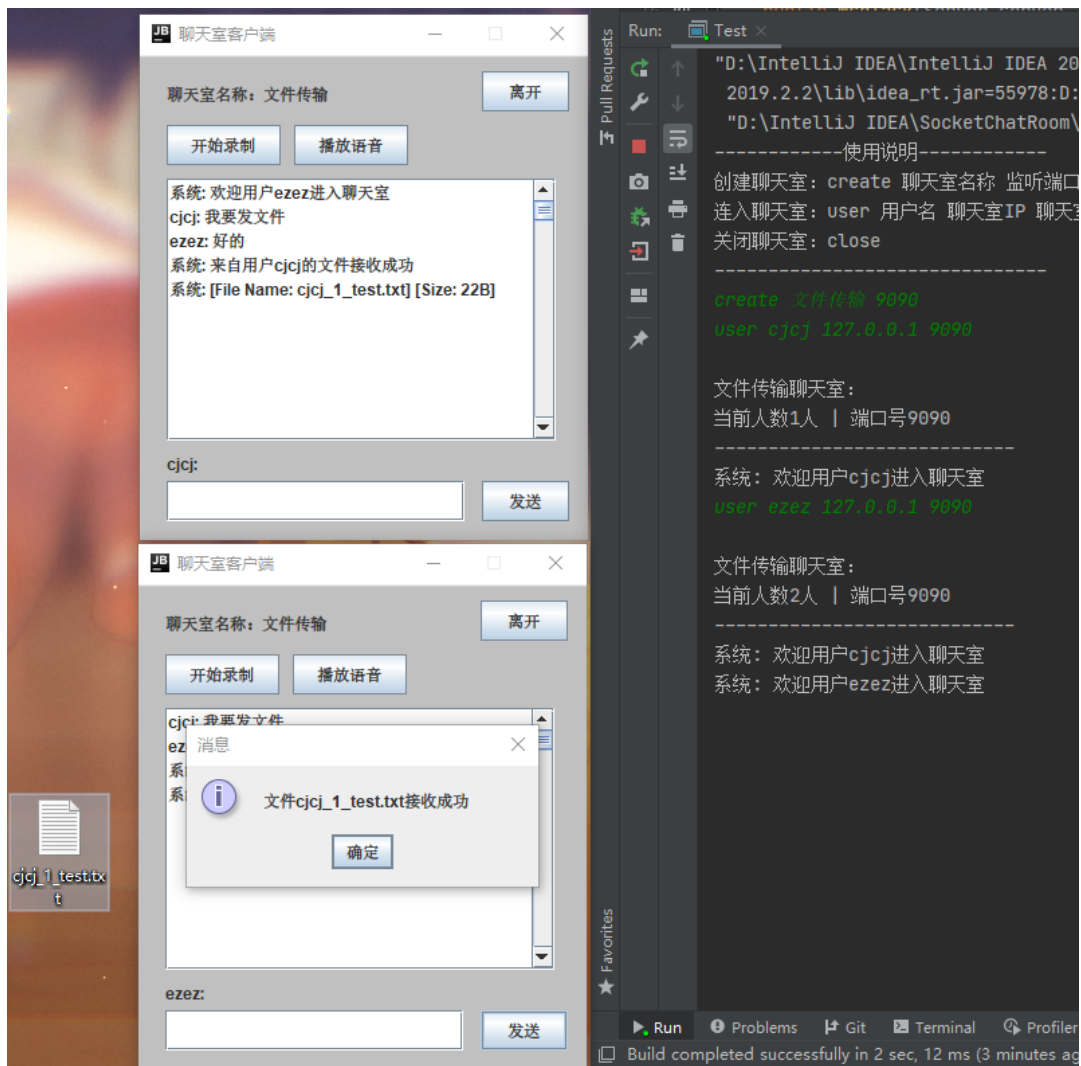
- 此时文件通过 `socket` 传输至服务器文件存储目录 `example` 下 (已被重命名处理, 防止文件名称冲突)



- 聊天室中的用户都可以看到来自服务器的消息, 提示 `cjcj` 的文件上传成功, 此时 `ezez` 通过 `download + 文件名` 指令下载该文件至本地

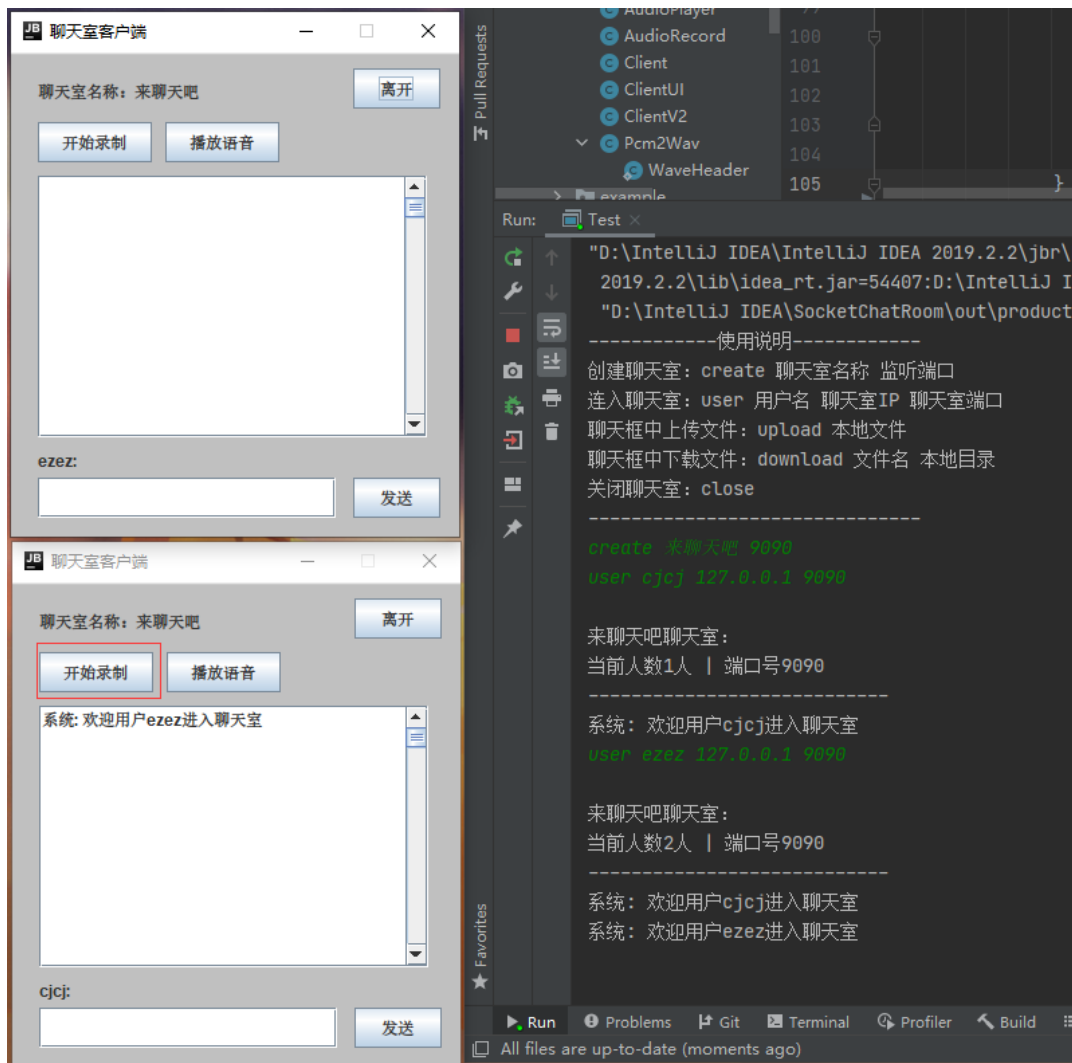


- ezez 下载该文件成功

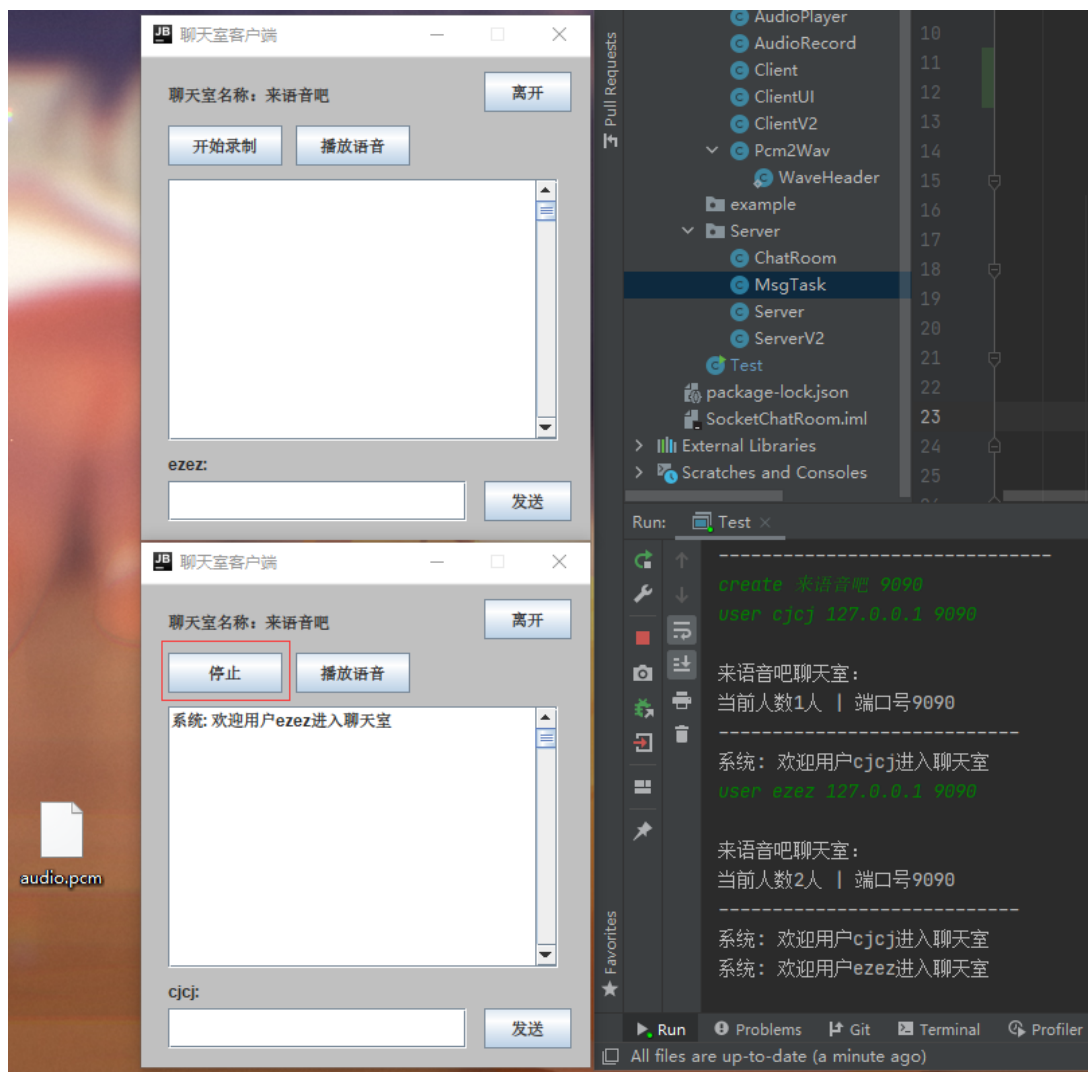


扩展四：语音聊天室

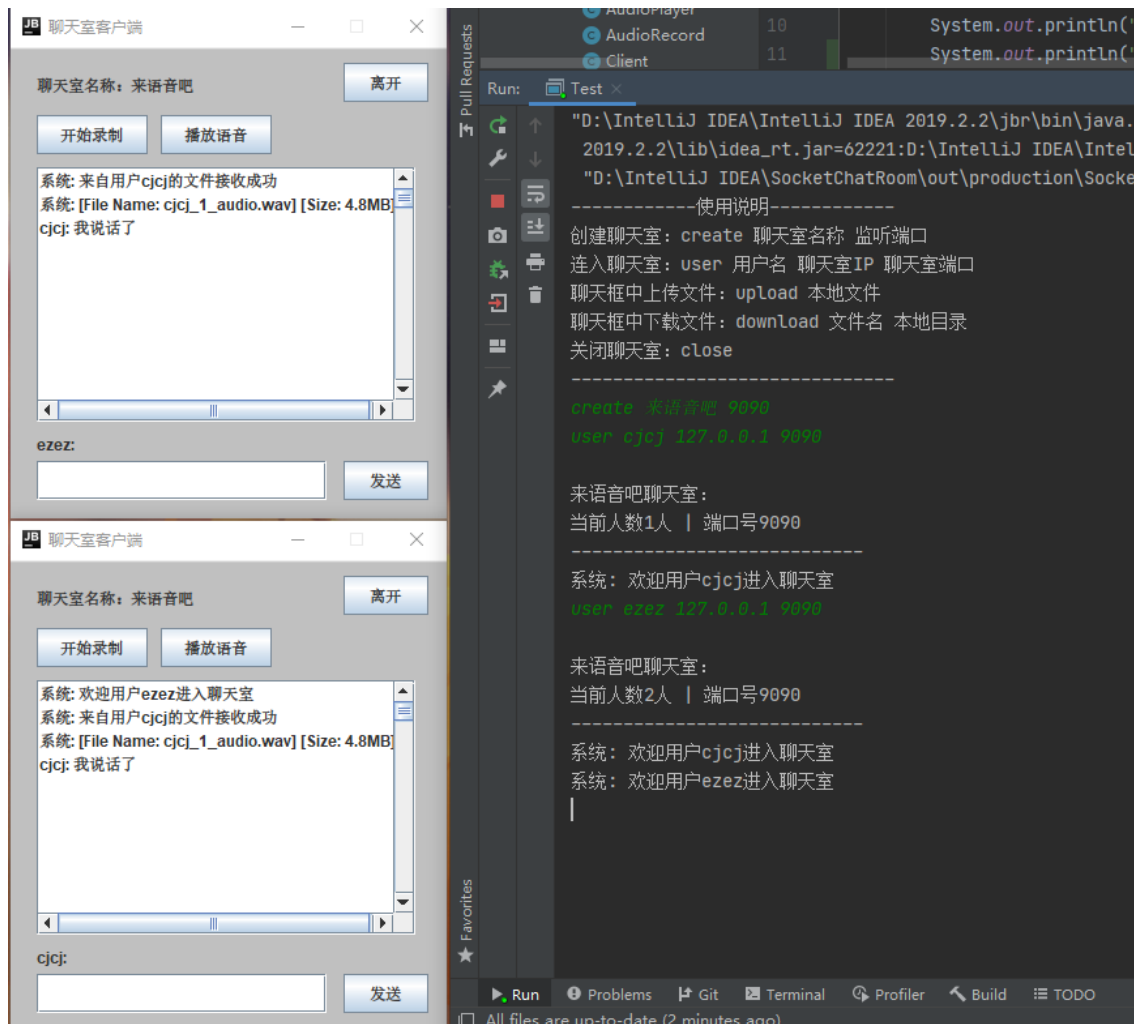
- 用户 `cjcj` 和 `ezez` 进入聊天室后，`cjcj` 点击 `开始录制` 按钮说话



- 点击按钮后, java程序会在本地新建PCM粗流文件, 并会持续从声卡向其录入数据, 直至点击 停止 键为止



- 录音停止后，程序会将PCM格式文件转为可播放的WAV格式，向服务器上传后再删除本地文件缓存



- 聊天室中的其他用户可以点击 **播放语音** 按钮播放上一条语音消息，语音文件会先从服务端下载至本地，待播放结束后再删除掉

