# 第5讲 多用户服务器程序设计

**教学与实践目的：**学会服务器支持多用户并发访问的程序设计技术。

多用户服务器是指服务器能同时支持多个用户并发访问服务器所提供的服务资源,如聊天服务、文件传输等。

第二讲的TCPServer是单用户版本，每次只能和一个用户对话。（请仔细阅读TCPServer.java程序，了解其中原理，找出关键语句），只有前一个用户退出后，后面的用户才能完成服务器连接。

## 一、单用户服务器程序演示

单用户版本TCPServer.java部分代码：

public class TCPServer {  
 private int port = 8008; *//服务器监听端口* private ServerSocket serverSocket; *//定义服务器套接字* public TCPServer() throws IOException {  
 serverSocket = new ServerSocket(port);  
 System.*out*.println("服务器启动监听在 " + port + " 端口");  
 }  
 *//省略封装的返回输入输出流的相关代码……*  
   
 *//单客户版本，主服务方法，每一次只能与一个客户建立通信连接* public void Service() {  
 while (true) {  
 Socket socket = null;  
 try {  
 *//此处程序阻塞，监听并等待客户发起连接，有连接请求就生成一个套接字。* socket = serverSocket.accept();  
 *//本地服务器控制台显示客户端连接的用户信息* System.*out*.println("New connection accepted： " + socket.getInetAddress());  
 *//省略输入输出流获取的代码，以下pw是字符输出流，br是字符输入流*

pw.println("From 服务器：欢迎使用本服务！");

String msg = null;  
 *//此处程序阻塞，每次从输入流中读入一行字符串* **while ((msg = br.readLine()) != null) {  
 *//如果客户发送的消息为"bye"，就结束通信* if (msg.equals("bye")) {  
 *//向输出流中输出一行字符串,远程客户端可以读取该字符串* pw.println("From服务器：服务器断开连接，结束服务！");**

**System.*out*.println("客户端离开");  
 break; *//结束循环，结束通信*  
 }  
 *//向输出流中输出一行字符串,远程客户端可以读取该字符串* pw.println("From服务器：" + msg);  
 }**  
 } catch (IOException e) {  
 e.printStackTrace();  
 } finally {  
 try {  
 if(socket != null)  
 socket.close(); *//关闭socket连接及相关的输入输出流* } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
  
 public static void main(String[] args) throws IOException{  
 new TCPServer().Service();  
 }  
}

27-37行是其中的核心代码，用于提供和客户端的对话功能。

**演示过程：**

（1）启动你的TCPServer.java程序(程序详见第二讲的附录代码)；

（2）启动第三讲的TCPClientThreadFX.java程序，完成一次对话，并保持连接；

（3）再启动一份TCPClientThreadFX.java程序实例（即同时运行两个相同的程序），并完成连接，尝试对话，发现没有服务器的返回信息（服务器和客户端各阻塞在哪条语句？）

**注意：**现在的新版本idea默认不允许同时运行多个相同的程序，需要修改默认配置才能多实例运行：

首先进入右上角的edit Configurations，如图5.1所示：

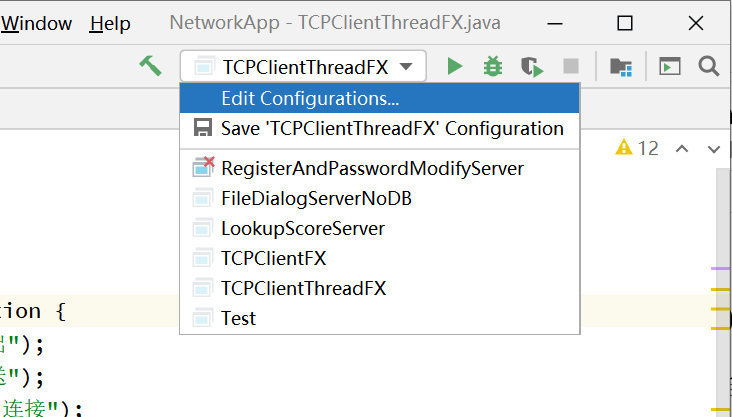


图5.1 修改运行配置

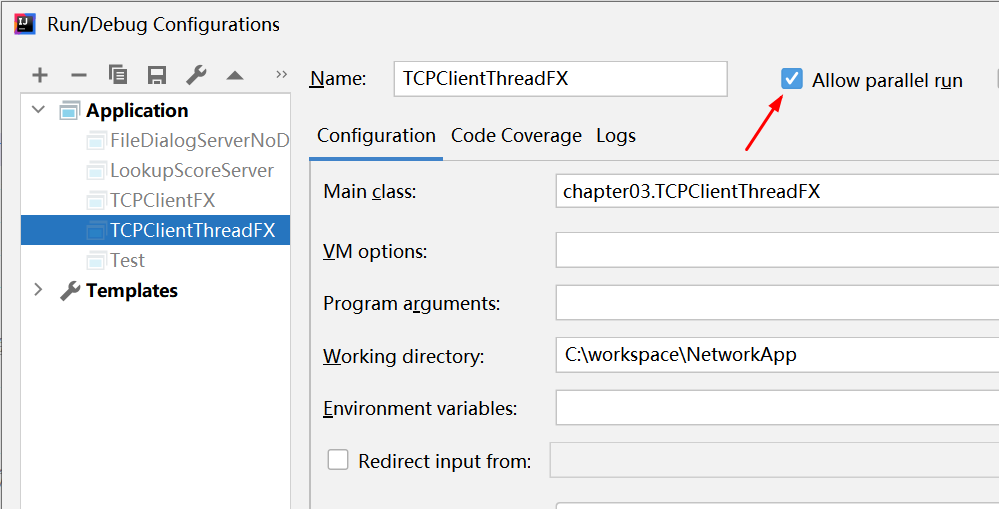
然后如图5.2所示，将此程序的“Allow paralle run”复选框勾选。

图5.2 改为允许多实例运行

（4）退出第一次启动的TCPClientThreadFX.java程序，发现第二次启动的客户端有返回信息了，说明在一个客户退出后，第二个客户才能和服务器进行对话。

**原因：**服务器的主进程一次只能处理一个客户，其它已连接的客户等候在监听队列中。

## 二、多用户服务器程序设计

单用户版本的TCPServer.java程序能同时支持多个用户并发连接（TCP三次握手），但不能同时服务多用户对话。

### 1. 线程池解决多用户对话问题

解决思路就是用多线程。服务器可能面临很多客户的并发连接，这种情况的方案一般是：主线程只负责监听客户请求和接受连接请求，用一个线程专门负责和一个客户对话，即一个客户请求成功后，创建一个新线程来专门负责该客户。对于这种多用户的情况，用第三讲的方式new Thread创建线程，频繁创建大量线程需要消耗大量系统资源。对于服务器，一般是使用线程池来管理和复用线程。线程池内部维护了若干个线程，没有任务的时候，这些线程都处于等待状态。如果有新任务，就分配一个空闲线程执行。如果所有线程都处于忙碌状态，新任务要么放入队列等待，要么增加一个新线程进行处理。ExecutorService代表线程池，其创建方式常见的有两种：

ExecutorService executorService = Executors.newFixedThreadPool(n);

ExecutorService executorService = Executors. newCachedThreadPool( );

创建后，就可以使用executorService.execute方法来取出一个线程执行，该方法的参数就是Runnable接口类型。我们可以将和客户对话部分的代码抽取到一个Runnable的实现类[Handler](#_Handler类源代码)的run方法中，然后丢给线程池去执行。

那么应该使用哪种线程池呢，使用第一个固定线程数的线程池，显然不够灵活，第二种方式的线程池会根据任务数量动态调整线程池的大小，我们的练习可以用这个动态调整线程池。但要特别说明：作为小并发使用问题不大，但其在实际生产环境使用并不合适，如果并发量过大，常常会引发OOM错误（OutOfMemoryError）。

**扩展要求：**

如果能提供自行确定最小值和最大值的动态调整的线程池会更满足要求，大家跟踪Executors. newCachedThreadPool( )方法，观察其源代码，会发现非常简单，而且也会明白为什么会出现OOM错误，大家可以尝试将其实现代码拷贝出来稍作修改，封装一个自己版本的myCachedThreadPool方法来使用。

**演示过程：**

（1）新建chapter05程序包，然后将单用户版的TCPserver.java拷贝进来，重命名为TCPThreadServer.java，参考附录中的Handler源代码（Handler是TCPThreadServer内部类），修改为多用户版本，调试通过后启动它；

（2）多次启动你的TCPClientThreadFX.java客户程序，并保持它们同时在线，完成每一个客户的有效对话；

### 2. 在服务程序中支持群组聊天技术

TCPClientThreadFX和TCPThreadServer只实现了客户端和服务器聊天，如何做到群组的聊天？如客户A的聊天信息通过服务器转发到同时在线的所有客户。

可以在服务器端新增记录登陆客户信息的功能，可用在线方式、用户文件方式或数据库方式。本讲的程序用种简单的“在线方式”记录客户套接字，即采用集合来保存用户登陆的套接字信息，来跟踪客户连接。Java有一些常用的集合类型：Map、List和Set。Map是保存Key-Value对，List类似数组，可保存可重复的值，而Set只保存不重复的值，相当于是只保存key，不保存value的Map。

如果是有用户名、学号登录的操作，就可以采用Map类型的集合来存储，例如key：value对应 用户名+学号：套接字。对于我们这个问题的需求，采用Set就够了，用来保存不同用户的socket。因为每一个客户端的IP地址+端口组合不一样，用用户套接字socket作为key来标识一个在线用户是比较方便的选择。

在多线程环境中，对共享资源的读写存在线程并发安全的问题，例如HashMap、HaspSet等都不是线程安全的，可以通过synchronized关键字进行加锁，但还有更方便的方案：可以直接使用Java标准库的java.util.concurrent包提供的线程安全的集合。例如对应HashSet的线程安全Set是CopyOnWriteArraySet。

一些核心步骤如下：

（1）将TCPThreadServer.java程序复制一份并重构、重命名为GroupServer.java；

（2）在GroupServer类中添加核心的群组发送方法sendToAllMembers，用于给所有在线客服发送信息：

private void sendToAllMembers(String msg,String hostAddress) throws IOException{  
 PrintWriter pw;  
 OutputStream out;  
*// Socket tempSocket;* for (Socket tempSocket : *members*) {  
 out = tempSocket.getOutputStream();  
 pw = new PrintWriter(  
 new OutputStreamWriter(out, "utf-8"), true);  
 pw.println(hostAddress + " 发言：" + msg );  
 }  
  
 */\*Iterator<Socket> iterator = members.iterator();  
 while (iterator.hasNext()) {//遍历在线客户Set集合  
 tempSocket = iterator.next(); //取出一个客户的socket  
// String hostAddress = tempSocket.getInetAddress().getHostAddress();  
 out = tempSocket.getOutputStream();  
 pw = new PrintWriter(  
 new OutputStreamWriter(out, "utf-8"), true);  
 pw.println(hostAddress + " 发言：" + msg );  
 }\*/* }

（3）在内部类Handler中用群组发送语句sendToAllMembers(msg, socket.getInetAddress().getHostAddress());

替换pw.println("From 服务器：" + msg);

（4）开启多个客户端，验证群发效果。

## 三、课堂成绩记录步骤

用第三讲的图形客户端TCPClientThreadFX访问172.16.229.253 : 8008， 按照服务器的提示信息完成信息登记任务。

## 四、扩展练习(不计分)：简易聊天室设计

设计聊天服务器ChatServer.java，客户端用学号-姓名的方式登录服务器，实现一对一、一对多私聊及群组广播聊天的功能；用户登录时，需要将用户上线的信息广播给所有在线用户；客户端发送特定命令，服务器要能够返回在线用户列表信息；

程序设计中，要能显示发言者的学号姓名（例如20181111111-程旭元），这种情况可以考虑使用线程安全的HashMap类型（自行搜索应该使用哪个类，这些线程安全的集合类型和普通的集合类型使用方式如出一辙）；

自行考虑如何设计服务端和客户端之间的交互约定（协议），编程中要小心处理好各种逻辑关系。

## 附录

### Handler类源代码

*/\*\*  
 \* Handler设计为内部类，实现Runnable接口，完成对话操作  
 \*/*

class Handler implements Runnable{  
 private Socket socket;  
 public Handler(Socket socket) {  
 this.socket = socket;  
 }  
  
 @Override  
 public void run() {  
 *//本地服务器控制台显示客户端连接的用户信息* System.*out*.println("New connection accepted： " + socket.getInetAddress());  
 try {  
 BufferedReader br = getReader(socket);*//定义字符串输入流* PrintWriter pw = getWriter(socket);*//定义字符串输出流  
  
 //客户端正常连接成功，则发送服务器欢迎信息，然后等待客户发送信息* pw.println("From 服务器：欢迎使用本服务！");  
  
 String msg = null;  
 *//此处程序阻塞，每次从输入流中读入一行字符串* while ((msg = br.readLine()) != null) {  
 *//如果客户发送的消息为"bye"，就结束通信* if (msg.trim().equalsIgnoreCase("bye")) {  
 *//向输出流中输出一行字符串,远程客户端可以读取该字符串* pw.println("From 服务器：服务器已断开连接，结束服务！");  
 System.*out*.println("客户端离开");  
 break;*//跳出循环读取* }  
 *//向输出流中回传字符串,远程客户端可以读取该字符串* pw.println("From 服务器：" + msg);  
  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 }finally {  
 try {  
 if(socket != null)  
 socket.close(); *//关闭socket连接及相关的输入输出流* } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}