

华东师范大学数据科学与工程学院上机项目报告

课程名称：计算机网络与编程

年级：2022 级

上机实践成绩：

指导教师：张召

姓名：李芳

学号：10214602404

上机实践名称： 基于 TCP 的 Socket 编程&基于 TCP 的 Socket 编程优化

上机实践日期：2024.04.12

上机实践编号：7&8

组号：

上机实践时间：

一、基于 TCP 的 Socket 编程题目要求及实现情况

Task1: 使用Scanner修改TCPClient类，达成如下效果，请将实现代码段及运行结果附在实验报告中。

客户端不断读取用户控制台输入的一行英文字母串并将数据发送给服务器
服务器将收到的字符全部转换为大写字母，服务器将修改后的数据发送给客户端
客户端收到修改后的数据，并在其屏幕上显示

储备知识：

一、java.io 包

- 作用：处理输入和输出流的标准 I/O 包。它提供了一组类和接口，用于从不同数据源（如文件、网络连接、内存等）读取数据以及向不同目标（如文件、网络连接、内存等）写入数据。
- 常用类和接口：
 - InputStream 和 OutputStream：所有输入流和输出流的父类，它们是抽象类，定义了读取和写入字节流的基本方法。
 - FileInputStream 和 FileOutputStream：FileInputStream 用于从文件中读取数据，而 FileOutputStream 用于向文件中写入数据。
 - BufferedInputStream 和 BufferedOutputStream：用于提高 I/O 操作性能的缓冲流，它们在读取和写入数据时会在内存中使用缓冲区，减少实际的物理 I/O 次数。
 - Reader 和 Writer：读取和写入字符流的抽象类，它们是所有字符输入流和字符输出流的父类。
 - FileReader 和 FileWriter：FileReader 用于从文件中读取字符数据，而 FileWriter 用于向文件中写入字符数据。
 - BufferedReader 和 BufferedWriter：用于提高字符流性能的缓冲流，它们在读取和写入字符数据时会使用内部缓冲区。

- `DataInputStream` 和 `DataOutputStream`: 用于读取和写入基本数据类型的输入流和输出流, 它们提供了一组方法来读写 Java 的基本数据类型 (如 `int`、`double`、`boolean` 等)。
- `ObjectInputStream` 和 `ObjectOutputStream`: 用于读取和写入对象的输入流和输出流, 它们可以用来序列化和反序列化 Java 对象。

二、*BufferedReader*

- 作用: 属于 `java.io` 包中的一个类, 继承自 `Reader` 类, 并且提供了缓冲功能, 可以一次读取一行或多行文本数据, 并且对数据进行缓冲以提高性能。因此, **常用于高效读取字符流**。
- 构造方法: **`BufferedReader(Reader in);`**
- 读取方法:
 - `int read()`: 读取单个字符, 并返回其 Unicode 编码值。
 - `int read(char[] cbuf, int off, int len)`: 读取字符到指定的字符数组中, 并返回读取的字符数。
 - **`String readLine()`**: 读取一行文本, 并返回一个包含该行内容的字符串, 不包括行终止符。如果已到达流末尾, 则返回 `null`。
- 跳过方法: `long skip(long n)`: 跳过指定数量 `n` 的字符, 返回实际跳过的字符数。
- 关闭方法: `void close()`: 关闭流并释放相关资源。
- 其他方法:
 - `boolean markSupported()`: 检查流是否支持标记功能。
 - `void mark(int readAheadLimit)`: 在当前位置设置一个标记。
 - `void reset()`: 将流的位置重置到最后一次设置的标记位置。
 - `int read(char[] cbuf)`: 从输入流中读取字符到字符数组中。
- 使用流程: **首先创建一个 `BufferedReader` 对象, 然后通过调用其 `readLine()` 方法逐行读取文本数据, 直到读取完整个文本或者返回 `null` 表示已到达流末尾。**

三、*PrintWriter*

- 作用: 属于 `java.io.Writer` 类的子类, 用于将格式化的文本输出到目标位置 (通常是文件或控制台), 主要用于简化文本输出操作。

➤ 常用方法：

- 创建 `PrintWriter` 对象：**`PrintWriter out = new PrintWriter(System.out);`**
- 输出数据：**`out.print/println();`**
- 刷新输出缓冲区：`out.flush();`
- 关闭 `PrintWriter`：**`out.close();`**
- 指定输出文件：`PrintWriter writer = new PrintWriter("output.txt");`
- 指定字符编码：**`PrintWriter writer = new PrintWriter(new OutputStreamWriter(new FileOutputStream("output.txt"), StandardCharsets.UTF_8));`**

四、`java.nio.charset`

- 作用：可以进行字符编码和解码操作，用于处理文本数据在网络传输、文件 I/O 等场景。三、`java.nio.charset` 提供了一种标准化和可扩展的方式来处理各种字符集，使得 Java 程序能够在多种环境和平台上正确地处理文本数据。

➤ 主要类和接口：

- `Charset`: 表示字符集的类，提供了字符集的名称、编解码器等信息。
- `CharsetDecoder`: 用于将字节序列解码为字符序列的类。
- `CharsetEncoder`: 用于将字符序列编码为字节序列的类。
- **`StandardCharsets`**: 一个包含了 Java 虚拟机本地字符集支持的标准字符集的类。
- `Charsets`: 一个早期版本的 `StandardCharsets`，包含了一组静态常量，代表了常用的字符集。

五、`Java.nio.charset.standardCharsets`

- 提供了对 Java 虚拟机的本地字符集支持的实现。这个类包含了一组静态常量，代表了 Java 虚拟机的本地字符集支持的标准字符集。

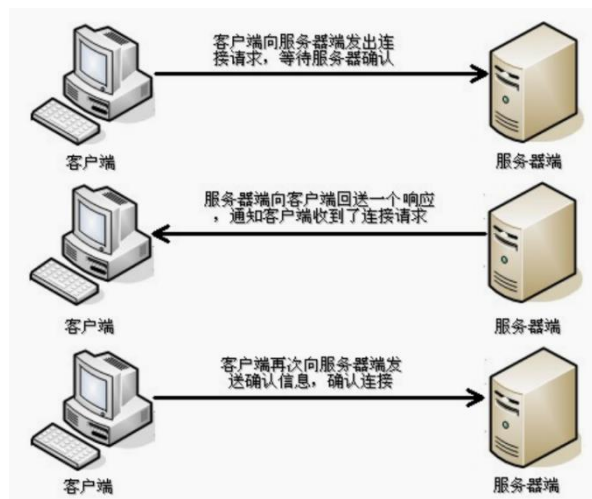
➤ 常量：

- `StandardCharsets.UTF_8`
- `StandardCharsets.UTF_16`
- `StandardCharsets.UTF_16BE`

- StandardCharsets.UTF_16LE
 - StandardCharsets.US_ASCII
 - StandardCharsets.ISO_8859_1
- 使用这些常量来指定字符集，而不必在代码中硬编码字符集名称，便于提高代码的可读性和可移植性，同时避免了拼写错误和不受支持的字符集名称带来的问题。

六、*java.net* 包

- 作用：用于网络通信，提供了许多类和接口，可以用于处理网络连接、数据传输、网络协议等。
- 支持的传输层协议：
- TCP：有连接且可靠，传输速度慢。**三次握手**：第一次：客户端向服务器端发出连接请求，等待服务器确认；第二次：服务器端向客户端回送一个响应，通知客户端收到了连接请求；第三次：客户端再次向服务器端发送确认信息，确认连接。



- UDP：无连接不可靠，传输速度快
- 类和接口举例：
- `java.net.URL`：表示统一资源定位符，可以通过它来访问网络上的资源。
 - `java.net.URLConnection`：表示应用程序和 URL 之间的通信链接，可以通过它来打开连接、读取和写入数据等。
 - `URLConnection` 类的子类（`HttpURLConnection`、`HttpsURLConnection` 等）：特定协议的连接，例如 HTTP 和 HTTPS。

- **java.net.Socket**: 实现客户端和服务端之间的 TCP 连接。
- **java.net.ServerSocket**: 创建服务端端的 Socket, 用于监听客户端的连接请求。
- **java.net.InetAddress**: 用于表示 IP 地址和主机名, 可以用于执行域名解析和网络连接等操作。
- **java.net.DatagramPacket**、**java.net.DatagramSocket**: 用于实现用户数据报协议 (UDP) 的通信。
- **java.net.Proxy**: 用于创建代理服务器对象, 可以在网络通信中使用代理。

七、*java.net.Socket* 包

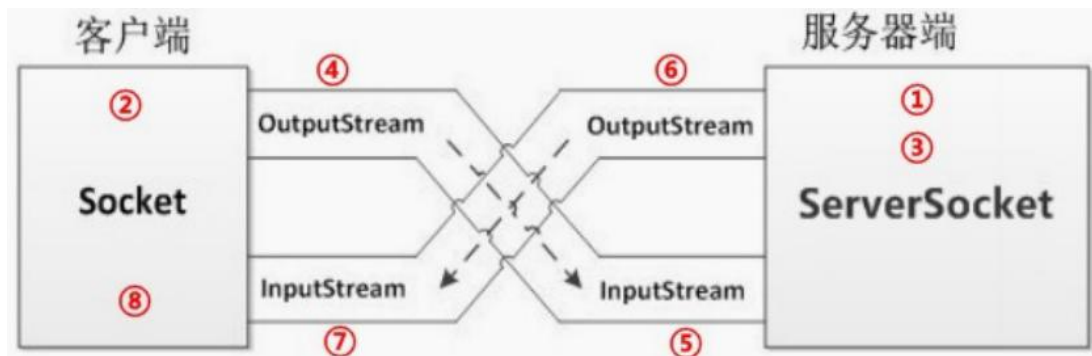
- 实现客户端套接字, 套接字: 两台设备之间通讯的端点。
- 构造方法: **public Socket(String host, int port)**, 创建套接字对象并将其连接到指定主机上的指定端口号。
 - 如果指定的 host 是 null, 则相当于指定地址为**回送地址**。回送地址(127.x.x.x)是本地回送地址, 下面**特殊 IP**有提到, 主要用于网络软件测试以及本地机进程间通信, 无论什么程序, 一旦使用回送地址发送数据, 立即返回, 不进行任何网络传输。
- 成员方法:
 - **public InputStream getInputStream()**: 返回此套接字的输入流。如果此 Socket 具有相关联的通道, 则生成的 InputStream 的所有操作也关联该通道。关闭生成的 InputStream 也将关闭相关的 Socket。
 - **public OutputStream getOutputStream()**: 返回此套接字的输出流。如果此 Socket 具有相关联的通道, 则生成的 OutputStream 的所有操作也关联该通道。关闭生成的 OutputStream 也将关闭相关的 Socket。
 - **public void close()**: 关闭此套接字。一旦一个 socket 被关闭, 它不可再使用。关闭此 socket 也将关闭相关的 InputStream 和 OutputStream。
 - **public void shutdownOutput()**: 禁用此套接字的输出流。任何先前写出的数据将被发送, 随后终止输出流。

八、*Java.net.ServerSocket* 包

- 实现服务器套接字，该对象等待通过网络请求，要**先于客户端**套接字建立。
- 构造方法：**public ServerSocket(int port)**，在创建 ServerSocket 对象时，就可以将其绑定到一个指定为参数 port 的端口号上。
- 成员方法：**public Socket accept()**：侦听并接受连接，返回一个新的 Socket 对象，用于和客户端实现通信。该方法会一直阻塞直到建立连接。

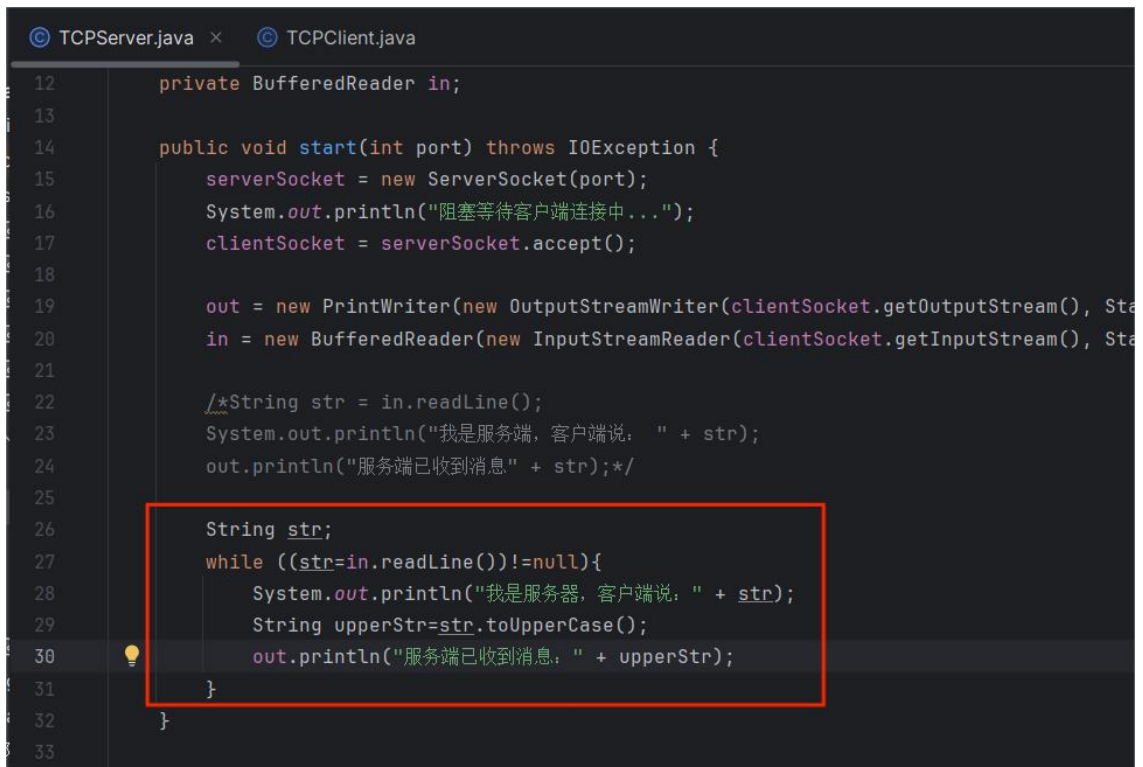
九、TCP 程序交互过程

- 1~5 客户端向服务器端写入数据；6~9 服务器端向客户端返回数据
- ① 服务器端：启动，创建 ServerSocket 对象，等待客户连接。
- ② 客户端：启动，创建 Socket 对象，与服务器端连接。
- ③ 服务器端：接受连接，调用 accept()方法，返回一个 Socket 对象
- ④ 客户端：Socket 对象，获取 OutputStream，向服务端写出数据。
- ⑤ 服务器端：Socket 对象，获取 InputStream，读取客户端发送的数据。
- ⑥ 服务器端：Socket 对象，获取 OutputStream，向客户端回写数据。
- ⑦ 客户端：Socket 对象，获取 InputStream，解析回写数据。
- ⑧ 客户端：释放资源，断开连接。



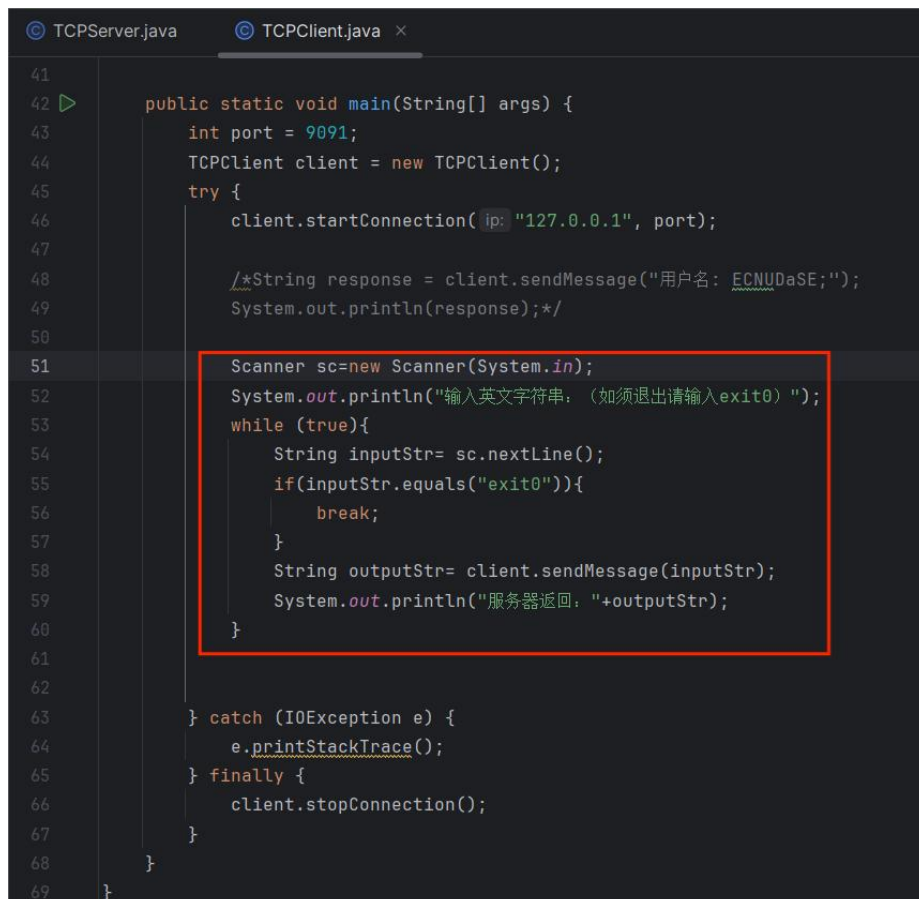
Task1 修改代码段以及实验结果截图:

- 修改代码段截图：
 - TCPServer，修改 start 函数，引入循环可以不断接收客户端的输入，并将所有输入包含小写字母的部分转化为对应大写：



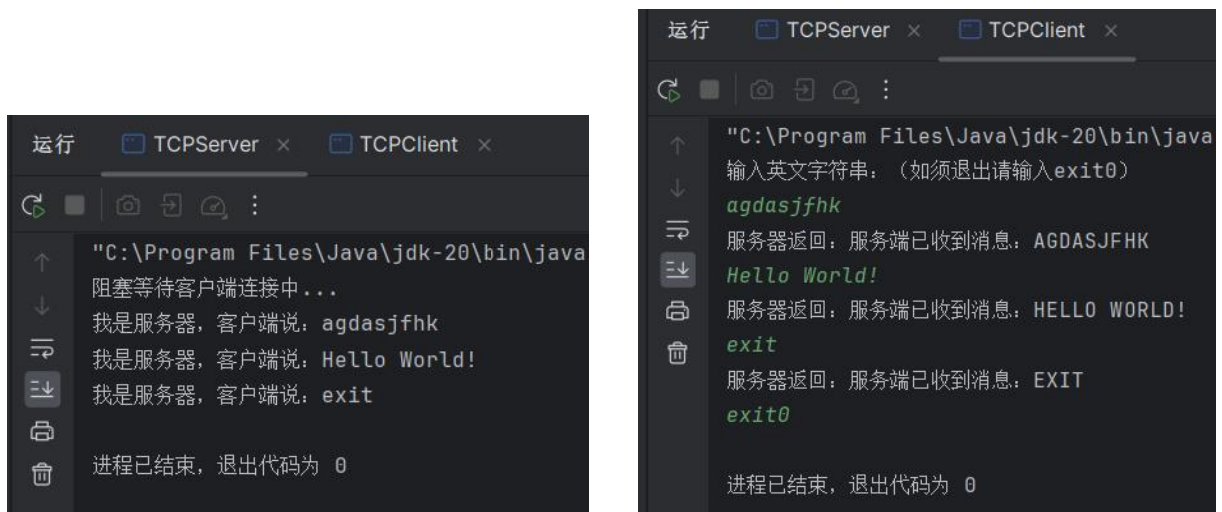
```
12     private BufferedReader in;
13
14     public void start(int port) throws IOException {
15         serverSocket = new ServerSocket(port);
16         System.out.println("阻塞等待客户端连接中...");
17         clientSocket = serverSocket.accept();
18
19         out = new PrintWriter(new OutputStreamWriter(clientSocket.getOutputStream(), Sta
20         in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream(), Sta
21
22         /*String str = in.readLine();
23         System.out.println("我是服务端, 客户端说: " + str);
24         out.println("服务端已收到消息" + str);*/
25
26         String str;
27         while ((str=in.readLine())!=null){
28             System.out.println("我是服务器, 客户端说: " + str);
29             String upperStr=str.toUpperCase();
30             out.println("服务端已收到消息: " + upperStr);
31         }
32     }
33 }
```

- TCPClient: 修改 main 函数, 引入 Scanner 使客户端可以不断读取命令行输入的内容, 并将其发送给服务端, 再将客户端返回内容输出显示到屏幕上:



```
41
42 public static void main(String[] args) {
43     int port = 9091;
44     TCPClient client = new TCPClient();
45     try {
46         client.startConnection(ip: "127.0.0.1", port);
47
48         /*String response = client.sendMessage("用户名: ECNUDeSE;");
49         System.out.println(response);*/
50
51         Scanner sc=new Scanner(System.in);
52         System.out.println("输入英文字符串: (如须退出请输入exit0)");
53         while (true){
54             String inputStr= sc.nextLine();
55             if(inputStr.equals("exit0")){
56                 break;
57             }
58             String outputStr= client.sendMessage(inputStr);
59             System.out.println("服务器返回: "+outputStr);
60         }
61
62     } catch (IOException e) {
63         e.printStackTrace();
64     } finally {
65         client.stopConnection();
66     }
67 }
68
69 }
```


➤ 运行结果截图：



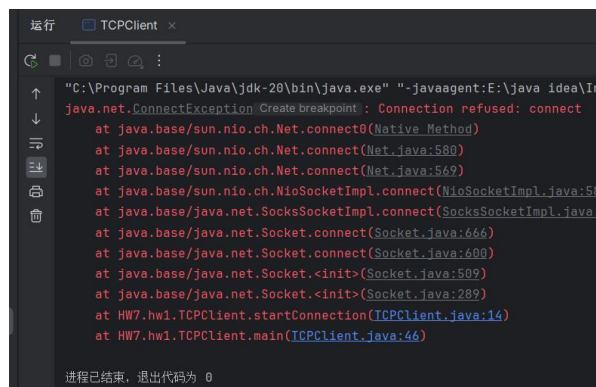
```
运行 TCPServer x TCPClient x
"C:\Program Files\Java\jdk-20\bin\java
阻塞等待客户端连接中...
我是服务器，客户端说: agdasjfhk
我是服务器，客户端说: Hello World!
我是服务器，客户端说: exit
进程已结束，退出代码为 0

运行 TCPServer x TCPClient x
"C:\Program Files\Java\jdk-20\bin\java
输入英文字符串: (如须退出请输入exit0)
agdasjfhk
服务器返回: 服务端已收到消息: AGDASJFHK
Hello World!
服务器返回: 服务端已收到消息: HELLO WORLD!
exit
服务器返回: 服务端已收到消息: EXIT
exit0
进程已结束，退出代码为 0
```

➤ Task1 遇到错误及反思：

- 第一：先运行了 TCPClient 文件，导致报错。一定让 Server 先运行等待客户端连接！

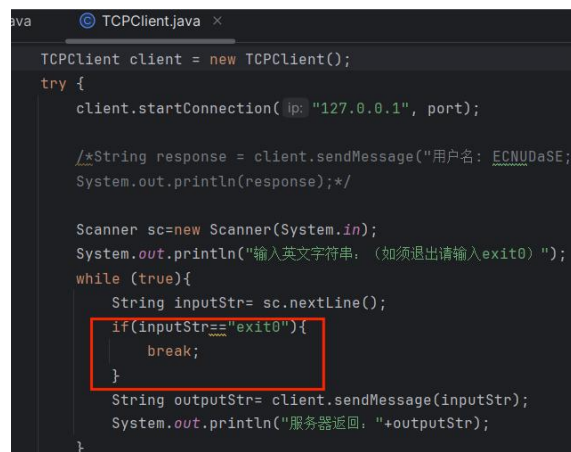
◆ 错误截图：



```
运行 TCPClient x
"C:\Program Files\Java\jdk-20\bin\java.exe" "-javaagent:E:\java_idea\In
java.net.ConnectException Create breakpoint: Connection refused: connect
at java.base/sun.nio.ch.Net.connect0(Native Method)
at java.base/sun.nio.ch.Net.connect(Net.java:580)
at java.base/sun.nio.ch.Net.connect(Net.java:562)
at java.base/sun.nio.ch.NioSocketImpl.connect(NioSocketImpl.java:58
at java.base/java.net.SocksSocketImpl.connect(SocksSocketImpl.java:
at java.base/java.net.Socket.connect(Socket.java:666)
at java.base/java.net.Socket.connect(Socket.java:600)
at java.base/java.net.Socket.<init>(Socket.java:509)
at java.base/java.net.Socket.<init>(Socket.java:289)
at HW7.hw1.TCPClient.startConnection(TCPClient.java:14)
at HW7.hw1.TCPClient.main(TCPClient.java:46)
进程已结束，退出代码为 0
```

- 第二：

◆ 比较字符串代码写错，if()中使用==判断：

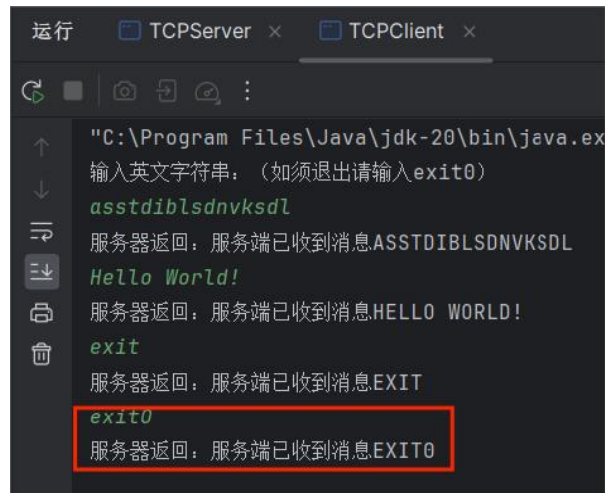


```
java TCPClient.java x
TCPClient client = new TCPClient();
try {
    client.startConnection(ip: "127.0.0.1", port);

    /*String response = client.sendMessage("用户名: ECNUDeSE");
    System.out.println(response);*/

    Scanner sc=new Scanner(System.in);
    System.out.println("输入英文字符串: (如须退出请输入exit0)");
    while (true){
        String inputStr= sc.nextLine();
        if(inputStr== "exit0"){
            break;
        }
        String outputStr= client.sendMessage(inputStr);
        System.out.println("服务器返回: "+outputStr);
    }
}
```


◆ 导致/即使用户端输入 exit0 也无法退出程序：



```
运行 TCPServer x TCPClient x
"C:\Program Files\Java\jdk-20\bin\java.exe"
输入英文字符串：（如须退出请输入exit0）
asstdiblsdnvksdl
服务器返回：服务端已收到消息ASSTDIBLSDNVKSDL
Hello World!
服务器返回：服务端已收到消息HELLO WORLD!
exit
服务器返回：服务端已收到消息EXIT
exit0
服务器返回：服务端已收到消息EXIT0
```

◆ 分析错误原因：

比较字符串内容应该使用.equals()方法而不是==运算符，因为==运算符用于比较对象引用是否相等，而.equals()方法用于比较对象内容是否相等。当使用 if (userInput == "exit")来比较字符串时，比较的是两个不同的字符串对象的引用，而不是它们的内容是否相等，所以即使用户输入的是 "exit"，也不会触发退出条件。

Task2: 修改代码使得每一次accept()的Socket都被一个线程接管，同时接管的逻辑保留Task1的功能，开启一个服务端和三个客户端进行测试，请将实现代码段及运行结果附在实验报告中。

实现代码段& 运行结果截图：

➤ 修改代码段截图：

■ TCPServer:

◆ 修改 start 函数开始部分：

start 方法先创建一个 ServerSocket 并绑定到 port 端口上，然后进入一个无限循环，不断等待客户端的连接请求。在循环中，通过 serverSocket.accept()方法阻塞等待客户端连接，一旦有客户端连接成功，就创建一个 ClientHandler 线程来处理客户端的请求。

具体修改代码如下：

```

public class TCPServer {
    4 个用法
    private ServerSocket serverSocket;
    public void start(int port) throws IOException{
        serverSocket=new ServerSocket(port);
        for(;;){//可以使用while(true), 效果不变
            System.out.println("阻塞等待客户端连接...");
            Socket clientSocket=serverSocket.accept();

            ClientHandler clientHandler =new ClientHandler(clientSocket)
            clientHandler.start();
        }
    }
}

```

创建线程

◆ 加入启动 ClientHandler 类:

继承 Thread 类，作为线程来处理每个客户端的连接请求。首先在 run()方法中，创建了一个 PrintWriter 对象用于向客户端发送数据、一个 BufferedReader 对象用于接收客户端发送的数据。然后进入一个循环，不断地读取客户端发送过来的消息并将其转换成大写形式，并发送回客户端。客户端连接时循环结束。最后添加 closeConnection()函数来关闭与客户端的连接

```

hw2\TCPServer.java x TCPCClient.java
44 class ClientHandler extends Thread{
    5 个用法
45     private Socket socket;
    1 个用法
46     ClientHandler(Socket s) { this.socket=s; }
49     public void run(){
50         super.run();
51         try{
52             PrintWriter out=new PrintWriter(new OutputStreamWriter(socket.getOutputStream(), StandardCharsets.UTF_8), true);
53             BufferedReader in=new BufferedReader(new InputStreamReader(socket.getInputStream(), StandardCharsets.UTF_8));
54             String str;
55             while ((str=in.readLine())!=null){
56                 System.out.println("我是服务器, 客户端说: "+str);
57                 String upperStr=str.toUpperCase();
58                 out.println("服务端已收到信息: "+upperStr);
59             }
60         }catch (IOException e){
61             e.printStackTrace();
62         }finally {
63             closeConnection();
64         }
65     }
66     private void closeConnection(){
67         try {
68             if(socket!=null){
69                 socket.close();
70             }
71         }catch (IOException e){
72             e.printStackTrace();
73         }
74     }
75 }

```

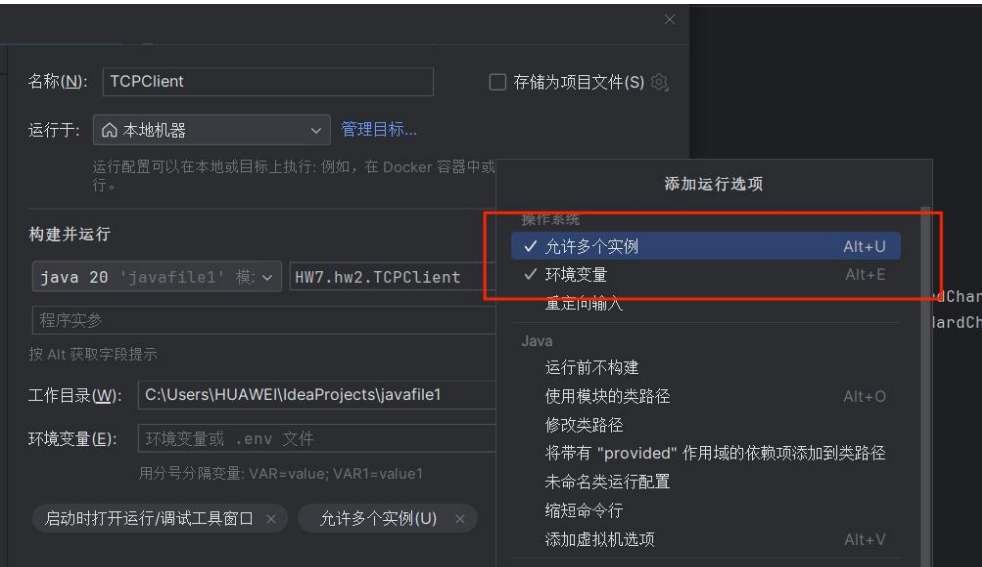
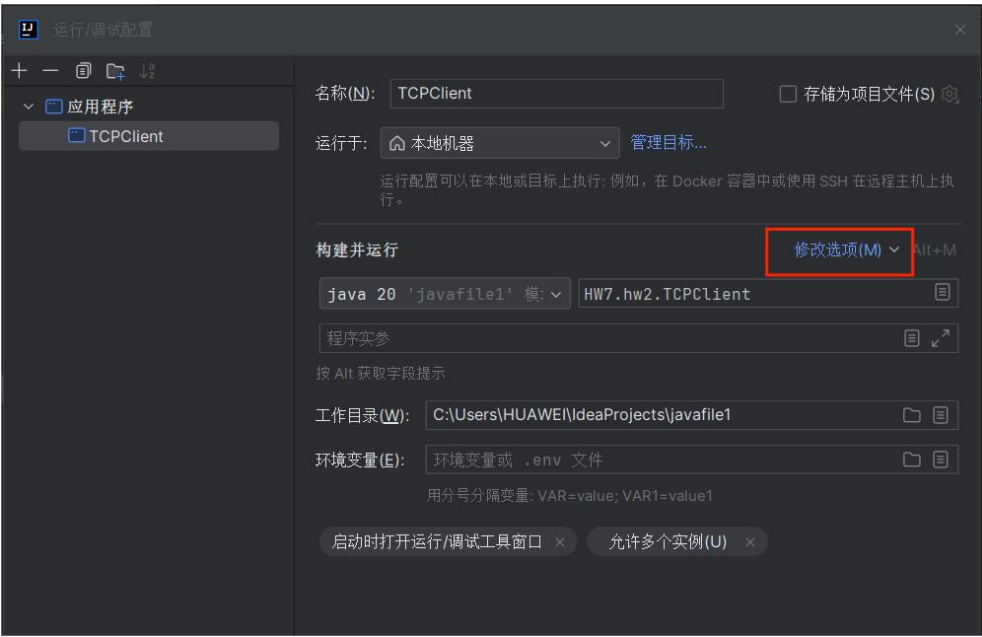
构造方法

重写run()函数: 循环处理客户端输入字符串

停止连接函数, 用于线程中socket的停止

■ TCPClient: 代码不变。

➤ 允许同一文件同时多次运行操作流程截图:



➤ 运行结果截图：

■ TCPServer：

首先输出四次“阻塞等待客户端连接...”，表示与三个 TCPClient 都建立了连接，再接着接收每个客户端输入的内容。

输出四次“阻塞等待客户端连接...”，而不是三次的原因：初始状态下，服务器启动，开始监听指定端口，此时输出一次“阻塞等待客户端连接...”。后面每个客户端连接时，服务器端都会接受连接并创建一个 ClientHandler 线程来处理该连接并输出一次“阻塞等待客户端连接...”，所以一共输出四次。

```
"C:\Program Files\Java\jdk-20\bin\java.exe" "-javaagent:E:\java_idea\
阻塞等待客户端连接...
阻塞等待客户端连接...
阻塞等待客户端连接...
阻塞等待客户端连接...
我是服务器, 客户端说: agdasjfhk
我是服务器, 客户端说: agdasjfhk
我是服务器, 客户端说: agdasjfhk
我是服务器, 客户端说: Hello World1
我是服务器, 客户端说: Hello World2
我是服务器, 客户端说: Hello World3
我是服务器, 客户端说: exit1
我是服务器, 客户端说: exit2
我是服务器, 客户端说: exit3
```

初始状态打印的一次

相同颜色划线部分
接收于一个客户端

■ TCPServer1:

```
运行 TCPServer x TCPClient x TCPClient x TCPClient x
"C:\Program Files\Java\jdk-20\bin\java.exe" "-javaagent:E:\java
请输入英文字符串 (输入exit0退出):
agdasjfhk
服务器返回: 服务端已收到信息: AGDASJFHK
Hello World1
服务器返回: 服务端已收到信息: HELLO WORLD1
exit1
服务器返回: 服务端已收到信息: EXIT1
exit0
进程已结束, 退出代码为 0
```

■ TCPServer2:

```
运行 TCPServer x TCPClient x TCPClient x TCPClient x
"C:\Program Files\Java\jdk-20\bin\java.exe" "-javaagent:E:\java
请输入英文字符串 (输入exit0退出):
agdasjfhk
服务器返回: 服务端已收到信息: AGDASJFHK
Hello World2
服务器返回: 服务端已收到信息: HELLO WORLD2
exit2
服务器返回: 服务端已收到信息: EXIT2
exit0
进程已结束, 退出代码为 0
```

■ TCPServer3:



```
运行 TCPServer x TCPClient x TCPClient x TCPClient x
"C:\Program Files\Java\jdk-20\bin\java.exe" "-javaagent:E:\java
请输入英文字符串（输入exit0退出）：
agdasjfhk
服务器返回：服务端已收到信息：AGDASJFHK
Hello World3
服务器返回：服务端已收到信息：HELLO WORLD3
exit3
服务器返回：服务端已收到信息：EXIT3
exit0

进程已结束，退出代码为 0
```

- **Task3:** 查阅资料，总结半包粘包产生的原因以及相关解决方案，尝试解决以上代码产生的半包粘包问题，将修改代码和解决思路附在实验报告中。

储备知识:

一、半包

- 定义：发送方在发送数据时，可能将数据分成多个部分，称为数据包。而接收方在读取数据时，可能一次性读取的数据量少于一个完整的数据包，导致接收到的数据只是数据包的一部分。
- 产生原因：
 - 数据发送速度快于接收速度：如果发送方以较快的速度发送数据包，而接收方的处理速度比发送方慢，就会导致接收方在处理一个数据包时，下一个数据包已经到达，从而造成半包问题。
 - 网络拥塞：网络拥塞可能导致数据包在传输过程中被分割成更小的部分进行传输。如果在传输过程中发生了数据包分割，而接收方在此时接收到部分数据，就会导致半包问题。
 - TCP 缓冲区大小限制：TCP 协议有一个接收缓冲区，用于存储接收到的数据。如果接收缓冲区的大小有限，并且接收方未能及时从缓冲区中读取数据，那么接收方可能会只读取到数据包的一部分，从而引发半包问题。
 - 操作系统接收缓冲区设置不当：操作系统的网络参数设置不当也可能导致半包问题。例如，如果操作系统的接收缓冲区大小设置过小，无法容纳完整的数据包，就可能导致半包问题。

- 数据包大小超过 MTU 限制：网络中的数据包有一个最大传输单元（MTU）限制，如果数据包的大小超过了 MTU 限制，那么数据包就会被分割成更小的部分进行传输。如果接收方在接收到这些部分时未能组装成完整的数据包，就可能产生半包问题。
- 对应解决方案：
 - 数据发送速度快于接收速度：
 - ◆ 流量控制：实现发送方与接收方之间的流量控制，使发送速度适应接收速度。可以使用滑动窗口等技术来实现流量控制。
 - ◆ 接收方缓冲区调整：增大接收方的缓冲区大小，以便能够容纳更多的数据，从而减少数据丢失的可能性。
 - 网络拥塞：
 - ◆ 拥塞控制：实现拥塞控制机制，如 TCP 的拥塞控制算法，以减少网络拥塞的发生。
 - ◆ 重传机制：在发生数据丢失时进行重传，确保数据的可靠传输。
 - TCP 缓冲区大小限制：
 - ◆ 调整 TCP 缓冲区大小：通过操作系统或应用程序的参数设置来调整 TCP 缓冲区的大小，以确保能够容纳足够大的数据包。
 - 操作系统接收缓冲区设置不当：
 - ◆ 优化操作系统参数：调整操作系统的网络参数，如接收缓冲区大小等，以适应实际的网络通信需求。
 - 数据包大小超过 MTU 限制：
 - ◆ 路径 MTU 发现：使用路径 MTU 发现技术，动态地确定网络路径上的最大传输单元，以避免数据包被分割成更小的部分。
 - ◆ 分段与重组：在应用层进行数据包的分段与重组，以确保数据包大小不会超过 MTU 限制。

二、粘包

- 定义：发送方在发送数据时，可能将多个数据包连续发送，但接收方却将它们当做一个数据包处理，导致多个数据包粘在一起，称为粘包。

➤ 产生原因：

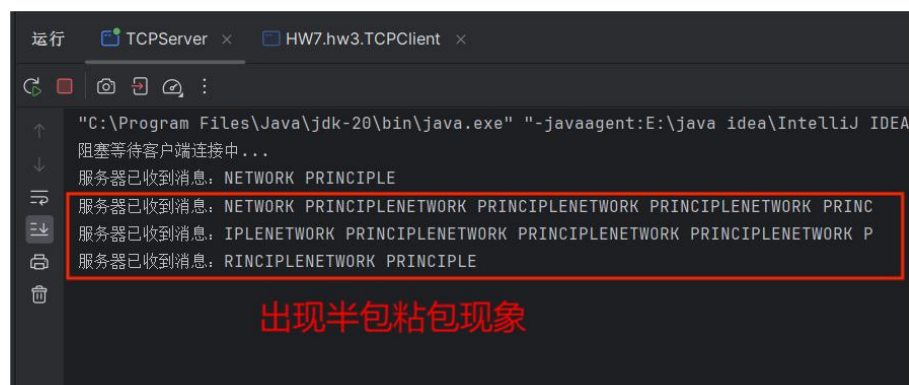
- 数据发送速度慢于接收速度：如果发送方以较慢的速度发送数据包，而接收方的处理速度比发送方快，就会导致接收方在处理一个数据包时，下一个数据包尚未到达，从而多个数据包被组合成一个粘包。
- TCP 缓冲区中的多个数据包被一次性读取：接收方从 TCP 缓冲区中一次性读取了多个数据包，这些数据包被合并成一个粘包。
- 缺乏消息边界：如果发送方没有在消息之间明确定义边界（如特殊字符或长度字段），接收方可能无法区分不同消息的边界，从而将多个消息组合成一个粘包。
- 操作系统 TCP/IP 协议栈的处理机制：操作系统的 TCP/IP 协议栈可能会对数据包进行合并或拆分，从而导致接收方接收到的数据包不是原始数据包的完整形式。

➤ 对应解决方案：

- 数据发送速度慢于接收速度->发送方应该以更快的速度发送数据，或者接收方应该调整处理速度以适应发送方的速度。
- TCP 缓冲区中的多个数据包被一次性读取->接收方可以通过设置合适的缓冲区大小来避免一次性读取过多数据，或者在接收端对数据进行适当的分段处理。
- 缺乏消息边界->发送方和接收方应该在消息之间明确定义边界，如添加特殊字符或长度字段作为消息的分隔符，以便接收方能够正确地地区分不同消息的边界。
- 操作系统 TCP/IP 协议栈的处理机制->可采用消息长度信息、消息结束标志等方式，确保接收方能够准确地分割接收到的数据，或者使用定长消息格式，确保每个消息的长度固定。

修改代码&解决思路截图：

➤ 修改前，运行结果截图：

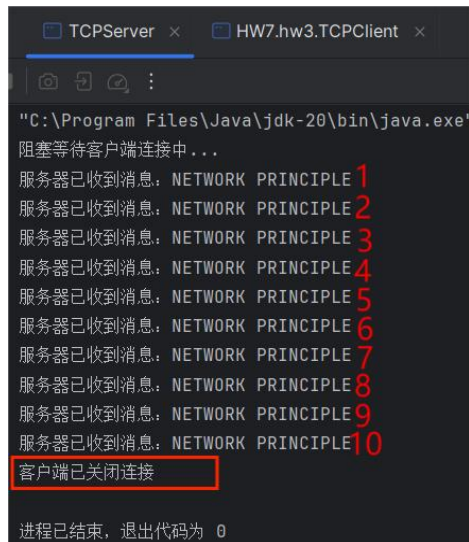


The screenshot shows a Java IDE with two tabs: 'TCPServer' and 'HW7.hw3.TCPClient'. The 'TCPServer' tab is active, displaying the following log output:

```
"C:\Program Files\Java\jdk-20\bin\java.exe" "-javaagent:E:\java idea\IntelliJ IDEA
阻塞等待客户端连接中...
服务器已收到消息: NETWORK PRINCIPLE
服务器已收到消息: NETWORK PRINCIPLENETWORK PRINCIPLENETWORK PRINCIPLENETWORK PRINC
服务器已收到消息: IPLENETWORK PRINCIPLENETWORK PRINCIPLENETWORK PRINCIPLENETWORK P
服务器已收到消息: RINCIPLNETWORK PRINCIPLE
```

A red rectangular box highlights the three lines of log output that contain fragmented data. Below the log output, the text '出现半包粘包现象' (Half-packet sticky packet phenomenon) is written in red.

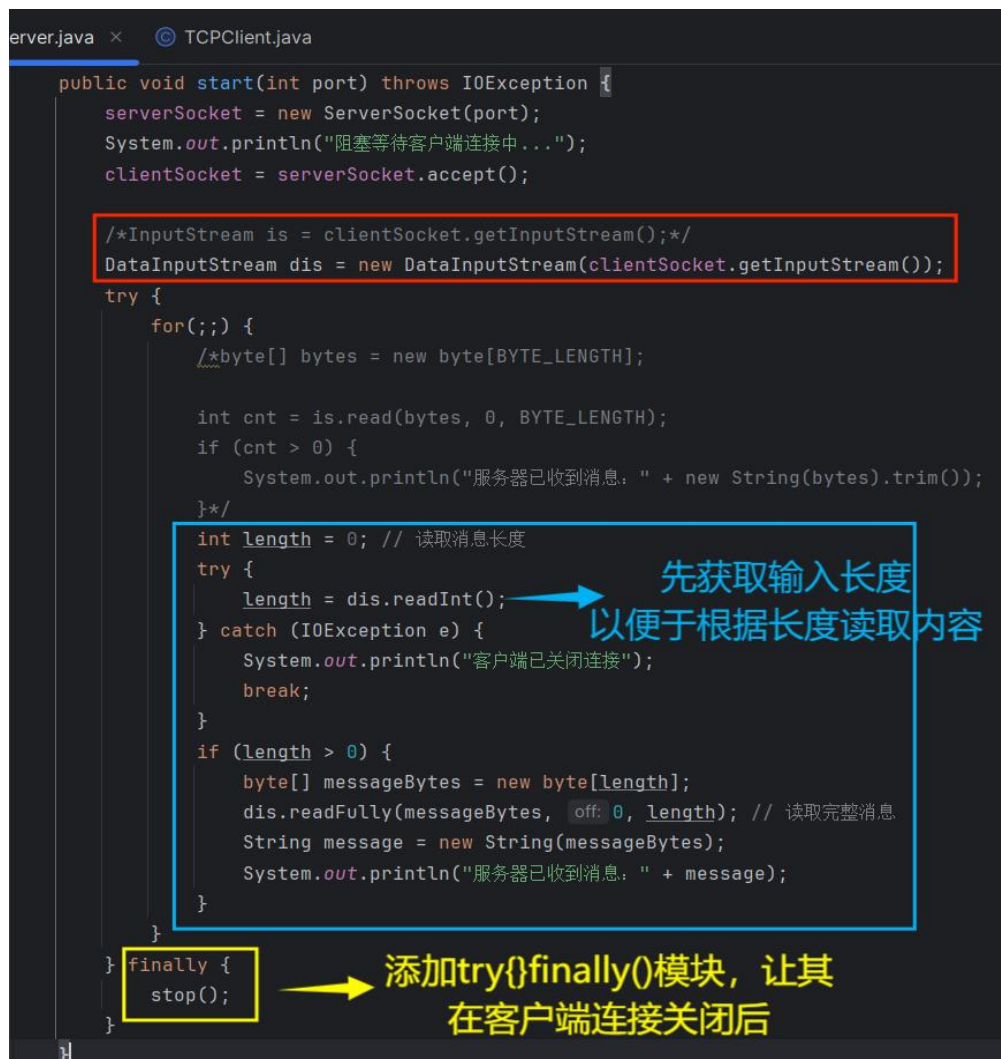
➤ 修改后，运行结果截图：



```
"C:\Program Files\Java\jdk-20\bin\java.exe"  
阻塞等待客户端连接中...  
服务器已收到消息: NETWORK PRINCIPLE 1  
服务器已收到消息: NETWORK PRINCIPLE 2  
服务器已收到消息: NETWORK PRINCIPLE 3  
服务器已收到消息: NETWORK PRINCIPLE 4  
服务器已收到消息: NETWORK PRINCIPLE 5  
服务器已收到消息: NETWORK PRINCIPLE 6  
服务器已收到消息: NETWORK PRINCIPLE 7  
服务器已收到消息: NETWORK PRINCIPLE 8  
服务器已收到消息: NETWORK PRINCIPLE 9  
服务器已收到消息: NETWORK PRINCIPLE 10  
客户端已关闭连接  
进程已结束，退出代码为 0
```

➤ 修改代码：

■ TCPServer:



```
server.java x TCPServer.java  
  
public void start(int port) throws IOException {  
    serverSocket = new ServerSocket(port);  
    System.out.println("阻塞等待客户端连接中...");  
    clientSocket = serverSocket.accept();  
  
    /*InputStream is = clientSocket.getInputStream();*/  
    DataInputStream dis = new DataInputStream(clientSocket.getInputStream());  
    try {  
        for(;;) {  
            /*byte[] bytes = new byte[BYTE_LENGTH];  
  
            int cnt = is.read(bytes, 0, BYTE_LENGTH);  
            if (cnt > 0) {  
                System.out.println("服务器已收到消息: " + new String(bytes).trim());  
            }*/  
  
            int length = 0; // 读取消息长度  
            try {  
                length = dis.readInt();  
            } catch (IOException e) {  
                System.out.println("客户端已关闭连接");  
                break;  
            }  
  
            if (length > 0) {  
                byte[] messageBytes = new byte[length];  
                dis.readFully(messageBytes, 0, length); // 读取完整消息  
                String message = new String(messageBytes);  
                System.out.println("服务器已收到消息: " + message);  
            }  
        }  
    } finally {  
        stop();  
    }  
}
```

先获取输入长度
以便于根据长度读取内容

添加try{}finally()模块，让其在客户端连接关闭后

■ TCPClient:

```
public void startConnection(String ip, int port) throws IOException {
    clientSocket = new Socket(ip, port);
    /*out=clientSocket.getOutputStream();*/
    out = new DataOutputStream(clientSocket.getOutputStream());
}

1个用法
public void sendMessage(String msg) throws IOException {
    for (int i = 0; i < 10; i++) {
        /*out.write(msg.getBytes());*/
        byte[] messageBytes = msg.getBytes();
        out.writeInt(messageBytes.length);
        out.write(messageBytes);
    }
}
```

先向服务端写入信息长度，再向服务端写入具体信息

➤ 解决思路:

- 让客户端每次发送信息之前先向服务端写入信息长度，再写入具体信息
- 让服务端先尝试读取服务端发来的信息长度，并根据该长度进行字节数组创建，这之后去读取服务端发来的该长度的信息，并进行打印。
- 由于需要信息长度的参与，因此使用可以读取格式化数据的 `DataInputStream` 和 `DataOutputStream` 类来处理输入输出流

➤ InputStream 和 OutputStream 类与 DataInputStream 和 DataOutputStream 类区别:

- `InputStream` 和 `OutputStream` 是处理字节流的基本抽象类；`DataInputStream` 和 `DataOutputStream` 是它们的子类，提供了对基本数据类型和字符串进行格式化输入输出的功能。
- 功能上，`DataInputStream` 和 `DataOutputStream` 则除了具有 `InputStream` 和 `OutputStream` 的基本功能外，还提供了对基本数据类型（如 `int`、`double`、`boolean` 等）和字符串进行格式化读写的方法。

➤ 出错反思:

■ 错误代码:

没有在 `start` 函数中，针对 `length` 进行预先异常处理，无 `try` 模块，导致客户端在发送完消息后立即关闭了连接，但服务器端仍然在尝试读取消息，无法正确地检测到输入流的末尾并退出循环，出现 EOF 异常。

```
public void start(int port) throws IOException {
    serverSocket = new ServerSocket(port);
    System.out.println("阻塞等待客户端连接中...");
    clientSocket = serverSocket.accept();

    /*InputStream is = clientSocket.getInputStream();*/
    DataInputStream dis = new DataInputStream(clientSocket.getInputStream());
    for (; ; ) {
        int length = dis.readInt();
        if (length > 0) {
            byte[] messageBytes = new byte[length];
            dis.readFully(messageBytes, 0, length);
            String message = new String(messageBytes);
            System.out.println("服务器已收到消息: " + message);
        }
    }
}
```

客户端不继续写入时，服务端的readInt就会开始出错

■ 错误输出：

```
运行 TCPServer x HW7.hw3.TCPClient x
"C:\Program Files\Java\jdk-20\bin\java.exe" "-javaagent:E:\java_idea\IntelT
阻塞等待客户端连接中...
服务器已收到消息: NETWORK PRINCIPLE
服务器已收到消息: NETWORK PRINCIPLE
服务器已收到消息: NETWORK PRINCIPLE
服务器已收到消息: NETWORK PRINCIPLE
服务器已收到消息: NETWORK PRINCIPLE
服务器已收到消息: NETWORK PRINCIPLE
服务器已收到消息: NETWORK PRINCIPLE
服务器已收到消息: NETWORK PRINCIPLE
服务器已收到消息: NETWORK PRINCIPLE
服务器已收到消息: NETWORK PRINCIPLE
java.io.EOFException Create breakpoint
    at java.base/java.io.DataInputStream.readInt(DataInputStream.java:386)
    at HW7.hw3.TCPServer.start(TCPServer.java:28)
    at HW7.hw3.TCPServer.main(TCPServer.java:55)
进程已结束，退出代码为 0
```

■ 解决思路：

在服务器端 start 函数外层添加 try{}finally{}模块，并配合 for 循环中的 try{}catch()模块，这时，当客户端关闭连接后，服务端会检测到 DataInputStream.readInt() 方法抛出的异常，并在捕获异常后退出循环，然后关闭连接。

二、基于 TCP 的 Socket 编程优化题目要求及实现情况

Task1: 继续修改TCPClient类，使其发送和接收并行，达成如下效果，当服务端和客户端建立连接后，无论是服务端还是客户端均能随时从控制台发送消息、将接收的信息打印在控制台，将修改后的TCPClient代码附在实验报告中，并展示运行结果。

➤ TCPClient 代码：

- TCPClient 类：添加开始连接和停止连接函数，开始连接函数与 TCPServer 类类似思路，建立一个无限循环，保证随时可以创建 serverHandler 线程处理服务端的收发信息。主函数，进行 ip 和端口号初始化之后与服务器连接，利用 try 模块捕获异常。

```
1 package HW8.task1;
2
3 import java.io.*;
4 import java.net.Socket;
5 import java.nio.charset.StandardCharsets;
6 import java.util.Scanner;
7
8 public class TCPClient {
9     4 个用法
10     private Socket clientSocket;
11
12     1 个用法
13     public void startConnection(String ip, int port) throws IOException {
14         clientSocket = new Socket(ip, port);
15         for (; ) {
16             ServerHandler serverHandler = new ServerHandler(clientSocket);
17             serverHandler.start();
18         }
19     }
20
21     1 个用法
22     public void stopConnection() {
23         try {
24             if (clientSocket != null) {
25                 clientSocket.close();
26             }
27         } catch (IOException e) {
28             e.printStackTrace();
29         }
30     }
31
32     public static void main(String[] args) {
33         int port = 9091;
34         TCPClient client = new TCPClient();
35         try {
36             client.startConnection(ip: "127.0.0.1", port);
37         } catch (IOException e) {
38             e.printStackTrace();
39         } finally {
40             client.stopConnection();
41         }
42     }
43 }
```

类比server处理思路
令客户端能随时收发数据

- **ServerReadHandler 线程：**首先定义一个私有且不变的成员变量 `bufferedReader`，添加以输入流为参数的构造方法。然后重写覆盖线程的 `run()` 函数：读取服务端输入的每一行数据，并立即打印输出，利用 `try` 模块捕获异常。

```

2 个用法
43 class ServerReadHandler extends Thread {
    2 个用法
44     private final BufferedReader bufferedReader;
45
    1 个用法
46     ServerReadHandler(InputStream inputStream) {
47         this.bufferedReader = new BufferedReader(new InputStreamReader(inputStream, StandardCharsets.UTF_8));
48     }
49
50     public void run() {
51         try {
52             while (true) {
53                 String str = bufferedReader.readLine();
54                 if (str == null) {
55                     System.out.println("读到服务端数据为空");
56                     break;
57                 } else {
58                     System.out.println("读到服务端数据为: " + str);
59                 }
60             }
61         } catch (IOException e) {
62             e.printStackTrace();
63         }
64     }
65 }
  
```

增强封装性、不可变性和线程安全性

- **ServerWriteHandler 线程：**与 `read` 线程类似的思路，首先定义两个私有且不变的成员变量 `printWriter` 和 `sc`，添加以输出流为参数的构造方法。然后添加 `send()` 函数用来发送输出的信息。最后重写覆盖线程的 `run()` 函数：使用 `scanner` 进行连续读取命令行输入，并将输入数据通过 `send()` 函数进行发送。

```

2 个用法
67 class ServerWriteHandler extends Thread {
    2 个用法
68     private final PrintWriter printWriter;
    3 个用法
69     private final Scanner sc;
70
    1 个用法
71     ServerWriteHandler(OutputStream outputStream) {
72         this.printWriter = new PrintWriter(new OutputStreamWriter(outputStream, StandardCharsets.UTF_8), autoFlush: true);
73         this.sc = new Scanner(System.in);
74     }
75
    1 个用法
76     void send(String str) { this.printWriter.println(str); }
77
78     public void run() {
79         while (sc.hasNext()) {
80             System.out.println("客户端请写入数据: ");
81             String str = sc.next();
82             send(str);
83         }
84     }
85 }
86 }
  
```

向客户端发送数据

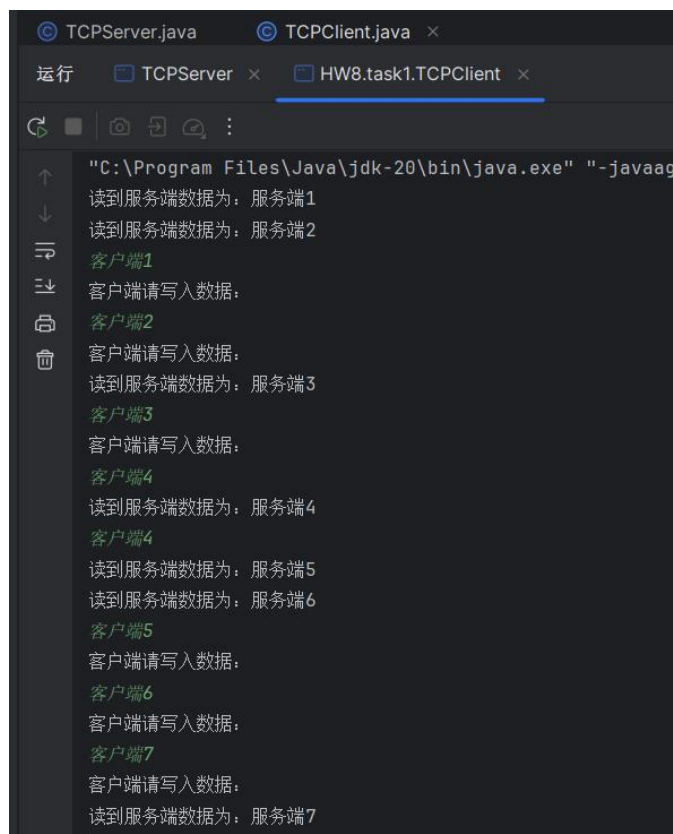
- **ServerHandler 线程：**与 TCPServer 类的 ClientHandler 线程类相同的思路，主要就是将读写线程进行一个汇合，这样直接在 TCPClient 类里调用 ServerHandler 更加方便，可读性更高，也便于管理。分别声明成员变量，定义构造函数并重写 run()方法。

```
89  class ServerHandler extends Thread {  
    1个用法  
90      private Socket socket;  
    2个用法  
91      private final ServerReadHandler serverReadHandler;  
    2个用法  
92      private final ServerWriteHandler serverWriteHandler;  
93  
    1个用法  
94  @ServerHandler(Socket socket) throws IOException {  
95      this.socket = socket;  
96      this.serverReadHandler = new ServerReadHandler(socket.getInputStream());  
97      this.serverWriteHandler = new ServerWriteHandler(socket.getOutputStream());  
98  }  
99  
100  public void run() {  
101      super.run();  
102      serverReadHandler.start();  
103      serverWriteHandler.start();  
104  }  
105  }
```

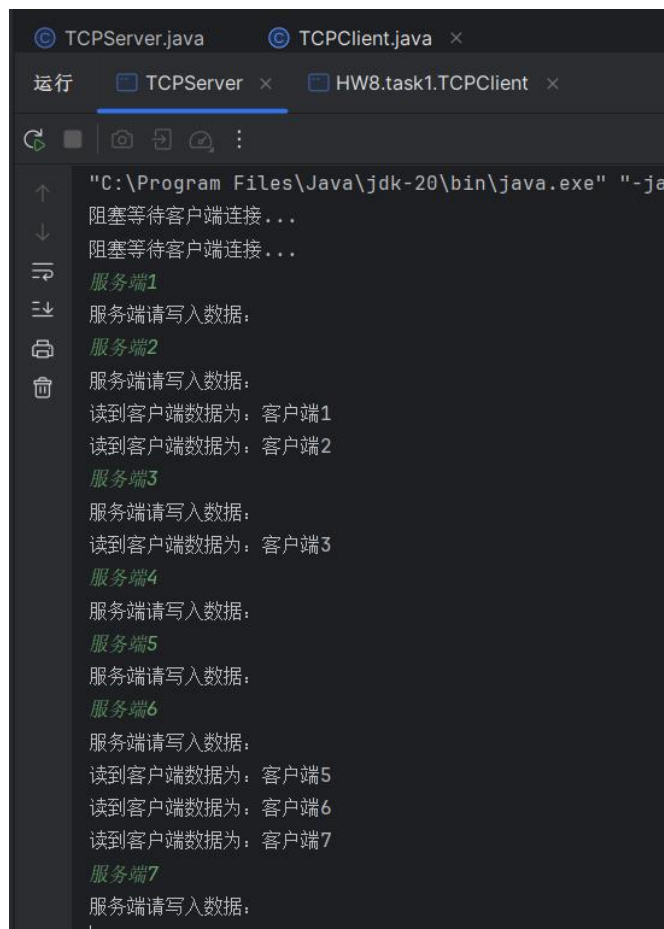
统一管理

- 运行结果截图：成功实现客户端和服务端都能随时读取信息和发送信息，并打印输出接收到的信息。

- **客户端截图：**



■ 服务端截图：

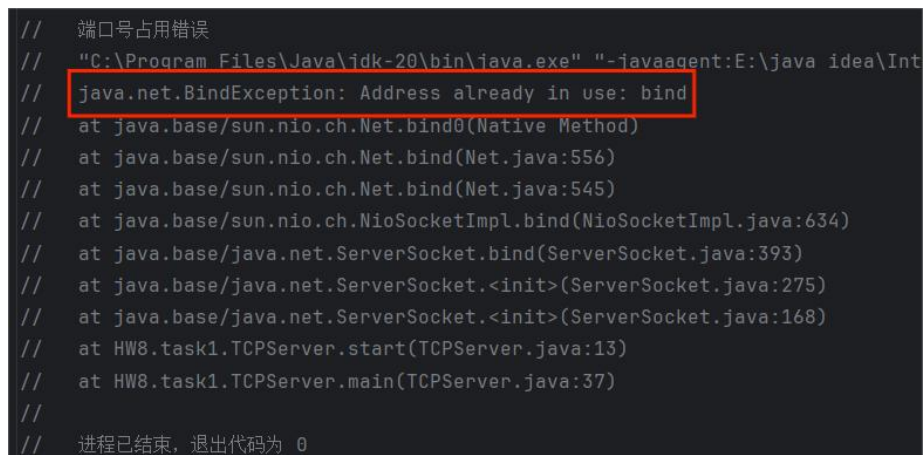


```
TCPServer.java  TCPClient.java x
运行  TCPServer x  HW8.task1.TCPClient x

"C:\Program Files\Java\jdk-20\bin\java.exe" "-ja
阻塞等待客户端连接...
阻塞等待客户端连接...
服务端1
服务端请写入数据:
服务端2
服务端请写入数据:
读到客户端数据为: 客户端1
读到客户端数据为: 客户端2
服务端3
服务端请写入数据:
读到客户端数据为: 客户端3
服务端4
服务端请写入数据:
服务端5
服务端请写入数据:
服务端6
服务端请写入数据:
读到客户端数据为: 客户端5
读到客户端数据为: 客户端6
读到客户端数据为: 客户端7
服务端7
服务端请写入数据:
```

➤ 出错反思：

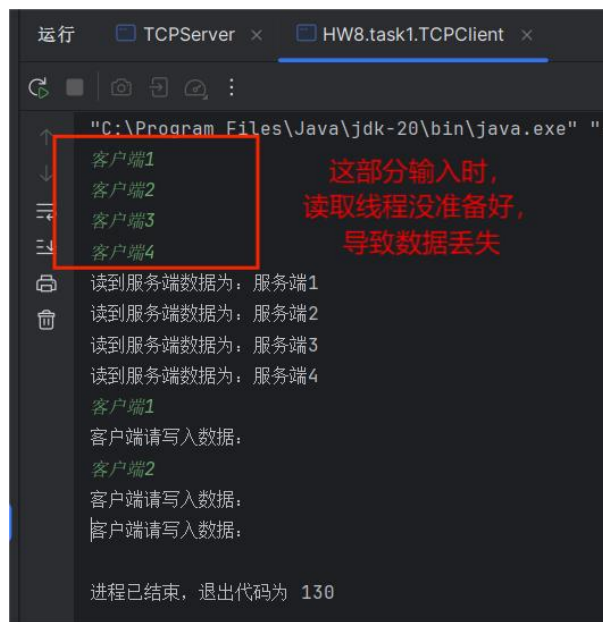
- 端口号占用问题：需要更换端口号或者把其他在该端口号上运行的进程杀掉，我重新运行程序或者更换成 9090 都无法解决，然后关机重启，就不会报错了，怀疑是当时电脑待机时，有应用占用端口号没有释放，重启解决了占用端口的进程。



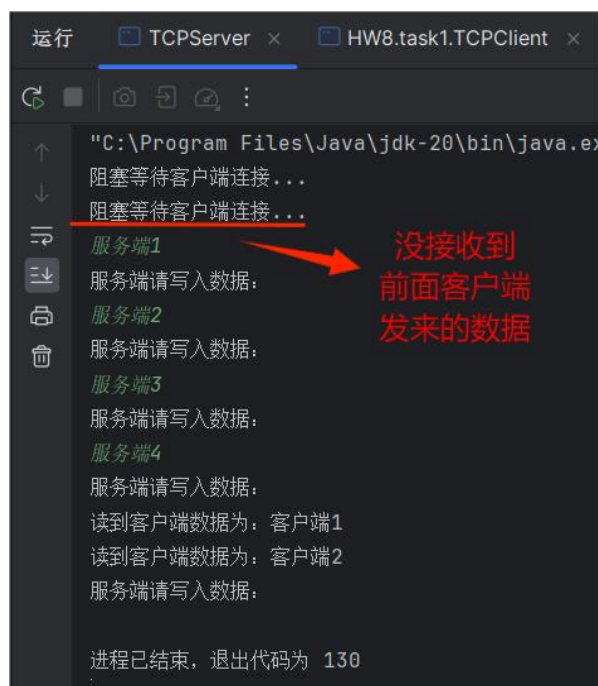
```
// 端口号占用错误
// "C:\Program Files\Java\jdk-20\bin\java.exe" "-javaagent:E:\java idea\Int
// java.net.BindException: Address already in use: bind
// at java.base/sun.nio.ch.Net.bind0(Native Method)
// at java.base/sun.nio.ch.Net.bind(Net.java:556)
// at java.base/sun.nio.ch.Net.bind(Net.java:545)
// at java.base/sun.nio.ch.NioSocketImpl.bind(NioSocketImpl.java:634)
// at java.base/java.net.ServerSocket.bind(ServerSocket.java:393)
// at java.base/java.net.ServerSocket.<init>(ServerSocket.java:275)
// at java.base/java.net.ServerSocket.<init>(ServerSocket.java:168)
// at HW8.task1.TCPServer.start(TCPServer.java:13)
// at HW8.task1.TCPServer.main(TCPServer.java:37)
//
// 进程已结束，退出代码为 0
```

- 有些时候，客户端读取线程准备较慢，会造成服务端接收不到客户端输入的数据：这时候最好是服务

端先进行交互，让客户端准备一会儿。



```
运行 TCPServer x HW8.task1.TCPClient x
"C:\Program Files\Java\jdk-20\bin\java.exe" "-
客户端1
客户端2
客户端3
客户端4
读服务端数据为: 服务端1
读服务端数据为: 服务端2
读服务端数据为: 服务端3
读服务端数据为: 服务端4
客户端1
客户端请写入数据:
客户端2
客户端请写入数据:
客户端请写入数据:
进程已结束, 退出代码为 130
```

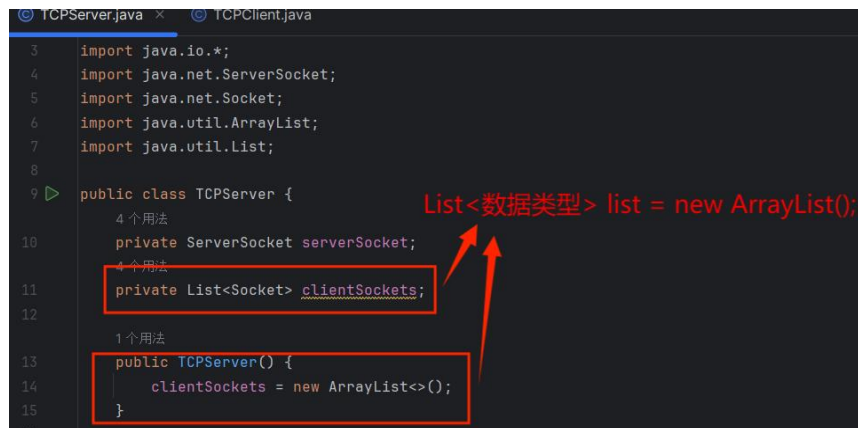


```
运行 TCPServer x HW8.task1.TCPClient x
"C:\Program Files\Java\jdk-20\bin\java.exe"
阻塞等待客户端连接...
阻塞等待客户端连接...
服务端1
服务端请写入数据:
服务端2
服务端请写入数据:
服务端3
服务端请写入数据:
服务端4
服务端请写入数据:
读到客户端数据为: 客户端1
读到客户端数据为: 客户端2
服务端请写入数据:
进程已结束, 退出代码为 130
```

Task2: 修改TCPServer和TCPClient类, 达成如下效果, 每当有新的客户端和服务端建立连接后, 服务端向当前所有建立连接的客户端发送消息, 消息内容为当前所有已建立连接的Socket对象的 `getRemoteSocketAddress()` 的集合, 请测试客户端加入和退出的情况, 将修改后的代码附在实验报告中, 并展示运行结果。

➤ TCPServer 代码:

在 task1 中实现的 TCP 服务端代码地基础上修改, 需要创建一个 socket 数组, 用来存储当前状态下所有连接到服务端的客户端, 添加一个服务端构造方法来初始化这个 socket 数组。



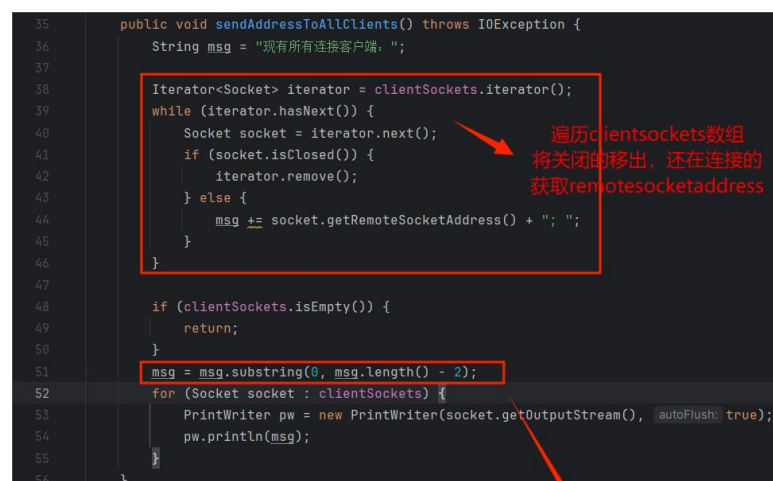
```
3 import java.io.*;
4 import java.net.ServerSocket;
5 import java.net.Socket;
6 import java.util.ArrayList;
7 import java.util.List;
8
9 public class TCPServer {
10     private ServerSocket serverSocket;
11     private List<Socket> clientSockets;
12
13     public TCPServer() {
14         clientSockets = new ArrayList<>();
15     }
16 }
```

与此同时，为了实现题干中要求，新建一个 `sendAddressToAllClient` 函数，并在 `start` 函数中，每次有 `clientsocket` 建立时，及时将该套接字加入 `clientsockets` 数组，并调用一遍 `sendAddressToAllClient` 函数。



```
20 public void start(int port) throws IOException {
21     serverSocket = new ServerSocket(port);
22     for (; ; ) {
23         System.out.println("阻塞等待客户端连接...");
24         Socket clientSocket = serverSocket.accept();
25
26         clientSockets.add(clientSocket);
27         ClientHandler clientHandler = new ClientHandler(clientSocket);
28         clientHandler.start();
29
30         System.out.println("客户端已连接: " + clientSocket.getRemoteSocketAddress());
31         sendAddressToAllClients();
32     }
33 }
```

`sendAddressToAllClient` 函数：首先声明并初始化一个字符串变量 `msg`，用于存储要发送给客户端的消息。然后创建一个迭代器来遍历客户端套接字列表中的每个套接字：检查当前套接字是否已关闭。如果是，从列表中移除它；否则，将其远程套接字地址添加到消息中。最后，如果客户端套接字列表为空，直接返回，不需要发送任何消息；反之，遍历客户端套接字列表，并向每个客户端发送消息。



```
35 public void sendAddressToAllClients() throws IOException {
36     String msg = "现有所有连接客户端: ";
37
38     Iterator<Socket> iterator = clientSockets.iterator();
39     while (iterator.hasNext()) {
40         Socket socket = iterator.next();
41         if (socket.isClosed()) {
42             iterator.remove();
43         } else {
44             msg += socket.getRemoteSocketAddress() + "; ";
45         }
46     }
47
48     if (clientSockets.isEmpty()) {
49         return;
50     }
51     msg = msg.substring(0, msg.length() - 2);
52     for (Socket socket : clientSockets) {
53         PrintWriter pw = new PrintWriter(socket.getOutputStream(), autoFlush: true);
54         pw.println(msg);
55     }
56 }
```

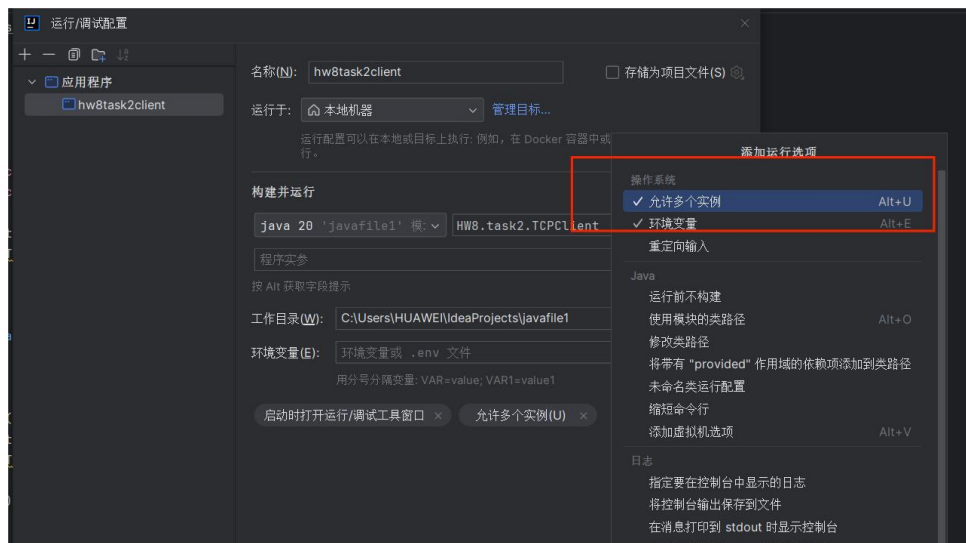
```

1 个用法
40 public void stop() {
41     try {
42         if (serverSocket != null) {
43             serverSocket.close();
44         }
45     } catch (IOException e) {
46         e.printStackTrace();
47     }
48 }

```

substring()方法：从字符串中提取指定范围的子串。
 1).substring(int beginIndex): 返回从指定索引开始到字符串末尾的子字符串。
 2).substring(int beginIndex,int endIndex): 返回从指定索引开始到指定索引结束之间的子字符串。(半开半闭区间)

- TCPClient 代码：操作代码与 task1 保持不变。
- 运行代码操作：参考 Lab7task2 的操作，要先增加允许多个实例运行的 TCPClient 程序，然后在运行了 TCPServer 的基础上运行多个 TCPClient，测试他们加入退出的运行情况。



- 代码运行结果：

■ TCPServer:



■ 多客户端 TCPClient:

运行 TCPServer x hw8task2client x hw8task2client x hw8task2client x hw8task2client x

```
"C:\Program Files\Java\jdk-20\bin\java.exe" "-javaagent:E:\java_idea\IntelliJ IDEA 2023.3.4\lib\idea_rt.jar"
收到客户端信息: 现有所有连接客户端: /127.0.0.1:56200
收到客户端信息: 现有所有连接客户端: /127.0.0.1:56200; /127.0.0.1:56206
收到客户端信息: 现有所有连接客户端: /127.0.0.1:56200; /127.0.0.1:56206; /127.0.0.1:56209
收到客户端信息: 现有所有连接客户端: /127.0.0.1:56200; /127.0.0.1:56206; /127.0.0.1:56209; /127.0.0.1:56214
进程已结束, 退出代码为 130
```

第一个连接的客户端, 打印了四次

运行 TCPServer x hw8task2client x hw8task2client x hw8task2client x hw8task2client x

```
"C:\Program Files\Java\jdk-20\bin\java.exe" "-javaagent:E:\java_idea\IntelliJ IDEA 2023.3.4\lib\idea_rt.jar"
收到客户端信息: 现有所有连接客户端: /127.0.0.1:56200; /127.0.0.1:56206
收到客户端信息: 现有所有连接客户端: /127.0.0.1:56200; /127.0.0.1:56206; /127.0.0.1:56209
收到客户端信息: 现有所有连接客户端: /127.0.0.1:56200; /127.0.0.1:56206; /127.0.0.1:56209; /127.0.0.1:56214
进程已结束, 退出代码为 130
```

第二个客户端, 打印三次

依次类推... ..

运行 TCPServer x hw8task2client x hw8task2client x hw8task2client x hw8task2client x

```
"C:\Program Files\Java\jdk-20\bin\java.exe" "-javaagent:E:\java_idea\IntelliJ IDEA 2023.3.4\lib\idea_rt.jar"
收到客户端信息: 现有所有连接客户端: /127.0.0.1:56200; /127.0.0.1:56206; /127.0.0.1:56209
收到客户端信息: 现有所有连接客户端: /127.0.0.1:56200; /127.0.0.1:56206; /127.0.0.1:56209; /127.0.0.1:56214
进程已结束, 退出代码为 130
```

运行 TCPServer x hw8task2client x hw8task2client x hw8task2client x hw8task2client x

```
"C:\Program Files\Java\jdk-20\bin\java.exe" "-javaagent:E:\java_idea\IntelliJ IDEA 2023.3.4\lib\idea_rt.jar"
收到客户端信息: 现有所有连接客户端: /127.0.0.1:56200; /127.0.0.1:56206; /127.0.0.1:56209; /127.0.0.1:56214
进程已结束, 退出代码为 130
```

➤ 出错反思:

- 一开始没有及时对遍历的 `clientsockets` 数组进行检查, 未将停止连接的 `clientsocket` 移出数组。这就会使当前连接的 `clientSockets` 数组并不准确, 发送的远程地址偏多。

Task3: 尝试运行 `NIOserver` 并运行 `TCPClient`, 观察 `TCPserver` 和 `NIOserver` 的不同之处, 并说明当有并发的1万个客户端(C10K)想要建立连接时, 在Lab7中实现的 `TCPserver` 可能会存在哪些问题。

储备知识:

一、Selector 类

- 作用: 属于 `NIO` 类, 用于多路复用非阻塞 I/O 操作, 允许一个线程监听多个通道的事件, 并在通道就绪时进行响应。
- 基于事件驱动: `Selector` 是事件驱动的, 当通道上发生感兴趣的事件时, `Selector` 会通知相应的线程进

行处理。

➤ 使用步骤及方法：

- 创建 Selector 对象：通过 `Selector.open()` 静态方法创建一个 Selector 对象。
- 将通道注册到 Selector：通过通道的 `register(SelectableChannel channel, int ops)` 方法将通道注册到 Selector，并指定所关注的事件类型，例如读 (`SelectionKey.OP_READ`)、写 (`SelectionKey.OP_WRITE`) 等。
- 轮询就绪通道：通过 `select()` 方法轮询已就绪的通道，当有通道就绪时，返回就绪通道的数量。
- 处理就绪事件：通过 `selectedKeys()` 方法获取已就绪的 `SelectionKey` 集合，遍历集合处理就绪事件。
- 关闭 Selector 对象：通过 `close()` 函数关闭。

二、ByteBuffer 类

➤ 作用：属于 NIO 类，用于在内存中存储字节数据并进行读写操作，同时实现缓冲区的概念，可以灵活地管理缓冲区的大小和状态，支持控制字节数据的存储顺序（大端序或小端序）

➤ 常用方法：

- 创建 ByteBuffer 对象：使用 `allocate()` 方法创建一个指定大小的缓冲区->读；使用 `wrap()` 方法将现有的字节数组包装为 ByteBuffer 对象->写。
- 读写操作：
 - ◆ 向缓冲区写入数据：
 - `put` 方法：ByteBuffer 类的成员方法，用于向缓冲区中写入数据，适用于控制写入数据类型和顺序的场景。
 - `buffer.put((byte) 65);` // 写入一个字节 `buffer.put(new byte[] {65, 66, 67});` // 写入一个字节数组
 - `write` 方法：是 `WritableByteChannel` 接口的方法，用于将缓冲区中的数据写入到通道中，适用于将数据发送给其他节点或写入到文件的场景。
 - 对于 `SocketChannel` 对象 `channel`，可以使用 `channel.write(buffer)` 方法将缓冲区中

的数据写入到该通道。

◆ 从缓冲区读取数据：

- get 方法：属于 ByteBuffer 类的成员方法，用于从缓冲区中读取数据，适用于控制读取数据类型和顺序的场景
- byte b = buffer.get(); // 读取一个字节 byte[] byteArray = new byte[10];
buffer.get(byteArray); // 读取多个字节到字节数组中
- read 方法：是 ReadableByteChannel 接口的方法，用于将数据从通道读取到缓冲区中，适用于从其他节点接收数据或从文件中读取数据的场景。
- 对于 SocketChannel 对象 channel，可以使用 **channel.read(buffer)** 方法将数据从该通道读取到缓冲区中。

■ 缓冲区状态控制：

- ◆ **flip()：将缓冲区从写模式切换为读模式，重置位置并设置界限。**
- ◆ **rewind()：将缓冲区的位置重置为 0，保持限制不变，用于重新读取已经写入的数据。**
- ◆ **clear()：清空缓冲区，重置位置和限制，但不会清除数据，数据仍然存在但处于可被覆写状态。**
- ◆ **compact()：压缩缓冲区，将未读取的数据移到缓冲区的起始位置，重置位置和限制，用于读取后继续写入数据。**

■ 其他常用方法：

- ◆ **order()：设置字节顺序。buffer.order(ByteOrder.BIG_ENDIAN); // 设置为大端序**
buffer.order(ByteOrder.LITTLE_ENDIAN); // 设置为小端序
- ◆ **hasRemaining()：检查缓冲区是否还有剩余的未读取数据。**
- ◆ **remaining()：获取缓冲区剩余可读取字节数。**
- ◆ **array()：获取缓冲区支持的字节数组。**

三、Channel 类

- 作用：属于 NIO 类的抽象类，提供了统一的接口，可以通过相同的方式进行对文件、套接字等实体的读写操作；实现非阻塞式的 I/O 操作，使得一个线程可以同时处理多个通道上的 I/O 操作；**Channel 类**

与 **Buffer** 类配合使用，支持数据从缓冲区传输到通道，或者从通道传输到缓冲区。

➤ 主要子类：

- **FileChannel**：用于对文件进行读写操作的通道，提供了读取、写入、映射文件到内存等功能。
- **SocketChannel**：用于 **TCP** 网络套接字的通道，提供了连接、读取、写入等功能。
- **ServerSocketChannel**：用于监听 **TCP** 网络套接字的通道，提供了接受连接的功能。
- **DatagramChannel**：用于 **UDP** 网络套接字的通道，提供了连接、发送、接收数据包等功能。

➤ **主要方法（重要!!!）：**

- **open()**：静态方法，用于打开一个新的通道。
- **close()**：关闭通道，释放相关资源。
- **read(ByteBuffer dst)**：从通道读取数据到缓冲区。
- **write(ByteBuffer src)**：将数据从缓冲区写入到通道。
- **configureBlocking(boolean block)**：设置通道的阻塞模式，可以设置为阻塞或非阻塞模式。
- **register(Selector sel, int ops)**：将通道注册到选择器，并指定感兴趣的事件类型，例如读、写、连接等。
- **connect(SocketAddress remote)**：连接到指定的远程地址。
- **accept()** (**ServerSocketChannel** 特有)：接受连接请求，返回一个新的 **SocketChannel** 对象。

四、*iterator* 迭代器

➤ 在 task2\3\4\5 中，都用到了 *iterator* 迭代器。

➤ 作用：*iterator* 提供了一种统一的方式来遍历集合（*List*、*Set*、*Map* 等）中的元素，而不需要暴露集合的内部结构，不会出现由于在遍历过程中修改集合而导致的并发修改异常，提供了 *remove()* 方法，允许在遍历集合的同时安全地删除元素，而不会影响到其他元素的遍历过程。

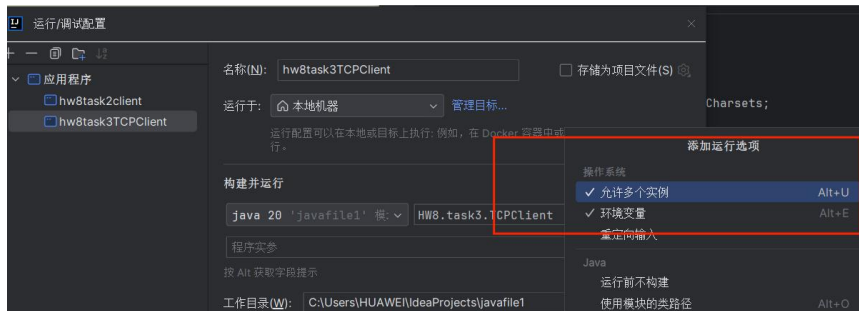
➤ 使用方法：

- 获取迭代器：调用集合类的 *iterator()* 方法来获取一个迭代器实例。集合.*iterator()* 获取一个该集合实例。
- 遍历集合：*hasNext()*：检查是否还有元素可以遍历；*next()*：获取下一个元素。

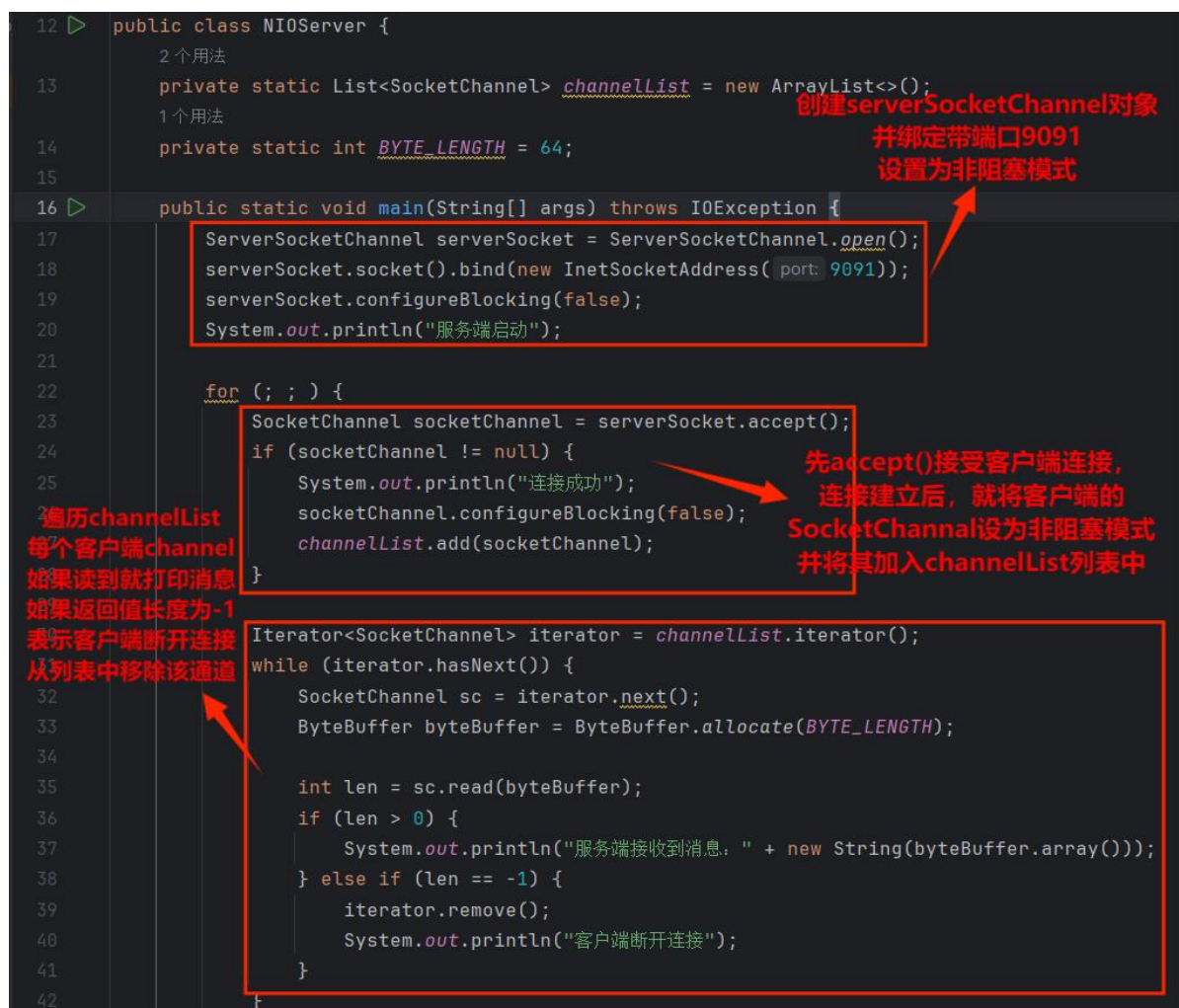
- 删除元素：使用迭代器遍历集合时，调用 `iterator.remove()` 方法来删除迭代器返回的当前元素。

运行代码流程&结果截图：

- 操作流程：让多个客户端可以同时运行

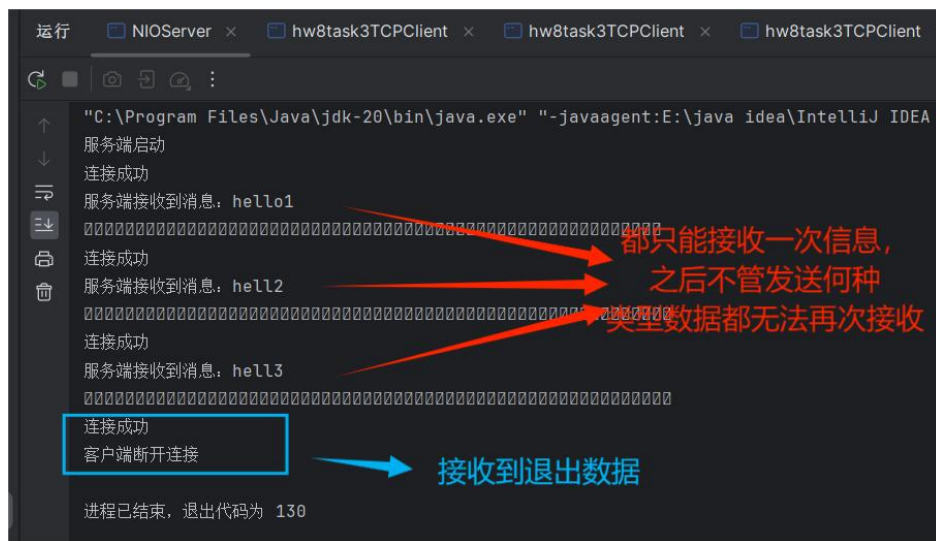


- NIOServer 代码解读：

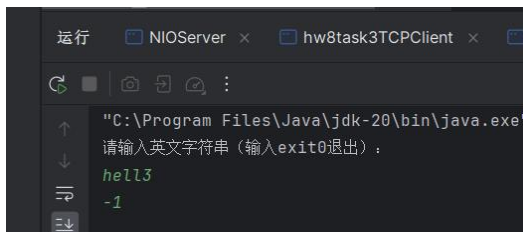
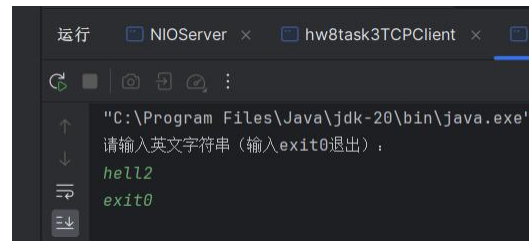
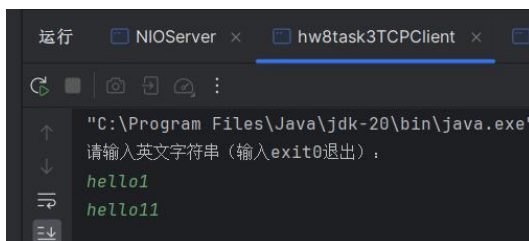


- NIOServer 和 TCPCClient 运行结果：

服务端：



客户端：都只有第一次发送数据成功，必须手动退出执行。



运行结果分析：

服务器代码在每次接收到客户端的消息后，**只会发送一次响应**，并在发送完响应后**关闭了与客户端的连接**。因此，客户端只能与服务器进行一次通信，之后再发送消息服务器就无法接收到了。

问题回答：

➤ NIOServer 和 TCPServer 的区别：

本质是**阻塞式 I/O 模型**和**非阻塞式 I/O 模型**之间的区别：

■ 区别：

- ◆ TCPServer->阻塞式 I/O 模型；NIOServer->非阻塞式 I/O 模型

- ◆ TCPServer: 在 ServerSocket 的 accept() 方法中, 当没有客户端连接时, 程序会一直阻塞, 直到有客户端连接进来; NIOServer: 在循环中, 不断尝试接受客户端的连接请求, 但即使没有连接进来, 程序也不会被阻塞。
 - ◆ TCPServer: 当有新的客户端连接时, 会创建一个新的 Socket 对象来处理与该客户端的通信; NIOServer: 当有新的客户端连接时, 会将其添加到一个 List 中, 并在循环中处理读取数据和客户端断开连接的情况。
 - ◆ TCPServer: 每个客户端连接**都会在一个独立的线程中处理**, 这样可以同时处理多个客户端的请求; NIOServer: 程序不需要为每个客户端连接创建一个新的线程, 而是使用**单线程处理多个连接**, 通过非阻塞 I/O 的特性实现多路复用。
- 题干情境下, Lab7 中实现的 TCPServer 可能会存在的问题:
- 会使性能下降、资源告竭, 甚至会引发系统崩溃。具体以下几点:**
- 线程资源消耗: 它会为每个客户端连接创建一个新的线程, 消耗大量的系统资源, 包括内存和 CPU 资源。
 - ◆ 后果: 当连接数量增加时, 系统需要为每个连接创建一个线程, 线程数量迅速增加, 可能会导致系统资源不足, 甚至引发系统崩溃。
 - ◆ 假设有 1 万个客户端同时连接, 那么 TCPServer 将会创建 1 万个线程来处理这些连接, 大量线程的创建将消耗大量的系统资源, 可能会导致系统变得非常缓慢或不可用。
 - 线程管理开销: 线程的创建、销毁和切换都会带来一定的开销, 当线程数量巨大时, 线程管理开销将会非常大。
 - ◆ 后果: 线程管理开销可能成为系统的性能瓶颈, 降低系统的整体性能。
 - ◆ 每个线程的创建、销毁和切换都需要消耗一定的时间和资源, 当线程数量巨大时, 这些开销会迅速增加, 可能会导致系统的响应变慢, 降低系统的性能。
 - 竞争和阻塞: 大量的线程可能会因为竞争系统资源而发生阻塞, 导致系统响应变慢甚至失去响应。
 - ◆ 后果: 竞争和阻塞会导致系统的性能下降, 影响服务的可用性和性能。
 - ◆ 当多个线程同时竞争某些系统资源 (如 CPU、内存、网络等), 可能会发生阻塞, 导致系统无

法及时响应客户端的请求，造成服务不可用或响应延迟。

- 内存占用：每个线程都会占用一定的内存空间，当线程数量巨大时，会消耗大量的内存资源。
 - ◆ 后果：大量的线程可能会导致内存资源不足，甚至引发内存溢出。
 - ◆ 每个线程都需要一定的内存空间来存储线程的上下文信息，当线程数量增加时，将会消耗大量的内存资源，可能会导致系统的内存资源不足，无法满足系统的需求，导致内存溢出错误。
- 上下文切换：大量的线程会频繁地进行上下文切换，增加了系统的负载，降低了系统的效率。
 - ◆ 后果：频繁的上下文切换会增加系统的负载，降低系统的效率，可能会导致系统的响应时间增加。
 - ◆ 当系统中有大量的线程需要进行上下文切换时，操作系统需要不断地保存和恢复线程的上下文信息，增加了系统的负载，可能会导致系统的性能下降，响应时间变长。

Task4: 尝试运行上面提供的NIO Server，试猜测该代码中的I/O多路复用调用了你操作系统中的哪些API，并给出理由。

➤ NIO Server 代码解读：

```
private void startServer() throws IOException {
    this.selector = Selector.open();
    ServerSocketChannel serverSocket = ServerSocketChannel.open();
    serverSocket.socket().bind(new InetSocketAddress(port: 9091));
    serverSocket.configureBlocking(false);
    serverSocket.register(this.selector, SelectionKey.OP_ACCEPT);
    System.out.println("服务端已启动");

    for (; ; ) {
        int readyCount = selector.select();
        if (readyCount == 0) {
            continue;
        }

        Set<SelectionKey> readyKeys = selector.selectedKeys();
        Iterator iterator = readyKeys.iterator();
        while (iterator.hasNext()) {
            SelectionKey key = (SelectionKey) iterator.next();
            iterator.remove();

            if (!key.isValid()) {
                continue;
            }

            if (key.isAcceptable()) {
                this.accept(key);
            } else if (key.isReadable()) {
                this.read(key);
            } else if (key.isWritable()) {
            }
        }
    }
}
```

监听事件，是否有事件准备就绪

创建一个Selector对象
将ServerSocketChannel注册到Selector中，并指定关注为接受连接事件

遍历处理准备就绪的键集合：
调用accept、read、write方法

```
1 个用法
private void accept(SelectionKey key) throws IOException {
    ServerSocketChannel serverChannel = (ServerSocketChannel) key.channel();
    SocketChannel channel = serverChannel.accept();
    channel.configureBlocking(false);
    Socket socket = channel.socket();
    SocketAddress remoteAddr = socket.getRemoteSocketAddress();
    System.out.println("已连接: " + remoteAddr);
    // 监听读事件
    channel.register(this.selector, SelectionKey.OP_READ);
}

1 个用法
private void read(SelectionKey key) throws IOException {
    SocketChannel channel = (SocketChannel) key.channel();
    ByteBuffer buffer = ByteBuffer.allocate(BYTE_LENGTH);
    int numRead = -1;
    numRead = channel.read(buffer);
    if (numRead == -1) {
        Socket socket = channel.socket();
        SocketAddress remoteAddr = socket.getRemoteSocketAddress();
        System.out.println("连接关闭: " + remoteAddr);
        channel.close();
        key.cancel();
        return;
    }
    byte[] data = new byte[numRead];
    System.arraycopy(buffer.array(), 0, data, 0, numRead);
    System.out.println("服务端已收到消息: " + new String(data));
}
```

接受客户端连接:
先从键中获取通道
调用accept接受连接
并设置为非阻塞,
接着将通道注册到
Selector中,并指定
关注事件为读取事件

从用户端读取数据: 从键中
获取通道,并创建ByteBuffer
接着调用通道类read方法读取
读取字节数不为-1时,将读
到数据转化为字节数组并打印

➤ 运行结果:

- 服务端: 还是只能收到同一个客户端发来的第一条消息

```
运行 NIOServer x hw8task3TCPCli
"C:\Program Files\Java\jdk-20\bin\
服务端已启动
已连接: /127.0.0.1:52278
服务端已收到消息: hell1
已连接: /127.0.0.1:52284
连接关闭: /127.0.0.1:52284
```

- 客户端: 第一条发送 exit0 的能够正常退出, 其他需要手动停止

```
运行 NIOServer x hw8task3TCPCli
"C:\Program Files\Java\jdk-20\bin\
请输入英文字符串 (输入exit0退出):
hell1
exit0
```

```
运行 NIOServer x hw8task3TCPCli
"C:\Program Files\Java\jdk-20\bin\
请输入英文字符串 (输入exit0退出):
exit0
进程已结束, 退出代码为 0
```

- I/O 多路复用：一种高效处理多个 I/O 事件的机制，使得一个进程能够同时监控多个 I/O 流的状态，从而在有 I/O 事件发生时进行相应的处理，而不需要为每个 I/O 流创建一个对应的线程或进程，提高系统的并发性能。
 - 传统的阻塞 I/O 模型：当一个 I/O 操作阻塞时，程序会一直等待直到该 I/O 操作完成。这种模型在处理多个 I/O 流时会面临效率问题，因为每个 I/O 操作都需要一个线程来处理，如果有大量的 I/O 操作，就会导致大量线程的创建和上下文切换，从而降低系统的性能。
 - I/O 多路复用：通过操作系统提供的机制（如 select、poll、epoll 等）实现了同时监控多个 I/O 流的状态。当有 I/O 事件发生时，程序可以从操作系统获取通知，然后针对发生事件的 I/O 流进行相应的处理，而无需为每个 I/O 流创建一个线程。这样就能够有效地减少线程的创建和上下文切换，提高系统的性能和并发能力。
- I/O 多路复用调用的 API：

这段 NIO Server 代码中，使用了 Java 的 NIO（New I/O）的 Selector、ServerSocketChannel、SocketChannel 等类，当 NIO 在 Windows 系统下运行时，它通过 JNI（允许 Java 代码调用本地（即操作系统特定）的代码）与 Windows API 进行交互。

- **我正在使用的 windows 系统：**

- ◆ select：代码中的 NIO Selector 类在底层使用 Windows 的 select 函数实现多路复用，select 函数允许一个进程同时监听多个套接字的状态，并在套接字就绪时进行相应的操作。
- ◆ WSAPoll：NIO 可能会使用 Windows 系统中的 Sockets API 的 WSAPoll 函数实现类似于 Linux 中的 poll 函数的功能。
- ◆ IOCP（I/O Completion Ports）：对于高性能的多路复用 I/O 模型，Java NIO 可能会使用 Windows 的 IOCP 机制。该机制允许一个进程同时处理多个 I/O 操作，并在完成时通知相应的处理程序。与 IOCP 相关的函数：CreateIoCompletionPort() 创建一个 I/O 完成端口对象、WSAEventSelect() 将套接字与事件对象关联，以便异步 I/O 操作完成时触发事件。

- Pdf 科普文章提到的 Linux 系统：

- ◆ select：Linux 中最早引入的多路复用 I/O 模型，允许一个进程监视多个文件描述符的状态，当其中某个文件描述符就绪时，该进程可以进行相应的 I/O 操作。

- 使用 `select` 需要手动维护文件描述符集合，调用 `select` 函数后会阻塞，直到有文件描述符就绪或超时。
- ◆ `poll`: 是 `select` 的改进版本，同样允许一个进程监视多个文件描述符的状态，但不再受到文件描述符数量的限制。
 - 使用时需要创建一个 `pollfd` 数组来维护文件描述符及其关注的事件，调用 `poll` 函数后会阻塞，直到有文件描述符就绪或超时。
- ◆ `epoll`: Linux 2.6 内核引入的高性能多路复用 I/O 模型，在处理大量连接时有着更高的性能。
 - 使用 `epoll` 时需要创建一个 `epoll` 实例，将文件描述符注册到 `epoll` 实例中，并设置关注的事件类型。调用 `epoll_wait` 函数后会阻塞，直到有文件描述符就绪或超时。Linux 下的一种高性能 I/O 多路复用机制，用于监听文件描述符的事件。
 - `epoll_create()`、`epoll_ctl()`、`epoll_wait()` 等 `epoll` 系列函数：`epoll_create()` 创建一个 `epoll` 实例，返回一个文件描述符。`epoll_ctl()` 向 `epoll` 实例中添加、修改或删除需要监听的文件描述符和事件。`epoll_wait()` 等待事件的发生，一旦有文件描述符上的事件发生，该函数就会返回，并返回就绪的文件描述符列表。
- macOS:
 - ◆ `kqueue`: 类似于 Linux 下的 `epoll`，允许一个进程监视多个文件描述符的状态，并在文件描述符就绪时进行相应的操作。
 - 使用 `kqueue` 需要创建一个 `kqueue` 实例，将文件描述符注册到 `kqueue` 实例中，并设置关注的事件类型。调用 `kevent` 函数后会阻塞，直到有文件描述符就绪或超时。

Task5 (Bonus): 编写基于NIO的NIOClient，当监听到和服务端建立连接后向服务端发送"Hello Server"，当监听到可读时将服务端发送的消息打印在控制台中。（自行补全NIOServer消息回写）

代码截图&运行结果:

➤ NIOServer 补全部分:

- 首先，在 `read` 函数最后，服务端收到信息后，调用 `write` 函数，向客户端回写服务端收到的信息
- 其次，添加一个 `write` 函数，负责服务端向客户端回写消息，以 `SelectionKey` 和字符串为参数，

```

85     String str=new String(data);
86     System.out.println("服务器已收到信息: "+str);
87
88     write(key, msg: "该消息已被服务器接收: "+str);
89 }
90
91 1个用法
92 @ private void write(SelectionKey key,String msg)throws IOException{
93     SocketChannel channel=(SocketChannel) key.channel();
94     ByteBuffer buffer=ByteBuffer.wrap(msg.getBytes());
95     channel.write(buffer);
96     System.out.println("服务器发送消息: "+msg);
97 }

```

➤ NIOClient 代码:

```

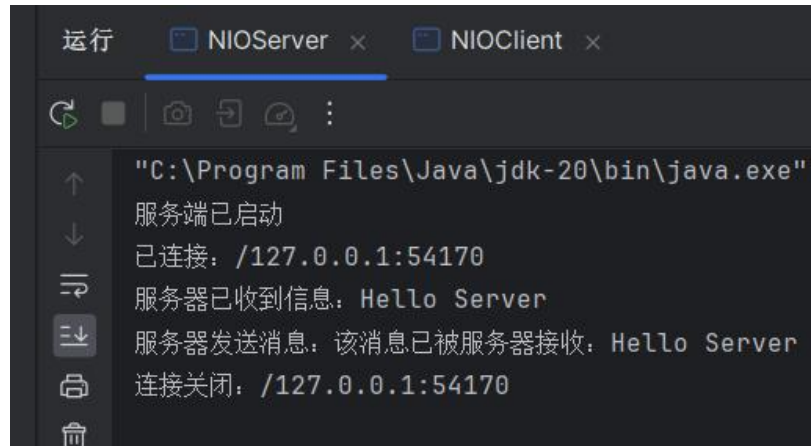
3 import java.io.IOException;
4 import java.net.InetSocketAddress;
5 import java.nio.ByteBuffer;
6 import java.nio.channels.SocketChannel;
7
8 public class NIOClient {
9     1个用法
10    private static int BYTE_LENGTH=64;
11    public static void main(String[] args) {
12        try {
13            SocketChannel socketChannel=SocketChannel.open();
14            socketChannel.connect(new InetSocketAddress( hostname: "127.0.0.1", port: 9091));
15            System.out.println("客户端已连接至服务器");
16
17            sendMessage(socketChannel, msg: "Hello Server");
18            receiveMessage(socketChannel);
19
20            socketChannel.close();
21        }catch (IOException e){
22            e.printStackTrace();
23        }
24    }
25    1个用法
26    @ private static void sendMessage(SocketChannel socketChannel,String msg)throws IOException{
27        ByteBuffer buffer=ByteBuffer.wrap(msg.getBytes());
28        socketChannel.write(buffer);
29        System.out.println("客户端发送消息: "+msg);
30    }
31
32    @ private static void receiveMessage(SocketChannel socketChannel)throws IOException{
33        ByteBuffer buffer=ByteBuffer.allocate(BYTE_LENGTH);
34        socketChannel.read(buffer);
35        buffer.flip();
36        byte[] bytes=new byte[buffer.remaining()];
37        buffer.get(bytes);
38        String msg=new String(bytes);
39        System.out.println("客户端收到消息: "+msg);
40    }
41 }

```


➤ 运行结果：

- NIOServer：能够接收服务端发来的 Hello Server 信息；并向服务端回写：该消息已被服务器接收：

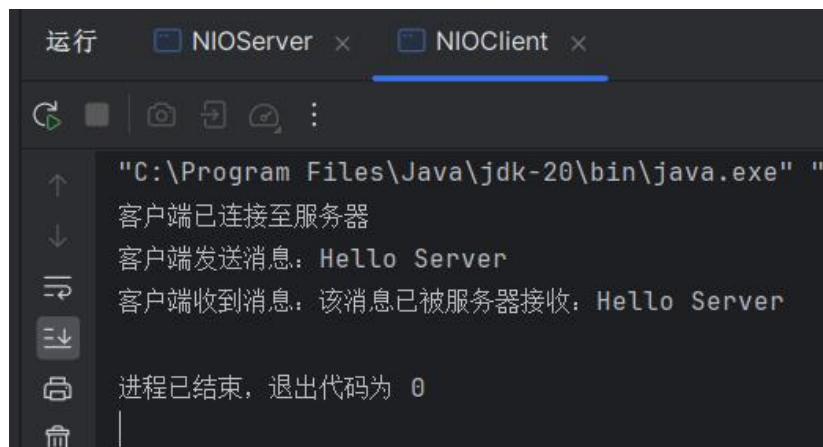
Hello Server



The screenshot shows the NIOServer console window with the following output:

```
"C:\Program Files\Java\jdk-20\bin\java.exe"  
服务端已启动  
已连接: /127.0.0.1:54170  
服务器已收到信息: Hello Server  
服务器发送消息: 该消息已被服务器接收: Hello Server  
连接关闭: /127.0.0.1:54170
```

- NIOClient：连接后就向服务端发送 Hello Server，并且能及时收到服务端回写来的消息，打印到控制台：该消息已被服务器接收：Hello Server



The screenshot shows the NIOClient console window with the following output:

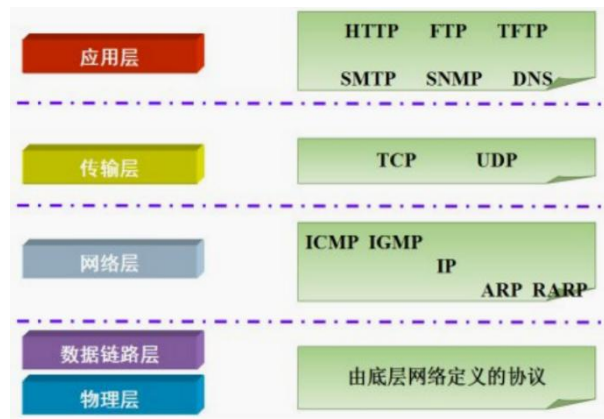
```
"C:\Program Files\Java\jdk-20\bin\java.exe" "  
客户端已连接至服务器  
客户端发送消息: Hello Server  
客户端收到消息: 该消息已被服务器接收: Hello Server  
进程已结束, 退出代码为 0
```

三、总结

拓展知识：

一、网络编程要素

- 三要素：协议+IP 地址+端口号
- 协议：



➤ IP 地址：唯一标识网络中的设备

- IPv4：32 位的二进制数，通常被分为 4 个字节，表示成 a.b.c.d 的形式，a、b、c、d 都是 0~255 之间的十进制整数。IPv6：

- IP 相关命令（cmd）：

◆ ipconfig：查看本机 IP 地址

```
C:\Users\HUAWEI>ipconfig

Windows IP 配置

无线局域网适配器 本地连接* 1:

    媒体状态 . . . . . : 媒体已断开连接
    连接特定的 DNS 后缀 . . . . . :

无线局域网适配器 本地连接* 2:

    媒体状态 . . . . . : 媒体已断开连接
    连接特定的 DNS 后缀 . . . . . :

以太网适配器 VMware Network Adapter VMnet1:

    连接特定的 DNS 后缀 . . . . . :
```

◆ ping+空格+IP 地址：查看网络是否连接

```
C:\Users\HUAWEI>ping 172.31.75.131
ping+空格+IP地址

正在 Ping 172.31.75.131 具有 32 字节的数据:
来自 172.31.75.131 的回复: 字节=32 时间<1ms TTL=128
来自 172.31.75.131 的回复: 字节=32 时间<1ms TTL=128
来自 172.31.75.131 的回复: 字节=32 时间<1ms TTL=128
来自 172.31.75.131 的回复: 字节=32 时间<1ms TTL=128

172.31.75.131 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
    往返行程的估计时间(以毫秒为单位):
        最短 = 0ms, 最长 = 0ms, 平均 = 0ms
```

- 特殊 IP 地址：127.0.0.1、localhost 都用来表示本机 IP 地址，可以用来本地回环测试和开发环境中的本地通信。

➤ 端口号：唯一标识设备中的进程（应用程序）

- 用两个字节表示的整数，它的取值范围是 0~65535。其中，0~1023 之间的端口号用于一些知名的网络服务和应用，普通的应用程序需要使用 1024 以上的端口号。
- 如果端口号被另外一个服务或应用所占用，会导致当前程序启动失败。

总结:

在本次实验中，我们学习了基于 TCP 的 Socket 编程，并进行了一系列的优化和扩展：

1. TCP Socket 编程：首先，我学会了如何使用 Java 中的 Socket 和 ServerSocket 类来实现基本的 TCP 客户端和服务端通信。了解了 TCP 连接的建立、数据传输和连接关闭的基本流程，以及如何处理异常和错误情况。
2. 多线程优化：在基础代码上，我对服务端进行了多线程优化，使得每次接受到客户端连接时，都启动一个新的线程来处理该连接，避免了阻塞其他连接的情况，提高了服务器的并发处理能力。
3. 半包粘包问题：又对半包粘包问题的产生原因和解决方案进行了研究，以便于接收端正确解析数据。
4. 并行发送和接收优化：在后来的优化实验部分，进一步优化了客户端的实现，使其能够实现发送和接收的并行处理，从而提高了通信效率和响应速度。
5. NIO 和 TCP 的对比：最后，能够使用 NIO 实现服务端，探究了其与传统 Socket 的实现之间的区别。并进一步实现了 NIOClient 的实现，并且进一步深入了解了 I/O 多路复用相关的知识点。

通过本次实验，我理解了 TCPSocket 编程的原理和技术，学会了如何优化和扩展基本的 Socket 通信，提高了我对网络编程的理解和应用能力。