

# 华东师范大学数据科学与工程学院实验报告

课程名称：计算机网络与编程	年级：21级	上机实践成绩：
指导教师：张召	姓名：包亦晟	学号：10215501451
上机实践名称：基于TCP的Socket编程优化	上机实践日期：2023.4.21	
上机实践编号：08	组号：1-451	上机实践时间：2023.4.21

## 一、实验目的

- 对数据发送和接收进行优化
- 实现信息共享
- 熟悉阻塞I/O与非阻塞I/O

## 二、实验任务

- 将数据发送与接收并行，实现全双工通信
- 实现服务端向所有客户端广播消息
- 了解非阻塞I/O

## 三、使用环境

- IntelliJ IDEA
- JDK 版本: Java 19

## 四、实验过程

### 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。

任务当中要求**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**，这意味着我们还要增加ServerReadHandler类和ServerWriteHandler类作为处理从服务器读数据的线程和向服务器写数据的线程。

ServerReadHandler类

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.net.Socket;
import java.nio.charset.StandardCharsets;
```

```

public class ServerReadHandler extends Thread{
    private final BufferedReader bufferedReader;
    ServerReadHandler(InputStream inputStream) {
        this.bufferedReader = new BufferedReader(new
InputStreamReader(inputStream,
        StandardCharsets.UTF_8));
    }

    public void run() {
        String str;
        while (true) {
            try {
                str = bufferedReader.readLine();
                System.out.println("客户端接收服务器发送的消息: " + str);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}

```

ServerWriteHandler类

```

import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.nio.charset.StandardCharsets;
import java.util.Scanner;

public class ServerWriteHandler extends Thread{
    private final PrintWriter printWriter;
    private final Scanner sc;
    ServerWriteHandler(OutputStream outputStream) {
        this.printWriter = new PrintWriter(new OutputStreamWriter(outputStream,
        StandardCharsets.UTF_8), true);
        this.sc = new Scanner(System.in);
    }

    public void run() {
        String str;
        while (true) {
            str = sc.nextLine();
            printWriter.println(str);
            printWriter.flush();
        }
    }
}

```

模仿实验讲义中用ClientHandler整合ClientReadHandler和ClientWriteHandler的思路，我也添加了一个ServerHandler类：

```

import java.io.IOException;

```

```

import java.net.Socket;

public class ServerHandler extends Thread{
    private Socket socket;
    private final ServerReadHandler serverReadHandler;
    private final ServerWriteHandler serverWriteHandler;
    ServerHandler(Socket socket) throws IOException {
        this.socket = socket;
        this.serverReadHandler = new ServerReadHandler(socket.getInputStream());
        this.serverWriteHandler = new
ServerWriteHandler(socket.getOutputStream());
    }

    public void run() {
        super.run();
        serverReadHandler.start();
        serverWriteHandler.start();
    }
}

```

TCPClient类修改如下:

```

import java.io.*;
import java.net.Socket;
import java.nio.charset.StandardCharsets;
public class TCPClient {
    private Socket socket;
    public void start(String ip, int port) throws IOException {
        socket = new Socket(ip, port);
        ServerHandler sh = new ServerHandler(socket);
        sh.start();
    }
    public static void main(String[] args) {
        int port = 9091;
        TCPClient client = new TCPClient();
        try {
            client.start("127.0.0.1", port);
        }catch (IOException e){
            e.printStackTrace();
        }
    }
}

```

运行结果:

```
TCPServer x TCPClient x
E:\jdk-19.0.2\bin\java.exe "-javaagent:
阻塞等待客户端连接中...
hello,client
服务器接收客户端发送的消息: hello,server
服务器接收客户端发送的消息: nice to meet you
服务器接收客户端发送的消息: how are you
nice to meet you too
i am fine
thank you
```

```
TCPServer x TCPClient x
E:\jdk-19.0.2\bin\java.exe "-ja
客户端接收服务器发送的消息: hello,cl
hello,server
nice to meet you
how are you
客户端接收服务器发送的消息: nice
客户端接收服务器发送的消息: to
客户端接收服务器发送的消息: meet
客户端接收服务器发送的消息: you
客户端接收服务器发送的消息: too
客户端接收服务器发送的消息: i
客户端接收服务器发送的消息: am
客户端接收服务器发送的消息: fine
客户端接收服务器发送的消息: thank
客户端接收服务器发送的消息: you
```

我在完成task1的时候碰到的困难是类命名的理解问题。这里的Client和Server指的并不是类本身是客户端还是服务器，而是要处理的消息来自哪里。我在修改TCPClient中将ServerHandler写成了ClientHandler，出现了很大问题，直到最后仔细排查才发现这个问题。

## 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。

获取要发送的信息内容，我们需要记录下Socket；而要向当前所有已经建立连接的客户端发送消息，我们需要记录所有的ClientHandler线程。因此，我们要在服务器中建立两个数组分别存放Socket和ClientHandler。

```
private List<Socket> sockets = new ArrayList<Socket>();
private List<ClientHandler> handlers = new ArrayList<ClientHandler>();
```

服务器每次和客户端成功连接后就将Socket和ClientHandler放入对应的数组当中。

```
socket = serverSocket.accept();
System.out.println("阻塞等待客户端连接中...");
ClientHandler ch = new ClientHandler(socket);
sockets.add(socket);
handlers.add(ch);
ch.start();
```

由于发送的内容是所有的已经建立连接的Socket对象的getRemoteSocketAddress()所拼接起来的，字符串是要动态变化的，所以我们这里用StringBuilder，而不是String。最后再将StringBuilder转化为String。

```
public String getSocketAddresses() {
    StringBuilder sb = new StringBuilder();
    for (Socket socket : sockets) {
        sb.append(socket.getRemoteSocketAddress() + " ");
    }
    return sb.toString();
}
```

最后就是发送信息了，这里我们如果细看ClientWriteHandler类的话，我们可以看到一个send方法。所以，我们很自然地就可以想到在ClientHandler内部添加一个send方法，调用其成员ClientWriteHandler的send方法。

```
public void send(String msg) {
    this.clientWriteHandler.send(msg);
}
```

```
for (ClientHandler handler : handlers) {
    handler.send(msg);
}
```

完整的服务器的start部分如下：

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

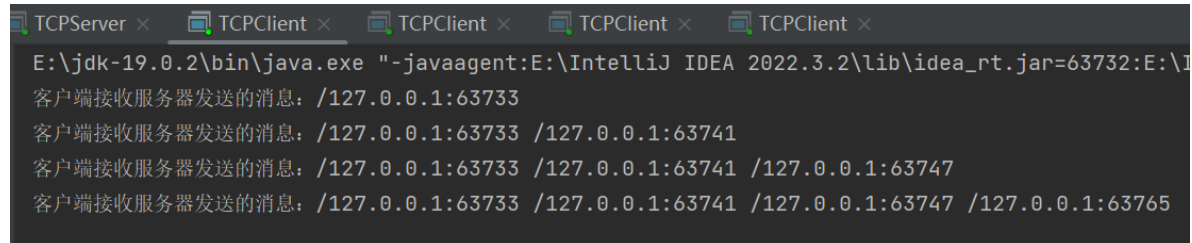
```

        sockets.add(socket);
        handlers.add(ch);
        ch.start();
        String msg = getSocketAddresses();
        for (ClientHandler handler : handlers) {
            handler.send(msg);
        }
    }
}

```

运行结果:

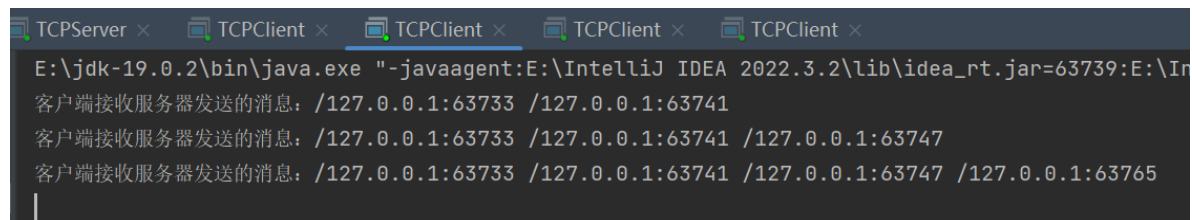
建立四个客户端:



```

TCPClient x TCPClient x TCPClient x TCPClient x
E:\jdk-19.0.2\bin\java.exe "-javaagent:E:\IntelliJ IDEA 2022.3.2\lib\idea_rt.jar=63732:E:\IntelliJ IDEA 2022.3.2\bin" -jar E:\IntelliJ IDEA 2022.3.2\bin\java.exe
客户端接收服务器发送的消息: /127.0.0.1:63733
客户端接收服务器发送的消息: /127.0.0.1:63733 /127.0.0.1:63741
客户端接收服务器发送的消息: /127.0.0.1:63733 /127.0.0.1:63741 /127.0.0.1:63747
客户端接收服务器发送的消息: /127.0.0.1:63733 /127.0.0.1:63741 /127.0.0.1:63747 /127.0.0.1:63765

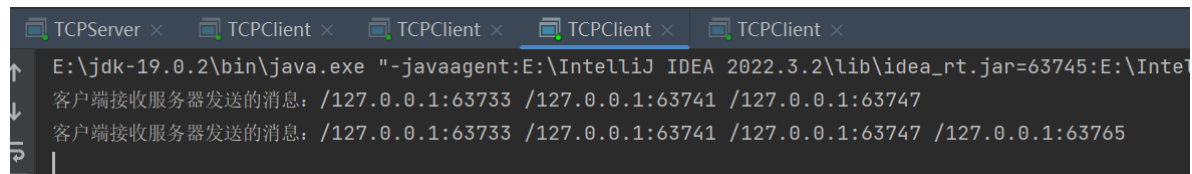
```



```

TCPClient x TCPClient x TCPClient x TCPClient x
E:\jdk-19.0.2\bin\java.exe "-javaagent:E:\IntelliJ IDEA 2022.3.2\lib\idea_rt.jar=63739:E:\IntelliJ IDEA 2022.3.2\bin" -jar E:\IntelliJ IDEA 2022.3.2\bin\java.exe
客户端接收服务器发送的消息: /127.0.0.1:63733 /127.0.0.1:63741
客户端接收服务器发送的消息: /127.0.0.1:63733 /127.0.0.1:63741 /127.0.0.1:63747
客户端接收服务器发送的消息: /127.0.0.1:63733 /127.0.0.1:63741 /127.0.0.1:63747 /127.0.0.1:63765

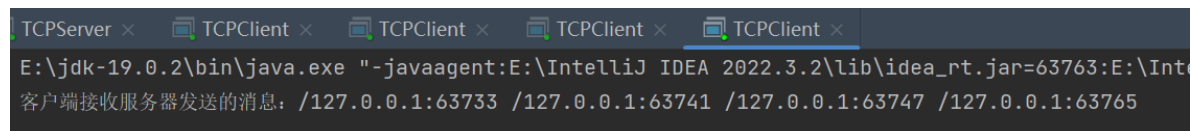
```



```

TCPClient x TCPClient x TCPClient x TCPClient x
E:\jdk-19.0.2\bin\java.exe "-javaagent:E:\IntelliJ IDEA 2022.3.2\lib\idea_rt.jar=63745:E:\IntelliJ IDEA 2022.3.2\bin" -jar E:\IntelliJ IDEA 2022.3.2\bin\java.exe
客户端接收服务器发送的消息: /127.0.0.1:63733 /127.0.0.1:63741 /127.0.0.1:63747
客户端接收服务器发送的消息: /127.0.0.1:63733 /127.0.0.1:63741 /127.0.0.1:63747 /127.0.0.1:63765

```

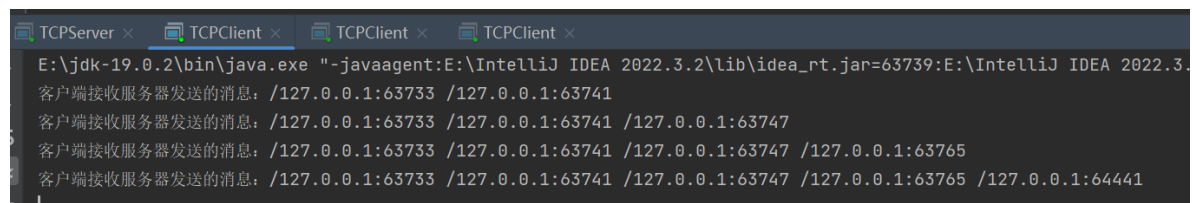


```

TCPClient x TCPClient x TCPClient x TCPClient x
E:\jdk-19.0.2\bin\java.exe "-javaagent:E:\IntelliJ IDEA 2022.3.2\lib\idea_rt.jar=63763:E:\IntelliJ IDEA 2022.3.2\bin" -jar E:\IntelliJ IDEA 2022.3.2\bin\java.exe
客户端接收服务器发送的消息: /127.0.0.1:63733 /127.0.0.1:63741 /127.0.0.1:63747 /127.0.0.1:63765

```

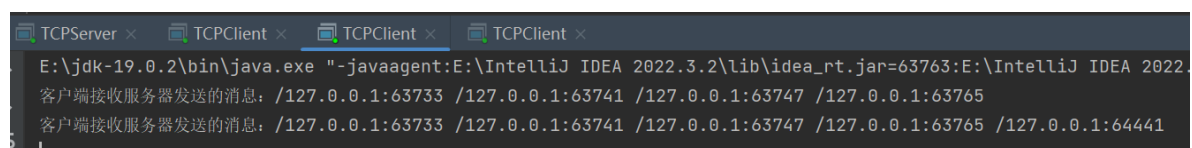
现在关闭第一个和第三个客户端, 并新建一个客户端。以下是剩下的客户端的截图:



```

TCPClient x TCPClient x TCPClient x
E:\jdk-19.0.2\bin\java.exe "-javaagent:E:\IntelliJ IDEA 2022.3.2\lib\idea_rt.jar=63739:E:\IntelliJ IDEA 2022.3.2\bin" -jar E:\IntelliJ IDEA 2022.3.2\bin\java.exe
客户端接收服务器发送的消息: /127.0.0.1:63733 /127.0.0.1:63741
客户端接收服务器发送的消息: /127.0.0.1:63733 /127.0.0.1:63741 /127.0.0.1:63747
客户端接收服务器发送的消息: /127.0.0.1:63733 /127.0.0.1:63741 /127.0.0.1:63747 /127.0.0.1:63765
客户端接收服务器发送的消息: /127.0.0.1:63733 /127.0.0.1:63741 /127.0.0.1:63747 /127.0.0.1:63765 /127.0.0.1:64441

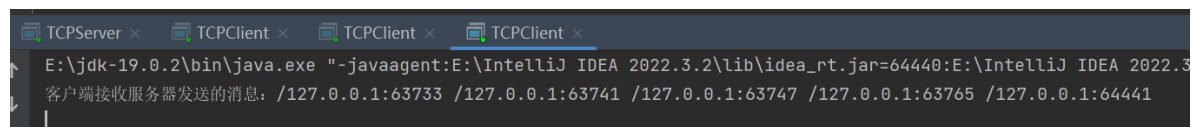
```



```

TCPClient x TCPClient x TCPClient x
E:\jdk-19.0.2\bin\java.exe "-javaagent:E:\IntelliJ IDEA 2022.3.2\lib\idea_rt.jar=63763:E:\IntelliJ IDEA 2022.3.2\bin" -jar E:\IntelliJ IDEA 2022.3.2\bin\java.exe
客户端接收服务器发送的消息: /127.0.0.1:63733 /127.0.0.1:63741 /127.0.0.1:63747 /127.0.0.1:63765
客户端接收服务器发送的消息: /127.0.0.1:63733 /127.0.0.1:63741 /127.0.0.1:63747 /127.0.0.1:63765 /127.0.0.1:64441

```



```

TCPClient x TCPClient x TCPClient x
E:\jdk-19.0.2\bin\java.exe "-javaagent:E:\IntelliJ IDEA 2022.3.2\lib\idea_rt.jar=64440:E:\IntelliJ IDEA 2022.3.2\bin" -jar E:\IntelliJ IDEA 2022.3.2\bin\java.exe
客户端接收服务器发送的消息: /127.0.0.1:63733 /127.0.0.1:63741 /127.0.0.1:63747 /127.0.0.1:63765 /127.0.0.1:64441

```

## task3

尝试运行NIOServer并运行TCPClient, 观察TCPClient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TCPClient可能会存在哪些问题。

在lab7中实现的TCPServer中有一个while循环，用来监听客户端的连接请求。每当有一个客户端连接请求到来时，就会创建一个ClientHandler线程来处理该客户端的请求。之所以采取这样的方式，是因为原本的TCPServer可能会因为accept而阻塞，只有创建另外的线程才能保证服务器可以并发地接受客户端连接并进行通信。而对于NIOserver来说，由于有了下面这行代码：

```
serverSocket.configureBlocking(false);
```

ServerSocketChannel就被设置为了非阻塞模式。在非阻塞模式下，当我们调用accept方法的时候，如果没有客户端连接请求到来，它就会立刻返回，不会产生阻塞。这样我们就可以在单个线程中处理多个客户端连接请求了。

当有并发的1万个客户端想要建立连接时，在lab7中实现的TCPServer就会创建上万个线程，服务器端的线程资源很容易就被消耗殆尽了。同时，每个线程都需要占用一定的内存资源，若创建的线程太多，TCPServer会导致内存资源耗尽。多个线程同时运行也会造成较大的调度压力。

## task4

尝试运行上面提供的NIOserver，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。

- Selector.open():使用epoll\_create系统调用，因为该方法会创建一个Selector对象，用于监听。而epoll\_create就是起到创建epoll句柄的作用。
- serverSocket.register()和channel.register(): 都使用了epoll\_ctl系统调用，因为这两个方法会将ServerSocketChannel和SocketChannel注册到Selector上，而这需要通过epoll\_ctl系统调用将文件描述符添加到epoll实例中。
- selector.select(): 使用epoll\_wait系统调用。因为该方法就是等待epoll\_wait返回来检测是否有I/O事件已经就绪。
- serverSocket.configureBlocking(false)和channel.configureBlocking(false): 使用fcntl系统调用，因为它们会分别将ServerSocketChannel和SocketChannel设置为非阻塞模式，而这需要通过fcntl系统调用设置文件描述符的属性来实现。
- channel.read(): 使用read系统调用。因为首先名字就一样，而且它的作用就是从Channel中读取数据。

## task5

编写基于NIO的NIOClient，当监听到和服务器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）

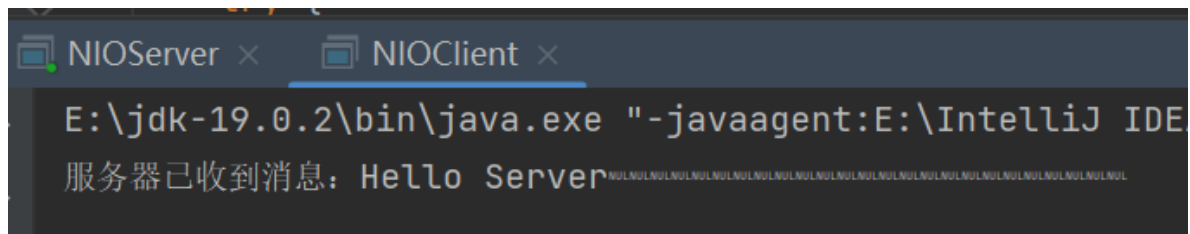
首先我们需要在NIOserver的read方法添加一条消息回写的代码：

```
channel.write(ByteBuffer.wrap(("服务器已收到消息: " + new  
String(data)).getBytes()));
```

接下来在写客户端向服务器端发送方"Hello Server"的时候，我写下了如下的代码：

```
ByteBuffer buffer = ByteBuffer.allocate(64);  
channel.read(buffer);  
System.out.println(new String(buffer.array()));
```

发现运行结果是这样的：



后来经过查证，原来还需要指定字节数组开始读取的起始位置，以及要读取的长度。要读取的长度是由read方法返回的，于是把代码改成下面这样：

```
ByteBuffer buffer = ByteBuffer.allocate(64);
int num = channel.read(buffer);
System.out.println(new String(buffer.array(),0,num,
    StandardCharsets.UTF_8));
```

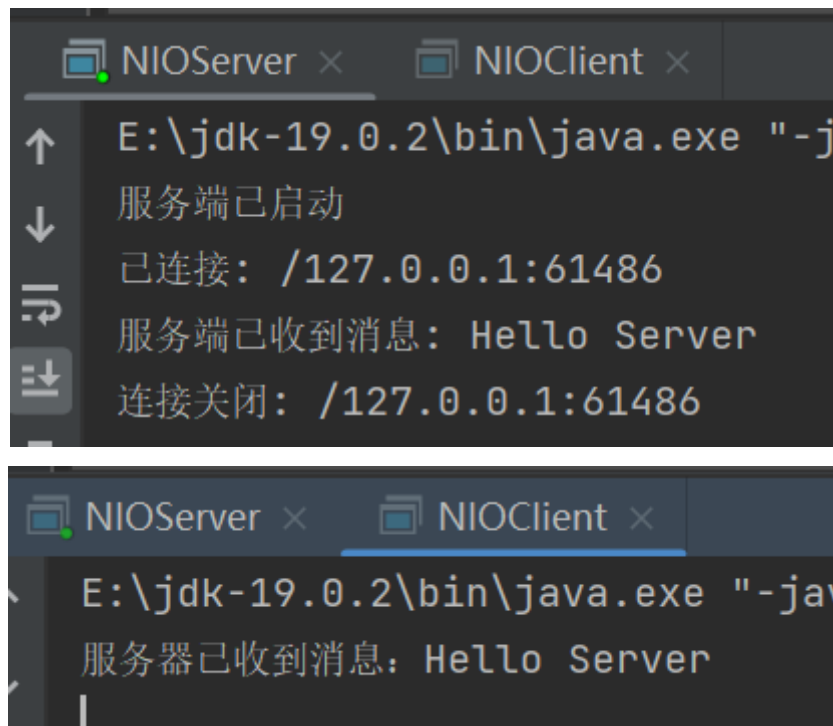
完整的TCPClient类如下：

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;
import java.nio.charset.StandardCharsets;

public class NIOClient {
    private SocketChannel channel;
    public void start(String ip, int port) throws IOException {
        this.channel = SocketChannel.open();
        channel.connect(new InetSocketAddress(ip,port));
        channel.write(ByteBuffer.wrap("Hello Server".getBytes()));
        ByteBuffer buffer = ByteBuffer.allocate(64);
        int num = channel.read(buffer);
        System.out.println(new String(buffer.array(),0,num,
            StandardCharsets.UTF_8));
        channel.close();
    }
    public static void main(String[] args) {
        NIOClient NIOclient = new NIOClient();
        try {
            NIOclient.start("127.0.0.1",9091);
        }catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

运行结果：





The image contains two screenshots of a Java IDE. The top screenshot shows the NIOServer console with the following output: `E:\jdk-19.0.2\bin\java.exe "-j`, `服务端已启动`, `已连接: /127.0.0.1:61486`, `服务端已收到消息: Hello Server`, and `连接关闭: /127.0.0.1:61486`. The bottom screenshot shows the NIOClient console with the output: `E:\jdk-19.0.2\bin\java.exe "-jav` and `服务器已收到消息: Hello Server`. Both screenshots have tabs for NIOServer and NIOClient at the top.

## 五、总结

在本次实验当中，我学习了对数据发送和接收进行优化，将数据发送与接收并行，实现了全双工通信；我也学习了消息共享，实现了服务端向所有客户端广播消息；我同时也初步了解了非阻塞I/O。

最初在task1上卡了很长很长的时间，事后才发现是自己对于发送和接收并行的理解存在问题，我们要实现的是全双工通信，发送方可以同时向接收方发送数据，而接收方也可以同时向发送方发送数据。为此，仅仅修改TCPClient类是不够的，还要模仿讲义中的内容，创建读取服务器信息和向服务器写信息的线程。

我一开始并没有仔细阅读讲义中给出的那篇公众号文章，导致后续的task不是很会做。再重新认真阅读并且理解之后，才顺利地完成了后续的task。