

## 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：2021 级

上机实践成绩：

指导教师：张召

姓名：彭一琄

学号：10215501412

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：8

组号：

上机实践时间：9:50

### 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。

本任务的目标是仿照 server 处理 client 信息的方式，在客户端创建两个新线程，分别收到服务器输入的信息并输出、将输入在客户端的控制台的信息发送给服务器。

首先，由于 TCPClient 类将 `PrintWriter`、`BufferedReader` 都定义在了类内部，而不是像代码中给出的 `ClientReadHandler`、`ClientWriteHandler` 一样，服务器需要给每个客户端分别定义一个输入流和输出流，而客户端只需要连接一个服务器，保存一对输入输出流即可。因此增加 `Scanner`、`ServerReadHandler`、`ServerWriteHandler` 等对象，方便在内部类中访问它们。

```
public class TCPClient {
    private Socket clientSocket;
    private PrintWriter out;
    private BufferedReader in;
    private Scanner sc;
    private ServerReadHandler serverReadHandler;
    private ServerWriteHandler serverWriteHandler;
    private ServerHandler serverHandler;
}
```

修改 startConnection 的代码，创建新对象

```
public void startConnection(String ip, int port) throws IOException {
    1. 创建客户端Socket, 指定服务器地址, 端口
    clientSocket = new Socket(ip, port);
    2. 获取输入输出流
    out = new PrintWriter(new OutputStreamWriter(clientSocket.getOutputStream(),
        StandardCharsets.UTF_8), autoFlush: true);
    in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream(),
        StandardCharsets.UTF_8));

    sc = new Scanner(System.in);
    serverReadHandler = new ServerReadHandler();
    serverWriteHandler = new ServerWriteHandler();
    serverHandler=new ServerHandler();
}
```

定义 ServerReadHandler 内部类，创建一个新线程，在循环中等待服务器数据，输出时通过“从服务器读到”来区分，每次读取以空格分割的一段数据，输出在控制台中。

```
class ServerReadHandler extends Thread {
    @Override
    public void run() {
        try {
            while (true) {
                拿到服务器一条数据, 显示在控制台中
                String resp = in.readLine();
                if (resp == null) {
                    System.out.println("从服务器读到的数据为空");
                    break;
                } else {
                    System.out.println("从服务器读到的数据为: " + resp);
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

定义内部类 `ServerWriteHandler`，创建一个新线程，在循环中不断读取客户端控制台的内容，发送给服务器。

```
class ServerWriteHandler extends Thread {
    //从控制台拿到数据，发送给服务器
    void send(String str){
        out.println(str);
    }
    @Override
    public void run() {
        while (sc.hasNext()) {
            String str = sc.next();
            send(str);
        }
    }
}
```

最后，将两个线程的行为汇总到另一个线程 `ServerHandler` 中，使主线程等待这两个子线程的结束。

```
class ServerHandler extends Thread {
    @Override
    public void run() {
        super.run();
        serverReadHandler.start();
        serverWriteHandler.start();
        try {
            serverReadHandler.join();
            serverWriteHandler.join();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }
}
```

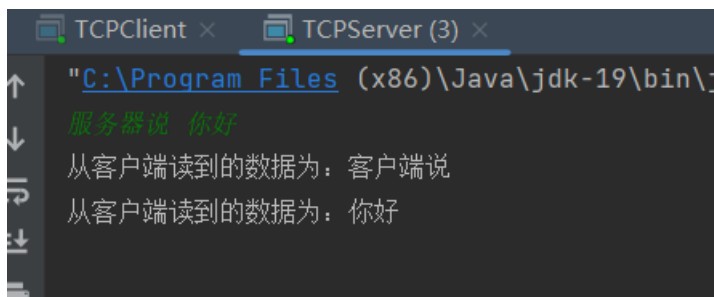
Main 函数执行过程中，也使主线程等待 handler 线程的执行，防止直接关闭客户端。

```

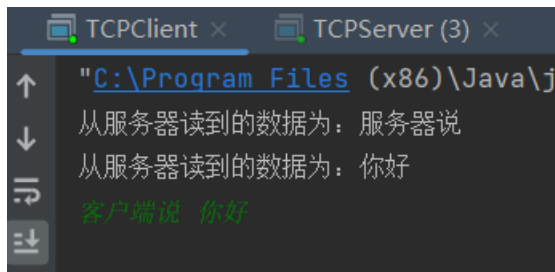
public static void main(String[] args) {
    int port = 9091;
    TCPClient client = new TCPClient();
    try {
        client.startConnection( ip: "127.0.0.1", port);
        client.serverHandler.start();
        client.serverHandler.join();
    } catch (IOException e){
        e.printStackTrace();
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    } finally {
        client.stopConnection();
    }
}

```

程序运行结果如下：



TCPClient x TCPServer (3) x  
 "C:\Program Files (x86)\Java\jdk-19\bin\j  
 服务器说 你好  
 从客户端读到的数据为：客户端说  
 从客户端读到的数据为：你好



TCPClient x TCPServer (3) x  
 "C:\Program Files (x86)\Java\j  
 从服务器读到的数据为：服务器说  
 从服务器读到的数据为：你好  
 客户端说 你好

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。

首先，给服务器设置一个列表，保存所有当前建立连接的 socket

```
private static final List<Socket> list=new ArrayList<>();
```

在每次接收到一个 Client 连接的时候，将 socket 加入到列表中

```
for (;;) {  
    Socket socket = serverSocket.accept();  
    SendToClient stc = new SendToClient(socket);  
    stc.start();  
    list.add(socket);  
}
```

创建 SendToClient 内部类（此时创建外部类也可以，只是为了方便访问 private 变量 list），用于将 list 里每个 socket 的信息发送给客户端。

首先，初始化一个 printWriter，记录此进程连接的客户端的 socket。

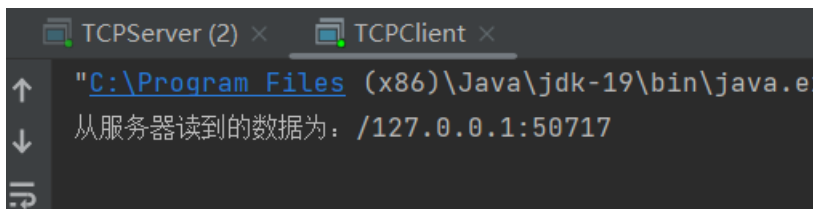
在运行过程中，调用 CheckClient 进程，然后进入一个忙等待，如果 list 的长度发生变化，就输出 list 中每个 socket 的 getRemoteSocketAddress 值，并更新长度 len

```
static class SendToClient extends Thread {  
    private final PrintWriter printWriter;  
    private final Socket socket;  
    private int len;  
    SendToClient(Socket socket) throws IOException{  
        this.socket=socket;  
        this.printWriter = new PrintWriter(new OutputStreamWriter(socket.getOutputStream(),  
            StandardCharsets.UTF_8), autoFlush: true);  
        this.len=0;  
    }  
    void send(String str) { this.printWriter.println(str); }  
    @Override  
    public void run() {  
        try {  
            CheckClient cc= new CheckClient(socket);  
            cc.start();  
        } catch (IOException e) {  
            throw new RuntimeException(e);  
        }  
        while(true){  
            synchronized (list){  
                if (list.size()!=len){  
                    for (Socket value : list) {  
                        send(value.getRemoteSocketAddress().toString());  
                    }  
                    len=list.size();  
                }  
            }  
        }  
    }  
}
```

CheckClient 类用于检测是否有客户端断开连接，通过读取客户端的输入流，看是否报错来判断。

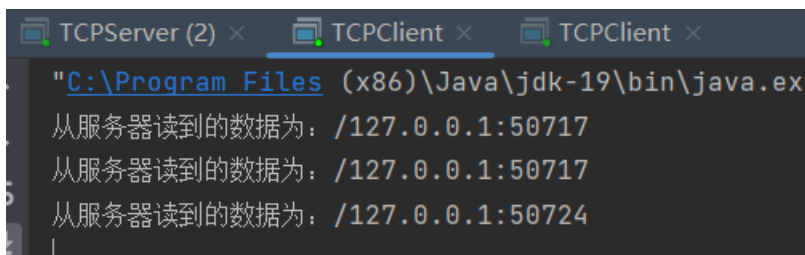
```
static class CheckClient extends Thread {
    private final BufferedReader bufferedReader;
    private final Socket socket;
    CheckClient(Socket socket) throws IOException {
        InputStream inputStream = socket.getInputStream();
        this.bufferedReader = new BufferedReader(new InputStreamReader(inputStream,
            StandardCharsets.UTF_8));
        this.socket=socket;
    }
    @Override
    public void run() {
        try {
            while (true) {
                bufferedReader.readLine();
            }
        } catch (IOException e) {
            synchronized (list){
                list.remove(socket);
            }
        }
    }
}
```

加入一个客户端，输出结果如下：

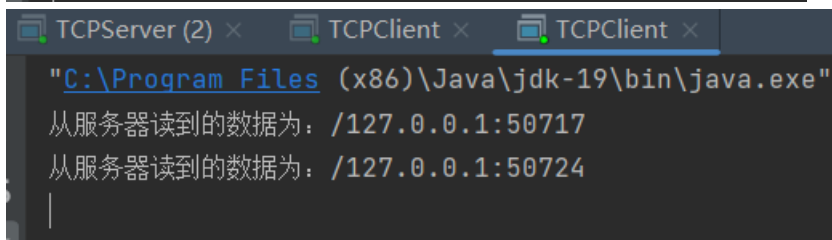


```
TCPClient x
"C:\Program Files (x86)\Java\jdk-19\bin\java.exe"
从服务器读到的数据为: /127.0.0.1:50717
```

再加入一个客户端，两个客户端分别输出结果如下：



```
TCPClient x TCPClient x
"C:\Program Files (x86)\Java\jdk-19\bin\java.exe"
从服务器读到的数据为: /127.0.0.1:50717
从服务器读到的数据为: /127.0.0.1:50717
从服务器读到的数据为: /127.0.0.1:50724
```



```
TCPClient x TCPClient x
"C:\Program Files (x86)\Java\jdk-19\bin\java.exe"
从服务器读到的数据为: /127.0.0.1:50717
从服务器读到的数据为: /127.0.0.1:50724
```

可以看到，第一个客户端连接的端口是 50717，第二个是 50724，而每当有一个新的客户端与服务器相连，服务器就会把所有客户端的信息都给每个客户端各传送一次。

删除第一个客户端 50717，输出结果如下：

```
TCPClient x
"C:\Program Files (x86)\Java\jdk-19\bin\java
从服务器读到的数据为: /127.0.0.1:50717
从服务器读到的数据为: /127.0.0.1:50724
从服务器读到的数据为: /127.0.0.1:50724
|
```

可以看到，服务器感应到了客户端不再连接，list 的值发生变化，并向第二个客户端发送了信息。

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。

运行两个 server，输出结果如下：

The screenshot displays three Java IDE windows, each showing the execution of a different part of a network program:

- Top Left Window (NIOServer):** The title bar indicates it is running. The console output shows:
  - Service endpoint started (服务端启动)
  - Connection successful (连接成功)
  - Connection successful (连接成功)
  - Server endpoint received message: 1 (服务端接收到消息: 1)
  - Server endpoint received message: 1 (服务端接收到消息: 1)
  - Server endpoint received message: 2 (服务端接收到消息: 2)
  - Server endpoint received message: 2 (服务端接收到消息: 2)
- Top Middle Window (TCPClient):** The title bar indicates it is running. The console output shows:
  - Connection successful (连接成功)
  - Connection successful (连接成功)
  - Client endpoint sent message: 1 (客户端发送消息: 1)
  - Client endpoint sent message: 1 (客户端发送消息: 1)
  - Client endpoint sent message: 2 (客户端发送消息: 2)
  - Client endpoint sent message: 2 (客户端发送消息: 2)
- Top Right Window (TCPClient):** The title bar indicates it is running. The console output shows:
  - Connection successful (连接成功)
  - Connection successful (连接成功)
  - Client endpoint sent message: 1 (客户端发送消息: 1)
  - Client endpoint sent message: 1 (客户端发送消息: 1)
  - Client endpoint sent message: 2 (客户端发送消息: 2)
  - Client endpoint sent message: 2 (客户端发送消息: 2)

观察代码，对比两者的不同点，可以发现：

1. accept 不阻塞

在 TCPServer 的 for 循环中，只有有客户端连入了服务器，这个循环才会进行一轮，这是因为 accept 方法是阻塞的，而在 NIOServer 中，把 serverSocket.configureBlocking 设置为 false，可以使这个 for 循环不断执行，每次循环用 if 条件判断是否有客户端连入。

```
// 2. 调用accept()方法开始监听，阻塞等待客户端的连接
    for(;;){
        System.out.println("阻塞等待客户端连接中...");
        Socket clientSocket = serverSocket.accept();
        ClientHandler c= new ClientHandler(clientSocket)
        c.start();
    }
}
```

```
for (;;) {
    方法不阻塞
    SocketChannel socketChannel = serverSocket.accept();
    if (socketChannel != null) {
        System.out.println("连接成功");
        socketChannel.configureBlocking(false);
        channelList.add(socketChannel);
    }
}
```

## 2. read 不阻塞

TCPServer 每次接收到一个新的连接，就创建新线程处理，而 NIO Server 没有创建新线程，而是在 for 循环中嵌套 while 循环，每次都把所有连接的 socket 遍历一遍，寻找是否有某个客户端输入了信息，并进行处理。

```
Iterator<SocketChannel> iterator = channelList.iterator();
while (iterator.hasNext()) {
    SocketChannel sc = iterator.next();
    ByteBuffer byteBuffer = ByteBuffer.allocate(BYTE_LENGTH);
    阻塞
    int len = sc.read(byteBuffer);

    把数据打印出来
    if (len > 0) {
        System.out.println("服务端接收到消息: " + new
            String(byteBuffer.array()));
    } else if (len == -1) {
        断开，把socket从集合中去掉
        iterator.remove();
        System.out.println("客户端断开连接");
    }
}
```

如果有大量客户端想要建立连接，对于阻塞的 read 来讲，如果没有用多线程来处理，那么如果这个连接的客户端一直不发数据，那么服务端线程将会一直阻塞在 read 函数上不返回，也无法接受其他客户端连接。而为每个客户端创建一个线程，服务器端的线程资源很容易被耗光。因此，NIO Server 通过每 accept 一个客户端连接后，将 socket 放到



一个数组里，不断遍历这个数组，调用每一个元素的非阻塞 `read` 方法，成功用一个线程处理了多个客户端连接。

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。

1. 调用了 `select` 函数，将已连接的 `Socket` 都放到一个文件描述符集合，然后调用 `select` 函数将文件描述符集合拷贝到内核里，通过遍历文件描述符集合的方式，当检查到有事件产生后，将此 `Socket` 标记为可读或可写，接着再把整个文件描述符集合拷贝回用户态里，然后用户态还需要再通过遍历的方法找到可读或可写的 `Socket`，然后再对其处理。

这一部分代码将 `channel` 注册到 `selector`，然后 `selector` 函数在内核中遍历每一个 `socket`，`readycount` 表示 `select` 函数选出来列表中需要被 `accept` 的个数

```
serverSocket.register(this.selector, SelectionKey.OP_ACCEPT);
System.out.println("服务端已启动");
for (;;) {
    系统提供的非阻塞I/O
    int readyCount = selector.select();
    if (readyCount == 0) {
        continue;
    }
}
```

在 `accept` 方法中，又监听了可以读的客户端个数：

```
// 监听读事件
channel.register(this.selector, SelectionKey.OP_READ);
}
```

Task5 (Bonus): 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，加入 `NIOserver` 的消息回写功能，先增加可写的判断：

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

然后仿照 `read` 方法的实现，写出 `write` 函数。与 `read` 方法相对比，相同的是先从 `channel` 中取得 `channel`，然后给缓冲区 `sendbuffer` 分配空间。清除缓冲区防止额外字符干扰。发送收到信息给客户端，放入 `buffer` 中。最后只需要再注册一个读事件，准备好读客户端输

入的信息。

```
private void write(SelectionKey key) throws IOException {
    SocketChannel channel = (SocketChannel) key.channel();
    ByteBuffer sendBuffer = ByteBuffer.allocate(BYTE_LENGTH);
    sendBuffer.clear();
    sendBuffer.put("收到!".getBytes());
    sendBuffer.flip();
    channel.write(sendBuffer);
    channel.register(selector, SelectionKey.OP_READ);
}
```

然后实现 NIOClient 类。首先定义读和写所使用的缓冲区：

```
private static final int BYTE_LENGTH = 64;
ByteBuffer writeBuffer = ByteBuffer.allocate(BYTE_LENGTH);
ByteBuffer readBuffer = ByteBuffer.allocate(BYTE_LENGTH);
```

参考服务端的实现，打开 socketchannel、selector、设置非阻塞、连接服务器、最后注册连接事件。

```
public void start() throws IOException {
    int port = 9091;
    int connected=0;
    // 打开socket通道
    SocketChannel sc = SocketChannel.open();
    // 设置为非阻塞
    sc.configureBlocking(false);
    // 连接服务器地址和端口
    sc.connect(new InetSocketAddress( hostnames: "127.0.0.1", port));
    // 创建选择器
    Selector selector = Selector.open();
    // 注册连接服务器socket的CONNECT事件
    sc.register(selector, SelectionKey.OP_CONNECT);
}

private void startServer() throws IOException {
    this.selector = Selector.open();
    // ServerSocketChannel与serverSocket类似
    ServerSocketChannel serverSocket = ServerSocketChannel.open();
    serverSocket.socket().bind(new InetSocketAddress( port: 9091));
    // 设置无阻塞
    serverSocket.configureBlocking(false);
    // 将channel注册到selector
    serverSocket.register(this.selector, SelectionKey.OP_ACCEPT);
}
```

在内核里对 key 集进行迭代，选出发生事件的 key。

```
selector.select();
// 获取发生事件的SelectionKey
Set<SelectionKey> keys = selector.selectedKeys();

Iterator<SelectionKey> keyIterator = keys.iterator();
```

将事件的处理方法写在了条件判断块内部。首先是连接事件，由于每个客户端连接一次，将 connected 设为 1，然后发送 Hello Server 消息给服务器，最后注册一个写事件，等待服务器发来消息。

```
if (key.isConnectable() && connected == 0) {
    System.out.println("OP_CONNECT");
    sc.finishConnect();

    System.out.println("Server connected...");
    if (connected == 0) {
        sc.write(ByteBuffer.wrap("Hello Server".getBytes()));
        connected = 1;
    }
    // 注册WRITE事件，准备读取用户输入
    sc.register(selector, SelectionKey.OP_READ);
}
```

然后是客户端给服务器写内容的功能。将控制台输入的内容放入到缓冲区里，写给服务器，然后等待读取服务器发来的消息。

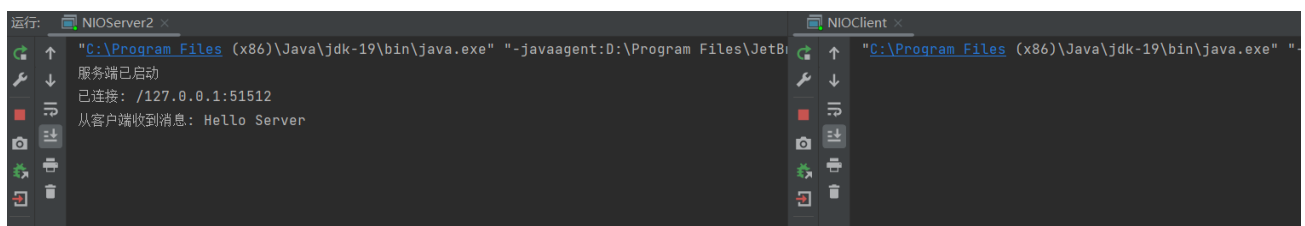
```
} else if (key.isWritable()) {
    System.out.print("请输入消息:");
    String message = scanner.next();
    writeBuffer.clear();
    writeBuffer.put(message.getBytes());
    writeBuffer.flip();
    sc.write(writeBuffer);
    // 注册读操作，下一次读取
    sc.register(selector, SelectionKey.OP_READ);
}
```

最后是读取服务器发来消息的功能。首先清空缓冲区，然后读取缓冲区的长度，将缓冲区转换成字符串，按照其长度输出在控制台里，防止输出冗余字符。最后再注册一个写事件，提供用户给服务器发消息的机会。

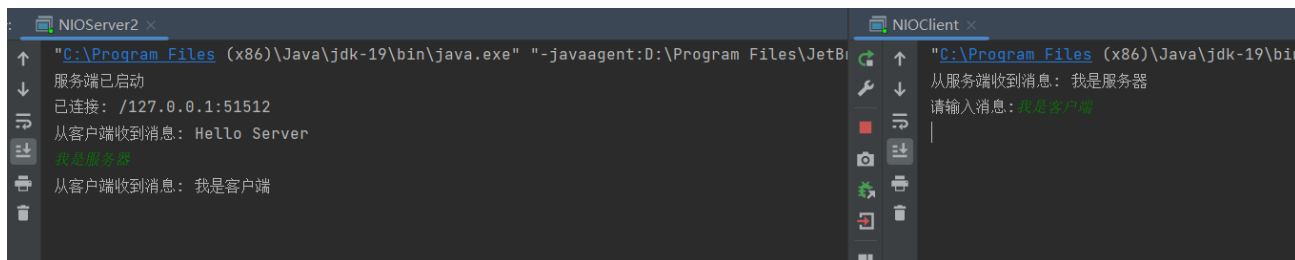
```
} else if (key.isReadable()) {
    System.out.print("从服务端收到消息: ");
    readBuffer.clear();
    int numRead = sc.read(this.readBuffer);
    System.out.println(new String(readBuffer.array(), offset: 0, numRead));
    sc.register(selector, SelectionKey.OP_WRITE);
}
```

最后执行 `keyIterator.remove()` 删除已经处理过的事件。

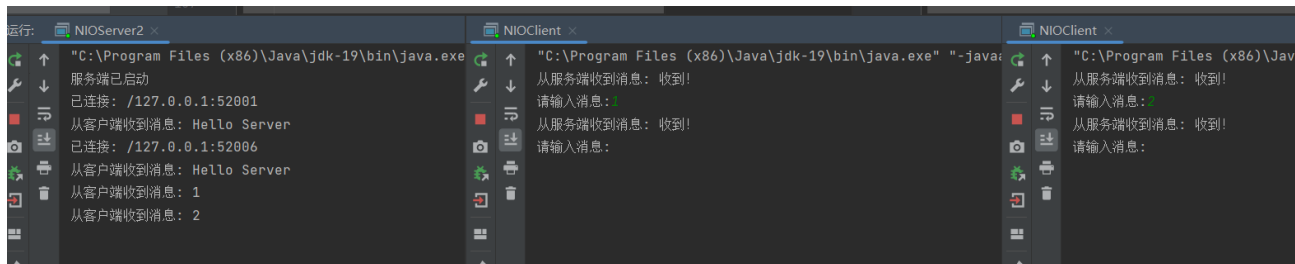
开启服务器和一个客户端，运行结果如下：



将“收到！”改写为从标准输入读取信息，然后服务器、客户端分别发送一条消息，运行结果如下：



服务器同时给多个客户端提供服务，效果如下：



如果服务器是通过标准输入来读取要发送给客户端的信息，就会出现一些问题，如果服务器没有及时写入，不论用户发送什么信息，服务器都不会再回应，直到服务器终端输入信息为止。

## 五、总结

本次实验中我更加熟练地运用了 Socket 编程，给客户端和服务端实现读写操作。数据发送与接收并行，是通过创建并行的两个线程，分别收取客户端输入的数据、写入数据到客户端，从而可以让读和写操作不要相互阻塞等待。在此过程中，需要注意的问题是要使用 join 线程同步方法，防止用户端的主线程提前结束，用户关闭 io 通道，从而发生报错的情况。

服务器给每个客户端进行消息共享，很容易想到这要求服务器保存所有当前正在连接的客户端的 socket，并在这个数组改变的时候给所有客户端发送消息。

传统网络 IO 有许多弊端，首先，对于阻塞 IO，服务器的线程会阻塞在 accept 函数和 read 函数上，而非阻塞 IO，通过创建新的线程来处理每个客户端，很容易消耗掉服务器的线程资源。IO 多路复用的原理是进行系统调用 select，让操作系统遍历文件描述符，省去了在应用层遍历的时间成本，而 poll 去掉了 select 只能监听 1024 个文件描述符的限制，epoll 利用红黑树加快了遍历的速度，是 IO 多路复用的最优方法。