

华东师范大学数据科学与工程学院上机实践报告

课程名称：计算机网络与编程	年级：22级	上机实践成绩：
指导教师：张召	姓名：朱天祥	
上机实践名称：Socket与多路IO	学号：10225501461	上机实践日期：23/4/14
上机实践编号：No.8	组号：	

一、目的

- 1.使用ServerSocket和Socket实现TCP通信
- 2.了解粘包概念并尝试解决

二、实验内容

1. 使用ServerSocket和Socket编写代码
2. 解决粘包问题

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。

大致思路：因为现在服务器也可以发送数据了，所以client端也需要并行地读和写，因此client也需要两个线程来分别进行读写，所以我们需要read和write两个线程类，首先是client端中的readHandler类：

```
class ReadHandler extends Thread{
    Socket socket;

    ReadHandler(Socket socket){
        this.socket = socket;
    }

    @Override
    public void run() {
```

```

        BufferedReader bufferedReader;
        try {
            bufferedReader = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        String msg = null;
        while(true){
            try {
                msg = bufferedReader.readLine();
                System.out.println("客户端接受到服务器发来的信息: " + msg);
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
        }
    }
}

```

writeHandler类:

```

class WriteHandler extends Thread{
    Socket socket;

    WriteHandler(Socket socket){
        this.socket = socket;
    }

    @Override
    public void run() {
        PrintWriter printWriter;
        try {
            printWriter = new PrintWriter(socket.getOutputStream());
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        String msg = null;
        Scanner scanner = new Scanner(System.in);
        while(true){
            msg = scanner.nextLine();
            printWriter.println(msg);
            printWriter.flush();
        }
    }
}

```

同时还需要修改main方法，把socket传到两个new出来的线程中，并且开启线程，需要注意的是main线程需要等待两个线程执行完毕后再结束，也就是需要在main函数中调用两个join方法：

```

public static void main(String[] args) {
    int port = 9091;
    TCPClient client = new TCPClient();
    try {
        client.startConnection("127.0.0.1", port);
    }
}

```

```

        ReadHandler readHandler = new ReadHandler(client.getClientSocket());
        WriteHandler writeHandler = new WriteHandler(client.getClientSocket());
        readHandler.start();
        writeHandler.start();
        readHandler.join();
        writeHandler.join();
    } catch (IOException e) {
        e.printStackTrace();
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    } finally {
        client.stopConnection();
    }
}

```

server运行结果:

```

D:\Java\jdk1.8.0_40\bin\java.exe ...
阻塞等待客户端连接中...
读到的数据为: hello server
hello client
hhello sss
hello,client
读到的数据为: hello server

```

client运行结果:

```

hello server
客户端接受到服务器发来的信息: hello
客户端接受到服务器发来的信息: client
客户端接受到服务器发来的信息: hhello
客户端接受到服务器发来的信息: sss
客户端接受到服务器发来的信息: hello,client
hello server

```

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。

由于需要所有socket的信息以及所有的ch线程来发送信息，因此需要在server中维护两个数组

```

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

```

server的start方法:

```

public void start(int port) throws IOException {

```

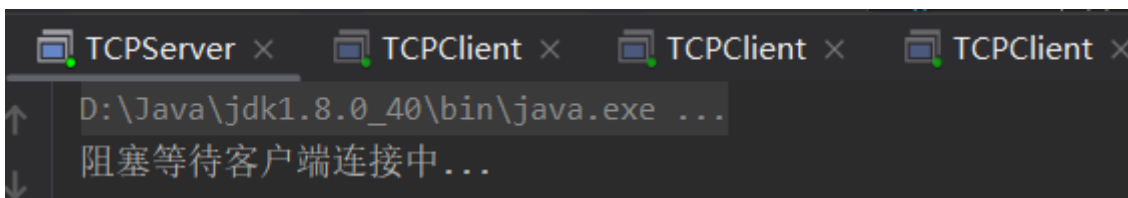
```

serverSocket = new ServerSocket(port);
System.out.println("阻塞等待客户端连接中...");
for (;;) {
    Socket socket = serverSocket.accept();
    ClientHandler ch = new ClientHandler(socket);
    sockets.add(socket);
    handlers.add(ch);
    ch.start();
    StringBuilder builder = new StringBuilder();
    for(Socket s : sockets) {
        if(!s.isClosed() && s.isConnected() && !s.isInputShutdown() &&
!s.isOutputShutdown())
            builder.append(s.getRemoteSocketAddress().toString() + " ");
    }
    for(ClientHandler handler : handlers) {
        handler.sendMsg(builder.toString());
    }
}
}

```

start方法需要在每次成功与客户端连接后将socket和handler线程放到两个数组当中，并且获取所有的socket信息，再把信息通过遍历handler数组的方式来发送出去。

server运行结果：

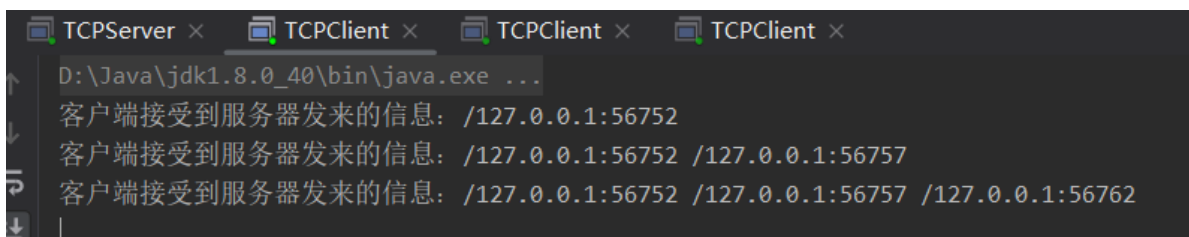


```

TCPServer x TCPClient x TCPClient x TCPClient x
D:\Java\jdk1.8.0_40\bin\java.exe ...
阻塞等待客户端连接中...

```

第一个建立连接的client:

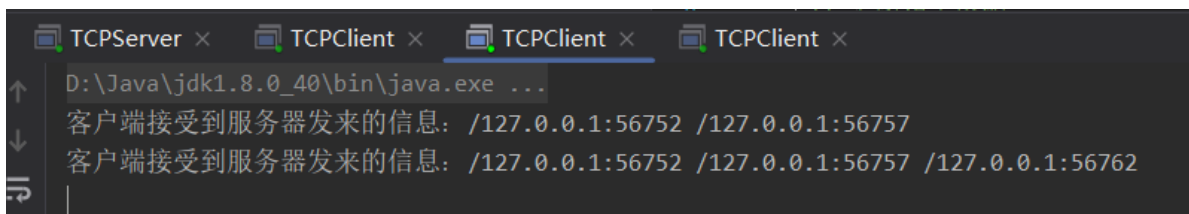


```

TCPServer x TCPClient x TCPClient x TCPClient x
D:\Java\jdk1.8.0_40\bin\java.exe ...
客户端接受到服务器发来的信息: /127.0.0.1:56752
客户端接受到服务器发来的信息: /127.0.0.1:56752 /127.0.0.1:56757
客户端接受到服务器发来的信息: /127.0.0.1:56752 /127.0.0.1:56757 /127.0.0.1:56762

```

第二个建立连接的client:

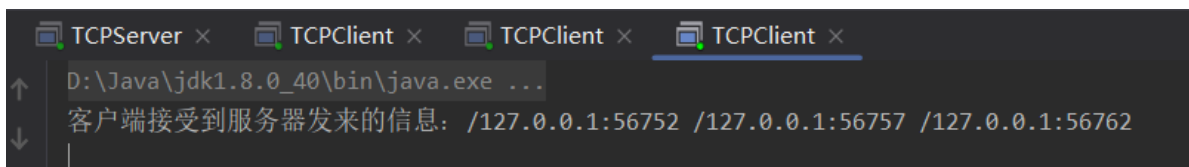


```

TCPServer x TCPClient x TCPClient x TCPClient x
D:\Java\jdk1.8.0_40\bin\java.exe ...
客户端接受到服务器发来的信息: /127.0.0.1:56752 /127.0.0.1:56757
客户端接受到服务器发来的信息: /127.0.0.1:56752 /127.0.0.1:56757 /127.0.0.1:56762

```

第三个建立连接的client:



```

TCPServer x TCPClient x TCPClient x TCPClient x
D:\Java\jdk1.8.0_40\bin\java.exe ...
客户端接受到服务器发来的信息: /127.0.0.1:56752 /127.0.0.1:56757 /127.0.0.1:56762

```

此后断开第二个和第三个建立连接的进程，再开启一个client，得到如下运行结果：

第一个建立连接的client:

```
TCPServer x TCPClient x TCPClient x
D:\Java\jdk1.8.0_40\bin\java.exe ...
客户端接受到服务器发来的信息: /127.0.0.1:56752
客户端接受到服务器发来的信息: /127.0.0.1:56752 /127.0.0.1:56757
客户端接受到服务器发来的信息: /127.0.0.1:56752 /127.0.0.1:56757 /127.0.0.1:56762
客户端接受到服务器发来的信息: /127.0.0.1:56752 /127.0.0.1:56757 /127.0.0.1:56762 /127.0.0.1:51147
```

第二个建立连接的client:

```
TCPServer x TCPClient x TCPClient x
D:\Java\jdk1.8.0_40\bin\java.exe ...
客户端接受到服务器发来的信息: /127.0.0.1:56752 /127.0.0.1:56757 /127.0.0.1:56762 /127.0.0.1:51147
```

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。

TCPserver与NIOserver不同点: TCPserver因为accept以及接受客户端信息等方法会阻塞, 因此需要通过创建大量线程来保证能够并发地接受客户端连接同时与客户端进行通信。而NIOserver的accept方法不会阻塞, 读取客户端发来的信息也不用等待, 因此无需创建线程即可完成并发通信, 每次循环accept一次客户端, 如果有则连接, 无则继续运行, 每次只需一个线程便可同时完成与客户端连接并处理客户端发送的数据。

当并发量超过一万之后, 如果按照lab7的代码的话, 每次建立一个客户端连接就需要创建一个读取客户端信息的线程, 因此创建线程数超过一万, 会导致系统资源的浪费和调度负担增加。

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。

在NIOserver代码中, selector和channel的方法有很多封装了底层的系统调用, 以下是各个方法与系统调用的对应关系:

1. Selector.open(): 调用的是 epoll系统调用。
2. channel.configureBlocking(false): 在内核中调用了fcntl 系统调用将文件描述符设置为非阻塞模式。
3. SelectionKey.OP_READ: 相当于注册监听读取事件, 调用了epoll_ctl 添加监听给定文件描述符上的读取事件。

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)

在NIOserver的read方法中添加一行代码, 当服务器读到消息后给客户端信息:

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

NIOclient代码如下:

```
public class NIOclient {

    public static void main(String[] args) {
```

```

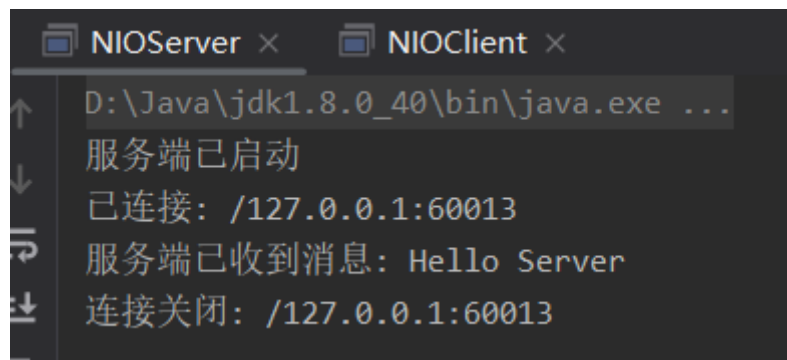
SocketChannel channel = null;
try {
    channel = SocketChannel.open();
    channel.connect(new InetSocketAddress("127.0.0.1", 9091));
    channel.write(ByteBuffer.wrap("Hello Server".getBytes()));

    ByteBuffer buffer = ByteBuffer.allocate(64);
    int numRead = channel.read(buffer);
    System.out.println(new String(buffer.array(), 0, numRead,
StandardCharsets.UTF_8));
    channel.close();
} catch (IOException e) {
    if(channel != null) {
        try {
            channel.close();
        } catch (IOException ex) {
            throw new RuntimeException(ex);
        }
    }
}
}
}

```

运行结果如下:

NIOServer的输出:



A screenshot of a Java IDE console window showing the output of the NIOServer application. The window has two tabs: 'NIOServer' and 'NIOClient'. The 'NIOServer' tab is active, displaying the following text: 'D:\Java\jdk1.8.0_40\bin\java.exe ...', '服务端已启动' (Server started), '已连接: /127.0.0.1:60013' (Connected: /127.0.0.1:60013), '服务端已收到消息: Hello Server' (Server received message: Hello Server), and '连接关闭: /127.0.0.1:60013' (Connection closed: /127.0.0.1:60013).

NIOClient的输出:



A screenshot of a Java IDE console window showing the output of the NIOClient application. The window has two tabs: 'NIOServer' and 'NIOClient'. The 'NIOClient' tab is active, displaying the following text: 'D:\Java\jdk1.8.0_40\bin\java.exe ...' and '服务器已收到消息: Hello Server' (Server received message: Hello Server).