

## 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

上机实践日期：2023.4.14

上机实践编号：7

组号：

上机实践时间：9:50

### 一、实验目的

使用ServerSocket 和 Socket 实现 TCP 通信

了解粘包概念并尝试解决

### 二、实验任务

使用ServerSocket 和 Socket 编写代码

解决粘包问题

### 三、使用环境

IntelliJ IDEA

JDK 版本：Java 19

### 四、实验过程

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。

1. 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

```
Scanner sc=new Scanner(System.in);
while(true){
    String myWord=sc.next();
    String response = client.sendMessage(myWord);
    System.out.println(response);
}
```

2. 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

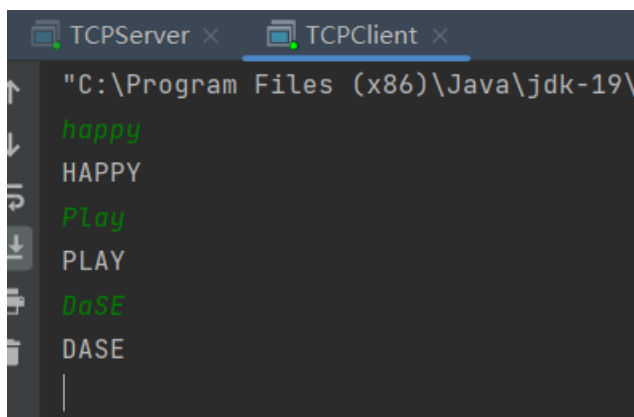
创建字符串 newWord，遍历用户输入字符串的每个字符，将 toUpperCase 函数输出的大写字母不断拼接到字符串中，最后输出给客户端。如果客户端本来就输入的大写字母，则不加以改变直接拼接进去。

```
String newWord= "";
char[] arrays = str.toCharArray();
for (char c1 : arrays) {
    if (c1 >= 97 && c1 <= 122) {
        char c2 = toUpperCase(c1);
        newWord=String.join( delimiter: "",newWord,Character.toString(c2));
    }
    else{
        newWord=String.join( delimiter: "",newWord,Character.toString(c1));
    }
}
out.println(newWord);
```

toUpperCase 函数的实现如下，通过 ASCII 码顺序将小写转换为大写：

```
public char toUpperCase(char c1){
    int b = (int) c1 -32;
    return (char)b;
}
```

3. 客户端收到修改后的数据，并在其屏幕上显示



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。

主线程在循环中等待客户端的连接，每连接一次，就接收 clientSocket，然后创建线程 ClientHandler，让客户端的接管程序在另一个线程里运行，主线程继续等待其他客户端的连接。

```
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();
}
```

然后重写 run 方法:

```
PrintWriter out = null;
try {
    out = new PrintWriter(new OutputStreamWriter(this.socket.getOutputStream(),
        StandardCharsets.UTF_8), autoFlush: true);
} catch (IOException e) {
    throw new RuntimeException(e);
}
BufferedReader in = null;
try {
    in = new BufferedReader(new InputStreamReader(this.socket.getInputStream(),
        StandardCharsets.UTF_8));
} catch (IOException e) {
    throw new RuntimeException(e);
}
```

需要单独获取每个客户端的 io, 赋值给 in 和 out 变量

```
String str;

while(true){
    try {
        if ((str = in.readLine()) == null) break;
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
    System.out.println("我是服务端, 客户端说: " + str);

    String newWord= "";
    char[] arrays = str.toCharArray();
    for (char c1 : arrays) {
        if (c1 >= 97 && c1 <= 122) {
            char c2 = toUpperCase(c1);
            newWord=String.join( delimiter: "",newWord,Character.toString(c2));
        }
        else{
            newWord=String.join( delimiter: "",newWord,Character.toString(c1));
        }
    }
    out.println(newWord);
}
```

然后复制 task1 的内容即可。

测试结果如下:

```

TCPClient x TCPClient x TCPClient x
"C:\Program Files (x86)\Java\jdk-19\bin\java.exe" "-javaagent:D:
阻塞等待客户端连接中...
阻塞等待客户端连接中...
阻塞等待客户端连接中...
阻塞等待客户端连接中...
我是服务端，客户端说: mao
我是服务端，客户端说: mao2
我是服务端，客户端说: mao3

TCPClient x TCPClient x TCPClient x
"C:\Program Files (x86)\Java\jdk-19\bin\java.exe" "-javaa
mao3
MA03
|

TCPClient x TCPClient x TCPClient x
"C:\Program Files (x86)\Java\jdk-19\bin\java.exe" "-javaagent:D:\P
mao2
MA02

TCPClient x TCPClient x TCPClient x
"C:\Program Files (x86)\Java\jdk-19\bin\java.exe" "-javaag
mao
MA0

```

可以看到，服务器给每个客户端返回了相应内容，进程之间互不干扰。

**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。

```

阻塞等待客户端连接中...
服务端已收到消息: NETWORK PRINCIPLE
服务端已收到消息: NETWORK PRINCIPLENETWORK PRINCIPLENETWORK PRINCIPLENETWORK PRINC
服务端已收到消息: IPLENETWORK PRINCIPLENETWORK PRINCIPLENETWORK PRINCIPLENETWORK P
服务端已收到消息: RINCIPLENETWORK PRINCIPLE
|

```

运行上述代码，可以看到服务器输出了如图信息，而本应当输出的信息是 10 次 NETWORK PRINCIPLE 字符串。

这就是我们常说的拆包（也有人叫半包），对应的还有粘包，就是在通过 TCP 协议交互数据过程中，TCP 底层并不了解它的上层业务数据（比如此文例子中放入 ByteBuffer 中要发送的数据，或者 HTTP 报文等）的具体含义，可能会根据实际情况（比如 TCP 缓冲区或者此文中定义的 NIO 接收数据的缓冲区 ByteBuffer）对数据包进行

拆分或合并。

因此，当客户端发送了一段较长的数据包时，在客户端可能会分成若干个较小的数据包分别发送，或者在服务端也可能分成了若干个较小的数据包来接收。

为了解决这个问题，基本思路是将每个数据包的长度写在数据包的第一个字节，在后面的字节里写入数据包的内容，而服务端读取的时候，先获取第一个字节中的长度信息 `length`，然后依此读取 `length` 个字节。

代码实现如下：

```
public void sendMessage(String msg) throws IOException {
    // 重复发送十次
    for(int i=0;i<10;i++){
        byte[] bytes=msg.getBytes();
        ByteBuffer bB=ByteBuffer.allocate( capacity: 1+bytes.length);
        bB.put((byte) bytes.length);
        bB.put(bytes);
        bB.flip();
        out.write(bB.array());
    }
}
```

在客户端，将直接写入改成在开头处分配一个字节作为包长度的记录。

```
byte[] readBuffer = new byte[1024];
int bufferCount=0;
for(;;) {
    byte[] tmp=new byte[BYTE_LENGTH];
    int cnt = is.read(tmp, off: 0, BYTE_LENGTH);
    if (cnt>0)
        System.arraycopy(tmp, srcPos: 0,readBuffer,bufferCount,cnt);
    bufferCount+=cnt;
    if (bufferCount<=1){
        continue;
    }
    int length=readBuffer[0];
    if(bufferCount<1+length){
        continue;
    }
    byte[] content=Arrays.copyOfRange(readBuffer, from: 1, to: length + 1);
    System.out.println("服务端已收到消息: " + new String(content).trim());
    bufferCount-=(1+length);
    System.arraycopy(readBuffer, srcPos: length+1,readBuffer, destPos: 0,bufferCount);
}
```

在服务端，首先定义一个缓冲区数组 `readBuffer`，用来记录还未输出的数据。  
`bufferCount` 记录缓冲区中的字节数。

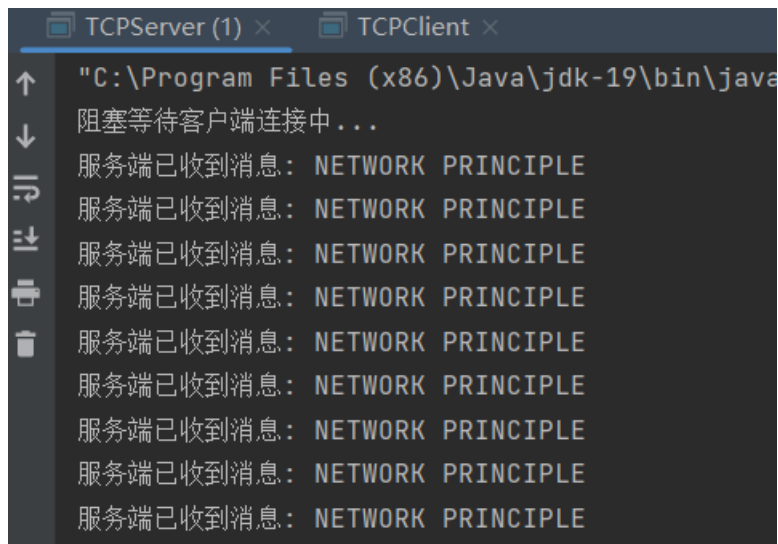
然后在循环中，每次读取一部分字节放入 `tmp` 字节数组中，如果读取到了数据，就把数据加到缓冲区数组的最后，并延长缓冲区数组的长度。

接下来判断此时缓冲区是否足够长，如果长度只有一个字节，说明只接收到了一个表示数据包字节数的长度信息，此时再等待一轮循环。

如果长度多于一个字节，则读取这个长度信息，判断此时缓冲区中的数据是否足够一个包大小。

按照 `length` 的长度，将缓冲区数组的前面一部分复制到 `content` 中，并输出到服务器。

对缓冲区数组进行处理，删去已经输出过的部分，把后面的内容前移。

A screenshot of a Java IDE window titled 'TCPClient' and 'TCPClient'. The console output shows the following sequence of messages:

```
"C:\Program Files (x86)\Java\jdk-19\bin\java
阻塞等待客户端连接中...
服务端已收到消息: NETWORK PRINCIPLE
服务端已收到消息: NETWORK PRINCIPLE
服务端已收到消息: NETWORK PRINCIPLE
服务端已收到消息: NETWORK PRINCIPLE
服务端已收到消息: NETWORK PRINCIPLE
服务端已收到消息: NETWORK PRINCIPLE
服务端已收到消息: NETWORK PRINCIPLE
服务端已收到消息: NETWORK PRINCIPLE
服务端已收到消息: NETWORK PRINCIPLE
```

可以看到现在服务器输出了正确的信息。

## 五、总结

基于 java 实现 socket 网络编程的主要步骤是先创建一个 `Socket` 对象，这个 `Socket` 对象实质上提供了进程通信的端点。服务器端的 `Socket` 先等待客户端连接，如果连接到了，那么用 `accept` 方法获取用户端 `Socket`，可以通过这个 `API` 获取到用户的输入输出流，从而实现通信。

用户端先创建 `Socket`，然后将输入输出流接入到 `Socket`，实现与服务端的通信。最终，服务端和客户端都要实现关闭 `Socket` 功能。

在服务端与客户端传输信息过程中，可能会发生半包粘包情况，这是因为底层的 `TCP` 协议不知道字节流的具体含义，从而无法正确按照包中数据的含义分开。因此在应用层传输数据时，需要在数据包中加入数据长度信息，或者将数据包用分隔符分隔开，指导服务端按照正确顺序切割字节流。