

计算机网络实验报告

实验二 网络基础编程实验

实验目的

通过本实验，学习采用Socket（套接字）设计简单的网络数据收发程序，理解应用数据包是如何通过传输层进行传送的。在本次实验中，我使用的是JAVA语言。

实验步骤

一 采用TCP进行数据发送的简单程序

针对本小题，我编写了两个文件，分别代表客户端和服务端。服务器接收来自客户端的消息，并作出“Hello”的回应；客户端接收用户的标准输入，并将其发送给服务器。

- 编写服务器端代码（TCPServer.java）

要进行 TCP 连接，服务端首先要在本地指定一个端口，创建（开放）一个 server socket；

```
ServerSocket server = new ServerSocket(8888); // 建立端口
```

随后服务器在端口进行阻塞式监听，等待客户与之进行TCP连接；

```
Socket connection = server.accept(); // 监听端口
```

当 TCP 通过三次握手成功建立连接之后，服务端将会创建 TCP 连接的输入输出流；

```
// 包装data输入输出流
DataInputStream inputStream = new
DataInputStream(connection.getInputStream());
DataOutputStream outputStream = new
DataOutputStream(connection.getOutputStream());
```

随后就可以不断地获取从 client 发送过来的数据，并通过输出流作出响应了；

```
while (true) {
    // 获取客户发送过来的信息
    String clientSentence = inputStream.readUTF();
    System.out.println("FROM CLIENT: " + clientSentence);

    // 向客户发送信息
    String send = "hello, " + clientSentence;
    outputStream.writeUTF(send);
    outputStream.flush();
}
```

- 编写客户端代码（TCPClient.java）

当服务器部署完成后，客户端想要创建 Socket 与之进行TCP连接，则必须要指定服务器的 IP 地址与端口号。而我们是本机的局域网内进行实验，因此 IP 地址可以填写为 `localhost`，而端口号则为我们开放的服务器端口号 `8888`。这样就建立了客户端到服务器的 TCP 连接。

```
Socket client = new Socket("localhost", 8888); // 建立到服务器的TCP连接
```

随后客户端不断地从用户输入缓冲区读取一行，并通过输出数据流发送给服务器；于此同时通过输入数据流得到服务器对其作出的响应，代码与服务器端类似。

```
while (true) {
    // 从键盘读入一行
    String send = userInput.readLine();
    if(send.equals("exit")) break; // 如果是"exit"，将退出程序

    // 向服务器发送
    DataOutputStream output = new
    DataOutputStream(client.getOutputStream());
    output.writeUTF(send);
    output.flush();

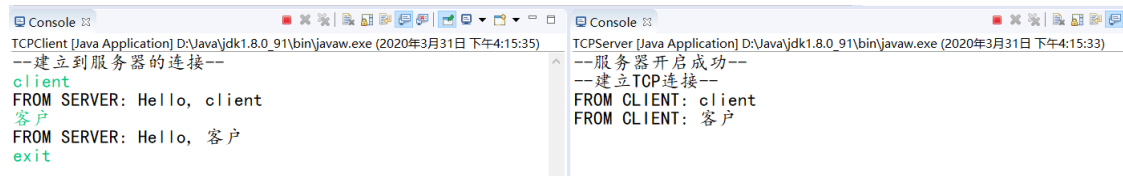
    // 从服务器获取数据
    DataInputStream dataInputStream = new
    DataInputStream(client.getInputStream());
    String serverSentence = dataInputStream.readUTF();
    System.out.println("FROM SERVER: " + serverSentence);
}
```

值得一提的是，当用户输入"exit"时，客户端将退出循环并将通过 TCP四次挥手断开连接。而服务器端将一直开放端口，Socket保持开启状态。

```
client.close(); // 客户端断开连接
```

• 运行测试结果

左侧为客户端，而右侧为服务器。



二 采用UDP进行数据发送的简单程序

同样分为客户端与服务器，交互逻辑与上面相同。重点是将TCP连接替换成了UDP连接，让我们来看看它们有什么不同之处。

• 编写服务器代码

与 TCP 连接类似，服务器仍然需要开放一个端口等待用户进行 UDP 连接，但采用的Socket类是不同的。在 TCP 连接中，采用的是 `ServerSocket` 和 `Socket` 类；而 UDP 连接采用的是 `DatagramSocket`。

```
DatagramSocket serverSocket = new DatagramSocket(8888);
```

然后需要准备容器，并封装成数据包，用以作客户端发送的数据包的缓冲。

```
// 准备容器接收
byte[] container = new byte[1024];
// 等待包裹容器封包
DatagramPacket receivePacket = new DatagramPacket(container,
    container.length);
```

接下来调用receive()方法阻塞进程，等待客户端发送数据包。接收到数据包之后，使用getData()方法即可将客户端发送的字节流解析为我们能够看到的ASCII码。

```
// 接收包裹
serverSocket.receive(receivePacket);
String clientMessage =
    new String(receivePacket.getData(), 0, receivePacket.getLength());
```

当服务器收到 Packet 之后，如果想要向客户端进行回应应该怎么做呢？

这里 UDP 和 TCP 就有一个很大的不同。由于 TCP 是“面向连接”的，因此服务器可以直接通过建立好的 TCP 连接，将信息通过输出流发送到客户端；而 UDP 不是“面向连接”的，因此它需要从客户端发来的Packet中提取客户端的 IP 和 Port，随后将待发送的字符串转化为字节流，然后封装数据包发送给客户端。

```
// 1.从UDP包中获取IP，端口号
InetAddress IPAddress = receivePacket.getAddress();
int port = receivePacket.getPort();

// 2.将要发送的字符串
String send = "hello, " + clientMessage;

// 3.将字符串转化为字节流
byte[] sendData = new byte[1024];
sendData = send.getBytes();

// 4.封装数据包并发送
DatagramPacket sendPacket =
    new DatagramPacket(sendData, sendData.length, IPAddress, port);
serverSocket.send(sendPacket);
```

• 编写客户端代码

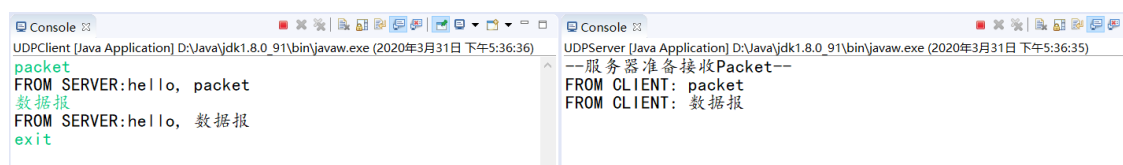
首先创建客户端的Socket，由于我们不需要提前建立连接，因此无需指定服务器 IP 和 Port。

```
DatagramSocket clientSocket = new DatagramSocket();
```

发送数据包的流程与服务器端的代码一致，这里就不再赘述了。

• 测试结果

左侧为客户端，而右侧为服务器。



```
Console [UDPClient (Java Application)] D:\Java\jdk1.8.0_91\bin\javaw.exe (2020年3月31日 下午5:36:36)
packet
FROM SERVER:hello, packet
数据报
FROM SERVER:hello, 数据报
exit

Console [UDPServer (Java Application)] D:\Java\jdk1.8.0_91\bin\javaw.exe (2020年3月31日 下午5:36:35)
--服务器准备接收Packet--
FROM CLIENT: packet
FROM CLIENT: 数据报
```

三 多线程/线程池对比

本题编写了4个文件，其中包括：客户端client.java，使用普通多线程的服务器ServerThread.java，使用线程池的服务器ServerPool.java，处理服务器应答的线程Process.java

客户端：与服务器建立TCP连接，发送当前的时间，随后关闭连接

普通多线程的服务器：使用一客户一线程模式，处理多个TCP连接

使用线程池的服务器：使用固定线程个数的线程池，每个线程不断接收不同客户的信息

处理服务器应答的线程代码：使用TCP连接，接收客户端发送的消息，并打印在屏幕上

- **客户端代码**

与之前讲述的 TCP 连接代码类似，增加了获取当前时间的功能。

```
// 获取当前日期，作为发送数据
Date date = new Date();
SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
String data = formatter.format(date);
```

- **处理服务器应答的线程代码**

该线程以一个 TCP 连接的Socket作为参数，从输入流得到客户端发送的日期，并打印在屏幕上。

```
public class Process implements Runnable {

    public static void handleClient(Socket client) throws IOException {

        // 包装data输入输出流
        DataInputStream inputStream = new
        DataInputStream(client.getInputStream());

        // 获取客户发送过来的信息并打印在屏幕上
        String message = inputStream.readUTF();
        System.out.println("FROM CLIENT: " + message);

        // 关闭资源
        inputStream.close();
        client.close();
    }

    // 线程执行的入口函数
    @Override
    public void run() {
        handleClient(client);
    }
}
```

- **普通多线程的服务器代码**

主线程不断监听端口，若与某个客户端建立了 TCP 连接，则将这个连接交给一个新的线程进行处理。

```

while (true) {
    // 监听端口，获得新连接
    Socket connection = server.accept();
    // 创建线程去处理
    Thread thread = new Thread(new Process(connection));
    thread.start();
    // 打印处理线程的名称
    System.out.println("创建线程: " + thread.getName());
}

```

• 普通多线程的服务器测试情况

当我们启动ServerThread服务器后，重复运行Client.java，模拟有多个客户与服务器连接的情况，观察服务器的运行结果如下。可以看到，每当客户端尝试与服务器进行连接，服务器都将开启一个新的线程进行处理。

```

ServerThread [Java Application] D:\Java\jdk1.8.0_91\bin\javaw.exe (2020年3月31日 下午6:06:53)
--服务器启动--
创建线程: Thread-0
FROM CLIENT: 2020-03-31 18:06:56
创建线程: Thread-1
FROM CLIENT: 2020-03-31 18:06:56
创建线程: Thread-2
FROM CLIENT: 2020-03-31 18:06:57
创建线程: Thread-3
FROM CLIENT: 2020-03-31 18:06:58
创建线程: Thread-4
FROM CLIENT: 2020-03-31 18:06:58
创建线程: Thread-5
FROM CLIENT: 2020-03-31 18:06:58
创建线程: Thread-6
FROM CLIENT: 2020-03-31 18:06:59
创建线程: Thread-7
FROM CLIENT: 2020-03-31 18:06:59
创建线程: Thread-8
FROM CLIENT: 2020-03-31 18:07:00
创建线程: Thread-9
FROM CLIENT: 2020-03-31 18:07:00
创建线程: Thread-10

```

• 线程池服务器代码

普通的多线程是由主线程进行端口监听，而线程池则是有很多个线程同时进行端口监听。一旦客户端尝试与服务器进行连接，服务器系统将会选择一个幸运的线程从accept()方法中返回Socket并进行处理，其余线程则继续阻塞。

```

for (int i = 0; i < poolSize; i++) {
    // 创建poolSize个线程
    Thread thread = new Thread() {
        @Override
        // 每个线程的功能是反复循环，从（共享的）ServerSocket实例接收客户端连接
        public void run() {
            while (true) {
                Socket socket = server.accept();
                Process.handleClient(socket);
            }
        }
    };
    // 启动线程
    thread.start();
    System.out.println(" 线程 " + thread.getName() + " 开始工作 ");
}

```

```
}
```

• 线程池服务器运行结果

这里将线程池的大小设置为5，当服务器启动时，这5个线程就会依次启动，并行地处理与客户端的TCP连接。

```
Console
ServerPool [Java Application] D:\Java\jdk1.8.0_91\bin\javaw.exe (2020年3月31日 下午6:34:49)
--服务器开启--
线程 Thread-0 开始工作
线程 Thread-1 开始工作
线程 Thread-2 开始工作
线程 Thread-3 开始工作
线程 Thread-4 开始工作
FROM CLIENT: 2020-03-31 18:34:57
FROM CLIENT: 2020-03-31 18:34:57
FROM CLIENT: 2020-03-31 18:34:58
FROM CLIENT: 2020-03-31 18:34:59
FROM CLIENT: 2020-03-31 18:34:59
FROM CLIENT: 2020-03-31 18:34:59
FROM CLIENT: 2020-03-31 18:34:59
FROM CLIENT: 2020-03-31 18:34:59
```

• 两种方式的比较（各有优劣）

一客户一线程模式：优势是实现方法简单，当客户端数量较少时能够快速地进行处理。但是当客户数量增多，并发压力加大时，会过度消耗系统资源（包括CPU、保存线程状态的内存），当线程由阻塞变为活跃状态时，会由于上下文切换而导致时间的浪费，此时一个额外的线程可能增加客户端总服务时间。

线程池：优势是可以重复利用每一个线程，当一个线程处理完一个客户端后，它将返回线程池并为下一次请求处理做好准备。如果连接请求到达服务器时，线程池中的所有线程都已经被占用，它们则在一个队列中等待，直到有空闲的线程可用。但是线程池的大小不好事先规定，如果线程池太小则容易无法对众多的客户端进行响应，若线程池太大又会消耗系统过多的资源。

四 写一个简单的chat程序，并能互传文件

本题模拟了两个用户，实现了聊天和传文件两个功能。其中每台用户主机开放两个TCP端口、运行四个线程。

两个端口分别是：接收对方信息的端口、接收文件的端口

四个线程分别是：收信息线程、发信息线程、收文件线程、发文件线程

双方开启**收信息线程**、**发信息线程**，采用前面所述的 TCP 通信的方式，达到可以相互聊天的效果。同时要先开启**收文件线程**，循环等待文件的到来。当一方发送信息的格式为 `[file]name.txt`，即以 `[file]` 为前缀的字符串，则表明将要发送一个文件 `name.txt`，此时**发信息线程将调用发文件线程**，完成文件的发送。

由于普通的聊天在**利用TCP进行数据传输**当中讲解过，这里主要描述收发文件的逻辑实现。

• 发送文件（SendFile.java）

对于 Socket 来说，文件不过是字节流。如何将文件在发送端转化为字节流，在接收端将字节流转化为完整的文件就成了最大的问题。一个文件主要包含两个内容：文件名，文件内容。为了构建包含文件名和文件内容的字节序列，并让接收端能够提取这两个内容，我们还需要标识文件名的长度。

byte byte		
文件名长度	文件名	文件内容

根据资料查询，Windows文件命名长度最长为237个字符，GBK编码后需要474bit，因此我们不妨使用两个字节来保存文件名的长度的高8位和低8位比特。给定文件名 `name`，我们计算这两个字节，并将其写入输入流当中：

```
// 写入文件名称长度
outputStream.write(name.getBytes().length / 256);
outputStream.write(name.getBytes().length % 256 - 128);
```

随后写入文件名称和文件内容：

```
// 写入文件名称
outputStream.write(name.getBytes());

// 写入内容
byte[] b = new byte[1024]; // 设置缓冲区
int len; // 设置内容长度

// 创建字节输出流，读取内容将它输出到outputStream对象中
FileInputStream fileInputStream = new FileInputStream(file);
while ((len = fileInputStream.read(b)) != -1) {
    outputStream.write(b, 0, len);
}
```

• 接收文件 (RecieveFile.java)

对于接收方，我们只需要从输入数据流中依次获取文件名长度、文件名、文件内容即可。

读取两个字节，并计算出文件名的长度：

```
// 获取名称长度
byte len1 = (byte) inputStream.read();
byte len2 = (byte) inputStream.read();
// 根据名称设置的规则，获取名称的长度
int fileLength = len1 * 256 + len2 + 128;
```

根据获得的长度，读取文件名：

```
// 获取文件名称
inputStream.read(data);
String fileName = new String(data, 0, fileLength);
```

随后创建一个文件名为 `fileName` 的空文件，从 `inputStream` 当中读取剩余的内容写入空文件中：

```
// 并在指定位置，创建文件
FileOutputStream fileOutputStream = new FileOutputStream(
    new File(folderPath + new String(data, 0, fileLength)));

// 写入内容
while ((fileLength = inputStream.read(data)) != -1) {
    fileOutputStream.write(data, 0, fileLength);
}
```


- 用户主线程代码

```
public static void main(String[] args) throws IOException,
InterruptedException {
    // 启动接收消息线程
    ServerSocket recieveMessageSocket = new ServerSocket(8601);
    new Thread(new RecieveMessage(recieveMessageSocket)).start();

    // 启动文件接收线程
    ServerSocket recieveFileSocket = new ServerSocket(9601);
    new Thread(new RecieveFile(recieveFileSocket, userOnePath)).start();

    // 等待对方就绪
    Thread.sleep(3000);

    // 启动发送消息线程
    Socket sendMessageSocket = new Socket("localhost", 8602);
    Socket sendFileSocket = new Socket("localhost", 9602);
    new Thread(new SendMessage(
        sendMessageSocket, sendFileSocket, userOnePath)).start();
}
```

- Chat 演示

左侧控制台为UserOne，右侧控制台为UserTwo，它们相互发送了一个文件给对方，并储存在他们对应的用户文件夹下。

UserOne (1) [Java Application] D:\Java\jdk1.8.0_91\bin\javaw.exe (2020年3月31日)	UserTwo (1) [Java Application] D:\Java\jdk1.8.0_91\bin\javaw.exe (2020年3月31日 下午8:49:58)
你好	对方 2020/03/31 20:50:35
对方 2020/03/31 20:50:53	你好
你好，请问你要做什么	你好，请问你要做什么
我想给您发送一个文件	对方 2020/03/31 20:51:03
对方 2020/03/31 20:51:09	我想给您发送一个文件
好的	好的
[file]1.txt	对方 2020/03/31 20:51:29
系统消息: 1.txt发送成功!	[file]1.txt
对方 2020/03/31 20:51:46	系统消息: 成功接收文件1.txt
谢谢，我收到了	谢谢，我收到了
不客气	对方 2020/03/31 20:51:54
对方 2020/03/31 20:52:08	不客气
我也给你发个文件吧	我也给你发个文件吧
对方 2020/03/31 20:52:12	[file]2.txt
[file]2.txt	系统消息: 2.txt发送成功!
系统消息: 成功接收文件2.txt	对方 2020/03/31 20:52:23
wow，我也收到啦	wow，我也收到啦

遇到的问题

编码问题：字节流的writeBytes()方法无法正常传输汉字，后来改为writeUTF()方法解决了汉字传输的问题。

Chat程序当中的端口使用问题：由于两个用户均在同一台主机上运行，因此对于收文件端口不能相同。一开始我没有发现这个问题，遇到莫名的端口占用错误，才意识到这一点。

Chat程序当中的TCP连接顺序问题：由于我实现的是基于 TCP 的 P2P 通信，因此用户既是服务器又是客户端。而服务器开放端口必然要先于客户端进行连接，否则会产生连接失败的异常。为了解决这个问题，我先打开了双方的服务端口，随后让两个用户共同休眠了3秒，再进行客户连接。这样就可以顺利地连接通信了。

总结

通过本次实验，我对Socket的理解更加深刻了。它是连接应用层和网络层的桥梁，而端系统中为我们提供了很多的API，使得我们使用Socket就像使用文件一样简单可靠。

附录

一 采用TCP进行数据发送的简单程序

TCPClient.java

```
package TCP;

import java.io.*;
import java.net.*;

class TCPClient {

    public static void main(String argv[]) throws Exception {

        // 建立到localhost:8888的连接
        Socket client = new Socket("localhost", 8888);
        System.out.println("--建立到服务器的连接--");

        // 从标准输入缓冲区读取用户数据
        BufferedReader userInput = new BufferedReader(new
        InputStreamReader(System.in));

        while (true) {
            // 从键盘读入一行
            String send = userInput.readLine();
            if(send.equals("exit")) break;

            // 向服务器发送
            DataOutputStream output = new
            DataOutputStream(client.getOutputStream());
            output.writeUTF(send);
            output.flush();

            // 从服务器获取数据
            DataInputStream dataInputStream = new
            DataInputStream(client.getInputStream());
            String serverSentence = dataInputStream.readUTF();
            System.out.println("FROM SERVER: " + serverSentence);
        }

        client.close();
    }
}
```

TCPServer.java

```
package TCP;

import java.io.*;
import java.net.*;
```

```

class TCPServer {

    public static void main(String argv[]) throws Exception {

        // 建立端口
        ServerSocket server = new ServerSocket(8888);
        System.out.println("--服务器开启成功--");

        // 监听端口
        Socket connection = server.accept();
        System.out.println("--建立TCP连接--");

        // 包装data输入输出流
        DataInputStream inputStream = new
        DataInputStream(connection.getInputStream());
        DataOutputStream outputStream = new
        DataOutputStream(connection.getOutputStream());

        while (true) {
            // 获取客户发送过来的信息
            String clientSentence = inputStream.readUTF();
            System.out.println("FROM CLIENT: " + clientSentence);

            // 向客户发送信息
            String send = "Hello, " + clientSentence;
            outputStream.writeUTF(send);
            outputStream.flush();
        }
    }
}

```

二 采用UDP进行数据发送的简单程序

UDPClient.java

```

package UDP;

import java.io.*;
import java.net.*;

class UDPClient {
    public static void main(String args[]) throws Exception {

        // 创建Socket, 并获取服务器IP
        DatagramSocket clientSocket = new DatagramSocket();
        InetAddress IPAddress = InetAddress.getByName("localhost");

        // 从标准输入缓冲区读取数据
        BufferedReader inputBuffer = new BufferedReader(new
        InputStreamReader(System.in));

        while (true) {
            String sentence = inputBuffer.readLine();

            // 封装UDP包

```

```

        byte[] sendData = new byte[1024];
        sendData = sentence.getBytes();

        // 发送包到目的IP的8888端口
        DatagramPacket sendPacket = new DatagramPacket(sendData,
        sendData.length, IPAddress, 8888);
        clientSocket.send(sendPacket);

        // 获取服务器发来的包
        byte[] receiveData = new byte[1024];
        DatagramPacket receivePacket = new DatagramPacket(receiveData,
        receiveData.length);
        clientSocket.receive(receivePacket);

        String recieveSentence = new String(receivePacket.getData(), 0,
        receivePacket.getLength());

        System.out.println("FROM SERVER:" + recieveSentence);
    }

    //    clientSocket.close();
}
}

```

UDPServer.java

```

package UDP;

import java.net.*;

class UDPServer {

    public static void main(String args[]) throws Exception {

        // 开放一个端口
        DatagramSocket serverSocket = new DatagramSocket(8888);
        System.out.println("--服务器准备接收Packet--");

        // 准备容器接收
        byte[] container = new byte[1024];
        // 等待包裹容器封包
        DatagramPacket receivePacket = new DatagramPacket(container,
        container.length);

        byte[] sendData = new byte[1024];

        while (true) {
            // 接收包裹
            serverSocket.receive(receivePacket);
            String clientMessage = new String(receivePacket.getData(), 0,
            receivePacket.getLength());
            System.out.println("FROM CLIENT: " + clientMessage);

            // 从UDP包中获取IP, 端口号
            InetAddress IPAddress = receivePacket.getAddress();
            int port = receivePacket.getPort();

```

```

        // 将要发送的字符串
        String send = "hello, " + clientMessage;

        // 将字符串转化为字节流
        sendData = send.getBytes();

        DatagramPacket sendPacket = new DatagramPacket(sendData,
        sendData.length, IPAddress, port);
        serverSocket.send(sendPacket);
    }
}
}

```

三 多线程/线程池对比

Client.java

```

package EchoServer;

import java.io.DataOutputStream;
import java.io.IOException;
import java.net.Socket;
import java.net.UnknownHostException;
import java.text.SimpleDateFormat;
import java.util.Date;

public class Client {

    private static Socket socket;

    public static void main(String[] args) throws UnknownHostException,
    IOException {

        // 获取当前日期，作为发送数据
        Date date = new Date();
        SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd
        HH:mm:ss");
        String data = formatter.format(date);

        // 建立TCP连接并发送数据
        socket = new Socket("localhost", 8888);
        DataOutputStream output = new
        DataOutputStream(socket.getOutputStream());
        output.writeUTF(data);

        // 关闭Socket
        socket.close();
    }
}

```

Process.java

```

package EchoServer;

import java.io.DataInputStream;
import java.io.IOException;

```

```

import java.net.Socket;

public class Process implements Runnable {

    private Socket client;

    public Process(Socket socket) {
        this.client = socket;
    }

    public static void handleClient(Socket client) throws IOException {

        // 包装data输入输出流
        DataInputStream inputStream = new
DataInputStream(client.getInputStream());

        // 获取客户发送过来的信息
        String message = inputStream.readUTF();
        System.out.println("FROM CLIENT: " + message);

        // 关闭资源
        inputStream.close();
        client.close();
    }

    @Override
    public void run() {
        try {
            handleClient(client);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

ServerThread.java

```

package EchoServer;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class ServerThread {

    public static void main(String[] args) throws IOException {

        // 服务器侦听端口
        ServerSocket server = new ServerSocket(8888);
        System.out.println("--服务器启动--");

        while (true) {
            Socket connection = server.accept();
            Thread thread = new Thread(new Process(connection));
            thread.start();
            System.out.println("创建线程: " + thread.getName());
        }
    }
}

```

```
}  
}
```

ServerPool.java

```
package EchoServer;  
  
import java.io.IOException;  
import java.net.ServerSocket;  
import java.net.Socket;  
  
public class ServerPool {  
  
    static int serverPort = 8888; // 服务器侦听的端口号  
    static int poolSize = 5; // 自定义线程池的大小  
  
    public static void main(String[] args) throws IOException {  
  
        ServerSocket server = new ServerSocket(serverPort);  
  
        System.out.println("--服务器开启--");  
  
        // 每个线程都反复循环，从（共享的）ServerSocket实例接收客户端连接。  
        for (int i = 0; i < poolSize; i++) {  
            Thread thread = new Thread() {  
                @Override  
                public void run() {  
                    while (true) {  
                        try {  
                            Socket socket = server.accept();  
                            Process.handleClient(socket);  
                        } catch (IOException e) {  
                            e.printStackTrace();  
                        }  
                    }  
                }  
            };  
  
            thread.start();  
            System.out.println("线程 " + thread.getName() + " 开始工作 ");  
        }  
    }  
}
```

四 写一个简单的chat程序，并能互传文件

UserOne.java

```
package FileTransfer;  
  
import java.io.IOException;  
import java.net.ServerSocket;  
import java.net.Socket;  
  
public class UserOne {
```

```

static String userOnePath = System.getProperty("user.dir") + "\\UserOne\\";

public static void main(String[] args) throws IOException,
InterruptedException {

    // 启动接收消息线程
    ServerSocket recieveMessageSocket = new ServerSocket(8601);
    new Thread(new RecieveMessage(recieveMessageSocket)).start();

    // 启动文件接收线程
    ServerSocket recieveFileSocket = new ServerSocket(9601);
    new Thread(new RecieveFile(recieveFileSocket, userOnePath)).start();

    // 等待对方就绪
    Thread.sleep(3000);

    // 启动发送消息线程
    Socket sendMessageSocket = new Socket("localhost", 8602);
    Socket sendFileSocket = new Socket("localhost", 9602);
    new Thread(new SendMessage(sendMessageSocket, sendFileSocket,
userOnePath)).start();
    }
}

```

UserTwo.java

```

package FileTransfer;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;

public class UserTwo {

    static String userTwoPath = System.getProperty("user.dir") + "\\UserTwo\\";

    public static void main(String[] args) throws IOException,
InterruptedException {

        // 启动接收消息线程
        ServerSocket recieveMessageSocket = new ServerSocket(8602);
        new Thread(new RecieveMessage(recieveMessageSocket)).start();

        // 启动文件接收线程
        ServerSocket recieveFileSocket = new ServerSocket(9602);
        new Thread(new RecieveFile(recieveFileSocket, userTwoPath)).start();

        // 等待对方就绪
        Thread.sleep(3000);

        // 启动发送消息线程
        Socket sendMessageSocket = new Socket("localhost", 8601);
        Socket sendFileSocket = new Socket("localhost", 9601);
        new Thread(new SendMessage(sendMessageSocket, sendFileSocket,
userTwoPath)).start();
    }
}

```


SendMessage.java

```
package FileTransfer;

import java.io.BufferedReader;
import java.io.DataOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.Socket;

public class SendMessage implements Runnable {
    Socket client;
    Socket sendFileSocket;
    String userPath;

    SendMessage(Socket client, Socket sendFileSocket, String userPath) {
        this.client = client;
        this.sendFileSocket = sendFileSocket;
        this.userPath = userPath;
    }

    @Override
    public void run() {
        try {
            // 从标准输入缓冲区读取用户数据
            BufferedReader userInput = new BufferedReader(new
InputStreamReader(System.in));

            while (true) {
                String send = userInput.readLine();
                // 向服务器发送
                DataOutputStream output = new
DataOutputStream(client.getOutputStream());
                output.writeUTF(send);

                if (send.length() > 6 && send.substring(0, 6).equals("[file]"))
                {
                    String fileName = send.substring(6);
                    new Thread(new SendFile(sendFileSocket, userPath +
fileName)).start();
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

RecieveMessage.java

```
package FileTransfer;

import java.io.DataInputStream;
import java.text.SimpleDateFormat;
import java.io.IOException;
import java.net.ServerSocket;
```

```

import java.net.Socket;
import java.util.Date;

public class RecieveMessage implements Runnable {
    ServerSocket server;

    RecieveMessage(ServerSocket server) {
        this.server = server;
    }

    @Override
    public void run() {
        try {
            // 监听端口
            Socket connection = server.accept();

            // 包装data输入输出流
            DataInputStream dataInputStream = new
DataInputStream(connection.getInputStream());

            while (true) {
                // 获取客户发送过来的信息
                String clientSentence = dataInputStream.readUTF();
                // 获取时间
                Date date = new Date(); // this object contains the current date
value
                SimpleDateFormat formatter = new SimpleDateFormat("yyyy/MM/dd
HH:mm:ss");
                System.out.println("对方 " + formatter.format(date));
                System.out.println(clientSentence);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

SendFile.java

```

package FileTransfer;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.net.Socket;

public class SendFile implements Runnable {

    Socket client;
    String filePath;

    public SendFile(Socket client, String filePath) {
        this.client = client;
        this.filePath = filePath;
    }
}

```

```

@Override
public void run() {
    try {
        OutputStream outputStream = client.getOutputStream();
        // 创建文件对象
        File file = new File(filePath);
        // 获取文件名称，为上传文件设置名称
        String name = file.getName();

        // 写入文件名称长度
        outputStream.write(name.getBytes().length / 256);
        outputStream.write(name.getBytes().length % 256 - 128);

        // 写入文件名称
        outputStream.write(name.getBytes());

        // 写入内容
        byte[] b = new byte[1024]; // 设置缓冲区
        int len; // 设置内容长度

        // 创建字节输出流，读取内容将它输出到outputStream对象中
        FileInputStream fileInputStream = new FileInputStream(file);
        while ((len = fileInputStream.read(b)) != -1) {
            outputStream.write(b, 0, len);
        }

        System.out.println("系统消息: " + name + "发送成功!");

        // 告知服务器输入完毕
        client.shutdownInput();
        // 关流
        fileInputStream.close();
        outputStream.close();
        client.close();

    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

RecieveFile.java

```

package FileTransfer;

import java.io.File;
import java.net.ServerSocket;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.net.Socket;

public class RecieveFile implements Runnable {

```

```

ServerSocket server;
String folderPath;

public RecieveFile(ServerSocket server, String folderPath) {
    this.server = server;
    this.folderPath = folderPath;
}

@Override
public void run() {
    try {
        Socket socket = server.accept();

        // 获取字节输入流
        InputStream inputStream = socket.getInputStream();

        // 获取名称长度
        byte len1 = (byte) inputStream.read();
        byte len2 = (byte) inputStream.read();

        // 根据名称设置的规则，获取名称的长度
        int fileLength = len1 * 256 + len2 + 128;

        // 设置缓冲区
        byte[] data = new byte[fileLength];
        // 获取文件名称
        inputStream.read(data);
        String fileName = new String(data, 0, fileLength);

        // 创建字节输出流
        // 并在指定位置，创建文件
        FileOutputStream fileOutputStream = new FileOutputStream(
            new File(folderPath + new String(data, 0, fileLength)));

        // 写入内容
        while ((fileLength = inputStream.read(data)) != -1) {
            fileOutputStream.write(data, 0, fileLength);
        }

        // 告知客户端接收完毕
        socket.shutdownInput();

        // 关流
        fileOutputStream.close();
        inputStream.close();
        socket.close();

        server.close();
        System.out.println("系统消息：成功接收文件" + fileName);

    } catch (IOException e) {
        e.printStackTrace();
    }
}
}

```

