# 网络编程概述

IO流中我们实现了内存层面与本地硬盘的数据交互。

那么硬盘，内存，网络层面的交互的需求诞生了网络编程。

假设现在要跟别人寄东西。

那么确定对方的地址就是第一个问题。

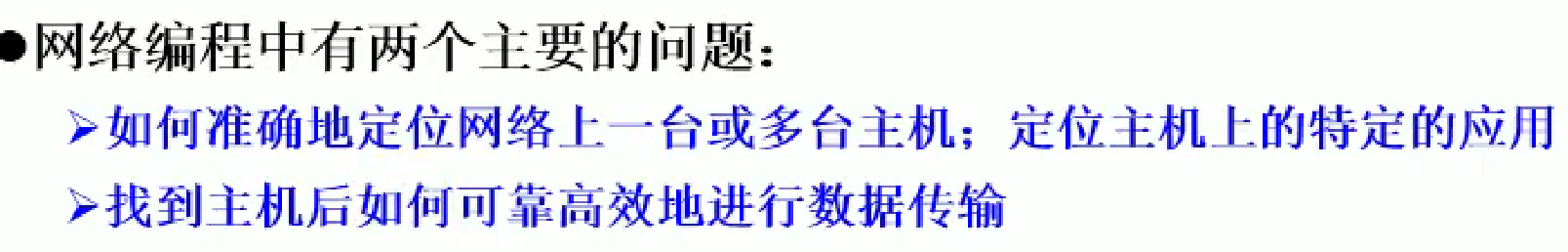
怎么将东西快速高效尽量不出错不损坏寄过去又是一个问题。

在计算机网络中我们知道发送方从应用层开始，从上往下，进行数据的封装，传递到下一层再根据下一层的协议封装，再传给下一层。

直到物理层相当于快递员把快递送过去。

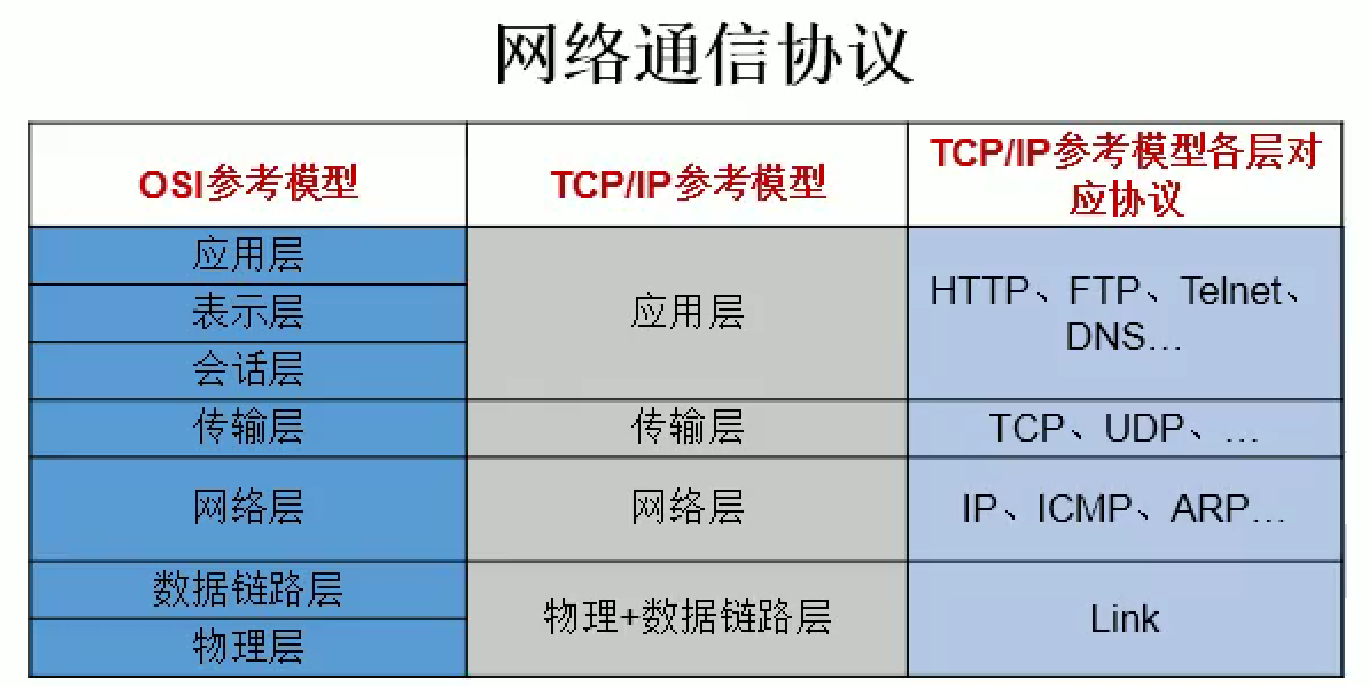
送到对方那里后，对方再根据每一层的协议，封装数据，拆封数据，差错检测，差错后重传等。

所以数据在网络中就是U型传输。

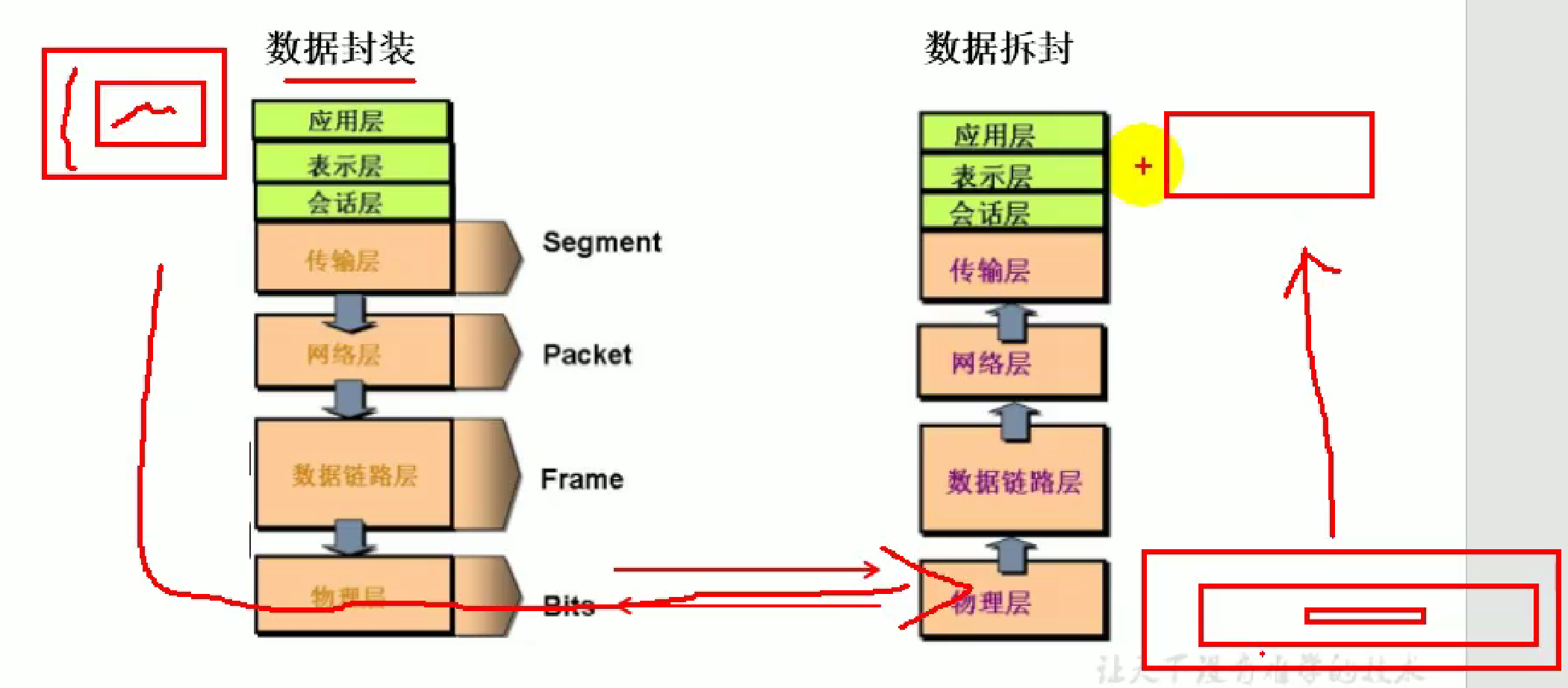


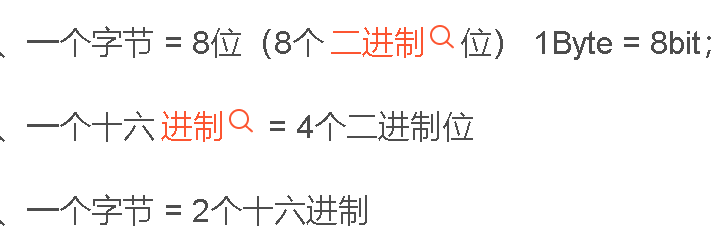


IP即为每台主机的地址。端口号即为主机上需要进行网络通信的应用所必须具备的东西。



实际应用为中间的5层协议。





一个位有0/1有两种情况 。

我们把一位叫做二进制。

此时一位就代表一个数。一个二进制数只有两种情况，最大表示1（从0开始）

2个二进制数就是2^2种情况

但是表示的数大的时候又臭又长不好书写。如11 11 11 11

于是如果将两位代表一个数，就叫做4进制。

一个4进制数包含2^2种情况，所以一个4进制数最大表示3（从0开始）.

2个4进制数就是(2^2)(2^2)种情况。

再上升，再精简，8进制数，三位代表一个8进制数。此时一个8进制数就代表2^2^2种情况。

2个8进制数代表（2^2^2）（2^2^2）种情况

再上升，16进制。4位代表一个16进制数。

此时一个16进制数就代表2^2^2^2种情况。

2个16进制数代表（2^2^2^2）（2^2^2^2）种情况

在计算机中，我们规定，一个字节用8个位表示。

所以一个字节=8个二进制数=4个4进制数=2个16进制数

IPV4使用的是4个字节来表示网络及主机地址，世界上大概有40亿个IP地址。但是已经不够用了，所以现在的IP都是动态IP。即你用的时候随机给你一个没用的。你用完之后再收回去给别人用。

现在IPV6是趋势，采用16个字节表示地址，分成8块。一块就有2个字节，1个字节2个16进制数，2个字节4个16进制数，所以每块用4个16进制数表示。



每个16进制数是2^2^2^2表示0到16，但是计算机中不认识10，11，12，13，14，15，所以将这6个数用a到f表示。

总体来看，16字节，16×8位，也就是可以表示2^(16×8)种情况也就是340万亿亿亿亿，可以将世界上所有事物分配一个IP地址，从而实现真正意义上的万物互联！

# InetAddress类

在通信中我们需要知道对方主机的标识-IP地址，才能将数据传输过去。万事万物皆对象，这个IP地址在java代码中我们定义了InetAddress类来存储IP地址。

就类似于File，Path类存储文件路径一样。

域名：由于IP地址数字难记，为了用户好记，现在都使用域名代替IP地址，如[www.baidu.com](http://www.baidu.com)

其中本机域名叫做localhost，本机回路地址也叫本机对本地的服务器是127.0.0.1，但是如果计算机介入了万维网或者局域网，本机对外的IP地址就是被分配的一个IP了。对内还是127.0.0.1

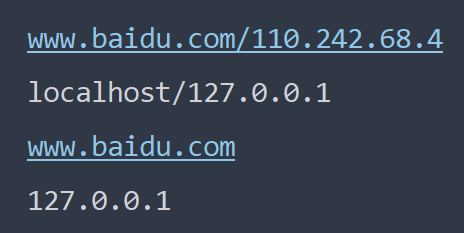
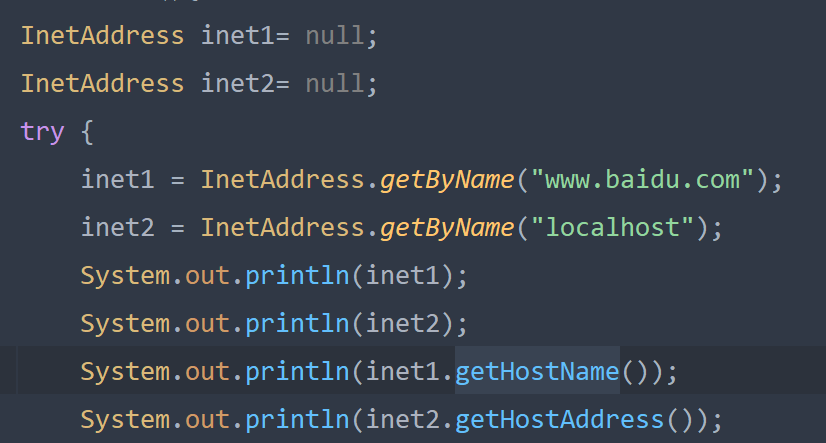
由于InetAddress隐藏了构造函数，所以提供了方法getByName（“xxx”）来获取对象。其中方法形参中应该可以填入IP地址或者域名

当填入IP地址时，就直接将IP地址存入对象属性种。

填入域名时，实例化对象时，先在本地host文件种查找该域名是否有对应的IP地址，若没有，就会交给DNS服务器解析该域名，并将解析出来的IP地址返回，存储到属性中。如果域名是错误的则直接抛出异常

InetAddress对象提供了方法getHostName（）用来返回当前对象的域名。getHostAddress返回当前对象的IP地址。

默认sout 对象输出其域名+IP



# 端口号

当一台计算机与另一台计算机通信时，必定是应用层面的通信，通过IP地址锁定了对方的主机，那么对方主机上运行着很多应用程序，怎么找到对应通信的应用呢？

所以每个运行着的应用都会占用计算机的一个端口号

端口号为16位2字节的整数 即0-65535

但是0到1023被系统服务层面占用。

所以第三方的软件一般都会使用1024到65535 比如MySql默认3306 Tomcat 8080

同时注意，一个端口号只能被一个应用占用，比如安装了两个Tomcat，第一个启动了。第二个就会启动失败，原因是端口被占用，此时将第二个的默认端口号改为8080以外的就可以了。

在网络中，IP地址+应用端口号组成一个网络套接字：Socket

网络编程也叫做Socket编程，即将信息封装为Socket对象。

# TCP/UDP

属于运输层协议，数据从一台主机到另一台主机需要IP地位主机，然后端口号定位应用。定位之后开始建立链接传输数据，这时需要遵守TCP/UDP协议后才能传输。

**TCP协议：**

连接时需要进行3次握手：

1. 主机1向主机2发送信息：我是主机1，我要与你链接。
2. 主机2向主机1发送信息：我是主机2，我知道你要与我建立链接。
3. 主机1向主机2发送信息：我是主机1，我知道 你知道我要与你建立链接。
4. 建立链接。

3次握手的原因是这样双方立接通的概率在99%以上。如果仅仅是一次握手，主机2不一定在，二次握手，主机1不一定还在，3次保证性就大了。

断开时4次握手：断开时双方任意一方都可以主动断开。

1. 主机1向主机2发送信息：我是主机1，我要断开链接。
2. 主机2向主机1发送信息：我是主机2，我知道你要与我断开链接。
3. 主机2向主机1发送信息：我是主机2，我已经断开了与你的链接。
4. 主机1向主机2发送信息：探测主机2是否已经断开链接，如果发送后主机2还发回来，就让主机2再断开，如果主机2已经无响应，说明已断开，此时这个链接才算真正断开。

**UDP协议：**

UDP与TCP最大的区别就是不需要建立链接。

1. 主机1每次将数据封装成最大64K的数据报。可多次封装。
2. 主机1的运输层直接定位到主机2的运输层发送过去，不管主机2能不能收到。

所以TCP协议适合在重要场景中比如微信电话使用，如果对方网络不通畅，就会卡住，然后声音再接着前面的话穿过来。打电话也是。

而UDP适合实时的网络电视，比如在线看球赛，实时转播的奥运会，春晚等，这要求可以丢失几帧画面，不影响观感。

QQ也是UDP，毕竟发送的时候对方不在线也能发送。等对方上线了服务器自会帮我们发送过去。

# TCP测试

## 测试1传输字符串

客户端发送字符串到服务端，并在服务端的控制台显示。

我们第一想到的是客户端的向哪里传送。，传送什么。怎么传送。

客户端

1. 将目标服务器的IP和应用端口号封装为一个Socket对象。

Socket socket=new Socket（IP+端口号）。

那么还需要一个InetAdress对象来封装服务器的IP

InetAddress.getByName（127.0.0.1,8099）。因为是测试所以自己发给自己。将这个对象放入socket构造器。

1. 要向服务器传递一句话，需要IO流写出到网络，那么IO流的目的地址就是网络中的服务器。怎么实现OutputStream输出目标是服务器端呢？我们先将服务器的IP和端口号实例化为一个Socket对象。

Socket socket=new Socket（IP，8099）

再用socket对象返回一个OutputStream对象，再调用write（）写出的数据目的地址就是Socket中封装的IP+端口了。

即写出到服务器。

关闭流和Socket对象。

服务器端要接收数据，但是前提是自己也是一个Socket，这样客户端发送过来的数据才能与其建立链接。

服务器：

1. 既然自己是接收端，就需要将自己的端口和IP实例化为一个ServerSocket对象存储在网络中，等待对面的握手。

但是这种对象特点就是不用将IP放入构造器，因为作为接收端默认IP就是自己的IP。

ServerSocket ss=new ServerSocket(8099)

此时作为接收端，握手以后要从网络中接收客户端写过来的数据，调用ss.accpet()方法，返回一个新的Socket对象，用来存储客户端的信息，如果没有客户端握手，就一直卡在accept()这里等待客户端，相当于一个阻塞方法。

之所以不用ServerSocket接收客户端的原因是客户端不止一个可能，所以每次需要返回新的Socket对象。

并且可以多次使用accept（）方法返回Socket对象，等待多个客户端链接。

1. 当一旦有客户端握手时，说明此客户端要与自己建立链接了，返回客户端的Socket对象后，调用socket.getInputStream（）返回一个客户端的输入流，因为数据是从客户端来的，所以InputStream流的构造器中应该是客户端，同输出时，直接使用客户端的socket.getInputStream()返回的就是输入源是客户端的IO流了。即从客户端输入。
2. 关闭流。

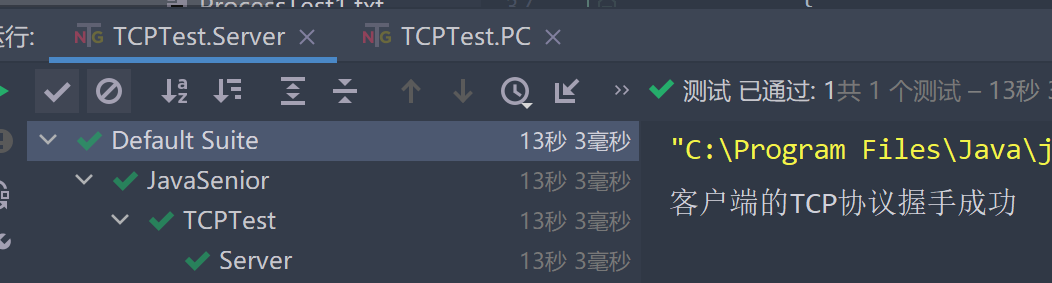
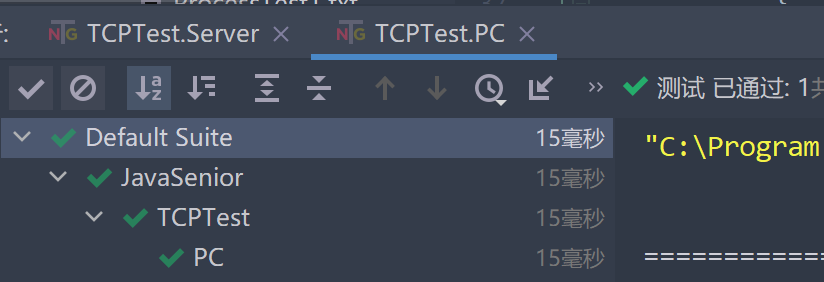
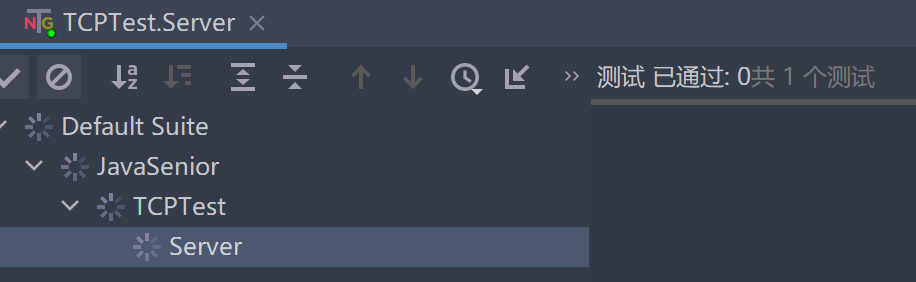
即只有客户端使用目标Socket成功定位到服务器并且服务器调用accept（）等待客户端。然后双方握手之后，客户端的write（）才执行，服务器端的read（）也才能执行。

实际上只要客户端实例化服务端的对象一运行，jvm立马去网络中找对应的服务器去握手，跟客户端这边调用write（）没关系。

而服务器这边accept方法等待中，一旦握手成功，就返回客户端Socket对象。

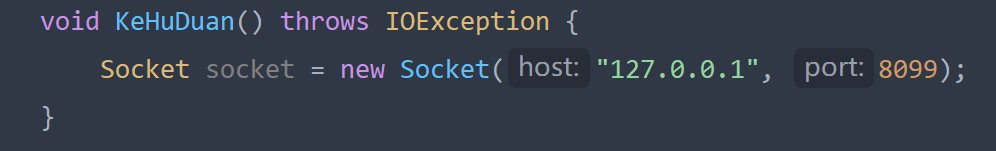
运行时先将服务器端的测试方法运行起来，再运行客户端的测试方法。



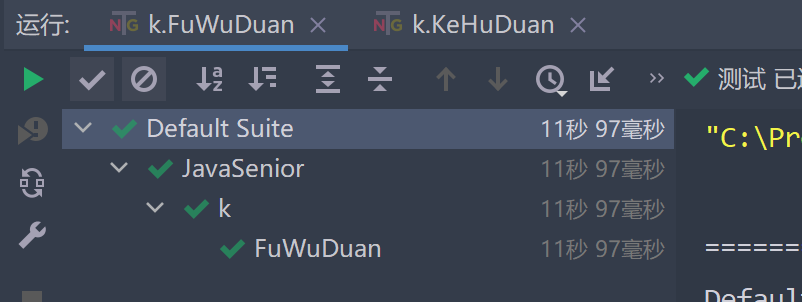
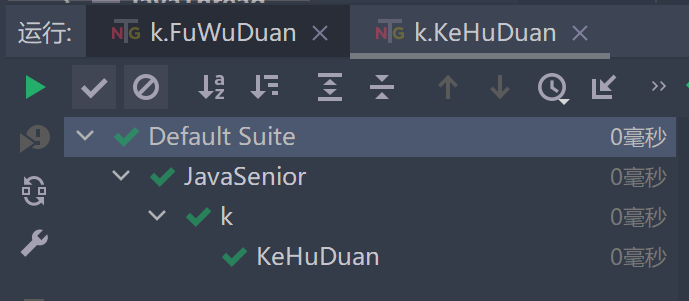
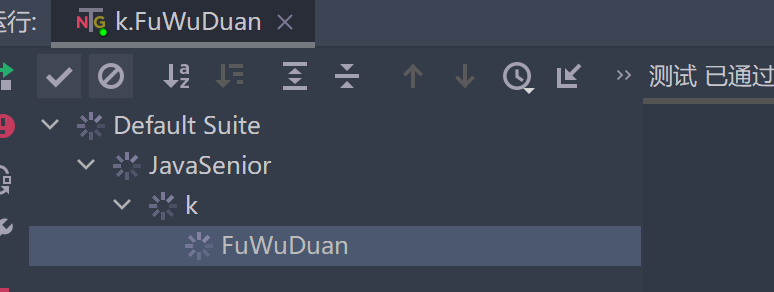
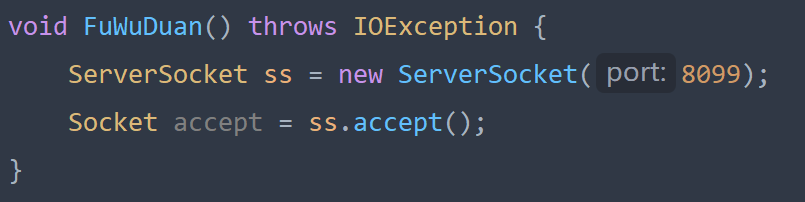


## 补充

事实上，客户端实例化Scoket对象写入服务器端的IP+端口号之后，测试方法一运行，Jvm立马通过网络寻找此IP+端口号。



而服务器端只要成功与过来的客户端握手后，accept方法立马返回客户都对象。

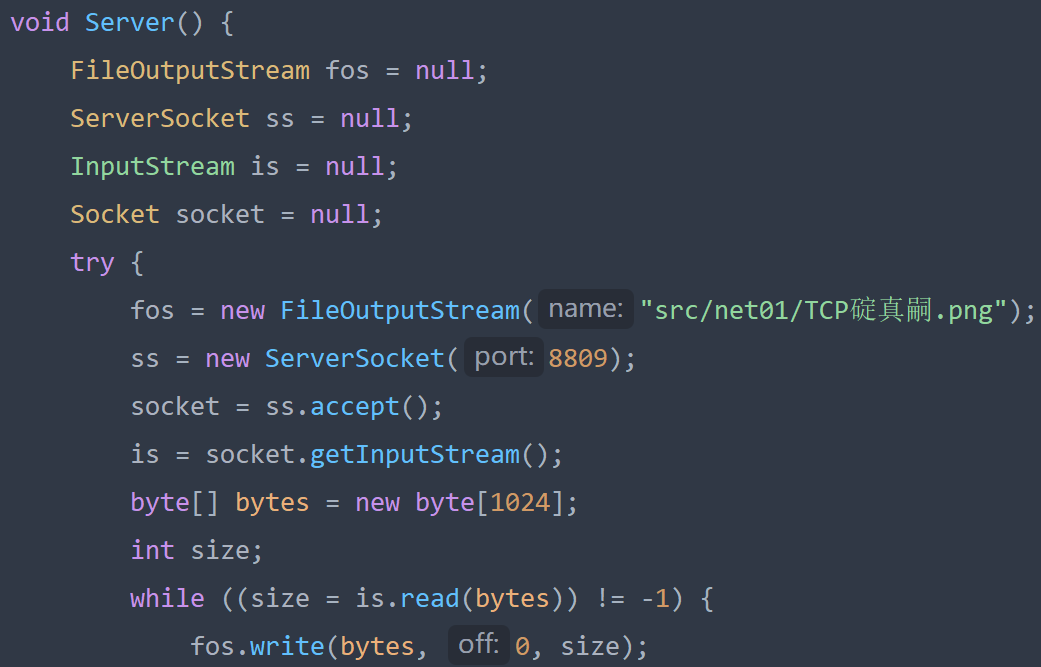


可以看到客户端根本就没有任何其他的操作，只是实例化了服务器端的对象，jvm就会去握手。即实例化这个对象的意义在jvm看来就是让他直接立刻马上去通过网络去跟IP+端口号去握手。

## 测试2传输图片

**首先是服务器端：**

1. 实例化自己的ServerSocket对象，指定自己的端口号。
2. ss.accept()等待客户端通过jvm找到自己并与自己握手。返回客户端Socket对象。
3. socket.getInputStream()返回输入流，将客户端写出到网络中的byte再写入到服务器端。
4. 实例化FileOutputStream对象，指定本地路径。
5. byte[] bytes，int size，循环，is.read( bytes)，再使用fos.write(bytes,0,size)写出到服务器端的本地路径中。
6. 关闭流。



**客户端：**

1. 实例化目标Socke，实例化目标IP InetAddress（“xx.xx.xx.xx”）与目标应用端口号。
2. jvm迅速通过网络寻找IP+端口对应的应用，一旦握手成功，服务器端执行accept方法，客户端则开始写出数据。
3. 客户端写出数据。实例化FileInputStream对象将本地路径的图片写入到内存。byte[] bytes,int size

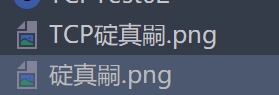
fis.read(bytes)

将内存中数据写出到网络 socket.getOutputStream

循环 并在循环中使用os.write(bytes，0，size)写出

关闭流。





## 测试3

原理解释：

Socket作为主机IP+端口号的实例化对象。那肯定是代表网络中的通信的一方的信息的。

而实际上客户端要向服务器端写出数据时。要实例化服务器端的Socket，客户端自己的Socket，是不需要我们操心的，java自动把客户端的socket发送出去给服务器端。

对于客户端来说，实例化服务器端的Socket对象，当jvm执行这一条语句时，迅速通过网络去寻找对应的服务器的应用。

而往服务器写出数据，调用的也是服务器socket对象的write方法。这就有点别扭，明明是客户端，却实例化的是服务器端的socket，调用的是服务器端socket的write方法。

其实可以这么理解：我们以前调用FileOutputStream对象的write方法的时候，前提是将File对象放入构造器中，write方法才能有目标去写出。而现在，目标就是网络中的服务器。所以，看似是通过服务器Socket对象.getOutputStream返回一个输出流，实际上可以看作是将服务器的Socket对象放入了OutputStream的构造器中，并实例化了输出流对象。

这就很好理解了。由于write方法必须指定目标。所以我们需要实例化服务器的socket，生成write方法，此write方法目标就是服务器。

对于服务器端也是一样的道理，但是我们必须先实例化服务器自己的socket，因为客户端会来与自己握手。就是与自己的socket匹配信息对不对。

那么握手之后呢，我们需要将网络中的信息写入，但是，同样，我们以前的FileInputStream构造器中必如果多个客户端与自己握手，我怎么知道网络中这么多数据，哪些数据属于哪个客户端？

正好服务器的socket有个方法叫accept（），意为阻塞服务器端进程，等待客户端与自己握手，返回客户端的socket，并继续下去。

既然返回了客户端的socket，我们拿此客户端的socket返回一个InputStream，不就相当于这个输入流输入的数据，是来自此客户端的吗？

完美解决。

所以大体路线就是这样的：客户端实例化服务器端的Socket，返回输出流。写出数据到服务器。

服务器端实例化自己的socket握手使用，并调用accept方法返回客户端的socket，返回其输入流，将数据输入进来。

## read()阻塞问题

输入流的read()方法其实是一个阻塞方法，即在并明确指出读取的数据还有没有时，read方法永不会返回-1 whle循环也一直循环。

我们FileInputStream读取本地文件时，读取到文件末尾时会有一个定界符/n，read（）读取到此符号时，就会认为没有数据可以读了，返回-1.

但是在网络编程中。服务器read（）的源地址是客户端，如果客户端那边没有明确说明不再向服务器写数据了，那么服务器的read（）永不返回-1.

标红的是socket返回的方法（服务器或客户端的socket）

假设客户端用服务器的socket的输出流的write（）向服务器写出了数据。服务器while循环用客户端socket的输入流的read接收。在服务器这边，继续写代码，用客户端socket的write（）向客户端写出数据。客户端也追加代码，while循环用服务器socket的read（）读取数据。

此时就会造成双双进死锁程永不结束。

原因：服务器接收客户端写过来的数据时，客户端并没有明确指出我不再写出数据了。并且客户端在传送玩数据之后，没有关闭所有的流与socket。而是继续接收服务器的数据，while+read。

服务器那边读客户端发过来的信息时，没有收到客户端的指令我不再写数据了，并且客户端也没有断开连接，所以read一直循环（根本原因是read是客户端socket调出来的）

服务器刚开始就一直循环了。自然也就执行不了下面的向客户端发送数据的代码。而且由于死循环服务器也没法断开连接。

所以客户端也不知道服务器到底还写不写数据。此时也卡死在while循环中read方法。（这个read也是服务器socket调出来的）。

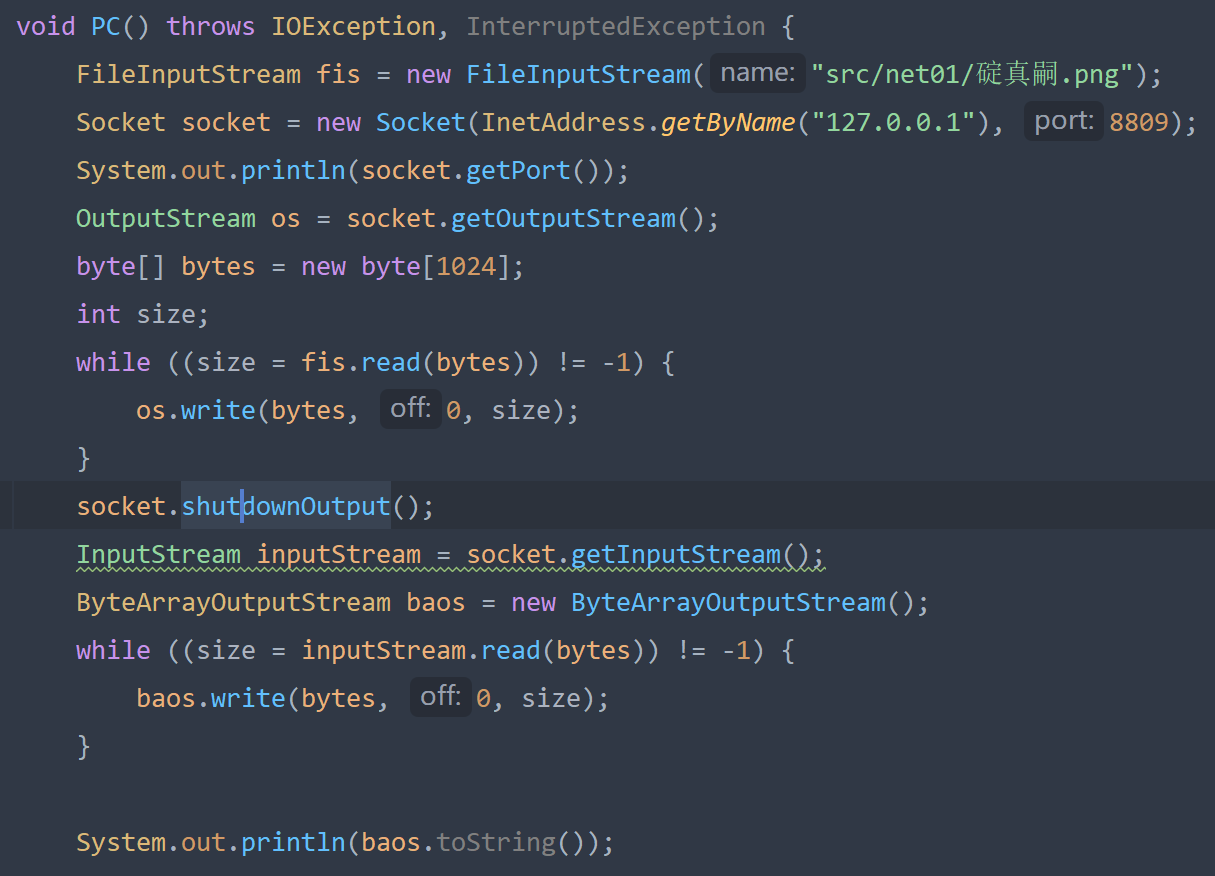
解决方法：

我们需要在客户端发送完数据之后，告诉服务器，我发送完了，不再发送了，所以在客户端的第一个while从本地读完数据再write之后，第一次while循环的大括号之外，加一个socket.shutdownOutput。代表的就是写向服务器的输出流，关闭。

这样服务器那边就知道了，这个客户端不再写数据了，于是从客户端读取数据的read也就返回-1了。下面的代码就可以执行了。

服务器端继续执行代码，向客户端写出数据，写出之后，直接就将流关闭了，并且断开链接。

客户端那边用while循环read方法读完服务器发的数据之后，检测到服务器直接关了，read也就返回-1了。客户端继续执行下面的代码，关闭自己的所有流。





# Tomcat与浏览器

cmd命令启动tomcat后，可以打开浏览器，访问localhost：8080，此时浏览器相当于客户端，服务器就是Tomcat，浏览器通过输入IP+8080锁定Tomcat位置并去访问。可以访问到Tomcat的主页面。

打开Tomcat的example文件夹，放入txt文本文件，通过浏览器在原有路径之后加上/example/xxx.txt则可以直接访问到文本内容。

后期我们可以通过Tomcat实现与数据库的交互。即后台主程序处理数据的代码交给Tomcat。前台浏览器提供数据。Tomcat接收到前台数据之后利用代码与数据库等交互计算出新的数据。再返回给前台。

一般来说程序的网页前台随时可以打开，Tomcat是一直开启的，除非我们手动关闭。

# UDP测试

因为不同于TCP握手后大数据传输，UDP只能每次发送不超过64KB的数据报。

**客户端：**

首先定义一个DatagramSocket对象，作为Socket。并将对方的

定义一个数据报对象。将服务器的IP 端口号放入构造器中。

将数据封装到数组中，再将数组放入数据报对象构造器中。

调用ds.send将数据报发送出去。

流关闭。

**服务器端：**

实例化DatagramSocket对象，并将自己端口号放入构造器。

实例化数据报对象，将byte数组放入。

调用ds.recive(数据报对象)

此时数据已经被存入到数据报中了（本质是在bytes中）。

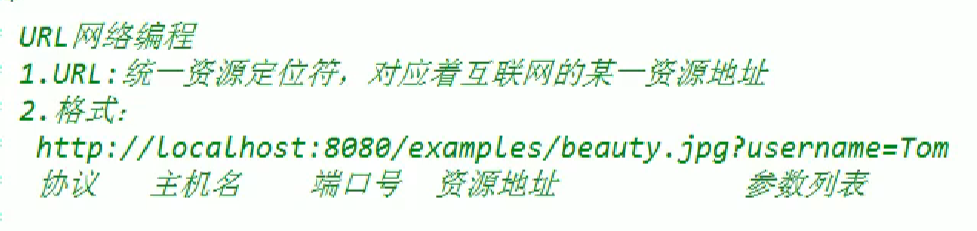
此时如果我们想输出数据。但是不知道写道数组的哪个位置了。

就可以sout(new String(数据报.getData,0,数据报.getLength))。

流关闭。



# URL



平时我们使用的种子，大小只有B级别。

是因为资源是保存在服务器中的，要想获取资源，需要知道服务器的IP，端口，资源的具体目录，以及操作该资源所须的密码账户等，这些信息都封装在一个种子中。

种子只是供我们去跟服务器链接。

**客户端**

假设我们需要获取Tomcat的一个资源，然后再写出到本机。

实例化URL对象，构造器中写入url路径（协议，主机端口，具体地址，参数列表）

注意，资源地址不能有中文。

调用url的openConnection方法得到一个链接对象。并强转为HttpURLConnection

调用连接对象的connect方法开始建立连接。

调用连接对象的getInputStream方法得到从服务器读取数据的流。

实例化FileOutputStream对象向本地中写出数据。

int size。byte[] bytes。

while循环。

关闭流。

**服务器端：**

提前在Tomcat的bin目录下双击start.bat运行Tomcat服务器。

