# 1. 网络编程概述

网络编程用于实现网络互连的不同计算机上运行的程序间进行数据交换。计算机网络体系结构：TCP/IP模型。建议先学习计算机网络知识。

网络通信的三要素：

（1）IP地址：网络中设备的标识。用于找到设备。

（2）端口号：设备中不同软件进程的标识，用于找到具体的应用进程。

（3）通信协议：就是通信的规则，常用的是TCP和UDP协议。我们编程关心的是应用层的协议。

网络编程还会涉及到IO和多线程的知识，IO用于数据的传输，多线程是为了处理多个网络请求。

# 2. InetAddress和套接字Socket

InetAddress其实就是Java封装了IP地址对象，而套接字（Socket）就是IP值和端口号的组合。通信的两端是通过套接字Socket进行连接，两个Socket间通过IO方式传输数据。

Java中，UDP和TCP的客户端/服务端Socket是有所区别的，下面讲。这里先讲下InetAddress，因为套接字Socket是要用到InetAddress的。

InetAddress类没有构造方法，可以使用类中的静态方法getByName()得到该类的对象：

public static InetAddress getByName(String host)：根据主机名或者IP地址的字符串形式得到InetAddress对象。

之后，通过getHostAddress()方法可得到此对象对应的主机地址；通过getHostName()方法得到主机名。

例子：

|  |
| --- |
| **package** com.zhang.test;  **import** java.net.InetAddress; **import** java.net.UnknownHostException;  **public class** Demo {  **public static void** main(String[] args) **throws** UnknownHostException {  InetAddress ipObj = InetAddress.*getByName*(**"192.168.1.1"**);  System.***out***.println(ipObj.getHostName());  System.***out***.println(ipObj.getHostAddress());  } } |

# 3. UDP通信

UDP协议是用户数据报协议，是无连接的传输层协议。

UDP通信不分服务端与客户端，只分为发送端与接收端。UDP通信的要点是：创建DatagramSocket，建立DatagramPacket数据包，最后调用receive()方法或者send()方法接受或者发送数据包。

发送端Socket连接接收端Socket就能建立连接。而接收端需要监听自己的端口，等待发送端连接，管理很多过来的请求。

下面是一个UDP简单的例子。主要是要看例子！

（1）UDP接收端程序：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.IOException; **import** java.net.DatagramPacket; **import** java.net.DatagramSocket; **import** java.net.InetAddress;  **public class** MainReceive {  **public static void** main(String[] args) **throws** IOException {  *// 1. 创建UDP接收端Socket。  // 用DatagramSocket类，只要指明监听的本机端口即可。这里用10000* DatagramSocket ds = **new** DatagramSocket(10000);  *// 2.创建一个数据包作为接收数据的容器。  // 需要新建一个字节数组传递进去，并指明数组长度* **byte**[] data = **new byte**[4 \* 1024]; *// 4K的大小* DatagramPacket dp = **new** DatagramPacket(data, data.**length**);  *// 3. 用DatagramSocket接收数据到数据包中，用receive方法* ds.receive(dp);  */\*  API对receive()方法的解释：  从此套接字接收数据报包。当此方法返回时，DatagramPacket的缓冲区填充了接收的数据。数据报包也包含发送方的 IP 地址和端口号。此方法在接收到数据报前一直阻塞。数据报包对象的length 字段包含所接收信息的长度。如果信息比包的长度长，该信息将被截断。  \*/  // 也就是说此时若发送端传递了数据，那么DatagramPacket中不仅存放了数据，还存放了发送端信息   // 获得发送端的IP地址和端口信息：* InetAddress ip = dp.getAddress();  **int** port = dp.getPort();  *// 获得传送来的数据* **byte**[] receiveData = dp.getData();  *// 传送过来的是普通文本的话，可解析为字符串。注：dp.getLength()获得的是有效长度* String dataStr = **new** String(receiveData, 0, dp.getLength());   *// 输出上述的信息* System.***out***.printf(**"客户端地址：%s，端口：%d，数据：%s"**, ip.getHostAddress(), port, dataStr);  ds.close(); // 关闭  } } |

（2）发送端程序：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.IOException; **import** java.net.\*;  **public class** MainSend {  **public static void** main(String[] args) **throws** IOException {  *// 1. 建立UDP发送端Socket，还是用DatagramSocket  // 发送端不需监听。直接创建对象。发送时才指定连接的Socket信息* DatagramSocket ds = **new** DatagramSocket();  *// 2. 同样，创建DatagramPacket对象用于设置发送的信息  // 使用的构造为：public DatagramPacket(byte[] buf, int length, InetAddress addr, int port)。正好包含了以下信息：数据、长度、远程主机IP和端口号* **byte**[] buf = **"hello 你好啊"**.getBytes();  DatagramPacket dp = **new** DatagramPacket(buf, buf.**length**, InetAddress.*getByName*(**"127.0.0.1"**), 10000);  *// 调用Socket的send()方法发送数据包* ds.send(dp);  *// 释放Socket连接* ds.close();  System.***out***.println(**"已发送！"**);  } } |

上述程序应该先运行MainReceive程序，因为recevice方法会阻塞，也就是如果没有程序连接过来，程序会一直等待。所以先运行MainReceive程序，再运行SendReceive程序发送数据。这样MainReceive程序中就会打印出传递过来的信息。注意的是，MainReceive程序使用的端口是10000，但是MainSend程序也有自己的端口，所以MainReceive输出的链接过来的程序的端口不是10000，而是系统自动分配的，这要清楚。

中文接收是没问题的，因为是字节全部转换的。

上述中，MainReceive只能监听一次程序数据的传递，一旦MainSend程序运行，MainReceive接收到数据后也会停止程序。我们可以使用while循环来让MainReceive一直receive进行监听。

在代码上看来，UDP不区分服务端和客户端，因此在编程时，服务端和客户端如果要互相传输数据，都是通过receive()和send()方法来接收和发送数据。下面是示例：

实现的功能为：服务端在循环中不断等待接收数据，若成功收到数据则给此客户端发送反馈数据；而客户端发送完数据并得到反馈后就结束程序。

服务端代码：

|  |
| --- |
| **package** com.demo.udp;  **import** java.io.IOException; **import** java.net.DatagramPacket; **import** java.net.DatagramSocket; **import** java.util.Arrays;  **public class** Server {  **public static void** main(String[] args) **throws** IOException, InterruptedException {  DatagramSocket ds = **new** DatagramSocket(10000);  **byte**[] data = **new byte**[1024]; *// 一次课接收/发送1024个字节* DatagramPacket dp = **new** DatagramPacket(data, data.**length**);  **while** (**true**) {  *// 在循环中接收和发送数据。* ds.receive(dp);  System.***out***.println(**new** String(dp.getData(), 0, dp.getLength())); *// 打印接收到的数据  // 向客户端发送反馈信息。发送的流程同样是设置数据并调用send()发送  // 但注意：  // 1. 发送反馈给客户端时，需要使用同一个DatagramPacket，即上面的dp，因为dp中有对方的信息，否则不知道对方是谁。  // 2. 不能直接使用dp.setData("server received".getBytes())这种方式  // 原因是一旦这样做，dp中可存储的数据长度只能是15个字节了。这样下次客户端发送超过15个字节的数据就会被截取。  // 因此我们要保留dp中可存数据的长度。我们通过将要发送的字节数据“拷贝”到原来的data数组中实现保留长度。* Arrays.*fill*(data, (**byte**) 0); *// 需要先把data先全部清空！！！否则下面可能会发送多余数据* **byte**[] sendData = **"server received"**.getBytes(); *// 待发送的数据* System.*arraycopy*(sendData, 0, data, 0, sendData.**length**); *// 拷贝数组到data中* dp.setData(data); *// 重新设置data数据* ds.send(dp); *// 发送* Thread.*sleep*(200);  }  *//ds.close(); // 上面用了while(true)，这里执行不到了...* } } |

客户端代码：

|  |
| --- |
| **package** com.demo.udp;  **import** java.io.IOException; **import** java.net.DatagramPacket; **import** java.net.DatagramSocket; **import** java.net.InetAddress;  **public class** Client {  **public static void** main(String[] args) **throws** IOException {  DatagramSocket ds = **new** DatagramSocket();  **byte**[] buf = **"我们建立了连接，传输了数据"**.getBytes();  DatagramPacket dp = **new** DatagramPacket(buf, buf.**length**, InetAddress.*getByName*(**"127.0.0.1"**), 10000);  ds.send(dp); *// 发送数据* System.***out***.println(**"数据已发送"**);  **byte**[] data = **new byte**[1024]; *// 也规定一次可传输1024字节* dp.setData(data);  ds.receive(dp); *// 接收数据。  // 下面接收数据的处理比较简单，只要得到获取的数据即可。* System.***out***.println(**"收到服务器数据："** + **new** String(dp.getData(), 0, dp.getLength()));  ds.close();  } } |

# 4. TCP通信

TCP是面向连接的、可靠的协议。一般连接的双方一个充当服务器，一个充当客户端。客户端请求服务器相应数据，而服务器返回给客户端数据。当然，两端都能发送和接收数据。

TCP通信的要点就是：创建Socket；拿到Socket中的输入/输出流；发送/接收此数据。

两端Socket有所区别，即服务端Socket（ServerSocket）和客户端Socket。

例子：

（1）TCP接收端：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.BufferedReader; **import** java.io.IOException; **import** java.io.InputStream; **import** java.io.InputStreamReader; **import** java.net.ServerSocket; **import** java.net.Socket;  **public class** MainReceive {  **public static void** main(String[] args) **throws** IOException {  *// 1. 创建服务器端Socket，指定监听端口。  // 这里使用的是ServerSocket类* ServerSocket ss = **new** ServerSocket(10000);  *// 2.用accept方法监听客户端连接，此方法返回一个TCP客户端Socket对象。直接用Socket类型接收  // 若无连接，则一直阻塞。* Socket client = ss.accept();  *// 与UDP不同，此时Socket中就有了需要的所有信息   // 获得IP和端口* String ip = client.getInetAddress().getHostAddress();  **int** port = client.getPort();  System.***out***.printf(**"IP：%s，端口：%d\n"**, ip, port);  *// 获得数据的输入流来读取传递过来的数据。* InputStream is = client.getInputStream();  *// IO方式读取数据。这里用BufferedReader包装一下，防止中文乱码* BufferedReader br = **new** BufferedReader(**new** InputStreamReader(is));  String data = **null**;  **while**((data = br.readLine()) != **null**) {  System.***out***.println(data);  }  *// 关闭client和server的Socket* client.close(); *// 会自动关闭上述的流* ss.close();  } } |

（2）TCP发送端：

|  |
| --- |
| **package** com.zhang.test;  **import** java.io.IOException; **import** java.io.OutputStream; **import** java.net.Socket;  **public class** MainSend {  **public static void** main(String[] args) **throws** IOException {  *// 1. 创建客户端Socket，指定服务器的IP和端口* Socket s = **new** Socket(**"127.0.0.1"**, 10086);  *// 拿到Socket的输出流，写数据* OutputStream os = s.getOutputStream();  os.write(**"TCP 你好啊"**.getBytes());  *// 释放资源* s.close();  } } |

Socket也能拿到输出流OutputStream也能拿到InputStream，可以互相发送数据。我们在这里同样用TCP实现数据交互的示例：

实现的功能为：服务端在循环中不断等待接收数据，若成功收到数据则给此客户端发送反馈数据；而客户端发送完数据并得到反馈后就结束程序。

服务端代码：

|  |
| --- |
| **package** com.demo.tcp;  **import** java.io.\*; **import** java.net.ServerSocket; **import** java.net.Socket;  **public class** Server {  **public static void** main(String[] args) **throws** IOException, InterruptedException {  ServerSocket ss = **new** ServerSocket(10000);  **while** (**true**) {  Socket client = ss.accept();  *// 获得数据* InputStream is = client.getInputStream();  BufferedReader br = **new** BufferedReader(**new** InputStreamReader(is));  String data = **null**;  **while** ((data = br.readLine()) != **null**) {  System.***out***.println(data);  }  client.shutdownInput(); *// shutdownInput()，重要！！！  // 发送反馈数据* OutputStream os = client.getOutputStream();  os.write(**"data received"**.getBytes());  client.close();  Thread.*sleep*(200);  }  *// 关闭ServerSocket  //ss.close();* } } |

客户端代码：

|  |
| --- |
| **package** com.demo.tcp;  **import** java.io.\*; **import** java.net.Socket;  **public class** Client {  **public static void** main(String[] args) **throws** IOException {  Socket s = **new** Socket(**"127.0.0.1"**, 10000);  *// 1. 通过输出流和write()发送数据* OutputStream os = s.getOutputStream();  os.write(**"TCP 你好啊"**.getBytes());  s.shutdownOutput(); *// shutdownOutput()，重要！！！  // 2. 再通过输入流和read()读取反馈的数据* InputStream is = s.getInputStream();  BufferedReader br = **new** BufferedReader(**new** InputStreamReader(is));  String data = **null**;  **while** ((data = br.readLine()) != **null**) {  System.***out***.println(data);  }  *// 释放资源* s.close();  } } |

值得注意的是，在上述的服务器和客户端代码中，如果要在接收（或发送）后，还需要再发送（或接收）数据，那么必须要在此之前调用shutdownInput()或者shutdownOutput()方法，表示当前接收（或发送）数据操作已经完毕，可以进行下一波数据传递。否则，如果不使用shutdownInput()或者shutdownOutput()，将导致程序的IO阻塞，双方程序都无法正常进行。

实际上，上述的**服务器代码**还是有很大的改进空间的。因为Socket监听客户端（accept）、使用IO流发送和接收数据是阻塞的，只有有Socket连接、IO流可写和可读数据时，程序才会执行，否则一直是阻塞状态。为了能让服务器具有同时响应多个Socket连接的能力，一般服务器端需要进行多线程处理。例如改进为如下代码：

（1）HandleTcp线程类：接收客户端Socket，处理具体的IO。

|  |
| --- |
| **package** com.demo.tcp;  **import** java.io.\*; **import** java.net.Socket;  **public class** HandleTcp **extends** Thread {  **private** Socket **client**;   *// 接收客户端socket* **public** HandleTcp(Socket client) {  **this**.**client** = client;  }   @Override  **public void** run() {  **try** {  *// 获得数据* InputStream is = **client**.getInputStream();  BufferedReader br = **new** BufferedReader(**new** InputStreamReader(is));  String data = **null**;  **while** ((data = br.readLine()) != **null**) {  System.***out***.println(data);  }  **client**.shutdownInput(); *// shutdownInput()，重要！！！  // 发送反馈数据* OutputStream os = **client**.getOutputStream();  os.write(**"data received"**.getBytes());  } **catch** (Exception ex) {   } **finally** {  **try** {  **client**.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  } } |

（2）TcpServer线程类：不断接收客户端Socket，然后再将Socket传递给HandleTcp具体处理。

|  |
| --- |
| **package** com.demo.tcp;  **import** java.io.IOException; **import** java.net.ServerSocket; **import** java.net.Socket;  **public class** TcpServer **extends** Thread {  **private** ServerSocket **serverSocket**;   *// 接收serverSocket* **public** TcpServer(ServerSocket serverSocket) {  **this**.**serverSocket** = serverSocket;  }   @Override  **public void** run() {  **try** {  **while** (**true**) {  Socket client = **serverSocket**.accept();  **new** HandleTcp(client).start();  }  } **catch** (IOException e) {  e.printStackTrace();  }  } } |

（3）主类：创建TcpServer线程执行，同时也可运行其他代码。注意，监听端口new ServerSocket(10000)不能写在上述的循环中，因为只能new一次，否则会导致端口占用问题。

|  |
| --- |
| **package** com.demo.tcp;  **import** java.io.\*; **import** java.net.ServerSocket;  **public class** Server {  **public static void** main(String[] args) **throws** IOException {  ServerSocket serverSocket = **new** ServerSocket(10000);  **new** TcpServer(serverSocket).start();  System.***out***.println(**"其他程序正常运行"**);  } } |

上述多线程无需考虑线程安全问题，因为每个客户端Socket不是共享的，无共享变量。