# Java网络

网络这块全部转自

<http://ifeve.com/java-networking/>

## Java网络教程-基础

Java提供了非常易用的网络API，调用这些API我们可以很方便的通过建立TCP/IP或UDP套接字，在网络之间进行相互通信，其中TCP要比UDP更加常用，但在本教程中我们对这两种方式都有说明。

在网站上还有其他三个与Java网络相关的教程，如下：

1.[Java IO 教程](http://tutorials.jenkov.com/java-io/index.html)

2.[Java NIO 教程](http://ifeve.com/java-nio-all/" \o "Java NIO 系列教程)

3.[Java服务器多线程教程](http://tutorials.jenkov.com/java-multithreaded-servers/index.html) （参与翻译可以联系我们）

尽管Java网络API允许我们通过套接字（Socket）打开或关闭网络连接，但所有的网络通信均是基于Java IO类 [InputStream](http://tutorials.jenkov.com/java-io/inputstream.html)和[OutputStream](http://tutorials.jenkov.com/java-io/outputstream.html)实现的。

此外，我们还可以使用Java NIO API中相关的网络类，用法与Java网络API基本类似，Java NIO API可以以非阻塞模式工作，在某些特定的场景中使用非阻塞模式可以获得较大的性能提升。

**Java TCP网络基础**

通常情况下，客户端打开一个连接到服务器端的TCP/IP连接，然后客户端开始与服务器之间通信，当通信结束后客户端关闭连接，过程如下图所示：



​客户端通过一个已打开的连接可以发送不止一个请求。事实上在服务器处于接收状态下，客户端可以发送尽可能多的数据，服务器也可以主动关闭连接。

**Java中Socket类和ServerSocket类**

当客户端想要打开一个连接到服务器的TCP/IP连接时，就要使用到[Java Socket](http://tutorials.jenkov.com/java-networking/sockets.html)类。socket类只需要被告知连接的IP地址和TCP端口，其余的都有Java实现。

假如我们想要打开一个监听服务，来监听客户端连接某些指定TCP端口的连接，那就需要使用[Java ServerSocket](http://tutorials.jenkov.com/java-networking/server-sockets.html)类。当客户端通过Socket连接服务器端的ServerSocket监听时，服务器端会指定这个连接的一个Socket，此时客户端与服务器端间的通信就变成Socket与Socket之间的通信。

关于Socket类和ServerSocket类会在后面的文章中有详细的介绍。

**Java UDP网络基础**

UDP的工作方式与TCP相比略有不同。使用UDP通信时，在客户端与服务器之间并没有建立连接的概念，客户端发送到服务器的数据，服务器可能（也可能并没有）收到这些数据，而且客户端也并不知道这些数据是否被服务器成功接收。当服务器向客户端发送数据时也是如此。

正因为是不可靠的数据传输，UDP相比与TCP来说少了很多的协议开销。

## Java网络教程之Socket

当我们想要在Java中使用TCP/IP通过网络连接到服务器时，就需要创建java.net.Socket对象并连接到服务器。假如希望使用Java NIO，也可以创建Java NIO中的[SocketChannel](http://tutorials.jenkov.com/java-nio/socketchannel.html)对象。

**创建Socket**

下面的示例代码是连接到IP地址为78.64.84.171服务器上的80端口，这台服务器就是我们的Web服务器（www.jenkov.com），而80端口就是Web服务端口。

Socket socket = new Socket("78.46.84.171", 80);

我们也可以像如下示例中使用域名代替IP地址：

Socket socket = new Socket("jenkov.com", 80);

**Socket发送数据**

要通过Socket发送数据，我们需要获取Socket的输出流（[OutputStream](http://tutorials.jenkov.com/java-io/outputstream.html)），示例代码如下：

Socket socket = new Socket("jenkov.com", 80);

OutputStream out = socket.getOutputStream();

out.write("some data".getBytes());

out.flush();

out.close();

socket.close();

代码非常简单，但是想要通过网络将数据发送到服务器端，一定不要忘记调用flush()方法。操作系统底层的TCP/IP实现会先将数据放入一个更大的数据缓存块中，而缓存块的大小是与TCP/IP的数据包大小相适应的。（译者注：调用flush()方法只是将数据写入操作系统缓存中，并不保证数据会立即发送）

**Socket读取数据**

从Socket中读取数据，我们就需要获取Socket的输入流（[InputStream](http://tutorials.jenkov.com/java-io/inputstream.html)），代码如下：

Socket socket = new Socket("jenkov.com", 80);

InputStream in = socket.getInputStream();

int data = in.read();

//... read more data...

in.close();

socket.close();

代码也并不复杂，但需要注意的是，从Socket的输入流中读取数据并不能读取文件那样，一直调用read()方法直到返回-1为止，因为对Socket而言，只有当服务端关闭连接时，Socket的输入流才会返回-1，而是事实上服务器并不会不停地关闭连接。假设我们想要通过一个连接发送多个请求，那么在这种情况下关闭连接就显得非常愚蠢。

因此，从Socket的输入流中读取数据时我们必须要知道需要读取的字节数，这可以通过让服务器在数据中告知发送了多少字节来实现，也可以采用在数据末尾设置特殊字符标记的方式连实现。

**关闭Socket**

当使用完Socket后我们必须将Socket关闭，断开与服务器之间的连接。关闭Socket只需要调用Socket.close()方法即可，代码如下：

Socket socket = new Socket("jenkov.com", 80);

**socket.close();**

***原创文章，转载请注明：****转载自[并发编程网 – ifeve.com](http://ifeve.com/)****本文链接地址:****[Java网络教程之Socket](http://ifeve.com/java-socket/)*

## Java 网络教程: ServerSocket

用java.net.ServerSocket实现java服务通过TCP/IP监听客户端连接，你也可以用Java NIO 来代替java网络标准API，这时候需要用到 ServerSocketChannel。

**创建一个 ServerSocket连接**

以下是一个创建ServerSocket类来监听9000端口的一个简单的代码

ServerSocket serverSocket = new ServerSocket(9000);

**监听请求的连接**

要获取请求的连接需要用ServerSocket.accept()方法。该方法返回一个Socket类，该类具有普通java Socket类的所有特性。代码如下：

ServerSocket serverSocket = new ServerSocket(9000); boolean isStopped = false;while(!isStopped){   Socket clientSocket = serverSocket.accept();    //do something with clientSocket}

对每个调用了accept()方法的类都只获得一个请求的连接。

另外，请求的连接也只能在线程运行的server中调用了accept()方法之后才能够接受请求。线程运行在server中其它所有的方法上的时候都不能接受客户端的连接请求。所以”接受”请求的线程通常都会把Socket的请求连接放入一个工作线程池中，然后再和客户端连接。更多关于多线程服务端设计的文档请参考 java多线程服务

**关闭客户端Socket**

客户端请求执行完毕，并且不会再有该客户端的其它请求发送过来的时候，就需要关闭Socket连接，这和关闭一个普通的客户端Socket连接一样。如下代码来执行关闭：

socket.close();

**关闭服务端Sockets**

要关闭服务的时候需要关掉 ServerSocket连接。通过执行如下代码：

serverSocket.close();

***原创文章，转载请注明：****转载自[并发编程网 – ifeve.com](http://ifeve.com/)****本文链接地址:****[Java 网络教程: ServerSocket](http://ifeve.com/java-network-serversocket-2/)*

## Java Networking: UDP DatagramSocket

**DatagramSocket**'s are Java's mechanism for network communication via UDP instead of TCP. UDP is still layered ontop of IP. You can use Java's DatagramSocket both for sending and receiving UPD datagrams.

### UDP vs. TCP

UDP works a bit differently from TCP. When you send data via TCP you first create a connection. Once the TCP connection is established TCP guarantess that your data arrives at the other end, or it will tell you that an error occurred.

With UDP you just send packets of data (datagrams) to some IP address on the network. You have no guarantee that the data will arrive. You also have no guarantee about the order which UDP packets arrive in at the receiver. This means that UDP has less protocol overhead (no stream integrity checking) than TCP.

UDP is appropriate for data transfers where it doesn't matter if a packet is lost in transition. For instance, imagine a transfer of a live TV-signal over the internet. You want the signal to arrive at the clients as close to live as possible. Therefore, if a frame or two are lost, you don't really care. You don't want the live broadcast to be delayed just to make sure all frames are shown at the client. You'd rather skip the missed frames, and move directly to the newest frames at all times.

This could also be the case with a surveillance camera broadcasting over the internet. Who cares what happened in the past, when you are trying to monitor the present. You don't want to end up being 30 seconds behind reality, just because you want to show all frames to the person monitoring the camera. It is a bit different with the storage of the camera recordings. You may not want to lose a single frame when recording the images from the camera to disk. You may rather want a little delay, than not have those frames to go back and examine, if something important occurs.

### Sending Data via a DatagramSocket

To send data via Java's DatagramSocket you must first create a DatagramPacket. Here is how that is done:

byte[] buffer = new byte[65508];

InetAddress address = InetAddress.getByName("jenkov.com");

DatagramPacket packet = new DatagramPacket(

buffer, buffer.length, address, 9000);

The byte buffer (the byte array) is the data that is to be sent in the UDP datagram. The length of the above buffer, 65508 bytes, is the maximum amount of data you can send in a single UDP packet.

The length given to the DatagramPacket constructor is the length of the data in the buffer to send. All data in the buffer after that amount of data is ignored.

The InetAddress instance contains the address of the node (e.g. server) to send the UDP packet to. TheInetAddress class represents an IP address (Internet Address). The getByName() method returns anInetAddress instance with the IP address matching the given host name.

The port parameter is the UDP port the server to receiver the data is listeing on. UDP and TCP ports are not the same. A computer can have different processes listening on e.g. port 80 in UDP and in TCP at the same time.

To send the DatagramPacket you must create a DatagramSocket targeted at sending data. Here is how that is done:

DatagramSocket datagramSocket = new DatagramSocket();

To send data you call the send() method, like this:

datagramSocket.send(packet);

Here is a full example:

DatagramSocket datagramSocket = new DatagramSocket();

byte[] buffer = "0123456789".getBytes();

InetAddress receiverAddress = InetAddress.getLocalHost();

DatagramPacket packet = new DatagramPacket(

buffer, buffer.length, receiverAddress, 80);

datagramSocket.send(packet);

### Receiving Data via a DatagramSocket

Receiving data via a DatagramSocket is done by first creating a DatagramPacket and then receiving data into it via the DatagramSocket's receive() method. Here is an example:

DatagramSocket datagramSocket = new DatagramSocket(80);

byte[] buffer = new byte[10];

DatagramPacket packet = new DatagramPacket(buffer, buffer.length);

datagramSocket.receive(packet);

Notice how the DatagramSocket is instantiated with the parameter value 80 passed to its constructor. This parameter is the UDP port the DatagramSocket is to receive UDP packets on. As mentioned earlier, TCP and UDP ports are not the same, and thus do not overlap. You can have two different processes listening on both TCP and UDP port 80, without any conflict.

Second, a byte buffer and a DatagramPacket is created. Notice how the DatagramPacket has no information about the node to send data to, as it does when creating a DatagramPacket for sending data. This is because we are going to use the DatagramPacket for receiving data, not sending it. Thus no destination address is needed.

Finally the DatagramSocket's receive() method is called. This method blocks until a DatagramPacket is received.

The data received is located in the DatagramPacket's byte buffer. This buffer can be obtained by calling:

byte[] buffer = packet.getData();

How much data was received in the buffer is up to you to find out. The protocol you are using should specify either how much data is sent per UDP packet, or specify an end-of-data marker you can look for instead.

A real server program would probably call the receive() method in a loop, and pass all receivedDatagramPacket's to a pool of worker threads, just like a TCP server does with incoming connections (see**[Java Multithreaded Servers](http://tutorials.jenkov.com/java-multithreaded-servers/index.html)** for more details).

## Java网络教程:URL + URLConnection

### 目录

* HTTP GET和POST
* 从URLs到本地文件

在java.net包中包含两个有趣的类：URL类和URLConnection类。这两个类可以用来创建客户端到web服务器（HTTP服务器）的连接。下面是一个简单的代码例子：

[查看源代码](http://ifeve.com/java-netword-url-urlconnection/" \l "viewSource" \o "查看源代码)[打印](http://ifeve.com/java-netword-url-urlconnection/" \l "printSource" \o "打印)[帮助](http://ifeve.com/java-netword-url-urlconnection/" \l "about" \o "帮助)

|  |  |
| --- | --- |
| 1 | URL url = **new** URL("[http://jenkov.com](http://jenkov.com/)"); |

|  |  |
| --- | --- |
| 2 | URLConnection urlConnection = url.openConnection(); |

|  |  |
| --- | --- |
| 3 | InputStream input = urlConnection.getInputStream(); |

|  |  |
| --- | --- |
| 4 | **int** data = input.read(); |

|  |  |
| --- | --- |
| 5 | **while**(data != -1){ |

|  |  |
| --- | --- |
| 6 | System.out.print((**char**) data); |

|  |  |
| --- | --- |
| 7 | data = input.read(); |

|  |  |
| --- | --- |
| 8 | } |

|  |  |
| --- | --- |
| 9 | input.close(); |

### HTTP GET和POST

默认情况下URLConnection发送一个HTTP GET请求到web服务器。如果你想发送一个HTTP POST请求，要调用URLConnection.setDoOutput(true)方法，如下：

|  |  |
| --- | --- |
| 1 | URL url = **new** URL("[http://jenkov.com](http://jenkov.com/)"); |

|  |  |
| --- | --- |
| 2 | URLConnection urlConnection = url.openConnection(); |

|  |  |
| --- | --- |
| 3 | urlConnection.setDoOutput(**true**); |

一旦你调用了setDoOutput(true)，你就可以打开URLConnection的OutputStream，如下：

|  |  |
| --- | --- |
| 1 | OutputStream output = urlConnection.getOutputStream(); |

你可以使用这个OutputStream向相应的HTTP请求中写任何数据，但你要记得将其转换成URL编码（关于URL编码的解释，自行Google）（译者注：具体名字是：application/x-www-form-urlencoded MIME 格式编码）。  
当你写完数据的时候要记得关闭OutputStream。

### 从URLs到本地文件

URL也被叫做统一资源定位符。如果你的代码不关心文件是来自网络还是来自本地文件系统，URL类是另外一种打开文件的方式。  
下面是一个如何使用URL类打开一个本地文件系统文件的例子：

|  |  |
| --- | --- |
| 1 | URL url = **new** URL("file:/c:/data/test.txt"); |

|  |  |
| --- | --- |
| 2 | URLConnection urlConnection = url.openConnection(); |

|  |  |
| --- | --- |
| 3 | InputStream input = urlConnection.getInputStream(); |

|  |  |
| --- | --- |
| 4 | **int** data = input.read(); |

|  |  |
| --- | --- |
| 5 | **while**(data != -1){ |

|  |  |
| --- | --- |
| 6 | System.out.print((**char**) data); |

|  |  |
| --- | --- |
| 7 | data = input.read(); |

|  |  |
| --- | --- |
| 8 | } |

|  |  |
| --- | --- |
| 9 | input.close(); |

注意：这和通过HTTP访问一个web服务器上的文件的唯一不同处就是URL：”file:/c:/data/test.txt”。

***原创文章，转载请注明：****转载自[并发编程网 – ifeve.com](http://ifeve.com/)****本文链接地址:****[Java网络教程:URL + URLConnection](http://ifeve.com/java-netword-url-urlconnection/)*

## Java网络教程: JarURLConnection

JarURLConnection是一种连接到**[Java](http://lib.csdn.net/base/javaee" \o "Java EE知识库" \t "http://blog.csdn.net/hhzxj2008/article/details/_blank)**文档文件或Jar文件中的一个入口连接定位

String urlString = "http://butterfly.jenkov.com/"

+ "container/download/"

+ "jenkov-butterfly-container-2.9.9-beta.jar";

URL jarUrl = new URL(urlString);

JarURLConnection connection = new JarURLConnection(jarUrl);

Manifest manifest = connection.getManifest();

JarFile jarFile = connection.getJarFile();

//do something with Jar file....

## Java网络教程: InetAddress

### java.net.InetAddress类是Java对IP地址（包括IPv4和IPv6）的高层表示。大多数其他网络类都要用到这个类，包括Socket，ServerSocket，URL，DatagramSocket，DatagramPacket等。一般地讲，它包括一个主机名和一个IP地址。

### 创建InetAddress 实例

InetAddress 没有公开的构造函数，所以需要通过它提供的静态方法来获取

static InetAddress[] getAllByName(String host)  
static InetAddress getByAddress(byte[] addr)  
static InetAddress getByAddress(String host,byte[] addr)  
static InetAddress getByName(String host)  
static InetAddress getLocalHost()

## Java网络教程:Protocol Design

如果设计一个客户端到服务器的系统，那么同时也需要设计客户端和服务器之间的通信协议。当然，有时候协议已经为你决定好了，比如HTTP、XML\_RPC（http response 的 body 使用xml）、或者SOAP(也是http response 的 body 使用xml)。设计客户端到服务端协议的时候，一旦协议决定开启一会儿，来看一些你必须考虑的地方：

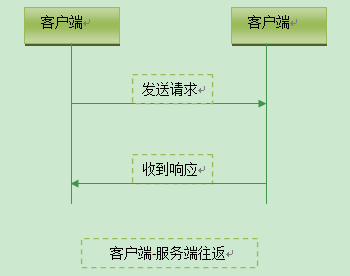
1. 客户端到服务端的往返通讯

2.区分请求结束和响应结束。

3.防火墙穿透

### 客户端-服务端往返

当客户端和服务端通信，执行操作时，他们在交换信息。比如，客户端执行一个服务请求，服务端尝试完成这个请求，发回响应告诉客户端结果。这种客户端和服务端的信息交换就叫做往返。示意图如下：



当一个计算机（客户端或者服务端）在网络中发送数据到另一个计算机时，从数据发送到另一端接收数据完会花费一定时间。这就是数据在网络间的传送的时间花费。这个时间叫做延迟。

协议中含有越多的往返，协议变得越慢，延迟特别高。HTTP协议只包含一个单独的响应来执行服务。换句话说就是一个单独的往返。另一方面，在一封邮件发送前，SMTP协议包含了几个客户端和服务端的往返。

在协议中有多个往返的原因是：有大量的数据从客户端发送到服务端。这种情况下你有2个选择：

1.在分开往返中发送头信息；

2.将消息分成更小的数据块。

如果服务端能完成头信息的一些初始验证 ，那么分开发送头信息是很明智的。如果头信息是空白的，发送大量数据本身就是浪费资源。

在传输大量数据时，如果网络连接失败了，得从头开始重新发送数据。数据分割发送时，只需要在网络连接失败处重新发送数据块。已经发送成功的数据块不需要重新发送。

### 区分请求结束和响应结束

如果协议容许在同一个连接中发送多个请求，需要一个让服务端知道当前请求何时结束、下一个请求何时开始。客户端也需要知道一个响应何时结束了，下一个响应何时开始。

对于请求有2个方法区分结束：

1.在请求的开始处发送请求的字长

2.在请求数据的最后发送一个结束标记。

HTTP用第一个机制。在请求头中 发送了“Content-Length”。请求头会告诉服务端在头文件后有多少字节是属于请求的。

这个模型的优势在于没有请求结束标志的开销。为了避免数据看上去像请求结束标志，也不需要对数据体进行编码。

第一个方法的劣势：在数据传输前，发送者必须知道多少字节数将被传输。如果数据时动态生成的，在发送前，首先你得缓存所有的数据，这样才能计算出数据的字节数。

运用请求结束标志时，不需要知道发送了多少字节数。只需要知道请求结束标志在数据的末尾。当然，必须确认已发送的数据中不包含会导致请求结束标志错误的数据。可以这样做：

可以说请求结束标志是字节值255。当然数据可能包含值255。因此，对数据中包含值255的每一个字节添加一个额外的字节，还有值255。结束请求标志被从字节值255到255之后的值为0。如下编码：

255 in data –>255， 255

end-of-request –> 255, 0

这种255，0的序列永远不会出现在数据中，因为你把所有的值255变成了255,255。同时，255,255,0也不会被错认为255,0。255,255被理解成在一起的，0是单独的。

### 防火墙穿透

比起HTTP协议，大多数防火墙会拦截所有的其他通信。因此把协议放在HTTP的上层是个好方法，像XML-RPC,SOAP和REST也可以这样做。

协议置于HTTP的上层，在客户端和服务端的HTTP请求和响应中可以来回发送数据。记住，HTTP请求和响应不止包含text或者HTML。也可以在里面发送二进制数据。

将请求放置在HTTP协议上，唯一有点奇怪的是：HTTP请求必须包含一个“主机”头字段。如果你在HTTP协议上设计P2P协议，同样的人最可能不会运行多个“主机”。在这种情况下需要头字段是不必要的开销（但是个小开销）。