

## 《不再犹豫 Java 网络编程》

1 Internet 地址概述 .....	2
2 创建 InetAddress 对象(IP)的四个静态方法 .....	3
3 不能直接通过 IP 访问网站 .....	7
4 DNS 缓存 .....	8
5 用 InetAddress 类的 getHostName 方法获得域名 .....	11
6 使用 getCanonicalHostName 方法获得主机名 .....	13
7 用 getHostAddress 方法获得 IP .....	15
8 用 getAddress 方法获得 IP 地址 .....	16
9 使用 is*方法判断地址类型 .....	17
10 Inet4Address 类和 Inet6Address 类 .....	20
11 用 NetworkInterface 类获得网络接口信息 .....	21
12 用 isReachable 方法探测主机是否可达 .....	25
13 用 Socket 类接收和发送数据 .....	25
14 多种多样的建立网络连接的方式 .....	32
15 为什么要使用 SocketAddress 来管理网络地址 .....	37
16 客户端套接字 (Socket) 的超时 .....	39
17 Socket 类的 getter 和 setter 方法 A .....	41
18 Socket 类的 getter 和 setter 方法 B .....	43
19 套接字异常 .....	48
20 HTTP 协议简介 .....	50
21 HTTP 消息的格式 .....	51
22 实现 HTTP 模拟器 .....	53
23 HTTP 消息头字段 .....	59
24 实现 HTTP 断点续传下载工具 .....	62
25 创建 ServerSocket 对象 .....	71
26 在服务端接收和发送数据 .....	77
27 关闭服务端连接 .....	78
28 获取 ServerSocket 信息的方法及 FTP 原理 .....	80
29 服务端 Socket 的选项 .....	83
30 定制 Accept 方法 .....	88
31 非阻塞 I/O 简介 .....	89
32 一个非阻塞 I/O 的例子 .....	91
33 非阻塞 I/O 的缓冲区 .....	92
34 读写缓冲区: 用 get 和 put 方法顺序读写单个数据 .....	95

原著出处: <http://androidguy.blog.51cto.com/all/974126>

请支持原版, 下载后 24 小时内删除

# 1 Internet 地址概述

所有连入 Internet 的终端设备(包括计算机、PDA、打印机以及其他的电子设备)都有一个唯一的索引,这个索引被称为 IP 地址。现在 Internet 上的 IP 地址大多由四个字节组成,这种 IP 地址叫做 IPv4。除了这种由四个字节组成的 IP,在 Internet 上还存在一种 IP,这种 IP 由 16 个字节组成,叫做 IPv6。IPv4 和 IPv6 后面的数字是 Internet 协议(Internet Protocol, IP)的版本号。

IPv4 地址的一般表现形式为: X.X.X.X。其中 X 为 0 到 255 的整数。这四个整数用“.”隔开。从理论上说,IPv4 地址可以表示 2 的 32 次幂,也就是 4,294,967,296 个 IP 地址,但由于要排除一些具有特殊意义的 IP(如 0.0.0.0、127.0.0.1、224.0.0.1、255.255.255.255 等),因此,IPv4 地址可自由分配的 IP 数量要小于它所能表示的 IP 地址数量。

为了便于管理,人为地将 IPv4 划分为 A 类、B 类和 C 类 IP 地址。

- A 类 IP 地址

范围: 0.0.0.0 — 127.255.255.255, 标准的子网掩码是 255.0.0.0。

- B 类 IP 地址

范围: 128.0.0.0 — 191.255.255.255, 标准的子网掩码是 255.255.0.0。

- C 类 IP 地址

范围: 192.0.0.0 — 223.255.255.255, 标准的子网掩码是 255.255.255.0。

从上面的描述可看出,第一个字节在 0 和 127 之间的是 A 类 IP 地址,在 128 和 191 之间的是 B 类 IP 地址,而在 192 和 223 之间的是 C 类 IP 地址。如果两个 IP 地址分别和它们的子网掩码进行按位与后得到的值是一样的,就说明这两个 IP 在同一个网段。下面是两个 C 类 IP 地址 IP1、IP2 和它们的子网掩码。

IP1: 192.168.18.10    子网掩码: 255.255.255.0

IP2: 192.168.18.20    子网掩码: 255.255.255.0

这两个 IP 和它们的子网掩码按位与后,得到的值都是 192.168.18.0。因此,IP1 和 IP2 在同一个网段。当用户使用 Modem 或 ADSL Modem 上网后,临时分配给本机的 IP 一般都是 C 类地址,也就是说,第一个字节都会在 192 和 223 之间。

上面给出的 IP 地址和子网掩码只是标准的形式。用户也可以根据自己的需要使用其他的 IP 和子网掩码,如 IP 地址设为 10.0.0.1,子网掩码设为 255.255.255.128。但为了便于分类和管理,在局域网中设置 IP 地址时,建议按着标准的分类来设置。

IPv6 地址由 16 个字节组成,共分为 8 段。每一段由 16 个字节组成,并用 4 个十六进制数表示,段与段之间用“:”隔开。如 A34E:DD3D:1234:4400:A123:B231:A111:DDAA 是一个标准的 IPv6 地址。IPv6 在两种情况下可以简写:

1. 以 0 开头的段可省略 0。如 A34E:003D:0004:4400:A123:B231:A111:DDAA 可简写为 A34E:3D:4:4400:A123:B231:A111:DDAA。
2. 连续出现 0 的多个段可使用“::”来代替多个为 0 的段。如 A34E:0000:0000:0000:A123:B231:0:DDAA 可简写为 A34E::A123:B231:0:DDAA。在使用这种简写方式时,“::”只能出现一次,如果出现多次,IPv6 地址将会产生歧义。

在 IPv4 和 IPv6 混合的网络中,IPv6 地址的后四个字节可以被写成 IPv4 的地址格式。如 A34E::A123:B231:A111:DDAA 可以写成 A34E::A123:B231:161.17.221.170。当访问网络资源的计算机使用的是 IPv4 的地址时,系统会自动使用 IPv6 的后四个字节作为 IPv4 的地址。

无论是 IPv4 地址,还是 IPv6 地址,都是很难记忆的。因此,为了使这些地址便于记忆,Internet 的设计师们发明了 DNS(Domain Name System, 域名系统)。DNS 将 IP 地址和域名(一个容易记忆的字符串,如 microsoft)联系在一起,当计算机通过域名访问 Internet 资源时,系统首先通过 DNS 得到域名对

应的 IP 地址，再通过 IP 地址访问 Internet 资源。在这个过程中，IP 地址对用户是完全透明的。如果一个域名对应了多个 IP 地址，DNS 从这些 IP 地址中随机选取一个返回。

域名可以分为不同的层次，如常见的有顶层域名、顶级域名、二级域名和三级域名。

- 顶层域名

顶层域名可分为类型顶层域名和地域顶层域名。如 `www.microsoft.com`、`www.w3c.org` 中的 `com` 和 `org` 就是类型顶层域名，它们分别代表商业(`com`)和非盈利组织(`org`)。而 `www.dearbook.com.cn` 中的 `cn` 就是地域顶层域名，它表示了中国(`cn`)。主要的类型顶层域名有 `com`(商业)、`edu`(教育)、`gov`(政府)、`int`(国际组织)、`mil`(美国军方)、`net`(网络部门)、`org`(非盈利组织)。大多数国家都有自己的地域顶层域名，如中国(`cn`)、美国(`us`)、英国(`uk`)等。

- 顶级域名

如 `www.microsoft.com` 中的 `microsoft.com` 就是一个顶级域名。在 Email 地址的“@”后面跟的都是顶级域名，如 `abc@126.com`、`mymail@sina.com` 等。

- 二级域名

如 `blog.csdn.net` 就是顶级域名 `csdn.net` 的二级域名。有很多人认为 `www.csdn.net` 是顶级域名，其实这是一种误解。实际上 `www.csdn.net` 是顶级域名 `csdn.net` 的二级域名。`www.csdn.net` 和 `blog.csdn.net` 在本质上是一样的，只是我们已经习惯了使用 `www` 表示一个使用 HTTP 或 HTTPS 协议的网址，因此，给人的误解就是 `www.csdn.net` 是一个顶级域名。

- 三级域名

如 `abc.photo.163.com` 就是二级域名 `photo.163.com` 的三级域名。有很多 blog 或电子相册之类的网站都为每个用户分配一个三级域名。

## 2 创建 InetAddress 对象(IP)的四个静态方法

`InetAddress` 类是 Java 中用于描述 IP 地址的类。它在 `java.net` 包中。在 Java 中分别用 `Inet4Address` 和 `Inet6Address` 类来描述 IPv4 和 IPv6 的地址。这两个类都是 `InetAddress` 的子类。由于 `InetAddress` 没有 `public` 的构造方法，因此，要想创建 `InetAddress` 对象，必须得依靠它的四个静态方法。`InetAddress` 可以通过 `getLocalHost` 方法得到本机的 `InetAddress` 对象，也可以通过 `getByName`、`getAllByName` 和 `getByAddress` 得到远程主机的 `InetAddress` 对象。

### 一、getLocalHost 方法

使用 `getLocalHost` 可以得到描述本机 IP 的 `InetAddress` 对象。这个方法的定义如下：

```
public static InetAddress getLocalHost() throws UnknownHostException
```

这个方法抛出了一个 `UnknownHostException` 异常，因此，必须在调用这个方法的程序中捕捉或抛出这个异常。下面的代码演示了如何使用 `getLocalHost` 来得到本机的 IP 和计算机名。

```
package inet;

import java.net.*;

public class MyInetAddress1
{
    public static void main(String[] args) throws Exception
    {
```

```

    InetAddress localAddress = InetAddress.getLocalHost();
    System.out.println(localAddress);
}
}

```

运行结果：

ComputerName/192.168.18.10

在 `InetAddress` 类中覆盖了 `Object` 类的 `toString` 方法，实现如下：

```

public String toString()
{
    return ((hostName != null) ? hostName : "") + "/" + getHostAddress();
}

```

从上面的代码可以看出，`InetAddress` 方法中的 `toString` 方法返回了用“/”隔开的主机名和 IP 地址。因此，在上面的代码中直接通过 `localAddress` 对象来输出本机计算机名和 IP 地址（将对象参数传入 `println` 方法后，`println` 方法会调用对象参数的 `toString` 方法来输出结果）。

当本机绑定了多个 IP 时，`getLocalHost` 只返回第一个 IP。如果想返回本机全部的 IP，可以使用 `getAllByName` 方法。

## 二、getByName 方法

这个方法是 `InetAddress` 类最常用的方法。它可以通过指定域名从 DNS 中得到相应的 IP 地址。`getByName` 一个 `String` 类型参数，可以通过这个参数指定远程主机的域名，它的定义如下：

```

public static InetAddress getByName(String host) throws UnknownHostException

```

如果 `host` 所指的域名对应多个 IP，`getByName` 返回第一个 IP。如果本机名已知，可以使用 `getByName` 方法来代替 `getLocalHost`。当 `host` 的值是 `localhost` 时，返回的 IP 一般是 127.0.0.1。如果 `host` 是不存在的域名，`getByName` 将抛出 `UnknownHostException` 异常，如果 `host` 是 IP 地址，无论这个 IP 地址是否存在，`getByName` 方法都会返回这个 IP 地址（因此 `getByName` 并不验证 IP 地址的正确性）。下面代码演示了如何使用 `getByName` 方法：

```

package inet;

import java.net.*;

public class MyInetAddress2
{
    public static void main(String[] args) throws Exception
    {
        if (args.length == 0)
            return;
        String host = args[0];
        InetAddress address = InetAddress.getByName(host);
        System.out.println(address);
    }
}

```

- 测试 1：远程主机 `www.csdn.net`

执行如下命令：

```

java inet.MyInetAddress2 www.csdn.net

```

运行结果：

```
www.csdn.net/211.100.26.124
```

- **测试 2：本机名 ComputerName**

执行如下命令：

```
java inet.MyInetAddress2 ComputerName
```

运行结果：

```
ComputerName/192.168.18.10
```

- **测试 3：代表本机的 localhost**

执行如下命令：

```
java inet.MyInetAddress2 localhost
```

运行结果：

```
localhost/127.0.0.1
```

对于本机来说，除了可以使用本机名或 localhost 外，还可以在 hosts 文件中对本机做“IP/域名”映射（在 Windows 操作系统下）。这个文件在 C:\WINDOWS\system32\drivers\etc 中。打开这两个文件中，在最后加一行如下所示的字符串：

```
192.168.18.100 www.mysite.com
```

- **测试 4：本机域名 www.mysite.com**

执行如下命令：

```
java inet.MyInetAddress2 www.mysite.com
```

运行结果：

```
www.mysite.com/192.168.18.100
```

getByName 方法除了可以使用域名作为参数外，也可以直接使用 IP 地址作为参数。如果使用 IP 地址作为参数，输出 InetAddress 对象时域名为空（除非调用 getHostName 方法后，再输出 InetAddress 对象。getHostName 方法将在下面的内容介绍）。读者可以使用 129.42.58.212 作为 MyInetAddress2 的命令行参数（这是 www.ibm.com 的 IP），看看会得到什么结果。

### 三、getAllByName 方法

使用 getAllByName 方法可以从 DNS 上得到域名对应的所有的 IP。这个方法返回一个 InetAddress 类型的数组。这个方法的定义如下：

```
public static InetAddress[] getAllByName(String host) throws UnknownHostException
```

与 getByName 方法一样，当 host 不存在时，getAllByName 也会抛出 UnknownHostException 异常，getAllByName 也不会验证 IP 地址是否存在。下面的代码演示了 getAllByName 的用法：

```
package inet;

import java.net.*;

public class MyInetAddress3
{
    public static void main(String[] args) throws Exception
    {
        if (args.length == 0)
            return;
        String host = args[0];
        InetAddress addresses[] = InetAddress.getAllByName(host);
        for (InetAddress address : addresses)
```

```
        System.out.println(address);
    }
}
```

- **测试 1：远程主机 www.csdn.net**

执行如下命令：

```
java inet.MyInetAddress3 www.csdn.net
```

运行结果：

```
www.csdn.net/211.100.26.124
www.csdn.net/211.100.26.121
www.csdn.net/211.100.26.122
www.csdn.net/211.100.26.123
```

将上面的运行结果和例程 3-2 的测试 1 的运行结果进行比较，可以得出一个结论，`getByName` 方法返回的 IP 地址就是 `getAllByName` 方法返回的第一个 IP 地址。事实上，`getByName` 的确是这么实现的，`getByName` 的实现代码如下：

```
public static InetAddress getByName(String host) throws UnknownHostException
{
    return InetAddress.getAllByName(host)[0];
}
```

- **测试 2：使用 www.csdn.net 的一个 IP**

执行如下命令：

```
java inet.MyInetAddress3 211.100.26.122
```

运行结果：

```
/211.100.26.122
```

#### 四、getByAddress 方法

这个方法必须通过 IP 地址来创建 `InetAddress` 对象，而且 IP 地址必须是 `byte` 数组形式。`getByAddress` 方法有两个重载形式，定义如下：

```
public static InetAddress getByAddress(byte[] addr) throws UnknownHostException
public static InetAddress getByAddress(String host, byte[] addr) throws UnknownHostException
```

第一个重载形式只需要传递 `byte` 数组形式的 IP 地址，`getByAddress` 方法并不验证这个 IP 地址是否存在，只是简单地创建一个 `InetAddress` 对象。`addr` 数组的长度必须是 4（IPv4）或 16（IPv6），如果是其他长度的 `byte` 数组，`getByAddress` 将抛出一个 `UnknownHostException` 异常。第二个重载形式多了一个 `host`，这个 `host` 和 `getByName`、`getAllByName` 方法中的 `host` 的意义不同，`getByAddress` 方法并不使用 `host` 在 DNS 上查找 IP 地址，这个 `host` 只是一个用于表示 `addr` 的别名。下面的代码演示了 `getByAddress` 的两个重载形式的用法：

```
package inet;

import java.net.*;

public class MyInetAddress4
{
    public static void main(String[] args) throws Exception
    {
```

```
byte ip[] = new byte[] { (byte) 141, (byte) 146, 8, 66};  
InetAddress address1 = InetAddress.getByAddress(ip);  
InetAddress address2 = InetAddress.getByAddress("Oracle 官方网站", ip);  
System.out.println(address1);  
System.out.println(address2);  
}  
}
```

上面代码的运行结果如下：

```
/141.146.8.66  
Oracle 官方网站/141.146.8.66
```

从上面的运行结果可以看出，`getByAddress` 只是简单地将 `host` 参数作为域名放到“/”前面，因此，`host` 可以是任何字符串。

### 3 不能直接通过 IP 访问网站

在上文中通过 `getAllByName` 得到了 [www.csdn.net](http://www.csdn.net) 对应的四个 IP 地址。从理论上说，在 IE（或其他的 Web 浏览器，如 Firefox）的地址栏中输入这四个 IP 地址中的任何一个，都可能访问 [www.csdn.net](http://www.csdn.net)。如输入 <http://211.100.26.124>。但 IE 却返回了一个错误信息。在输入另外三个 IP 后，都会得到同样的错误信息。

这个错误并不是网页未找到错误（HTTP 状态号：404），而是拒绝访问错误（HTTP 状态号：403）。当在地址栏中再输入 <http://www.csdn.net>，仍然可以访问这个网站。从以上种种迹象表明这并不客户端的问题，而是服务端对此做了限制。

在 HTTP 协议（这个协议会在下一章详细讲解）的请求头有一个 `Host` 字段，一般通过 <http://www.csdn.net> 访问服务器时，`Host` 的值就是 [www.csdn.net](http://www.csdn.net)。如果是 <http://211.100.26.124>，那么 `Host` 的值就是 [211.100.26.124](http://211.100.26.124)。因此，我们可以推断，[www.csdn.net](http://www.csdn.net) 的服务器通过检测 `Host` 字段防止客户端直接使用 IP 进行访问。目前有很多网站，如 [www.sina.com.cn](http://www.sina.com.cn)、[www.126.com](http://www.126.com) 都是这样做的。有一些网站虽然未限制用 IP 地址来访问，但在使用 IP 地址访问网站时，却将 IP 地址又重定位到相应的域名上。如输入 <http://141.146.8.66> 会重定位到 <http://www.oracle.com/index.html> 上，输入 <http://129.42.60.212> 会重定位到 <http://www.ibm.com/us/> 上。

通过 `ping` 命令也可以得到一个域名的 IP 地址，如果域名绑定有之个 IP 地址，DNS 就随机返回一个 IP 地址。如在控制台输入下面的命令：

```
ping www.csdn.net
```

返回结果

```
Reply from 211.100.26.122: bytes=32 time=31ms TTL=48  
Reply from 211.100.26.122: bytes=32 time=35ms TTL=48
```

上面的返回结果中的 IP 地址就是《创建 `InetAddress` 对象的四个静态方法》一文中 `MyInetAddress3` 在测试 1 中所得到的第三个 IP 地址。



## 4 DNS 缓存

在通过 DNS 查找域名的过程中，可能会经过多台中间 DNS 服务器才能找到指定的域名，因此，在 DNS 服务器上查找域名是非常昂贵的操作。在 Java 中为了缓解这个问题，提供了 DNS 缓存。当 `InetAddress` 类第一次使用某个域名（如 `www.csdn.net`）创建 `InetAddress` 对象后，JVM 就会将这个域名和它从 DNS 上获得的信息（如 IP 地址）都保存在 DNS 缓存中。当下一次 `InetAddress` 类再使用这个域名时，就直接从 DNS 缓存里获得所需的信息，而无需再访问 DNS 服务器。

DNS 缓存在默认时将永远保留曾经访问过的域名信息，但我们可以修改这个默认值。一般有两种方法可以修改这个默认值：

1. 在程序中通过 `java.security.Security.setProperty` 方法设置安全属性 `networkaddress.cache.ttl` 的值（单位：秒）。如下面的代码将缓存超时设为 10 秒：

```
java.security.Security.setProperty("networkaddress.cache.ttl", 10);
```

2. 设置 `java.security` 文件中的 `networkaddress.cache.negative.ttl` 属性。假设 JDK 的安装目录是 `C:\jdk1.6`，那么 `java.security` 文件位于 `c:\jdk1.6\jre\lib\security` 目录中。打开这个文件，找到 `networkaddress.cache.ttl` 属性，并将这个属性值设为相应的缓存超时（单位：秒）。

如果将 `networkaddress.cache.ttl` 属性值设为 -1，那么 DNS 缓存数据将永远不会释放。下面的代码演示了使用和不使用 DNS 缓存所产生效果：

```
package mynet;

import java.net.*;

public class MyDNS
{
    public static void main(String[] args) throws Exception
    {
        // args[0]: 本机名 args[1]: 缓冲时间
        if (args.length < 2)
            return;
        java.security.Security.setProperty("networkaddress.cache.ttl", args[1]);
        long time = System.currentTimeMillis();
        InetAddress addresses1[] = InetAddress.getAllByName(args[0]);
        System.out.println("addresses1: "
            + String.valueOf(System.currentTimeMillis() - time)
            + "毫秒");
        for (InetAddress address : addresses1)
            System.out.println(address);

        System.out.print("按任意键继续    ");
        System.in.read();
        time = System.currentTimeMillis();
        InetAddress addresses2[] = InetAddress.getAllByName(args[0]);
        System.out.println("addresses2: "
            + String.valueOf(System.currentTimeMillis() - time)
```



```

        + "毫秒");
    for (InetAddress address : addresses2)
        System.out.println(address);
    }
}

```

在上面的代码中设置了 DNS 缓存超时(通过 args[1]参数),用户可以通过命令行参数将这个值传入 MyDNS 中。这个程序首先使用 getAllByName 建立一个 InetAddress 数组,然后通过 System.in.read 使程序暂停。当用户等待一段时间后,可以按任意键继续,并使用同一个域名(args[0])再建立一个 InetAddress 数组。如果用户等待的这段时间比 DNS 缓存超时小,那么无论情况如何变化,addresses2 和 addresses1 数组中的元素是一样的,并且创建 addresses2 数组所花费的时间一般为 0 毫秒(小于 1 毫秒后,Java 无法获得更精确的时间)。

#### 测试 1:

执行如下命令(将 DNS 缓存超时设为 5 秒):

```
java mynet.MyDNS www.126.com 5
```

运行结果 1 (在 5 秒之内按任意键):

```
addresses1:  344 毫秒
www.126.com/202.108.9.77
```

按任意键继续

```
addresses2: 0 毫秒
www.126.com/202.108.9.77
```

运行结果 2 (在 5 秒后按任意键):

```
addresses1:  344 毫秒
www.126.com/202.108.9.77
```

按任意键继续

```
addresses2: 484 毫秒
www.126.com/202.108.9.77
```

在上面的测试中可能出现两个运行结果。如果在出现“按任意键继续...”后,在 5 秒之内按任意键继续后,就会得到运行结果 1,从这个结果可以看出,addresses2 所用的时间为 0 毫秒,也就是说,addresses2 并未真正访问 DNS 服务器,而是直接从内存中的 DNS 缓存得到的数据。当在 5 秒后按任意键继续后,就会得到运行结果 2,这时,内存中的 DNS 缓存中的数据已经释放,所以 addresses2 还得再访问 DNS 服务器,因此,addresses2 的时间是 484 毫秒(addresses1 和 addresses2 后面的毫秒数可能在不同的环境下的值不一样,但一般情况下,运行结果 1 的 addresses2 的值为 0 或是一个接近 0 的数,如 5。运行结果 2 的 addresses2 的值一般会 and addresses1 的值很接近,或是一个远比 0 大的数,如 1200)。

#### 测试 2:

执行如下命令(ComputerName 为本机的计算机名, DNS 缓存超时设为永不过期[-1]):

```
java mynet.MyDNS ComputerName -1
```

运行结果(按任意键继续之前,将 192.168.18.20 删除):

```
addresses1:  31 毫秒
myuniverse/192.168.18.10
myuniverse/192.168.18.20
```

按任意键继续

addresses2: 0 毫秒

myuniverse/192.168.18.10

myuniverse/192.168.18.20

从上面的测试可以看出，将 DNS 缓存设为永不过期后，无论过多少时间，按任意键后，addresses2 任然得到了两个 IP 地址（192.168.18.10 和 192.168.18.20），而且 addresses2 的时间是 0 毫秒，但在这时 192.168.18.20 已经被删除。因此可以判断，addresses2 是从 DNS 缓存中得到的数据。如果运行如下的命令，并在 5 秒后按任意键继续后，addresses2 就会只剩下一个 IP 地址（192.168.18.10）。

```
java mynet.MyDNS ComputerName 5
```

如果域名在 DNS 服务器上不存在，那么客户端在进行一段时间的尝试后（平均为 5 秒），就会抛出一个 UnknownHostException 异常。为了让下一次访问这个域名时不再等待，DNS 缓存将这个错误信息也保存了起来。也就是说，只有第一次访问错误域名时才进行 5 秒左右的尝试，以后再访问这个域名时将直接抛出 UnknownHostException 异常，而无需再等待 5 秒钟，

访问域名失败的原因可能是这个域名真的不存在，也可能是因为 DNS 服务器或是其他的硬件或软件的临时故障，因此，一般不能将这个域名错误信息一直保留。在 Java 中可以通过 networkaddress.cache.negative.ttl 属性设置保留这些信息的时间。这个属性的默认值是 10 秒。它也可以通过 java.security.Security.setProperty 方法或 java.security 文件来设置。下面的代码演示了 networkaddress.cache.negative.ttl 属性的用法：

```
package mynet;

import java.net.*;

public class MyDNS1
{
    public static void main(String[] args) throws Exception
    {
        java.security.Security.setProperty("networkaddress.cache.negative.ttl",
            "5");
        long time = 0;
        try
        {
            time = System.currentTimeMillis();
            InetAddress.getByNames("www.ppp123.com");
        }
        catch (Exception e)
        {
            System.out.println("www.ppp123.com 不存在! address1: "
                + String.valueOf(System.currentTimeMillis() - time)
                + " 毫秒");
        }
        //Thread.sleep(6000); // 延迟 6 秒
    }
}
```

```

        time = System.currentTimeMillis();
        InetAddress.getByNames("www.ppp123.com");
    }
    catch (Exception e)
    {
        System.out.println("www.ppp123.com 不存在! address2: "
            + String.valueOf(System.currentTimeMillis() - time)
            + "毫秒");
    }
}
}

```

在上面的代码中将 `networkaddress.cache.negative.ttl` 属性值设为 5 秒。这个程序分别测试了 `address1` 和 `address2` 访问 `www.ppp123.com`（这是个不存在的域名，读者可以将其换成任何不存在的域名）后，用了多长时间抛出 `UnknownHostException` 异常。

运行结果：

```

www.ppp123.com 不存在! address1: 4688 毫秒
www.ppp123.com 不存在! address2: 0 毫秒

```

我们从上面的运行结果可以看出，`address2` 使用了 0 毫秒就抛出了异常，因此，可以断定 `address2` 是从 DNS 缓存里获得了域名 `www.ppp123.com` 不可访问的信息，所以就直接抛出了 `UnknownHostException` 异常。如果将上面代码中的延迟代码的注释去掉，那么可能得到如下的运行结果：

```

www.ppp123.com 不存在! address1: 4688 毫秒
www.ppp123.com 不存在! address1: 4420 毫秒

```

从上面的运行结果可以看出，在第 6 秒时，DNS 缓存中的数据已经被释放，因此，`address2` 仍需要访问 DNS 服务器才能知道 `www.ppp123.com` 是不可访问的域名。

**在使用 DNS 缓存时有两点需要注意：**

1. 可以根据实际情况来设置 `networkaddress.cache.ttl` 属性的值。一般将这个属性的值设为 -1。但如果访问的是动态映射的域名（如使用动态域名服务将域名映射成 ADSL 的动态 IP），就可能产生 IP 地址变化后，客户端得到的还是原来的 IP 地址的情况。
2. 在设置 `networkaddress.cache.negative.ttl` 属性值时最好不要将它设为 -1，否则如果一个域名因为暂时的故障而无法访问，那么程序再次访问这个域名时，即使这个域名恢复正常，程序也无法再访问这个域名了。除非重新运行程序。

## 5 用 InetAddress 类的 getHostName 方法获得域名

该方法可以得到远程主机的域名，也可以得到本机名。`getHostName` 方法的定义如下：

```

public String getHostName()

```

下面是三种创建 `InetAddress` 对象的方式，在这三种方式中，`getHostName` 返回的值是不同的。

### 1. 使用 getLocalHost 方法创建 InetAddress 对象

如果 `InetAddress` 对象是用 `getLocalHost` 方法创建的，`getHostName` 返回的是本机名。如下面的代码所示：

```

InetAddress address = InetAddress.getLocalHost();
System.out.println(address.getHostName()); // 输出本机名

```

## 2. 使用域名创建 InetAddress 对象

用域名作为 `getByName` 和 `getAllByName` 方法的参数调用这两个方法后，系统会自动记住这个域名。当调用 `getHostName` 方法时，就无需再访问 DNS 服务器，而是直接将这个域名返回。如下面的代码所示：

```
InetAddress address = InetAddress.getByName("www.oracle.com");
System.out.println(address.getHostName()); // 无需访问 DNS 服务器，直接返回域名
```

## 3. 使用 IP 地址创建 InetAddress 对象

使用 IP 地址创建 `InetAddress` 对象时（`getByName`、`getAllByName` 和 `getByAddress` 方法都可以通过 IP 地址创建 `InetAddress` 对象），并不需要访问 DNS 服务器。因此，通过 DNS 服务器查找域名的工作就由 `getHostName` 方法来完成。如果这个 IP 地址不存在或 DNS 服务器不允许进行 IP 地址和域名的映射，`getHostName` 方法就直接返回这个 IP 地址。如下面的代码所示：

```
InetAddress address = InetAddress.getByName("141.146.8.66");
System.out.println(address.getHostName()); // 需要访问 DNS 服务器才能得到域名
InetAddress address = InetAddress.getByName("1.2.3.4"); // IP 地址不存在
System.out.println(address.getHostName()); // 直接返回 IP 地址
```

从上面的三种情况可以看出，只有通过使用 IP 地址创建的 `InetAddress` 对象调用 `getHostName` 方法时才访问 DNS 服务器。在其他情况，`getHostName` 方法并不会访问 DNS 服务器，而是直接将域名或本机名返回。下面的代码演示了在不同情况下如何使用 `getHostName` 方法，并计算了各种情况所需的毫秒数。

```
package mynet;

import java.net.*;

public class DomainName
{
    public static void main(String[] args) throws Exception
    {
        long time = 0;
        // 得到本机名
        InetAddress address1 = InetAddress.getLocalHost();
        System.out.println("本机名: " + address1.getHostName());
        // 直接返回域名
        InetAddress address2 = InetAddress.getByName("www.oracle.com");
        time = System.currentTimeMillis();
        System.out.print("直接得到域名: " + address2.getHostName());
        System.out.println(" 所用时间: "
            + String.valueOf(System.currentTimeMillis() - time) + " 毫秒");
        // 通过 DNS 查找域名
        InetAddress address3 = InetAddress.getByName("141.146.8.66");
        System.out.println("address3: " + address3); // 域名为空
        time = System.currentTimeMillis();
        System.out.print("通过 DNS 查找域名: " + address3.getHostName());
        System.out.println(" 所用时间: " + String.valueOf(System.currentTimeMillis() - time)
            + " 毫秒");
        System.out.println("address3: " + address3); // 同时输出域名和 IP 地址
```

```
}  
}
```

运行结果:

本机名: ComputerName

直接得到域名: www.oracle.com 所用时间: 0 毫秒

address3: /141.146.8.66

通过 DNS 查找域名: bigip-otn-portal.oracle.com 所用时间: 92 毫秒

address3: bigip-otn-portal.oracle.com/141.146.8.66

从上面的运行结果可以看出, 第一个毫秒数是 0, 而第二个毫秒数是 92。这说时, 使用域名创建的 `InetAddress` 对象在使用 `getHostName` 方法时并不访问 DNS 服务器, 而使用 IP 地址创建的 `InetAddress` 对象在使用 `getHostName` 方法时需要访问 DNS 服务器。对于第三个毫秒数, 可能多次运行 `DomainName` 后会越来越小, 这是因为 DNS 服务器的缓存的缘故。但一般这个数都会比 0 大。也许有很多人会问, 第二行和第四行得到的域名怎么不一样, 其实 `www.oracle.com` 和 `bigip-otn-portal.oracle.com` 都是 oracle 的域名, 我们也可以通过 `http:// bigip-otn-portal.oracle.com` 来访问 oracle 的官方网站。至于这两个域名有什么区别, 将在下面的文章中讨论。

## 6 使用 `getCanonicalHostName` 方法获得主机名

`getCanonicalHostName` 方法和 `getHostName` 方法一样, 也是得到远程主机的域名。但它们有一个区别。`getCanonicalHostName` 得到的是主机名, 而 `getHostName` 得到的主机别名。`getCanonicalHostName` 的定义如下:

```
public String getCanonicalHostName()
```

在访问某些域名时, `getCanonicalHostName` 方法和 `getHostName` 方法的返回值是一样的, 这和 DNS 服务器如何解释主机名和主机别名以及它们的设置有关。如通过 `www.ibm.com` 创建 `InetAddress` 对象后, 使用 `getCanonicalHostName` 方法和 `getHostName` 方法返回的结果都是 `www.ibm.com` (有时直接返回 IP 地址, 这可能与 IBM 的 DNS 服务器的处理机制有关)。如果 DNS 不允许通过 IP 地址得到域名, 那么这两个方法就会返回 IP 地址来代替域名。`getCanonicalHostName` 方法可以分三种情况来讨论:

### 1. 使用 `getLocalHost` 创建 `InetAddress` 对象

在这种情况下 `getCanonicalHostName` 方法和 `getHostName` 方法得到的都是本机名。

### 2. 使用域名创建 `InetAddress` 对象

在这种情况下, `getCanonicalHostName` 方法是否要访问 DNS 服务器, 取决于 DNS 服务器如何解释主机名和主机别名——是否在创建 `InetAddress` 对象时就将主机名和主机别名都确定了。在前面已经讲过, 使用域名创建 `InetAddress` 对象后, 调用 `getHostName` 方法不会访问 DNS 服务器, 得到别名。但 `getCanonicalHostName` 方法就不一定了。这和 DNS 服务器的设置有关。如 `www.126.com` 就需要访问 DNS 服务器, 而 `www.ibm.com` 就不需要访问 DNS 服务器。

### 3. 使用 IP 地址创建 `InetAddress` 对象

在这种情况下, `getCanonicalHostName` 方法和 `getHostName` 方法是完全一样的, 也就是说, 它们得到的都是主机名, 而不是主机别名。

之所以要使用主机别名, 是因为有时主机名可能比较复杂, 如 Oracle 官方网站的主机名 `bigip-otn-portal.oracle.com`, 因此, 为了使用户访问网站更方便, 就增加了更简单的主机别名, 如 `www.oracle.com`。一个主机名可能对应多个主机别名, 如 `oracle.com` 也是 Oracle 的主机别名。在 IE 的地址栏中输入 `http:// bigip-otn-portal.oracle.com` 和 `http://oracle.com` 都可以访问 Oracle 官方网站。但我们发现, 有很多

网站通过主机名无法访问，只有通过一些别名才能访问，如 126 只能通过 www.126.com 和 126.com 两个主机别名访问，而不能通过它的主机名 zz-9-77-a8.bta.net.cn 来访问。这是因为在服务端通过 HTTP 协议做了限制，这个在前面已经讨论过了。例程 3-8 对比了 getCanonicalHostName 和 getHostName 方法在不同情况下的输出结果。

```
package mynet;

import java.net.*;

public class DomainName
{
    public static void outHostName(InetAddress address, String s)
    {
        System.out.println("通过" + s + "创建 InetAddress 对象");
        System.out.println("主 机 名:" + address.getCanonicalHostName());
        System.out.println("主机别名:" + address.getHostName());
        System.out.println("");
    }
    public static void main(String[] args) throws Exception
    {
        outHostName(InetAddress.getLocalHost(), "getLocalHost 方法");
        outHostName(InetAddress.getByName("www.ibm.com"), "www.ibm.com");
        outHostName(InetAddress.getByName("www.126.com"), "www.126.com");
        outHostName(InetAddress.getByName("202.108.9.77"), "202.108.9.77");
        outHostName(InetAddress.getByName("211.100.26.121"), "211.100.26.121");
    }
}
```

运行结果

通过 getLocalHost 方法创建 InetAddress 对象

主 机 名:ComputerName

主机别名:ComputerName

通过 www.ibm.com 创建 InetAddress 对象

主 机 名:www.ibm.com

主机别名:www.ibm.com

通过 www.126.com 创建 InetAddress 对象

主 机 名:zz-9-77-a8.bta.net.cn

主机别名:www.126.com

通过 202.108.9.77 创建 InetAddress 对象

主 机 名:zz-9-77-a8.bta.net.cn

主机别名:zz-9-77-a8.bta.net.cn

通过 211.100.26.121 创建 InetAddress 对象

主机名:211.100.26.121

主机别名:211.100.26.121

从上面的运行结果可以看出，如果 `InetAddress` 对象是通过 IP 地址创建的，`getCanonicalHostName` 方法和 `getHostName` 方法的值是完全一样的，它们的值可能同是主机名，也可能同是 IP 地址。而用域名创建的 `InetAddress` 对象就不一定了，它们的值可能相同（相同的 IP 地址或域名），也可能不相同，如上面运行结果中的 `www.126.com` 使用这两个方法得到的值就不同。在一般情况下，我们可以使用 `getHostName` 来获得域名，因为如果使用域名来创建 `InetAddress` 对象，`getHostName` 所得到的域名就是用来创建 `InetAddress` 对象的域名，如果使用 IP 地址来创建 `InetAddress` 对象，`getHostName` 方法等价于 `getCanonicalHostName` 方法。

## 7 用 `getHostAddress` 方法获得 IP

这个方法用来得到主机的 IP 地址，这个 IP 地址可能是 IPv4 的地址，也可能是 IPv6 的地址。`getHostAddress` 方法的定义如下：

```
public String getHostAddress(){...}
```

无论 `InetAddress` 对象是使用哪种方式创建的，`getHostAddress` 方法都不会访问 DNS 服务器。如果想访问使用 IPv6 地址的远程主机，需要在操作系统上安装 IPv6 协议。下面是 Windows 上安装 IPv6 协议的步骤：

**第一步：**打开“本地连接”属性对话框。

**第二步：**点击“安装”按钮，出现“选择网络组件类型”对话框，选择“协议”选项后，点击“添加”按钮，出现如图 2 的“选择网络协议”对话框，选择“Microsoft TCP/IP 版本 6”，最后点击“确定”按钮。

除了使用图形化界面来安装 IPv6 外，还可以使用如下命令行来安装 IPv6：

```
netsh interface ipv6 install
```

下面的代码演示了如何利用 `getHostAddress` 得到 IPv4 和 IPv6 地址，以及如何得到本机的所有 IP 地址（包括 IPv4 和 IPv6 地址）。

```
package mynet;
```

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

```
public class MyIP
```

```
{
```

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

```
    {
```

```
        // 输出 IPv4 地址
```

```
        InetAddress ipv4Address1 = InetAddress.getByName("1.2.3.4");
```

```
        System.out.println("ipv4Address1: " + ipv4Address1.getHostAddress());
```

```
        InetAddress ipv4Address2 = InetAddress.getByName("www.ibm.com");
```

```
        System.out.println("ipv4Address2: " + ipv4Address2.getHostAddress());
```

```
        InetAddress ipv4Address3 = InetAddress.getByName("myuniverse");
```

```
        System.out.println("ipv4Address3: " + ipv4Address3.getHostAddress());
```

```
        // 输出 IPv6 地址
```



```

    InetAddress ipv6Address1 = InetAddress.getByName("abcd:123::22ff");
    System.out.println("ipv6Address1: " + ipv6Address1.getHostAddress());
    InetAddress ipv6Address2 = InetAddress.getByName("www.neu6.edu.cn");
    System.out.println("ipv6Address2: " + ipv6Address2.getHostAddress());
    // 输出本机全部的 IP 地址
    InetAddress Addresses[] = InetAddress.getAllByName("myuniverse");
    for (InetAddress address : Addresses)
        System.out.println("本机地址: " + address.getHostAddress());
}
}

```

在上面代码中使用了 `www.neu6.edu.cn` 作为域名，这个域名是东北大学用于测试 IPv6 地址的域名。下面是其他一些可用于测试 IPv6 的域名，读者可以使用 `ping` 命令或例程 3-9 来测试这些域名。

```

www6.whu.edu.cn (武汉大学)
www.jlu6.edu.cn (吉林大学)
www6.usst.edu.cn (上海理工大学)
www.fudan6.edu.cn (复旦大学)

```

在访问这些域名之前，本机必须使用上述的方法或命令行安装 IPv6，否则 `getByName` 方法将抛出 `UnknownHostException` 异常。

运行结果：

```

ipv4Address1: 1.2.3.4
ipv4Address2: 129.42.60.212
ipv4Address3: 192.168.18.10
ipv6Address1: abcd:123:0:0:0:0:0:22ff
ipv6Address2: 2001:da8:9000:b255:200:ε8ff:feb0:5c5e
本机地址: 192.168.18.10
本机地址: 192.168.83.1
本机地址: 192.168.189.1
本机地址: 193.10.10.10
本机地址: 0:0:0:0:0:0:0:1

```

在上面的运行结果中的 IP 地址 `192.168.18.10` 和 `192.10.10.10` 是和本机网卡绑定的两个 IP，而 `192.168.83.1` 和 `192.168.189.1` 是 VMware 虚拟机软件在本机安装的两个虚拟网卡的地址。最后一个 IPv6 地址 `0:0:0:0:0:0:0:1` 是代表本机的 IPv6 网址，相当于 IPv4 地址的 `127.0.0.1`。读者可以使用如下命令行添加 IPv6 地址和删除 IPv6：

添加 IPv6 地址

```
netsh interface ipv6 add address "本地连接" aa:bb::cc
```

删除 IPv6

```
netsh interface ipv6 uninstall
```

**注意：**安装 IPv6 不需要重新启动计算机，但添加 IPv6 地址或删除 IPv6 后，必须重新启动计算机才能生效。

## 8 用 getAddress 方法获得 IP 地址

`getAddress` 方法和 `getHostAddress` 类似，它们的唯一区别是 `getHostAddress` 方法返回的是字符串形式的 IP 地址，而 `getAddress` 方法返回的是 `byte` 数组形式的 IP 地址。`getAddress` 方法的定义如下：

```
public byte[] getAddress()
```

这个方法返回的 `byte` 数组是有符号的。在 Java 中 `byte` 类型的取值范围是 `-128~127`。如果返回的 IP 地址的某个字节是大于 127 的整数，在 `byte` 数组中就是负数。由于 Java 中没有无符号 `byte` 类型，因此，要想显示正常的 IP 地址，必须使用 `int` 或 `long` 类型。下面代码演示了如何利用 `getAddress` 返回 IP 地址，以及如何将 IP 地址转换成正整数形式。

```
package mynet;
```

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

```
public class MyIP
```

```
{  
    public static void main(String[] args) throws Exception  
    {  
        InetAddress address = InetAddress.getByName("www.csdn.net");  
        byte ip[] = address.getAddress();  
        for (byte ipSegment : ip)  
            System.out.print(ipSegment + " ");  
        System.out.println("");  
        for (byte ipSegment : ip)  
        {  
            int newIPSegment = (ipSegment < 0) ? 256 + ipSegment : ipSegment;  
            System.out.print(newIPSegment + " ");  
        }  
    }  
}
```

运行结果：

```
-45 100 26 122  
211 100 26 122
```

从上面的运行结果可以看出，第一行输出了未转换的 IP 地址，由于 `www.csdn.net` 的 IP 地址的第一个字节大于 127，因此，输出了一个负数。而第二行由于将 IP 地址的每一个字节转换成了 `int` 类型，因此，输出了正常的 IP 地址。

## 9 使用 `is*` 方法判断地址类型

IP 地址分为普通地址和特殊地址。在前面的文章中所使用的大多数都是普通的 IP 地址，在本文中介绍如何利用 `InetAddress` 类提供的十个方法来确定一个 IP 地址是否是一个特殊的 IP 地址。

### 一、`isAnyLocalAddress` 方法

当 IP 地址是[通配符地址](#)时返回 `true`，否则返回 `false`。这个通配符地址对于拥有多个网络接口（如两块网卡）的计算机非常拥有。[使用通配符地址可以允许在服务器主机接受来自任何网络接口的客户端连接](#)。IPv4 的通配符地址是 `0.0.0.0`。IPv6 的通配符地址是 `0:0:0:0:0:0:0:0`，也可以简写成：

## 二、isLoopbackAddress 方法

当 IP 地址是 loopback 地址时返回 true，否则返回 false。loopback 地址就是代表本机的 IP 地址。IPv4 的 loopback 地址的范围是 127.0.0.0 ~ 127.255.255.255，也就是说，只要第一个字节是 127，就是 loopback 地址。如 127.1.2.3、127.0.200.200 都是 loopback 地址。IPv6 的 loopback 地址是 0:0:0:0:0:0:0:1，也可以简写成::1。我们可以使用 ping 命令来测试 loopback 地址。如下面的命令行所示：

```
ping 127.200.200.200
```

运行结果：

```
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
Reply from 127.0.0.1: bytes=32 time<1ms TTL=128
```

```
Ping statistics for 127.200.200.200:
```

```
Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
```

```
Approximate round trip times in milli-seconds:
```

```
Minimum = 0ms, Maximum = 0ms, Average = 0ms
```

虽然 127.255.255.255 也是 loopback 地址，但 127.255.255.255 在 Windows 下是无法 ping 通的。这是因为 127.255.255.255 是广播地址，在 Windows 下对发给广播地址的请求不做任何响应，而在其他操作系统上根据设置的不同，可能会得到不同的结果。

```
st1":*{behavior:url(#ieooui) }
```

## 三、isLinkLocalAddress 方法

当 IP 地址是本地连接地址(LinkLocalAddress)时返回 true，否则返回 false。IPv4 的本地连接地址的范围是 169.254.0.0 ~ 169.254.255.255。IPv6 的本地连接地址的前 12 位是 FE8，其他的位可以是任意取值，如 FE88::、FE80::ABCD::都是本地连接地址。

## 四、isSiteLocalAddress 方法

当 IP 地址是地区本地地址 (SiteLocalAddress) 时返回 true，否则返回 false。IPv4 的地区本地地址分为三段：10.0.0.0 ~ 10.255.255.255、172.16.0.0 ~ 172.31.255.255、192.168.0.0 ~ 192.168.255.255。IPv6 的地区本地地址的前 12 位是 FEC，其他的位可以是任意取值，如 FED0::、FEF1::都是地区本地地址。

## 五、isMulticastAddress 方法

当 IP 地址是广播地址 (MulticastAddress) 时返回 true，否则返回 false。通过广播地址可以向网络中的所有计算机发送信息，而不是只向一台特定的计算机发送信息。IPv4 的广播地址的范围是 224.0.0.0 ~ 239.255.255.255。IPv6 的广播地址第一个字节是 FF，其他的字节可以是任意值。关于广播地址的详细内容将在以后的章节中讨论。

## 六、isMCGlobal 方法

当 IP 地址是全球范围的广播地址时返回 true，否则返回 false。全球范围的广播地址可以向 Internet 中的所有的计算机发送信息。IPv4 的广播地址除了 224.0.0.0 和第一个字节是 239 的 IP 地址都是全球范围的广播地址。IPv6 的全球范围的广播地址中第一个字节是 FF，第二个字节的范围是 0E ~ FE，其他的字节可以是任意值，如 FFBE::、FF0E::都是全球范围的广播地址。

## 七、isMCLinkLocal 方法

当 IP 地址是子网广播地址时返回 true，否则返回 false。使用子网的广播地址只能向子网内的计算机发送信息。IPv4 的子网广播地址的范围是 224.0.0.0 ~ 224.0.0.255。IPv6 的子网广播地址的第一个字节是 FF，第二个字节的范围是 02 ~ F2，其他的字节可以是任意值，如 FFB2::、FF02:ABCD::都是子网广播地址。

## 八、isMCNodeLocal 方法

当 IP 地址是本地接口广播地址时返回 `true`，否则返回 `false`。本地接口广播地址不能将广播信息发送到产生广播信息的网络接口，即使是同一台计算机的另一个网络接口也不行。**所有的 IPv4 广播地址都不是本地接口广播地址**。IPv6 的本地接口广播地址的第一个字节是 `FF`，第二个字节的范围是 `01 ~ F1`，其他的字节可以是任意值，如 `FFB1::`、`FF01:A123::` 都是本地接口广播地址。

## 九、isMCOrgLocal 方法

当 IP 地址是组织范围的广播地址时返回 `true`，否则返回 `false`。使用组织范围广播地址可以向公司或企业内部的所有的计算机发送广播信息。**IPv4 的组织范围广播地址的第一个字节是 239，第二个字节不小于 192，第三个字节不大于 195**，如 `239.193.100.200`、`239.192.195.0` 都是组织范围广播地址。IPv6 的组织范围广播地址的第一个字节是 `FF`，第二个字节的范围是 `08 ~ F8`，其他的字节可以是任意值，如 `FF08::`、`FF48::` 都是组织范围的广播地址。

## 十、isMCSiteLocal 方法

当 IP 地址是站点范围的广播地址时返回 `true`，否则返回 `false`。使用站点范围的广播地址，可以向站点范围内的计算机发送广播信息。**IPv4 的站点范围广播地址的范围是 239.255.0.0 ~ 239.255.255.255**，如 `239.255.1.1`、`239.255.0.0` 都是站点范围的广播地址。IPv6 的站点范围广播地址的第一个字节是 `FF`，第二个字节的范围是 `05 ~ F5`，其他的字节可以是任意值，如 `FF05::`、`FF45::` 都是站点范围的广播地址。

下面的代码可以确定一个 IP 地址是否在上述十种地址类型的范围内：

```
package test;

import java.net.*;
import java.lang.reflect.*;

public class MyNet
{
    public static void main(String[] args) throws Exception
    {
        if (args.length == 0)
            return;
        InetAddress address = InetAddress.getBy_name(args[0]);
        Method methods[] = InetAddress.class.getMethods();
        // 以 is 开头并且没有参数的方法
        for (Method method : methods)
        {
            if (method.getName().matches("is.*") && method.getParameterTypes().length == 0)
            {
                if (Boolean.parseBoolean(method.invoke(address).toString()))
                    System.out.println(method.getName() + " = true");
            }
        }
    }
}
```

- **测试 1**

执行如下命令：

```
java test.MyNet 224.0.0.1
```

运行结果:

```
isMCLinkLocal = true
```

```
isMulticastAddress = true
```

- 测试 2

执行如下命令:

```
java test.MyNet FFB1::
```

运行结果:

```
isMCNodeLocal = true
```

```
isMulticastAddress = true
```

如果未输出任何结果,说明指定的 IP 地址并不属性上述的十种 IP 地址类型的范围,只是一个普通的 IP 地址。

## 10 Inet4Address 类和 Inet6Address 类

为了区分 IPv4 和 IPv6 地址,Java 提供了两个类: Inet4Address 和 Inet6Address, 它们都是 InetAddress 类的子类, 这两个类的定义如下:

```
public final class Inet4Address extends InetAddress
```

```
public final class Inet6Address extends InetAddress
```

InetAddress 类中没有 public 方法, 这两个类分别按着 IPv4 和 IPv6 的规则实现了 InetAddress 类中的 public 方法。它们所不同的是 Inet6Address 类比 Inet4Address 类多了一个方法: isIPv4CompatibleAddress, 这个方法用来判断一个 IPv6 地址是否和 IPv4 地址兼容。和 IPv4 兼容的 IPv6 地址除了最后四个字节有值名, 其他的字节都是 0, 如 0:0:0:0:0:0:192.168.18.10 、 ::ABCD:FAFA 都是和 IPv4 兼容的 IPv6 地址。

当使用 InetAddress 类的四个静态方法创建 InetAddress 对象后, 可以通过 getAddress 返回的 byte 数组 length 来判断这个 IP 地址是 IPv4 还是 IPv6 地址 (byte 数组长度为 4 就是 IPv4 地址, byte 数组长度为 16 就是 IPv6 地址), 另外也可以通过 instanceof 来确定 InetAddress 对象是它的哪个子类的实例来断定。下面的代码演示了如何判断一个 IP 地址是 IPv4 还是 IPv6 地址:

```
package mynet;
```

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

```
public class MyIP
```

```
{
```

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

```
    {
```

```
        if (args.length == 0)
```

```
            return;
```

```
        InetAddress address = InetAddress.getByName(args[0]);
```

```
        System.out.println("IP: " + address.getHostAddress());
```

```
        switch (address.getAddress().length)
```

```
        {
```

```
            case 4:
```

```

        System.out.println("根据 byte 数组长度判断这个 IP 地址是 IPv4 地址!");
        break;
    case 16:
        System.out.println("根据 byte 数组长度判断这个 IP 地址是 IPv6 地址!");
        break;
    }
    if (address instanceof Inet4Address)
        System.out.println("使用 instanceof 判断这个 IP 地址是 IPv4 地址!");
    else if (address instanceof Inet6Address)
        System.out.println("使用 instanceof 判断这个 IP 地址是 IPv6 地址!");
    }
}

```

st1":\*{behavior:url(#ieooui) } 测试 1

执行如下命令:

```
java mynet.MyIP www.csdn.net
```

运行结果:

IP: 211.100.26.122

根据 byte 数组长度判断这个 IP 地址是 IPv4 地址!

使用 instanceof 判断这个 IP 地址是 IPv4 地址!

**测试 2**

执行如下命令:

```
java mynet.MyIP www.neu6.edu.cn
```

运行结果:

IP: 2001:da8:9000:b255:200:e8ff:feb0:5c5e

根据 byte 数组长度判断这个 IP 地址是 IPv6 地址!

使用 instanceof 判断这个 IP 地址是 IPv6 地址!

## 11 用 NetworkInterface 类获得网络接口信息

从 JDK1.4 开始, Java 提供了一个 NetworkInterface 类。这个类可以得到本机所有的物理网络接口和虚拟机等软件利用本机的物理网络接口创建的逻辑网络接口的信息。

### 一、创建 NetworkInterface 对象的两个静态方法

NetworkInterface 类和 InetAddress 一样, 也没有 public 的构造方法。因此, 必须通过它的两个静态方法来创建 NetworkInterface 对象: getByName 方法(网络接口名)和 getByInetAddress 方法(IP 地址)。NetworkInterface 对象的 toString 方法可以返回网络接口的名称、显示名和这个网络接口上绑定的所有 IP 地址等信息。

#### 1. getByName 方法

getByName 方法可以通过网络接口名来创建 NetworkInterface 对象。这个网络接口名并不是计算机名, 而是用于标识物理或逻辑网络接口的名字, 一般是由操作系统设置的。网络接口名在大多数操作系统上(包括 Windows、Linux 和 Unix)是以 eth 开头, 后面是网络接口的索引号, 从 0 开始。如本机安了三块网卡,

那么就对应三个接口，网络接口名就依次是 `eth0`、`eth1` 和 `eth2`。当网络接口名不存在时，`getByName` 返回 `null`。`getByName` 方法定义如下：

```
public static NetworkInterface getByName(String name) throws SocketException
```

下面的代码是一个显示指定网络接口信息的程序，网络接口名通过命令行参数传入。

```
package mynet;

import java.net.*;

public class MyNetworkInterface1
{
    public static void main(String[] args) throws Exception
    {
        if (args.length == 0)
            return;
        NetworkInterface ni = NetworkInterface.getByName(args[0]);
        System.out.println((ni == null) ? "网络接口不存在!" : ni);
    }
}
```

- **测试 1**

执行如下命令：

```
java mynet.MyNetworkInterface1 eth0
```

运行结果：

```
name:eth0 (Realtek RTL8139 Family PCI Fast Ethernet NIC) index: 4 addresses:
/192.168.18.10;
/192.168.18.20;
```

- **测试 2**

执行如下命令：

```
java mynet.MyNetworkInterface1 abcd
```

运行结果：

```
网络接口不存在!
```

## 2. `getByInetAddress` 方法

除了可以使用网络接口名来得到网络接口的信息，还可以利用 `getByInetAddress` 方法来确定一个 IP 地址属于哪一个网络接口。由于 `getByInetAddress` 方法必须使用一个 `InetAddress` 对象封装的 IP 地址来作为参数，因此，在使用 `getByInetAddress` 方法之前，必须先创建一个 `InetAddress` 对象。但要注意不能使用远程的 IP 的域名来创建 `InetAddress` 对象，否则 `getByInetAddress` 将返回 `null`。`getByInetAddress` 方法的定义如下：

```
public static NetworkInterface getByInetAddress(InetAddress addr) throws SocketException
```

下面代码可以确定一个 IP 地址属于哪一个网络接口，这个 IP 地址通过命令行参数传入。

```
package mynet;

import java.net.*;
```



```

public class MyNetworkInterface2
{
    public static void main(String[] args) throws Exception
    {
        if(args.length == 0) return;
        InetAddress local = InetAddress.getByName(args[0]);
        // 注意此行的 getByName 与上一个方法不再同一个类
        NetworkInterface ni = NetworkInterface.getByInetAddress(local);
        System.out.println((ni == null) ? "本机不存在此 IP 地址!" : ni);
    }
}

```

- **测试 1**

执行如下命令：

```
java mynet.MyNetworkInterface2 127.0.0.1
```

运行结果：

```

name: lo (MS TCP Loopback interface) index: 1 addresses:
/127.0.0.1;
/0:0:0:0:0:0:0:1;

```

- **测试 2**

执行如下命令：

```
java mynet.MyNetworkInterface2 218.61.151.22
```

运行结果：

```

name: ppp0 (WAN (PPP/SLIP) Interface) index: 0 addresses:
/218.61.151.22;

```

注意：测试 2 使用的 IP 地址 218.61.151.22 是 ADSL 连接临时分配给本机的 IP 地址，因此，运行结果返回的 ppp0 是 ADSL 网络接口。

## 二、得到本机所有的网络接口

NetworkInterface 可以通过 `getNetworkInterfaces` 方法来枚举本机所有的网络接口。我们也可以利用 `getNetworkInterfaces` 得到的网络接口来枚举本机的所有 IP 地址。当然，也可以通过 `InetAddress` 类的 `getAllByName` 来得到本机的所有 IP 地址。但 `getNetworkInterfaces` 方法可以按网络接口将这些 IP 地址进行分组，这对于只想得到某个网络接口上的所有 IP 地址是非常有用的。

`getNetworkInterfaces` 方法的定义如下：

```
public static Enumeration<NetworkInterface> getNetworkInterfaces() throws SocketException
```

下面代码演示了如何使用 `getNetworkInterfaces` 方法得到本机所有的网络接口。

```

package mynet;

import java.net.*;
import java.util.*;

public class MyNetworkInterface3
{
    public static void main(String[] args) throws Exception
    {

```

```

Enumeration<NetworkInterface> nis = NetworkInterface.getNetworkInterfaces();
while (nis.hasMoreElements())
    System.out.println(nis.nextElement());
}
}

```

运行结果（部分）：

```

name: lo (MS TCP Loopback interface) index: 1 addresses:
/127.0.0.1;
/0:0:0:0:0:0:0:1;
name: eth0 (Realtek RTL8139 Family PCI Fast Ethernet NIC ) index: 4 addresses:
/192.168.18.10;
/192.168.18.20;
name: ppp0 (WAN (PPP/SLIP) Interface) index: 0 addresses:
/218.61.151.22;

```

上面的运行结果只是一种可能的结果，读者在运行上面的程序时根据本机的硬件和软件的配置不同，运行结果可能会有所不同。

### 三、NetworkInterface 类的 Getter 方法

NetworkInterface 类提供了三个方法可以分别得到网络接口名(getName 方法)、网络接口别名(getDisplayName 方法)以及和网络接口绑定的所有 IP 地址(getInetAddresses 方法)。

#### 1. getName 方法

这个方法用来得到一个网络接口的名称。这个名称就是使用 getByname 方法创建 NetworkInterface 对象时使用的网络接口名，如 eth0、ppp0 等。getName 方法的定义如下：

```
public String getName()
```

#### 2. getDisplayName 方法

这个方法可以得到更容易理解的网络接口名，也可以将这个网络接口名称为网络接口别名。在一些操作系统中（如 Unix），getDisplayName 方法和 getName 方法的返回值相同，但在 Windows 中 getDisplayName 方法一般会返回一个更为友好的名字，如 Realtek RTL8139 Family PCI Fast Ethernet NIC。getDisplayName 方法的定义如下：

```
public String getDisplayName()
```

#### 3. getInetAddresses 方法

NetworkInterface 类可以通过 getInetAddresses 方法以 InetAddress 对象的形式返回和网络接口绑定的所有 IP 地址。getInetAddresses 方法的定义如下：

```
public Enumeration<InetAddress> getInetAddresses()
```

上面的代码演示了如果使用上述三个 Getter 方法。

```

package mynet;

import java.net.*;
import java.util.*;

public class MyNetworkInterface4
{
    public static void main(String[] args) throws Exception
    {

```

```

    if (args.length == 0)
        return;
    NetworkInterface ni = NetworkInterface.getBy_name(args[0]);
    System.out.println("Name: " + ni.getName());
    System.out.println("DisplayName: " + ni.getDisplayName());
    Enumeration<InetAddress> addresses = ni.getInetAddresses();
    while (addresses.hasMoreElements())
        System.out.println(addresses.nextElement().getHostAddress());
}
}

```

## 测试

执行如下命令：

```
java mynet.MyNetworkInterface4 eth0
```

运行结果：

```

Name: eth0
DisplayName: Realtek RTL8139 Family PCI Fast Ethernet NIC
192.168.18.10
192.168.18.20

```

## 12 用 isReachable 方法探测主机是否可达

在 J2SE5.0 中的 `InetAddress` 类中增加了一个 `isReachable` 方法。可以使用这个方法探测主机是否可以连通。这个方法有两个重载形式，它们的定义如下：

```

public boolean isReachable(int timeout) throws IOException
public boolean isReachable(NetworkInterface netif, int ttl, int timeout) throws IOException

```

第一个重载形式有一个 `timeout` 参数，可以通过这个参数设置连接超时（单位：毫秒）。第二个重载形式多了两个参数：`netif` 和 `ttl`。通过 `netif` 参数可以使用一个 `NetworkInterface` 对象来确定客户端使用哪个网络接口来测试主机的连通性。`ttl` 是指测试连通性过程中的最大连接跃点数（从客户机到达远程主机所经过的最大路由数就是最大连接跃点数，一个路由被称为一个跃点，在 Windows 网络连接中的“高级 TCP/IP 设置”对话框最下面可以设置接口跃点数），如果达到最大连接跃点数，还没找到远程主机，`isReachable` 方法就认为客户机和远程主机之间是不可连通的。

`isReachable` 方法是通过连接主机的 `echo` 端口来确定客户端和服务端是否可连通。但在 Internet 上使用这个方法可能会因为防火墙等因素而无法连通远程主机——实际上，远程主机是可以连通的。因此，`isReachable` 在 Internet 上并不可靠。但我们可以将 `isReachable` 方法应用于局域网中。

## 13 用 Socket 类接收和发送数据

网络应用分为客户端和服务端两部分，而 `Socket` 类是负责处理客户端通信的 Java 类。通过这个类可以连接到指定 IP 或域名的服务器上，并且可以和服务器互相发送和接受数据。

在本文及后面的数篇文章中将详细讨论 **Socket** 类的使用，内容包括 **Socket** 类基础、各式各样的连接方式、**get** 和 **set** 方法、连接过程中的超时以及关闭网络连接等。

在本文中，我们将讨论使用 **Socket** 类的基本步骤和方法。一般网络客户端程序在连接服务程序时要进行以下三步操作。

1. 连接服务器
2. 发送和接收数据
3. 关闭网络连接

## 一、连接服务器

在客户端可以通过两种方式连接服务器，一种是通过 **IP** 的方式来连接服务器，而另外一种是通过域名方式来连接服务器。

其实这两种方式从本质上来看是一种方式。在底层客户端都是通过 **IP** 来连接服务器的，但这两种方式有一定的差异，如果通过 **IP** 方式来连接服务端程序，客户端只简单地根据 **IP** 进行连接，如果通过域名来连接服务器，客户端必须通过 **DNS** 将域名解析成 **IP**，然后再根据这个 **IP** 来进行连接。

在很多程序设计语言或开发工具中（如 **C/C++**、**Delphi**）使用域名方式连接服务器时必须自己先将域名解析成 **IP**，然后再通过 **IP** 进行连接，而在 **Java** 中已经将域名解析功能包含在了 **Socket** 类中，因此，我们只需象使用 **IP** 一样使用域名即可。

通过 **Socket** 类连接服务器程序最常用的方法就是通过 **Socket** 类的构造函数将 **IP** 或域名以及端口号作为参数传入 **Socket** 类中。**Socket** 类的构造函数有很多重载形式，在这一节只讨论其中最常用的一种形式：**public Socket(String host, int port)**。从这个构造函数的定义来看，只需要将 **IP** 或域名以及端口号直接传入构造函数即可。下面的代码是一个连接服务端程序的例子程序：

```
package mysocket;

import java.net.*;

public class MyConnection
{
    public static void main(String[] args)
    {
        try
        {
            if (args.length > 0)
            {
                Socket socket = new Socket(args[0], 80);
                System.out.println(args[0] + "已连接成功!");
            }
            else
                System.out.println("请指定 IP 或域名!");
        }
        catch (Exception e)
        {
            System.err.println("错误信息: " + e.getMessage());
        }
    }
}
```

在上面的中,通过命令行参数将 IP 或域名传入程序,然后通过 `Socket socket = new Socket(args[0], 80)` 连接通过命令行参数所指定的 IP 或域名的 80 端口。由于 `Socket` 类的构造函数在定义时使用了 `throws`, 因此,在调用 `Socket` 类的构造函数时,必须使用 `try...catch` 语句来捕捉错误,或者对 `main` 函数使用 `throws` 语句来抛出错误。

#### 测试正确的 IP

```
java mysocket.MyConnection 127.0.0.1
```

输出结果: 127.0.0.1 已经连接成功!

#### 测试错误的 IP

```
java mysocket.MyConnection 10.10.10.10
```

输出结果: 错误信息: Connection timed out: connect

注: 10.10.10.10 是一个并不存在的 IP, 如果这个 IP 在你的网络中存在, 请使用其它的不存在的 IP。

#### 测试正确的域名

```
java mysocket.MyConnection www.ptpress.com.cn
```

输出结果: www.ptpress.com.cn 已经连接成功!

#### 测试错误的域名

```
java mysocket.MyConnection www.ptpress1.com.cn
```

输出结果: 错误信息: www.ptpress1.com.cn

另外, 喜欢病毒编程的同学可以使用 `Socket` 类连接服务器可以判断一台主机有哪些端口被打开。下面的代码是一个扫描本机有哪些端口被打开的程序。

```
package mysocket;

import java.net.*;

public class MyConnection1 extends Thread
{
    private int minPort, maxPort;

    public MyConnection1(int minPort, int maxPort)
    {
        this.minPort = minPort;
        this.maxPort = maxPort;
    }

    public void run()
    {
        for (int i = minPort; i <= maxPort; i++)
        {
            try
            {
                Socket socket = new Socket("127.0.0.1", i);
                System.out.println(String.valueOf(i) + ":ok");
                socket.close();
            }
            catch (Exception e)
```

```

        {
        }
    }
}
public static void main(String[] args)
{
    int minPort = Integer.parseInt(args[0]);
    int maxPort = Integer.parseInt(args[1]);
    int threadCount = Integer.parseInt(args[2]);
    //模拟线程
    int portIncrement = ((maxPort - minPort + 1) / threadCount)
        + (((maxPort - minPort + 1) % threadCount) == 0 ? 0 : 1);
    MyConnection1[] instances = new MyConnection1[threadCount];
    for (int i = 0; i < threadCount; i++)
    {
        instances[i] = new MyConnection1(minPort + portIncrement * i, minPort
            + portIncrement - 1 + portIncrement * i);
        instances[i].start();
    }
}
}

```

上面代码通过一个指定的端口范围（如 1 至 1000），并且利用多线程将这个端口范围分成不同的段进行扫描，这样可以大大提高扫描的效率。

可通过如下命令行去运行例程 4-2。

```
java mysocket.MyConnection1 1000 3000 20
```

## 二、发送和接收数据

在 `Socket` 类中最重要的两个方法就是 `getInputStream` 和 `getOutputStream`。这两个方法分别用来得到用于读取和写入数据的 `InputStream` 和 `OutputStream` 对象。在这里的术语是以客户端为中心的，`InputStream` 读取的是服务器程序向客户端发送过来的数据，而 `OutputStream` 是客户端要向服务端程序发送的数据。

在编写实际的网络客户端程序时，是使用 `getInputStream`，还是使用 `getOutputStream`，以及先使用谁后使用谁由具体的应用决定。如通过连接邮电出版社网站([www.ptpress.com.cn](http://www.ptpress.com.cn))的 80 端口（一般为 HTTP 协议所使用的默认端口），并且发送一个字符串，最后再读取从 [www.ptpress.com.cn](http://www.ptpress.com.cn) 返回的信息。

```

package mysocket;

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

public class MyConnection2
{
    public static void main(String[] args) throws Exception
    {
        Socket socket = new Socket("www.ptpress.com.cn", 80);
        // 向服务端程序发送数据
    }
}

```

```

OutputStream ops = socket.getOutputStream();
OutputStreamWriter opsw = new OutputStreamWriter(ops);
BufferedWriter bw = new BufferedWriter(opsw);
bw.write("hello world\r\n\r\n");
bw.flush();

// 从服务端程序接收数据
InputStream ips = socket.getInputStream();
InputStreamReader ipsr = new InputStreamReader(ips);
BufferedReader br = new BufferedReader(ipsr);
String s = "";
while((s = br.readLine()) != null)
    System.out.println(s);
socket.close();
}
}

```

在编写上面代码时要注意如下两点：

1. 为了提高数据传输的效率，Socket 类并没有在每次调用 write 方法后都进行数据传输，而是将这些要传输的数据写到一个缓冲区里（默认是 8192 个字节），然后通过 flush 方法将这个缓冲区里的数据一起发送出去，因此，bw.flush() 是必须的。
2. 在发送字符串时之所以在 Hello World 后加上 “\r\n\r\n”，这是因为 HTTP 协议头是以 “\r\n\r\n” 作为结束标志（HTTP 协议的详细内容将在以后讲解），因此，通过在发送字符串后加入 “\r\n\r\n”，可以使服务端程序认为 HTTP 头已经结束，可以处理了。如果不加 “\r\n\r\n”，那么服务端程序将一直等待 HTTP 头的结束，也就是 “\r\n\r\n”。如果是这样，服务端程序就不会向客户端发送响应信息，而 br.readLine() 将因无法读以响应信息而被阻塞，直到连接超时。

### 三、关闭网络连接

到现在为止，我们对 Socket 类的基本使用方法已经有了初步的了解，但在 Socket 类处理完数据后，最合理的收尾方法是使用 Socket 类的 close 方法关闭网络连接。虽然在中已经使用了 close 方法，但使网络连接关闭的方法不仅仅只有 close 方法，下面就让我们看看 Java 在什么情况下可以使网络连接关闭。

可以引起网络连接关闭的情况有以下 4 种：

1. 直接调用 Socket 类的 close 方法。
2. 只要 Socket 类的 InputStream 和 OutputStream 有一个关闭，网络连接自动关闭（必须通过调用 InputStream 和 OutputStream 的 close 方法关闭流，才能使网络连接自动关闭）。
3. 在程序退出时网络连接自动关闭。
4. 将 Socket 对象设为 null 或未关闭最使用 new Socket(...) 建立新对象后，由 JVM 的垃圾回收器回收为 Socket 对象分配的内存空间后自动关闭网络连接。

虽然这 4 种方法都可以达到同样的目的，但一个健壮的网络程序最好使用第 1 种或第 2 种方法关闭网络连接。这是因为第 3 种和第 4 种方法一般并不会马上关闭网络连接，如果是这样的话，对于某些应用程序，将会遗留大量无用的网络连接，这些网络连接会占用大量的系统资源。

在 Socket 对象被关闭后，我们可以通过 isClosed 方法来判断某个 Socket 对象是否处于关闭状态。然而使用 isClosed 方法所返回的只是 Socket 对象的当前状态，也就是说，不管 Socket 对象是否曾经连接成功，只要处于关闭状态，isClosed 就返回 true。如果只是建立一个未连接的 Socket 对象，isClose 则会返回 false。如下面的代码将输出 false。



```
Socket socket = new Socket();
System.out.println(socket.isClosed());
```

除了 `isClose` 方法，`Socket` 类还有一个 `isConnected` 方法来判断 `Socket` 对象是否连接成功。顾名思义，`isConnected` 方法所判断的并不是 `Socket` 对象的当前连接状态，而是 `Socket` 对象是否曾经连接成功过，如果成功连接过，即使现在 `isClosed` 返回 `true`，`isConnected` 仍然返回 `true`。因此，要判断当前的 `Socket` 对象是否处于连接状态，必须同时使用 `isClose` 和 `isConnected` 方法，即只有当 `isClosed` 返回 `false` 且 `isConnected` 返回 `true` 的时候 `Socket` 对象才处于连接状态。下面的代码演示了上述 `Socket` 对象的各种状态的产生过程。

```
package mysocket;

import java.net.*;

public class MyCloseConnection
{
    public static void printState(Socket socket, String name)
    {
        System.out.println(name + ".isClosed():" + socket.isClosed());
        System.out.println(name + ".isConnected():" + socket.isConnected());
        if (socket.isClosed() == false && socket.isConnected() == true)
            System.out.println(name + "处于连接状态!");
        else
            System.out.println(name + "处于非连接状态!");
        System.out.println();
    }

    public static void main(String[] args) throws Exception
    {
        Socket socket1 = null, socket2 = null;

        socket1 = new Socket("www.ptpress.com.cn", 80);
        printState(socket1, "socket1");

        socket1.getOutputStream().close();
        printState(socket1, "socket1");

        socket2 = new Socket();
        printState(socket2, "socket2");

        socket2.close();
        printState(socket2, "socket2");
    }
}
```

运行上面的代码后，将有如下的输出结果：

`socket1.isClosed():false`

```
socket1.isConnected(): true
socket1 处于连接状态!
socket1.isClosed(): true
socket1.isConnected(): true
socket1 处于非连接状态!
socket2.isClosed(): false    //socket 是打开的
socket2.isConnected(): false //但是他还没有连过服务器
socket2 处于非连接状态!    //所以....此中有深意!
socket2.isClosed(): true
socket2.isConnected(): false
socket2 处于非连接状态!
```

从输出结果可以看出，在 socket1 的 OutputStream 关闭后，socket1 也自动关闭了。而在上面的代码我们可以看出，对于一个并未连接到服务端的 Socket 对象 socket2，它的 isClosed 方法为 false，而要想让 socket2 的 isClosed 方法返回 true，必须使用 socket2.close 显示地调用 close 方法。

虽然在大多数的时候可以直接使用 Socket 类或输入输出流的 close 方法关闭网络连接，但有时我们只希望关闭 OutputStream 或 InputStream，而在关闭输入输出流的同时，并不关闭网络连接。这就需要用到 Socket 类的另外两个方法：shutdownInput 和 shutdownOutput，这两个方法只关闭相应的输入、输出流，而它们并没有同时关闭网络连接的功能。和 isClosed、isConnected 方法一样，Socket 类也提供了两个方法来判断 Socket 对象的输入、输出流是否被关闭，这两个方法是 isInputShutdown() 和 isOutputShutdown()。下面的代码演示了只关闭输入、输出流的过程：

```
package mysocket;

import java.net.*;

public class MyCloseConnection1
{
    public static void printState(Socket socket)
    {
        System.out.println("isInputShutdown:" + socket.isInputShutdown());
        System.out.println("isOutputShutdown:" + socket.isOutputShutdown());
        System.out.println("isClosed:" + socket.isClosed());
        System.out.println();
    }

    public static void main(String[] args) throws Exception
    {
        Socket socket = new Socket("www.ptpress.com.cn", 80);
        printState(socket);

        socket.shutdownInput();
        printState(socket);

        socket.shutdownOutput();
        printState(socket);
    }
}
```

```
}  
}
```

在运行上面的代码后，将得到如下的输出结果：

```
isInputShutdown: false  
isOutputShutdown: false  
isClosed: false  
isInputShutdown: true  
isOutputShutdown: false  
isClosed: false  
isInputShutdown: true  
isOutputShutdown: true  
isClosed: false
```

从输出结果可以看出，isClosed 方法一直返回 false，因此，可以肯定，shutdownInput 和 shutdownOutput 并不影响 Socket 对象的状态。

**注意：**要想看懂本文需要理解三个级别的开关状态：socket 级别，连接级别，流级别。

## 14 多种多样的建立网络连接的方式

在上一篇文章中我们讨论了 Socket 类的基本用法，并给出的例子中使用 Socket 类连接服务器时使用了一种最简单的连接方式，也就是通过 IP 和端口号来连接服务器。为了使连接服务器的方式更灵活，Socket 类不仅可以通过自身的构造方法连接服务器，而且也可以通过 connect 方法来连接数据库。

### 一、通过构造方法连接服务器

我们可以通过 6 个重载构造函数以不同的方式来连接服务器。这 6 个重载的构造函数可以分为两类：

#### 1. 自动选择 IP

这种方式是最常用的。所谓自动选择 IP，是指当本机有多块网卡或者在一个网卡上绑定了多个 IP 时，Socket 类会自动为我们选择一个可用的 IP。在上述 6 个构造方法中有 4 个是使用这种方法来连接服务器的。

##### (1) public Socket(String host, int port)

一个字符串类型的 IP 或域名以及一个整型的端口号即可。在这个构造方法中可能会抛出两个错误：UnknownHostException 和 IOException。发生第一个错误的原因是我们提供的 host 并不存在或不合法，而其它的错误被归为 IO 错误。因此，这个构造方法的完整定义是：

```
public Socket(String host, int port) throws UnknownHostException, IOException
```

##### (2) public Socket(InetAddress inetaddress, int port)

这个构造方法和第一种构造方法类似，只是将字符串形式的 host 改为 InetAddress 对象类型了（需要 getByName 转换）。在这个构造方法中之所以使用 InetAddress 类主要是因为考虑到在程序中可能需要使用 Socket 类多次连接同一个 IP 或域名，这样使用 InetAddress 类的效率比较高。另外，在使用字符串类型的 host 连接服务器时，可能会发生两个错误，但使用 InetAddress 对象来描述 host，只会发生 IOException 错误，这是因为当你将 IP 或域名传给 InetAddress 时，InetAddress 会自动检查这个 IP 或域名，如果这个 IP 或域名无效，那么 InetAddress 就会抛出 UnknownHostException 错误，而不会由 Socket 类的构造方法抛出。因此，这个构造方法的完整定义是：

```
public Socket(InetAddress inetaddress, int port) throws IOException
```

##### (3) public Socket(String host, int port, boolean stream)

这个构造方法和第一种构造方法差不多，只是多了一个 boolean 类型的 stream 参数。如果这个 stream 为 true，那么这个构造方法和第一种构造方法完全一样。如果 stream 为 false，则使用 UDP 协议建立一

个 UDP 连接(UDP 将在下面的章节详细讨论, 在这里只要知道它和 TCP 最大的区别是 UDP 是面向无连接的, 而 TCP 是面向有连接的), 也许是当初 Sun 的开发人员在编写 Socket 类时还未考虑编写处理 UDP 连接的 DatagramSocket 类, 所以才将建立 UDP 连接的功能加入到 Socket 类中, 不过 Sun 在后来的 JDK 中加入了 DatagramSocket 类, 所以, 这个构造方法就没什么用了, 因此, Sun 将其设为了 Deprecated 标记, 也就是说, 这个构造方法在以后的 JDK 版本中可以会被删除。其于以上原因, 在使用 Java 编写网络程序时, 尽量不要使用这个构造方法来建立 UDP 连接。

#### (4) public Socket(InetAddress inetaddress, int port, boolean flag)

这个构造方法和第三种构造方法的 flag 标记的含义一样, 也是不建议使用的。

下面的代码演示上述 4 种构造方法的使用:

```
package mysocket;

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

public class MoreConnection
{
    private static void closeSocket(Socket socket)
    {
        if (socket != null)
        {
            try
            {
                socket.close();
            }
            catch (Exception e) { }
        }
    }

    public static void main(String[] args)
    {
        Socket socket1 = null, socket2 = null, socket3 = null, socket4 = null;
        try
        {
            // 如果将 www.ptpress.com.cn 改成其它不存在的域名, 将抛出 UnknownHostException 错误
            // 测试 public Socket(String host, int port)
            socket1 = new Socket("www.ptpress.com.cn", 80);
            System.out.println("socket1 连接成功!");
            // 测试 public Socket(InetAddress inetaddress, int port)
            socket2 = new Socket(InetAddress.getByName("www.ptpress.com.cn"), 80);
            System.out.println("socket2 连接成功!");

            // 下面的两种建立连接的方式并不建议使用
            // 测试 public Socket(String host, int port, boolean stream)
            socket3 = new Socket("www.ptpress.com.cn", 80, false);
```

```

        System.out.println("socket3 连接成功!");
        // 测试 public Socket(InetAddress inetaddress, int i, boolean flag)
        socket4 = new Socket(InetAddress.getByName("www.ptpress.com.cn"), 80, false);
        System.out.println("socket4 连接成功!");
    }
    catch (UnknownHostException e)
    {
        System.out.println("UnknownHostException 被抛出!");
    }
    catch (IOException e)
    {
        System.out.println("IOException 被抛出!");
    }
    finally
    {
        closeSocket(socket1);
        closeSocket(socket2);
        closeSocket(socket3);
        closeSocket(socket4);
    }
}
}

```

在上面代码中的最后通过 **finally** 关闭了被打开的 **Socket** 连接，这是一个好习惯。因为只有在将关闭 **Socket** 连接的代码写在 **finally** 里，无论是否出错，都会执行这些代码。但要注意，在关闭 **Socket** 连接之前，必须检查 **Socket** 对象是否为 **null**，这是因为错误很可能在建立连接时发生，这样 **Socket** 对象就没有建立成功，也就用不着关闭了。

## 2. 手动绑定 IP

当本机有多个 IP 时（这些 IP 可能是多块网卡上的，也可能是一块网卡上绑定的多个 IP），在连接服务器时需要由客户端确定需要使用哪个 IP。这样就必须使用 **Socket** 类的另外两个构造方法来处理。下面让我们来看看这两个构造方法是如何来使用特定的 IP 来连接服务器的。

```
public Socket(String host, int port, InetAddress inetaddress, int localPort)
```

这个构造方法的参数分为两部分，第一部分为前两个参数：**host** 和 **port**，它们分别表示要连接的服务器的 IP 和端口号。第二部分为后两个参数：**inetaddress** 和 **localPort**。其中 **inetaddress** 则表示要使用的本地的（同一个子网，同一主机上的多个 IP 并不一定是同一子网）IP，而 **localPort** 则表示要绑定的本地端口号。这个 **localPort** 可以设置为本机的任何未被绑定的端口号：如果将 **localPort** 的值设为 0，java 将在 1024 到 65,535 之间随即选择一个未绑定的端口号。因此，在一般情况下将 **localPort** 设为 0。

```
public Socket(InetAddress inetaddress, int port, InetAddress inetaddress1, int localPort)
```

这个构造方法和第一个构造方法基本相同，只是将第一个参数 **host** 换成了 **inetaddress**。其它的使用方法和第一个构造方法类似。

在下面的代码中将使用这两个构造方法来做一个实验。我们假设有两台计算机：PC1 和 PC2。PC1 和 PC2 各有一块网卡。

PC1 绑定有两个 IP：192.168.18.252 和 200.200.200.200。

PC2 绑定有一个 IP: 200.200.200.4。

PC1 和 PC2 的子网掩码都是 255.255.255.0。而 PC1 的默认网关为: 192.168.28.254。

下面的代码在 PC1 上运行。

```
package mysocket;

import java.net.*;

public class MoreConnection1
{
    public static void main(String[] args)
    {
        try
        {
            InetAddress localAddress1 = InetAddress.getByName("200.200.200.200");
            InetAddress localAddress2 = InetAddress.getByName("192.168.18.252");
            // 如果将 localAddress1 改成 localAddress2, socket1 无法连接成功
            Socket socket1 = new Socket("200.200.200.4", 30, localAddress1, 0);
            System.out.println("socket1 连接成功!");
            Socket socket2 = new Socket("www.ptpress.com.cn", 80, localAddress2, 0);
            System.out.println("socket2 连接成功!");
            // 下面的语句将抛出一个 IOException 错误
            Socket socket3 = new Socket("www.ptpress.com.cn", 80, localAddress1, 0);
            System.out.println("socket3 连接成功!");
            socket1.close();
            socket2.close();
            socket3.close();
        }
        catch (Exception e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

运行上面代码的输出结果如下:

socket1 连接成功!

socket2 连接成功!

Connection timed out: connect

从上面的输出结果可以看出, socket1 和 socket2 已经连接成功, 而 socket3 并未连接成功。从例程可以看出, socket1 在连接时使用 localAddress1 绑定到了 PC1 的 200.200.200.200 上, 而 PC2 的 IP 是 200.200.200.4, 因此, socket1 所使用的 IP 和 PC2 的 IP 在同一个网段, 所以 socket1 可以连接成功。如果将 localAddress1 改成 localAddress2 后, socket1 将无法连接成功。另外两个 Socket 连接 socket2 和 socket3 是通过 Internet 连接 www.ptpress.com.cn。它们所不同的是 socket2 绑定的是 192.168.1

8.252, 而 socket3 绑定的是 200.200.200.200。它们执行的结果是 socket2 可以连接成功, 而 socket3 连接失败。这是因为 socket2 所绑定的 IP 和 PC1 的默认网关 192.168.18.254 在同一个网段, 因此, socket2 可以连接到 Internet。而 socket3 所绑定的 IP 和 PC1 的 IP 不在同一个网段, 因此, socket3 将无法连接到 Internet。

## 二、通过 connect 方法连接服务器

Socket 类不仅可以通过构造方法[\[直接\]](#)连接服务器, 而且还可以建立[\[未连接的\]](#)Socket 对象, 并通过 connect 方法来连接服务器。Socket 类的 connect 方法有两个重载形式:

### 1. public void connect(InetSocketAddress endpoint) throws IOException

Socket 类的 connect 方法与其构造方法在描述服务器信息 (IP 和端口) 上有一些差异。在 connect 方法中并未象构造方法中以字符串形式的 host 或整数形式的 port 作为参数, 而是直接将 IP 和端口封装在了 SocketAddress 类的子类 InetSocketAddress 中。可按如下形式使用这个 connect 方法:

```
Socket socket = new Socket();
socket.connect(new InetSocketAddress(host, port));
```

### 2. public void connect(InetSocketAddress endpoint, int timeout) throws IOException

这个 connect 方法和第一个 connect 类似, 只是多了一个 timeout 参数。这个参数表示连接的超时时间, 单位是毫秒。使用 timeout 设为 0, 则使用默认的超时时间。

在使用 Socket 类的构造方法连接服务器时可以直接通过构造方法绑定本地 IP, 而 connect 方法可以通过 Socket 类的 bind 方法来绑定本地 IP。例程 4-9 演示如何使用 connect 方法和 bind 方法。

```
package mysocket;

import java.net.*;

public class MoreConnection2
{
    public static void main(String[] args)
    {
        try
        {
            Socket socket1 = new Socket();
            Socket socket2 = new Socket();
            Socket socket3 = new Socket();
            socket1.connect(new InetSocketAddress("200.200.200.4", 80));
            socket1.close();
            System.out.println("socket1 连接成功!");
            /*
             * 将 socket2 绑定到 192.168.18.252 将产生一个 IOException 错误
             */
            socket2.bind(new InetSocketAddress("192.168.18.252", 0));
            socket2.bind(new InetSocketAddress("200.200.200.200", 0));
            socket2.connect(new InetSocketAddress("200.200.200.4", 80));

            socket2.close();
            System.out.println("socket2 连接成功!");
        }
    }
}
```



```

        socket3.bind(new InetSocketAddress("192.168.18.252", 0));
        socket3.connect(new InetSocketAddress("200.200.200.4", 80), 2000);
        socket3.close();
        System.out.println("socket3 连接成功!");
    }
    catch (Exception e)
    {
        System.out.println(e.getMessage());
    }
}
}

```

上面的代码的输出结果为：

```

socket1 连接成功!
socket2 连接成功!
Connection timed out: connect

```

在上面代码中的 `socket3` 连接服务器时为其设置了超时时间（2000 毫秒），因此，`socket3` 在非常短的时间就抛出了 `IOException` 错误。

## 15 为什么要使用 `SocketAddress` 来管理网络地址

在使用 `Socket` 来连接服务器时最简单的方式就是直接使用 IP 和端口，但 `Socket` 类中的 `connect` 方法并未提供这种方式，而是使用 `SocketAddress` 类来向 `connect` 方法传递服务器的 IP 和端口。虽然这种方式从表面上看要麻烦一些，但它会给我们带来另外一个好处，那就是网络地址的重用！

所谓网络地址的重用表现在两个方面：

1. 通过建立一个 `SocketAddress` 对象，可以在多次连接同一个服务器时使用这个 `SocketAddress` 对象。
2. 在 `Socket` 类中提供了两个方法：`getRemoteSocketAddress` 和 `getLocalSocketAddress`，通过这两个方法可以得到服务器和本机的网络地址。而且所得到的网络地址在有效地的 `Socket` 对象关闭后仍然可以使用。下面是这两个方法的声明：

```

public SocketAddress getRemoteSocketAddress()
public SocketAddress getLocalSocketAddress()

```

不管在使用 `Socket` 类连接服务器时是直接使用 IP 和端口，还是使用 `SocketAddress`，这两个方法都返回 `SocketAddress` 形式的网络地址。当 `Socket` 对象未连接时这两个方法返回 `null`，但**要注意的是只有在 `Socket` 对象从未连接时这两个方法才返回 `null`**，而当已经连接成功过的 `Socket` 对象关闭后仍可使用这两个方法得到相应的网络地址。

虽然上面曾多次提到 `SocketAddress`，但 `SocketAddress` 只是个抽象类，它除了有一个默认的构造方法外，其它的方法都是 `abstract` 的，因此，我们必须使用 `SocketAddress` 的子类来建立 `SocketAddress` 对象。在 `JDK1.4` 中为我们提供了 IP 网络地址的实现类：`java.net.InetSocketAddress`。这个类是从 `SocketAddress` 继承的，我们可以通过如下的方法来建立 `SocketAddress` 对象。

```
SocketAddress socketAddress = new InetSocketAddress(host, ip);
```

下面的代码演示了如何通过 SocketAddress 来共享网络地址：

```
package mynet;

import java.net.*;

public class MySocketAddress
{
    public static void main(String[] args)
    {
        try
        {
            Socket socket1 = new Socket("www.ptpress.com.cn", 80);
            SocketAddress socketAddress = socket1.getRemoteSocketAddress();
            socket1.close();
            Socket socket2 = new Socket();
            // socket2.bind(new InetSocketAddress("192.168.18.252", 0));
            socket2.connect(socketAddress);
            socket2.close();
            InetSocketAddress inetSocketAddress1 = (InetSocketAddress) socketAddress;
            System.out.println("服务器域名："
                + inetSocketAddress1.getAddress().getHostName());
            System.out.println("服务器 IP："
                + inetSocketAddress1.getAddress().getHostAddress());
            System.out.println("服务器端口：" + inetSocketAddress1.getPort());
            InetSocketAddress inetSocketAddress2 = (InetSocketAddress) socket2
                .getLocalSocketAddress();
            System.out.println("本地 IP："
                + inetSocketAddress2.getAddress().getLocalHost()
                .getHostAddress());
            System.out.println("本地端口：" + inetSocketAddress2.getPort());
        }
        catch (Exception e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

输出结果：

服务器域名: www.ptpress.com.cn

服务器 IP: 219.238.168.74

服务器端口: 80

本地 IP: 192.168.18.253

本地端口:4250

如果多次运行例程 4-10 后,本地端口的值可能在每次都不一样。这是因为在 `socket2` 在连接时并未使用 `bind` 来绑定本地的端口,而这个本地端口是由系统在 1024 至 65,535 中随机选取的,因此,在每次运行程序时这个本地端口不一定相同。

## 16 客户端套接字 (Socket) 的超时

客户端套接字的超时(timeout)就是指在客户端通过 `Socket` 和服务器进行通讯的过程中,由于网络延迟,网络阻塞等原因,造成服务器并未及时响应客户端的一种现象。在一段时间后,客户端由于未收到服务端的响应而抛出一个超时错误;其中客户端所等待的时间就是超时时间。

由于生产超时错误的一端都是被动端:也就是说,这一端是在接收数据,而不是发送数据,对于客户端 `Socket` 来说,只有两个地方是在接收数据:一个是在连接服务器时;另一个是在连接服务器成功后,接收服务器发过来的数据时。因此,客户端超时也分为两种类型:连接超时和读取数据超时。

### 一、连接超时

这种超时在前面的例子中已经使用过。在 `Socket` 类中只有通过 `connect` 方法的第二个参数才能指定连接超时的时间。由于使用 `connect` 方法连接服务器必须要指定 IP 和端口,因此,无效的 IP 或端口必将引发连接超时错误。

### 二、读取数据超时

在连接服务器成功后,`Socket` 所做的最重要的两件事就是“接收数据”和“发送数据”;而在接收数据时可能因为网络延迟、网络阻塞等原因,客户端一直处于等待状态;而客户端在等待一段时间后,如果服务器还没有发送数据到客户端,那么客户端 `Socket` 将会抛出一个超时错误。

我们可以通过 `Socket` 类的 `setSoTimeout` 方法来设置读取数据超时的时间;时间的单位是毫秒。这个方法必须在读取数据之前调用才会生效。如果将超时时间设为 0,则不使用超时时间;也就是说,客户端什么时候和服务端断开,将完全取决于服务端程序的超时设置。如下面的语句将读取数据超时时间设为 5 秒。

```
Socket socket = new Socket();
socket.setSoTimeout(5000);
socket.connect(... ..);
socket.getInputStream().read();
```

要注意的是不要将设置连接超时和读取数据超时设置得太小,如果值太小,如 100,可能会造成服务器的数据还没来得及发过来,客户端就抛出超时错误的现象。下面的代码给出了一个用于测试连接超时和读取数据超时的例子。

```
package mynet;

import java.net.*;

public class SocketTimeout
{
    public static void main(String[] args)
    {
        long time1 = 0, time2 = 0;
        Socket socket = new Socket();
        try
        {
```

```

    if (args.length < 4)
    {
        System.out.println("参数错误!");
        return;
    }

    time1 = System.currentTimeMillis();
    socket.connect(new InetSocketAddress(args[0], Integer
        .parseInt(args[1])), Integer.parseInt(args[2])); //设置连接超时
    socket.setSoTimeout(Integer.parseInt(args[3])); //设置读取数据超时
    time1 = System.currentTimeMillis(); //重置一次
    socket.getInputStream().read();
}
catch (SocketTimeoutException e)
{
    if (!socket.isClosed() && socket.isConnected())
        System.out.println("读取数据超时!");
    else // 看不懂...
        System.out.println("连接超时");
}
catch (Exception e)
{
    System.out.println(e.getMessage());
}
finally
{
    time2 = System.currentTimeMillis();
    System.out.println(time2 - time1);
}
}
}

```

SocketTimeout 类的主方法需要 4 个参数：IP（域名）、端口、连接超时、读取数据超时。下面让我们用一组数据来测试这个例子。

#### 测试 1：无效 IP 引发的超时错误

执行如下命令：

```
java mynet.SocketTimeout 192.168.18.24 80 3000 5000
```

运行结果：

连接超时

3045

注意：192.168.18.24 是一个无效的 IP；如果这个 IP 在网络环境中存在，请换其它的无效的 IP。**在这个测试用例中不能将无效的 IP 换成无效的域名！**这是因为如果使用了域名来连接服务器，Java 会先通过 DNS 将域名映射成相应的 IP；如果这个域名是无效的，在映射的过程中就会出错；因此，程序也就不会执行连接服务器操作，自然也就不会抛出“连接超时”错误了。

#### 测试 2：无效端口引发的超时错误

执行如下命令：

```
java mynet.SocketTimeout www.ptpress.com.cn 8888 3000 5000
```

运行结果：

连接超时

3075

**测试 3：读取数据超时错误**

执行如下命令：

```
java mynet.SocketTimeout www.ptpress.com.cn 80 3000 5000
```

运行结果：

读取数据超时！

5008

**测试 4：将读取数据超时设为 0 的效果**

执行如下命令：

```
java mynet.SocketTimeout www.ptpress.com.cn 80 3000 0
```

运行结果：

Connection reset

131519

从前 3 个测试的输出结果不难看出，每个测试用例都将连接超时和读取数据超时分别设为 3000 和 5000 毫秒；而它们的运行结果并不是 3000 和 5000 毫秒，而是在所设定的超时时间的左右摇摆；这主要是因为系统所输出的时间并不都是超时时间；如有一些时间是 Java 处理错误、向控制台输出信息的时间。另外，由于系统计时的误差，也会影响到超时时间的准确性。但不管怎样，超时时间总会在所设定的时间周围摇摆。

对于测试 4，将读取数据超时设为 0 后，SocketTimeout 类经过了 2 分多钟(131519 毫秒)才抛出 Connection reset 错误。[这个抛出错误的时间和服务端程序的超时设置有关](#)；也就是这个错误是由于服务端程序主动将客户端网络连接断开而产生的。

## 17 Socket 类的 getter 和 setter 方法 A

在 Java 类中，getter 和 setter 方法占了很大的比重。由于 Java 中没有定义属性的关键字。因此，getter 和 setter 方法用于获得和设置 Java 类的属性值，例如 getName 和 setName 方法用于设置 name 属性的值。如果某个属性只有 getter 方法，那这个属性是只读的；如果只有 setter 方法，那么这个属性是只写的。在 Socket 类中也有很多这样的属性来获得和 Socket 相关的信息，以及对 Socket 对象的状态进行设置。

### 一、用于获得信息的 getter 方法

我们可以从 Socket 对象中获得 3 种信息。

#### 1. 服务器信息

对于客户端来说，服务器的信息只有两个：IP 和端口。Socket 类为我们提供了 3 个方法来得到这两个信息。

##### (1) public InetAddress getInetAddress()

通过 Socket 的这个方法返回一个 InetAddress 对象。再通过这个对象的 get 方法，就可以得到服务器的 IP、域名等信息。

```
Socket socket = new Socket("www.ptpress.com.cn", 80);
System.out.println(socket.getInetAddress().getHostAddress());
System.out.println(socket.getInetAddress().getHostName());
```

## (2) public int getPort()

这个方法可以以整数形式获得服务器的端口号。

```
Socket socket = new Socket("www.ptpress.com.cn", 80);
System.out.println(socket.getInetAddress().getPort());
```

## (3) public SocketAddress getRemoteSocketAddress()

这个方法返回结果是将 `getInetAddress` 和 `getPort` 方法结合在了一起，利用这个方法可以同时得到服务器的 IP 和端口号。但这个方法返回了一个 `SocketAddress` 对象，这个对象只能作为 `connect` 方法的参数用于连接服务器；而要想获得服务器的 IP 和端口号，必须得将 `SocketAddress` 转换为它的子类 `InetSocketAddress`。

```
Socket socket = new Socket("www.ptpress.com.cn", 80);
System.out.println(((InetSocketAddress)socket.getRemoteSocketAddress()).getHostName());
System.out.println(((InetSocketAddress)socket.getRemoteSocketAddress()).getPort());
```

注意：以上 3 个方法都可以在调用 `Socket` 对象关闭后调用。它们所获得的信息在 `Socket` 对象关闭后仍然有效。如果直接使用 IP 连接服务器时，`getHostName` 和 `getHostAddress` 的返回值是一样的：都是服务器的 IP。

## 2. 本机信息

与服务器信息一样，本机信息也有两个：本地 IP 和绑定的本地端口号。这些信息也可以通过 3 个方法来获得。

### (1) public InetAddress getLocalAddress()

这个方法返回了本机的 `InetAddress` 对象。通过这个方法可以得到本机的 IP 和机器名。当本机绑定了多个 IP 时，`Socket` 对象使用哪一个 IP 连接服务器，就返回哪个 IP。如果本机使用 ADSL 上网，并且通过 `Socket` 对象连接到 Internet 上的某一个 IP 或域名上（如 `www.ptpress.com.cn`），则 `getLocalAddress` 将返回“ADSL 连接”所临时绑定的 IP；因此，我们可以通过 `getLocalAddress` 得到 ADSL 的临时 IP。

```
Socket socket = new Socket();
socket.connect(new InetSocketAddress("www.ptpress.com.cn", 80));
System.out.println(socket.getLocalAddress().getHostAddress());
System.out.println(socket.getLocalAddress().getHostName());
```

```
Socket socket = new Socket();
// 如果使用下面的 bind 方法进行端口绑定的话，getLocalPort 方法将返回 100
// socket.bind(new InetSocketAddress("127.0.0.1", 100));
socket.connect(new InetSocketAddress("www.ptpress.com.cn", 80));
System.out.println(socket.getLocalPort());
```

### (3) public SocketAddress getLocalSocketAddress()

这个方法和 `getRemoteSocketAddress` 方法类似，也是同时得到了本地 IP 和 `Socket` 对象所绑定的端口号。如果要得到本地 IP 和端口号，必须将这个方法的返回值转换为 `InetSocketAddress` 对象。

```
Socket socket = new Socket("www.ptpress.com.cn", 80);
System.out.println(((InetSocketAddress)socket.getLocalSocketAddress()).getHostName());
System.out.println(((InetSocketAddress)socket.getLocalSocketAddress()).getPort());
```

## 3. 用于传输数据的输入、输出流

输入、输出流在前面的章节已经被多次用到。在这里让我们来简单回顾一下。

### (1) public InputStream getInputStream() throws IOException



用于获得从服务器读取数据的输入流。它所得到的流是最原始的源。为了操作更方便，我们经常使用 `InputStreamReader` 和 `BufferedReader` 来读取从服务器传过来的字符串数据。

```
Socket socket = new Socket("www.ptpress.com.cn", 80);
InputStream inputStream = socket.getInputStream();
InputStreamReader inputStreamReader = new InputStreamReader(inputStream);
BufferedReader bufferedReader = new BufferedReader(inputStreamReader);
System.out.println(bufferedReader.readLine());
```

## (2) `public OutputStream getOutputStream() throws IOException`

用于获得向服务器发送数据的输出流。输出流可以通过 `OutputStreamWriter` 和 `BufferedWriter` 向服务器写入字符串数据。

```
Socket socket = new Socket("www.ptpress.com.cn", 80);
OutputStream outputStream = socket.getOutputStream();
OutputStreamWriter outputStreamWriter = new OutputStreamWriter(outputStream);
BufferedWriter bufferedWriter = new BufferedWriter(outputStreamWriter);
bufferedWriter.write("你好");
bufferedWriter.flush();
```

注意：在使用 `OutputStream` 的 `write` 方法输出数据后，必须使用 `flush` 方法刷新输出缓冲区，以便将输出缓冲区中的数据发送出去。如果要输出字符串，使用 `OutputStreamWriter` 和 `BufferedWriter` 都可以；它们的 `write` 方法都可以直接使用字符串作为参数来输出数据。而这一点与相应的 `InputStreamReader` 和 `BufferedReader` 不同；它们中只有 `BufferedReader` 有 `readLine` 方法，因此，必须使用 `BufferedReader` 才能直接读取字符串数据。

## 18 Socket 类的 getter 和 setter 方法

### 二、用于获得和设置 Socket 选项的 getter 和 setter 方法

Socket 选择可以指定 Socket 类发送和接受数据的方式。在 JDK1.4 中共有 8 个 Socket 选择可以设置。这 8 个选项都定义在 `java.net.SocketOptions` 接口中。定义如下：

```
public final static int TCP_NODELAY = 0x0001;    //非延迟
public final static int SO_REUSEADDR = 0x04;     // 重用地址
public final static int SO_LINGER = 0x0080;      // 设置返回时间
public final static int SO_TIMEOUT = 0x1006;     // 读取数据超时
public final static int SO_SNDBUF = 0x1001;      //发送缓存
public final static int SO_RCVBUF = 0x1002;      //接受缓存
public final static int SO_KEEPALIVE = 0x0008;   //检测活性
public final static int SO_OOBINLINE = 0x1003;   //单字节数据
```

有趣的是，这 8 个选项除了第一个没在 `SO` 前缀外，其他 7 个选项都以 `SO` 作为前缀。其实这个 `SO` 就是 **Socket Option** 的缩写；因此，在 Java 中约定所有以 `SO` 为前缀的常量都表示 Socket 选项。在 `Socket` 类中为每一个选项提供了一对 `get` 和 `set` 方法，分别用来获得和设置这些选项。

#### 1. `TCP_NODELAY`

```
public boolean getTcpNoDelay() throws SocketException
public void setTcpNoDelay(boolean on) throws SocketException
```



在默认情况下，客户端向服务器发送数据时，会根据数据包的大小决定是否立即发送。当数据包中的数据很少时，如只有 1 个字节，而数据包的头却有几十个字节（IP 头+TCP 头）时，系统会在发送之前先将较小的包合并到较大的包后，一起将数据发送出去。在发送下一个数据包时，系统会等待服务器对前一个数据包的响应，当收到服务器的响应后，再发送下一个数据包，这就是所谓的 Nagle 算法；在默认情况下，Nagle 算法是开启的。

这种算法虽然可以有效地改善网络传输的效率，但对于网络速度比较慢，而且对实现性的要求比较高的情况下（如游戏、Telnet 等），使用这种方式传输数据会使得客户端有明显的停顿现象。因此，最好的解决方案就是需要 Nagle 算法时就使用它，不需要时就关闭它。而使用 `setTcpToDelay` 正好可以满足这个需求。当使用 `setTcpNoDelay(true)` 将 Nagle 算法关闭后，客户端每发送一次数据，无论数据包的大小都会将这些数据发送出去。

## 2. SO\_REUSEADDR

```
public boolean getReuseAddress() throws SocketException
```

```
public void setReuseAddress(boolean on) throws SocketException
```

通过这个选项，可以使多个 `Socket` 对象绑定在同一个端口上。其实这样做并没有多大意义，但当使用 `close` 方法关闭 `Socket` 连接后，`Socket` 对象所绑定的端口并不一定马上释放，系统有时在 `Socket` 连接关闭才会再确认一下是否有因为延迟而未到达的数据包，这完全是在底层处理的，也就是说对用户是透明的。因此，在使用 `Socket` 类时完全不会感觉到。

这种处理机制对于随机绑定端口的 `Socket` 对象没有什么影响，但对于绑定在固定端口的 `Socket` 对象就可能会抛出“Address already in use: JVM\_Bind”例外。因此，使用这个选项可以避免个例外的发生。

```
package mynet;
```

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

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

```
public class Test
```

```
{
```

```
    public static void main(String[] args)
```

```
    {
```

```
        Socket socket1 = new Socket();
```

```
        Socket socket2 = new Socket();
```

```
        try
```

```
        {
```

```
            socket1.setReuseAddress(true);
```

```
            socket1.bind(new InetSocketAddress("127.0.0.1", 88));
```

```
            System.out.println("socket1.getReuseAddress():"
```

```
                + socket1.getReuseAddress());
```

```
            socket2.bind(new InetSocketAddress("127.0.0.1", 88));
```

```
        }
```

```
        catch (Exception e)
```

```
        {
```

```
            System.out.println("error:" + e.getMessage());
```

```
            try
```

```
            {
```

```
                socket2.setReuseAddress(true);
```

```

        socket2.bind(new InetSocketAddress("127.0.0.1", 88));
        System.out.println("socket2.getReuseAddress():"
            + socket2.getReuseAddress());
        System.out.println("端口 88 第二次绑定成功!");
    }
    catch (Exception e1)
    {
        System.out.println(e.getMessage());
    }
}
}
}

```

上面的代码的运行结果如下：

```

socket1.getReuseAddress():true
error:Address already in use: JVM_Bind
socket2.getReuseAddress():true
端口 88 第二次绑定成功!

```

使用 `SO_REUSEADDR` 选项时有两点需要注意：

1. 必须在调用 `bind` 方法之前使用 `setReuseAddress` 方法来打开 `SO_REUSEADDR` 选项。因此，要想使用 `SO_REUSEADDR` 选项，就不能通过 `Socket` 类的构造方法来绑定端口。
2. 必须将绑定同一个端口的所有的 `Socket` 对象的 `SO_REUSEADDR` 选项都打开才能起作用。如在例程 4-12 中，`socket1` 和 `socket2` 都使用了 `setReuseAddress` 方法打开了各自的 `SO_REUSEADDR` 选项。

### 3. `SO_LINGER`

```
public int getSoLinger() throws SocketException
```

```
public void setSoLinger(boolean on, int linger) throws SocketException
```

这个 `Socket` 选项可以影响 `close` 方法的行为。在默认情况下，当调用 `close` 方法后，将立即返回；如果这时仍然有未被送出的数据包，那么这些数据包将被丢弃。如果将 `linger` 参数设为一个正整数 `n` 时(`n` 的值最大是 65,535)，在调用 `close` 方法后，将最多被阻塞 `n` 秒。在这 `n` 秒内，系统将尽量将未送出的数据包发送出去；如果超过了 `n` 秒，如果还有未发送的数据包，这些数据包将全部被丢弃，届时 `close` 方法会立即返回。如果将 `linger` 设为 0，和关闭 `SO_LINGER` 选项的作用是一样的。

如果底层的 `Socket` 实现不支持 `SO_LINGER` 都会抛出 `SocketException` 例外。当给 `linger` 参数传递负数值时，`setSoLinger` 还会抛出一个 `IllegalArgumentException`。可以通过 `getSoLinger` 方法得到延迟关闭的时间，如果返回 -1，则表明 `SO_LINGER` 是关闭的。例如，下面的代码将延迟关闭的时间设为 1 分钟：

```
if(socket.getSoLinger() == -1) socket.setSoLinger(true, 60);
```

### 4. `SO_TIMEOUT`

```
public int getSoTimeout() throws SocketException
```

```
public void setSoTimeout(int timeout) throws SocketException
```

这个 `Socket` 选项在前面已经讨论过。可以通过这个选项来设置读取数据超时。当输入流的 `read` 方法被阻塞时，如果设置 `timeout` (`timeout` 的单位是毫秒)，那么系统在等待了 `timeout` 毫秒后会抛出一个 `InterruptedIOException` 例外。在抛出例外后，输入流并未关闭，你可以继续通过 `read` 方法读取数据。

如果将 `timeout` 设为 0，就意味着 `read` 将会无限等待下去，直到服务端程序关闭这个 `Socket`。这也是 `timeout` 的默认值。如下面的语句将读取数据超时设为 30 秒：

```
socket1.setSoTimeout(30 * 1000);
```

当底层的 Socket 实现不支持 SO\_TIMEOUT 选项时，这两个方法将抛出 SocketException 例外。不能将 timeout 设为负数，否则 setSoTimeout 方法将抛出 IllegalArgumentException 例外。

## 5. SO\_SNDBUF

```
public int getSendBufferSize() throws SocketException
public void setSendBufferSize(int size) throws SocketException
```

在默认情况下，输出流的发送缓冲区是 8096 个字节（8K）。这个值是 Java 所建议的输出缓冲区的大小。如果这个默认值不能满足要求，可以用 setSendBufferSize 方法来重新设置缓冲区的大小。将输出缓冲区设得太小，否则会导致传输数据过于频繁，从而降低网络传输的效率。

如果底层的 Socket 实现不支持 SO\_SNDBUF 选项，这两个方法将会抛出 SocketException。必须将 size 设为正整数，否则 setSendBufferedSize 方法将抛出 IllegalArgumentException。

## 6. SO\_RCVBUF

```
public int getReceiveBufferSize() throws SocketException
public void setReceiveBufferSize(int size) throws SocketException
```

在默认情况下，输入流的接收缓冲区是 8096 个字节（8K）。这个值是 Java 所建议的输入缓冲区的大小。如果这个默认值不能满足要求，可以用 setReceiveBufferSize 方法来重新设置缓冲区的大小。不要将输入缓冲区设得太小，否则会导致传输数据过于频繁，从而降低网络传输的效率。

如果底层的 Socket 实现不支持 SO\_RCVBUF 选项，这两个方法将会抛出 SocketException。必须将 size 设为正整数，否则 setReceiveBufferSize 方法将抛出 IllegalArgumentException。

## 7. SO\_KEEPALIVE

```
public boolean getKeepAlive() throws SocketException
public void setKeepAlive(boolean on) throws SocketException
```

如果将这个 Socket 选项打开，客户端 Socket 每隔一段时间（大约两个小时）就会利用空闲的连接向服务器发送一个数据包。这个数据包并没有其它的作用，只是为了检测一下服务器是否仍处于活动状态。如果服务器未响应这个数据包，在大约 11 分钟后，客户端 Socket 再发送一个数据包，如果在 12 分钟内，服务器还没响应，那么客户端 Socket 将关闭。如果将 Socket 选项关闭，客户端 Socket 在服务器无效的情况下可能会长时间不会关闭。SO\_KEEPALIVE 选项在默认情况下是关闭的，可以使用如下的语句将这个 SO\_KEEPALIVE 选项打开：

```
socket1.setKeepAlive(true);
```

## 8. SO\_OOBINLINE

```
public boolean getOOBInline() throws SocketException
public void setOOBInline(boolean on) throws SocketException
```

如果这个 Socket 选项打开，可以通过 Socket 类的 sendUrgentData 方法向服务器发送一个单字节的数据。这个单字节数据并不经过输出缓冲区，而是立即发出。虽然在客户端并不是使用 OutputStream 向服务器发送数据，但在服务端程序中这个单字节的数据是和其它的普通数据混在一起的。因此，在服务端程序中并不知道由客户端发过来的数据是由 OutputStream 还是由 sendUrgentData 发过来的。下面是 sendUrgentData 方法的声明：

```
public void sendUrgentData(int data) throws IOException
```

虽然 sendUrgentData 的参数 data 是 int 类型，但只有这个 int 类型的低字节被发送，其它的三个字节被忽略。下面的代码演示了如何使用 SO\_OOBINLINE 选项来发送单字节数据。

```
package mynet;
```

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

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

```

class Server
{
    public static void main(String[] args) throws Exception
    {
        ServerSocket serverSocket = new ServerSocket(1234);
        System.out.println("服务器已经启动，端口号：1234");
        while (true)
        {
            Socket socket = serverSocket.accept();
            socket.setOOBInline(true);
            InputStream in = socket.getInputStream();
            InputStreamReader inReader = new InputStreamReader(in);
            BufferedReader bReader = new BufferedReader(inReader);
            System.out.println(bReader.readLine());
            System.out.println(bReader.readLine());
            socket.close();
        }
    }
}

public class Client
{
    public static void main(String[] args) throws Exception
    {
        Socket socket = new Socket("127.0.0.1", 1234);
        socket.setOOBInline(true);
        OutputStream out = socket.getOutputStream();
        OutputStreamWriter outWriter = new OutputStreamWriter(out);
        outWriter.write(67); // 向服务器发送字符"C"
        outWriter.write("hello world\r\n");
        socket.sendUrgentData(65); // 向服务器发送字符"A"
        socket.sendUrgentData(322); // 向服务器发送字符"B"
        outWriter.flush(); // 显示目前缓冲区内容
        socket.sendUrgentData(214); // 向服务器发送汉字"中"
        socket.sendUrgentData(208);
        socket.sendUrgentData(185); // 向服务器发送汉字"国"
        socket.sendUrgentData(250);
        socket.close();
    }
}

```

由于运行上面的代码需要一个服务器类，因此，在此加一个类名为 `Server` 的服务器类，关于服务端套接字的使用方法将会在后面的文章中详细讨论。在类 `Server` 类中只使用了 `ServerSocket` 类的 `accept` 方法接收客户端的请求。并从客户端传来的数据中读取两行字符串，并显示在控制台上。

## 测试

由于本例使用了 127.0.0.1，因 Server 和 Client 类必须在同一台机器上运行。

运行 Server

```
java mynet.Server
```

运行 Client

```
java mynet.Client
```

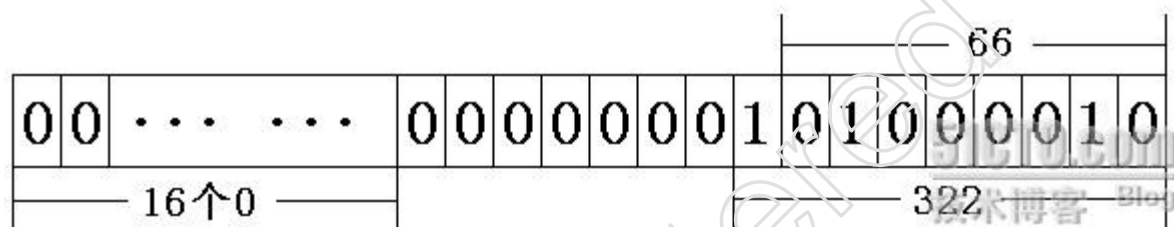
在服务端控制台的输出结果

服务器已经启动，端口号：1234

ABChello world

中国

在 Client 类中使用了 `sendUrgentData` 方法向服务器发送了字符 'A'(65) 和 'B'(66)。但发送 'B' 时实际发送的是 322，由于 `sendUrgentData` 只发送整型数的低字节。因此，实际发送的是 66。十进制整型 322 的二进制形式如图 1 所示。



在 Client 类中使用 `flush` 将缓冲区中的数据发送到服务器。我们可以从输出结果发现一个问题，在 Client 类中先后向服务器发送了 'C'、"hello world\r\n"、'A'、'B'。而在服务端程序的控制台上显示的却是 ABChello world。这种现象说明使用 `sendUrgentData` 方法发送数据后，系统会立即将这些数据发送出去；而使用 `write` 发送数据，必须要使用 `flush` 方法才会真正发送数据。

在 Client 类中向服务器发送 "中国" 字符串。由于 "中" 是由 214 和 208 两个字节组成的；而 "国" 是由 185 和 250 两个字节组成的；因此，可分别发送这四个字节来传送 "中国" 字符串。

**注意：**在使用 `setOOBInline` 方法打开 `SO_OOBINLINE` 选项时要注意是 **必须在客户端和服务端程序同时使用 `setOOBInline` 方法打开这个选项**，否则无法命名用 `sendUrgentData` 来发送数据。

## 19 Socket 异常概述

在 Socket 类中有很多方法在声明时使用 `throws` 抛出了一些异常，这些异常都是 `IOException` 的子类。在 Socket 类的方法中抛出最多的就是 `SocketException`，其余还有七个异常可供 Socket 类的方法抛出。这些异常的继承关系如图 1 所示。**灰色背景框所描述的例外就是 Socket 类的方法可能抛出的异常。**

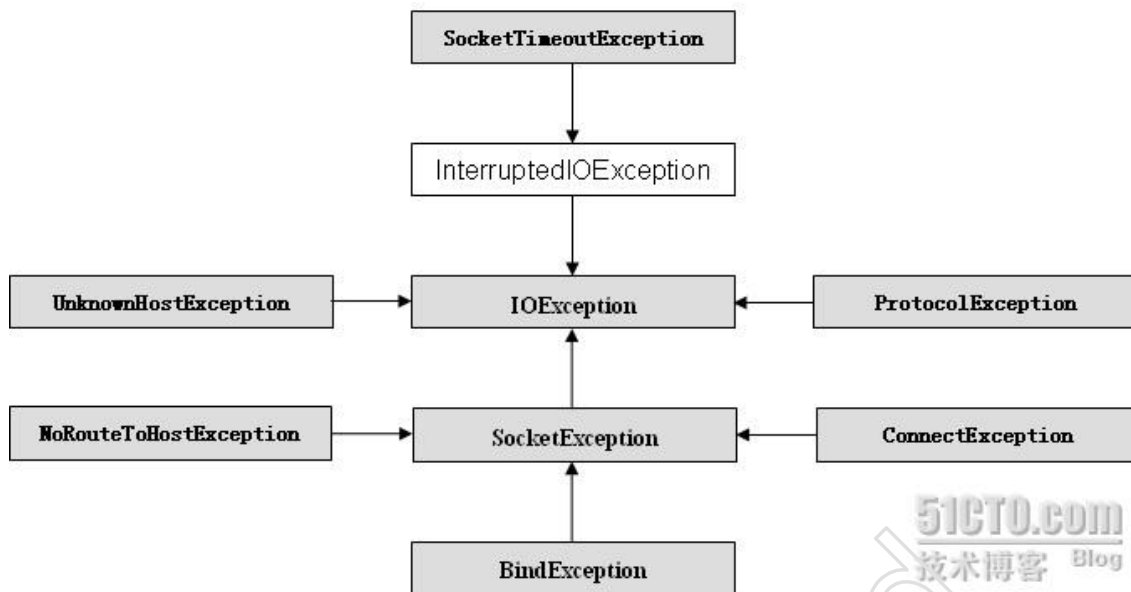


图 1 异常类继承关系图

- `public class IOException extends Exception`  
 这个异常是所有在 Socket 类的方法中抛出的异常的父类。因此，在使用 Socket 类时只要捕捉 (catch) 这个异常就可以了；当然，为了同时捕捉其它类中的方法的异常，也可以直接捕捉 Exception。
- `public class SocketException extends IOException`  
 这个异常在 Socket 类的方法中使用得最频繁。它也代表了所有和网络有关的异常。但如果要想知道具体发生的哪一类的异常，就需要捕捉更具体的异常了。
- `public class ConnectException extends SocketException`  
 ConnectException 异常通常发生在由于服务器忙而未响应或是服务器相应的监听端口未打开。如下面的语句将抛出一个 ConnectException 异常。  

```
Socket socket = new Socket("www.ptpress.com.cn", 1234);
```
- `public class BindException extends SocketException`  
 这个异常在多个 Socket 或 ServerSocket 对象绑定在同一个端口，而且未打开 SO\_REUSEADDR 选项时发生。如下面的四条语句将抛出一个 BindException 异常：  

```
Socket socket1 = new Socket();
Socket socket2 = new Socket();
socket1.bind(new InetSocketAddress("127.0.0.1", 1234));
socket2.bind(new InetSocketAddress("127.0.0.1", 1234));
```
- `public class NoRouteToHostException extends SocketException`  
 这个异常在遇到防火墙禁止或是路由无法找到主机的情况下发生。
- `public class UnknownHostException extends IOException`  
 这个异常在域名不正确时被抛出。如下面的语句将抛出一个 UnknownHostException 异常：  

```
Socket socket1 = new Socket("www.ptpress123.com.cn", 80);
```
- `public class ProtocolException extends IOException`  
 这个异常并不经常被抛出。如果由于不明的原因，TCP/IP 的数据包被破坏了，这时将抛出 ProtocolException 异常。
- `public class SocketTimeoutException extends InterruptedIOException`



如果在[连接超时](#)和[读取数据超时](#)时间过后，服务器仍然未响应，connect 或 read 方法将抛出 SocketTimeoutException 异常。

## 20 HTTP 协议简介

### 一、什么是 HTTP 协议

HTTP 协议是一种应用层协议，HTTP 是 HyperText Transfer Protocol (超文本传输协议) 的英文缩写。HTTP 可以通过传输层的 TCP 协议在客户端和服务端之间传输数据。HTTP 协议主要用于 Web 浏览器和 Web 服务器之间的数据交换。我们在使用 IE 或 Firefox 浏览网页或下载 Web 资源时，通过在地址栏中输入 <http://host:port/path>，开头的 4 个字母 http 就相当于通知浏览器使用 HTTP 协议来和 host 所确定的服务器进行通讯。

HTTP 协议诞生于上世纪 90 年代初；第一个被广泛使用的版本是 HTTP0.9。这个最初的版本非常简陋，它只向服务器发送一个非常简单的请求，而服务器也会返回一个很简单的响应以及相应的 HTML 文本。在随后的 HTTP1.0 中，增加了很多在 HTTP0.9 中没有的特性，如增加了资源重定位，大量的状态响应码等。在最新的 HTTP1.1 中，对 HTTP1.0 做了更进一步的改进，除了增加了一些请求方法外，最大的改进就是可以使 HTTP 保持连接状态。这对于一些频繁传输数据的应用是非常有益的。由于 HTTP 协议已经达到了它的目标，因此，负责制定规范的 W3C 已经停止了对 HTTP 的改进（和 HTTP 相关的协议或扩展并未停止），所以，HTTP1.1 将是 HTTP 协议的最后一个版本。

无论你是从事[网络程序开发](#)，还是[Web 开发](#)，或是[网站的维护](#)人员；都必须对 HTTP 协议有一个比较深入的了解。因此，HTTP 协议不仅是 Internet 上应用最为广泛的协议，也是应用协议家族中比较简单的一种入门级协议；而且所有的 Web 服务器无一例外地都支持 HTTP 协议。这也充分地说明，对于那些开发网络程序，尤其是开发各种类型的 Web 服务器的开发人员，透彻地掌握 HTTP 协议将对你所开发的基于 HTTP 协议的系统产生直接的影响。

由于本文的目的并不是讲解 HTTP 协议，因此，只讨论了 HTTP 协议的主要部分，如果读者对 HTTP 协议感兴趣，并想深入了解 HTTP 协议，请查看 RFC2616 或通过 [www.w3c.org](http://www.w3c.org) 来了解 HTTP 协议的详情。

### 二、HTTP 的工作方式

HTTP 协议采用了请求/响应的工作方式。基于 HTTP1.0 协议的客户端在每次向服务器发出请求后，服务器就会向客户端返回响应消息（包括请求是否正确以及所请求的数据），在确认客户端已经收到响应消息后，服务端就会关闭网络连接（其实是关闭 TCP 连接）。在这个数据传输过程中，并不保存任何历史信息 and 状态信息，因此，HTTP 协议也被认为是无状态的协议，图 1 描绘了 HTTP1.0 协议的通讯过程。

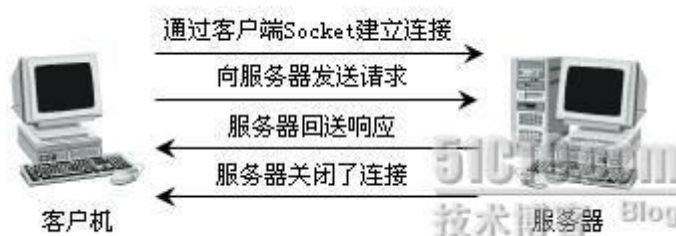


图 1 HTTP1.0 协议的通讯过程

在 HTTP1.0 协议中，当 Web 浏览器发出请求时，就意味着一个请求/响应会话已经开始。在请求、响应结束后，服务器就会立刻关闭这个连接。这种会话方式虽然简便，但它会带来另外一个问题。如果客户端浏览器访



问的某个或其他类型的 Web 页中包含有其他的 Web 资源，如 JavaScript 文件、图像文件、CSS 文件等；当浏览器每遇到这样一个 Web 资源，就会一一建立 HTTP 会话。如果这样的资源很多的话，就会加重服务器的负担，同时也会影响客户端浏览器加载 HTML 等 Web 资源的效率。

在对上述的缺陷进行改进和完善后，HTTP1.1 协议进入了我们的视线。HTTP1.1 和 HTTP1.0 相比较而言，最大的区别就是增加了持久连接支持。当客户端使用 HTTP1.1 协议连接到服务器后，服务器就将关闭客户端连接的主动权交还给客户端；也就是说，在客户端向服务器发送一个请求并接收以一个响应后，只要不调用 Socket 类的 close 方法关闭网络连接，就可以继续向服务器发送 HTTP 请求。当 HTML 中含有其他的 Web 资源时，浏览器就可以使用同一个网络连接向下载这些资源，这样就可以大大减轻服务器的压力。



图 2 HTTP1.1 协议的通讯过程

HTTP1.1 除了支持持久连接外，还将 HTTP1.0 的请求方法从原来的三个 (GET、POST 和 HEAD) 扩展到了八个 (OPTIONS、GET、HEAD、POST、PUT、DELETE、TRACE 和 CONNECT)。而且还增加了很多请求和响应字段，如上述的持久连接的字段 Connection。这个字段有两个值，Close 和 Keep-Alive。如果使用 Connection:Close，则关闭 HTTP1.1 的持久连接的功能，要打开 HTTP1.1 的持久连接的功能，必须使用 Connection:Keep-Alive，或者不加 Connection 字段（因为 HTTP1.1 在默认情况下就是持久连接的）。除了这些，还提供了身份认证、状态管理和缓存 (Cache) 等相关的请求头和响应头。

## 21 HTTP 消息的格式

当用户在浏览器中输入一个基于 HTTP 协议的 URL 时 (以 http://开头的 URL)，就相当于通知器按着这个 URL 组织生成一个 HTTP 请求，并交过个请求发送到服务器；同时，等待服务器的响应。无论是请求还是响应，都统称为 HTTP 消息。

HTTP 协议分为两部分：

1. HTTP 请求消息。
2. HTTP 响应消息。

### 一、HTTP 请求消息

HTTP 请求消息分为请求消息头以及请求实体内容两部分。请求消息头的第一行必须由以下三部分组成：请求的方法 (GET、POST 和 HEAD 等)。

Web 资源的路径 (http://www.website.com/test/test.html 中的 /test/test.html 部分)。

HTTP 协议的版本 (HTTP/1.0 或 HTTP/1.1)。

在请求消息头的其他行是请求头字段。每一行的格式是：

“头字段”：“:” “头字段的值”

请求头以一个空行结束。如下面上一个 GET 请求的例子：

```
GET / HTTP/1.1
Host: www.csdn.net
Connection: Keep-Alive
Accept: */*
```

如果是 POST 请求，将要提交的实体内容放到消息头的空行后面，如下面是一个 POST 请求的例子：

```
POST /servlets-examples/servlet/RequestParamExample HTTP/1.1
Host: localhost:8888
Content-Length: 29
Connection: Close
firstname=Bill&lastname=Gates
```

在上面的请求消息中 **Content-Length** 表示请求内容的以字节为单位的长度（“firstname=Bill&lastname=Gates”的长度）。在使用 POST 方法时，**这个字段必须提供，而且长度必须等于实体内容的长度，否则服务器将返回一个错误状态码。**

## 二、HTTP 响应消息

HTTP 响应消息同样也分为消息头和实体内容两部分。HTTP 的响应消息头和请求消息头类似：第一行是请求的结果，也就是说，在响应消息头的第一行表明了请求消息是否成功地获得了服务器上的 Web 资源。第一行必须由以下三部分组成：

### 1. 响应消息的 HTTP 版本。

格式为 HTTP/1.1 或 HTTP/1.0。这个版本号未必和请求消息头的版本号一致，这主要是因为，服务器未必支持 HTTP 请求中所描述的 HTTP 版本，如使用 GET / HTTP/1.1 去请求服务器，当服务器只支持 HTTP1.0 时，那么就会返回 HTTP/1.0。

### 2. 状态码。

这个状态码由三位的数字组成，分为五个档次。下面是 HTTP1.1 的响应码：

#### （1）以 1 开头的数字(1xx)。

**临时请求状态码。**由 100 和 101 组成。这类状态码并不经常使用，它们的作为主要是服务器为客户端返回的临时的状态。

#### （2）以 2 开头的数字(2xx)。

**请求成功状态码。**范围从 200 到 206。其中最常用的是 200，它表示客户端请求成功，服务器已经将所请求的 Web 资源返回到了客户端。其他的六个状态码类似，都表示请求成功，只是要指引客户端进和下一步的动作。如状态码 206 表示服务器只是返回了一部分请求资源，客户端要想获得全部的 Web 资源，必须继续发出 HTTP 请求。其他的响应码的含义可以参阅 HTTP1.1 的规范 RFC2616。

#### （3）以 3 开头的数字(3xx)。

**Web 资源重定向状态码。**范围从 300 到 307。所有以 3 开头的状态码都以不同的原因和方式使 Web 资源改变了原来的 URL。如 302 通过一个 Location 字段确定了 Web 资源改变 URL 后的位置。有了这种状态码，当某个网站或其他 Web 资源的 URL 变化后，而访问这些 Web 资源的用户并不能即使知道变化后的 URL；因此，可以在用户访问原来的 URL 时加一个 302 响应，使客户端自动去访问新的 URL。

#### （4）以 4 开头的数字(4xx)。

**客户端错误状态码。**范围从 400 到 417。也许没人希望看到这状态码。但它们确实在 Internet 上大量存在。当用户访问的 Web 资源不存在或是没有权限访问 Web 资源时，服务器将返回这类状态码。这类状态码中最常遇到的是 400。当用户发送一个不存在的 Web 资源路径时（GET、POST 和 HEAD 方法后面跟的路径），服务器就会返回这个状态码。

#### （5）以 5 开头的数字(5xx)。

**服务器错误状态码**。范围从 500 到 505。这类状态码也是一类错误状态码，只是它和 4xx 不同的是，5xx 的错误是由于服务器的原因而产生的；如用户向服务器发送一个 HTTP 协议不支持的方法，如 GET1，服务器将返回 501 错误，表示服务器不支持这个 HTTP 请求方法。

### 3. 状态信息。

响应信息的内容和状态码息息相关。如状态码为 200，则状态信息为“OK”。状态码为 501，状态信息为“Not Implemented”。要想详细了解每一个状态码所对应的状态信息，请参阅 HTTP1.1 的规范 RFC2616。

HTTP 响应消息的其他部分和请求消息一样，也是由很多响应头字段组成，每个字段和字段值占一行。响应消息头使用一个空行结束，空行的后面跟着 HTTP 响应消息的实体内容。如下面是一个完整的 HTTP 请求和响应的例子：

#### HTTP 请求消息

```
GET / HTTP/1.1
Host: www.csdn.net
```

#### HTTP 响应消息

```
HTTP/1.0 200 OK
Content-Length: 132273
Content-Type: text/html
Content-Location: http://www.csdn.net/index.htm
Last-Modified: Sun, 28 Jan 2007 09:20:00 GMT
Accept-Ranges: bytes
ETag: "eed72b7cbd42c71:1b0e"
Server: Microsoft-IIS/6.0
X-Powered-By: ASP.NET
Date: Sun, 28 Jan 2007 09:23:26 GMT
Age: 32
X-Cache: HIT from cachebj244.csdn.net
Connection: close
<!DOCTYPE html >
```

从上面的 HTTP 消息可以看出，**请求和响应消息中的 HTTP 版本并不一定一样**，这说明 CSDN 的 Web 服务器为了和只支持 HTTP1.0 的客户端兼容，也采用了 HTTP1.0 协议。

**注意：**在 HTTP 请求和响应消息中的第一行的请求方法（GET、POST 等）以及 HTTP/1.1 或 HTTP1.0 中的字母必须大写，请求方法后面的路径大小写都可。消息头字段对大小写不敏感，但大多数的请求和响应字段名都采取了首字母大写的方式。

## 22 实现 HTTP 模拟器

**源码下载：**<http://files.cnblogs.com/nokiaguy/HttpSimulator.rar>

在讨论 HTTP 协议的具体请求和响应头字段之前，让我们先来利用以前所学的知识来实现一个 HTTP 模拟器。所谓 HTTP 模拟器就是可以在用户输入 HTTP 的请求消息后，由这个模拟器将 HTTP 请求发送给相应的服务器，再接收服务器的响应消息。这个 HTTP 模拟器有以下几点：

1. 可以手工输入 HTTP 请求，并向服务器发送。
2. 接收服务器的响应消息。

3. 消息头和实体内容分段显示，也就是说，并不是象 Telnet 等客户端一样将 HTTP 响应消息全部显示，而是先显示消息头，然后由用户决定是否显示实体内容。
4. 集中发送请求。这个 HTTP 模拟器和 Telnet 不同的是，并不是一开始就连接服务器，而是将域名、端口以及 HTTP 请求消息都输完后，才连接服务器，并将这些请求发送给服务器。这样做的作用是可以预防服务器提前关闭网络连接的现象。
5. 可以循环做上述的操作。

从以上的描述看，要实现这个 HTTP 模拟器需要以下五步：

1. 建立一个循环，在循环内部是一个请求/响应对。这样就可以向服务器发送多次请求/响应对了。下面的四步都是被包括在循环内部的。
2. 从控制台读取域名和端口，这个功能可以由 readHostAndPort(...)来完成。
3. 从控制台读取 HTTP 请求消息，这个功能由 readHttpRequest(...)来完成。
4. 向服务器发送 HTTP 请求消息，这个功能由 sendHttpRequest()来完成。
5. 读取服务器回送的 HTTP 响应消息，这个功能由 readHttpResponse(...)来完成。

下面我们就来逐步实现这五步：

### 一、建立一个循环

在建立这个循环之前，先建立一个中叫 HttpSimulator 的类，并在这个类中定义一个 run 方法用来运行这个程序。实现代码如下：

```
001 package http;
002
003 import java.net.*;
004 import java.io.*;
005
006 public class HttpSimulator
007 {
008     private Socket socket;
009     private int port = 80;
010     private String host = "localhost";
011     private String request = ""; // HTTP 请求消息,局部全局
012     private boolean isPost, isHead;
013
014     public void run() throws Exception
015     {
016         BufferedReader reader = new BufferedReader(new InputStreamReader(
017             System.in));
018         while (true) // 开始大循环
019         {
020             try
021             {
022                 if (!readHostAndPort(reader))// 从控制台读取域名和端口
023                     break;
024                 readHttpRequest(reader);// 从控制台读取 HTTP 请求消息
025                 sendHttpRequest();//向服务器发送 HTTP 请求消息
026                 readHttpResponse(reader);// 读取服务器回送的 HTTP 响应消息
027             }
028         }
029     }
030 }
```

```

028         catch (Exception e)
029         {
030             System.out.println("err:" + e.getMessage());
031         }
032     }
033 }
034 public static void main(String[] args) throws Exception
035 {
036     new HttpSimulator().run();
037 }
038 }

```

从上面的代码可以看出，第 022、024、025 和 026 分别调用了上述的四个方法。这些方法的具体实现将在后面讨论。上面的代码除了调用这四个核心方法外，还做了一些准备工作。在 008 至 012 行定义了一些以后要用到的变量。在 016 和 017 行使用控制台的输入流建立了 `BufferedReader` 对象，通过这个对象，可以直接从控制台读取字符串，而不是一个个地字节。

## 二、`readHostAndPort(...)`方法的实现

这个方法的主要功能是从控制台读取域名和端口。域名和端口通过“:”隔开，“:”和域名以及端口之间不能有空格。当从控制台读取一个“q”时，这个函数返回 `false`，表示程序可以退出了，否则返回 `true`，表示输入的域名和端口是正确的。这个方法的实现代码如下：

```

001 private boolean readHostAndPort(BufferedReader consoleReader)
002     throws Exception
003 {
004     System.out.print("host:port>");
005     String[] ss = null;
006     String s = consoleReader.readLine();
007     if (s.equals("q"))
008         return false; //返回值退出
009     else
010     {
011         ss = s.split("[ :]");
012         if (!ss[0].equals(""))
013             host = ss[0];
014         if (ss.length > 1)
015             port = Integer.parseInt(ss[1]);
016         System.out.println(host + ":" + String.valueOf(port));
017         return true;
018     }
019 }

```

**第 001 行：**这个方法有一个 `BufferedReader` 类型的参数，这个参数的值就是在 `HttpSimulator.java` 中的第 016 和 017 行根据控制台输入流建立的 `BufferedReader` 对象。

**第 004 行：**这输出 HTTP 模拟器的控制符，就象 Windows 的控制台的“C:”>一样。

**第 006 行：**从控制台读取一行字符串。

**第 011 行：**通过字符串的 `split` 方法和响应的正则表示式 (“[ :]”) 将域名和端口分开。

**注意：**域名的默认值是 localhost，端口的默认值是 80。故在程序中没有初始化此二值。

### 三、readHttpRequest(...)方法的实现

这个方法的主要功能是从控制台读取 HTTP 请求消息，如果输入一个空行，表示请求消息头已经输完；如果使用的是 POST 方法，还要输入 POST 请求的实体内容。这个方法的实现代码如下：

```
001 private void readHttpRequest(BufferedReader consoleReader)
002     throws Exception
003 {
004     System.out.println("请输入 HTTP 请求:");
005     String s = consoleReader.readLine();
006     request = s + "\r\n";
007     boolean isPost = s.substring(0, 4).equals("POST");
008     boolean isHead = s.substring(0, 4).equals("HEAD");
009     while (!(s = consoleReader.readLine()).equals(""))
010         request = request + s + "\r\n";
011     request = request + "\r\n"; //跳出 while 循环
012     if (isPost)
013     {
014         System.out.println("请输入 POST 方法的内容:");
015         s = consoleReader.readLine();
016         request = request + s;
017     }
018 }
```

第 005 行：读入 HTTP 请求消息的第一行。

第 007、008 行：确定所输入的请求方法是不是 POST 和 HEAD。

第 009、010 行：读入 HTTP 请求消息的其余行。

第 012 ~ 017 行：如果 HTTP 请求使用的是 POST 方法，要求用户继续输入 HTTP 请求的实体内容。

### 四、sendHttpRequest()方法的实现

这个方法的功能是将 request 变量中的 HTTP 请求消息发送到服务器。下面是这个方法的实现代码：

```
001 private void sendHttpRequest() throws Exception
002 {
003     socket = new Socket();
004     socket.setSoTimeout(10 * 1000); // 设置读取数据超时为 10 秒。
005     System.out.println("正在连接服务器...");
006     socket.connect(new InetSocketAddress(host, port), 10 * 1000); // 连接超时为 10 秒
007     System.out.println("服务器连接成功!");
008     OutputStream out = socket.getOutputStream();
009     OutputStreamWriter writer = new OutputStreamWriter(out);
010     writer.write(request);
011     writer.flush();
012 }
```

第 004 行：设置读取数据超时为 10 秒。



第 006 行：连接服务器，并设置连接超时为 10 秒。

## 五、readHttpResponse(...)方法的实现

这个方法的主要功能是从服务器读取返回的响应消息。首先读取了响应消息头，然后要求用户输入 Y 或 N 以确定是否显示响应消息的实体内容。这个程序之所以这样做，主要有两个原因：

(1) 为了研究 HTTP 协议。

(2) 由于本程序是以字符串形式显示响应消息的，因此，如果用户请求了一个二进制 Web 资源，如一个 rar 文件，那么实体内容将会显示乱码。所以在显示完响应消息头后由用户决定是否显示实体内容。

这个方法的实现代码如下：

```
001 private void readHttpResponse(BufferedReader consoleReader)
002 {
003     String s = "";
004     try
005     {
006         InputStream in = socket.getInputStream();
007         InputStreamReader inReader = new InputStreamReader(in);
008         BufferedReader socketReader = new BufferedReader(inReader);
009         System.out.println("-----HTTP 头-----");
010         boolean b = true; // true: 未读取消息头 false: 已经读取消息头
011         while ((s = socketReader.readLine()) != null)
012         {
013             if (s.equals("") && b == true && !isHead)
014             {
015                 System.out.println("-----");
016                 b = false;
017                 System.out.print("是否显示 HTTP 的内容(Y/N):");
018                 String choice = consoleReader.readLine();
019                 if (choice.equals("Y") || choice.equals("y"))
020                 {
021                     System.out.println("-----HTTP 内容-----");
022                     continue;
023                 }
024                 else
025                     break;
026             }
027             else
028                 System.out.println(s);
029         }
030     }
031     catch (Exception e)
032     {
033         System.out.println("err: " + e.getMessage());
034     }
035     finally
```



```

036    {
037        try
038        {
039            socket.close();
040        }
041        catch (Exception e)
042        {
043        }
044    }
045    System.out.println("-----");
046 }

```

在上面的代码中 013 行是最值得注意的。其中 `s.equals("")` 表示读入一个空行（表明消息头已经结束）；由于在实体内容中也可以存在空行，因此，`b == true` 来标记消息头是否已经被读过，当读完消息头后，将 `b` 设为 `false`，如果以后再遇到空行，就不会当成消息头来处理了。当 HTTP 请求使用 HEAD 方法时，服务器只返回响应消息头；因此，使用 `!isHead` 来保证使用 HEAD 发送请求时不显示响应消息的内容实体。

现在我们已经实现了这个 HTTP 模拟器，下面让我们来运行并测试它。

### 运行

运行如下的命令

```
java http.HttpSimulator
```

运行以上的命令后，将显示如图 1 所示的界面。

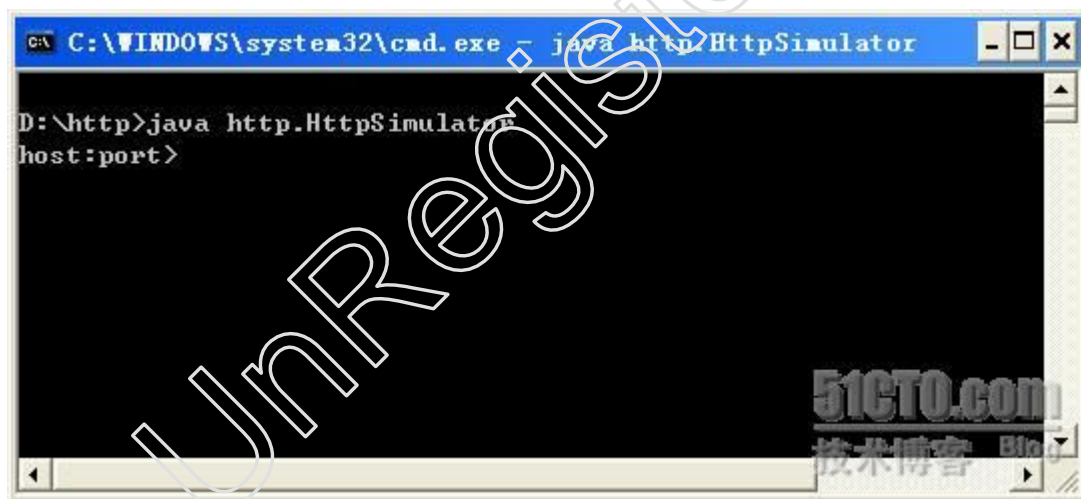


图 1

### 测试

在 HTTP 模拟器中输入如下的域名：

```
www.csdn.net
```

在 HTTP 模拟器中输入如下的 HTTP 请求消息：

```
GET / HTTP/1.1
```

```
Host: www.csdn.net
```

运行的结果如图 2 所示。

图 2

本文实现的 Http 模拟器在后面的文章中会经常使用，读者可以从本文的开始部分下载 Http 模拟器的源代码和.class 文件。

## 23 HTTP 消息头字段

### 一、通用头字段

#### 1. Connection

这个字段只在 HTTP1.1 协议中存在。它决定了客户端和服务端进行了一次会话后，服务器是否立即关闭网络连接。在客户端最直接的表现是使用 read 方法(readLine 方法也是一样)读完客户端请求的 Web 资源后，是否立即返回-1（readLine 返回 null）。Connection 有两个值：Close 和 Keep-Alive。当使用 Connection: Close 时，和 HTTP1.0 协议是一样的，当 read 方法读完数据时立即返回，而使用 Connection: Keep-Alive 时，read 方法在读完数据后还要被阻塞一段时间。直接读取数据超时时间过后，还继续往下执行。在上一篇文章中讨论的 readHttpResponse(...)方法实现的第 011 行可以验证 Connection 的作用。下面让我们来使用 HTTP 模拟器来做一个实验。

(1) 在 HTTP 模拟器中输入如下的域名：

```
www.baidu.com
```

(2) HTTP 模拟器中输入如下的 HTTP 请求信息：

```
GET / HTTP/1.1
```

```
Host: www.baidu.com
```

(3) 按两下回车（输入一个空行）后，发送请求消息，并得到如图 1 如示的 HTTP 响应消息头：

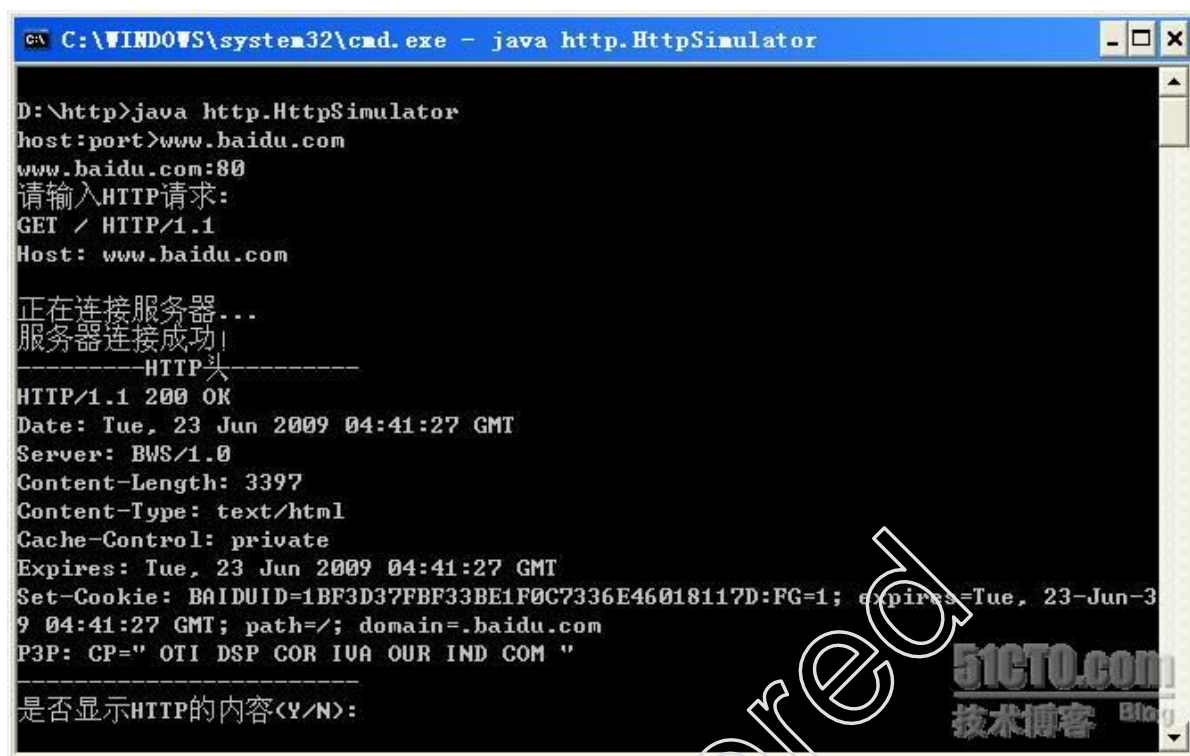


图 1

(4) 输入 y 或 Y 后（在显示 http 响应头后，要立刻输入 Y 或 y），显示响应消息的内容。在显示完内容后，大约过了 10 秒钟才进入“host:port>”提示符（因为在 `sendHttpRequest()` 的实现代码中的 004 行设置了读取数据超时）。

(5) 在“host:port>”提示符下直接按回车，输入最近一次使用的域名 `www.baidu.com` 和 80 端口。再次输入如下的 HTTP 请求：

```
GET / HTTP/1.1
Host: www.baidu.com
Connection: close
```

输入完以上的 HTTP 请求后，重新执行第 3、4 步操作。最后在显示 HTTP 响应消息内容后，直接直入了“host:port>”提示符。除了这种方法，将请求的第一行改为 `GET / HTTP/1.0`。这样也可以无需等待直接结束。

**通过设置 `Connection`，可以在下载 Web 资源（如多线程下载工具、Web 浏览器等）后，立即断开网络连接，这样可以有效地降低客户机的资源消耗。**

## 2. Date

这个 `Date` 头字段描述了请求消息和响应消息被创建的时间。这个字段值是一个 `HTTP-date` 类型，它的格式必须是 GMT（格林尼治）时间，GMT 时间就是北京时间减 8 小时。下面是 `Date` 字段的一个例子：

```
Date: Tue, 15 Nov 2007 08:12:31 GMT
```

## 3. Content-Length

指定消息实体的字节数。在请求消息中 `POST` 方法必须使用 `Content-Length` 来指定请求消息的实体内容的字节数。在响应消息中这个字段值指定了当前 HTTP 响应所返回的 Web 资源的字节数。

# 二、HTTP 请求消息头字段

## 1. Host

Host 字段用于指定客户端所访问的资源所在的主机名和端口号。如果端口号等于连接服务器时所使用的端口号，则端口号可以省略。下面是一个使用 Host 字段的一个例子：

```
Host: www.sina.com.cn
```

这个字段是必须的，如果 HTTP 请求不包含这个字段，服务器将返回 400(Bad Request)响应状态。

## 2. Accept

Accept 字段头确定客户端可以接收的媒体类型。一般的格式是"\*/\*"或"类型/"子类型"。这个子段头可以传递多个媒体类型，中间用"，"隔开。如下面是一个 Accept 的例子：

```
Accept: image/gif, image/jpg
```

如果请求头使用上述的 Accept 字段值，则服务器端在动态生成网页的 IMG 头时将首先包含 gif 格式的图像，如果 gif 图像不存在，则包含 jpg 格式的图像。

## 3. User-Agent

这个字段头用于指定客户端是用什么访问的服务器，如果是 IE6 浏览器，并且本机安装了 .net 2.0，则 User-Agent 会有如下的值：

```
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.2; SV1; Maxthon; .NET CLR 1.1.4322; .NET CLR 2.0.50727; InfoPath.1; InfoPath.2)
```

服务器可以通过这个字段检查客户机的浏览器版本，并根据不同的版本来确定向客户端发送的数据。

## 4. Range

Range 字段头通过服务器只传输一部分 Web 资源。这个字段头可以用来实现断点续传功能。有很多下载工具就是通过这个字段头进行断点续传的。Range 字段可以通过三种格式设置要传输的字节范围：

(1) Range: bytes=1000-2000

传输范围从 1000 到 2000 字节。

(2) Range: bytes=1000-

传输 Web 资源中第 1000 个字节以后的所有内容。

(3) Range bytes=1000

传输最后 1000 个字节。

# 三、HTTP 响应消息头字段

## 1. Accept-Ranges

这个字段说明 Web 服务器是否支持 Range（是否支持断点续传功能），如果支持，则返回 Accept-Ranges: bytes，如果不支持，则返回 Accept-Ranges: none。

## 2. Content-Range

指定了返回的 Web 资源的字节范围。这个字段值的格式是：

开始字节位置—结束字节位置/Web 资源的总字节数

下面是一个使用 Content-Range 的例子：

```
Content-Range: 1000-3000/5000 //可见支持断点续传
```

## 测试

在 HTTP 模拟器中连接服务器 files.cnblogs.com，并输入如下的 HTTP 请求消息：

```
GET /nokiaguy/HttpSimulator.rar HTTP/1.1
```

```
Host: files.cnblogs.com
```

```
Range: bytes=1000-
```

返回的响应消息头如图 2 所示：



```
C:\WINDOWS\system32\cmd.exe - java http.HttpSimulator

D:\http>java http.HttpSimulator
host:port>files.cnblogs.com
files.cnblogs.com:80
请输入HTTP请求:
GET /nokiaguy/HttpSimulator.rar HTTP/1.1
host:files.cnblogs.com
range:bytes=1000-

正在连接服务器...
服务器连接成功!
-----HTTP头-----
HTTP/1.1 206 Partial Content
Content-Type: application/octet-stream
Content-Range: bytes 1000-3759/3760
Last-Modified: Tue, 09 Jun 2009 04:38:54 GMT
Accept-Ranges: bytes
ETag: "b6301c32bce8c91:0"
Server: Microsoft-IIS/7.0
X-Powered-By: ASP.NET
Date: Tue, 23 Jun 2009 05:00:08 GMT
Content-Length: 2760
-----
是否显示HTTP的内容(Y/N):
```

图 2

从上图可以看出，服务器 files.cnblogs.com 支持断点继传功能。而且还可以验证 Content-Length 的值是当前会话传过来的字节数，并不是 Web 资源的总的字节数。而 Content-Range 字段值中"/"后面的数才是 Web 资源总的字节数。

### 3. Location

这个字段指定了上哪个新地址获得新的 Web 资源。当 Web 资源更换 URL 后，而一些老用户不能及时得知这个新的 URL；因此，在老用户访问原来的 URL 时，使用 Location 将这个新的 URL 返回给客户端。这个地址转换对用户来说是完全透明。

## 24 实现 HTTP 断点续传下载工具

源代码：<http://files.cnblogs.com/nokiaguy/download.rar>

在前面的文章曾讨论了 HTTP 消息头的三个和断点继传有关的字段。一个是请求消息的字段 Range，另两个是响应消息字段 Accept-Ranges 和 Content-Range。其中 Accept-Ranges 用来断定 Web 服务器是否支持断点继传功能。在这里为了演示如何实现断点继传功能，假设 Web 服务器支持这个功能；因此，我们只使用 Range 和 Content-Range 来完成一个断点继传工具的开发。

#### ● 要实现一个什么样的断点续传工具？

这个断点续工具是一个单线程的下载工具。它通过参数传入一个文本文件。这个文件的格式如下：

```
http://www.ishare.cc/d/1174254-2/106.jpg d:\ok1.jpg 8192
http://www.ishare.cc/d/1174292-2/156.jpg d:\ok2.jpg 12345
http://www.ishare.cc/d/1174277-2/147.jpg d:\ok3.jpg 3456
```

这个文本文件的每一行是一个下载项，这个下载项分为三部分：

- 要下载的 Web 资源的 URL。



- 要保存的本地文件名。
- 下载的缓冲区大小（单位是字节）。

使用至少一个空格来分隔这三部分。这个下载工具逐个下载这些文件，在这些文件全部下载完后程序退出。

## ● 断点续传的工作原理

“断点续传”顾名思义，就是一个文件下载了一部分后，由于服务器或客户端的原因，当前的网络连接中断了。在中断网络连接后，用户还可以再次建立网络连接来继续下载这个文件还没有下完的部分。

要想实现单线程断点续传，必须在客户端保存两个数据。

1. 已经下载的字节数。
2. 下载文件的 URL。

一旦重新建立网络连接后，就可以利用这两个数据接着未下载完的文件继续下载。在本下载工具中第一种数据就是文件已经下载的字节数，而第二个数据在上述的下载文件中保存。

在继续下载时检测已经下载的字节数，假设已经下载了 3000 个字节，那么 HTTP 请求消息头的 Range 字段被设为如下形式：

```
Range: bytes=3000-
```

HTTP 响应消息头的 Content-Range 字段被设为如下的形式：

```
Content-Range: bytes 3000-10000/10001
```

## ● 实现断点续传下载工具

一个断点续传下载程序可按如下几步实现：

1. 输入要下载文件的 URL 和要保存的本地文件名，并通过 Socket 类连接到这个 URL 所指的服务器上。
2. 在客户端根据下载文件的 URL 和这个本地文件生成 HTTP 请求消息。在生成请求消息时分为两种情况：
  - （1）第一次下载这个文件，按正常情况生成请求消息，也就是说生成不包含 Range 字段的请求消息。
  - （2）以前下载过，这次是接着下载这个文件。这就进入了断点续传程序。在这种情况下生成的 HTTP 请求消息中必须包含 Range 字段。由于是单线程下载，因此，这个已经下载了一部分的文件的大小就是 Range 的值。假设当前文件的大小是 1234 个字节，那么将 Range 设成如下的值：

```
Range: bytes=1234-
```

3. 向服务器发送 HTTP 请求消息。
4. 接收服务器返回的 HTTP 响应消息。
5. 处理 HTTP 响应消息。在本程序中需要从响应消息中得到下载文件的总字节数。如果是第一次下载，也就是说响应消息中不包含 Content-Range 字段时，这个总字节数也就是 Content-Length 字段的值。如果响应消息中不包含 Content-Length 字段，则这个总字节数无法确定。这就是为什么使用下载工具下载一些文件时没有文件大小和下载进度的原因。如果响应消息中包含 Content-Range 字段，总字节数就是 Content-Range: bytes m-n/k 中的 k，如 Content-Range 的值为：

```
Content-Range: bytes 1000-5000/5001
```

则总字节数为 5001。由于本程序使用的 Range 值类型是得到从某个字节开始往后的所有字节，因此，当前的响应消息中的 Content-Range 总是能返回还有多少个字节未下载。如上面的例子未下载的字节数为  $5000 - 1000 + 1 = 4001$ 。

6. 开始下载文件，并计算下载进度（百分比形式）。如果网络连接断开时，文件仍未下载完，重新执行第一步。如果文件已经下载完，退出程序。

分析以上六个步骤得知，有四个主要的功能需要实现：

1. 生成 HTTP 请求消息，并将其发送到服务器。这个功能由 generateHttpRequest 方法来完成。
2. 分析 HTTP 响应消息头。这个功能由 analyzeHttpHeader 方法来完成。
3. 得到下载文件的实际大小。这个功能由 getFileSize 方法来完成。
4. 下载文件。这个功能由 download 方法来完成。

以上四个方法均被包含在这个断点续传工具的核心类 `HttpDownload.java` 中。在给出 `HttpDownload` 类的实现之前先给出一个接口 `DownloadEvent` 接口，从这个接口的名字就可以看出，它是用来处理下载过程中的事件的。下面是这个接口的实现代码：

```
package download;

public interface DownloadEvent
{
    void percent(long n);           // 下载进度
    void state(String s);          // 连接过程中的状态切换
    void viewHttpHeaders(String s); // 枚举每一个响应消息字段
}
```

从上面的代码可以看出，`DownloadEvent` 接口中有三个事件方法。在以后的主函数中将实现这个接口，来向控制台输出相应的信息。下面给出了 `HttpDownload` 类的主体框架代码：

```
001 package download;
002
003 import java.net.*;
004 import java.io.*;
005 import java.util.*;
006
007 public class HttpDownload
008 {
009     private HashMap httpHeaders = new HashMap();
010     private String stateCode;
011
012     // generateHttpRequest 方法
013
014     /* analyzeHTTPHeader 方法
015     *
016     * addHeaderToMap 方法
017     *
018     * analyzeFirstLine 方法
019     */
020
021     // getFileSize 方法
022
023     // download 方法
024
025     /* getHeader 方法
026     *
027     * getIntHeader 方法
028     */
029 }
```

上面的代码只是 `HttpDownload` 类的框架代码，其中的方法并未真正实现。我们可以从中看出第 012、014、021 和 023 行就是上述的四个主要的方法。在 016 和 018 行的 `addHeaderToMap` 和 `analyzeFirst`



Line 方法将在 analyzeHttpRequest 方法中用到。而 025 和 027 行的 getHeader 和 getIntHeader 方法在 getFileSize 和 download 方法都会用到。上述的八个方法的实现都会在后面给出。

```
001 private void generateHttpRequest(OutputStream out, String host,
002     String path, long startPos) throws IOException
003 {
004     OutputStreamWriter writer = new OutputStreamWriter(out);
005     writer.write("GET " + path + " HTTP/1.1\r\n");
006     writer.write("Host: " + host + "\r\n");
007     writer.write("Accept: */*\r\n");
008     writer.write("User-Agent: My First Http Download\r\n");
009     if (startPos > 0) // 如果是断点续传，加入 Range 字段
010         writer.write("Range: bytes=" + String.valueOf(startPos) + "-\r\n");
011     writer.write("Connection: close\r\n\r\n");
012     writer.flush();
013 }
```

这个方法有四个参数：

- OutputStream out

使用 Socket 对象的 getOutputStream 方法得到的输出流。

String host

下载文件所在的服务器的域名或 IP。

String path

下载文件在服务器上的路径，也就跟在 GET 方法后面的部分。

long startPos

从文件的 startPos 位置开始下载。如果 startPos 为 0，则不生成 Range 字段。

```
001 private void analyzeHttpRequest(InputStream inputStream, DownloadEvent de)
002     throws Exception
003 {
004     String s = "";
005     byte b = -1;
006     while (true)
007     {
008         b = (byte) inputStream.read();
009         if (b == '\r')
010         {
011             b = (byte) inputStream.read();
012             if (b == '\n')
013             {
014                 if (s.equals(""))
015                     break;
016                 de.viewHttpHeaders(s);
017                 addHeaderToMap(s);
018                 s = "";
019             }
020         }
```

```

021     else
022         s += (char) b;
023     }
024 }
025
026 private void analyzeFirstLine(String s)
027 {
028     String[] ss = s.split("[ ]+");
029     if (ss.length > 1)
030         stateCode = ss[1];
031 }
032 private void addHeaderToMap(String s)
033 {
034     int index = s.indexOf(":");
035     if (index > 0)
036         httpHeaders.put(s.substring(0, index), s.substring(index + 1).trim());
037     else
038         analyzeFirstLine(s);
039 }

```

**第 001 ~ 024 行：**analyzeHTTPHeader 方法的实现。这个方法有两个参数。其中 inputStream 是用 Socket 对象的 getInputStream 方法得到的输入流。这个方法是直接使用字节流来分析的 HTTP 响应头（主要是因为下载的文件不一定是文本文件；因此，都统一使用字节流来分析和下载），每两个""r""n""之间的就是一个字段和字段值对。在 016 行调用了 DownloadEvent 接口的 viewHttpHeaders 事件方法来枚举每一个响应头字段。

**第 026 ~ 031 行：**analyzeFirstLine 方法的实现。这个方法的功能是分析响应消息头的第一行，并从中得到状态码后，将其保存在 stateCode 变量中。这个方法的参数 s 就是响应消息头的第一行。

**第 032 ~ 039 行：**addHeaderToMap 方法的实现。这个方法的功能是将每一个响应请求消息字段和字段值加到在 HttpDownload 类中定义的 httpHeaders 哈希映射中。在第 034 行查找每一行消息头是否包含":", 如果包含":", 这一行必是消息头的第一行。因此，在第 038 行调用了 analyzeFirstLine 方法从第一行得到响应状态码。

```

001 private String getHeader(String header)
002 {
003     return (String) httpHeaders.get(header);
004 }
005 private int getIntHeader(String header)
006 {
007     return Integer.parseInt(getHeader(header));
008 }

```

这两个方法将会在 getFileSize 和 download 中被调用。它们的功能是从响应消息中根据字段字得到相应的字段值。getHeader 得到字符串形式的字段值，而 getIntHeader 得到整数型的字段值。

```

001 public long getFileSize()
002 {
003     long length = -1;
004     try

```

```

005    {
006        length = getIntHeader("Content-Length");
007        String[] ss = getHeader("Content-Range").split("/");
008        if (ss.length > 1)
009            length = Integer.parseInt(ss[1]);
010        else
011            length = -1;
012    }
013    catch (Exception e)
014    {
015    }
016    return length;
017 }

```

getFileSize 方法的功能是得到下载文件的实际大小。首先在 006 行通过 Content-Length 得到了当前响应消息的实体内容大小。然后在 009 行得到了 Content-Range 字段值所描述的文件的实际大小( "" 后面的值)。如果 Content-Range 字段不存在，则文件的实际大小就是 Content-Length 字段的值。如果 Content-Length 字段也不存在，则返回-1，表示文件实际大小无法确定。

```

001 public void download(DownloadEvent de, String url, String localFN,
002     int cacheSize) throws Exception
003 {
004     File file = new File(localFN);
005     long finishedSize = 0;
006     long fileSize = 0; // localFN 所指的文件的实际大小
007     FileOutputStream fileOut = new FileOutputStream(localFN, true);
008     URL myUrl = new URL(url);
009     Socket socket = new Socket();
010     byte[] buffer = new byte[cacheSize]; // 下载数据的缓冲
011
012     if (file.exists())
013         finishedSize = file.length();
014
015     // 得到要下载的 Web 资源的端口号，未提供，默认是 80
016     int port = (myUrl.getPort() == -1) ? 80 : myUrl.getPort();
017     de.state("正在连接" + myUrl.getHost() + ":" + String.valueOf(port));
018     socket.connect(new InetSocketAddress(myUrl.getHost(), port), 20000);
019     de.state("连接成功!");
020
021     // 产生 HTTP 请求消息
022     generateHttpRequest(socket.getOutputStream(), myUrl.getHost(), myUrl
023         .getPath(), finishedSize);
024
025     InputStream inputStream = socket.getInputStream();
026     // 分析 HTTP 响应消息头
027     analyzeHTTPHeader(inputStream, de);

```

```

028     fileSize = getFileSize(); // 得到下载文件的实际大小
029     if (finishedSize >= fileSize)
030         return;
031     else
032     {
033         if (finishedSize > 0 && stateCode.equals("200"))
034             return;
035     }
036     if (stateCode.charAt(0) != '2')
037         throw new Exception("不支持的响应码");
038     int n = 0;
039     long m = finishedSize;
040     while ((n = inputStream.read(buffer)) != -1)
041     {
042         fileOut.write(buffer, 0, n);
043         m += n;
044         if (fileSize != -1)
045         {
046             de.percent(m * 100 / fileSize);
047         }
048     }
049     fileOut.close();
050     socket.close();
051 }

```

download 方法是断点续传工具的核心方法。它有四个参数：

**DownloadEvent de**

用于处理下载事件的接口。

**String url**

要下载文件的 URL。

**String localFN**

要保存的本地文件名，可以用这个文件的大小来确定已经下载了多少个字节。

**int cacheSize**

下载数据的缓冲区。也就是一次从服务器下载多个字节。这个值不宜太小，因为，频繁地从服务器下载数据，会降低网络的利用率。一般可以将这个值设为 8192（8K）。

为了分析下载文件的 url，在 008 行使用了 URL 类，这个类在以后还会介绍，在这里只要知道使用这个类可以将使用各种协议的 url（包括 HTTP 和 FTP 协议）的各个部分分解，以便单独使用其中的一部分。

**第 029 行：**根据文件的实际大小和已经下载的字节数(finishedSize)来判断是否文件是否已经下载完成。当文件的实际大小无法确定时，也就是 fileSize 返回-1 时，不能下载。

**第 033 行：**如果文件已经下载了一部分，并且返回的状态码仍是 200（应该是 206），则表明服务器并不支持断点续传。当然，这可以根据另一个字段 Accept-Ranges 来判断。

**第 036 行：**由于本程序未考虑重定向(状态码是 3xx)的情况，因此，在使用 download 时，不要下载返回 3xx 状态码的 Web 资源。

**第 040 ~ 048 行：**开始下载文件。第 046 行调用 DownloadEvent 的 percent 方法来返回下载进度。

```
001 package download;
002
003 import java.io.*;
004
005 class NewProgress implements DownloadEvent
006 {
007     private long oldPercent = -1;
008     public void percent(long n)
009     {
010         if (n > oldPercent)
011         {
012             System.out.print "[" + String.valueOf(n) + "%]";
013             oldPercent = n;
014         }
015     }
016     public void state(String s)
017     {
018         System.out.println(s);
019     }
020     public void viewHttpHeaders(String s)
021     {
022         System.out.println(s);
023     }
024 }
025
026 public class Main
027 {
028     public static void main(String[] args) throws Exception
029     {
030
031         DownloadEvent progress = new NewProgress();
032         if (args.length < 1)
033         {
034             System.out.println("用法: java class 下载文件名");
035             return;
036         }
037         FileInputStream fis = new FileInputStream(args[0]);
038         BufferedReader fileReader = new BufferedReader(new InputStreamReader(
039             fis));
040         String s = "";
041         String[] ss;
042         while ((s = fileReader.readLine()) != null)
043         {
044             try
```

```

045      {
046          ss = s.split("[ ]+");
047          if (ss.length > 2)
048          {
049              System.out.println("\r\n-----");
050              System.out.println("正在下载:" + ss[0]);
051              System.out.println("文件保存位置:" + ss[1]);
052              System.out.println("下载缓冲区大小:" + ss[2]);
053              System.out.println("-----");
054              HttpDownload httpDownload = new HttpDownload();
055              httpDownload.download(new NewProgress(), ss[0], ss[1],
056                                  Integer.parseInt(ss[2]));
057          }
058      }
059      catch (Exception e)
060      {
061          System.out.println(e.getMessage());
062      }
063  }
064  fileReader.close();
065  }
066  }

```

第 005 ~ 024 行：实现 DownloadEvent 接口的 NewDownloadEvent 类。用于在 Main 函数里接收相应事件传递的数据。

第 026 ~ 065 行：下载工具的 Main 方法。在这个 Main 方法里，打开下载资源列表文件，逐行下载相应的 Web 资源。

## 测试

假设 download.txt 在当前目录中，内容如下：

```

http://files.cnblogs.com/nokiaguy/HttpSimulator.rar HttpSimulator.rar 8192
http://files.cnblogs.com/nokiaguy/designpatterns.rar designpatterns.rar 4096
http://files.cnblogs.com/nokiaguy/download.rar download.rar 8192

```

这两个 URL 是在本机的 Web 服务器(如 IIS)的虚拟目录中的两个文件，将它们下载在 D 盘根目录。

运行下面的命令：

```
java download.Main download.txt
```

运行的结果如图 1 所示。



```
C:\WINDOWS\system32\cmd.exe
F:\>java download.Main download.txt

-----
正在下载:http://files.cnblogs.com/nokiaguy/HttpSimulator.rar
文件保存位置:HttpSimulator.rar
下载缓冲区大小:8192
-----
正在连接files.cnblogs.com:80
连接成功!
HTTP/1.1 200 OK
Content-Type: application/octet-stream
Last-Modified: Tue, 09 Jun 2009 04:38:54 GMT
Accept-Ranges: bytes
ETag: "b6301c32bce8c91:0"
Server: Microsoft-IIS/7.0
X-Powered-By: ASP.NET
Date: Thu, 02 Jul 2009 10:21:14 GMT
Connection: close
Content-Length: 3760
[100%]
-----
正在下载:http://files.cnblogs.com/nokiaguy/designpatterns.rar
文件保存位置:designpatterns.rar
下载缓冲区大小:4096
-----
正在连接files.cnblogs.com:80
连接成功!
HTTP/1.1 200 OK
Content-Type: application/octet-stream
Last-Modified: Wed, 11 Feb 2009 05:46:24 GMT
Accept-Ranges: bytes
ETag: "e6f33013c8cc91:0"
Server: Microsoft-IIS/7.0
X-Powered-By: ASP.NET
Date: Thu, 02 Jul 2009 10:21:14 GMT
Connection: close
Content-Length: 109590
[3%][7%][11%][15%][19%][23%][26%][30%][34%][37%][38%][41%][42%][45%][49%][53%][57%][61%][65%][68%][72%][75%][77%][81%][85%][86%][90%][94%][98%][100%]
-----
正在下载:http://files.cnblogs.com/nokiaguy/download.rar
文件保存位置:download.rar
下载缓冲区大小:8192
-----
正在连接files.cnblogs.com:80
连接成功!
HTTP/1.1 200 OK
Content-Type: application/octet-stream
Last-Modified: Thu, 02 Jul 2009 10:17:34 GMT
Accept-Ranges: bytes
ETag: "e01f6051fefaa91:0"
Server: Microsoft-IIS/7.0
X-Powered-By: ASP.NET
Date: Thu, 02 Jul 2009 10:21:15 GMT
Connection: close
Content-Length: 7024
[100%]
F:\>
```

图 1

## 25 创建 ServerSocket 对象

ServerSocket 类的构造方法有四种重载形式，它们的定义如下：

```
public ServerSocket() throws IOException
public ServerSocket(int port) throws IOException
public ServerSocket(int port, int backlog) throws IOException
public ServerSocket(int port, int backlog, InetAddress bindAddr) throws IOException
```

在上面的构造方法中涉及到了三个参数：port、backlog 和 bindAddr。其中 port 是 ServerSocket 对象要绑定的端口，*backlog 是请求队列的长度*，bindAddr 是 ServerSocket 对象要绑定的 IP 地址。

## 一、通过构造方法绑定端口

通过构造方法绑定端口是创建 ServerSocket 对象最常用的方式。可以通过如下的构造方法来绑定端口：

```
public ServerSocket(int port) throws IOException
```

如果 port 参数所指定的端口已经被绑定，构造方法就会抛出 IOException 异常。但实际上抛出的异常是 BindException。从图 4.2 的异常类继承关系图可以看出，所有和网络有关的异常都是 IOException 类的子类。为了 ServerSocket 构造方法还可以抛出其他的异常，就使用了 IOException。

如果 port 的值为 0，系统就会随机选取一个端口号。但随机选取的端口意义不大，因为客户端在连接服务器时需要明确知道服务端程序的端口号。可以通过 ServerSocket 的 toString 方法输出和 ServerSocket 对象相关的信息。下面的代码输入了和 ServerSocket 对象相关的信息。

```
ServerSocket serverSocket = new ServerSocket(1320);
System.out.println(serverSocket);
```

运行结果：

```
ServerSocket[addr=0.0.0.0/0.0.0.0,port=0,localport=1320]
```

上面的输出结果中的 addr 是服务端绑定的 IP 地址，*如果未绑定 IP 地址，这个值是 0.0.0.0*，在这种情况下，ServerSocket 对象将监听服务端所有网络接口的所有 IP 地址。port 永远是 0。localport 是 ServerSocket 绑定的端口，如果 port 值为 0（此处不指输出结果的 port，指 ServerSocket 构造方法的参数 port），localport（此处指输出结果的 port）是一个随机选取的端口号。

在操作系统中规定 1 ~ 1023 为系统使用的端口号。端口号的最小值是 1，最大值是 65535。在 Windows 中用户编写的程序可以绑定端口号小于 1024 的端口，但在 Linux/Unix 下必须使用 root 登录才可以绑定小于 1024 的端口。在前面的文章中曾使用 Socket 类来判断本机打开了哪些端口，其实使用 ServerSocket 类也可以达到同样的目的。基本原理是用 ServerSocket 来绑定本机的端口，如果绑定某个端口时抛出 BindException 异常，就说明这个端口已经打开，反之则这个端口未打开。

```
package server;
```

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

```
public class ScanPort
```

```
{
    public static void main(String[] args)
    {
        if (args.length == 0)
            return;
        int minPort = 0, maxPort = 0;
        String ports[] = args[0].split("-");
        minPort = Integer.parseInt(ports[0]);
        maxPort = (ports.length > 1) ? Integer.parseInt(ports[1]) : minPort;
        for (int port = minPort; port <= maxPort; port++)
```

```

try
{
    ServerSocket serverSocket = new ServerSocket(port);
    serverSocket.close();
}
catch (Exception e)
{
    System.err.println(e.getClass());
    System.err.println("端口" + port + "已经打开!");
}
}
}

```

在上面的代码中输出了创建 `ServerSocket` 对象时抛出的异常类的信息。`ScanPort` 通过命令行参数将待扫描的端口号范围传入程序，参数格式为：`minPort-maxPort`，如果只输入一个端口号，`ScanPort` 程序只扫描这个端口号。

测试：

```
java server.ScanPort 1-1023
```

运行结果

```

class java.net.BindException
端口 80 已经打开!
class java.net.BindException
端口 135 已经打开!

```

## 二、设置请求队列的长度

在编写服务端程序时，一般会通过多线程来同时处理多个客户端请求。也就是说，使用一个线程来接收客户端请求，当接到一个请求后（得到一个 `Socket` 对象），会创建一个新线程，将这个客户端请求交给这个新线程处理。而那个接收客户端请求的线程则继续接收客户端请求，这个过程的实现代码如下：

```

ServerSocket serverSocket = new ServerSocket(1234); // 绑定端口
....// 处理其他任务的代码
while(true)
{
    Socket socket = serverSocket.accept(); // 等待接收客户端请求
    ....// 处理其他任务的代码
    new ThreadClass(socket).start(); // 创建并运行处理客户端请求的线程
}

```

上面代码中的类 `ThreadClass` 是 `Thread` 类的子类，这个类的构造方法有一个 `Socket` 类型的参数，可以通过构造方法将 `Socket` 对象传入 `ThreadClass` 对象，并在 `ThreadClass` 对象的 `run` 方法中处理客户端请求。这段代码从表面上看好像是天衣无缝，无论有多少客户端请求，只要服务器的配置足够高，就都可以处理。但仔细思考上面的代码，我们可能会发现一些问题。如果在第 2 行和第 6 行有足够复杂的代码，执行时间也比较长，这就意味着服务端程序无法及时响应客户端的请求。

假设第 2 行和第 6 行的代码是 `Thread.sleep(3000)`，这将使程序延迟 3 秒。那么在这 3 秒内，程序不会执行 `accept` 方法。这段程序只是将端口绑定到了 1234 上，并未开始接收客户端请求。如果在这时一个客

户端向端口 1234 发来了一个请求，从理论上讲，客户端应该出现拒绝连接错误，但客户端却显示连接成功。究其原因，就是这节要讨论的请求队列在起作用。

在使用 `ServerSocket` 对象绑定一个端口后，操作系统就会为这个端口分配一个先进先出的队列（这个队列长度的默认值一般是 50），这个队列用于保存未处理的客户端请求，因此叫**请求队列**。而 `ServerSocket` 类的 `accept` 方法负责从这个队列中读取未处理的客户端请求。如果请求队列为空，`accept` 则处于阻塞状态。每当客户端向服务端发来一个请求，服务端会首先将这个客户端请求保存在请求队列中，然后 `accept` 再从请求队列中读取。这也可以很好地解释为什么上面的代码在还未执行到 `accept` 方法时，仍然可以接收一定数量的客户端请求。如果请求队列中的客户端请求数达到请求队列的最大容量时，服务端将无法再接收客户端请求。如果这时客户端再向服务端发请求，客户端将会抛出一个 `SocketException` 异常。

`ServerSocket` 类有两个构造方法可以使用 `backlog` 参数重新设置请求队列的长度。在以下几种情况，仍然会采用操作系统限定的请求队列的最大长度：

- `backlog` 的值小于等于 0。
- `backlog` 的值大于操作系统限定的请求队列的最大长度。
- 在 `ServerSocket` 构造方法中未设置 `backlog` 参数。

下面代码演示了请求队列的一些特性，请求队列长度通过命令行参数传入 `SetRequestQueue`。

```
package server;

import java.net.*;

class TestRequestQueue
{
    public static void main(String[] args) throws Exception
    {
        for (int i = 0; i < 10; i++)
        {
            Socket socket = new Socket("localhost", 1234);
            socket.getOutputStream().write(1);
            System.out.println("已经成功创建第" + String.valueOf(i + 1) + "个客户端连接!");
        }
    }
}

public class SetRequestQueue
{
    public static void main(String[] args) throws Exception
    {
        if (args.length == 0)
            return;
        int queueLength = Integer.parseInt(args[0]);
        ServerSocket serverSocket = new ServerSocket(1234, queueLength);
        System.out.println("端口(1234)已经绑定，请按回车键开始处理客户端请求！");
        System.in.read();
        int n = 0;
        while (true)
        {
```

```

        System.out.println("<准备接收第" + (++n) + "个客户端请求!");
        Socket socket = serverSocket.accept();

        System.out.println("正在处理第" + n + "个客户端请求...");

        Thread.sleep(3000);

        System.out.println("第" + n + "个客户端请求已经处理完毕!>");
    }
}
}

```

**测试:** (按着以下步骤操作)

1. 执行如下命令 (在执行这条命令后, 先不要按回车键):

```
java server.SetRequestQueue 2
```

运行结果:

端口(1234)已经绑定, 请按[回车键开始](#)处理客户端请求!

2. 执行如下命令:

```
java server.TestRequestQueue
```

运行结果:

已经成功创建第 1 个客户端连接!

已经成功创建第 2 个客户端连接!

Exception in thread "main" java.net.SocketException: Connection reset by peer: socket write error

```

    at java.net.SocketOutputStream.socketWrite0(Native Method)
    at java.net.SocketOutputStream.socketWrite(SocketOutputStream.java:92)
    at java.net.SocketOutputStream.write(SocketOutputStream.java:115)
    at server.TestRequestQueue.main(SetRequestQueue.java:12)

```

3. 按回车键继续执行 `SetRequestQueue` 后, 运行结果如下:

端口(1234)已经绑定, 请按回车键开始处理客户端请求!

<准备接收第 1 个客户端请求!

正在处理第 1 个客户端请求...

第 1 个客户端请求已经处理完毕!>

<准备接收第 2 个客户端请求!

正在处理第 2 个客户端请求...

第 2 个客户端请求已经处理完毕!>

<准备接收第 3 个客户端请求!

从第二步的运行结果可以看出, 当 `TestRequestQueue` 创建两个 `Socket` 连接之后, 服务端的请求队列已满, 并且服务端暂时无法继续执行 (由于 `System.in.read()` 的原因而暂停程序的执行, 等待用户的输入)。因此, 服务端程序无法再接收客户端请求。这时 `TestRequestQueue` 抛出了一个 `SocketException` 异常。在 `TestRequestQueue` 已经创建成功的两个 `Socket` 连接已经保存在服务端的请求队列中。在这时按任意键继续执行 `SetRequestQueue`, `accept` 方法就会从请求队列中将这两个客户端请求队列中依次读出来。从第三步的运行结果可以看出, 服务端处理完这两个请求后 (一个<...>包含的就是一个处理过程), 请求队列为空, 这时 `accept` 处理阻塞状态, 等待接收第三个客户端请求。如果这时再运行 `TestRequestQueue`, 服务端会接收几个客户端请求呢? 如果将请求队列的长度设为大于 10 的数, `TestRequestQueue` 的运行结果会



是什么呢？读者可以自己做一些实验，看看和自己认为的结果是否一致。

### 三、绑定 IP 地址

在有多网络接口或多个 IP 地址的计算机上可以使用如下的构造方法将服务端绑定在某一个 IP 地址上：

```
public ServerSocket(int port, int backlog, InetAddress bindAddr) throws IOException
```

`bindAddr` 参数就是要绑定的 IP 地址。如果将服务端绑定到某一个 IP 地址上，就只有可以访问这个 IP 地址的客户端才能连接到服务器上。如一台机器上有两块网卡，一块网卡连接内网，另一块连接外网。如果用 Java 实现一个 Email 服务器，并且只想让内网的用户使用它。就可以使用这个构造方法将 `ServerSocket` 对象绑定到连接内网的 IP 地址上。这样外网就无法访问 Email 服务器了。可以使用如下代码来绑定 IP 地址：

```
ServerSocket serverSocket = new  
ServerSocket(1234, 0, InetAddress.getByName("192.168.18.10"));
```

上面的代码将 IP 地址绑定到了 192.168.18.10 上，因此，服务端程序只能使用绑定了这个 IP 地址的网络接口进行通讯。

### 四、默认构造方法的使用

除了使用 `ServerSocket` 类的构造方法绑定端口外，还可以用 `ServerSocket` 的 `bind` 方法来完成构造方法所做的工作。要想使用 `bind` 方法，必须得用 `ServerSocket` 类的默认构造方法（没有参数的构造方法）来创建 `ServerSocket` 对象。`bind` 方法有两个重载形式，它们的定义如下：

```
public void bind(SocketAddress endpoint) throws IOException  
public void bind(SocketAddress endpoint, int backlog) throws IOException
```

`bind` 方法不仅可以绑定端口，也可以设置请求队列的长度以及绑定 IP 地址。`bind` 方法的作用是为了在建立 `ServerSocket` 对象后设置 `ServerSocket` 类的一些选项。而这些选项必须在绑定端口之前设置，一旦绑定了端口后，再设置这些选项将不再起作用。下面的代码演示了 `bind` 方法的使用及如何设置 `ServerSocket` 类的选项。

```
ServerSocket serverSocket1 = new ServerSocket();  
serverSocket1.setReuseAddress(true);  
serverSocket1.bind(new InetSocketAddress(1234));  
ServerSocket serverSocket2 = new ServerSocket();  
serverSocket2.setReuseAddress(true);  
serverSocket2.bind(new InetSocketAddress("192.168.18.10", 1234));  
ServerSocket serverSocket3 = new ServerSocket();  
serverSocket3.setReuseAddress(true);  
serverSocket3.bind(new InetSocketAddress("192.168.18.10", 1234), 30);
```

在上面的代码中设置了 `SO_REUSEADDR` 选项（这个选项将在后面的文章中详细讨论）。如果使用下面的代码，这个选项将不起作用。

```
ServerSocket serverSocket3 = new ServerSocket(1234);  
serverSocket3.setReuseAddress(true);
```

在第 6 行绑定了 IP 地址和端口。使用构造方法是无法得到这个组合的（绑定 IP 地址前必须得设置 `backlog` 参数），因此 `bind` 方法比构造方法更灵活。



## 26 在服务端接收和发送数据

在建立完 `ServerSocket` 对象后，通过 `accept` 方法返回的 `Socket` 对象，服务端就可以和客户端进行数据交互。

`Socket` 类和 `ServerSocket` 类都有两个得到输入输出流的方法：`getInputStream` 和 `getOutputStream`。对于 `Socket` 类而言，使用 `getInputStream` 方法得到的 `InputStream` 是从服务端获取数据，而 `getOutputStream` 方法得到的 `OutputStream` 是向服务端发送数据。而 `ServerSocket` 的 `getInputStream` 和 `getOutputStream` 方法也类似。`InputStream` 从客户端读取数据，`OutputStream` 向客户端发送数据。

下面的代码是一个接收 HTTP 请求，并返回 HTTP 请求头信息的程序，它演示了 `ServerSocket` 类如何读取和发送来自客户端的数据。

```
package server;

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

public class HttpEchoServer extends Thread
{
    private Socket socket;
    public void run()
    {
        try
        {
            InputStreamReader isr = new InputStreamReader(socket
                .getInputStream());
            BufferedReader br = new BufferedReader(isr);
            OutputStreamWriter osw = new OutputStreamWriter(socket
                .getOutputStream());
            osw.write("HTTP/1.1 200 OK\r\n\r\n");
            String s = "";
            while (!(s = br.readLine()).equals(""))
                osw.write("<html><body>" + s + "<br></body></html>");
            osw.flush();
            socket.close();
        }
        catch (Exception e)
        {
        }
    }
    public HttpEchoServer(Socket socket)
    {
        this.socket = socket;
    }
}
```

```

public static void main(String[] args) throws Exception
{
    ServerSocket serverSocket = new ServerSocket(8888);
    System.out.println("服务器已经启动，端口：8888");
    while (true)
    {
        Socket socket = serverSocket.accept();
        new HttpEchoServer(socket).start();
    }
}
}

```

编译并在 HttpEchoServer 运行时，在 IE 的地址栏中输入 URL: <http://localhost:8888> (相当于请求)。输出结果如图 1 所示。

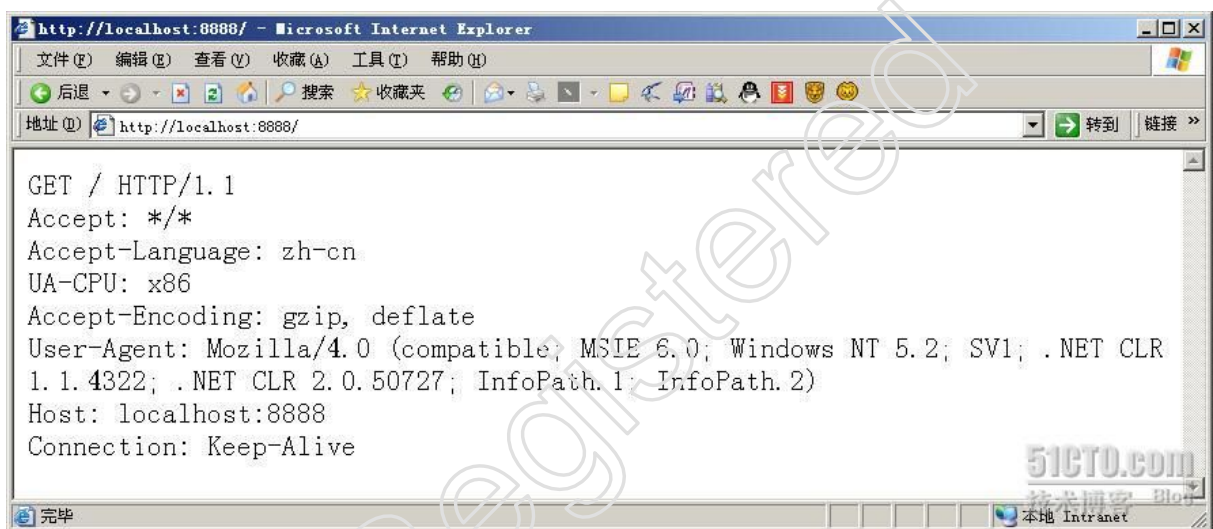


图 1

上面的代码并未验证 HTTP 请求类型，因此 GET、POST、HEAD 等 HTTP 请求都可以得到回应。在接收客户端请求后，只向客户端输出了行 HTTP 响应头信息（包括响应码和 HTTP 版本号），对于 HTTP 响应头来说，这一行是必须有的，其他的头字段都是可选的。上面的代码每读一行请求头信息，就向客户端写一行响应信息。最后使用了 flush 方法将输出缓冲区中的内容发送给客户端。这是必须的，只要使用 OutputStream，在最后就必须调用 flush 方法 (在 Socket 类中使用 OutputStream 也是一样)。

## 27 关闭服务端连接

在客户端和服务端的数据交互完成后，一般需要关闭网络连接。对于服务端来说，需要关闭服务端建立的 Socket 和 ServerSocket。

在关闭 Socket 后，客户端并不会马上感知自己的 Socket 已经关闭，也就是说，在服务端的 Socket 关闭后，客户端的 Socket 的 isClosed 和 isConnected 方法仍然会分别得到 false 和 true。但对已关闭的 Socket 的输入输出流进行操作会抛出一个 SocketException 异常。

在关闭服务端的 ServerSocket 后，ServerSocket 对象所绑定的端口被释放。这时客户端将无法连接服务端程序。下面的代码演示了在服务端关闭 Socket 后，客户端是所何反应的。

```

package server;

import java.net.*;

class Client
{
    public static void main(String[] args) throws Exception
    {
        Socket socket = new Socket("127.0.0.1", 1234);
        Thread.sleep(1000);
        // socket.getOutputStream().write(1);
        System.out.println("read() = " + socket.getInputStream().read());
        System.out.println("isConnected() = " + socket.isConnected());
        System.out.println("isClosed() = " + socket.isClosed());
    }
}

public class CloseSocket
{
    public static void main(String[] args) throws Exception
    {
        ServerSocket serverSocket = new ServerSocket(1234);
        while (true)
        {
            Socket socket = serverSocket.accept();
            socket.close();
        }
    }
}

```

#### 测试:

执行下面的命令

```
java server.CloseSocket
```

```
java server.Client
```

#### 运行结果:

```
read() = -1
```

```
isConnected() = true
```

```
isClosed() = false
```

从上面的运行结果可以看出例程 Client 并未抛出 SocketException 异常。而在 012 行的 read 方法返回了 -1。如果将 socket.close 去掉，客户端的 read 方法将处于阻塞状态。这是因为 Java 在发现无法从服务端的 Socket 得到数据后，就通过 read 方法返回了 -1。如果将 011 行的注释去掉，Client 就会抛出一个 SocketException 异常。大家可以试试，并将 socket.close 行改成 serverSocket.close 后，客户端就会抛出连接异常：

```
Exception in thread "main" java.net.SocketException: Connection reset
at java.net.SocketInputStream.read(SocketInputStream.java:168)
```

```
at java.net.SocketInputStream.read(SocketInputStream.java:182)
at chapter5.Client.main(CloseSocket.java:12)
```

显式地调用 `close` 方法关闭 `ServerSocket` 并不是必须的，在程序退出时将自动关闭 `ServerSocket`。但通过 `close` 方法关闭 `ServerSocket`，可以让给其他的 `ServerSocket` 对象绑定该端口。可以使用 `ServerSocket` 类的 `isClosed` 和 `isBound` 方法判断 `ServerSocket` 是否处于活动状态，如下面的代码所示：

```
ServerSocket serverSocket = new ServerSocket(1234);
if (serverSocket.isBound() == true && serverSocket.isClosed() == false)
    System.out.println("serverSocket 处于活动状态!");
else
    System.out.println("serverSocket 处于非活动状态!");
```

上面代码所示的“非活动状态”可能是 `serverSocket` 对象已经关闭，也可能是 `serverSocket` 对象是使用 `ServerSocket` 类的默认构造方法创建的，而且未调用 `bind` 方法绑定端口。在这里要注意的是 `isBound` 方法返回 `true` 并不意味着 `serverSocket` 对象处于活动状态，调用 `close` 方法并不会将绑定状态置为 `false`。这一点和 `Socket` 类的 `isConnected` 方法类似。

## 28 获取 ServerSocket 信息的方法及 FTP 原理

与 `ServerSocket` 对象相关的信息有两个：绑定端口和绑定 IP 地址。绑定端口可以通过 `getLocalPort` 方法获得。绑定 IP 地址可以通过 `getInetAddress` 方法获得。

### 一、getLocalPort 方法

`getLocalPort` 方法的返回值可分为以下三种情况：

1. `ServerSocket` 对象未绑定端口，`getLocalPort` 方法的返回值为 -1。
2. `ServerSocket` 对象绑定了一个固定的端口，`getLocalPort` 方法返回这个固定端口。
3. `ServerSocket` 对象的绑定端口为 0，`getLocalPort` 方法返回一个随机的端口（这类端口被称为匿名端口）。

`getLocalPort` 方法的定义如下：

```
public int getLocalPort()
```

`getLocalPort` 方法主要是为这些匿名端口而准备的。下面的代码演示了 `ServerSocket` 对象产生随机端口的过程：

```
package server;

import java.net.*;

public class RandomPort
{
    public static void main(String[] args) throws Exception
    {
        for (int i = 1; i <= 5; i++)
        {
            System.out.print("Random Port" + i + ": ");
            System.out.println(new ServerSocket(0).getLocalPort());
        }
    }
}
```

```
}  
}
```

运行结果：

Random Port1: 1397

Random Port2: 1398

Random Port3: 1399

Random Port4: 1400

Random Port5: 1401

在大多数时候 `ServerSocket` 对象都会绑定一个固定的端口。但有时客户端只需要和服务端进行短暂的连接，这时就可以使用匿名端口。如我们经常用的 FTP 服务就是如此。

FTP 服务器一般分为两种工作模式：主动模式（Port 模式）和被动模式（PASV 模式）。在这里主动和被动都是指 FTP 服务器。

## 主动模式

在主动模式中，FTP 服务器绑定了两个端口：21 和 20（这两个端口是默认值，可以设成别的端口）。其中 21 端口负责客户端和服务端之间的命令传送。一开始，由客户端主动连接服务端的 21 端口，并且向服务器发送相应的 FTP 命令。另外一个端口 20 是负责客户端和服务端的数据传送。但要注意，并不是客户端主动连接服务端的 20 端口，而是在客户端创建一个使用匿名端口的服务端连接（在 Java 中就是创建一个 `ServerSocket` 对象，并且绑定端口是 0）。然后客户端通过 21 端口将这个匿名端口通知服务端。最后，服务端主动连接客户端的这个匿名端口（所以这种模式叫主动模式，就是服务器主动连接客户端）。图 1 描述主动模式的工作原理。

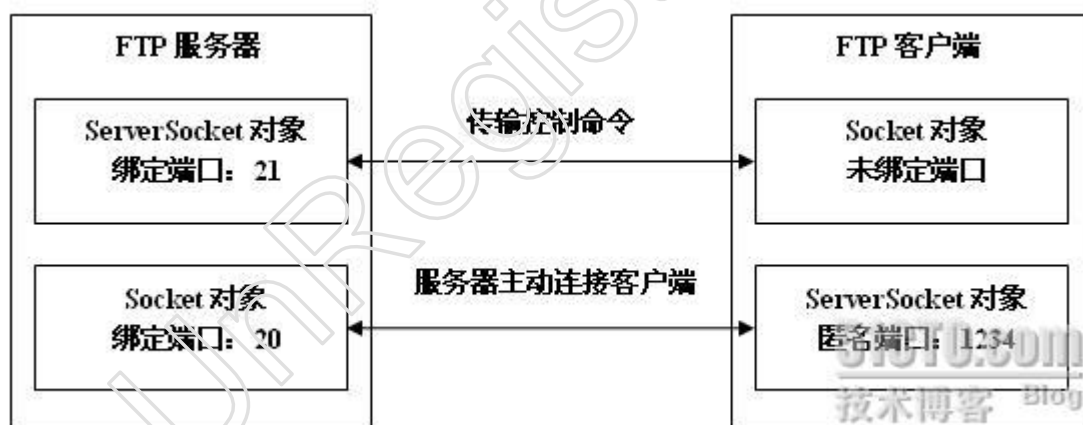


图 1 主动模式的工作原理

从图 1 可以看出，在主动模式中，在传送命令和数据时，他们建立连接的过程是相反的。也就是说，在传送命令时，由客户端主动连接服务器的 21 端口。而传送数据时，由服务器主动连接客户端的匿名端口。这种方式是 FTP 服务器最初的工作模式，但这种模式有很大的局限性。如客户端通过代理上网，而且未做端口映射。在这种情况下，服务端是无法主动和客户端建立连接的。因此，这就产生的另一种模式：被动模式。

## 被动模式

被动模式和主动模式在传送命令的方式上是一样的。它们的区别就在于数据的传输上。被动模式在建立命令传输通道后，服务端建立一个绑定到匿名端口的 `ServerSocket` 对象。并通过命令传输通道将这个匿名端口通知客户端，然后由客户端主动连接服务端的这个匿名端口。这对于服务端就是被动的，因此，这种模式叫被动模式。

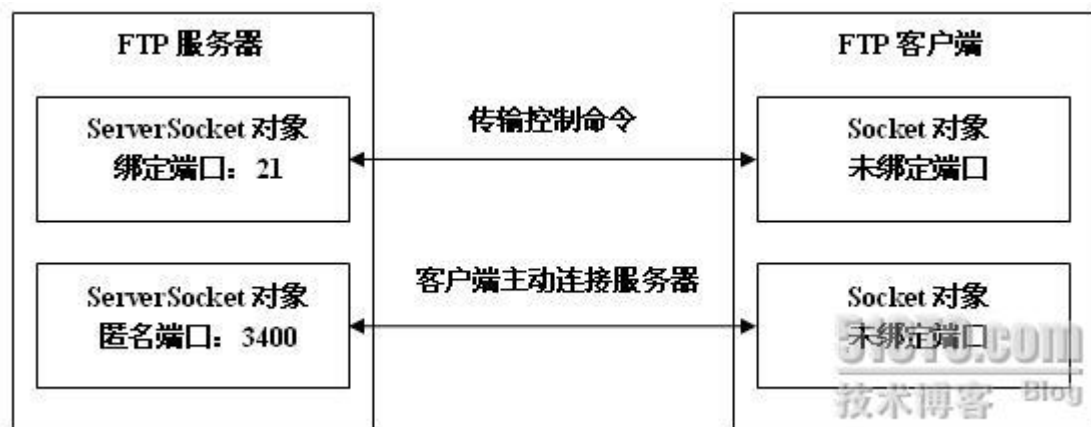


图 2 被动模式的工作原理

现在的大多数 FTP 客户端软件的默认工作模式都是被动模式。因此，这种模式可以克服防火墙等的限制，并且客户端不需要有固定 IP。但这种模式也有它的缺点，这就是在服务端要为客户开大量的端口（大多数 FTP 服务器开的端口范围是 1024 ~ 5000，但有的服务器的范围达到 1024 ~ 65535）。这对于服务器来说存在着一定的安全隐患。因此，如果可能的话，最好还是采用主动模式。

## 二、getInetAddress 方法

getInetAddress 可以得到 ServerSocket 对象绑定的 IP 地址。如果 ServerSocket 对象未绑定 IP 地址，返回 0.0.0.0。

getInetAddress 方法的定义如下：

```
public InetAddress getInetAddress()
```

下面的代码演示了 getInetAddress 的使用方法：

```
ServerSocket serverSocket = new ServerSocket();
serverSocket.bind(new InetSocketAddress("192.168.18.100", 0));
System.out.println(serverSocket.getInetAddress().getHostAddress());
```

运行结果：

```
192.168.18.100
```

## 三、getLocalSocketAddress 方法

这个方法其实是将 getLocalPort 和 getInetAddress 方法的功能集成到了一起。也就是说，使用 getLocalSocketAddress 方法可以同时得到绑定端口和绑定 IP 地址。这个方法返回了一个 SocketAddress 对象。SocketAddress 类是一个抽象类，要想分别得到端口和 IP 地址，必须将 SocketAddress 对象转换成 InetSocketAddress 对象（InetSocketAddress 类是从 SocketAddress 类继承的）。getLocalSocketAddress 方法的定义如下：

```
public SocketAddress getLocalSocketAddress()
```

下面的代码演示了 getLocalSocketAddress 的使用方法。

```
ServerSocket serverSocket = new ServerSocket();
serverSocket.bind(new InetSocketAddress("192.168.18.100", 1234));
System.out.println(serverSocket.getLocalSocketAddress());
InetSocketAddress nsa = (InetSocketAddress)serverSocket.getLocalSocketAddress();
System.out.println( nsa.getAddress().getHostAddress());
System.out.println( nsa.getPort());
```

运行结果：



```
/192.168.18.100:1234
192.168.18.100
1234
```

## 29 服务端 Socket 的选项

ServerSocket 类有以下三个选项：

1. `SO_TIMEOUT`：设置 `accept` 方法超时时间。
2. `SO_REUSEADDR`：设置服务端同一个端口是否可以多次绑定。
3. `SO_RECVBUF`：设置接收缓冲区的大小。

### 一、SO\_TIMEOUT 选项

可以通过 `ServerSocket` 类的两个方法(`setSoTimeout` 和 `getSoTimeout`)来设置和获得 `SO_TIMEOUT` 选项的值，这两个方法的定义如下：

```
public synchronized void setSoTimeout(int timeout) throws SocketException
public synchronized int getSoTimeout() throws IOException
```

`setSoTimeout` 方法的 `timeout` 参数表示 `accept` 方法的超时时间，单位是毫秒。在通常情况下，`ServerSocket` 类的 `accept` 方法在等待客户端请求时处于无限等待状态。如 HTTP 服务器在没有用户访问网页时会一直等待用户的请求。一般不需要对服务端设置等待客户端请求超时，但在某些特殊情况下，服务端规定客户端必须在一定时间内向服务端发出请求，这时就要设置等待客户端请求超时，也就是 `accept` 方法的超时时间。当设置客户端请求超时后，`accept` 方法在等待超时时间后抛出一个 `SocketTimeoutException` 异常。下面的代码演示了如何设置和获得 `SO_TIMEOUT` 选项的值，超时时间通过命令行参数方式传入 `AcceptTimeout`。

```
package server;

import java.net.*;

public class AcceptTimeout
{
    public static void main(String[] args) throws Exception
    {
        if (args.length == 0)
            return;
        ServerSocket serverSocket = new ServerSocket(1234);
        int timeout = Integer.parseInt(args[0]);

        serverSocket.setSoTimeout(Integer.parseInt(args[0]));
        System.out.println((timeout > 0) ? "accept 方法将在"
            + serverSocket.getSoTimeout() + "毫秒后抛出异常！" : "accept 方法永远阻塞！");
        serverSocket.accept();
    }
}
```

执行下面的命令：

```
java server.AcceptTimeout 3000
```

运行结果：

accept 方法将在 3000 毫秒后抛出异常！

```
Exception in thread "main" java.net.SocketTimeoutException: Accept timed out
    at java.net.PlainSocketImpl.socketAccept(Native Method)
    at java.net.PlainSocketImpl.accept(PlainSocketImpl.java:384)
    at java.net.ServerSocket.implAccept(ServerSocket.java:450)
    at java.net.ServerSocket.accept(ServerSocket.java:421)
    at chapter5.AcceptTimeout.main(AcceptTimeout.java:16)
```

setSoTimeout 方法可以在 ServerSocket 对象绑定端口之前调用，也可以在绑定端口之后调用。

如下面的代码也是正确的：

```
ServerSocket serverSocket = new ServerSocket();
serverSocket.setSoTimeout(3000);
serverSocket.bind(new InetSocketAddress(1234));
```

## 二、SO\_REUSEADDR 选项

SO\_REUSEADDR 选项决定了一个端口是否可以被绑定多次。可以通过 ServerSocket 类的两个方法(setReuseAddress 和 getReuseAddress)来设置和获得 SO\_REUSEADDR 选项的值，这两个方法的定义如下：

```
public void setReuseAddress(boolean on) throws SocketException
public boolean getReuseAddress() throws SocketException
```

在大多数操作系统中都不允许一个端口被多次绑定。如果一个 ServerSocket 对象绑定了已经被占用的端口，那么 ServerSocket 的构造方法或 bind 方法就会抛出一个 BindException 异常。

Java 提供这个选项的主要目的是为了防止由于频繁绑定释放一个固定端口而使系统无法正常工作。当 ServerSocket 对象关闭后，如果 ServerSocket 对象中仍然有未处理的数据，那么它所绑定的端口可能在一段时间内不会被释放。这就会造成其他的 ServerSocket 对象无法绑定这个端口。在设置这个选项时，如果某个端口是第一次被绑定，无需调用 setReuseAddress 方法，而再次绑定这个端口时，必须使用 setReuseAddress 方法将这个选项设为 true。而且这个方法必须在调用 bind 方法之前调用。下面的代码演示了如何设置和获得这个选项的值：

```
package server;

import java.net.*;

public class TestReuseAddr1
{
    public static void main(String[] args) throws Exception
    {
        ServerSocket serverSocket1 = new ServerSocket(1234);
        System.out.println(serverSocket1.getReuseAddress());

        ServerSocket serverSocket2 = new ServerSocket();
        serverSocket2.setReuseAddress(true);
        serverSocket2.bind(new InetSocketAddress(1234));
```

```

        ServerSocket serverSocket3 = new ServerSocket();
        serverSocket3.setReuseAddress(true);
        serverSocket3.bind(new InetSocketAddress(1234));
    }
}

```

运行结果：false

在上面代码中第一次绑定端口 1234，因此，serverSocket1 对象无需设置 SO\_REUSEADDR 选项（这个选项在大多数操作系统上的默认值是 false）。而 serverSocket2 和 serverSocket3 并不是第一次绑定端口 1234，因此，必须设置这两个对象的 SO\_REUSEADDR 值为 true。在设置 SO\_REUSEADDR 选项时要注意，必须在 ServerSocket 对象绑定端口之前设置这个选项。

也许有的读者可能有这样的疑问。如果多个 ServerSocket 对象同时绑定到一个端口上，那么当客户端向这个端口发出请求时，该由哪个 ServerSocket 对象来接收客户端请求呢？在给出答案之前，让我们先看看下面的代码的输出结果是什么。

```

package server;

import java.net.*;

public class TestReuseAddr2 extends Thread
{
    String s;
    public void run()
    {
        try
        {
            ServerSocket serverSocket = new ServerSocket();
            serverSocket.setReuseAddress(true);
            serverSocket.bind(new InetSocketAddress(1234));
            Socket socket = serverSocket.accept();
            System.out.println(s + ":" + socket);
            socket.close();
            serverSocket.close();
        }
        catch (Exception e)
        {
        }
    }
    public TestReuseAddr2(String s)
    {
        this.s = s;
    }
    public static void main(String[] args)
    {

```

```

        for (int i = 1; i <= 5; i++)
            new TestReuseAddr2("ServerSocket" + i).start();
    }
}

```

执行下面的命令：

```
java server.TestReuseAddr2
```

连续执行 5 次下面的命令：

```
telnet localhost 1234
```

执行结果：

```

ServerSocket1: Socket[addr=/127.0.0.1,port=11724,localport=1234]
ServerSocket3: Socket[addr=/127.0.0.1,port=11725,localport=1234]
ServerSocket5: Socket[addr=/127.0.0.1,port=11726,localport=1234]
ServerSocket2: Socket[addr=/127.0.0.1,port=11727,localport=1234]
ServerSocket4: Socket[addr=/127.0.0.1,port=11728,localport=1234]

```

上面的运行结果只是一种可能，如果多次按着上面的步骤操作，可能得到不同的运行结果。由此可以断定，当多个 `ServerSocket` 对象同时绑定一个端口时（关闭之前一直是他），系统会随机选择一个 `ServerSocket` 对象来接收客户端请求。但要注意，这个接收客户端请求的 `ServerSocket` 对象必须关闭（如 019 行如示），才能轮到其他的 `ServerSocket` 对象接收客户端请求。如果不关闭这个 `ServerSocket` 对象，那么其他的 `ServerSocket` 对象将永远无法接收客户端请求。读者可以将 `serverSocket.close()` 去掉，再执行上面操作步骤，看看会有什么结果。

### 三、SO\_RCVBUF 选项

可以通过 `ServerSocket` 类的两个方法 (`setReceiveBufferSize` 和 `getReceiveBufferSize`) 来设置和获得 `SO_RCVBUF` 选项的值，这两个方法的定义如下：

```

public synchronized void setReceiveBufferSize (int size) throws SocketException
public synchronized int getReceiveBufferSize() throws SocketException

```

其中 `size` 参数表示接收缓冲区的大小，单位是字节。设置了 `ServerSocket` 类的 `SO_RCVBUF` 选项，就相当于设置了 `Socket` 对象的接收缓冲区大小。这个是指由 `accept` 返回的 `Socket` 对象。下面代码演示了如何使用这两个方法来设置和获得接收缓冲区的大小：

```

package server;

import java.net.*;

public class TestReceiveBufferSize
{
    public static void main(String[] args) throws Exception
    {
        ServerSocket serverSocket = new ServerSocket(1234);
        serverSocket.setReceiveBufferSize(2048); // 将接收缓冲区设为 2K
        while (true)
        {
            Socket socket = serverSocket.accept();

```

```

// 如果客户端请求使用的是本地 IP 地址，重新将 Socket 对象的接收缓冲区设为 1K
if (socket.getInetAddress().isLoopbackAddress())
    socket.setReceiveBufferSize(1024);
System.out.println("serverSocket:"
    + serverSocket.getReceiveBufferSize());
System.out.println("socket:" + socket.getReceiveBufferSize());
socket.close();
}
}
}

```

执行如下命令：

```
java server.TestReceiveBufferSize
```

执行如下三个命令（192.168.18.100 为本机 IP 地址）：

```
telnet 192.168.18.100 1234
```

```
telnet localhost 1234
```

```
telnet 192.168.18.100 1234
```

运行结果：

```
serverSocket: 2048
```

```
socket: 2048
```

```
serverSocket: 2048
```

```
socket: 1024
```

```
serverSocket: 2048
```

```
socket: 2048
```

从上面的运行结果可以看出，在执行 `telnet localhost 1234` 命令后，由于 `localhost` 是本地地址，因此程序通过将 `Socket` 对象的接收缓冲区设为 1024，而在执行其他两条命令后，由于 192.168.18.100 不是本机地址，所以 `Socket` 对象的接收缓冲区仍然保留着 `serverSocket` 的值：2048。因此，我们可以得出一个结论，设置 `ServerSocket` 对象的接收缓冲区就相当于设置了所有从 `accept` 返回的 `Socket` 对象的接收缓冲区，只要不单独对某个 `Socket` 对象重新设置，这些 `Socket` 对象的接收缓冲区就会都保留 `ServerSocket` 对象被设置的这个值。

无论在 `ServerSocket` 对象绑定到端口之前还是之后设置 `SO_RCVBUF` 选项都有效，但如果要设置大于 64K 的接收缓冲区时，就必须在 `ServerSocket` 对象绑定端口之前设置 `SO_RCVBUF` 选项。如下面的代码将接收缓冲区的大小设为 100K。

```
ServerSocket serverSocket = new ServerSocket();
```

```
serverSocket.setReceiveBufferSize(100 * 1024); // 将接收缓冲区的大小设为 100K。
```

```
serverSocket.bind(new InetSocketAddress(1234));
```

一般情况下，并不需要设置这个选项，它的默认值（一般为 8K）足可以满足大多数情况。但有时为了适应特殊的需要，必须更改接收缓冲区的值。如在一些网络游戏中，需要实时地向服务器传送各种动作、指令信息。这就需要将接收缓冲区设小一点。这样可以在一定程度上增加游戏客户端的灵敏度。如果需要传送大量的数据，如 HTTP、FTP 等协议。这就需要较大的接收缓冲区。

#### 四、设置 ServerSocket 的性能偏好

在 Java SE5.0 及以上版本中为 `ServerSocket` 类增加了一个 `setPerformancePreferences` 方法。这个方法的方法和 `Socket` 类中的 `setPerformancePreferences`(此前还未涉及)的作用一样, 用来设置连接时间、延迟和带宽的相对重要性。`setPerformancePreferences` 方法的定义如下:

```
public void setPerformancePreferences(int connectionTime, int latency, int bandwidth)
```

## 30 定制 Accept 方法

使用 `ServerSocket` 类的 `implAccept` 方法可以使用 `accept` 方法返回一个 `Socket` 子类对象(当然也可以直接使用 `accept` 返回 `Socket` 对象)。由于 `implAccept` 是 `protected` 方法, 在 `ServerSocket` 类的子类中只能覆盖(重写)`accept` 方法, 然后在 `accept` 方法中再使用 `implAccept` 方法重新设置 `Socket` 对象。

`implAccept` 方法的定义如下:

```
protected final void implAccept(Socket s) throws IOException
```

只要通过 `implAccept` 方法设置一个未连接的 `Socket` 子类对象, `accept` 方法就会返回一个已经连接的 `Socket` 子类对象(`accept` 返回的是 `Socket` 对象, 要想得到 `Socket` 子类对象, 必须进行类型转换)。在大多数时候不需要改变 `accept` 方法的行为, 但有时需要一个有更多特性的 `Socket` 类, 通过 `implAccept` 方法(他里面可以调用原始的 `accept` 方法)就可以达到这个目的。

下面的代码自定义了一个从 `Socket` 继承的 `HttpSocket` 类, 这个类除了具有 `Socket` 类的一切特性外, 还增加了一个方法 `getRequestHeaders`, 用于返回 HTTP 请求的头信息。由于定制的 `socket` 子类有了新功能, 新的 `serversocket` 也需要反映出来。

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

class HttpSocket extends Socket // Socket 的子类
{
    public String getRequestHeaders() throws Exception
    {
        InputStreamReader isr = new InputStreamReader(getInputStream());
        BufferedReader br = new BufferedReader(isr);
        String s = "", result = "";
        while (!(s = br.readLine()).equals(""))
            result = result + s + "\r\n";
        return result;
    }
}

class HttpServerSocket extends ServerSocket // ServerSocket 的子类
{
    public HttpServerSocket(int port) throws IOException
    {
        super(port);
    }
}
```



```

public Socket accept() throws IOException // 覆盖/重写 accept 方法
{
    Socket s = new HttpSocket();
    implAccept(s); // 将 accept 方法返回的对象类型设为 HttpSocket
    return s; // implAccept()作用见本文第一行
}
}
public class CustomAccept
{
    public static void main(String[] args) throws Exception
    {
        HttpServerSocket httpServerSocket = new HttpServerSocket(1234);
        HttpSocket httpSocket = (HttpSocket) httpServerSocket.accept();
        System.out.println(httpSocket.getRequestHeaders()); // 向控制台输出 HTTP 请求头
        httpServerSocket.close();
    }
}

```

测试

执行如下命令：

```
java server.CustomAccept
```

在 IE 的地址栏中输入如下 Url：

```
http://localhost:1234
```

CustomAccept 在控制台中的运行结果：

```
GET / HTTP/1.1
```

```
Accept: */*
```

```
Accept-Language: zh-cn
```

```
UA-CPU: x86
```

```
Accept-Encoding: gzip, deflate
```

```
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.2; SV1; .NET CLR 1.1.4322; .NET CLR 2.0.50727; InfoPath.1; InfoPath.2)
```

```
Host: localhost:1234
```

```
Connection: Keep-Alive
```

上面的运行结果就是 IE 向服务端发出的 HTTP 请求头的内容。这个运行结果会根据客户机配置的不同而有所差异。

## 31 非阻塞 I/O 简介

在网络应用中，一般可以采用同步 I/O（阻塞 I/O）和非阻塞 I/O 两种方式进行数据通讯。这两种方式并非互相排斥和互相取代。我们可以在平时的应用中单独采用其中一种通讯方式，也可以混合使用这两种通讯方式。在本文中就什么是非阻塞 I/O 以及为什么要使用这种通讯方式进行了介绍，在下一篇文章中给出了一个简单的例子来演示在网络应用中如何使用非阻塞 I/O 进行通讯。

## 一、什么是非阻塞 I/O

我们可以将**同步 I/O**称为**阻塞 I/O**，**非阻塞 I/O**称为**异步 I/O**。在本书中采用了比较常用的叫法：同步 I/O 和非阻塞 I/O。虽然它们的叫法不同，但含义是一样的。读者在阅读其他书时应注意这一点。

在讲解什么是非阻塞 I/O 之前，首先应了解什么是同步 I/O，这里的同步指的是什么。同步这个概念在程序设计中主要是指代码按顺序执行的过程。如在 Java 程序中的 main 方法，如果不使用多线程，这个方法中的代码一定是从前往后**按顺序执行的，这就叫做同步**。如果使用了多线程，从宏观角度来看会有不同的代码段同时执行，这就叫做异步。在同步 I/O 中的同步概念也类似，也就是说，在 I/O 通讯过程中，只要是某一步的通讯未结束，就无法进行其他的通讯。那么这里的同步指的是什么呢？要回答这个问题之前，首先让我们先回忆一下，在网络通讯中有哪些地方可能会被阻塞。如果读者看了前面的章节就会知道答案。**对于客户端来说，有两个地方可能会被阻塞：连接服务器（调用 connect 方法时）和读写数据。而在服务端也有两个地方可能会被阻塞：等待客户端请求（调用 accept 方法时）和读写数据**（在一般情况下，写数据不会被阻塞，但如果网络环境比较差的时候，客户端和服务端的写数据操作也可能发生阻塞现象）。也就是说，可以设置超时时间的地方就可能被阻塞。

而同步 I/O 中的同步就是指除了以下两种情况外程序会一直处于等待状态：

1. 连接服务器、读写数据或等待客户端请求正常地执行。
2. 在等待超时时间后，抛出了超时异常。

在上面我们了解了什么是同步 I/O。而非阻塞 I/O 和同步 I/O 最明显的不同就是同步 I/O 所有可能被阻塞的地址在非阻塞 I/O 中都不会被阻塞。如在读取数据时，如果数据暂时无法被读取。那么在非阻塞 I/O 中会立刻返回，以便程序可以执行其他的代码，然后系统会不断侦测这个未完成的读取操作，直到可以继续读数据时再来完成这个操作。

Java 在 JDK1.4 及以后版本中提供了一套 API 来专门操作非阻塞 I/O，我们可以在 java.nio 包及其子包中找到相关的类和接口。由于这套 API 是 JDK 新提供的 I/O API，因此，也叫 New I/O，这就是包名 ni o 的由来。这套 API 由三个主要的部分组成：缓冲区(Buffers)、通道(Channels)、非阻塞 I/O 核心类。

这三部分的详细内容将在本章的后面介绍。

## 二、为什么要使用非阻塞 I/O

在使用同步 I/O 的网络应用中，如果要同时处理多个客户端请求，或是在客户端要同时和多个服务器进行通讯，就必须**使用多线程**来处理。也就是说，将每一个客户端请求分配给一个线程来单独处理。这样做虽然可以达到我们的要求，但同时又会带来另外一个问题。由于每创建一个线程，就要为这个线程分配一定的内存空间（也叫工作存储器），而且操作系统本身也对线程的总数有一定的限制。如果客户端的请求过多，服务端程序可能会因为不堪重负而拒绝客户端的请求，甚至服务器可能会因此而瘫痪。

当然，**也可以使用线程池**（将在第三部分讲解）来缓解服务器的压力，但这并不能解决客户端因访问过于密集而造成的服务器拒绝响应的问题。虽然在服务端还有**请求缓冲区**作为保障，但这个缓冲区的大小是有限的（一般为 50），如果客户端的请求数远超过这个数，客户端还是会收到拒绝服务的信息。

**在这种情况下，使用非阻塞 I/O 就可以解决这个问题**。由于使用非阻塞 I/O 的程序一般是单线程的（有时可能将使用非阻塞 I/O 的程序段放到一个单独的线程里，而主线程负责处理用户的输入），因此，服务端接收的客户端请求数并不随着工作线程数的增加而增加。所以使用非阻塞 I/O 模式就不会受到操作系统对线程总数的限制，也不会占用大量的服务器资源。

非阻塞 I/O 虽然可以到达在处理大量客户端请求的同时，又不占用大量的服务器资源的目的。但这种通讯方式并**不能完全取代同步 I/O**。**如非阻塞 I/O 并不适合象 FTP 服务器那样需要保持连接状态的应用（原因将在以后的章节中说明）**。非阻塞 I/O 一般应用在服务端比较多一些，因为客户端一般并不需要处理大量的连接（但某些应用除外，如象百度、Google 的 Web Spider，需要同时下载多个网页，这时就需要在客户端建

立大量的连接来满足需求），而服务端程序一般需要接收并处理大量的客户端请求，因此，就需要使用多线程（使用同步 I/O）或非阻塞 I/O 来达到这个目的。如果某个服务端应用处理的客户端请求没那么多时，使用多线程和同步 I/O 可能会更好一点，因为这种方式要比非阻塞 I/O 方式更灵活。

在前面一直将非阻塞 I/O 和网络应用放到一起讲。其实我们也可以将非阻塞 I/O 应用到非网络的应用中，如文件复制。由于同步 I/O 是基于字节流的，而非阻塞 I/O 是基于缓冲区和通道的。因此，从理论上，所操作的文件越大，非阻塞 I/O 的优势越能体现出来。而对于比较小的文件操作，这两种方式的效率差不多。根据实验得知，复制一个 4G 左右的文件，一般情况下，非阻塞 I/O 方式比同步 I/O 方式快大约 15% 左右。

## 32 一个非阻塞 I/O 的例子

为了使读者更好地理解非阻塞 I/O，本节给出了一个简单的例子用来演示如何将非阻塞 I/O 应用到网络程序中。读者可以先不必管这个例子的具体细节。因为这个例子的主要目的并不是讲解非阻塞 I/O 的使用，而是先让读者对非阻塞 I/O 有一个笼统的感性认识。在看完这个例子后，读者可能会有很多疑问，在本章后面的部分将会逐渐揭开这些迷团。

这个例子的主要功能是访问新浪网，并将新浪网的首页在控制台上输出。

```
package test;

import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class FirstNonBlockingIO
{
    public static void main(String[] args) throws Exception
    {
        SocketAddress remote = new InetSocketAddress("www.sina.com.cn", 80);
        SocketChannel channel = SocketChannel.open(remote);
        String request = "GET / HTTP/1.1\r\n" +
            "Host: www.sina.com.cn\r\n" +
            "Connection: close\r\n\r\n";
        ByteBuffer header = ByteBuffer.wrap(request.getBytes());
        channel.write(header);
        ByteBuffer buffer = ByteBuffer.allocate(1024);
        WritableByteChannel out = Channels.newChannel(System.out);
        while (channel.read(buffer) != -1)
        {
            buffer.flip();
            out.write(buffer);
            buffer.clear();
        }
        channel.close();
    }
}
```

```
}  
}
```

测试

执行如下命令：

```
java test.FirstNonBlockingIO > sina.txt
```

打开 sina.txt 后，会看到如下的文件内容：

```
HTTP/1.0 200 OK  
Date: Sun, 01 Apr 2007 06:53:50 GMT  
Server: Apache/2.0.58 (Unix)  
Last-Modified: Sun, 01 Apr 2007 06:50:47 GMT  
Connection: close  
</body>  
</html>
```

由于新浪网的主页内容太多，因此，为了方便查看程序运行结果，使用输出重定向符“>”将本该输出到控制台的内容输出到 sina.txt 文件中。从例程 7-1 可以看出，主要有三点和同步 I/O 存在差异。

1. 连接服务器（第 013 行）。使用 SocketChannel 类，而不是 Socket 类。
2. 向服务端写数据（第 018 行）。使用 SocketChannel 类中的 write 方法，而不是 OutputStream。
3. 从服务端读数据（第 021 行）。使用 SocketChannel 类中的 read 方法，而不是 InputStream。

除了上面的三点外，在本例中还使用了缓冲区来处理输入输出数据。因此，通道(Channels)和缓冲区(Buffers)是学习非阻塞 I/O 之前必须掌握的知识。在下面的文章等将详细讲解这两部分的内容。

## 33 非阻塞 I/O 的缓冲区

如果将同步 I/O 方式下的数据传输比做数据传输的零星方式（这里的零星是指在数据传输的过程中是以零星的字节方式进行的），那么就可以将非阻塞 I/O 方式下的数据传输比做数据传输的集装箱方式（在字节和低层数据传输之间，多了一层缓冲区，因此，可以将缓冲区看做是装载字节的集装箱）。大家可以想象，如果我们要运送比较少的货物，用集装箱好象有点不太合算，而如果要运送上百吨的货物，用集装箱来运送的成本会更低。

在数据传输过程中也是一样，如果数据量很小时，使用同步 I/O 方式会更适合，如果数据量很大时（一般以 G 为单位），使用非阻塞 I/O 方式的效率会更高。因此，从理论上说，数据量越大，使用非阻塞 I/O 方式的单位成本就会越低。产生这种结果的原因和缓冲区的一些特性有着直接的关系。

在本节中，将对缓冲区的一些主要特性进行讲解，使读者可以充分理解缓冲区的概念，并能通过缓冲区来提高程序的执行效率。

### 缓冲区类创建缓冲区

Java 提供了七个基本的缓冲区，分别由七个类来管理，它们都可以在 java.nio 包中找到。这七个类如下所示：

- ByteBuffer
- ShortBuffer

- IntBuffer
- CharBuffer
- FloatBuffer
- DoubleBuffer
- LongBuffer

这七个类中的方法类似，只是它们的返回值或参数和相应的简单类型相对应，如 ByteBuffer 类的 get 方法返回了 byte 类型的数据，而 put 方法需要一个 byte 类型的参数。在 CharBuffer 类中的 get 和 put 方法返回和传递的数据类型就是 char。

这七个类都没有 public 构造方法，因此，它们不能通过 new 来创建相应的对象实例。这些类都可以通过 allocate 或者 wrap 两种方法来创建相应的对象实例。

#### 1. 通过静态方法 allocate 来创建缓冲区。

这七类都有一个静态的 allocate 方法，通过这个方法可以创建有最大容量限制的缓冲区对象。allocate 的定义如下：

**ByteBuffer 类中的 allocate 方法：**

```
public static ByteBuffer allocate(int capacity)
```

**IntBuffer 类中的 allocate 方法：**

```
public static IntBuffer allocate(int capacity)
```

其他五个缓冲区类中的 allocate 方法定义和上面的定义类似，只是返回值的类型是相应的缓冲区类。

allocate 方法有一个参数 capacity，用来指定缓冲区容量的最大值。capacity 的值不能小于 0，否则会抛出一个 IllegalArgumentException 异常。使用 allocate 来创建缓冲区，并不是一下子就分配给缓冲区 capacity 大小的空间，而是根据缓冲区中存储数据的情况来动态分配缓冲区的大小（实际上，在低层 Java 采用了数据结构中的堆来管理缓冲区的大小），因此这个 capacity 可以是一个很大的值，如 1024\*1024（1 M）。allocate 的使用方法如下：

```
ByteBuffer byteBuffer = ByteBuffer.allocate(1024);
```

```
IntBuffer intBuffer = IntBuffer.allocate(1024);
```

在使用 allocate 创建缓冲区时应用注意，capacity 的含义随着缓冲区类不同而不同。如创建字节缓冲区时，capacity 指的是字节数。而在创建整型(int)缓冲区时，capacity 指的是 int 型值的数目，如果转换成字数，capacity 的值应该乘 4。如上面代码中的 intBuffer 缓冲区最大可容纳的字节数是 1024\*4 = 4096 个。

#### 2. 通过静态方法 wrap 来创建缓冲区。

使用 allocate 方法可以创建一个空的缓冲区。而 wrap 方法可以利用已经存在的数据来创建缓冲区。wrap 方法可以将数组直接转换成相应类型的缓冲区。wrap 方法有两种重载形式，它们的定义如下：

**ByteBuffer 类中的 wrap 方法：**

```
public static ByteBuffer wrap(byte[] array)
```

```
public static ByteBuffer wrap(byte[] array, int offset, int length)
```

**IntBuffer 类中的 wrap 方法：**

```
public static IntBuffer wrap(int [] array)
```

```
public static IntBuffer wrap(int[] array, int offset, int length)
```

其他五个缓冲区类中的 wrap 方法定义和上面的定义类似，只是返回值的类型是相应的缓冲区类。

在 wrap 方法中的 array 参数是要转换的数组（如果是其他的缓冲区类，数组的类型就是相应的简单类型，如 IntBuffer 类中的 wrap 方法的 array 就是 int[] 类型）。offset 是要转换的子数组的偏移量，也就是子数组在 array 中的开始索引。length 是要转换的子数组的长度。利用后两个参数可以将 array 数组中的一部分转换成缓冲区对象。它们的使用方法如下：

```
byte[] myByte = new byte[] { 1, 2, 3 };
```

```
int[] myInt = new int[] { 1, 2, 3, 4 };
```



```
ByteBuffer byteBuffer = ByteBuffer.wrap(myByte);
```

```
IntBuffer intBuffer = IntBuffer.wrap(myInt, 1, 2);
```

可以通过缓冲区类的 `capacity` 方法来得到缓冲区的大小。`capacity` 方法的定义如下：

```
public final int capacity()
```

如果使用 `allocate` 方法来创建缓冲区，`capacity` 方法的返回值就是 `capacity` 参数的值。而使用 `wrap` 方法来创建缓冲区，`capacity` 方法的返回值是 `array` 数组的长度，但要注意，**使用 `wrap` 来转换 `array` 的数组时，`capacity` 的长度仍然是原数组的长度**，如上面代码中的 `intBuffer` 缓冲区的 `capacity` 值是 4，而不是 2。除了可以将数组转换成缓冲区外，也可以通过缓冲区类的 `array` 方法将缓冲区转换成相应类型的数组。`IntBuffer` 类的 `array` 方法的定义方法如下（其他缓冲区类的 `array` 的定义类似）：

```
public final int[] array()
```

下面的代码演示了如何使用 `array` 方法将缓冲区转换成相应类型的数组。

```
int[] myInt = new int[] { 1, 2, 3, 4, 5, 6 };
```

```
IntBuffer intBuffer = IntBuffer.wrap(myInt, 1, 3); //数组到缓冲区
```

```
for (int v : intBuffer.array()) //缓冲区到数组
```

```
System.out.print(v + " ");
```

在执行上面代码后，我们发现输出的结果是 1 2 3 4 5 6，而不是 2 3 4。这说明在将子数组转换成缓冲区的过程中实际上是将整个数组转换成了缓冲区，这就是用 `wrap` 包装子数组后，`capacity` 的值仍然是原数组长度的真正原因。在使用 `array` 方法时应注意，在以下两种缓冲区中不能使用 `array` 方法：

1 只读的缓冲区：

如果使用只读缓冲区的 `array` 方法，将会抛出一个 `ReadOnlyBufferException` 异常。

2 使用 `allocateDirect` 方法(下面讲)创建的缓冲区：

如果调用这种缓冲区中的 `array` 方法，将会抛出一个 `UnsupportedOperationException` 异常。

可以通过缓冲区类的 `hasArray` 方法来判断这个缓冲区是否可以使用 `array` 方法，如果返回 `true`，则说明这个缓冲区可以使用 `array` 方法，否则，使用 `array` 方法将会抛出上述的两种异常之一。

**注意：** 使用 `array` 方法返回的数组并不是缓冲区数据的副本。被返回的数组实际上就是缓冲区中的数据，也就是说，`array` 方法只返回了缓冲区数据的引用。当数组中的数据被修改后，缓冲区中的数据也会被修改，反之也是如此。关于这方面内容将在下一节“读写缓冲区中的数据”中详细讲解。

在上述的七个缓冲区类中，`ByteBuffer` 类和 `CharBuffer` 类各自还有另外一种方法来创建缓冲区对象。

## ByteBuffer 类

可以通过 `ByteBuffer` 类的 `allocateDirect` 方法来创建 `ByteBuffer` 对象。`allocateDirect` 方法的定义如下：

```
public static ByteBuffer allocateDirect(int capacity)
```

使用 `allocateDirect` 方法可以一次性分配 `capacity` 大小的连续字节空间。通过 `allocateDirect` 方法来创建具有连续空间的 `ByteBuffer` 对象虽然可以在一定程度上提高效率，**但这种方式并不是平台独立的**。也就是说，在一些操作系统平台上使用 `allocateDirect` 方法来创建 `ByteBuffer` 对象会使效率大幅度提高，而在另一些操作系统平台上，性能会表现得非常差。**而且 `allocateDirect` 方法需要较长的时间来分配内存空间，在释放空间时也较慢**。因此，在使用 `allocateDirect` 方法时应谨慎。

通过 `isDirect` 方法可以判断缓冲区对象（其他的缓冲区类也有 `isDirect` 方法，因为 **`ByteBuffer` 对象可以转换成其他的缓冲区对象**，这部分内容将在后面讲解）是用哪种方式创建的，如果 `isDirect` 方法返回 `true`，则这个缓冲区对象是用 `allocateDirect` 方法创建的，否则，就是用其他方法创建的缓冲区对象。

## CharBuffer 类

我们可以发现，上述的七种缓冲区中并没有字符串缓冲区，而字符串在程序中却是最常用的一种数据类型。不过不要担心，虽然 `java.nio` 包中并未提供字符串缓冲区，但却**可以将字符串通过共同父类转换成字符缓冲**



区（就是 CharBuffer 对象）。在 CharBuffer 类中的 wrap 方法除了上述的两种重载形式外，又多了两种重载形式，它们的定义如下：

```
public static CharBuffer wrap(CharSequence csq)
public static CharBuffer wrap(CharSequence csq, int start, int end)
```

其中 csq 参数表示要转换的字符串，但我们注意到 csq 的类型并不是 String，而是 CharSequence。CharSequence 类 Java 中四个可以表示字符串的类的父类，这四个类是 String、StringBuffer、StringBuilder 和 CharBuffer（注意 *StringBuffer* 和本节讲的缓冲区类一点关系都没有，这个类在 *java.lang* 包中）。也就是说，CharBuffer 类的 wrap 方法可以将这四个类的对象转换成 CharBuffer 对象。

另外两个参数 start 和 end 分别是子字符串的开始索引和结束索引的下一个位置，如将字符串 "1234" 中的 "23" 转换成 CharBuffer 对象的语句如下：

```
CharBuffer cb = CharBuffer.wrap("1234", 1, 3);
```

下面的代码演示了如何使用 wrap 方法将四种不同形式的字符串转换成 CharBuffer 对象。

```
StringBuffer stringBuffer = new StringBuffer("通过 StringBuffer 创建 CharBuffer 对象");
StringBuilder stringBuilder = new StringBuilder("通过 StringBuilder 创建 CharBuffer 对象");
CharBuffer charBuffer1 = CharBuffer.wrap("通过 String 创建 CharBuffer 对象");
CharBuffer charBuffer2 = CharBuffer.wrap(stringBuffer);
CharBuffer charBuffer3 = CharBuffer.wrap(stringBuilder);
CharBuffer charBuffer4 = CharBuffer.wrap(charBuffer1, 1, 3);
```

## 34 读写缓冲区：用 get 和 put 方法顺序读写单个数据

对于缓冲区来说，最重要的操作就是读写操作。缓冲区提供了两种函数方法来读写缓冲区中的数据：get、put 方法和 array 方法。而 get、put 方法可以有三种读写数据的方式：按顺序读写单个数据、在指定位置读写单个数据和读写数据块。除了上述的几种读写数据的方法外，CharBuffer 类还提供了用于专门写字符串的 put 和 append 方法。在本文及后面的文章中将分别介绍这些读写缓冲区的方法。

虽然使用 allocate 方法创建的缓冲区并不是一次性地分配内存空间，但我们可以从用户地角度将一个缓冲区想象成一个长度为 capacity 的数组。当缓冲区创建后，和数组一样，缓冲区的大小（capacity 值）将无法改变，也无法访问缓冲区外的数据。如下面的代码创建了一个大小为 6 的字节缓冲区。

```
ByteBuffer byteBuffer = ByteBuffer.allocate(6);
```

对于 byteBuffer 来说，只能访问属于这个缓冲区的六个字节的数据，如果超过了这个范围，将抛出一个 BufferOverflowException 异常，这是一个运行时错误，因为这个错误只能在程序运行时被发现。

既然缓冲区和数组类似，那么缓冲区也应该象数组一样可以标识当前的位置。缓冲区的 position 方法为我们提供了这个功能。position 方法有两种重载形式，它们的定义如下：

```
public final int position()
public final Buffer position(int newPosition)
```

第一个重载形式用来获取缓冲区的当前位置。在创建缓冲区后，position 的初始值是 0，也就是缓冲区第一个元素的位置。当从缓冲区读取一个元素后，position 的值加 1。我们从这一点可以看出，position 方法返回的位置就是当前可以读取的元素的位置。position 的取值范围从 0 到 capacity - 1。如果 position 的值等于 capacity，说明缓冲区当前已经没有数据可读了。

第二个重载形式可以设置缓冲区的当前位置。参数 newPosition 的取值范围是 0 到 capacity - 1。如果 newPosition 的值超出这个范围，position 方法就会抛出一个 IllegalArgumentException 异常。

在大多数情况下不需要直接控制缓冲区的位置。缓冲区类提供的用于读写数据的方法可以自动地设置缓冲区的当前位置。在缓冲区类中，get 和 put 方法用于读写缓冲区中的数据。get 和 put 方法的定义如下：

ByteBuffer 类的 get 和 put 方法：

```
public abstract byte get()
public abstract ByteBuffer put(byte b)
```

IntBuffer 类的 get 和 put 方法：

```
public abstract int get()
public abstract IntBuffer put(int i)
```

其他五个缓冲区类中的 get 和 put 方法定义和上面的定义类似，只是 get 方法返回相应的数据类型，而 put 方法的参数是相应的数据类型，并且返回值的类型是相应的缓冲区类。

每当 put 方法向缓冲区写入一个数据后，缓冲区的当前位置都会加 1。如果缓冲区的当前位置已经等于 capacity，调用 put 方法就会抛出一个 java.nio.BufferOverflowException 异常。在缓冲区未初赋值的区域将被 0 填充。使用 get 方法可以得到缓冲区当前位置的数据，并使缓冲区的当前位置加 1，所以和 put 方法一样，在缓冲区当前位置等于 capacity 时使用 get 方法也会抛出 java.nio.BufferOverflowException 异常。。

也可以使用 rewind 方法将缓冲区的当前位置设为 0，rewind 方法的定义如下：

```
public final Buffer rewind()
```

缓冲区除了 position 和 capacity 外，还提供了一个标识来限制缓冲区可访问的范围。这个标识就是 limit。limit 和 position 一样，在缓冲区类中也提供了两个重载方法。用于获得和设置 limit 的值。limit 方法的定义如下：

```
public final int limit()
public final Buffer limit(int newLimit)
```

在初始状态下，缓冲区的 limit 和 capacity 值相同。但 limit 和 capacity 的区别是 limit 可以通过 limit 方法进行设置，而 capacity 在创建缓冲区时就已经指定了，并且不能改变。（在上面所讲的 position 方法的新位置参数的取值范围时曾说是  $0 \leq \text{newPosition} < \text{capacity}$ ，其实严格地说，应是  $0 \leq \text{newPosition} < \text{limit}$ ）limit 的其他性质和 capacity 一样。

position 的值等于 limit，就不能访问缓冲区的当前数据，也就是说不能使用 get 和 put 方法。否则将抛出 BufferOverflowException 异常。由于使用 allocate 创建的缓冲区并不是一次性地分配内存空间，因此，可以将缓冲区的 capacity 设为很大的值，如 10M。缓冲区过大可能在某些环境中会使系统性能降低（如在 PDA 或智能插秧机中），这时可以使用 limit 方法根据具体的情况来限定缓冲区的大小。当然，limit 还可以表示缓冲区中实际的数据量，这将在后面讲解。下面的代码演示了如何使用 limit 方法来枚举缓冲区中的数据：

```
while(byteBuffer.position() < byteBuffer.limit())
    System.out.println(byteBuffer.get());
```

我们还可以用 flip 和 hasRemaining 方法来重写上面的代码。flip 方法将 limit 设为缓冲区的当前位置。当 limit 等于 position 时，hasRemaining 方法返回 false，而则返回 true。flip 和 hasRemaining 方法的定义如下：

```
public final Buffer flip()
public final boolean hasRemaining()
```

下面的代码演示了如何使用 hasRemaining 方法来枚举缓冲区中的数据：

```
while(byteBuffer.hasRemaining())
    System.out.println(byteBuffer.get());
```

如果从缓冲区的第一个位置依次使用 put 方法向缓冲区写数据，当写完数据后，再使用 flip 方法。这样 limit 的值就等于缓冲区中实际的数据量了。在网络中传递数据时，可以使用这种方法来设置数据的结束位置。

为了回顾上面所讲内容，下面的代码总结了创建缓冲区、读写缓冲区中的数据、设置缓冲区的 limit 和 position 的方法。

```
package net;
import java.nio.*;
```

```

public class GetPutData
{
    public static void main(String[] args)
    {
        // 创建缓冲区的四种方式
        IntBuffer intBuffer = IntBuffer.allocate(10);
        ByteBuffer byteBuffer = ByteBuffer.allocateDirect(10);
        CharBuffer charBuffer = CharBuffer.wrap("abcdefg");
        DoubleBuffer doubleBuffer = DoubleBuffer.wrap(new double[] { 1.1, 2.2 });

        // 向缓冲区中写入数据
        intBuffer.put(1000);
        intBuffer.put(2000);

        System.out.println("intBuffer 的当前位置: " + intBuffer.position());

        intBuffer.position(1); // 将缓冲区的当前位置设为 1
        System.out.println(intBuffer.get()); // 输出缓冲区的当前数据

        intBuffer.rewind(); // 将缓冲区的当前位置设为 0
        System.out.println(intBuffer.get()); // 输出缓冲区的当前数据

        byteBuffer.put((byte)20);
        byteBuffer.put((byte)33);
        byteBuffer.flip(); // 将 limit 设为 position, 在这里是 2
        byteBuffer.rewind();
        while(byteBuffer.hasRemaining()) // 枚举 byteBuffer 中的数据
            System.out.print(byteBuffer.get() + " ");

        while(charBuffer.hasRemaining()) // 枚举 charBuffer 中的数据
            System.out.print(charBuffer.get() + " ");

        // 枚举 doubleBuffer 中的数据
        while(doubleBuffer.position() < doubleBuffer.limit())
            System.out.print(doubleBuffer.get() + " ");

    }
}

```

运行结果:

```

intBuffer 的当前位置: 2
2000
1000
20 33 a b c d e f g 1.1 2.2

```

**注意：**如果必须使用缓冲区的大小来读取缓冲区的数据，尽量不要使用 `capacity`，而要使用 `limit`。如尽量不要写成如下的代码：

```
while(byteBuffer.position() < byteBuffer.capacity())  
System.out.println(byteBuffer.get());
```

这是因为当 `limit` 比 `capacity` 小时，上面的代码将会抛出一个 `BufferUnderflowException` 异常。

UnRegistered