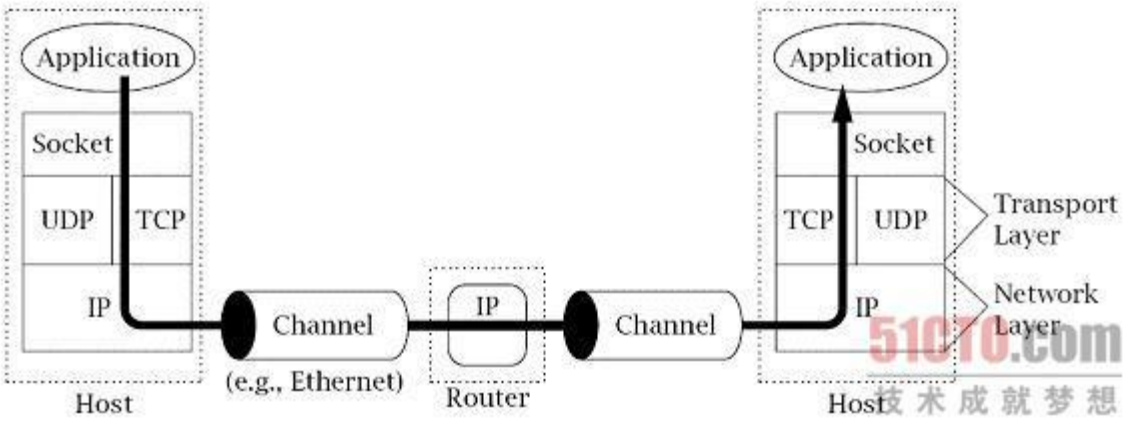
java socket 通信

1：简介

Java 语言从一开始就是为了让人们使用互联网而设计的，它为实现程序的相互通信提供了许多有用的抽象应用程序接口（API， Application Programming Interface），这类应用程序接口被称为套接字（sockets）。

信息（information）是指由程序创建和解释的字节序列。在计算机网络环境中，这些字节序列被称为分组报文（packets）。一组报文包括了网络用来完成工作的控制信息，有时还包括一些用户数据。用于定位分组报文目的地址的信息就是一个例子。路由器正是利用了这些控制信息来实现对每个报文的转发。

协议（protocol）相当于是相互通信的程序间达成的一种约定，它规定了分组报文的交换方式和它们包含的意义。一组协议规定了分组报文的结构（例如报文中的哪一部分表明了其目的地址）以及怎样对报文中所包含的信息进行解析。设计一组协议，通常是为了在一定约束条件下解决某一特定的问题。比如，超文本传输协议（HTTP，HyperText Transfer Protocol）是为了解决在服务器间传递超文本对象的问题，这些超文本对象在服务器中创建和存储，并由 Web 浏览器进行可视化，以使其对用户有用。即时消息协议是为了使两个或更多用户间能够交换简短的文本信息。



Application：应用程序；Socket：套接字；Host：主机；Channel：通信信道；Ethernet：以太网；Router：路由器；Network Layer：网络层；Transport Layer：传输层。

IP 协议提供了一种数据报服务：每组分组报文都由网络独立处理和分发，就像信件或包裹通过邮政系统发送一样。为了实现这个功能，每个 IP 报文必须包含一个保存其目的地址（address）的字段，就像你所投递的每份包裹都写明了收件人地址。（我们随即会对地址进行更详细的说明。）尽管绝大部分递送公司会保证将包裹送达，但 IP 协议只是一个"尽力而为"（best-effort）的协议：它试图分发每一个分组报文，但在网络传输过程中，偶

尔也会发生丢失报文，使报文顺序被打乱，或重复发送报文的情况。
IP 协议层之上称为传输层（transport layer）。它提供了两种可选择的协议：TCP 协议和 UDP 协议。这两种协议都建立在 IP 层所提供的服务基础上，但根据应用程序协议（application protocols）的不同需求，它们使用了不同的方法来实现不同方式的传输。TCP 协议和 UDP 协议有一个共同的功能，即寻址。回顾一下，IP 协议只是将分组报文分发到了不同的主机，很明显，还需要更细粒度的寻址将报文发送到主机中指定的应用程序，因为同一主机上可能有多个应用程序在使用网络。TCP 协议和 UDP 协议使用的地址叫做端口号（port numbers），都是用来区分同一主机中的不同应用程序。TCP 协议和 UDP 协议也称为端到端传输协议（end-to-end transport protocols），因为它们将数据从一个应用程序传输到另一个应用程序，而 IP 协议只是将数据从一个主机传输到另一主机。

TCP 协议能够检测和恢复 IP 层提供的主机到主机的信道中可能发生的报文丢失、重复及其他错误。TCP 协议提供了一个可信赖的字节流（reliable byte-stream）信道，这样应用程序就不需要再处理上述的问题。TCP 协议是一种面向连接（connection-oriented）的协议：在使用它进行通信之前，两个应用程序之间首先要建立一个 TCP 连接，这涉及到相互通信的两台电脑的 TCP 部件间完成的握手消息（handshake messages）的交换。使用 TCP 协议在很多方面都与文件的输入输出（I/O, Input/Output）相似。实际上，由一个程序写入的文件再由另一个程序读取就是一个 TCP 连接的适当模型。另一方面，UDP 协议并不尝试对 IP 层产生的错误进行修复，它仅仅简单地扩展了 IP 协议"尽力而为"的数据报服务，使它能够在应用程序之间工作，而不是在主机之间工作。因此，使用了 UDP 协议的应用程序必须为处理报文丢失、顺序混乱等问题做好准备。

在 TCP/IP 协议中，有两部分信息用来定位一个指定的程序：互联网地址（Internet address）和端口号（port number）。其中互联网地址由 IP 协议使用，而附加的端口地址信息由传输协议（TCP 或 IP 协议）对其进行解析。互联网地址由二进制的数字组成，有两种型式，分别对应了两个版本的标准互联网协议。现在最常用的版本是版本 4，即 IPv4，另一个版本是刚开始开发的版本 6，即 IPv。IPv4 的地址长 32 位，只能区分大约 40 亿个独立地址，对于如今的互联网来说，这是不够大的。（也许看起来很多，但由于地址的分配方式的原因，有很多都被浪费了）出于这个原因引入了 IPv6，它的地址有 128 位长。

一台主机，只要它连接到网络，一个互联网地址就能定位这条主机。但是反过来，一台主机并不对应一个互联网地址。因为每台主机可以有多个接口，每个接口又可以有多个地址。（实际上一个接口可以同时拥有 IPv4 地址和 IPv6 地址）。端口号是一组 16 位的无符号二进制数，每个端口号的范围是 1 到 65535。（0 被保留）。每个版本的 IP 协议都定义了一些特殊用途的地址。其中值得注意的一个是回环地址（loopback address），该地

址总是被分配个一个特殊的回环接口（loopback interface）。回环接口是一种虚拟设备，它的功能只是简单地将发送给它的报文直接回发给发送者。IPv4 的回环地址是 127.0.0.1[，IPv6 的回环地址是 0:0:0:0:0:0:0:1。

IPv4 地址中的另一种特殊用途的保留地址包括那些"私有用途"的地址。它们包括 IPv4 中所有以 10 或 192.168 开头的地址，以及第一个数是 172，第二个数在 16 到 31 的地址。（在 IPv6 中没有相应的这类地址）这类地址最初是为了在私有网络中使用而设计的，不属于公共互联网的一部分。现在这类地址通常被用在家庭或小型办公室中，这些地方通过 NAT（Network Address Translation，网络地址转换）设备连接到互联网。NAT 设备的功能就像一个路由器，转发分组报文时将转换（重写）报文中的地址和端口。更准确地说，它将一个接口中报文的私有地址端口对（private address, port pairs）映射成另一个接口中的公有地址端口对（public address, port pairs）。这就使一小组主机（如家庭网络）能够有效地共享同一个 IP 地址。重要的是这些内部地址不能从公共互联网访问。

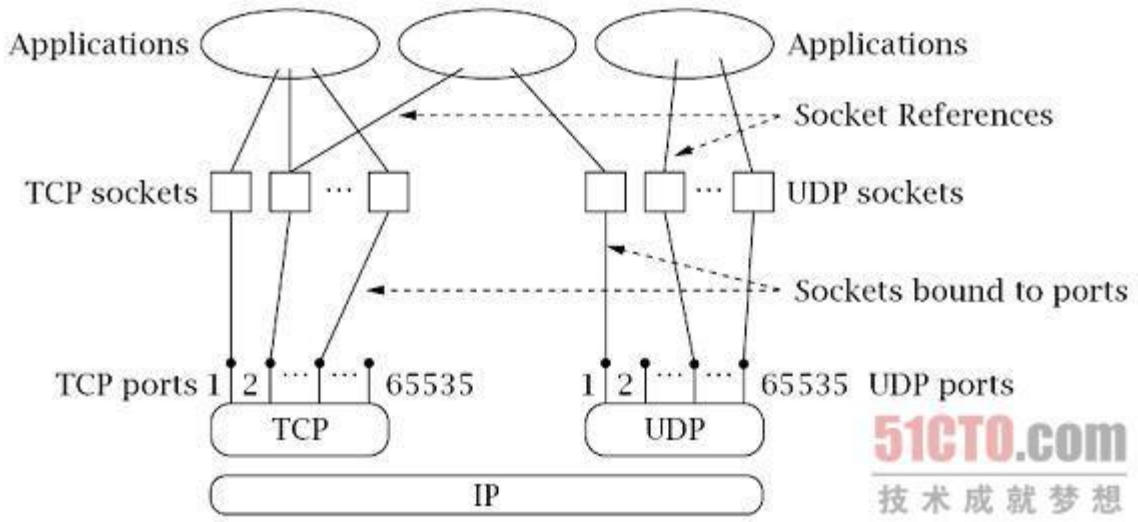
多播（multicast）地址。普通的 IP 地址（有时也称为"单播"地址）只与唯一一个目的地址相关联，而多播地址可能与任意数量的目的地址关联。IPv4 中的多播地址在点分格式中，第一个数字在 224 到 239 之间。IPv6 中，多播地址由 FF 开始。

习惯于通过名字来指代一个主机，例如：host.example.com。然而，互联网协议只能处理二进制的网络地址，而不是主机名。首先应该明确的是，使用主机名而不使用地址是出于方便性的考虑，这与 TCP/IP 提供的基本服务是相互独立的。你也可以不使用名字来编写和使用 TCP/IP 应用程序。当使用名字来定位一个通信终端时，系统将做一些额外的工作把名字解析成地址。有两个原因证明这额外的步骤是值得的：第一，相对于点分形式（或 IPv6 中的十六进制数字串），人们更容易记住名字；第二，名字提供了一个间接层，使 IP 地址的变化对用户不可见。如网络服务器 www.mkp.com 的地址就改变过。由于我们通常都使用网络服务器的名字，而且地址的改变很快就被反应到映射主机名和网络地址的服务上，如 www.mkp.com 从之前的地址 208.164.121.48 对应到了现在的地址，这种变化对通过名字访问该网络服务器的程序是透明的。名字解析服务可以从各种各样的信息源获取信息。两个主要的信息源是域名系统（DNS，Domain Name System）和本地配置数据库。DNS 是一种分布式数据库。DNS 协议允许连接到互联网的主机通过 TCP 或 UDP 协议从 DNS 数据库中获取信息。本地配置数据库通常是一种与具体操作系统相关的机制，用来实现本地名称与互联网地址的映射。

客户端（client）和服务器（server）这两个术语代表了两种角色：客户端是通信的发起者，而服务器程序则被动等待客户端发起通信，并对其作出响应。客户端与服务器组成了应用程序（application）。服务器具有一定的特殊能力，如提供数据库服务，并使任何客

户端能够与之通信。一个程序是作为客户端还是服务器，决定了它在与其对等端（peer）建立通信时使用的套接字 API 的形式（客户端的对等端是服务器，反之亦然）。更进一步来说，客户端与服务器端的区别非常重要，因为客户端首先需要知道服务器的地址和端口号，反之则不需要。如果有必要，服务器可以使用套接字 API，从收到的第一个客户端通信消息中获取其地址信息。这与打电话非常相似：被呼叫者不需要知道拨电话者的电话号码。就像打电话一样，只要通信连接建立成功，服务器和客户端之间就没有区别了。服务器可以使用任何端口号，但客户端必须能够获知这些端口号。在互联网上，一些常用的端口号被约定赋给了某些应用程序。

Socket（套接字）是一种抽象层，应用程序通过它来发送和接收数据，就像应用程序打开一个文件句柄，将数据读写到稳定的存储器上一样。一个 socket 允许应用程序添加到网络中，并与处于同一个网络中的其他应用程序进行通信。一台计算机上的应用程序向 socket 写入的信息能够被另一台计算机上的另一个应用程序读取，反之亦然。



Applications：应用程序；TCP sockets：TCP 套接字；TCP ports：TCP 端口；Socket References：套接字引用；UDP sockets：UDP 套接字；Sockets bound to ports：套接字绑定到端口；UDP ports：UDP 端口。

不同类型的 socket 与不同类型的底层协议族以及同一协议族中的不同协议栈相关联。现在 TCP/IP 协议族中的主要 socket 类型为流套接字（sockets sockets）和数据报套接字（datagram sockets）。流套接字将 TCP 作为其端对端协议（底层使用 IP 协议），提供了一个可信赖的字节流服务。一个 TCP/IP 流套接字代表了 TCP 连接的一端。数据报套接字使用 UDP 协议（底层同样使用 IP 协议），提供了一个"尽力而为"（best-effort）的数据报服务，应用程序可以通过它发送最长 65500 字节的个人信息。当然，其他协议族也支持

流套接字和数据报套接字，本文只对 TCP 流套接字和 UDP 数据报套接字进行讨论。一个 TCP/IP 套接字由一个互联网地址，一个端对端协议（TCP 或 UDP 协议）以及一个端口号唯一确定。主机中的多个程序可以同时访问同一个套接字。在实际应用中，访问相同套接字的不同程序通常都属于同一个应用（例如，Web 服务程序的多个拷贝），但从理论上讲，它们是可以属于不同应用的。

2：基本套接字

一个客户端要发起一次通信，首先必须知道运行服务器端程序的主机的 IP 地址。然后由网络的基础结构利用目标地址（destination address），将客户端发送的信息传递到正确的主机上。在 Java 中，地址可以由一个字符串来定义，这个字符串可以是数字型的地址（不同版本的 IP 地址有不同的型式，如 192.0.2.27 是一个 IPv4 地址，fe20:12a0::0abc:1234 是一个 IPv6 地址），也可以是主机名（如 server.example.com）。主机名必须能够被解析（resolved）成数字型地址才能用来进行通信。

NetworkInterface：NetworkInterface 类提供了访问主机所有接口的信息的功能。(IP 地址实际上是分配给了主机与网络之间的连接，而不是主机本身)

InetAddress：网络接口，代表了一个网络目标地址，包括主机名和数字类型的地址信息。该类有两个子类，Inet4Address 和 Inet6Address，分别对应了目前 IP 地址的两个版本。InetAddress 实例是不可变的，一旦创建，每个实例就始终指向同一个地址。

SocketAddress：抽象类，代表了套接字地址的一般型式，它的子类 InetSocketAddress 是针对 TCP/IP 套接字的特殊型式，封装了一个 InetAddress 和一个端口号。InetSocketAddress 类为主机地址和端口号提供了一个不可变的组合。只接收端口号作为参数的构造函数将使用特殊的"任何"地址来创建实例，这点对于服务器端非常有用。接收字符串主机名的构造函数会尝试将其解析成相应的 IP 地址。

Socket 和 ServerSocket：Java 为 TCP 协议提供了两个类：Socket 类和 ServerSocket 类。一个 Socket 实例代表了 TCP 连接的一端。一个 TCP 连接（TCP connection）是一条抽象的双向信道，两端分别由 IP 地址和端口号确定。在开始通信之前，要建立一个 TCP 连接，这需要先由客户端 TCP 向服务器端 TCP 发送连接请求。ServerSocket 实例则监听 TCP 连接请求，并为每个请求创建新的 Socket 实例。也就是说，服务器端要同时处理 ServerSocket 实例和 Socket 实例，而客户端只需要使用 Socket 实例。

DatagramPacket：Java 程序员通过 DatagramPacket 类和 DatagramSocket 类来使用 UDP 套接字。客户端和服务器端都使用 DatagramSockets 来发送数据，使用 DatagramPackets 来接收数据。

DatagramPacket：UDP 终端交换的是一种称为数据报文的自包含（self-contained）信息。这种信息在 Java 中表示为 DatagramPacket 类的实例，发送信息时，Java 程序创建一个包含了待发送信息的 DatagramPacket 实例，并将其作为参数传递给 DatagramSocket 类的 send()方法。接收信息时，Java 程序首先创建一个 DatagramPacket 实例，该实例中预先分配了一些空间（一个字节数组 byte[]），并将接收到的信息存放在该空间中。然后把该实例作为参数传递给 DatagramSocket 类的 receive()方法。

InetAddressExample.java

[java] view plaincopy

```
1.  <span style="font-size:14px;">import java.util.Enumeration;
2.  import java.net.*;
3.
4.  public class InetAddressExample {
5.
6.      public static void main(String[] args) {
7.
8.          // Get the network interfaces and associated addresses for this host
9.          try {
10.             Enumeration<NetworkInterface> interfaceList = NetworkInterface.g
    etNetworkInterfaces();
11.             if (interfaceList == null) {
12.                 System.out.println("--No interfaces found--");
13.             } else {
14.                 while (interfaceList.hasMoreElements()) {
15.                     NetworkInterface iface = interfaceList.nextElement();
16.                     System.out.println("Interface " + iface.getName() + ":")
    ;
17.                     Enumeration<InetAddress> addrList = iface.getInetAddress
    es();
18.                     if (!addrList.hasMoreElements()) {
19.                         System.out.println("\t(No addresses for this interfa
    ce)");
20.                     }
21.                     while (addrList.hasMoreElements()) {
```

```
                        InetAddress address = addrList.nextElement();
                        System.out.print("\tAddress "+ ((address instanceof
Inet4Address ? "(v4)": (address instanceof Inet6Address ? "(v6)": "(?)"))));

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

                    }
                }
            }
        } catch (SocketException se) {
            System.out.println("Error getting network interfaces:"+ se.getMe
ssage());
        }

        // Get name(s)/address(es) of hosts given on command line
        for (String host : args) {
            try {
                System.out.println(host + ":");
                InetAddress[] addressList = InetAddress.getAllByName(host);

                for (InetAddress address : addressList) {
                    System.out.println("\t" + address.getHostName() + "/"+ a
ddress.getHostAddress());
                }
            } catch (UnknownHostException e) {
                System.out.println("\tUnable to find address for " + host);

            }
        }
    }
}</span>
```

运行结果：

% java InetAddressExample www.mkp.com blah.blah 129.35.69.7

Interface lo:

Address (v4): 127.0.0.1

Address (v6): 0:0:0:0:0:0:0:1

Address (v6): fe80:0:0:0:0:0:0:1%1

Interface eth0:

Address (v4): 192.168.159.1

Address (v6): fe80:0:0:0:250:56ff:fec0:8%4

www.mkp.com:
www.mkp.com/129.35.69.7
blah.blah:
Unable to find address for blah.blah
129.35.69.7:
129.35.69.7/129.35.69.7

地址解析器在放弃对一个主机名的解析之前，会到多个不同的地方查找该主机名。如果由于某些原因使名字服务失效（例如由于程序所运行的机器并没有连接到所有的网络），试图通过名字来定位一个主机就可能失败。而且这还将耗费大量的时间，因为系统将尝试各种不同的方法来将主机名解析成 IP 地址，因此最好能直接使用点分形式的 IP 地址来访问一个主机

InetAddress: 创建和访问
static InetAddress[ ] getAllByName(String host)

static InetAddress getByName(String host)

static InetAddress getLocalHost()
byte[] getAddress()

InetAddress: 字符串表示

String toString()
String getHostAddress()
String getHostName()
String getCanonicalHostName()

InetAddress: 检测属性
boolean isAnyLocalAddress()
boolean isLinkLocalAddress()
boolean isLoopbackAddress()
boolean isMulticastAddress()
boolean isMCGlobal()
boolean isMCLinkLocal()
boolean isMCNodeLocal()
boolean isMCOrgLocal()

boolean isMCSiteLocal()
boolean isReachable(int timeout)
boolean isReachable(NetworkInterface netif, int ttl, int timeout)

最后两个方法检查是否真能与 InetAddress 地址确定的主机进行数据报文交换。注意，与其他句法检查方法不一样的是，这些方法引起网络系统执行某些动作，即发送数据报文。系统不断尝试发送数据报文，直到指定的时间（以毫秒为单位）用完才结束。后面这种形式更详细：它明确指出数据报文必须经过指定的网络接口（NetworkInterface），并检查其是否能在指定的生命周期（time-to-live，TTL）内联系上目的地址。TTL 限制了一个数据报文在网络上能够传输的距离。后面两个方法的有效性通常还受到安全管理配置方面的限制。

NetworkInterface: 创建，获取信息

static EnumerationNetworkInterface————————————————
getNetworkInterfaces()
static NetworkInterface getByInetAddress(InetAddress addr)
static NetworkInterface getByName(String name)
EnumerationInetAddress———————————————— getInetAddresses()
String getName()
String getDisplayName()

上面第一个方法非常有用，使用它可以很容易获取到运行程序的主机的 IP 地址：通过 getNetworkInterfaces()方法可以获取一个接口列表，再使用实例的 getInetAddresses()方法就可以获取每个接口的所有地址。注意：这个列表包含了主机的所有接口，包括不能够向网络中的其他主机发送或接收消息的虚拟回环接口。同样，列表中可能还包括外部不可达的本地链接地址。由于这些列表都是无序的，所以你不能简单地认为，列表中第一个接口的第一个地址一定能够通过互联网访问，而是要通过前面提到的 InetAddress 类的属性检查方法，来判断一个地址不是回环地址，不是本地链接地址等等。getName()方法返回一个接口（interface）的名字（不是主机名）。这个名字由字母字符串加上一个数字组成，如 eth0。在很多系统中，回环地址的名字都是 lo0。

TCP 套接字

TCP 客户端

客户端向服务器发起连接请求后，就被动地等待服务器的响应。典型的 TCP 客户端要经过下面三步：

1.创建一个 Socket 实例：构造器向指定的远程主机和端口建立一个 TCP 连接。

2. 通过套接字的输入输出流（I/O streams）进行通信：一个 Socket 连接实例包括一个 InputStream 和一个 OutputStream，它们的用法同于其他 Java 输入输出流。
3. 使用 Socket 类的 close()方法关闭连接。

TCPEchoClient.java（这是一个通过 TCP 协议与回馈服务器（echo server）进行通信的客户端）

[java] view plaincopy

```java
<span style="font-size:14px;">import java.net.Socket;
import java.net.SocketException;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

public class TCPEchoClient {
    public static void main(String[] args) throws IOException {
        if ((args.length < 2) || (args.length > 3)) // Test for correct # of args
            throw new IllegalArgumentException("Parameter(s): <Server> <Word> [<Port>]");
        String server = args[0]; // Server name or IP address
        // Convert argument String to bytes using the default character encoding
        byte[] data = args[1].getBytes();
        int servPort = (args.length == 3) ? Integer.parseInt(args[2]) : 7;
        // Create socket that is connected to server on specified port
        Socket socket = new Socket(server, servPort);
        System.out.println("Connected to server...sending echo string");
        InputStream in = socket.getInputStream();
        OutputStream out = socket.getOutputStream();
        out.write(data); // Send the encoded string to the server
        // Receive the same string back from the server
        int totalBytesRcvd = 0; // Total bytes received so far
        int bytesRcvd; // Bytes received in last read
        while (totalBytesRcvd < data.length) {
            if ((bytesRcvd = in.read(data, totalBytesRcvd, data.length- totalBytesRcvd)) == -1)
                throw new SocketException("Connection closed prematurely");

            totalBytesRcvd += bytesRcvd;
        } // data array is full
        System.out.println("Received: " + new String(data));
```

```
30.         socket.close(); // Close the socket and its streams
31.     }
32. }</span>
```

为什么不只用一个 read 方法呢？TCP 协议并不能确定在 read()和 write()方法中所发送信息的界限，也就是说，虽然我们只用了一个 write()方法来发送回馈字符串，回馈服务器也可能从多个块（chunks）中接受该信息。即使回馈字符串在服务器上存于一个块中，在返回的时候，也可能被 TCP 协议分割成多个部分。对于初学者来说，最常见的错误就是认为由一个 write()方法发送的数据总是会由一个 read()方法来接收。

Socket: 创建

Socket(InetAddress remoteAddr, int remotePort)
Socket(String remoteHost, int remotePort)
Socket(InetAddress remoteAddr, int remotePort, InetAddress localAddr, int localPort)
Socket(String remoteHost, int remotePort, InetAddress localAddr, int localPort)
Socket()
前两个构造函数没有指定本地地址和端口号，因此将采用默认地址和可用的端口号。在有多个接口的主机上指定本地地址是有用的。指定的目的地址字符串参数可以使用与 InetAddress 构造函数的参数相同的型式。最后一个构造函数创建一个没有连接的套接字，在使用它进行通信之前，必须进行显式连接（通过 connect()方法）。

Socket: 操作

void connect(SocketAddress destination)
void connect(SocketAddress destination, int timeout)
InputStream getInputStream()
OutputStream getOutputStream()
void close()
void shutdownInput()
void shutdownOutput()

connect()方法将使指定的终端打开一个 TCP 连接。SocketAddress 抽象类代表了套接字地址的一般型式，它的子类 InetSocketAddress 是针对 TCP/IP 套接字的特殊型式。与远程主机的通信是通过与套接字相关联的输入输出流实现的。可以使用 get...Stream()方法来获取这些流。close()方法关闭套接字及其关联的输入输出流，从而阻止对其的进一步操作。shutDownInput()方法关闭 TCP 流的输入端，任何没有读取的数据都将被舍弃，包括那些已经被套接字缓存的数据、正在传输的数据以及将要到达的数据。后续的任何从套接

字读取数据的尝试都将抛出异常。shutDownOutput()方法在输出流上也产生类似的效果，但在具体实现中，已经写入套接字输出流的数据，将被尽量保证能发送到另一端。注意：默认情况下，Socket 是在 TCP 连接的基础上实现的，但是在 Java 中，你可以改变 Socket 的底层连接。

Socket: 获取/检测属性

InetAddress getInetAddress()
int getPort()
InetAddress getLocalAddress()
int getLocalPort()
SocketAddress getRemoteSocketAddress()
SocketAddress getLocalSocketAddress()

Socket 类实际上还有大量的其他相关属性，称为套接字选项（socket options）。这些属性对于编写基本应用程序是不必要的

InetSocketAddress: 创建与访问

InetSocketAddress(InetAddress addr, int port)
InetSocketAddress(int port)
InetSocketAddress(String hostname, int port)
static InetSocketAddress createUnresolved(String host, int port)
boolean isUnresolved()
InetAddress getAddress()
int getPort()
String getHostName()
String toString()

createUnresolved()静态方法允许在不对主机名进行解析情况下创建实例

TCP 服务器端

服务器端的工作是建立一个通信终端，并被动地等待客户端的连接。典型的 TCP 服务器有如下两步工作：

1. 创建一个 ServerSocket 实例并指定本地端口。此套接字的功能是侦听该指定端口收到的连接。

2. 重复执行：

a. 调用 ServerSocket 的 accept()方法以获取下一个客户端连接。基于新建立的客户端连接，创建一个 Socket 实例，并由 accept()方法返回。

b. 使用所返回的 Socket 实例的 InputStream 和 OutputStream 与客户端进行通信。

c. 通信完成后，使用 Socket 类的 close()方法关闭该客户端套接字连接。

TCPEchoServer.java(为我们前面的客户端程序实现了一个回馈服务器。这个服务器程序非常简单，它将一直运行，反复接受连接请求，接收并返回字节信息。直到客户端关闭了连接，它才关闭客户端套接字。)

[java] view plaincopy

```
1.  <span style="font-
    size:14px;">import java.net.*; // for Socket, ServerSocket, and InetAddress

2.  import java.io.*; // for IOException and Input/OutputStream
3.
4.  public class TCPEchoServer {
5.
6.      private static final int BUFSIZE = 32; // Size of receive buffer
7.
8.      public static void main(String[] args) throws IOException {
9.          if (args.length != 1) // Test for correct # of args
10.             throw new IllegalArgumentException("Parameter(s): <Port>");
11.         int servPort = Integer.parseInt(args[0]);
12.         // Create a server socket to accept client connection requests
13.         ServerSocket servSock = new ServerSocket(servPort);
14.         int recvMsgSize; // Size of received message
15.         byte[] receiveBuf = new byte[BUFSIZE]; // Receive buffer
16.         while (true) { // Run forever, accepting and servicing connections
17.             Socket clntSock = servSock.accept(); // Get client connection
18.             SocketAddress clientAddress = clntSock.getRemoteSocketAddress();

19.             System.out.println("Handling client at " + clientAddress);
20.             InputStream in = clntSock.getInputStream();
21.             OutputStream out = clntSock.getOutputStream();
22.             // Receive until client closes connection, indicated by -
    1 return
23.             while ((recvMsgSize = in.read(receiveBuf)) != -1) {
24.                 out.write(receiveBuf, 0, recvMsgSize);
25.             }
26.             clntSock.close(); // Close the socket. We are done with this cli
    ent!
27.         }
28.         /* NOT REACHED */
```

```
29.     }
30. }</span>
```

ServerSocket: 创建

ServerSocket(int localPort)
ServerSocket(int localPort, int queueLimit)
ServerSocket(int localPort, int queueLimit, InetAddress localAddr)
ServerSocket()

如果指定了本地地址，该地址就必须是主机的网络接口之一；如果没有指定，套接字将接受指向主机任何 IP 地址的连接。这将对有多个接口而服务器端只接受其中一个接口连接的主机非常有用。第四个构造函数能创建一个没有关联任何本地端口的 ServerSocket 实例。在使用该实例前，必须为其绑定（bind()方法）一个端口号。

ServerSocket: 操作

void bind(int port)
void bind(int port, int queuelimit)
Socket accept()
void close()
bind()方法为套接字关联一个本地端口。每个 ServerSocket 实例只能与唯一一个端口相关联。如果该实例已经关联了一个端口，或所指定的端口已经被占用，则将抛出 IOException 异常。accept()方法为下一个传入的连接请求创建 Socket 实例，并将已成功连接的 Socket 实例返回给服务器端套接字。如果没有连接请求等待，accept()方法将阻塞等待，直到有新的连接请求到来或超时。close()方法关闭套接字。调用该方法后，服务器将拒绝接受传入该套接字的客户端连接请求。

ServerSocket: 获取属性

InetAddress getInetAddress()
SocketAddress getLocalSocketAddress()
int getLocalPort()

如果在一个 TCP 套接字关联的输出流上进行操作，当大量的数据已发送，而连接的另一端所关联的输入流最近没有调用 read()方法时，OutputStream 中的方法可能会阻塞。如果不作特殊处理，这可能会产生一些不想得到的后果。在一个 TCP 套接字关联的输入流上没有数据可读，而又没有检测到流结束标记时，所有的 read()方法都将阻塞等待，直到至少有

一个字节可读。在没有数据可读，同时又检测到流结束标记时，InputStream 中的方法都将返回-1。

UDP 套接字

UDP 协议提供了一种不同于 TCP 协议的端到端服务。实际上 UDP 协议只实现两个功能：1）在 IP 协议的基础上添加了另一层地址（端口），2）对数据传输过程中可能产生的数据错误进行了检测，并抛弃已经损坏的数据。由于其简单性，UDP 套接字具有一些与我们之前所看到的 TCP 套接字不同的特征。例如，UDP 套接字在使用前不需要进行连接。TCP 协议与电话通信相似，而 UDP 协议则与邮件通信相似：你寄包裹或信件时不需要进行"连接"，但是你得为每个包裹和信件指定目的地址。类似的，每条信息（即数据报文，datagram）负载了自己的地址信息，并与其他信息相互独立。在接收信息时，UDP 套接字扮演的角色就像是一个信箱，从不同地址发送来的信件和包裹都可以放到里面。一旦被创建，UDP 套接字就可以用来连续地向不同的地址发送信息，或从任何地址接收信息。UDP 套接字与 TCP 套接字的另一个不同点在于他们对信息边界的处理方式不同：UDP 套接字将保留边界信息。这个特性使应用程序在接受信息时，从某些方面来说比使用 TCP 套接字更简单。最后一个不同点是，UDP 协议所提供的端到端传输服务是尽力而为（best-effort）的，即 UDP 套接字将尽可能地传送信息，但并不保证信息一定能成功到达目的地址，而且信息到达的顺序与其发送顺序不一定一致（就像通过邮政部门寄信一样）。因此，使用了 UDP 套接字的程序必须准备好处理信息的丢失和重排。

既然 UDP 协议为程序带来了这个额外的负担，为什么还会使用它而不使用 TCP 协议呢？原因之一是效率：如果应用程序只交换非常少量的数据，例如从客户端到服务器端的简单请求消息，或一个反方向的响应消息，TCP 连接的建立阶段就至少要传输其两倍的信息量（还有两倍的往返延迟时间）。另一个原因是灵活性：如果除可靠的字节流服务外，还有其他的需求，UDP 协议则提供了一个最小开销的平台来满足任何需求的实现。

与 TCP 协议发送和接收字节流不同，UDP 终端交换的是一种称为数据报文的自包含（self-contained）信息。这种信息在 Java 中表示为 DatagramPacket 类的实例。发送信息时，Java 程序创建一个包含了待发送信息的 DatagramPacket 实例，并将其作为参数传递给 DatagramSocket 类的 send()方法。接收信息时，Java 程序首先创建一个 DatagramPacket 实例，该实例中预先分配了一些空间（一个字节数组 byte[]），并将接收到的信息存放在该空间中。然后把该实例作为参数传递给 DatagramSocket 类的 receive()方法。除传输的信息本身外，每个 DatagramPacket 实例中还附加了地址和端口信息，其具体含义取决于该数据报文是被发送还是被接收。若是要发送的数据报文，

DatagramPacket 实例中的地址则指明了目的地址和端口号，若是接收到的数据报文，DatagramPacket 实例中的地址则指明了所收信息的源地址。

DatagramPacket: 创建

DatagramPacket(byte[ ] data, int length)
DatagramPacket(byte[ ] data, int offset, int length)
DatagramPacket(byte[ ] data, int length, InetAddress remoteAddr, int remotePort)
DatagramPacket(byte[ ] data, int offset, int length, InetAddress remoteAddr, int remotePort)
DatagramPacket(byte[ ] data, int length, SocketAddress sockAddr)
DatagramPacket(byte[ ] data, int offset, int length, SocketAddress sockAddr)

以上构造函数都创建一个数据部分包含在指定的字节数组中的数据报文，前两种形式的构造函数主要用来创建接收的端的 DatagramPackets 实例，因为没有指定其目的地址（尽管可以通过 setAddress() 和 setPort()方法，或 setSocketAddress()方法来指定）。后四种形式主要用来创建发送端的 DatagramPackets 实例。如果指定了 offset，数据报文的数据部分将从字节数组的指定位置发送或接收数据。length 参数指定了字节数组中在发送时要传输的字节数，或在接收数据时所能接收的最多字节数。length 参数可能比 data.length 小，但不能比它大。

DatagramPacket: 地址处理

InetAddress getAddress()
void setAddress(InetAddress address)
int getPort()
void setPort(int port)
SocketAddress getSocketAddress()
void setSocketAddress(SocketAddress sockAddr)

DatagramPacket: 处理数据

int getLength()
void setLength(int length)
int getOffset()
byte[ ] getData()

void setData(byte[ ] data)

void setData(byte[ ] buffer, int offset, int length)

UDP 客户端

UDP 客户端首先向被动等待联系的服务器端发送一个数据报文。一个典型的 UDP 客户端主要执行以下三步：

1. 创建一个 DatagramSocket 实例，可以选择对本地地址和端口号进行设置。
2. 使用 DatagramSocket 类的 send() 和 receive()方法来发送和接收 DatagramPacket 实例，进行通信。
3. 通信完成后，使用 DatagramSocket 类的 close()方法来销毁该套接字。

与 Socket 类不同，DatagramSocket 实例在创建时并不需要指定目的地址。这也是 TCP 协议和 UDP 协议的最大不同点之一。在进行数据交换前，TCP 套接字必须跟特定主机和另一个端口号上的 TCP 套接字建立连接，之后，在连接关闭前，该套接字就只能与相连接的那个套接字通信。而 UDP 套接字在进行通信前则不需要建立连接，每个数据报文都可以发送到或接收于不同的目的地址。（DatagramSocket 类的 connect()方法确实允许指定远程地址和端口，但该功能是可选的。）

使用 UDP 协议的一个后果是数据报文可能丢失。在我们的回馈协议中，客户端的回馈请求信息和服务器端的响应信息都有可能在网络中丢失。回顾前面所介绍的 TCP 回馈客户端，其发送了一个回馈字符串后，将在 read()方法上阻塞等待响应。如果试图在我们的 UDP 回馈客户端上使用相同的策略，数据报文丢失后，我们的客户端就会永远阻塞在 receive()方法上。为了避免这个问题，我们在客户端使用 DatagramSocket 类的 setSoTimeout()方法来指定 receive()方法的最长阻塞时间，因此，如果超过了指定时间仍未得到响应，客户端就会重发回馈请求。我们的回馈客户端执行以下步骤：

1. 向服务器端发送回馈字符串。
2. 在 receive()方法上最多阻塞等待 3 秒钟，在超时前若没有收到响应，则重发请求（最多重发 5 次）。
3. 终止客户端。

UDPEchoClientTimeout.java（UDP 版本的回馈客户端，在客户端使用 DatagramSocket 类的 setSoTimeout()方法来指定 receive()方法的最长阻塞时间，因此，如果超过了指定时间仍未得到响应，客户端就会重发回馈请求）

[java] view plaincopy

```
1.  <span style="font-size:14px;">import java.net.DatagramSocket;
2.  import java.net.DatagramPacket;
```

```
3.  import java.net.InetAddress;
4.  import java.io.IOException;
5.  import java.io.InterruptedIOException;
6.
7.  public class UDPEchoClientTimeout {
8.
9.      private static final int TIMEOUT = 3000; // Resend timeout (milliseconds
    )
10.     private static final int MAXTRIES = 5; // Maximum retransmissions
11.
12.     public static void main(String[] args) throws IOException {
13.         if ((args.length < 2) || (args.length > 3)) { // Test for correct #
    of args
14.             throw new IllegalArgumentException("Parameter(s): <Server> <Word
    > [<Port>]");
15.         }
16.         InetAddress serverAddress = InetAddress.getByName(args[0]); // Serve
    r address
17.         // Convert the argument String to bytes using the default encoding
18.         byte[] bytesToSend = args[1].getBytes();
19.         int servPort = (args.length == 3) ? Integer.parseInt(args[2]) : 7;
20.         DatagramSocket socket = new DatagramSocket();
21.         socket.setSoTimeout(TIMEOUT); // Maximum receive blocking time (mill
    iseconds)
22.         DatagramPacket sendPacket = new DatagramPacket(bytesToSend,bytesToSe
    nd.length, serverAddress, servPort); // Sending packet
23.         DatagramPacket receivePacket =new DatagramPacket(new byte[bytesToSen
    d.length], bytesToSend.length);<span style="font-
    family: Arial, Helvetica, sans-serif;">// Receiving packet</span>
24.         int tries = 0; // Packets may be lost, so we have to keep trying
25.         boolean receivedResponse = false;
26.         do {
27.             socket.send(sendPacket); // Send the echo string
28.             try {
29.                 socket.receive(receivePacket); // Attempt echo reply recepti
    on
30.                 if (!receivePacket.getAddress().equals(serverAddress)) {// C
    heck source
31.                     throw new IOException("Received packet from an unknown s
    ource");
32.                 }
33.                 receivedResponse = true;
34.             } catch (InterruptedIOException e) { // We did not get anything
```

```
35.                 tries += 1;
36.                 System.out.println("Timed out, " + (MAXTRIES - tries)+ " mor
    e tries...");
37.             }
38.         } while ((!receivedResponse) && (tries < MAXTRIES));
39.         if (receivedResponse) {
40.             System.out.println("Received: "+ new String(receivePacket.getDat
    a()));
41.         } else {
42.             System.out.println("No response -- giving up.");
43.         }
44.         socket.close();
45.     }
46. }</span>
```

receive()方法将阻塞等待，直到收到一个数据报文或等待超时。超时信息由 InterruptedIOException 异常指示。一旦超时，发送尝试计数器（tries)）加 1，并重新发送。若尝试了最大次数后，仍没有接收到数据报文，循环将退出。如果 receive()方法成功接收了数据，我们将循环标记 receivedResponse 设为 true，以退出循环。由于数据报文可能发送自任何地址，我们需要验证所接收的数据报文，检查其源地址和端口号是否与所指定的回馈服务器地址和端口号相匹配。

DatagramSocket: 创建

DatagramSocket()
DatagramSocket(int localPort)
DatagramSocket(int localPort, InetAddress localAddr)

以上构造函数将创建一个 UDP 套接字。可以分别或同时设置本地端口和地址。如果没有指定本地端口，或将其设置为 0，该套接字将与任何可用的本地端口绑定。如果没有指定本地地址， 数据包（packet）可以接收发送向任何本地地址的数据报文。

DatagramSocket: 连接与关闭

void connect(InetAddress remoteAddr, int remotePort)
void connect(SocketAddress remoteSockAddr)

void disconnect()
void close()

connect()方法用来设置套接字的远程地址和端口。一旦连接成功，该套接字就只能与指定的地址和端口进行通信，任何向其他地址和端口发送数据报文的尝试都将抛出一个异常。套接字也将只接收从指定地址和端口发送来的数据报文，从其他地址或端口发送来的数据报文将被忽略。重点提示：连接到多播地址或广播地址的套接字只能发送数据报文，因为数据报文的源地址总是一个单播地址

DatagramSocket: 地址处理

InetAddress getInetAddress()
int getPort()
SocketAddress getRemoteSocketAddress()
InetAddress getLocalAddress()
int getLocalPort()
SocketAddress getLocalSocketAddress()

DatagramSocket: 发送和接收

void send(DatagramPacket packet)
void receive(DatagramPacket packet)

receive()方法将阻塞等待，直到接收到数据报文，并将报文中的数据复制到指定的 DatagramPacket 实例中。

DatagramSocket: 选项

int getSoTimeout()
void setSoTimeout(int timeoutMillis)

以上方法分别获取和设置该套接字中 receive()方法调用的最长阻塞时间。如果在接收到数据之前超时，则抛出 InterruptedIOException 异常。超时时间以毫秒为单位。

UDP 服务器端

与 TCP 服务器一样，UDP 服务器的工作是建立一个通信终端，并被动等待客户端发起连接。但由于 UDP 是无连接的，UDP 通信通过客户端的数据报文初始化，并没有 TCP 中建立连接那一步。典型的 UDP 服务器要执行以下三步：

1. 创建一个 DatagramSocket 实例，指定本地端口号，并可以选择指定本地地址。此时，服务器已经准备好从任何客户端接收数据报文。
2. 使用 DatagramSocket 类的 receive()方法来接收一个 DatagramPacket 实例。当

receive()方法返回时，数据报文就包含了客户端的地址，这样我们就知道了回复信息应该发送到什么地方。

3. 使用 DatagramSocket 类的 send() 和 receive()方法来发送和接收 DatagramPackets 实例，进行通信。

UDPEchoServer.java（UDP 版本的回馈服务器。非常简单：它不停地循环，接收数据报文后将相同的数据报文返回给客户端，规定：我们的服务器只接收和发送数据报文中的前 255（ECHOMAX）个字符，超出的部分将在套接字的具体实现中无提示地丢弃。）

[java] view plaincopy

```
1. <span style="font-size:14px;">import java.io.IOException;
2. import java.net.DatagramPacket;
3. import java.net.DatagramSocket;
4. 
5. public class UDPEchoServer {
6.     private static final int ECHOMAX = 255; // Maximum size of echo datagram
7. 
8.     public static void main(String[] args) throws IOException {
9.         if (args.length != 1) { // Test for correct argument list
10.             throw new IllegalArgumentException("Parameter(s): <Port>");
11.         }
12.         int servPort = Integer.parseInt(args[0]);
13.         DatagramSocket socket = new DatagramSocket(servPort);
14.         DatagramPacket packet = new DatagramPacket(new byte[ECHOMAX], ECHOMA
    X);
15.         while (true) { // Run forever, receiving and echoing datagrams
16.             socket.receive(packet); // Receive packet from client
17.             System.out.println("Handling client at "
18.                     + packet.getAddress().getHostAddress() + " on port "
19.                     + packet.getPort());
20.             socket.send(packet); // Send the same packet back to client
21.             packet.setLength(ECHOMAX); // Reset length to avoid shrinking bu
    ffer
22.         }
23.         /* NOT REACHED */
24.     }
25. }</span>
```

当在 TCP 套接字的输出流上调用的 write()方法返回后，所有的调用者都知道数据已经被复制到一个传输缓存区中，实际上此时数据可能已经被传送，也可能还没有被传送。而 UDP

协议没有提供从网络错误中恢复的机制，因此，并不对可能需要重传的数据进行缓存。这就意味着，当 send()方法调用返回时，消息已经被发送到了底层的传输信道中，并正处在（或即将处在）发送途中。

消息从网络到达后，其所包含数据被 read()方法或 receive()方法返回前，数据存储在一个先进先出（first-in, first-out，FIFO）的接收数据队列中。对于已连接的 TCP 套接字来说，所有已接收但还未传送的字节都看作是一个连续的字节序列（见第 6 章）。然而，对于 UDP 套接字来说，接收到的数据可能来自于不同的发送者。一个 UDP 套接字所接收的数据存放在一个消息队列中，每个消息都关联了其源地址信息。每次 receive()调用只返回一条消息。然而，如果 receive()方法在一个缓存区大小为 n 的 DatagramPacket 实例中调用，而接收队列中的第一条消息长度大于 n，则 receive()方法只返回这条消息的前 n 个字节。超出部分的其他字节都将自动被丢弃，而且对接收程序也没有任何消息丢失的提示！出于这个原因，接收者应该提供一个有足够大的缓存空间的 DatagramPacket 实例，以完整地存放调用 receive()方法时应用程序协议所允许的最大长度的消息。这个技术能够保证数据不会丢失。一个 DatagramPacket 实例中所运行传输的最大数据量为 65507 字节，即 UDP 数据报文所能负载的最多数据。因此，使用一个有 65600 字节左右缓存数组的数据包总是安全的。
每一个 DatagramPacket 实例都包含一个内部消息长度值，而该实例一接收到新消息，这个长度值都可能改变（以反映实际接收的消息的字节数）。如果一个应用程序使用同一个 DatagramPacket 实例多次调用 receive()方法，每次调用前就必须显式地将消息的内部长度重置为缓存区的实际长度。另一个潜在的问题根源是 DatagramPacket 类的 getData()方法，该方法总是返回缓冲区的原始大小，忽略了实际数据的内部偏移量和长度信息。消息接收到 DatagramPacket 的缓存区时，只是修改了存放消息数据的地址。在 Java1.6 中我们可以使用 Arrays.copyOfRange()方法，只需要一步就能方便地实现以上功能：byte[] destBuf = Arrays.copyOfRange(dg.getData(),dg.getOffset(), dg.getOffset()+dg.getLength());

3：发送和接收数据

任何要交换信息的程序之间在信息的编码方式上必须达成共识（如将信息表示为位序列），以及哪个程序发送信息，什么时候和怎样接收信息都将影响程序的行为。程序间达成的这种包含了信息交换的形式和意义的共识称为协议，用来实现特定应用程序的协议叫做应用程序协议，客户端和服务器的行为都要依赖于它们所交换的信息，因此应用程序协议通常更加复杂。

大部分的应用程序协议是根据由字段序列组成的离散信息定义的，其中每个字段中都包含了一段以位序列编码的特定的信息。应用程序协议中明确定义了信息的发送者应该怎样排列和解释这些位序列，同时还要定义接收者应该怎样解析，这样才使信息的接收者能够抽取出每个字段的意义。TCP/IP 协议的唯一约束是，信息必须在块（chunks）中发送和接收，而块的长度必须是 8 位的倍数，因此，我们可以认为在 TCP/IP 协议中传输的信息是字节序列。鉴于此，我们可以进一步把传输的信息看作数字序列或数组，每个数字的取值范围是 0 到 255。

信息编码

OutputStream、InputStream、DatagramPacket 实例中所能处理的唯一数据类型是字节和字节数组。作为一种强类型语言，Java 需要把其他数据类型（int，String 等）显式转换成字节数组。

(1)使用"位操作（bit-diddling）"将消息的正确值存入字节数组

[java] view plaincopy

```
1.  <span style="font-size:14px;">public class BruteForceCoding {
2.      private static byte byteVal = 101; // one hundred and one
3.      private static short shortVal = 10001; // ten thousand and one
4.      private static int intVal = 100000001; // one hundred million and one
5.      private static long longVal = 1000000000001L;// one trillion and one
6.
7.      private final static int BSIZE = Byte.SIZE / Byte.SIZE;
8.      private final static int SSIZE = Short.SIZE / Byte.SIZE;
9.      private final static int ISIZE = Integer.SIZE / Byte.SIZE;
10.     private final static int LSIZE = Long.SIZE / Byte.SIZE;
11.
12.     private final static int BYTEMASK = 0xFF; // 8 bits
13.
14.     public static String byteArrayToDecimalString(byte[] bArray) {
15.         StringBuilder rtn = new StringBuilder();
16.         for (byte b : bArray) {
17.             rtn.append(b & BYTEMASK).append(" ");
18.         }
19.         return rtn.toString();
20.     }
21.
22.     // Warning: Untested preconditions (e.g., 0 <= size <= 8)
23.     public static int encodeIntBigEndian(byte[] dst, long val, int offset,
24.             int size) {
```

```
25.         for (int i = 0; i < size; i++) {
26.             dst[offset++] = (byte) (val >> ((size - i - 1) * Byte.SIZE));
27.         }
28.         return offset;
29.     }
30.
31.     // Warning: Untested preconditions (e.g., 0 <= size <= 8)
32.     public static long decodeIntBigEndian(byte[] val, int offset, int size)
    {
33.         long rtn = 0;
34.         for (int i = 0; i < size; i++) {
35.             rtn = (rtn << Byte.SIZE) | ((long) val[offset + i] & BYTEMASK);

36.         }
37.         return rtn;
38.     }
39.
40.     public static void main(String[] args) {
41.         byte[] message = new byte[BSIZE + SSIZE + ISIZE + LSIZE];
42.         // Encode the fields in the target byte array
43.         int offset = encodeIntBigEndian(message, byteVal, 0, BSIZE);
44.         offset = encodeIntBigEndian(message, shortVal, offset, SSIZE);
45.         offset = encodeIntBigEndian(message, intVal, offset, ISIZE);
46.         encodeIntBigEndian(message, longVal, offset, LSIZE);
47.         System.out.println("Encoded message: "+ byteArrayToDecimalString(mes
    sage));
48.
49.         // Decode several fields
50.         long value = decodeIntBigEndian(message, BSIZE, SSIZE);
51.         System.out.println("Decoded short = " + value);
52.         value = decodeIntBigEndian(message, BSIZE + SSIZE + ISIZE, LSIZE);
53.         System.out.println("Decoded long = " + value);
54.
55.         // Demonstrate dangers of conversion
56.         offset = 4;
57.         value = decodeIntBigEndian(message, offset, BSIZE);
58.         System.out.println("Decoded value (offset " + offset + ", size "+ BS
    IZE + ") = " + value);
59.         byte bVal = (byte) decodeIntBigEndian(message, offset, BSIZE);
60.         System.out.println("Same value as byte = " + bVal);
61.     }
62.
63. }</span>
```

上面的强制（brute-force）编码方法需要程序员做很多工作：要计算和命名每个数值的偏移量和大小，并要为编码过程提供合适的参数。如果没有将 encodeIntBigEndian()方法提出来作为一个独立的方法，情况会更糟。基于以上原因，强制编码方法是不推荐使用的，而且 Java 也提供了一些更加易用的内置机制。不过，值得注意的是强制编码方法也有它的优势，除了能够对标准的 Java 整型进行编码外，encodeIntegerBigEndian() 方法对 1 到 8 字节的任何整数都适用--例如，如果愿意的话，你可以对一个 7 字节的整数进行编码。

(2)使用 Java 的内置工具将消息的正确值存入字节数组

所幸的是 Java 的内置工具能够帮助我们完成这些转换。如 String 类的 getBytes()方法，该方法就是将一个 Sring 实例中的字符转换成字节的标准方式，如 DataOutputStream 类和 ByteArrayOutputStre 类,DataOutputStream 类允许你将基本数据类型，如整型，写入一个流中：它提供了 writeByte()，writeShort()，writeInt()，以及 writeLong()方法，这些方法按照 big-endian 顺序，将整数以适当大小的二进制补码的形式写到流中。ByteArrayOutputStream 类获取写到流中的字节序列，并将其转换成一个字节数组。用这两个类来构建我们的消息的代码:

```
static byte byteVal = 101; // one hundred and one

static short shortVal = 10001; // ten thousand and one

static int intVal = 100000001; // onehundred million and one

static long longVal = 1000000000001L;// one trillion and one

ByteArrayOutputStream buf = new ByteArrayOutputStream();

DataOutputStream out = new DataOutputStream(buf);

out.writeByte(byteVal);

out.writeShort(shortVal);

out.writeInt(intVal);

out.writeLong(longVal);

out.flush();
```

byte[] msg = buf.toByteArray();

接收方将如何恢复传输的数据呢？正如你想的那样，Java 中也提供了与输出工具类相似的输入工具类，分别是 DataInputStream 类和 ByteArrayInputStream 类。

基本整型

发送者和接收者必须先在一些方面达成共识。

(1)要传输的每个整数的字节大小（size）

Java 程序中，int 数据类型由 32 位表示，因此，我们可以使用 4 个字节来传输任意的 int 型变量或常量；short 数据类型由 16 位表示，传输 short 类型的数据只需要两个字节；同理，传输 64 位的 long 类型数据则需要 8 个字节。

(2)字节的发送顺序

有两种选择：从整数的右边开始，由低位到高位地发送，即 little-endian 顺序；或从左边开始，由高位到低位发送，即 big-endian 顺序。对于任何多字节的整数，发送者和接收者必须在使用 big-endian 顺序还是使用 little-endian 顺序上达成共识。（注意，幸运的是字节中位的顺序在实现时是以标准的方式处理的,以 big-endian 顺序为主）

(3)所传输的数值是有符号的（signed）还是无符号的（unsigned）

Java 中的四种基本整型都是有符号的，它们的值以二进制补码（two's-complement）的方式存储，由于 Java 并不支持无符号整型，如果要在 Java 中编码和解码无符号数，则需要做一点额外的工作。

字符串和文本

发送者与接收者必须在符号与整数的映射方式上即字符集编码达成共识，，在一组符号与一组整数之间的映射称为编码字符集（coded character set.）,Java 使用了一种称为 Unicode 的国际标准编码字符集来表示 char 型和 String 型值。Unicode 字符集将"世界上大部分的语言和符号映射到整数 0 至 65535 之间，能更好地适用于国际化程序。Unicode 包含了 ASCII 码：每个 ASCII 码中定义的符号在 Unicode 中所映射整数与其在 ASCII 码中映射的整数相同。这就为 ASCII 与 Unicode 之间提供了一定程度的向后兼容性。

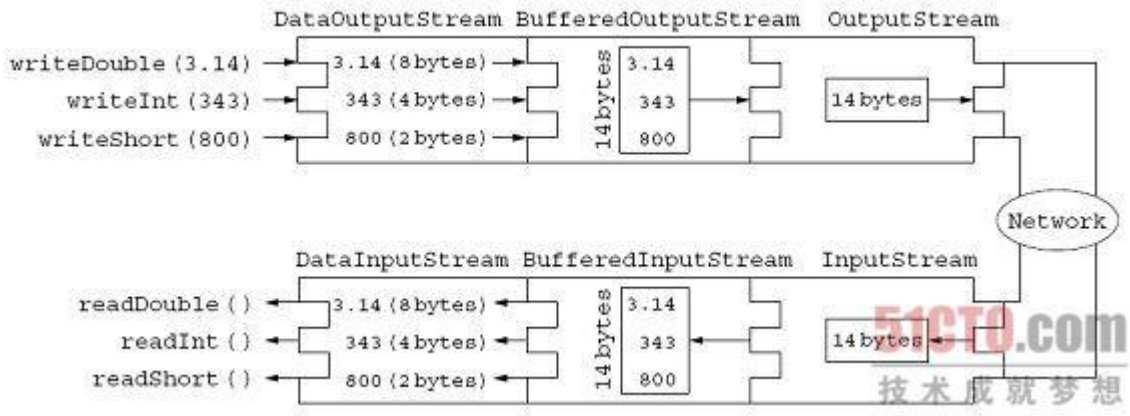
组合输入输出流

Java 中与流相关的类可以组合起来从而提供强大的功能。例如，我们可以将一个 Socket

实例的 OutputStream 包装在一个 BufferedOutputStream 实例中，这样可以先将字节暂时缓存在一起，然后再一次全部发送到底层的通信信道中，以提高程序的性能。我们还能再将这个 BufferedOutputStream 实例包裹在一个 DataOutputStream 实例中，以实现发送基本数据类型的功能。以下是实现这种组合的代码：

```
Socket socket = new Socket(server, port);

DataOutputStream out = new DataOutputStream( new
BufferedOutputStream(socket.getOutputStream()));
```



在这个例子中，我们先将基本数据的值，一个一个写入 DataOutputStream 中，DataOutputStream 再将这些数据以二进制的形式写入 BufferedOutputStream 将三次写入的数据缓存起来，然后再由 BufferedOutputStream 一次性地将这些数据写入套接字的 OutputStream，最后由 OutputStream 将数据发送到网络。在另一个终端，我们创建了相应的组合 InputStream，以有效地接收基本数据类型。

成帧与解析

将数据转换成在线路上传输的格式只完成了一半工作，在接收端还必须将接收到的字节序列还原成原始信息。应用程序协议通常处理的是由一组字段组成的离散的信息。成帧（Framing）技术则解决了接收端如何定位消息的首尾位置的问题。无论信息是编码成了文本、多字节二进制数、或是两者的结合，应用程序协议必须指定消息的接收者如何确定何时消息已完整接收。

如果一条完整的消息负载在一个 DatagramPacket 中发送，这个问题就变得很简单了：DatagramPacket 负载的数据有一个确定的长度，接收者能够准确地知道消息的结束位置。然而，如果通过 TCP 套接字来发送消息，情况将变得更复杂，因为 TCP 协议中没有消息边界的概念。如果一个消息中的所有字段都有固定的长度，同时每个消息又是由固定数量的字段组成的话，消息的长度就能够确定，接收者就可以简单地将消息长度对应的字

节数读到一个 byte[]缓存区中。但是如果消息的长度是可变的（例如消息中包含了一些变长的文本字符串），我们事先就无法知道需要读取多少字节。

如果接收者试图从套接字中读取比消息本身更多的字节，将可能发生以下两种情况之一：如果信道中没有其他消息，接收者将阻塞等待，同时无法处理接收到的消息；如果发送者也在等待接收端的响应信息，则会形成死锁（deadlock）；另一方面，如果信道中还有其他消息，则接收者会将后面消息的一部分甚至全部读到第一条消息中去，这将产生一些协议错误。因此，在使用 TCP 套接字时，成帧就是一个非常重要的考虑因素。
一些相同的考虑也适用于查找消息中每个字段的边界：接收者需要知道每个字段的结束位置和下一个字段的开始位置。因此，我们在此介绍的消息成帧技术几乎都可以应用到字段上。然而，最简单并使代码最简洁的方法是将这两个问题分开处理：首先定位消息的结束位置，然后将消息作为一个整体进行解析。

主要有两个技术使接收者能够准确地找到消息的结束位置：
(1)基于定界符（Delimiter-based）：消息的结束由一个唯一的标记（unique marker,）指出，即发送者在传输完数据后显式添加的一个特殊字节序列。这个特殊标记不能在传输的数据中出现。

(2)显式长度（Explicit length）：在变长字段或消息前附加一个固定大小的字段，用来指示该字段或消息中包含了多少字节。

基于定界符的方法通常用在以文本方式编码的消息中：定义一个特殊的字符或字符串来标识消息的结束。接收者只需要简单地扫描输入信息（以字节的方式）来查找定界序列，并将定界符前面的字符串返回。这种方法的缺点是消息本身不能包含有定界字符，否则接收者将提前认为消息已经结束。在基于定界符的成帧方法中，发送者要保证满足这个先决条件。缺点是发送者和接收者双方都必须扫描消息。
基于长度的方法更简单一些，不过要使用这种方法必须知道消息长度的上限。发送者先要确定消息的长度，将长度信息存入一个整数，作为消息的前缀。消息的长度上限定义了用来编码消息长度所需要的字节数：如果消息的长度小于 256 字节，则需要 1 个字节；如果消息的长度小于 65536 字节，则需要 2 个字节，等等。

为了展示以上技术，我们将介绍下面定义的 Framer 接口。它有两个方法：frameMsg()方法用来添加成帧信息并将指定消息输出到指定流，nextMsg()方法则扫描指定的流，从中抽取出下一条消息。
Framer.java

[java] view plaincopy

```
1. <span style="font-size:14px;">import java.io.IOException;
2. import java.io.OutputStream;
3. 
4. public interface Framer {
5.     void frameMsg(byte[] message, OutputStream out) throws IOException;
6.     byte[] nextMsg() throws IOException;
7. }</span>
```

DelimFramer.java 类实现了基于定界符的成帧方法，其定界符为"换行"符（"\n", 字节值为10）。 frameMethod()方法并没有实现填充，当成帧的字节序列中包含有定界符时，它只是简单地抛出异常。（扩展该方法以实现填充功能：结束符\n，数据中的\n-->ESCy，数据中的 ESC-->ESCz,避免数据中碰巧出现 ESCy 时而被误转化为\n，ESC 被称为转义符）nextMsg()方法扫描流，直到读取到了定界符，并返回定界符前面的所有字符，如果流为空则返回 null。如果累积了一个消息的不少字符，但直到流结束也没有找到定界符，程序将抛出一个异常来指示成帧错误。

DelimFramer.java

[java] view plaincopy

```
1. <span style="font-size:14px;">import java.io.ByteArrayOutputStream;
2. import java.io.EOFException;
3. import java.io.IOException;
4. import java.io.InputStream;
5. import java.io.OutputStream;
6. 
7. public class DelimFramer implements Framer {
8. 
9.     private InputStream in; // data source
10.     private static final byte DELIMITER = "\n"; // message delimiter
11. 
12.     public DelimFramer(InputStream in) {
13.         this.in = in;
14.     }
15. 
16.     public void frameMsg(byte[] message, OutputStream out) throws IOExceptio
    n {
17.         // ensure that the message does not contain the delimiter
18.         for (byte b : message) {
19.             if (b == DELIMITER) {
20.                 throw new IOException("Message contains delimiter");
21.             }
22.         }
23.         out.write(message);
24.         out.write(DELIMITER);
```

```
25.         out.flush();
26.     }
27.
28.     public byte[] nextMsg() throws IOException {
29.         ByteArrayOutputStream messageBuffer = new ByteArrayOutputStream();
30.         int nextByte;
31.         // fetch bytes until find delimiter
32.         while ((nextByte = in.read()) != DELIMITER) {
33.             if (nextByte == -1) { // end of stream?
34.                 if (messageBuffer.size() == 0) { // if no byte read
35.                     return null;
36.                 } else { // if bytes followed by end of stream: framing erro
    r
37.                     throw new EOFException("Non-
    empty message without delimiter");
38.                 }
39.             }
40.             messageBuffer.write(nextByte); // write byte to buffer
41.         }
42.         return messageBuffer.toByteArray();
43.     }
44. }</span>
```

LengthFramer.java 类实现了基于长度的成帧方法，适用于长度小于 65535 (216？1)字节的消息。发送者首先给出指定消息的长度，并将长度信息以 big-endian 顺序存入两个字节的整数中，再将这两个字节放在完整的消息内容前，连同消息一起写入输出流。在接收端，我们使用 DataInputStream 以读取整型的长度信息；readFully() 方法将阻塞等待，直到给定的

数组完全填满，这正是我们需要的。值得注意的是，使用这种成帧方法，发送者不需要检查要成帧的消息内容，而只需要检查消息的长度是否超出了限制。

LengthFramer.java

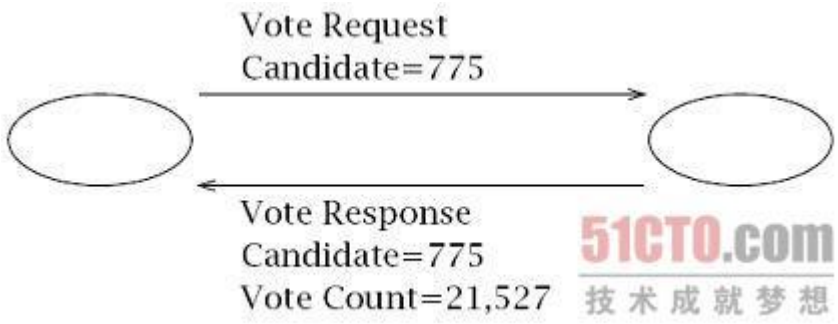
[java] view plaincopy

```
1.  <span style="font-size:14px;">import java.io.DataInputStream;
2.  import java.io.EOFException;
3.  import java.io.IOException;
4.  import java.io.InputStream;
5.  import java.io.OutputStream;
6.
7.  public class LengthFramer implements Framer {
8.      public static final int MAXMESSAGELENGTH = 65535;
9.      public static final int BYTEMASK = 0xff;
10.     public static final int SHORTMASK = 0xffff;
```

```
11.     public static final int BYTESHIFT = 8;
12.
13.     private DataInputStream in; // wrapper for data I/O
14.
15.     public LengthFramer(InputStream in) throws IOException {
16.         this.in = new DataInputStream(in);
17.     }
18.
19.     public void frameMsg(byte[] message, OutputStream out) throws IOExceptio
    n {
20.         if (message.length > MAXMESSAGELENGTH) {
21.             throw new IOException("message too long");
22.         }
23.         // write length prefix
24.         out.write((message.length >> BYTESHIFT) & BYTEMASK);//OutPutStream.w
    rite(int b)只写 b 的低位 8bit 到流中，高 24bit 忽略
25.         out.write(message.length & BYTEMASK);
26.         // write message
27.         out.write(message);
28.         out.flush();
29.     }
30.
31.     public byte[] nextMsg() throws IOException {
32.         int length;
33.         try {
34.             length = in.readUnsignedShort(); // read 2 bytes
35.         } catch (EOFException e) { // no (or 1 byte) message
36.             return null;
37.         }
38.         // 0 <= length <= 65535
39.         byte[] msg = new byte[length];
40.         in.readFully(msg); // if exception, it's a framing error.
41.         return msg;
42.     }
43. }</span>
```

构建和解析协议消息

简单的"投票"协议，如图所示。一个客户端向服务器发送了一个请求消息，消息中包含了一个候选人 ID，范围是 0 至 1000。



程序支持两种请求。一种是查询（inquiry），即向服务器询问给定候选人当前获得的投票总数。服务器发回一个响应消息，包含了原来的候选人 ID 和该候选人当前（查询请求收到时）获得的选票总数。另一种是投票（voting）请求，即向指定候选人投一票。服务器对这种请求也发回响应消息，包含了候选人 ID 和其获得的选票数（包括了刚投的一票）。

在实现一个协议时，定义一个专门的类来存放消息中所包含的信息是大有裨益的。该类提供了操作消息中的字段的方法--同时用来维护不同字段之间的不变量。在我们的例子中，客户端和服务器端发送的消息都非常简单，它们唯一的区别是服务器端发送的消息包含了选票总数和一个表示响应消息（不是请求消息）的标志。因此，我们可以用一个类来表示客户端和服务器端的两种消息。

消息类 VoteMsg（展示了每条消息中的基本信息）

布尔值 isInquiry，其值为 true 时表示该消息是查询请求（为 false 时表示该消息是投票信息）；

布尔值 isResponse，指示该消息是响应（由服务器发送）还是请求；

整型变量 candidateID 指示了候选人的 ID；

长整型变量 voteCount 指示出所查询的候选人获得的总选票数。

这个类还维护了以下字段间的不变量：

candidateID 的范围是 0 到 1000。

voteCount 在响应消息中只能是一个非零值（isResponse 为 true）。

voteCount 不能为负数。

[java] view plaincopy

```
1.  <span style="font-size:14px;">public class VoteMsg {
2.      private boolean isInquiry; // true if inquiry; false if vote
3.      private boolean isResponse;// true if response from server
4.      private int candidateID; // in [0,1000]
5.      private long voteCount; // nonzero only in response
6.
7.      public static final int MAX_CANDIDATE_ID = 1000;
8.
```

```
9.      public VoteMsg(boolean isResponse, boolean isInquiry, int candidateID,
10.             long voteCount) throws IllegalArgumentException {
11.         // check invariants
12.         if (voteCount != 0 && !isResponse) {
13.             throw new IllegalArgumentException("Request vote count must be z
    ero");
14.         }
15.         if (candidateID < 0 || candidateID > MAX_CANDIDATE_ID) {
16.             throw new IllegalArgumentException("Bad Candidate ID: "+ candida
    teID);
17.         }
18.         if (voteCount < 0) {
19.             throw new IllegalArgumentException("Total must be >= zero");
20.         }
21.         this.candidateID = candidateID;
22.         this.isResponse = isResponse;
23.         this.isInquiry = isInquiry;
24.         this.voteCount = voteCount;
25.     }
26.
27.     public void setInquiry(boolean isInquiry) {
28.         this.isInquiry = isInquiry;
29.     }
30.
31.     public void setResponse(boolean isResponse) {
32.         this.isResponse = isResponse;
33.     }
34.
35.     public boolean isInquiry() {
36.         return isInquiry;
37.     }
38.
39.     public boolean isResponse() {
40.         return isResponse;
41.     }
42.
43.     public void setCandidateID(int candidateID) throws IllegalArgumentExcept
    ion {
44.         if (candidateID < 0 || candidateID > MAX_CANDIDATE_ID) {
45.             throw new IllegalArgumentException("Bad Candidate ID: "+ candida
    teID);
46.         }
47.         this.candidateID = candidateID;
48.     }
```

```
49.
50.     public int getCandidateID() {
51.         return candidateID;
52.     }
53.
54.     public void setVoteCount(long count) {
55.         if ((count != 0 && !isResponse) || count < 0) {
56.             throw new IllegalArgumentException("Bad vote count");
57.         }
58.         voteCount = count;
59.     }
60.
61.     public long getVoteCount() {
62.         return voteCount;
63.     }
64.
65.     public String toString() {
66.         String res = (isInquiry ? "inquiry" : "vote") + " for candidate "+ c
    andidateID;
67.         if (isResponse) {
68.             res = "response to " + res + " who now has " + voteCount+ " vote
    (s)";
69.         }
70.         return res;
71.     }
72. }</span>
```

编码和解码类接口 VoteMsgCoder

VoteMsgCoder 接口提供了对投票消息进行序列化和反序列化的方法

VoteMsgCoder.java

[java] view plaincopy

```
1.  <span style="font-size:14px;">import java.io.IOException;
2.
3.  public interface VoteMsgCoder {
4.      byte[] toWire(VoteMsg msg) throws IOException;
5.
6.      VoteMsg fromWire(byte[] input) throws IOException;
7.  }</span>
```

toWire()方法用于根据一个特定的协议，将投票消息转换成一个字节序列，fromWire()方法则根据相同的协议，对给定的字节序列进行解析，并根据信息的内容构造出消息类的一个实例。

为了介绍不同的信息编码方法，我们展示了两个实现 VoteMsgCoder 接口的类。一个使用的是基于文本的编码方式，另一个使用的是二进制的编码方式。

基于文本的编码、解码类 VoteMsgTextCoder

用文本方式对消息进行编码的版本。该协议指定使用 US-ASCII 字符集对文本进行编码。消息的开头是一个所谓的"魔术字符串"，即一个字符序列，用于接收
者快速将投票协议的消息和网络中随机到来的垃圾消息区分开。投票/查询布尔值被编码成字符形式，'v'表示投票消息，'i'表示查询消息。消息的状态，即是否为服务器的响应，由字符'R'指示。状态标记后面是候选人 ID，其后跟的是选票总数，它们都编码成十进制字符串。

VoteMsgTextCoder.java

[java] view plaincopy

```
1.  <span style="font-size:14px;">import java.io.ByteArrayInputStream;
2.  import java.io.IOException;
3.  import java.io.InputStreamReader;
4.  import java.util.Scanner;
5.
6.  public class VoteMsgTextCoder implements VoteMsgCoder {
7.      /*
8.       * Wire Format "VOTEPROTO" <"v"|"i"> [<RESPFLAG>] <CANDIDATE> [<VOTECNT>
    ]
9.       * Charset is fixed by the wire format.
10.      */
11.     // Manifest constants for encoding
12.     public static final String MAGIC = "Voting";
13.     public static final String VOTESTR = "v";
14.     public static final String INQSTR = "i";
15.     public static final String RESPONSESTR = "R";
16.     public static final String CHARSETNAME = "US-ASCII";
17.     public static final String DELIMSTR = " ";
18.     public static final int MAX_WIRE_LENGTH = 2000;
19.
20.     public byte[] toWire(VoteMsg msg) throws IOException {
21.         String msgString = MAGIC + DELIMSTR
```

```
                + (msg.isInquiry() ? INQSTR : VOTESTR) + DELIMSTR
                + (msg.isResponse() ? RESPONSESTR + DELIMSTR : "")
                + Integer.toString(msg.getCandidateID()) + DELIMSTR
                + Long.toString(msg.getVoteCount());
        byte data[] = msgString.getBytes(CHARSETNAME);
        return data;
    }

    public VoteMsg fromWire(byte[] message) throws IOException {
        ByteArrayInputStream msgStream = new ByteArrayInputStream(message);

        Scanner s = new Scanner(new InputStreamReader(msgStream, CHARSETNAME
    ));
        boolean isInquiry;
        boolean isResponse;
        int candidateID;
        long voteCount;
        String token;
        try {
            token = s.next();
            if (!token.equals(MAGIC)) {
                throw new IOException("Bad magic string: " + token);
            }
            token = s.next();
            if (token.equals(VOTESTR)) {
                isInquiry = false;
            } else if (!token.equals(INQSTR)) {
                throw new IOException("Bad vote/inq indicator: " + token);
            } else {
                isInquiry = true;
            }
            token = s.next();
            if (token.equals(RESPONSESTR)) {
                isResponse = true;
                token = s.next();
            } else {
                isResponse = false;
            }
            // Current token is candidateID
            // Note: isResponse now valid
            candidateID = Integer.parseInt(token);
            if (isResponse) {
                token = s.next();
                voteCount = Long.parseLong(token);
```

```
64.             } else {
65.                 voteCount = 0;
66.             }
67.         } catch (IOException ioe) {
68.             throw new IOException("Parse error...");
69.         }
70.         return new VoteMsg(isResponse, isInquiry, candidateID, voteCount);
71.     }
72. }</span>
```

toWire()方法简单地创建一个字符串，该字符串中包含了消息的所有字段，并由空白符隔开。fromWire()方法首先检查"魔术"字符串，如果在消息最前面没有魔术字符串，则抛出一个异常。这里说明了在实现协议时非常重要的一点：永远不要对从网络来的任何输入进行任何假设。你的程序必须时刻为任何可能的输入做好准备，并能够很好地对其进行处理。在这个例子中，如果接收到的不是期望的消息，fromWire()方法将抛出一个异常，否则，就使用 Scanner 实例，根据空白符一个一个地获取字段。注意，消息的字段数与其是请求消息（由客户端发送）还是响应消息（由服务器发送）有关。如果输入流提前结束或格式错误，fromWire()方法将抛出一个异常。

基于二进制的编码、解码类 VoteMsgBinCoder

与基于文本的格式相反，二进制格式使用固定大小的消息。每条消息由一个特殊字节开始，该字节的最高六位为一个"魔术"值 010101。这一点少量的冗余信息为接收者收到适当的投票消息提供了一定程度的保证。该字节的最低两位对两个布尔值进行了编码。消息的第二个字节总是 0，第三、第四个字节包含了 candidateID 值。只有响应消息的最后 8 个字节才包含了选票总数信息。

[java] view plaincopy

```
1.  <span style="font-size:14px;">import java.io.ByteArrayInputStream;
2.  import java.io.ByteArrayOutputStream;
3.  import java.io.DataInputStream;
4.  import java.io.DataOutputStream;
5.  import java.io.IOException;
6.
7.  /* Wire Format
8.   * 1 1 1 1 1 1
9.   * 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
10.  * +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
11.  * | Magic |Flags| ZERO |
12.  * +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
13.  * | Candidate ID |
14.  * +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
15.  * | |
```

```
 * | Vote Count (only in response) |
 * | |
 * | |
 * +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 */
public class VoteMsgBinCoder implements VoteMsgCoder {
    // manifest constants for encoding
    public static final int MIN_WIRE_LENGTH = 4;
    public static final int MAX_WIRE_LENGTH = 16;
    public static final int MAGIC = 0x5400;
    public static final int MAGIC_MASK = 0xfc00;
    public static final int MAGIC_SHIFT = 8;
    public static final int RESPONSE_FLAG = 0x0200;
    public static final int INQUIRE_FLAG = 0x0100;

    public byte[] toWire(VoteMsg msg) throws IOException {
        ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
        DataOutputStream out = new DataOutputStream(byteStream); // converts
ints

        short magicAndFlags = MAGIC;
        if (msg.isInquiry()) {
            magicAndFlags |= INQUIRE_FLAG;
        }
        if (msg.isResponse()) {
            magicAndFlags |= RESPONSE_FLAG;
        }
        out.writeShort(magicAndFlags);
        // We know the candidate ID will fit in a short: it's > 0 && < 1000

        out.writeShort((short) msg.getCandidateID());
        if (msg.isResponse()) {
            out.writeLong(msg.getVoteCount());
        }
        out.flush();
        byte[] data = byteStream.toByteArray();
        return data;
    }

    public VoteMsg fromWire(byte[] input) throws IOException {
        // sanity checks
        if (input.length < MIN_WIRE_LENGTH) {
            throw new IOException("Runt message");
        }
```

```
58.         ByteArrayInputStream bs = new ByteArrayInputStream(input);
59.         DataInputStream in = new DataInputStream(bs);
60.         int magic = in.readShort();
61.         if ((magic & MAGIC_MASK) != MAGIC) {
62.             throw new IOException("Bad Magic #: "+ ((magic & MAGIC_MASK) >>
    MAGIC_SHIFT));
63.         }
64.         boolean resp = ((magic & RESPONSE_FLAG) != 0);
65.         boolean inq = ((magic & INQUIRE_FLAG) != 0);
66.         int candidateID = in.readShort();
67.         if (candidateID < 0 || candidateID > 1000) {
68.             throw new IOException("Bad candidate ID: " + candidateID);
69.         }
70.         long count = 0;
71.         if (resp) {
72.             count = in.readLong();
73.             if (count < 0) {
74.                 throw new IOException("Bad vote count: " + count);
75.             }
76.         }
77.         // Ignore any extra bytes
78.         return new VoteMsg(resp, inq, candidateID, count);
79.     }
80. }</span>
```

服务器中记录投票过程的服务类 VoteService

通过流发送消息非常简单，只需要创建消息，调用 toWire()方法，添加适当的成帧信息，再写入流。当然，接收消息就要按照相反的顺序执行。这个过程适用于 TCP 协议，而对于 UDP 协议，不需要显式地成帧，因为 UDP 协议中保留了消息的边界信息。为了对发送与接收过程进行展示，我们考虑投票服务的如下几点：1）维护一个候选人 ID 与其获得选票数的映射，2）记录提交的投票，3）根据其获得的选票数，对查询指定的候选人和为其投票的消息做出响应。首先，我们实现一个投票服务器所用到的服务。当接收到投票消息时，投票服务器将调用 VoteService 类的 handleRequest() 方法对请求进行处理。

[java] view plaincopy

```
1.  <span style="font-size:14px;">import java.util.HashMap;
2.  import java.util.Map;
3.
4.  public class VoteService {
5.      // Map of candidates to number of votes
6.      private Map<Integer, Long> results = new HashMap<Integer, Long>();
```

```
7.
8.      public VoteMsg handleRequest(VoteMsg msg) {
9.          if (msg.isResponse()) { // If response, just send it back
10.             return msg;
11.         }
12.         msg.setResponse(true); // Make message a response
13.         // Get candidate ID and vote count
14.         int candidate = msg.getCandidateID();
15.         Long count = results.get(candidate);
16.         if (count == null) {
17.             count = 0L; // Candidate does not exist
18.         }
19.         if (!msg.isInquiry()) {
20.             results.put(candidate, ++count); // If vote, increment count
21.         }
22.         msg.setVoteCount(count);
23.         return msg;
24.     }
25. }
26. </span>
```

TCP 投票客户端类 VoteClientTCP

该客户端通过 TCP 套接字连接到投票服务器，在一次投票后发送一个查询请求，并接收查询和投票结果。

发送：消息对象-->编码/解码对象将消息对象编码成字节数组-->成帧/解帧对象将字节数组成帧后通过输出流发送

接收：成帧/解帧对象将接收到的输入流解帧成字节数组-->编码/解码对象将字节数组解码成消息对象-->消息对象

[java] view plaincopy

```
1.  <span style="font-size:14px;">import java.io.OutputStream;
2.  import java.net.Socket;
3.
4.  public class VoteClientTCP {
5.      public static final int CANDIDATEID = 888;
6.
7.      public static void main(String args[]) throws Exception {
8.          if (args.length != 2) { // Test for correct # of args
```

```
            throw new IllegalArgumentException("Parameter(s): <Server> <Port>");
        }
        String destAddr = args[0]; // Destination address
        int destPort = Integer.parseInt(args[1]); // Destination port

        Socket sock = new Socket(destAddr, destPort);
        OutputStream out = sock.getOutputStream();
        // Change Bin to Text for a different framing strategy
        VoteMsgCoder coder = new VoteMsgBinCoder();
        // Change Length to Delim for a different encoding strategy
        Framer framer = new LengthFramer(sock.getInputStream());
        // Create an inquiry request (2nd arg = true)
        VoteMsg msg = new VoteMsg(false, true, CANDIDATEID, 0);
        byte[] encodedMsg = coder.toWire(msg);

        // Send request
        System.out.println("Sending Inquiry (" + encodedMsg.length+ " bytes) : ");
        System.out.println(msg);
        framer.frameMsg(encodedMsg, out);

        // Now send a vote
        msg.setInquiry(false);
        encodedMsg = coder.toWire(msg);
        System.out.println("Sending Vote (" + encodedMsg.length + " bytes): ");
        framer.frameMsg(encodedMsg, out);

        // Receive inquiry response
        encodedMsg = framer.nextMsg();
        msg = coder.fromWire(encodedMsg);
        System.out.println("Received Response (" + encodedMsg.length+ " bytes): ");
        System.out.println(msg);

        // Receive vote response
        msg = coder.fromWire(framer.nextMsg());
        System.out.println("Received Response (" + encodedMsg.length+ " bytes): ");
        System.out.println(msg);

        sock.close();
    }
```

```
48. }</span>
```

TCP 投票服务器端类 VoteServerTCP

该服务器反复地接收新的客户端连接，并使用 VoteService 类为客户端的投票消息作出响应。

[java] view plaincopy

```
1.  <span style="font-size:14px;">import java.io.IOException;
2.  import java.net.ServerSocket;
3.  import java.net.Socket;
4.
5.  public class VoteServerTCP {
6.
7.      public static void main(String args[]) throws Exception {
8.          if (args.length != 1) { // Test for correct # of args
9.              throw new IllegalArgumentException("Parameter(s): <Port>");
10.         }
11.         int port = Integer.parseInt(args[0]); // Receiving Port
12.         ServerSocket servSock = new ServerSocket(port);
13.         // Change Bin to Text on both client and server for different encodi
    ng
14.         VoteMsgCoder coder = new VoteMsgBinCoder();
15.         VoteService service = new VoteService();
16.         while (true) {
17.             Socket clntSock = servSock.accept();
18.             System.out.println("Handling client at "+ clntSock.getRemoteSock
    etAddress());
19.             // Change Length to Delim for a different framing strategy
20.             Framer framer = new LengthFramer(clntSock.getInputStream());
21.             try {
22.                 byte[] req;
23.                 while ((req = framer.nextMsg()) != null) {
24.                     System.out.println("Received message (" + req.length+ "
    bytes)");
25.                     VoteMsg responseMsg = service.handleRequest(coder.fromWi
    re(req));
26.                     framer.frameMsg(coder.toWire(responseMsg), clntSock.getO
    utputStream());
27.                 }
28.             } catch (IOException ioe) {
29.                 System.err.println("Error handling client: " + ioe.getMessag
    e());
30.             } finally {
31.                 System.out.println("Closing connection");
```

```
32.                 clntSock.close();
33.             }
34.         }
35.     }
36. }</span>
```

UDP 投票客户端类 VoteClientUDP

UDP 版本的投票客户端与 TCP 版本非常相似。需要注意的是，在 UDP 客户端中我们不需要使用成帧器，因为 UDP 协议为我们维护了消息的边界信息。对于 UDP 协议，我们使用基于文本的编码方式对消息进行编码，不过只要客户端与服务器能达成一致，也能够很方便地改成其他编码方式。

[java] view plaincopy

```
1.  <span style="font-size:14px;">import java.io.IOException;
2.  import java.net.DatagramPacket;
3.  import java.net.DatagramSocket;
4.  import java.net.InetAddress;
5.  import java.util.Arrays;
6.
7.  public class VoteClientUDP {
8.
9.      public static void main(String args[]) throws IOException {
10.         if (args.length != 3) { // Test for correct # of args
11.             throw new IllegalArgumentException("Parameter(s): <Destination>"
    + " <Port> <Candidate#>");
12.         }
13.         InetAddress destAddr = InetAddress.getByName(args[0]); // Destinatio
    n addr
14.         int destPort = Integer.parseInt(args[1]); // Destination port
15.         int candidate = Integer.parseInt(args[2]); // 0 <= candidate <= 1000
    req'd
16.         DatagramSocket sock = new DatagramSocket(); // UDP socket for sendin
    g
17.         sock.connect(destAddr, destPort);
18.         // Create a voting message (2nd param false = vote)
19.         VoteMsg vote = new VoteMsg(false, false, candidate, 0);
20.         // Change Text to Bin here for a different coding strategy
21.         VoteMsgCoder coder = new VoteMsgTextCoder();
22.         // Send request
23.         byte[] encodedVote = coder.toWire(vote);
24.         System.out.println("Sending Text-
    Encoded Request ("+ encodedVote.length + " bytes): ");
25.         System.out.println(vote);
```

```
26.         DatagramPacket message = new DatagramPacket(encodedVote,encodedVote.
    length);
27.         sock.send(message);
28.         // Receive response
29.         message = new DatagramPacket(new byte[VoteMsgTextCoder.MAX_WIRE_LENG
    TH],VoteMsgTextCoder.MAX_WIRE_LENGTH);
30.         sock.receive(message);
31.         encodedVote = Arrays.copyOfRange(message.getData(), 0, message.getLe
    ngth());
32.         System.out.println("Received Text-
    Encoded Response ("+ encodedVote.length + " bytes): ");
33.         vote = coder.fromWire(encodedVote);
34.         System.out.println(vote);
35.     }
36. }</span>
```

UDP 投票服务器端类 VoteServerUDP

UDP 投票服务器，同样，也与 TCP 版本非常相似。

[java] view plaincopy

```
1.  <span style="font-size:14px;">import java.io.IOException;
2.  import java.net.DatagramPacket;
3.  import java.net.DatagramSocket;
4.  import java.util.Arrays;
5.
6.  public class VoteServerUDP {
7.
8.      public static void main(String[] args) throws IOException {
9.
10.         if (args.length != 1) { // Test for correct # of args
11.             throw new IllegalArgumentException("Parameter(s): <Port>");
12.         }
13.         int port = Integer.parseInt(args[0]); // Receiving Port
14.         DatagramSocket sock = new DatagramSocket(port); // Receive socket
15.         byte[] inBuffer = new byte[VoteMsgTextCoder.MAX_WIRE_LENGTH];
16.         // Change Bin to Text for a different coding approach
17.         VoteMsgCoder coder = new VoteMsgTextCoder();
18.         VoteService service = new VoteService();
19.         while (true) {
20.             DatagramPacket packet = new DatagramPacket(inBuffer,inBuffer.len
    gth);
21.             sock.receive(packet);
22.             byte[] encodedMsg = Arrays.copyOfRange(packet.getData(), 0, pack
    et.getLength());
```

```
23.             System.out.println("Handling request from "+ packet.getSocketAddress() + " (" + encodedMsg.length+ " bytes)");
24.             try {
25.                 VoteMsg msg = coder.fromWire(encodedMsg);
26.                 msg = service.handleRequest(msg);
27.                 packet.setData(coder.toWire(msg));
28.                 System.out.println("Sending response (" + packet.getLength() + " bytes):");
29.                 System.out.println(msg);
30.                 sock.send(packet);
31.             } catch (IOException ioe) {
32.                 System.err.println("Parse error in message: "+ ioe.getMessage());
33.             }
34.         }
35.     }
36. }</span>
```

4：多任务处理

"迭代服务器（iterative server）"：按顺序处理客户端的请求，也就是说在完成了对前一客户端的服务后，才会对下一个客户端进行响应。这种服务器最适用于每个客户端所请求的连接时间都被限制在较小范围内的应用中，而对于允许客户端请求长时间服务的情况，后续客户端将面临无法接受的长时间等待。需要一种方法可以独立处理每一个连接，并使它们不会产生相互干扰，而 Java 的多线程技术刚好满足了这一需求，这一机制使服务器能够方便地同时处理多个客户端的请求。通过使用多线程，一个应用程序可以并行执行多项任务，就好像有多个 Java 虚拟机在同时运行。（实际上是多个线程共享了同一个 Java 虚拟机。）在我们的响应服务器中，可以为每个客户端分配一个执行线程来实现。

两种实现并行服务器（concurrent servers）的编程方法：

（1）一客户一线程（thread-per-client），即为每一个客户端连接创建一个新的线程；

（2）线程池（threadpool），即将客户端连接分配给一组事先创建好的线程。

如果客户端的执行过程涉及到需要更新服务器端线程间的共享信息，这将变得相当麻烦。在这种情况下，必须非常小心，以确保不同的线程间在共享数据上得到了妥善的同步，否则，会导致共享信息不一致的状况发生，更麻烦的是这些问题追踪起来还非常困难。

服务器协议类(封装了对每个客户端的处理过程，以回显程序为例)

EchoProtocol 中给出了回显协议的代码。这个类的静态方法 handleEchoClient()中封装了对每个客户端的处理过程。

[java] view plaincopy

```java
<span style="font-size:14px;">import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.util.logging.Level;
import java.util.logging.Logger;

public class EchoProtocol implements Runnable {
    private static final int BUFSIZE = 32; // Size (in bytes) of I/O buffer

    private Socket clntSock; // Socket connect to client
    private Logger logger; // Server logger

    public EchoProtocol(Socket clntSock, Logger logger) {
        this.clntSock = clntSock;
        this.logger = logger;
    }

    public static void handleEchoClient(Socket clntSock, Logger logger) {
        try {
            // Get the input and output I/O streams from socket
            InputStream in = clntSock.getInputStream();
            OutputStream out = clntSock.getOutputStream();
            int recvMsgSize; // Size of received message
            int totalBytesEchoed = 0; // Bytes received from client
            byte[] echoBuffer = new byte[BUFSIZE]; // Receive Buffer
            // Receive until client closes connection, indicated by -1
            while ((recvMsgSize = in.read(echoBuffer)) != -1) {
                out.write(echoBuffer, 0, recvMsgSize);
                totalBytesEchoed += recvMsgSize;
            }
            logger.info("Client " + clntSock.getRemoteSocketAddress()+ ", ec
hoed " + totalBytesEchoed + " bytes.");

        } catch (IOException ex) {
            logger.log(Level.WARNING, "Exception in echo protocol", ex);
        } finally {
            try {
                clntSock.close();
```

```
            } catch (IOException e) {
            }
        }
    }

    public void run() {
        handleEchoClient(clntSock, logger);
    }
}</span>
```

一客户一线程

在一客户一线程（thread-per-client）的服务器中，为每个连接都创建了一个新的线程来处理。服务器循环执行一些任务，在指定端口上侦听连接，反复接收客户端传入的连接请求，并为每个连接创建一个新的线程来对其进行处理。

[java] view plaincopy

```
<span style="font-size:14px;">import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.logging.Logger;

public class TCPEchoServerThread {

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

        if (args.length != 1) { // Test for correct # of args
            throw new IllegalArgumentException("Parameter(s): <Port>");
        }
        int echoServPort = Integer.parseInt(args[0]); // Server port
        // Create a server socket to accept client connection requests
        ServerSocket servSock = new ServerSocket(echoServPort);
        Logger logger = Logger.getLogger("practical");
        // Run forever, accepting and spawning a thread for each connection

        while (true) {
            Socket clntSock = servSock.accept(); // Block waiting for connec
tion
            // Spawn thread to handle new connection
            Thread thread = new Thread(new EchoProtocol(clntSock, logger));

            thread.start();
            logger.info("Created and started Thread " + thread.getName());
```

```
24.         }
25.         /* NOT REACHED */
26.     }
27. }</span>
```

线程池

每个新线程都会消耗系统资源：创建一个线程将占用 CPU 周期，而且每个线程都自己的数据结构（如，栈）也要消耗系统内存。另外，当一个线程阻塞（block）时，JVM 将保存其状态，选择另外一个线程运行，并在上下文转换（context switch）时恢复阻塞线程的状态。随着线程数的增加，线程将消耗越来越多的系统资源。这将最终导致系统花费更多的时间来处理上下文转换和线程管理，更少的时间来对连接进行服务。那种情况下，加入一个额外的线程实际上可能增加客户端总服务时间。

通过限制总线程数并重复使用线程来避免这个问题。与为每个连接创建一个新的线程不同，服务器在启动时创建一个由固定数量线程组成的线程池（thread pool）。当一个新的客户端连接请求传入服务器，它将交给线程池中的一个线程处理。当该线程处理完这个客户端后，又返回线程池，并为下一次请求处理做好准备。如果连接请求到达服务器时，线程池中的所有线程都已经被占用，它们则在一个队列中等待，直到有空闲的线程可用。

与一客户一线程服务器一样，线程池服务器首先创建一个 ServerSocket 实例。然后创建 N 个线程，每个线程都反复循环，从（共享的）ServerSocket 实例接收客户端连接。当多个线程同时调用同一个 ServerSocket 实例的 accept()方法时，它们都将阻塞等待，直到一个新的连接成功建立。然后系统选择一个线程，新建立的连接对应的 Socket 实例则只在选中的线程中返回。其他线程则继续阻塞，直到成功建立下一个连接和选中另一个幸运的线程。由于线程池中的所有线程都反复循环，一个接一个地处理客户端连接，线程池服务器的行为就像是一组迭代服务器。与一客户一线程服务器不同，线程池中的线程在完成对一个客户端的服务后并不终止，相反，它又重新开始在 accept()方法上阻塞等待。

[java] view plaincopy

```
1.  <span style="font-size:14px;">import java.io.IOException;
2.  import java.net.ServerSocket;
3.  import java.net.Socket;
4.  import java.util.logging.Level;
5.  import java.util.logging.Logger;
6.
7.  public class TCPEchoServerPool {
8.
9.      public static void main(String[] args) throws IOException {
```

```
10.
11.         if (args.length != 2) { // Test for correct # of args
12.             throw new IllegalArgumentException("Parameter(s): <Port> <Thread
    s>");
13.         }
14.         int echoServPort = Integer.parseInt(args[0]); // Server port
15.         int threadPoolSize = Integer.parseInt(args[1]);
16.
17.         // Create a server socket to accept client connection requests
18.         final ServerSocket servSock = new ServerSocket(echoServPort);
19.         final Logger logger = Logger.getLogger("practical");
20.         // Spawn a fixed number of threads to service clients
21.         for (int i = 0; i < threadPoolSize; i++) {
22.             Thread thread = new Thread() {
23.                 public void run() {
24.                     while (true) {
25.                         try {
26.                             Socket clntSock = servSock.accept(); // Wait for
    a connection
27.                             EchoProtocol.handleEchoClient(clntSock, logger);
    // Handle it
28.                         } catch (IOException ex) {
29.                             logger.log(Level.WARNING, "Client accept failed"
    ,
30.                                     ex);
31.                         }
32.                     }
33.                 }
34.             };
35.             thread.start();
36.             logger.info("Created and started Thread = " + thread.getName());

37.         }
38.     }
39. }</span>
```

由于线程的重复使用，线程池的方法只需要付出创建 N 次线程的系统开销，而与客户端连接总数无关。由于可以控制最大并发执行线程数，我们就可以控制线程的调度和资源开销。当然，如果我们创建的线程太少，客户端还是有可能等很长时间才获得服务，因此，线程池的大小需要根据负载情况进行调整，以使客户端连接的时间最短。理想的情况是有一个调度工具，可以在系统负载增加时扩展线程池的大小（低于大小上限），负载较轻时缩减线程池的大小。

利用 JDK 提供的线程池(java.util.concurrent 包中)来实现并行服务器

[java] view plaincopy

```
1. <span style="font-size:14px;">import java.io.IOException;
2. import java.net.ServerSocket;
3. import java.net.Socket;
4. import java.util.concurrent.Executor;
5. import java.util.concurrent.Executors;
6. import java.util.logging.Logger;
7.
8. public class TCPEchoServerExecutor {
9.
10.     public static void main(String[] args) throws IOException {
11.         if (args.length != 1) { // Test for correct # of args
12.             throw new IllegalArgumentException("Parameter(s): <Port>");
13.         }
14.         int echoServPort = Integer.parseInt(args[0]); // Server port
15.         // Create a server socket to accept client connection requests
16.         ServerSocket servSock = new ServerSocket(echoServPort);
17.         Logger logger = Logger.getLogger("practical");
18.         <span style="color: rgb(54, 46, 43); line-
    height: 25.99431800842285px;"><span style="font-
    family:SimSun;">ExecutorService</span></span><span style="color: rgb(54, 46,
    43); font-family: Arial; line-
    height: 25.99431800842285px;"> </span>service= Executors.newCachedThreadPool
    (); // Dispatch svc
19.         // Run forever, accepting and spawning a thread for each connection

20.         while (true) {
21.             Socket clntSock = servSock.accept(); // Block waiting for connec
    tion
22.             service.execute(new EchoProtocol(clntSock, logger));
23.         }
24.         /* NOT REACHED */
25.     }
26. }</span>
```

Executors 类的 newCachedThreadPool()静态工厂方法创建了一个 ExecutorService 实例。在使用一个实现了 Runnable 接口的实例调用它的 execute()方法时，如果必要它将创建一个新的线程来处理任务。然而，它首先会尝试使用已有的线程。如果一个线程空闲了 60 秒以上，则将移出线程池。这个策略几乎总是比前面两个 TCPEchoServer*例子的效率高。service 的 execute()方法，该方法要么将其分配给一个已有的线程，要么创建一个新的线程来处理它。值得注意的是，当达到稳定状态时，缓存线程池服务最终将保持合适的线程数，以使每个线程都保持忙碌，同时又很少创建或销毁线程。

阻塞和超时

Socket 的 I/O 调用可能会因为多种原因而阻塞。数据输入方法 read()和 receive()在没有数据可读时会阻塞。TCP 套接字的 write()方法在没有足够的空间缓存传输的数据时可能阻塞。 ServerSocket 的 accept()方法和 Socket 的构造函数都会阻塞等待，直到连接建立。同时，长的信息往返时间，高错误率的连接和慢速的（或已发生故障的）服务器，都可能导致需要很长的时间来建立连接。NIO 包中的更加强大的非阻塞工具。

对于 read()、accept()和 receive()的阻塞，使用 Socket 类、ServerSocket 类和 DatagramSocket 类的 setSoTimeout()方法，设置其阻塞的最长时间（以毫秒为单位）。如果在指定时间内这些方法没有返回，则将抛出一个 InterruptedIOException 异常。对于 Socket 实例，在调用 read()方法前，我们还可以使用该套接字的 InputStream 的 available()方法来检测是否有可读的数据。

对于 Socket 连接服务器的阻塞（Socket 类的构造函数会尝试根据参数中指定的主机和端口来建立连接，并阻塞等待，直到连接成功建立或发生了系统定义的超时）不幸的是，系统定义的超时时间很长，而 Java 又没有提供任何缩短它的方法。要改变这种情况，可以使用 Socket 类的无参数构造函数，它返回的是一个没有建立连接的 Socket 实例。需要建立连接时，调用该实例的 connect()方法，并指定一个远程终端和超时时间（毫秒）。

对于 write 方法阻塞（write()方法调用也会阻塞等待，直到最后一个字节成功写入到了 TCP 实现的本地缓存中）如果可用的缓存空间比要写入的数据小，在 write()方法调用返回前，必须把一些数据成功传输到连接的另一端。因此，write()方法的阻塞总时间最终还是取决于接收端的应用程序。不幸的是 Java 现在还没有提供任何使 write()超时或由其他线程将其打断的方法。所以如果一个可以在 Socket 实例上发送大量数据的协议可能会无限期地阻塞下去。

限制每个客户端的时间，实现一个为每个客户端限定了服务时间的回显协议。也就是说我们定义一个了目标，TIMELIMIT，并在协议中实现经过 TIMELIMIT 毫秒后，实例就自动终止。协议实例保持了对剩余服务时间的跟踪，并使用 setSoTimeout()方法来保证 read()方法的阻塞时间不会超过 TIMELIMIT。由于没有办法限制 write()调用的时间，我们并不能保证所定义的时间限制真正有效。尽管如此，TimelimitEchoProtocol.java 还是实现了这种方法；要与 TCPEchoServerExecutor.java 一起使用，只需要简单地将 while 循环的第二行改为：service.execute(new TimeLimitEchoProtocol(clntSock, logger));

[java] view plaincopy

```
1.  <span style="font-size:14px;">import java.io.IOException;
```

```
2.  import java.io.InputStream;
3.  import java.io.OutputStream;
4.  import java.net.Socket;
5.  import java.util.logging.Level;
6.  import java.util.logging.Logger;
7.
8.  class TimelimitEchoProtocol implements Runnable {
9.      private static final int BUFSIZE = 32; // Size (bytes) of buffer
10.     private static final String TIMELIMIT = "10000"; // Default limit (ms)
11.     private static final String TIMELIMITPROP = "Timelimit"; // Property
12.
13.     private static int timelimit;
14.     private Socket clntSock;
15.     private Logger logger;
16.
17.     public TimelimitEchoProtocol(Socket clntSock, Logger logger) {
18.         this.clntSock = clntSock;
19.         this.logger = logger;
20.         // Get the time limit from the System properties or take the default

21.         timelimit = Integer.parseInt(System.getProperty(TIMELIMITPROP,TIMELI
    MIT));
22.     }
23.
24.     public static void handleEchoClient(Socket clntSock, Logger logger) {
25.         try {
26.             // Get the input and output I/O streams from socket
27.             InputStream in = clntSock.getInputStream();
28.             OutputStream out = clntSock.getOutputStream();
29.             int recvMsgSize; // Size of received message
30.             int totalBytesEchoed = 0; // Bytes received from client
31.             byte[] echoBuffer = new byte[BUFSIZE]; // Receive buffer
32.             long endTime = System.currentTimeMillis() + timelimit;
33.             int timeBoundMillis = timelimit;
34.             // Receive until client closes connection, indicated by -1
35.             while ((timeBoundMillis > 0) && // catch zero values
36.                     ((recvMsgSize = in.read(echoBuffer)) != -1)) {
37.                 out.write(echoBuffer, 0, recvMsgSize);
38.                 totalBytesEchoed += recvMsgSize;
39.                 timeBoundMillis = (int) (endTime - System.currentTimeMillis(
    ));
40.                 clntSock.setSoTimeout(timeBoundMillis);
41.             }
```

```
            logger.info("Client " + clntSock.getRemoteSocketAddress()+ ", ec
    hoed " + totalBytesEchoed + " bytes.");
        } catch (IOException ex) {
            logger.log(Level.WARNING, "Exception in echo protocol", ex);
        }
    }

    public void run() {
        handleEchoClient(this.clntSock, this.logger);
    }
}</span>
```

TimelimitEchoProtocol 类与 EchoProtocol 类非常相似，唯一的区别在于它试图将回显连接的总服务时间限制在 10 秒钟之内。当 handleEchoClient() 方法被调用时，就通过当前时间和服务期限计算出了服务的截止时间。每次 read()调用结束后将重新计算当前时间与截止时间的差值，即剩余服务时间，并将套接字超时设置为该剩余时间。

多接收者

一个服务器和一个客户端。这种一对一的通信方法有时称为单播（unicast），有两种类型的一对多（one-to-many）服务：广播（broadcast）和多播（multicast）。对于广播，（本地）网络中的所有主机都会接收到一份数据副本。对于多播，消息只是发送给一个多播地址（multicast address），网络只是将数据分发给那些表示想要接收发送到该多播地址的数据的主机。总的来说，只有 UDP 套接字允许广播或多播。

广播

广播 UDP 数据报文与单播数据报文相似，唯一的区别是其使用的是一个广播地址而不是一个常规的（单播）IP 地址。注意，IPv6 并没有明确地提供广播地址；然而，有一个特殊的全节点（all - nodes）、本地连接范围（link-local-scope）的多播地址，FFO2::1，发送给该地址的消息将多播到一个连接上的所有节点。IPv4 的本地广播地址（255.255.255.255）将消息发送到在同一广播网络上的每个主机。本地广播信息决不会被路由器转发。在以太网上的一个主机可以向在同一以太网内的其他主机发送消息，但是该消息不会被路由器转发。IPv4 还指定了定向广播地址，允许向指定网络中的所有主机进行广播；然而，互联网上的大部分路由器都不转发定向广播消息。

并不存在可以向网络范围内所有主机发送消息的广播地址。至于为什么没有，请考虑向互联网上每台主机发送广播消息可能产生的影响。在这种地址发送单个数据报文就可能会由路由器产生非常大量的数据包副本，并可能会耗尽所有网络的带宽。误用（恶意的或意外的）该地址的后果会非常严重，因此 IP 协议的设计者故意没有定义互联网范围的广播机

制。在 Java 中，单播和广播的代码是相同的。要实现具有广播功能的应用程序，我们可以简单地在 VoteClientUDP.java 中使用广播目的地址。

多播

与广播一样，多播与单播之间的一个主要区别是地址的形式。一个多播地址指示了一组接收者。IP 协议的设计者为多播分配了一定范围的地址空间，IPv4 中的多播地址范围是 224.0.0.0 到 239.255.255.255，IPv6 中的多播地址是任何由 FF 开头的地址。除了少数系统保留的多播地址外，发送者可以向以上范围内的任何地址发送数据。Java 中多播应用程序主要通过 MulticastSocke 实例进行通信，它是 DatagramSocket 的一个子类。重点需要理解的是，一个 MulticastSocket 实例实际上就是一个 UDP 套接字（DatagramSocket），其包含了一些额外的可以控制的多播特定属性。

多播发送者：

[java] view plaincopy

```
1.  <span style="font-size:14px;">import java.io.IOException;
2.  import java.net.DatagramPacket;
3.  import java.net.InetAddress;
4.  import java.net.MulticastSocket;
5.
6.  public class VoteMulticastSender {
7.
8.      public static final int CANDIDATEID = 475;
9.
10.     public static void main(String args[]) throws IOException {
11.
12.         if ((args.length < 2) || (args.length > 3)) { // Test # of args
13.             throw new IllegalArgumentException("Parameter(s): <Multicast Add
    r> <Port> [<TTL>]");
14.         }
15.
16.         InetAddress destAddr = InetAddress.getByName(args[0]); // Destinatio
    n
17.         if (!destAddr.isMulticastAddress()) { // Test if multicast address
18.             throw new IllegalArgumentException("Not a multicast address");
19.         }
20.
21.         int destPort = Integer.parseInt(args[1]); // Destination port
22.         int TTL = (args.length == 3) ? Integer.parseInt(args[2]) : 1; // Set
    TTL
23.         MulticastSocket sock = new MulticastSocket();
```

```
24.         sock.setTimeToLive(TTL); // Set TTL for all datagrams
25.         VoteMsgCoder coder = new VoteMsgTextCoder();
26.         VoteMsg vote = new VoteMsg(true, true, CANDIDATEID, 1000001L);
27.         // Create and send a datagram
28.         byte[] msg = coder.toWire(vote);
29.         DatagramPacket message = new DatagramPacket(msg, msg.length, destAdd
    r,destPort);
30.         System.out.println("Sending Text-
    Encoded Request (" + msg.length+ " bytes): ");
31.         System.out.println(vote);
32.         sock.send(message);
33.         sock.close();
34.     }
35. }</span>
```

与广播不同，网络多播只将消息副本发送给指定的一组接收者。这组接收者叫做多播组（multicast group），通过共享的多播（组）地址确定。接收者需要一种机制来通知网络它对发送到某一特定地址的消息感兴趣，以使网络将数据包转发给它。这种通知机制叫做加入一组（joining a group），可以由 MulticastSocket 类的 joinGroup()方法实现。我们的多播接收者加入了一个特定的组，接收并打印该组的一条多播消息，然后退出。

多播接收者：

[java] view plaincopy

```
1.  <span style="font-size:14px;">import java.io.IOException;
2.  import java.net.DatagramPacket;
3.  import java.net.InetAddress;
4.  import java.net.MulticastSocket;
5.  import java.util.Arrays;
6.
7.  public class VoteMulticastReceiver {
8.
9.      public static void main(String[] args) throws IOException {
10.
11.         if (args.length != 2) { // Test for correct # of args
12.             throw new IllegalArgumentException("Parameter(s): <Multicast Add
    r> <Port>");
13.         }
14.
15.         InetAddress address = InetAddress.getByName(args[0]); // Multicast a
    ddress
```

```
16.         if (!address.isMulticastAddress()) { // Test if multicast address
17.             throw new IllegalArgumentException("Not a multicast address");
18.         }
19.         int port = Integer.parseInt(args[1]); // Multicast port
20.         MulticastSocket sock = new MulticastSocket(port); // for receiving
21.         sock.joinGroup(address); // Join the multicast group
22.         VoteMsgTextCoder coder = new VoteMsgTextCoder();
23.         // Receive a datagram
24.         DatagramPacket packet = new DatagramPacket(new byte[VoteMsgTextCoder
    .MAX_WIRE_LENGTH],VoteMsgTextCoder.MAX_WIRE_LENGTH);
25.         sock.receive(packet);
26.         VoteMsg vote = coder.fromWire(Arrays.copyOfRange(packet.getData(), 0
    ,packet.getLength()));
27.         System.out.println("Received Text-
    Encoded Request ("+ packet.getLength() + " bytes): ");
28.         System.out.println(vote);
29.         sock.close();
30.     }
31. }</span>
```

多播和单播接收者唯一的重要区别是，多播接收者表明希望从哪个多播地址接收数据来加入多播组。不过 MulticastSocket 还有一些 DatagramSocket 没有的能力，包括 1）允许指定数据报文的 TTL，和 2）允许指定和改变通过哪个接口将数据报文发送到组（接口由其互联网地址确定）。

决定使用广播还是使用多播需要考虑多方面的因素，包括接收者的网络地址和通信双方的知识。互联网广播的范围是限定在一个本地广播网络之内的，并对广播接收者的位置进行了严格的限制。多播通信可能包含网络中任何位置的接收者，[]因此多播有个好处就是它能够覆盖一组分布在各处的接收者。IP 多播的不足在于接收者必须知道要加入的多播组的地址。而接收广播信息则不需要指定地址信息。在某些情况下，广播是一个比多播更好更易于发现的机制。所有主机在默认情况下都可以接收广播

Keep-Alive 机制

如果一段时间内没有数据交换，通信的每个终端可能都会怀疑对方是否还处于活跃状态。TCP 协议提供了一种 keep-alive 的机制，该机制在经过一段不活动时间后，将向另一个终端发送一个探测消息。如果另一个终端还出于活跃状态，它将回复一个确认消息。如果经过几次尝试后依然没有收到另一终端的确认消息，则终止发送探测信息，关闭套接字，并在下一次尝试 I/O 操作时抛出一个异常。注意，应用程序只要在探测信息失败时才能察觉

到 keep-alive 机制的工作。

boolean getKeepAlive()

void setKeepAlive(boolean on)

默认情况下，keep-alive 机制是关闭的。通过调用 setKeepAlive()方法将其设置为 true 来开启 keep-alive 机制。

发送和接收缓存区的大小

一旦创建了一个 Socket 或 DatagramSocket 实例，操作系统就必须为其分配缓存区以存放接收的和要发送的数据。Socket, DatagramSocket: 设置和获取发送接收缓存区大小（以字节为单位）

int getReceiveBufferSize()

void setReceiveBufferSize(int size)

int getSendBufferSize()

void setSendBufferSize(int size)

还可以在 ServerSocket 上指定接收缓冲区大小。不过，这实际上是为 accept()方法所创建的新 Socket 实例设置接收缓冲区大小。为什么可以只设置接收缓冲区大小而不设置发送缓冲区的大小呢？当接收了一个新的 Socket，它就可以立刻开始接收数据，因此需要在 accept()方法完成连接之前设置好缓冲区的大小。另一方面，由于可以控制什么时候在新接受的套接字上发送数据，因此在发送之前还有时间设置发送缓冲区的大小。

ServerSocket: 设置/获取所接受套接字的接收缓冲区大小

int getReceiveBufferSize()

void setReceiveBufferSize(int size)

消除缓冲延迟

TCP 协议将数据缓存起来直到足够多时一次发送，以避免发送过小的数据包而浪费网络资源。虽然这个功能有利于网络，但应用程序可能对所造成的缓冲延迟不能容忍。好在可以人为禁用缓存功能。

Socket: 设置/获取 TCP 缓冲延迟

boolean getTcpNoDelay()

void setTcpNoDelay(boolean on)
getTcpNoDelay()和 setTcpNoDelay()方法用于获取和设置是否消除缓冲延迟。将值设置为 true 表示禁用缓冲延迟功能。

关闭连接

调用 Socket 的 close()方法将同时终止两个方向（输入和输出）的数据流。一旦一个终端（客户端或服务器端）关闭了套接字，它将无法再发送或接收数据。这就意味着 close()方法只能在调用者完成通信之后用来给另一端发送信号。

Socket 类的 shutdownInput()和 shutdownOutput()方法能够将输入输出流相互独立地关闭。调用 shutdownInput()后，套接字的输入流将无法使用。任何没有发送的数据都将毫无提示地被丢弃，任何想从套接字的输入流读取数据的操作都将返回-1。当 Socket 调用 shutdownOutput() 方法后，套接字的输出流将无法再发送数据，任何尝试向输出流写数据的操作都将抛出一个 IOException 异常。在调用 shutdownOutput()之前写出的数据可能能够被远程套接字读取，之后，在远程套接字输入流上的读操作将返回-1。应用程序调用 shutdownOutput()后还能继续从套接字读取数据，类似的，在调用 shutdownInput()后也能够继续写数据。

4：NIO

NIO 利用操作 Buffer 的信道 Channel（轮询的目标）的非阻塞特性和轮询 I/O 状态的选择器 Selector（一次轮询一组客户端）就可以在单线程下为任意数量的连接提供服务。

NIO 主要包括两个部分：java.nio.channels 包介绍了 Selector 和 Channel 抽象，java.nio 包介绍了 Buffer 抽象

为什么需要 NIO？

(1):：由于创建、维护和切换线程需要的系统开销，一客户一线程方式在系统扩展性方面受到了限制。使用线程池可以节省那种系统开销，同时允许实现者利用并行硬件的优势。但对于连接生存期比较长的协议来说，线程池的大小仍然限制了系统可以同时处理的客户端数量。

(2):在使用线程的扩展性方面还涉及一些更加难以把握的挑战。其中一个挑战就是程序员几乎不能对什么时候哪个线程将获得服务进行控制。你可以设置一个线程实例的优先级（priority）（高优先级的线程相对于低优先级的线程有优先权），但是这个优先级只是一种"建议"--下一个选择执行的线程完全取决于具体实现

(3):所有客户之间共享一些状态信息（即调度表）需要通过使用锁（locks）机制或其他互斥机制对依次访问状态进行严格的同步（synchronized），对共享状态进行同步访问，要同时考虑到多线程服务器的正确性和高效性就变得非常困难，同时使用同步机制将增加更多的系统调度和上下文切换开销，而程序员对这些开销又无法控制。

一个 Channel 实例代表了一个"可轮询的（pollable）"I/O 目标，如套接字（或一个文件、设备等）。Channel 能够注册一个 Selector 类的实例。Selector 的 select()方法允许你询问"在一组信道中，哪一个当前需要服务（即，被接受，读或写）？"。

Buffer 抽象代表了一个有限容量（finite-capacity）的数据容器--其本质是一个数组，由指针指示了在哪存放数据和从哪读取数据。Buffer 则提供了比 Stream 抽象更高效和可预测的 I/O。 Stream 抽象好的方面是隐藏了底层缓冲区的有限性，提供了一个能够容纳任意长度数据的容器的假象。坏的方面是要实现这样一个假象，要么会产生大量的内存开销，要么会引入大量的上下文切换，甚至可能两者都有。在使用线程时，这些开销都隐藏在了具体实现中，因此也失去了对其的可控性和可预测性。这种方法使编写程序变得容易，但要调整它们的性能则变得更困难。不幸的是，如果要使用 Java 的 Socket 抽象，流就是唯一的选择

使用 Buffer 有两个主要好处。第一，与读写缓冲区数据相关联的系统开销暴露给了程序员。例如，如果想要向缓冲区存入数据，但又没有足够的空间时，就必须采取一些措施来获得空间（即，移出一些数据，或移开已经在那个位置的数据来获得空间，或者创建一个新的实例）。这意味着需要额外的工作，但是你（程序员）可以控制它什么时候发生，如何发生，以及是否发生。一个聪明的程序员如果清楚地了解了应用程序的需求，就那能通过权衡这些选择来降低系统开销。第二，一些对 Java 对象的特殊 Buffer 映射操作能够直接操作底层平台的资源（例如，操作系统的缓冲区）。这些操作节省了在不同地址空间中复制数据的开销--这在现代计算机体系结构中是开销很大的操作。

Channel 实例代表了一个与设备的连接，通过它可以进行输入输出操作。实际上 Channel 的基本思想与我们见过的普通套接字非常相似。对于 TCP 协议，可以使用 ServerSocketChannel 和 SocketChannel。还有一些针对其他设备的其他类型信道（如，FileChannel），信道（channel）和套接字（socket）之间的不同点之一，可能是信道通常要调用静态工厂方法来获取实例：

```
SocketChannel clntChan = SocketChannel.open();

 ServerSocketChannel servChan = ServerSocketChannel.open();
```

Channel 使用的不是流，而是缓冲区来发送或读取数据。Buffer 类或其任何子类的实例都可以看作是一个定长的 Java 基本数据类型元素序列。与流不同，缓冲区有固定的、有限

的容量，并由内部（但可以被访问）状态记录了有多少数据放入或取出，就像是有限容量的队列一样。Buffer 是一个抽象类，只能通过创建它的子类来获得 Buffer 实例，而每个子类都设计为用来容纳一种 Java 基本数据类型（boolean 除外）。因此，这些实例分别为 FloatBuffer，或 lntBuffer，或 ByteBuffer，等等（ByteBuffer 是这些实例中最灵活的）。在 channel 中使用 Buffer 实例通常不是使用构造函数创建的，而是通过调用 allocate()方法创建指定容量的 Buffer 实例，

ByteBuffer buffer = ByteBuffer.allocate(CAPACITY);

或通过包装一个已有的数组来创建：

ByteBuffer buffer = ByteBuffer.wrap(byteArray);

套接字的某些操作可能会无限期地阻塞，如创建/接收连接或读写数据等 I/O 调用，都可能无限期地阻塞等待，直到底层的网络实现发生了什么。慢速的、有损耗的网络，或仅仅是简单的网络故障都可能导致任意时间的延迟。然而不幸的是，在调用一个方法之前无法知道其是否会阻塞。而 NIO 的强大功能部分来自于 channel 的非阻塞特性。NIO 的 channel 抽象的一个重要特征就是可以通过配置它的阻塞行为，以实现非阻塞式的信道。

clntChan.configureBlocking(false);

在非阻塞式信道上调用一个方法总是会立即返回。这种调用的返回值指示了所请求的操作完成的程度。例如，在一个非阻塞式 ServerSocketChannel 上调用 accept()方法，如果有连接请求在等待，则返回客户端 SocketChannel，否则返回 null。

[java] view plaincopy

```
1.  <span style="font-size:14px;">import java.net.InetSocketAddress;
2.  import java.net.SocketException;
3.  import java.nio.ByteBuffer;
4.  import java.nio.channels.SocketChannel;
5.
6.  public class TCPEchoClientNonblocking {
7.
8.      public static void main(String args[]) throws Exception {
9.          if ((args.length < 2) || (args.length > 3)) // Test for correct # of
    args
10.             throw new IllegalArgumentException("Parameter(s): <Server> <Word
    > [<Port>]");
11.         String server = args[0]; // Server name or IP address
12.         // Convert input String to bytes using the default charset
```

```
13.         byte[] argument = args[1].getBytes();
14.         int servPort = (args.length == 3) ? Integer.parseInt(args[2]) : 7;
15.         // Create channel and set to nonblocking
16.         SocketChannel clntChan = SocketChannel.open();
17.         clntChan.configureBlocking(false);
18.         // Initiate connection to server and repeatedly poll until complete

19.         if (!clntChan.connect(new InetSocketAddress(server, servPort))) {
20.             while (!clntChan.finishConnect()) {
21.                 System.out.print("."); // Do something else
22.             }
23.         }
24.         ByteBuffer writeBuf = ByteBuffer.wrap(argument);
25.         ByteBuffer readBuf = ByteBuffer.allocate(argument.length);
26.         int totalBytesRcvd = 0; // Total bytes received so far
27.         int bytesRcvd; // Bytes received in last read
28.         while (totalBytesRcvd < argument.length) {
29.             if (writeBuf.hasRemaining()) {
30.                 clntChan.write(writeBuf);
31.             }
32.             if ((bytesRcvd = clntChan.read(readBuf)) == -1) {
33.                 throw new SocketException("Connection closed prematurely");

34.             }
35.             totalBytesRcvd += bytesRcvd;
36.             System.out.print("."); // Do something else
37.         }
38.         System.out.println("Received: " +new String(readBuf.array(), 0, tota
    lBytesRcvd)); // convert to String per default charset
39.         clntChan.close();
40.     }
41. }
42. </span>
```

由于该套接字是非阻塞式的，因此对 connect()方法的调用可能会在连接建立之前返回，如果在返回前已经成功建立了连接，则返回 true，否则返回 false。对于后一种情况，任何试图发送或接收数据的操作都将抛出 NotYetConnectedException 异常，因此，我们通过持续调用 finishConnect()方法来"轮询"连接状态，该方法在连接成功建立之前一直返回 false。打印操作演示了在等待连接建立的过程中，程序还可以执行其他任务。不过，这种忙等的方法非常浪费系统资源，这里这样做只是为了演示该方法的使用。

[java] view plaincopy

```
1.  <span style="font-size:14px;">import java.io.IOException;
```

```
2.  import java.net.InetSocketAddress;
3.  import java.nio.channels.SelectionKey;
4.  import java.nio.channels.Selector;
5.  import java.nio.channels.ServerSocketChannel;
6.  import java.util.Iterator;
7.
8.  public class TCPServerSelector {
9.
10.     private static final int BUFSIZE = 256; // Buffer size (bytes)
11.     private static final int TIMEOUT = 3000; // Wait timeout (milliseconds)
12.
13.     public static void main(String[] args) throws IOException {
14.         if (args.length < 1) { // Test for correct # of args
15.             throw new IllegalArgumentException("Parameter(s): <Port> ...");
16.         }
17.         // Create a selector to multiplex listening sockets and connections
18.         Selector selector = Selector.open();
19.         // Create listening socket channel for each port and register selector
20.         for (String arg : args) {
21.             ServerSocketChannel listnChannel = ServerSocketChannel.open();
22.             listnChannel.socket().bind(new InetSocketAddress(Integer.parseInt(arg)));
23.             listnChannel.configureBlocking(false); // must be nonblocking to register
24.             // Register selector with channel. The returned key is ignored
25.             listnChannel.register(selector, SelectionKey.OP_ACCEPT);
26.         }
27.         // Create a handler that will implement the protocol
28.         TCPProtocol protocol = new EchoSelectorProtocol(BUFSIZE);
29.         while (true) { // Run forever, processing available I/O operations
30.         // Wait for some channel to be ready (or timeout)
31.             if (selector.select(TIMEOUT) == 0) { // returns # of ready chans
32.                 System.out.print(".");
33.                 continue;
34.             }
35.             // Get iterator on set of keys with I/O to process
36.             Iterator<SelectionKey> keyIter = selector.selectedKeys().iterator();
37.             while (keyIter.hasNext()) {
```

```
                SelectionKey key = keyIter.next(); // Key is bit mask
                // Server socket channel has pending connection requests?
                if (key.isAcceptable()) {
                    protocol.handleAccept(key);
                }
                // Client socket channel has pending data?
                if (key.isReadable()) {
                    protocol.handleRead(key);
                }
                // Client socket channel is available for writing and
                // key is valid (i.e., channel not closed)?
                if (key.isValid() && key.isWritable()) {
                    protocol.handleWrite(key);
                }
                keyIter.remove(); // remove from set of selected keys
            }
        }
    }
}</span>
```

只有非阻塞信道才可以注册选择器，因此需要将其配置为适当的状态。由于 select()操作只是向 Selector 所关联的键集合中添加元素，因此，如果不移除每个处理过的键，它就会在下次调用 select()方法是仍然保留在集合中，而且可能会有无用的操作来调用它。TCPServerSelector 的大部分内容都与协议无关，只有协议赋值那一行代码是针对的特定协议。所有协议细节都包含在了 TCPProtocol 接口的具体实现中。EchoSelectorProtocol 类就实现了该回显协议的操作器。

[java] view plaincopy

```
<span style="font-size:14px;">import java.nio.channels.SelectionKey;
import java.nio.channels.SocketChannel;
import java.nio.channels.ServerSocketChannel;
import java.nio.ByteBuffer;
import java.io.IOException;

public class EchoSelectorProtocol implements TCPProtocol {

    private int bufSize; // Size of I/O buffer

    public EchoSelectorProtocol(int bufSize) {
        this.bufSize = bufSize;
    }

    public void handleAccept(SelectionKey key) throws IOException {
```

```
16.         SocketChannel clntChan = ((ServerSocketChannel) key.channel()).accep
    t();
17.         clntChan.configureBlocking(false); // Must be nonblocking to registe
    r
18.         // Register the selector with new channel for read and attach byte
19.         // buffer
20.         clntChan.register(key.selector(), SelectionKey.OP_READ, ByteBuffer.a
    llocate(bufSize));
21.     }
22.
23.     public void handleRead(SelectionKey key) throws IOException {
24.         // Client socket channel has pending data
25.         SocketChannel clntChan = (SocketChannel) key.channel();
26.         ByteBuffer buf = (ByteBuffer) key.attachment();
27.         long bytesRead = clntChan.read(buf);
28.         if (bytesRead == -1) { // Did the other end close?
29.             clntChan.close();
30.         } else if (bytesRead > 0) {
31.             // Indicate via key that reading/writing are both of interest no
    w.
32.             key.interestOps(SelectionKey.OP_READ | SelectionKey.OP_WRITE);
33.         }
34.     }
35.
36.     public void handleWrite(SelectionKey key) throws IOException {
37.         /*
38.          * Channel is available for writing, and key is valid (i.e., client
39.          * channel not closed).
40.          */
41.         // Retrieve data read earlier
42.         ByteBuffer buf = (ByteBuffer) key.attachment();
43.         buf.flip(); // Prepare buffer for writing
44.         SocketChannel clntChan = (SocketChannel) key.channel();
45.         clntChan.write(buf);
46.         if (!buf.hasRemaining()) { // Buffer completely written?
47.             // Nothing left, so no longer interested in writes
48.             key.interestOps(SelectionKey.OP_READ);
49.         }
50.         buf.compact(); // Make room for more data to be read in
51.     }
52. }</span>
```

[Buffer 详解]

NIO 中，数据的读写操作始终是与缓冲区相关联的。Channel 将数据读入缓冲区，然后我们又从缓冲区访问数据。写数据时，首先将要发送的数据按顺序填入缓冲区。基本上，缓冲区只是一个列表，它的所有元素都是基本数据类型（通常为字节型）。缓冲区是定长的，它不像一些类那样可以扩展容量（例如，List，StringBuffer 等）。注意，ByteBuffer 是最常用的缓冲区，因为：1）它提供了读写其他数据类型的方法，2）信道的读写方法只接收 ByteBuffer。

Buffer 索引：

缓冲区不仅仅是用来存放一组元素的列表。在读写数据时，它有内部状态来跟踪缓冲区的当前位置，以及有效可读数据的结束位置等，为了实现这些功能，每个缓冲区维护了指向其元素列表的 4 个索引

索引	描述	存取器/修改器/用法
capacity	缓冲区中的元素总数（不可修改）	int capacity()
position	下一个要读/写的元素（从 0 开始）	int position() Buffer position(int newPosition)
limit	第一个不可读/写元素	int limit() Buffer limit(int newLimit)
mark	用户选定的 *position* 的前一个位置，或 0	Buffer mark() Buffer reset()

position 和 limit 之间的距离指示了可读取/存入的字节数。Java 中提供了两个方便的方法来计算这个距离。

ByteBuffer: 剩余字节

boolean hasRemaining()

int remaining()
当缓冲区至少还有一个元素时，hasRemaining()方法返回 true，remaining()方法返回剩余元素的个数。

在这些变量中，以下关系保持不变：
0 ≤ mark ≤ position ≤ limit ≤ capacity
mark 变量的值"记录"了一个将来可返回的位置，reset()方法则将 position 的值还原成上次调用 mark()方法后的 position 值（除非这样做会违背上述的不变关系）。

创建 Buffer：

方法	Capacity	Position	Limit
ByteBuffer allocate(int capacity)	*capacity*	0	*capacity*
ByteBuffer allocateDirect(int capacity)	*capacity*	0	*capacity*
ByteBuffer wrap(byte[] array)	array.length	0	array.length
ByteBuffer wrap(byte[] array, int offset, int length)	array.length	offset	offset + length

使用分配空间的方式来创建缓冲区其实与使用包装的方法区别不大。惟一的区别是 allocate()方法创建了自己的后援数组。在缓冲区上调用 array()方法即可获得后援数组的引用。

通过包装的方法创建的缓冲区保留了被包装数组内保存的数据。实际上，wrap()方法只是简单地创建了一个具有指向被包装数组的引用的缓冲区，该数组称为后援数组。对后援数组中的数据做的任何修改都将改变缓冲区中的数据，反之亦然。如果我们为 wrap()方法指定了偏移量（offset）和长度（length），缓冲区将使用整个数组为后援数组，同时将 position 和 limit 的值初始化为偏移量（offset）和偏移量+长度（offset+length）。在偏移量之前和长度之后的元素依然可以通过缓冲区访问。

到目前为止，我们实现的所有缓冲区都将数据存放在 Java 分配的后援数组中。通常，底层平台（操作系统）不能使用这些缓冲区进行 I/O 操作。操作系统必须使用自己的缓冲区来进行 I/O，并将结果复制到缓冲区的后援数组中。这些复制过程可能非常耗费系统资源，尤其是在有很多读写需求的时候。Java 的 NIO 提供了一种直接缓冲区（direct buffers）来解决这个问题。使用直接缓冲区，Java 将从平台能够直接进行 I/O 操作的存储空间中为缓冲区分配后援存储空间，从而省略了数据的复制过程。这种低层的、本地的 I/O 通常在字节层进行操作，因此只能为 ByteBuffer 进行直接缓冲区分配。

ByteBuffer byteBufDirect = ByteBuffer.allocateDirect(BUFFERSIZE);

通过调用 isDirect()方法可以查看一个缓冲区是否是直接缓冲区。由于直接缓冲区没有后援数组，在它上面调用 array()或 arrayOffset()方法都将抛出 UnsupportedOperationException 异常。在考虑是否使用直接缓冲区时需要牢记几点。首先，要知道调用 allocateDirect()方法并不能保证能成功分配直接缓冲区--有的平台或 JVM 可能不支持这个操作，因此在尝试分配直接缓冲区后必须调用 isDirect()方法进行检查。其次，要知道分配和销毁直接缓冲区通常比分配和销毁非直接缓冲区要消耗更多的系统资源，因为直接缓冲区的后援存储空间通常存在与 JVM 之外，对它的管理需要与操作系统进行交互。所以，只有当需要在很多 I/O 操作上长时间使用时，才分配直接缓冲区。实际上，在相对于非直接缓冲区能明显提高系统性能时，使用直接缓冲区是个不错的主意。

存储和接收数据

只要有了缓冲区，就可以用它来存放数据了。作为数据的"容器"，缓冲区既可用来输入也可用来输出。这一点就与流不同，流只能向一个方向传递数据。使用 put()方法可以将数据放入缓冲区，使用 get()方法则可以从缓冲区获取数据。信道的 read()方法隐式调用了给定缓冲区的 put()，而其 write()方法则隐式调用了缓冲区的 get()方法。下面展示了 ByteBuffer 的 get()和 put()方法，当然，其他类型的缓冲区也有类似的方法。

ByteBuffer: 获取和存放字节
相对位置：
byte get()

ByteBuffer get(byte[] dst)

ByteBuffer get(byte[] dst, int offset, int length)

ByteBuffer put(byte b) ByteBuffer put(byte[] src)

ByteBuffer put(byte[] src, int offset, int length)

ByteBuffer put(ByteBuffer src)
绝对位置：
byte get(int index)

ByteBuffer put(int index, byte b)

每次调用 put()方法，都是在缓冲区中的已有元素后面追加数据，每次调用 get()方法，都是读取缓冲区的后续元素。不过，如果这些操作会导致 position 的值超出 limit 的限制，get()方法将抛出 BufferUnderflowException 异常，put()方法将抛出 BufferOverflowException 异常。基于绝对位置的 get()和 put()以指定的索引位置为参数，从该位置读取数据或向该位置写入数据。绝对位置形式的 get 和 put 不会改变 position 的值

ByteBuffer: 读取和存放 Java 多字节基本数据
<type> get<Type>()

<type> get<Type>(int index)

ByteBuffer put<Type>(<type> value)

ByteBuffer put<Type>(int index,<type> value)
其中"<Type>"代表 Char，Double，Int，Long，Short 之一，而"<type>"代表 char，double，int，long，short 之一。

很多 get/put 方法都返回一个 ByteBuffer。实际上它们返回的就是调用它们的那个 ByteBuffer。这样做可以实现链式调用（call chaining），即第一次调用的结果可以直接用来进行后续的方法调用。

Java 默认使用 big-endian。通过使用内置的 ByteOrder.BIG_ENDIAN 和 ByteOrder.LITTLE_ENDIAN 实例，可以获取和设定多字节数据类型写入字节缓冲区时的字节顺序。
ByteBuffer: 缓冲区中的字节顺序
ByteOrder order()

ByteBuffer order(ByteOrder order)

第一个方法以 ByteOrder 常量的形式返回缓冲区的当前字节顺序。第二个方法用来设置写多字节数据时的字节顺序。

准备 Buffer：clear()，flip()，和 rewind()

ByteBuffer 方法	准备 Buffer 以实现	结果值		
		Position	Limit	Mark
ByteBuffer clear()	将数据 read()/put() 进缓冲区	0	*capacity*	未定义
ByteBuffer flip()	从缓冲区 write()/get()	0	*position*	未定义
ByteBuffer rewind()	从缓冲区 rewrite()/get()	0	unchanged	未定义

clear()不会改变缓冲区中的数据，而只是简单地重置了缓冲区的主要索引值。

压缩 Buffer 中的数据

compact()方法将 position 与 limit 之间的元素复制到缓冲区的开始位置，从而为后续的 put()/read()调用让出空间

Buffer 透视：duplicate()，slice()等

方法	Capacity	新缓冲区的初始值		
		Position	Limit	Mark
ByteBuffer duplicate()	*capacity*	*position*	*limit*	*mark*
ByteBuffer slice()	remaining()	0	remaining()	未定义
ByteBuffer asReadOnlyBuffer()	*capacity*	*position*	*limit*	*mark*
CharBuffer asCharBuffer()	remaining() /2	0	remaining() /2	未定义
DoubleBuffer asDoubleBuffer()	remaining() /8	0	remaining() /8	未定义
FloatBuffer asFloatBuffer()	remaining() /4	0	remaining() /4	未定义
IntBuffer asIntBuffer()	remaining() /4	0	remaining() /4	未定义
LongBuffer asLongBuffer()	remaining() /8	0	remaining() /8	未定义
ShortBuffer asShortBuffer()	remaining() /2	0	remaining() /2	未定义

NIO 提供了多种方法来创建一个与给定缓冲区共享内容的新缓冲区，这些方法对元素的处理过程各有不同。基本上，这种新缓冲区有自己独立的状态变量（position，limit，capacity 和 mark），但与原始缓冲区共享了同一个后援存储空间。任何对新缓冲区内容的修改都将反映到原始缓冲区上。可以将新缓冲区看作是从另一个角度对同一数据的透视。

字符编码

字符是由字节序列进行编码的，而且在字节序列与字符集合之间有各种映射（称为字符集）方式。NIO 缓冲区的另一个用途是在各种字符集之间进行转换。要使用这个功能，还需要了解 java.nio.charset 包中另外两个类：CharsetEncoder 和 CharsetDecoder 类。

要进行编码，需要使用一个 Charset 实例来创建一个编码器并调用 encode 方法：

```
Charset charSet = Charset.forName("US-ASCII");

CharsetEncoder encoder = charSet.newEncoder();

ByteBuffer buffer = encoder.encode(CharBuffer.wrap("Hi mom"));
```

要进行解码，需要使用 Charset 实例来创建一个解码器，并调用 decode 方法：

```
CharsetDecoder decoder = charSet.newDecoder();

CharBuffer cBuf = decoder.decode(buffer);
```

虽然这种方法能够正常工作，但当需要进行多次编码时，效率就会变得较低。例如，每次调用 encode/decode 方法都会创建一个新 Byte/CharBuffer 实例。其他导致低效率的地方与编码器的创建和操作有关。

流（TCP）信道详解

流信道有两个变体：SocketChannel 和 ServerSocketChannel。像其对应的 Socket 一样，SocketChannel 是相互连接的终端进行通信的信道。

SocketChannel: 创建，连接和关闭

```
static SocketChannel open(SocketAddress remote)

static SocketChannel open()

 boolean connect(SocketAddress remote)

boolean isConnected()

void close()

boolean isOpen()

Socket socket()
```

SocketChannel: 读和写

```
int read(ByteBuffer dst)

long read(ByteBuffer[] dsts)

long read(ByteBuffer[] dsts, int offset, int length)
```

int write(ByteBuffer src)

long write(ByteBuffer[] srcs)

long write(ByteBuffer[] srcs, int offset, int length)

ServerSocketChannel: 创建，接受和关闭

static ServerSocketChannel open()

ServerSocket socket()

SocketChannel accept()

void close()

boolean isOpen()

SocketChannel, Server SocketChannel: 设置阻塞行为

SelectableChannel configureBlocking(boolean block)

boolean isBlocking()

SocketChannel: 测试连接性

boolean finishConnect()

boolean isConnected()

boolean isConnectionPending()

Selector 详解

Selector: 创建和关闭

static Selector open()

boolean isOpen() void close()

调用 Selector 的 open()工厂方法可以创建一个选择器实例。选择器的状态是"打开"或"关闭"的。创建时选择器的状态是打开的，并保持该状态

信道中注册

每个选择器都有一组与之关联的信道，选择器对这些信道上"感兴趣的"I/O 操作进行监听。Selector 与 Channel 之间的关联由一个 SelectionKey 实例表示。（注意，一个信道可以注册多个 Selector 实例，因此可以有多个关联的 SelectionKey 实例）SelectionKey 维护了一个信道上感兴趣的操作类型信息，并将这些信息存放在一个 int 型的位图（bitmap）中，该 int 型数据的每一位都有相应的含义。SelectionKey 类中的常量定义了信道上可能感兴趣的操作类型，每个这种常量都是只有一位设置为 1 的位掩码（bitmask）。

SelectionKey: 兴趣操作集

static int OP_ACCEPT

static int OP_CONNECT

static int OP_READ

static int OP_WRITE

int interestOps()

SelectionKey interestOps(int ops)

任何对 key（信道）所关联的兴趣操作集的改变，都只在下次调用了 select()方法后才会生效。

SocketChannel, ServerSocketChannel: 注册 Selector

SelectionKey register(Selector sel, int ops)

SelectionKey register(Selector sel, int ops, Object attachment)

int validOps() boolean isRegistered()

SelectionKey keyFor(Selector sel)

调用信道的 register()方法可以将一个选择器注册到该信道。在注册过程中，通过存储在 int 型数据中的位图来指定该信道上的初始兴趣操作集（见上文的"SelectionKey：兴趣操作集"）。register()方法将返回一个代表了信道和给定选择器之间的关联的 SelectionKey 实例。validOps()方法用于返回一个指示了该信道上的有效 I/O 操作集的位图。对于 ServerSocketChannel 来说，accept 是惟一的有效操作，而对于 SocketChannel 来说，有效操作包括读、写和连接。对于 DatagramChannel，只有读写操作是有效的。

以下代码注册了一个信道，支持读和写操作：

SelectionKey key = clientChannel.register(selector, SelectionKey.OP_READ | SelectionKey.OP_WRITE);

SelectionKey: 获取和取消

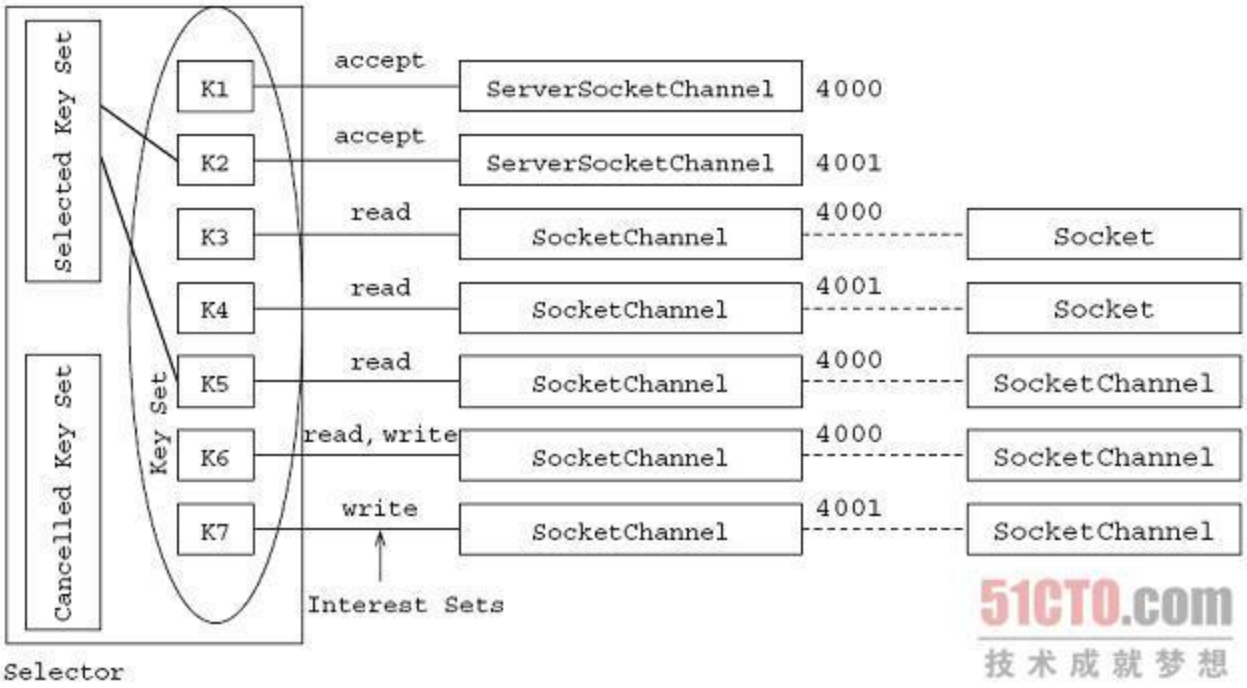
Selector selector()

SelectableChannel channel()

void cancel()

键关联的 Selector 实例和 Channel 实例可以分别使用该键的 selector()和 channel()方法获得。cancel()方法用于（永久性地）注销该键，并将其放入选择器的注销集（canceled set），在下一次调用 select()方法时，这些键将从该选择器的所有键集中移除，其关联的信道也将不再被监听（除非它又重新注册）。

下图展示了一个选择器，其键集中包含了 7 个代表注册信道的键：两个在端口 4000 和 4001 上的服务器信道，以及从服务器信道创建的 5 个客户端信道：



Selected Key Set: 选择键集； Cancelled Key Set：注销键集； Key Set：键集；Interest Sets：兴趣操作集

选取和识别准备就绪的信道

在信道上注册了选择器，并由关联的键指定了感兴趣的 I/O 操作集后，我们就只需要坐下来等待 I/O 了。这要使用选择器来完成。

Selector: 等待信道准备就绪

```
int select()

 int select(long timeout)
int selectNow()

 Selector wakeup()
```

select()方法用于从已经注册的信道中返回在感兴趣的 I/O 操作集上准备就绪的信道总数。（例如，兴趣操作集中包含 OP_READ 的信道有数据可读，或包含 OP_ACCEPT 的信道有连接请求待接受。）以上三个 select 方法的惟一区别在于它们的阻塞行为。无参数的 select 方法会阻塞等待，直到至少有一个注册信道中有感兴趣的操作准备就绪，或有别的线程调用了该选择器的 wakeup()方法（这种情况下 select 方法将返回 0）。以超时时长作为参数的 select 方法也会阻塞等待，直到至少有一个信道准备就绪，或等待时间超过了指定的毫秒数（正数），或者有另一个线程调用其 wakeup()方法。selectNow()方法是一个非阻塞版本：它总是立即返回，如果没有信道准备就绪，则返回 0。wakeup()方法可以使当前阻塞（也就是说在另一个线程中阻塞）的任何一种 select 方法立即返回；如果当前没有 select 方法阻塞，下一次调用这三种方法的任何一个都将立即返回。

Selector: 获取键集

```
Set<SelectionKey> keys()

Set<SelectionKey> selectedKeys()
```

以上方法返回选择器的不同键集。keys()方法返回当前已注册的所有键。返回的键集是不可修改的：任何对其进行直接修改的尝试（如，调用其 remove()方法）都将抛出 UnsupportedOperationException 异常。selectedKeys()方法用于返回上次调用 select()方法时，被"选中"的已准备好进行 I/O 操作的键。重要提示：selectedKeys()方法返回的键集是可修改的，实际上在两次调用 select()方法之间，都必须"手工"将其清空。换句话说，select 方法只会在已有的所选键集上添加键，它们不会创建新的键集。

选择之后，我们需要知道哪些信道准备好了特定的 I/O 操作。每个选择器都维护了一个已选键集（selected-key set），与这些键关联的信道都有即将发生的特定 I/O 操作。通过调用 selectedKeys()方法可以访问已选键集，该方法返回一组 SelectionKey。我们可以在这组键上进行迭代，分别处理等待在每个键关联的信道上的 I/O 操作。

```
Iterator<SelectionKey> keyIter = selector.selectedKeys().iterator();

while (keyIter.hasNext()) {

SelectionKey key = keyIter.next();

 // ...Handle I/O for key's channel...

keyIter.remove();

}
```

以上方法返回选择器的不同键集。keys()方法返回当前已注册的所有键。返回的键集是不可修改的：任何对其进行直接修改的尝试（如，调用其 remove()方法）都将抛出 UnsupportedOperationException 异常。selectedKeys()方法用于返回上次调用 select()方法时，被"选中"的已准备好进行 I/O 操作的键。重要提示：selectedKeys()方法返回的键集是可修改的，实际上在两次调用 select()方法之间，都必须"手工"将其清空。换句话说，select 方法只会在已有的所选键集上添加键，它们不会创建新的键集。

对于选中的每个信道，我们需要知道它们各自准备好的特定 I/O 操作。除了兴趣操作集外，每个键还维护了一个即将进行的 I/O 操作集，称为就绪操作集（ready set）。

SelectionKey: 查找就绪的 I/O 操作

```
int readyOps()

 boolean isAcceptable()

boolean isConnectable()

 boolean isReadable()

boolean isValid()
```

boolean isWritable()
对于给定的键，可以使用 readyOps()方法或其他指示方法来确定兴趣集中的哪些 I/O 操作可以执行。readyOps()方法以位图的形式返回所有准备就绪的操作集。其他方法用于分别检查各种操作是否可用。

例如，查看键关联的信道上是否有正在等待的读操作，可以使用以下代码：
(key.readyOps() & SelectionKey.OP_READ) != 0 或 key.isReadable()

选择器的已选键集中的键，以及每个键中准备就绪的操作，都是由 select()方法来确定的。随着时间的推进，这些信息可能会过时。其他线程可能会处理准备就绪的 I/O 操作。同时，键也不是永远存在的。当其关联的信道或选择器关闭时，键也将失效。通过调用其 cancel()方法可以显示地将键设置为无效。调用其 isValid()方法可以检测一个键的有效性。无效的键将添加到选择器的注销键集中，并在下次调用任一种形式的 select()方法或 close()方法时从键集中移除。（当然，从键集中移除键意味着与它关联的信道也不再受监听。）

信道附件
当一个信道准备好进行 I/O 操作时，通常还需要额外的信息来处理请求。例如，在前面的回显协议中，当客户端信道准备好写操作时，就需要有数据可写。当然，我们所需要的可写数据是由之前同一信道上的读操作收集的，但是在其可写之前，这些数据存放在什么地方呢？另一个例子是第 3 章中的成帧过程。如果一个消息一次传来了多个字节，我们需要保存已接收的部分消息，直到完整个消息接收完成。这两种情况都需要维护每个信道的状态信息。然而，我们非常幸运！SelectionKey 通过使用附件使保存每个信道的状态变得容易。
SelectionKey: 查找准备就绪的 I/O 操作
Object attach(Object ob)
Object attachment()
每个键可以有一个附件，数据类型只能是 Object 类。附件可以在信道第一次调用 register()方法时与之关联，或者后来再使用 attach()方法直接添加到键上。通过 SelectionKey 的 attachment()方法可以访问键的附件。

总的来说，使用 Selector 的步骤如下：
I.创建一个 Selector 实例。
II.将其注册到各种信道，指定每个信道上感兴趣的 I/O 操作。
III.重复执行：

1.调用一种 select 方法。

2.获取选取的键列表。

3.对于已选键集中的每个键，

a.获取信道，并从键中获取附件（如果合适的话）

b.确定准备就绪的操作并执行。如果是 accept 操作，将接受的信道设置为非阻塞模式，并将其与选择器注册。

c.如果需要，修改键的兴趣操作集

d.从已选键集中移除键

数据报（UDP）信道

Java 的 NIO 包通过 DatagramChannel 类实现了数据报（UDP）信道。与我们之前看到的其他形式的 SelectableChannel 一样，DatagramChannel 在 DatagramSocket 上添加了选择和非阻塞行为，以及基于缓冲区的 I/O 操作能力。

DatagramChannel: 创建，连接和关闭

static DatagramChannel open()

boolean isOpen()

DatagramSocket socket()

void close()

需要调用 DatagramChannel 的 open()工厂方法来创建一个 DatagramChannel 实例，该实例是未绑定的。DatagramChannel 只是对基本 DatagramSocket 的一个包装器（wrapper）。使用其 socket()方法可以直接访问内部的 DatagramSocket 实例。这就允许通过调用基本的 DatagramSocket 方法进行绑定、设置套接字选项等操作。用完 DatagramChannel 后，要调用它的 close()方法将其关闭。

只要创建了一个 DatagramChannel 实例，就可以非常直接地发送和接收数据。

DatagramChannel: 发送和接收

int send(ByteBuffer src, SocketAddress target)

SocketAddress receive(ByteBuffer dst)

send()方法用于创建一个包含了给定 ByteBuffer 中的数据的数据报文，并将其发送到目的地址指定的 SocketAddress 上。receive()方法用于将接收到的数据报文存入指定缓冲区并返回发送者的地址。重要提示：如果缓冲区的剩余空间小于数据报文中的数据大小，多余的数据将毫无提示地丢弃。

以下代码段用于创建一个 DatagramChannel 实例，并将 UTF-16 编码的字符串"Hello"发送

到运行在同一主机的 5000 端口上的 UDP 服务器上。

```
DatagramChannel channel = DatagramChannel.open();

 ByteBuffer buffer = ByteBuffer.wrap("Hello".getBytes("UTF-16"));

channel.send(buffer, new InetSocketAddress("localhost", 5000));
```

以下代码段用于创建一个 DatagramChannel 实例，将底层的套接字绑定到 5000 端口，接收最长为 20 字节的数据报文，并将字节转换成使用 UTF-16 编码的字符串。

```
DatagramChannel channel = DatagramChannel.open();

 channel.socket().bind(new InetSocketAddress(5000));

ByteBuffer buffer = ByteBuffer.allocateDirect(20);

SocketAddress address = channel.receive(buffer);

buffer.flip();

String received = Charset.forName("UTF-16"). newDecoder().decode(buffer).toString();
```

在上面的 send()实例中，调用 send()方法时并没有显式地绑定本地端口，因此将随机选择一个可用端口。相应的 receive()方法用于返回一个 SocketAddress，其中包含了端口号。如果总是向同一个远程终端发送或接收数据，我们可以选择调用 connect()方法，并使用 SocketAddress 指定远程终端的地址。

DatagramChannel: 连接 DatagramChannel

```
DatagramChannel connect(SocketAddress remote)

DatagramChannel disconnect()

boolean isConnected()

 int read(ByteBuffer dst)

 long read(ByteBuffer[] dsts)

long read(ByteBuffer[] dsts, int offset, int length)

 int write(ByteBuffer src)

 long write(ByteBuffer[] srcs)
```

long write(ByteBuffer[] srcs, int offset, int length)

这些方法限制我们只能通过指定的地址发送和接收数据。为什么要这样做呢？原因之一是调用 connect()方法后，可以使用 read()和 write()方法来代替 receive()和 send()方法，并且不需要处理远程地址。read()和 write()方法分别用于接收和发送一个数据报文。分散式读操作以一个 ByteBuffer 数组为参数，只接收一个数据报文，并按顺序将其填入缓冲区中。聚集式写操作将缓冲区数组中的所有字节连接起来创建一个要传输的数据报文。重要提示：现在能够发送的最大数据报文可以包含 65507 个字节，试图发送更多的数据将被无提示地截断。使用 connect()方法的另一个好处是，已建立连接的数据报文信道可能只接收从指定终端发送来的数据，因此我们不需要测试接收端的有效性。注意，DatagramChannel 的 connect()方法只起到限制发送和接收终端的作用，连接时并没有数据包在 SocketChannel 上进行交换，而且也不需要像 SocketChannel 那样等待或测试连接是否完成。（见第 6 章）

到目前为止 DatagramChannel 看起来与 DatagramSocket 非常相似。数据报文信道和套接字的主要区别是，信道可以进行非阻塞 I/O 操作和使用选择器。DatagramChannel 中选择器的创建，信道的注册、选择等，与 SocketChannel 几乎一模一样。有一个区别是 DatagramChannel 不能注册连接 I/O 操作，不过也不需要这样做，因为 DatagramChannel 的 connect()方法永远不会阻塞。