Java 网络编程

引言

网络编程就是在两个或两个以上的设备(例如计算机)之间传输数据。程序员所作的事情就是把数据发送到指定的位置，或者接收到指定的数据，这个就是狭义的网络编程范畴。在发送和接收数据时，大部分的程序设计语言都设计了专门的 API 实现这些功能，程序员只需要调用即可。所以，基础的网络编程可以和打电话一样简单。

一、 网络概述

网络编程技术是当前一种主流的编程技术，随着联网趋势的逐步增强以及网络应用程序的大量出现，所以在实际的开发中网络编程技术获得了大量的使用。本章中以浅显的基础知识说明和实际的案例使广大初学者能够进入网络编程技术的大门，至于以后的实际修行就要阅读进阶的书籍以及进行大量的实际练习。

1. 计算机网络概述

网络编程的实质就是两个(或多个)设备(例如计算机)之间的数据传输。

按照计算机网络的定义，通过一定的物理设备将处于不同位置的计算机连接起来组成的网络，这个网络中包含的设备有：计算机、路由器、交换机等等。

其实从软件编程的角度来说，对于物理设备的理解不需要很深刻，就像你打电话时不需要很熟悉通信网络的底层实现是一样的，但是当深入到网络编程的底层时，这些基础知识是必须要补的。

路由器和交换机组成了核心的计算机网络，计算机只是这个网络上的节点以及控制等，通过光纤、网线等连接将设备连接起来，从而形成了一张巨大的计算机网络。

网络最主要的优势在于共享：共享设备和数据，现在共享设备最常见的是打印机，一个公司一般一个打印机即可，共享数据就是将大量的数据存储在一组机器中，其它的计算机通过网络访问这些数据，例如网站、银行服务器等等。

如果需要了解更多的网络硬件基础知识，可以阅读《计算机网络》教材，对于基础进行强化，这个在基础学习阶段不是必须的，但是如果想在网络编程领域有所造诣，则是一个必须的基本功。

对于网络编程来说，最主要的是计算机和计算机之间的通信，这样首要的问题就是如何找到网络上的计算机呢？这就需要了解 IP 地址的概念。

为了能够方便的识别网络上的每个设备，网络中的每个设备都会有一个唯一的数字标识，

这个就是 IP 地址。在计算机网络中,现在命名 IP 地址的规定是 IPv4 协议，该协议规定每个 IP 地址由 4 个 0-255 之间的数字组成，例如 10.0.120.34。每个接入网络的计算机都拥有唯一的 IP 地址，这个 IP 地址可能是固定的，例如网络上各种各样的服务器，也可以是动态的，例如使用 ADSL 拨号上网的宽带用户，无论以何种方式获得或是否是固定的，每个计算机在联网以后都拥有一个唯一的合法 IP 地址，就像每个手机号码一样。

但是由于 IP 地址不容易记忆，所以为了方便记忆，有创造了另外一个概念——域名(Domain Name)，例如 sohu.com 等。一个 IP 地址可以对应多个域名，一个域名只能对应一个 IP 地址。域名的概念可以类比手机中的通讯簿，由于手机号码不方便记忆，所以添加一个姓名标识号码，在实际拨打电话时可以选择该姓名，然后拨打即可。

在网络中传输的数据，全部是以 IP 地址作为地址标识，所以在实际传输数据以前需要将域名转换为 IP 地址，实现这种功能的服务器称之为 DNS 服务器，也就是通俗的说法叫做域名解析。例如当用户在浏览器输入域名时，浏览器首先请求 DNS 服务器，将域名转换为 IP 地址，然后将转换后的 IP 地址反馈给浏览器，然后再进行实际的数据传输。

当 DNS 服务器正常工作时，使用 IP 地址或域名都可以很方便的找到计算机网络中的某个设备，例如服务器计算机。当 DNS 不正常工作时，只能通过 IP 地址访问该设备。所以 IP 地址的使用要比域名通用一些。

IP 地址和域名很好的解决了在网络中找到一个计算机的问题，但是为了让一个计算机可以同时运行多个网络程序，就引入了另外一个概念——端口(port)。

在介绍端口的概念以前，首先来看一个例子，一般一个公司前台会有一个电话，每个员工会有一个分机，这样如果需要找到这个员工的话，需要首先拨打前台总机，然后转该分机号即可。这样减少了公司的开销，也方便了每个员工。在该示例中前台总机的电话号码就相当于 IP 地址，而每个员工的分机号就相当于端口。

有了端口的概念以后，在同一个计算机中每个程序对应唯一的端口，这样一个计算机上就可以通过端口区分发送给每个端口的数据了，换句话说，也就是一个计算机上可以并发运行多个网络程序，而不会在互相之间产生干扰。

在硬件上规定 ,端口的号码必须位于 0-65535 之间 ,每个端口唯一的对应一个网络程序，一个网络程序可以使用多个端口。这样一个网络程序运行在一台计算上时，不管是客户端还是服务器 ,都是至少占用一个端口进行网络通讯。在接收数据时 ,首先发送给对应的计算机，然后计算机根据端口把数据转发给对应的程序。

有了 IP 地址和端口的概念以后，在进行网络通讯交换时，就可以通过 IP 地址查找到该台计算机，然后通过端口标识这台计算机上的一个唯一的程序。这样就可以进行网络数据的交换了。

但是，进行网络编程时，只有 IP 地址和端口的概念还是不够的，下面就介绍一下基础的网络编程相关的软件基础知识。

2. 网络编程概述

按照前面的介绍，网络编程就是两个或多个设备之间的数据交换，其实更具体的说，网络编程就是两个或多个程序之间的数据交换，和普通的单机程序相比，网络程序最大的不同

就是需要交换数据的程序运行在不同的计算机上，这样就造成了数据交换的复杂。虽然通过 IP 地址和端口可以找到网络上运行的一个程序，但是如果需要进行网络编程，则还需要了解网络通讯的过程。

网络通讯基于“请求-响应”模型。为了理解这个模型，先来看一个例子，经常看电视的人肯定见过审讯的场面吧，一般是这样的：

警察：姓名

嫌疑犯：xxx

警察：性别

嫌疑犯：男

警察：年龄

嫌疑犯：29

……

在这个例子中，警察问一句，嫌疑犯回答一句，如果警察不问，则嫌疑犯保持沉默。这种一问一答的形式就是网络中的“请求-响应”模型。也就是通讯的一端发送数据，另外一端反馈数据，网络通讯都基于该模型。

在网络通讯中，第一次主动发起通讯的程序被称作客户端(Client)程序，简称客户端，而在第一次通讯中等待连接的程序被称作服务器端(Server)程序，简称服务器。一旦通讯建立，则客户端和服务器端完全一样，没有本质的区别。

由此，网络编程中的两种程序就分别是客户端和服务器端，例如 QQ 程序，每个 QQ 用户安装的都是 QQ 客户端程序，而 QQ 服务器端程序则运行在腾讯公司的机房中，为大量的 QQ 用户提供服务。这种网络编程的结构被称作客户端/服务器结构，也叫做 Client/Server 结构，简称 C/S 结构。

使用 C/S 结构的程序，在开发时需要分别开发客户端和服务器端，这种结构的优势在于由于客户端是专门开发的，所以根据需要实现各种效果，专业点说就是表现力丰富，而服务器端也需要专门进行开发。但是这种结构也存在着很多不足，例如通用性差，几乎不能通用等，也就是说一种程序的客户端只能和对应的服务器端通讯，而不能和其它服务器端通讯，在实际维护时，也需要维护专门的客户端和服务器端，维护的压力比较大。

其实在运行很多程序时，没有必要使用专用的客户端，而需要使用通用的客户端，例如浏览器，使用浏览器作为客户端的结构被称作浏览器/服务器结构，也叫做 Browser/Server 结构，简称为 B/S 结构。

使用 B/S 结构的程序，在开发时只需要开发服务器端即可，这种结构的优势在于开发的压力比较小，不需要维护客户端。但是这种结构也存在着很多不足，例如浏览器的限制比较大，表现力不强，无法进行系统级操作等。

总之 C/S 结构和 B/S 结构是现在网络编程中常见的两种结构，B/S 结构其实也就是一种特殊的 C/S 结构。

另外简单的介绍一下 P2P(Point to Point)程序，常见的如 BT、电驴等。P2P 程序是一种特殊的程序，应该一个 P2P 程序中既包含客户端程序，也包含服务器端程序，例如 BT，使用客户端程序部分连接其它的种子(服务器端)，而使用服务器端向其它的 BT 客户端传输数据。

如果这个还不是很清楚，其实 P2P 程序和手机是一样的，当手机拨打电话时就是使用客户端的作用，而手机处于待机状态时，可以接收到其它用户拨打的电话则起的就是服务器端的功能，只是一般的手机不能同时使用拨打电话和接听电话的功能，而 P2P 程序实现了该功能。

最后再介绍一个网络编程中最重要，也是最复杂的概念——协议(Protocol)。按照前面的介绍，网络编程就是运行在不同计算机中两个程序之间的数据交换。在实际进行数据交换时，为了让接收端理解该数据，计算机比较笨，什么都不懂的，那么就需要规定该数据的格式，这个数据的格式就是协议。

如果没有理解协议的概念，那么再举一个例子，记得有个电影叫《永不消逝的电波》，讲述的是地下党通过电台发送情报的故事，这里我们不探讨电影的剧情，而只关心电台发送的数据。在实际发报时，需要首先将需要发送的内容转换为电报编码，然后将电报编码发送出去，而接收端接收的是电报编码，如果需要理解电报的内容则需要根据密码本翻译出该电报的内容。这里的密码本就规定了一种数据格式，这种对于网络中传输的数据格式在网络编程中就被称作协议。

那么如何来编写协议格式呢？答案是随意。只要按照这种协议格式能够生成唯一的编码，按照该编码可以唯一的解析出发送数据的内容即可。也正因为各个网络程序之间协议格式的不同，所以才导致了客户端程序都是专用的结构。

在实际的网络程序编程中，最麻烦的内容不是数据的发送和接收，因为这个功能在几乎所有的程序语言中都提供了封装好的 API 进行调用，最麻烦的内容就是协议的设计以及协议的生产和解析，这个才是网络编程中最核心的内容。

关于网络编程的基础知识，就介绍这里，深刻理解 IP 地址、端口和协议等概念，将会极大的有助于后续知识的学习。

3. 网络通讯方式

在现有的网络中，网络通讯的方式主要有两种：

- TCP(传输控制协议)方式
- UDP(用户数据报协议)方式

为了方便理解这两种方式，还是先来看一个例子。大家使用手机时，向别人传递信息时有两种方式：拨打电话和发送短信。使用拨打电话的方式可以保证将信息传递给别人，因为别人接听电话时本身就确认接收到了该信息。而发送短信的方式价格低廉，使用方便，但是接收人有可能接收不到。

在网络通讯中，TCP 方式就类似于拨打电话，使用该种方式进行网络通讯时，需要建立专门的虚拟连接，然后进行可靠的数据传输，如果数据发送失败，则客户端会自动重发该数据。而 UDP 方式就类似于发送短信，使用这种方式进行网络通讯时，不需要建立专门的虚拟连接，传输也不是很可靠，如果发送失败则客户端无法获得。

这两种传输方式都是实际的网络编程中进行使用，重要的数据一般使用 TCP 方式进行数据传输，而大量的非核心数据则都通过 UDP 方式进行传递，在一些程序中甚至结合使用这两种方式进行数据的传递。

由于 TCP 需要建立专用的虚拟连接以及确认传输是否正确，所以使用 TCP 方式的速度

稍微慢一些，而且传输时产生的数据量要比 UDP 稍微大一些。

关于网络编程的基础知识就介绍这么多，如果需要深入了解相关知识请阅读专门的计算机网络书籍，下面开始介绍 Java 语言中网络编程的相关技术。

二、 网络编程技术

1. 网络编程步骤

按照前面的基础知识介绍，无论使用 TCP 方式还是 UDP 方式进行网络通讯，网络编程都是由客户端和服务器端组成。当然，B/S 结构的编程中只需要实现服务器端即可。所以，下面介绍网络编程的步骤时，均以 C/S 结构为基础进行介绍。

说明：这里的步骤实现和语言无关，也就是说，这个步骤适用于各种语言实现，不局限于 Java 语言。

(一) 客户端网络编程步骤

客户端(Client)是指网络编程中首先发起连接的程序，客户端一般实现程序界面和基本逻辑实现，在进行实际的客户端编程时，无论客户端复杂还是简单，以及客户端实现的方式，客户端的编程主要由三个步骤实现：

1) 建立网络连接

客户端网络编程的第一步都是建立网络连接。在建立网络连接时需要指定连接到的服务器的 IP 地址和端口号，建立完成以后，会形成一条虚拟的连接，后续的操作就可以通过该连接实现数据交换了。

2) 交换数据

连接建立以后，就可以通过这个连接交换数据了。交换数据严格按照请求响应模型进行，由客户端发送一个请求数据到服务器，服务器反馈一个响应数据给客户端，如果客户端不发送请求则服务器端就不响应。

根据逻辑需要，可以多次交换数据，但是还是必须遵循请求响应模型。

3) 关闭网络连接

在数据交换完成以后，关闭网络连接，释放程序占用的端口、内存等系统资源，结束网络编程。

最基本的步骤一般都是这三个步骤，在实际实现时，步骤 2 会出现重复，在进行代码组

织时，由于网络编程是比较耗时的操作，所以一般开启专门的现场进行网络通讯。

(二) 服务器端网络编程步骤

服务器端(Server)是指在网络编程中被动等待连接的程序，服务器端一般实现程序的核心逻辑以及数据存储等核心功能。服务器端的编程步骤和客户端不同，是由四个步骤实现，依次是：

1) 监听端口

服务器端属于被动等待连接，所以服务器端启动以后，不需要发起连接，而只需要监听本地计算机的某个固定端口即可。

这个端口就是服务器端开放给客户端的端口，服务器端程序运行的本地计算机的 IP 地址就是服务器端程序的 IP 地址。

2) 获得连接

当客户端连接到服务器端时，服务器端就可以获得一个连接，这个连接包含客户端的信息，例如客户端 IP 地址等等，服务器端和客户端也通过该连接进行数据交换。

一般在服务器端编程中，当获得连接时，需要开启专门的线程处理该连接，每个连接都由独立的线程实现。

3) 交换数据

服务器端通过获得的连接进行数据交换。服务器端的数据交换步骤是首先接收客户端发送过来的数据，然后进行逻辑处理，再把处理以后的结果数据发送给客户端。简单来说，就是先接收再发送，这个和客户端的数据交换数序不同。

其实，服务器端获得的连接和客户端连接是一样的，只是数据交换的步骤不同。

当然，服务器端的数据交换也是可以多次进行的。

在数据交换完成以后，关闭和客户端的连接。

4) 关闭连接

当服务器程序关闭时，需要关闭服务器端，通过关闭服务器端使得服务器监听的端口以及占用的内存可以释放出来，实现了连接的关闭。

其实服务器端编程的模型和呼叫中心的实现是类似的，例如移动的客服电话 10086 就是典型的呼叫中心，当一个用户拨打 10086 时，转接给一个专门的客服人员，由该客服实现和该用户的问题解决，当另外一个用户拨打 10086 时，则转接给另一个客服，实现问题解决，依次类推。

在服务器端编程时，10086 这个电话号码就类似于服务器端的端口号码，每个用户就相当于一个客户端程序，每个客服人员就相当于服务器端启动的专门和客户端连接的线程，每个线程都是独立进行交互的。

这就是服务器端编程的模型，只是 TCP 方式是需要建立连接的，对于服务器端的压力比较大，而 UDP 是不需要建立连接的，对于服务器端的压力比较小罢了。

(三) 小结

总之，无论使用任何语言，任何方式进行基础的网络编程，都必须遵循固定的步骤进行操作，在熟悉了这些步骤以后，可以根据需要进行逻辑上的处理，但是还是必须遵循固定的步骤进行。

其实，基础的网络编程本身不难，也不需要很多的基础网络知识，只是由于编程的基础功能都已经由 API 实现，而且需要按照固定的步骤进行，所以在入门时有一定的门槛，希望下面的内容能够将你快速的带入网络编程技术的大门。

2. Java 网络编程技术

Java 语言是在网络环境下诞生的，所以 Java 语言虽然不能说是对于网络编程的支持最好的语言，但是必须说是一种对于网络编程提供良好支持的语言，使用 Java 语言进行网络编程将是一件比较轻松的工作。

和网络编程有关的基本 API 位于 java.net 包中，该包中包含了基本的网络编程实现，该包是网络编程的基础。该包中既包含基础的网络编程类，也包含封装后的专门处理 WEB 相关的处理类。在本章中，将只介绍基础的网络编程类。

首先来介绍一个基础的网络类——InetAddress 类。该类的功能是代表一个 IP 地址，并且将 IP 地址和域名相关的操作方法包含在该类的内部。

关于该类的使用，下面通过一个基础的代码示例演示该类的使用，代码如下：

```
package inetaddressdemo;
import java.net.*;
/**
 * 演示 InetAddress 类的基本使用
 */
public class InetAddressDemo {
        public static void main(String[] args) {
                try{
                        //使用域名创建对象
                        InetAddress inet1 =
InetAddress.getByName("www.163.com");
                        System.out.println(inet1);
                        //使用 IP 创建对象
```

```
                        InetAddress inet2 = InetAddress.getByName("127.0.0.1");
                        System.out.println(inet2);
                        //获得本机地址对象
                        InetAddress inet3 = InetAddress.getLocalHost();
                        System.out.println(inet3);
                        //获得对象中存储的域名
                        String host = inet3.getHostName();
                        System.out.println("域名：" + host);
                        //获得对象中存储的 IP
                        String ip = inet3.getHostAddress();
                        System.out.println("IP:" + ip);
                }catch(Exception e){}
        }
}
```

在该示例代码中，演示了 InetAddress 类的基本使用，并使用了该类中的几个常用方法，该代码的执行结果是：

```
www.163.com/220.181.28.50
/127.0.0.1
chen/192.168.1.100
域名：chen
IP:192.168.1.100
```

说明：由于该代码中包含一个互联网的网址，所以运行该程序时需要联网，否则将产生异常。

在后续的使用中，经常包含需要使用 InetAddress 对象代表 IP 地址的构造方法，当然，该类的使用不是必须的，也可以使用字符串来代表 IP 地址进行实现。

3. TCP 编程

按照前面的介绍，网络通讯的方式有 TCP 和 UDP 两种，其中 TCP 方式的网络通讯是指在通讯的过程中保持连接，有点类似于打电话，只需要拨打一次号码(建立一次网络连接)，就可以多次通话(多次传输数据)。这样方式在实际的网络编程中，由于传输可靠，类似于打电话，如果甲给乙打电话，乙说没有听清楚让甲重复一遍，直到乙听清楚为止，实际的网络传输也是这样，如果发送的一方发送的数据接收方觉得有问题，则网络底层会自动要求发送方重发，直到接收方收到为止。

在 Java 语言中，对于 TCP 方式的网络编程提供了良好的支持，在实际实现时，以 java.net.Socket 类代表客户端连接，以 java.net.ServerSocket 类代表服务器端连接。在进行网络编程时，底层网络通讯的细节已经实现了比较高的封装，所以在程序员实际编程时，只需要指定 IP 地址和端口号码就可以建立连接了。正是由于这种高度的封装，一方面简化了 Java 语言网络编程的难度，另外也使得使用 Java 语言进行网络编程时无法深入到网络的底层，所以使用 Java 语言进行网络底层系统编程很困难，具体点说，Java 语言无法实现底层的

网络嗅探以及获得 IP 包结构等信息。但是由于 Java 语言的网络编程比较简单，所以还是获得了广泛的使用。

在使用 TCP 方式进行网络编程时，需要按照前面介绍的网络编程的步骤进行，下面分别介绍一下在 Java 语言中客户端和服务器端的实现步骤。

在客户端网络编程中，首先需要建立连接，在 Java API 中以 java.net.Socket 类的对象代表网络连接，所以建立客户端网络连接，也就是创建 Socket 类型的对象，该对象代表网络连接，示例如下：

```
Socket socket1 = new Socket("192.168.1.103",10000);
Socket socket2 = new Socket("www.sohu.com",80);
```

上面的代码中，socket1 实现的是连接到 IP 地址是 192.168.1.103 的计算机的 10000 号端口，而 socket2 实现的是连接到域名是 www.sohu.com 的计算机的 80 号端口，至于底层网络如何实现建立连接，对于程序员来说是完全透明的。如果建立连接时，本机网络不通，或服务器端程序未开启，则会抛出异常。

连接一旦建立，则完成了客户端编程的第一步，紧接着的步骤就是按照“请求-响应”模型进行网络数据交换，在 Java 语言中，数据传输功能由 Java IO 实现，也就是说只需要从连接中获得输入流和输出流即可，然后将需要发送的数据写入连接对象的输出流中，在发送完成以后从输入流中读取数据即可。示例代码如下：

```
OutputStream os = socket1.getOutputStream(); //获得输出流
InputStream is = socket1.getInputStream();     //获得输入流
```

上面的代码中，分别从 socket1 这个连接对象获得了输出流和输入流对象，在整个网络编程中，后续的数据交换就变成了 IO 操作，也就是遵循“请求-响应”模型的规定，先向输出流中写入数据，这些数据会被系统发送出去，然后在从输入流中读取服务器端的反馈信息，这样就完成了一次数据交换过程，当然这个数据交换过程可以多次进行。

这里获得的只是最基本的输出流和输入流对象，还可以根据前面学习到的 IO 知识，使用流的嵌套将这些获得到的基本流对象转换成需要的装饰流对象，从而方便数据的操作。

最后当数据交换完成以后，关闭网络连接，释放网络连接占用的系统端口和内存等资源，完成网络操作，示例代码如下：

```
socket1.close();
```

这就是最基本的网络编程功能介绍。下面是一个简单的网络客户端程序示例，该程序的作用是向服务器端发送一个字符串“Hello”，并将服务器端的反馈显示到控制台，数据交换只进行一次，当数据交换进行完成以后关闭网络连接，程序结束。实现的代码如下：

```
package tcp;
import java.io.*;
import java.net.*;
/**
 * 简单的 Socket 客户端
 * 功能为：发送字符串“Hello”到服务器端，并打印出服务器端的反馈
 */
```

```
public class SimpleSocketClient {
        public static void main(String[] args) {
                Socket socket = null;
                InputStream is = null;
                OutputStream os = null;
                //服务器端 IP 地址
                String serverIP = "127.0.0.1";
                //服务器端端口号
                int port = 10000;
                //发送内容
                String data = "Hello";
                try {
                        //建立连接
                        socket = new Socket(serverIP,port);
                        //发送数据
                        os = socket.getOutputStream();
                        os.write(data.getBytes());
                        //接收数据
                        is = socket.getInputStream();
                        byte[] b = new byte[1024];
                        int n = is.read(b);
                        //输出反馈数据
                        System.out.println("服务器反馈：" + new String(b,0,n));
                } catch (Exception e) {
                        e.printStackTrace(); //打印异常信息
                }finally{
                        try {
                                //关闭流和连接
                                is.close();
                                os.close();
                                socket.close();
                        } catch (Exception e2) {}
                }
        }
}
```

在该示例代码中建立了一个连接到 IP 地址为 127.0.0.1，端口号码为 10000 的 TCP 类型的网络连接，然后获得连接的输出流对象，将需要发送的字符串“Hello”转换为 byte 数组写入到输出流中，由系统自动完成将输出流中的数据发送出去，如果需要强制发送，可以调用输出流对象中的 flush 方法实现。在数据发送出去以后，从连接对象的输入流中读取服

务器端的反馈信息，读取时可以使用 IO 中的各种读取方法进行读取，这里使用最简单的方法进行读取，从输入流中读取到的内容就是服务器端的反馈，并将读取到的内容在客户端的控制台进行输出，最后依次关闭打开的流对象和网络连接对象。

这是一个简单的功能示例，在该示例中演示了 TCP 类型的网络客户端基本方法的使用，该代码只起演示目的，还无法达到实用的级别。

如果需要在控制台下面编译和运行该代码，需要首先在控制台下切换到源代码所在的目录，然后依次输入编译和运行命令：

```
javac -d . SimpleSocketClient.java
        java tcp.SimpleSocketClient
```

和下面将要介绍的 SimpleSocketServer 服务器端组合运行时，程序的输出结果为：

```
服务器反馈：Hello
```

介绍完一个简单的客户端编程的示例，下面接着介绍一下 TCP 类型的服务器端的编写。首先需要说明的是，客户端的步骤和服务器端的编写步骤不同，所以在学习服务器端编程时注意不要和客户端混淆起来。

在服务器端程序编程中，由于服务器端实现的是被动等待连接，所以服务器端编程的第一个步骤是监听端口，也就是监听是否有客户端连接到达。实现服务器端监听的代码为：

```
ServerSocket ss = new ServerSocket(10000);
```

该代码实现的功能是监听当前计算机的 10000 号端口，如果在执行该代码时，10000 号端口已经被别的程序占用，那么将抛出异常。否则将实现监听。

服务器端编程的第二个步骤是获得连接。该步骤的作用是当有客户端连接到达时，建立一个和客户端连接对应的 Socket 连 接对象 ,从而释放客户端连接对于服务器端端口的占用。实现功能就像公司的前台一样，当一个客户到达公司时，会告诉前台我找某某某，然后前台就通知某某某， 然后就可以继续接待其它客户了。通过获得连接，使得客户端的连接在服务器端获得了保持，另外使得服务器端的端口释放出来，可以继续等待其它的客户端连接。 实现获得连接的代码是：

```
Socket socket = ss.accept();
```

该代码实现的功能是获得当前连接到服务器端的客户端连接。需要说明的是 accept 和前面 IO 部分介绍的 read 方法一样，都是一个阻塞方法，也就是当无连接时，该方法将阻塞程序的执行 ,直到连接到达时才执行该行代码。另外获得的连接会在服务器端的该端口注册，这样以后就可以通过在服务器端的注册信息直接通信，而注册以后服务器端的端口就被释放出来，又可以继续接受其它的连接了。

连接获得以后，后续的编程就和客户端的网络编程类似了，这里获得的 Socket 类型的连接就和客户端的网络连接一样了，只是服务器端需要首先读取发送过来的数据，然后进行逻辑处理以后再发送给客户端，也就是交换数据的顺序和客户端交换数据的步骤刚好相反。这部分的内容和客户端很类似，所以就不重复了，如果还不熟悉，可以参看下面的示例代码。

最后，在服务器端通信完成以后，关闭服务器端连接。实现的代码为：

```
ss.close();
```

这就是基本的TCP类型的服务器端编程步骤。下面以一个简单的echo服务实现为例子，介绍综合使用示例。echo 的意思就是“回声”，echo 服务器端实现的功能就是将客户端发送的内容再原封不动的反馈给客户端。实现的代码如下：

```
package tcp;
import java.io.*;
import java.net.*;
/**
 * echo 服务器
 * 功能：将客户端发送的内容反馈给客户端
 */
public class SimpleSocketServer {
	public static void main(String[] args) {
		ServerSocket serverSocket = null;
		Socket socket = null;
		OutputStream os = null;
		InputStream is = null;
		//监听端口号
		int port = 10000;
		try {
			//建立连接
			serverSocket = new ServerSocket(port);
			//获得连接
			socket = serverSocket.accept();
			//接收客户端发送内容
			is = socket.getInputStream();
			byte[] b = new byte[1024];
			int n = is.read(b);
			//输出
			System.out.println("客户端发送内容为：" + new
String(b,0,n));
			//向客户端发送反馈内容
			os = socket.getOutputStream();
			os.write(b, 0, n);
		} catch (Exception e) {
			e.printStackTrace();
		}finally{
			try{
				//关闭流和连接
				os.close();
				is.close();
```

```
                                socket.close();
                                serverSocket.close();
                        }catch(Exception e){}
                }
        }
}
```

在该示例代码中建立了一个监听当前计算机 10000 号端口的服务器端 Socket 连接，然后获得客户端发送过来的连接，如果有连接到达时，读取连接中发送过来的内容，并将发送的内容在控制台进行输出，输出完成以后将客户端发送的内容再反馈给客户端。最后关闭流和连接对象，结束程序。

在控制台下面编译和运行该程序的命令和客户端部分的类似。

这样，就以一个很简单的示例演示了 TCP 类型的网络编程在 Java 语言中的基本实现，这个示例只是演示了网络编程的基本步骤以及各个功能方法的基本使用，只是为网络编程打下了一个基础，下面将就几个问题来深入介绍网络编程深层次的一些知识。

为了一步一步的掌握网络编程，下面再研究网络编程中的两个基本问题，通过解决这两个问题将对网络编程的认识深入一层。

(一) 如何复用 Socket 连接？

在前面的示例中，客户端中建立了一次连接，只发送一次数据就关闭了，这就相当于拨打电话时，电话打通了只对话一次就关闭了，其实更加常用的应该是拨通一次电话以后多次对话，这就是复用客户端连接。

那么如何实现建立一次连接，进行多次数据交换呢？其实很简单，建立连接以后，将数据交换的逻辑写到一个循环中就可以了。这样只要循环不结束则连接就不会被关 闭。按照这种思路，可以改造一下上面的代码，让该程序可以在建立连接一次以后，发送三次数据，当然这里的次数也可以是多次，示例代码如下：

```
package tcp;
import java.io.*;
import java.net.*;
/**
 * 复用连接的 Socket 客户端
 * 功能为：发送字符串“Hello”到服务器端，并打印出服务器端的反馈
 */
public class MulSocketClient {
        public static void main(String[] args) {
                Socket socket = null;
                InputStream is = null;
                OutputStream os = null;
                //服务器端 IP 地址
```

```
                String serverIP = "127.0.0.1";
                //服务器端端口号
                int port = 10000;
                //发送内容
                String data[] ={"First","Second","Third"};
                try {
                        //建立连接
                        socket = new Socket(serverIP,port);
                        //初始化流
                        os = socket.getOutputStream();
                        is = socket.getInputStream();
                        byte[] b = new byte[1024];
                        for(int i = 0;i < data.length;i++){
                                //发送数据
                                os.write(data[i].getBytes());
                                //接收数据
                                int n = is.read(b);
                                //输出反馈数据
                                System.out.println("服务器反馈：" + new
String(b,0,n));
                        }
                } catch (Exception e) {
                        e.printStackTrace(); //打印异常信息
                }finally{
                        try {
                                //关闭流和连接
                                is.close();
                                os.close();
                                socket.close();
                        } catch (Exception e2) {}
                }
        }
}
```

该示例程序和前面的代码相比，将数据交换部分的逻辑写在一个 for 循环的内容，这样就可以建立一次连接，依次将 data 数组中的数据按照顺序发送给服务器端了。

如果还是使用前面示例代码中的服务器端程序运行该程序，则该程序的结果是：

```
java.net.SocketException: Software caused connection abort: recv failed
                atjava.net.SocketInputStream.socketRead0(Native Method)
                at java.net.SocketInputStream.read(SocketInputStream.java:129)
                at java.net.SocketInputStream.read(SocketInputStream.java:90)
```

```
                    at tcp.MulSocketClient.main(MulSocketClient.java:30)
服务器反馈：First
```

显然，客户端在实际运行时出现了异常，出现异常的原因是什么呢？如果仔细阅读前面的代码，应该还记得前面示例代码中的服务器端是对话一次数据以后就关闭了连接，如果服务器端程序关闭了，客户端继续发送数据肯定会出现异常，这就是出现该问题的原因。

按照客户端实现的逻辑，也可以复用服务器端的连接，实现的原理也是将服务器端的数据交换逻辑写在循环中即可，按照该种思路改造以后的服务器端代码为：

```
package tcp;
import java.io.*;
import java.net.*;
/**
 * 复用连接的 echo 服务器
 * 功能：将客户端发送的内容反馈给客户端
 */
public class MulSocketServer {
        public static void main(String[] args) {
                ServerSocket serverSocket = null;
                Socket socket = null;
                OutputStream os = null;
                InputStream is = null;
                //监听端口号
                int port = 10000;
                try {
                        //建立连接
                        serverSocket = new ServerSocket(port);
                        System.out.println("服务器已启动：");
                        //获得连接
                        socket = serverSocket.accept();
                        //初始化流
                        is = socket.getInputStream();
                        os = socket.getOutputStream();
                        byte[] b = new byte[1024];
                        for(int i = 0;i < 3;i++){
                                int n = is.read(b);
                                //输出
                                System.out.println("客户端发送内容为：" + new
String(b,0,n));
                                //向客户端发送反馈内容
                                os.write(b, 0, n);
```

```
                        }
                } catch (Exception e) {
                        e.printStackTrace();
                }finally{
                        try{
                                //关闭流和连接
                                os.close();
                                is.close();
                                socket.close();
                                serverSocket.close();
                        }catch(Exception e){}
                }
        }
}
```

在该示例代码中，也将数据发送和接收的逻辑写在了一个 for 循环内部，只是在实现时硬性的将循环次数规定成了 3 次，这样代码虽然比较简单，但是通用性比较差。

以该服务器端代码实现为基础运行前面的客户端程序时，客户端的输出为：

```
服务器反馈：First
服务器反馈：Second
服务器反馈：Third
```

服务器端程序的输出结果为：

```
服务器已启动：
客户端发送内容为：First
客户端发送内容为：Second
客户端发送内容为：Third
```

在该程序中，比较明显的体现出了“请求-响应”模型，也就是在客户端发起连接以后，首先发送字符串“First”给服务器端，服务器端输出客户端发送的内容“First”，然后将客户端发送的内容再反馈给客户端，这样客户端也输出服务器反馈“First”，这样就完成了客户端和服务器端的一次对话，紧接着客户端发送“Second”给服务器端，服务端输出“Second”，然后将“Second”再反馈给客户端，客户端再输出“Second”，从而完成第二次会话，第三次会话的过程和这个一样。在这个过程中，每次都是客户端程序首先发送数据给服务器端，服务器接收数据以后，将结果反馈给客户端，客户端接收到服务器端的反馈，从而完成一次通讯过程。

在该示例中，虽然解决了多次发送的问题，但是客户端和服务器端的次数控制还不够灵活，如果客户端的次数不固定怎么办呢？是否可以使用某个特殊的字符串，例如 quit，表示客户端退出呢,这就涉及到网络协议的内容了，会在后续的网络应用示例部分详细介绍。下面开始介绍另外一个网络编程的突出问题。

(二) 如何使服务器端支持多个客户端同时工作？

前面介绍的服务器端程序，只是实现了概念上的服务器端，离实际的服务器端程序结构距离还很遥远，如果需要让服务器端能够实际使用，那么最需要解决的问题就是——如何支持多个客户端同时工作。

一个服务器端一般都需要同时为多个客户端提供通讯，如果需要同时支持多个客户端，则必须使用前面介绍的线程的概念。简单来说，也就是当服务器端接收到一个连接时，启动一个专门的线程处理和该客户端的通讯。

按照这个思路改写的服务端示例程序将由两个部分组成，MulThreadSocketServer 类实现服务器端控制，实现接收客户端连接，然后开启专门的逻辑线程处理该连接，LogicThread 类实现对于一个客户端连接的逻辑处理，将处理的逻辑放置在该类的 run 方法中。该示例的代码实现为：

```
package tcp;
import java.net.ServerSocket;
import java.net.Socket;
/**
 * 支持多客户端的服务器端实现
 */
public class MulThreadSocketServer {
        public static void main(String[] args) {
                ServerSocket serverSocket = null;
                Socket socket = null;
                //监听端口号
                int port = 10000;
                try {
                        //建立连接
                        serverSocket = new ServerSocket(port);
                        System.out.println("服务器已启动：");
                        while(true){
                                //获得连接
                                socket = serverSocket.accept();
                                //启动线程
                                new LogicThread(socket);
                        }
                } catch (Exception e) {
                       e.printStackTrace();
                }finally{
                        try{
                                //关闭连接
```

```
                                serverSocket.close();
                        }catch(Exception e){}
                }
        }
}
```

在该示例代码中，实现了一个 while 形式的死循环，由于 accept 方法是阻塞方法，所以当客户端连接未到达时，将阻塞该程序的执行，当客户端到达时接收该连接，并启动一个新的 LogicThread 线程处理该连接，然后按照循环的执行流程，继续等待下一个客户端连接。这样当任何一个客户端连接到达时，都开启一个专门的线程处理，通过多个线程支持多个客户端同时处理。

下面再看一下 LogicThread 线程类的源代码实现：

```
package tcp;
import java.io.*;
import java.net.*;
/**
 * 服务器端逻辑线程
 */
public class LogicThread extends Thread {
        Socket socket;
        InputStream is;
        OutputStream os;
        public LogicThread(Socket socket){
                this.socket = socket;
                start(); //启动线程
        }
        public void run(){
                byte[] b = new byte[1024];
                try{
                        //初始化流
                        os = socket.getOutputStream();
                        is = socket.getInputStream();
                        for(int i = 0;i < 3;i++){
                                //读取数据
                                int n = is.read(b);
                                //逻辑处理
                                byte[] response = logic(b,0,n);
                                //反馈数据
                                os.write(response);
                        }
```

```
                }catch(Exception e){
                        e.printStackTrace();
                }finally{
                       close();
                }
        }
        /**
         * 关闭流和连接
         */
        private void close(){
                try{
                        //关闭流和连接
                        os.close();
                        is.close();
                        socket.close();
                }catch(Exception e){}
        }
        /**
         * 逻辑处理方法,实现 echo 逻辑
         * @param b 客户端发送数据缓冲区
         * @param off 起始下标
         * @param len 有效数据长度
         * @return
         */
        private byte[] logic(byte[] b,int off,int len){
                byte[] response = new byte[len];
                //将有效数据拷贝到数组 response 中
                System.arraycopy(b, 0, response, 0, len);
                return response;
        }
}
```

在该示例代码中，每次使用一个连接对象构造该线程，该连接对象就是该线程需要处理的连接，在线程构造完成以后，该线程就被启动起来了，然后在 run 方法内部对客户端连接进行处理，数据交换的逻辑和前面的示例代码一致，只是这里将接收到客户端发送过来的数据并进行处理的逻辑封装成了 logic 方法，按照前面介绍的 IO 编程的内容，客户端发送过来的内容存储在数组 b 的起始下标为 0，长度为 n 个中，这些数据是客户端发送过来的有效数据，将有效的数据传递给 logic 方法，logic 方法实现的是 echo 服务的逻辑，也就是将客户端发送的有效数据形成以后新的 response 数组，并作为返回值反馈。

在线程中将 logic 方法的返回值反馈给客户端 这样就完成了服务器端的逻辑处理模拟，

其他的实现和前面的介绍类似，这里就不在重复了。

这里的示例还只是基础的服务器端实现，在实际的服务器端实现中，由于硬件和端口数的限制，所以不能无限制的创建线程对象，而且频繁的创建线程对象效率也比较低，所以程序中都实现了线程池来提高程序的执行效率。

这里简单介绍一下线程池的概念，线程池(Thread pool)是池技术的一种，就是在程序启动时首先把需要个数的线程对象创建好，例如创建 5000 个线程对象，然后当客户端连接到达时从池中取出一个已经创建完成的线程对象使用即可。当客户端连接关闭以后，将该线程对象重新放入到线程池中供其它的客户端重复使用，这样可以提高程序的执行速度，优化程序对于内存的占用等。

关于基础的 TCP 方式的网络编程就介绍这么多，下面介绍 UDP 方式的网络编程在 Java 语言中的实现。

4. 网络编程 UDP

网络通讯的方式除了 TCP 方式以外，还有一种实现的方式就是 UDP 方式。UDP(User Datagram Protocol)，中文意思是用户数据报协议，方式类似于发短信息，是一种物美价廉的通讯方式，使用该种方式无需建立专用的虚拟连接，由于无需建立专用的连接，所以对于服务器的压力要比 TCP 小很多，所以也是一种常见的网络编程方式。但是使用该种方式最大的不足是传输不可靠，当然也不是说经常丢失，就像大家发短信息一样，理论上存在收不到的可能，这种可能性可能是 1%，反正比较小，但是由于这种可能的存在，所以平时我们都觉得重要的事情还是打个电话吧(类似 TCP 方式)，一般的事情才发短信息(类似 UDP 方式)。网络编程中也是这样，必须要求可靠传输的信息一般使用 TCP 方式实现，一般的数据才使用 UDP 方式实现。

UDP 方式的网络编程也在 Java 语言中获得了良好的支持，由于其在传输数据的过程中不需要建立专用的连接等特点，所以在 Java API 中设计的实现结构和 TCP 方式不太一样。当然，需要使用的类还是包含在 java.net 包中。

在 Java API 中，实现 UDP 方式的编程，包含客户端网络编程和服务器端网络编程，主要由两个类实现，分别是：

- **DatagramSocket**

DatagramSocket 类实现“网络连接”，包括客户端网络连接和服务器端网络连接。虽然 UDP 方式的网络通讯不需要建立专用的网络连接，但是毕竟还是需要发送和接收数据，DatagramSocket 实现的就是发送数据时的发射器，以及接收数据时的监听器的角色。类比于 TCP 中的网络连接，该类既可以用于实现客户端连接，也可以用于实现服务器端连接。

- **DatagramPacket**

DatagramPacket 类实现对于网络中传输的数据封装，也就是说，该类的对象代表网络中交换的数据。在 UDP 方式的网络编程中，无论是需要发送的数据还是需要接收的数据，都必须被处理成 DatagramPacket 类型的对象，该对象中包含发送到的地址、发送到的端口号以及发送的内容等。其实 DatagramPacket 类的作用类似于现实中的信件，在信件中包含信件发送到的地址以及接收人，还有发送的内容等，邮局只需要按照地址传递即可。在接收数

据时，接收到的数据也必须被处理成 DatagramPacket 类型的对象，在该对象中包含发送方的地址、端口号等信息，也包含数据的内容。和 TCP 方式的网络传输相比，IO 编程在 UDP 方式的网络编程中变得不是必须的内容，结构也要比 TCP 方式的网络编程简单一些。

下面介绍一下 UDP 方式的网络编程中，客户端和服务器端的实现步骤，以及通过基础的示例演示 UDP 方式的网络编程在 Java 语言中的实现方式。

UDP 方式的网络编程，编程的步骤和 TCP 方式类似，只是使用的类和方法存在比较大的区别，下面首先介绍一下 UDP 方式的网络编程客户端实现过程。

UDP 客户端编程涉及的步骤也是 4 个部分：建立连接、发送数据、接收数据和关闭连接。

首先介绍 UDP 方式的网络编程中建立连接的实现。其中 UDP 方式的建立连接和 TCP 方式不同，只需要建立一个连接对象即可，不需要指定服务器的 IP 和端口号码。实现的代码为：

```
DatagramSocket ds = new DatagramSocket();
```

这样就建立了一个客户端连接，该客户端连接使用系统随机分配的一个本地计算机的未用端口号。在该连接中，不指定服务器端的 IP 和端口，所以 UDP 方式的网络连接更像一个发射器，而不是一个具体的连接。

当然，可以通过制定连接使用的端口号来创建客户端连接。

```
DatagramSocket ds = new DatagramSocket(5000);
```

这样就是使用本地计算机的 5000 号端口建立了一个连接。一般在建立客户端连接时没有必要指定端口号码。

接着，介绍一下 UDP 客户端编程中发送数据的实现。在 UDP 方式的网络编程中，IO 技术不是必须的，在发送数据时，需要将需要发送的数据内容首先转换为 byte 数组，然后将数据内容、服务器 IP 和服务器端口号一起构造成一个 DatagramPacket 类型的对象，这样数据的准备就完成了，发送时调用网络连接对象中的 send 方法发送该对象即可。例如将字符串“Hello”发送到 IP 是 127.0.0.1，端口号是 10001 的服务器，则实现发送数据的代码如下：

```
String s = “Hello”;
                String host = “127.0.0.1”;
                int port = 10001;
               //将发送的内容转换为 byte 数组
                byte[] b = s.getBytes();
                //将服务器 IP 转换为 InetAddress 对象
                InetAddress server = InetAddress.getByName(host);
                //构造发送的数据包对象
               DatagramPacket sendDp = new
DatagramPacket(b,b.length,server,port);
                //发送数据
                ds.send(sendDp);
```

在该示例代码中，不管发送的数据内容是什么，都需要转换为 byte 数组，然后将服务器端的 IP 地址构造成 InetAddress 类型的对象，在准备完成以后，将这些信息构造成一个

DatagramPacket 类型的对象，在 UDP 编程中，发送的数据内容、服务器端的 IP 和端口号，都包含在 DatagramPacket 对象中。在准备完成以后，调用连接对象 ds 的 send 方法把 DatagramPacket 对象发送出去即可。

按照 UDP 协议的约定，在进行数据传输时，系统只是尽全力传输数据，但是并不保证数据一定被正确传输，如果数据在传输过程中丢失，那就丢失了。

UDP 方式在进行网络通讯时，也遵循“请求-响应”模型，在发送数据完成以后，就可以接收服务器端的反馈数据了。

下面介绍一下 UDP 客户端编程中接收数据的实现。当数据发送出去以后，就可以接收服务器端的反馈信息了。接收数据在 Java 语言中的实现是这样的 :首先构造一个数据缓冲数组，该数组用于存储接收的服务器端反馈数据，该数组的长度必须大于或等于服务器端反馈的实际有效数据的长度。然后以该缓冲数组为基础构造一个 DatagramPacket 数据包对象，最后调用连接对象的 receive 方法接收数据即可。接收到的服务器端反馈数据存储在 DatagramPacket 类型的对象内部。实现接收数据以及显示服务器端反馈内容的示例代码如下：

```
//构造缓冲数组
                byte[] data = new byte[1024];
                //构造数据包对象
                DatagramPacket received = new DatagramPacket(data,data.length);
                //接收数据
                ds.receive(receiveDp);
                //输出数据内容
                byte[] b = receiveDp.getData(); //获得缓冲数组
                int len = receiveDp.getLength(); //获得有效数据长度
                String s = new String(b,0,len);
                System.out.println(s);
```

在该代码中，首先构造缓冲数组 data，这里设置的长度 1024 是预估的接收到的数据长度，要求该长度必须大于或等于接收到的数据长度，然后以该缓冲数组为基础，构造数据包对象，使用连接对象 ds 的 receive 方法接收反馈数据，由于在 Java 语言中，除 String 以外的其它对象都是按照地址传递，所以在 receive 方法内部可以改变数据包对象 receiveDp 的内容，这里的 receiveDp 的功能和返回值类似。数据接收到以后，只需要从数据包对象中读取出来就可以了，使用 DatagramPacket 对象中的 getData 方法可以获得数据包对象的缓冲区数组，但是缓冲区数组的长度一般大于有效数据的长度，换句话说，也就是缓冲区数组中只有一部分数据是反馈数据，所以需要使用 DatagramPacket 对象中的 getLength 方法获得有效数据的长度，则有效数据就是缓冲数组中的前有效数据长度个内容，这些才是真正的服务器端反馈的数据的内容。

UDP 方式客户端网络编程的最后一个步骤就是关闭连接。虽然 UDP 方式不建立专用的虚拟连接，但是连接对象还是需要占用系统资源，所以在使用完成以后必须关闭连接。关闭连接使用连接对象中的 close 方法即可，实现的代码如下：

```
ds.close();
```

需要说明的是，和 TCP 建立连接的方式不同，UDP 方式的同一个网络连接对象，可以发送到达不同服务器端 IP 或端口的数据包，这点是 TCP 方式无法做到的。

介绍完了 UDP 方式客户端网络编程的基础知识以后，下面再来介绍一下 UDP 方式服务器端网络编程的基础知识。

UDP 方式网络编程的服务器端实现和 TCP 方式的服务器端实现类似，也是服务器端监听某个端口，然后获得数据包，进行逻辑处理以后将处理以后的结果反馈给客户端，最后关闭网络连接，下面依次进行介绍。

首先 UDP 方式服务器端网络编程需要建立一个连接，该连接监听某个端口，实现的代码为：

```
DatagramSocket ds = new DatagramSocket(10010);
```

由于服务器端的端口需要固定，所以一般在建立服务器端连接时，都指定端口号。例如该示例代码中指定 10010 端口为服务器端使用的端口号，客户端端在连接服务器端时连接该端口号即可。

接着服务器端就开始接收客户端发送过来的数据，其接收的方法和客户端接收的方法一直，其中 receive 方法的作用类似于 TCP 方式中 accept 方法的作用，该方法也是一个阻塞方法，其作用是接收数据。

接收到客户端发送过来的数据以后，服务器端对该数据进行逻辑处理，然后将处理以后的结果再发送给客户端，在这里发送时就比客户端要麻烦一些，因为服务器端需要获得客户端的 IP 和客户端使用的端口号，这个都可以从接收到的数据包中获得。示例代码如下：

```
//获得客户端的 IP
    InetAddress clientIP = receiveDp.getAddress();
        //获得客户端的端口号
        Int clientPort = receiveDp.getPort();
```

使用以上代码，就可以从接收到的数据包对象 receiveDp 中获得客户端的 IP 地址和客户端的端口号，这样就可以在服务器端中将处理以后的数据构造成数据包对象，然后将处理以后的数据内容反馈给客户端了。

最后，当服务器端实现完成以后，关闭服务器端连接，实现的方式为调用连接对象的 close 方法，示例代码如下：

```
ds.close();
```

介绍完了 UDP 方式下的客户端编程和服务器端编程的基础知识以后，下面通过一个简单的示例演示 UDP 网络编程的基本使用。

该示例的功能是实现将客户端程序的系统时间发送给服务器端，服务器端接收到时间以后，向客户端反馈字符串“OK”。实现该功能的客户端代码如下所示：

```
package udp;
import java.net.*;
import java.util.*;
/**
```

```
 * 简单的 UDP 客户端，实现向服务器端发生系统时间功能
 */
public class SimpleUDPClient {
           public static void main(String[] args) {
                  DatagramSocket ds = null; //连接对象
                  DatagramPacket sendDp; //发送数据包对象
              DatagramPacket receiveDp; //接收数据包对象
                  String serverHost = "127.0.0.1"; //服务器 IP
              int serverPort = 10010; //服务器端口号
                  try{
                     //建立连接
                     ds = new DatagramSocket();
                     //初始化发送数据
                     Date d = new Date(); //当前时间
                    String content = d.toString(); //转换为字符串
                     byte[] data = content.getBytes();
                     //初始化发送包对象
                     InetAddress address = InetAddress.getByName(serverHost);
                     sendDp = new
DatagramPacket(data,data.length,address,serverPort);
                     //发送
                     ds.send(sendDp);
                     //初始化接收数据
                     byte[] b = new byte[1024];
                     receiveDp = new DatagramPacket(b,b.length);
                     //接收
                     ds.receive(receiveDp);
                     //读取反馈内容，并输出
                     byte[] response = receiveDp.getData();
                     int len = receiveDp.getLength();
                     String s = new String(response,0,len);
                     System.out.println("服务器端反馈为：" + s);
              }catch(Exception e){
                     e.printStackTrace();
              }finally{
                     try{
                        //关闭连接
                        ds.close();
                     }catch(Exception e){}
              }
```

```
        }
    }
```

在该示例代码中，首先建立 UDP 方式的网络连接，然后获得当前系统时间，这里获得的系统时间是客户端程序运行的本地计算机的时间，然后将时间字符串以及服务器端的 IP 和端口，构造成发送数据包对象，调用连接对象 ds 的 send 方法发送出去。在数据发送出去以后，构造接收数据的数据包对象，调用连接对象 ds 的 receive 方法接收服务器端的反馈，并输出在控制台。最后在 finally 语句块中关闭客户端网络连接。

和下面将要介绍的服务器端一起运行时，客户端程序的输出结果为：

```
服务器端反馈为：OK
```

下面是该示例程序的服务器端代码实现：

```
package udp;
    import java.net.*;
    /**
     * 简单 UDP 服务器端，实现功能是输出客户端发送数据，
       并反馈字符串“OK"给客户端
     */
   public class SimpleUDPServer {
       public static void main(String[] args) {
               DatagramSocket ds = null; //连接对象
               DatagramPacket sendDp; //发送数据包对象
               DatagramPacket receiveDp; //接收数据包对象
               final int PORT = 10010; //端口
                                   try{
                  //建立连接，监听端口
                  ds = new DatagramSocket(PORT);
                 System.out.println("服务器端已启动：");
                  //初始化接收数据
                  byte[] b = new byte[1024];
                  receiveDp = new DatagramPacket(b,b.length);
                  //接收
                  ds.receive(receiveDp);
                  //读取反馈内容，并输出
                  InetAddress clientIP = receiveDp.getAddress();
                  int clientPort = receiveDp.getPort();
                  byte[] data = receiveDp.getData();
                  int len = receiveDp.getLength();
                  System.out.println("客户端 IP：" +
clientIP.getHostAddress());
                  System.out.println("客户端端口：" + clientPort);
```

```
                    System.out.println("客户端发送内容：" + new
String(data,0,len));
                   //发送反馈
                    String response = "OK";
                    byte[] bData = response.getBytes();
                    sendDp = new
DatagramPacket(bData,bData.length,clientIP,clientPort);
                    //发送
                    ds.send(sendDp);
                                        }catch(Exception e){
                    e.printStackTrace();
                                        }finally{
                    try{
                       //关闭连接
                       ds.close();
                    }catch(Exception e){}
                                        }
           }
       }
```

在该服务器端实现中，首先监听 10010 号端口，和 TCP 方式的网络编程类似，服务器端的 receive 方法是阻塞方法，如果客户端不发送数据，则程序会在该方法处阻塞。当客户端发送数据到达服务器端时，则接收客户端发送过来的数据，然后将客户端发送的数据内容读取出来，并在服务器端程序中打印客户端的相关信息，从客户端发送过来的数据包中可以读取出客户端的 IP 以及客户端端口号，将反馈数据字符串“OK”发送给客户端，最后关闭服务器端连接，释放占用的系统资源，完成程序功能示例。

和前面 TCP 方式中的网络编程类似，这个示例也仅仅是网络编程的功能示例，也存在前面介绍的客户端无法进行多次数据交换，以及服务器端不支持多个客户端的问题，这两个问题也需要对于代码进行处理才可以很方便的进行解决。

在解决该问题以前，需要特别指出的是 UDP 方式的网络编程由于不建立虚拟的连接，所以在实际使用时和 TCP 方式存在很多的不同，最大的一个不同就是“无状态”。该特点指每次服务器端都收到信息，但是这些信息和连接无关，换句话说，也就是服务器端只是从信息是无法识别出是谁发送的，这样就要求发送信息时的内容需要多一些，这个在后续的示例中可以看到。

下面是实现客户端多次发送以及服务器端支持多个数据包同时处理的程序结构，实现的原理和 TCP 方式类似，在客户端将数据的发送和接收放入循环中，而服务器端则将接收到的每个数据包启动一个专门的线程进行处理。实现的代码如下：

```
package udp;
    import java.net.*;
    import java.util.*;
    /**
```

```
    * 简单的 UDP 客户端，实现向服务器端发生系统时间功能
    * 该程序发送 3 次数据到服务器端
    */
   public class MulUDPClient {
               public static void main(String[] args) {
              DatagramSocket ds = null; //连接对象
                                  DatagramPacket sendDp; //发送数据包对象
                                  DatagramPacket receiveDp; //接收数据包对象
                                  String serverHost = "127.0.0.1"; //服务器 IP
                                  int serverPort = 10012; //服务器端口号
                                  try{
                  //建立连接
                  ds = new DatagramSocket();
                  //初始化
                           InetAddress address =
InetAddress.getByName(serverHost);
                  byte[] b = new byte[1024];
                  receiveDp = new DatagramPacket(b,b.length);
                  System.out.println("客户端准备完成");
                  //循环 30 次，每次间隔 0.01 秒
                  for(int i = 0;i < 30;i++){
                                                  //初始化发送数据
                                                  Date d = new Date(); //当前时间
                                                 String content = d.toString();
//转换为字符串
                                                  byte[] data =
content.getBytes();
                                                  //初始化发送包对象
                                                  sendDp = new
DatagramPacket(data,data.length,address, serverPort);
                                                  //发送
                                                  ds.send(sendDp);
                                                  //延迟
                                                  Thread.sleep(10);
                                                  //接收
                                                  ds.receive(receiveDp);
                                                  //读取反馈内容，并输出
                                                  byte[] response =
receiveDp.getData();
                                                  int len =
receiveDp.getLength();
```

```
                                                    String s = new
String(response,0,len);
                                                    System.out.println("服务器端反
馈为：" + s);
                    }
                }catch(Exception e){
                    e.printStackTrace();
                }finally{
                    try{
                                                    //关闭连接
                                                    ds.close();
                   }catch(Exception e){}
                }
        }
    }
```

在该示例中，将和服务器端进行数据交换的逻辑写在一个 for 循环的内部，这样就可以实现和服务器端的多次交换了，考虑到服务器端的响应速度，在每次发送之间加入 0.01 秒的时间间隔。最后当数据交换完成以后关闭连接，结束程序。

实现该逻辑的服务器端程序代码如下：

```
package udp;
import java.net.*;
/**
* 可以并发处理数据包的服务器端
* 功能为：显示客户端发送的内容，并向客户端反馈字符串“OK”
*/
public class MulUDPServer {
public static void main(String[] args) {
DatagramSocket ds = null; //连接对象
DatagramPacket receiveDp; //接收数据包对象
final int PORT = 10012; //端口
byte[] b = new byte[1024];
receiveDp = new DatagramPacket(b,b.length);
try{
//建立连接，监听端口
ds = new DatagramSocket(PORT);
System.out.println("服务器端已启动：");
while(true){
//接收
ds.receive(receiveDp);
//启动线程处理数据包
```

```
new LogicThread(ds,receiveDp);
}
}catch(Exception e){
        e.printStackTrace();
}finally{
try{
//关闭连接
ds.close();
}catch(Exception e){}
}
}
}
```

该代码实现了服务器端的接收逻辑，使用一个循环来接收客户端发送过来的数据包，当接收到数据包以后启动一个 LogicThread 线程处理该数据包。这样服务器端就可以实现同时处理多个数据包了。

实现逻辑处理的线程代码如下：

```
package udp;
import java.net.*;
/**
 * 逻辑处理线程
 */
public class LogicThread extends Thread {
/**连接对象*/
DatagramSocket ds;
/**接收到的数据包*/
DatagramPacket dp;
public LogicThread(DatagramSocket ds,DatagramPacket dp){
this.ds = ds;
this.dp = dp;
start(); //启动线程
}
public void run(){
try{
//获得缓冲数组
byte[] data = dp.getData();
//获得有效数据长度
int len = dp.getLength();
//客户端 IP
InetAddress clientAddress = dp.getAddress();
```

```
//客户端端口
int clientPort = dp.getPort();
//输出
System.out.println("客户端 IP：" + clientAddress.getHostAddress());
System.out.println("客户端端口号：" + clientPort);
System.out.println("客户端发送内容：" + new String(data,0,len));
//反馈到客户端
byte[] b = "OK".getBytes();
DatagramPacket sendDp = new DatagramPacket(b,b.length,clientAddress,clientPort);
//发送
ds.send(sendDp);
}catch(Exception e){
e.printStackTrace();
}
}
}
```

在该线程中，只处理一次 UDP 通讯，当通讯结束以后线程死亡，在线程内部，每次获得客户端发送过来的信息，将获得的信息输出到服务器端程序的控制台，然后向客户端反馈字符串“OK”。

由于 UDP 数据传输过程中可能存在丢失，所以在运行该程序时可能会出现程序阻塞的情况。如果需要避免该问题，可以将客户端的网络发送部分也修改成线程实现。

关于基础的 UDP 网络编程就介绍这么多了，下面将介绍一下网络协议的概念。

5. 网络协议

网络协议是指对于网络中传输的数据格式的规定。对于网络编程初学者来说，没有必要深入了解 TCP/IP 协议簇，所以对于初学者来说去读大部头的《TCP/IP 协议》也不是一件很合适的事情，因为深入了解 TCP/IP 协议是网络编程提高阶段，也是深入网络编程底层时才需要做的事情。

对于一般的网络编程来说，更多的是关心网络上传输的逻辑数据内容，也就是更多的是应用层上的网络协议，所以后续的内容均以实际应用的数据为基础来介绍网络协议的概念。

那么什么是网络协议呢，下面看一个简单的例子。春节晚会上“小沈阳”和赵本山合作的小品《不差钱》中，小沈阳和赵本山之间就设计了一个协议，协议的内容为：

```
如果点的菜价钱比较贵是，就说没有。
```

按照该协议的规定，就有了下面的对话：

```
赵本山：4 斤的龙虾
小沈阳：（经过判断，得出价格比较高），没有
赵本山：鲍鱼
小沈阳：（经过判断，得出价格比较高），没有
```

这就是一种双方达成的一种协议约定,其实这种约定的实质和网络协议的实质是一样的。网络协议的实质也是客户端程序和服务器端程序对于数据的一种约定，只是由于以计算机为基础，所以更多的是使用数字来代表内容，这样就显得比较抽象一些。

下面再举一个简单的例子，介绍一些基础的网络协议设计的知识。例如需要设计一个简单的网络程序：网络计算器。也就是在客户端输入需要计算的数字和运算符，在 服务器端实现计算，并将计算的结果反馈给客户端。在这个例子中，就需要约定两个数据格式：客户端发送给服务器端的数据格式，以及服务器端反馈给客户端的数 据格式。

可能你觉得这个比较简单,例如客户端输入的数字依次是 12 和 432,输入的运算符是加号,可能最容易想到的数据格式是形成字符串“12+432”,这样格式的确比较容易阅读,但是服务器端在进行计算时，逻辑就比较麻烦，因为需要首先拆分该字符串，然后才能进行计算，所以可用的数据格式就有了一下几种：

“12，432，+”　　格式为：第一个数字，第二个数字，运算符

“12，+，432”　　格式为：第一个数字，运算符，第二个数字

其实以上两种数据格式很接近，比较容易阅读，在服务器端收到该数据格式以后，使用“,”为分隔符分割字符串即可。

假设对于运算符再进行一次约定，例如约定数字 0 代表+，1 代表减，2 代表乘，3 代表除，整体格式遵循以上第一种格式，则上面的数字生产的协议数据为：

“12，432，0”

这就是一种基本的发送的协议约定了。

另外一个需要设计的协议格式就是服务器端反馈的数据格式，其实服务器端主要反馈计算结果，但是在实际接受数据时，有可能存在格式错误的情况，这样就需要简单 的设计一下服务器端反馈的数据格式了。例如规定，如果发送的数据格式正确，则反馈结果，否则反馈字符串“错误”。这样就有了以下的数据格式：

客户端：“1,111,1”　　服务器端：”-110”

客户端：“123，23，0”　　服务器端：“146”

客户端：“1，2，5”　　服务器端：“错误”

这样就设计出了一种最最基本的网络协议格式，从该示例中可以看出，网络协议就是一种格式上的约定,可以根据逻辑的需要约定出各种数据格式,在进行设计时一般遵循“简单、通用、容易解析”的原则进行。

而对于复杂的网络程序来说，需要传输的数据种类和数据量都比较大，这样只需要依次设计出每种情况下的数据格式即可，例如 QQ 程序，在该程序中需要进行传输的网络数据种类很多，那么在设计时就可以遵循：登录格式、注册格式、发送消息格式等等，一一进行设计即可。所以对于复杂的网络程序来说，只是增加了更多的命令格式，在实际设计时的工作量增加不是太大。

不管怎么说，在网络编程中，对于同一个网络程序来说，一般都会涉及到两个网络协议格式：客户端发送数据格式和服务器端反馈数据格式，在实际设计时，需要一一对应。这就是最基本的网络协议的知识。

网络协议设计完成以后，在进行网络编程时，就需要根据设计好的协议格式，在程序中

进行对应的编码了，客户端程序和服务器端程序需要进行协议处理的代码分别如下。
客户端程序需要完成的处理为：

- 客户端发送协议格式的生成
- 服务器端反馈数据格式的解析

服务器端程序需要完成的处理为：

- 服务器端反馈协议格式的生成
- 客户端发送协议格式的解析

这里的生成是指将计算好的数据，转换成规定的数据格式，这里的解析指，从反馈的数据格式中拆分出需要的数据。在进行对应的代码编写时，严格遵循协议约定即可。

所以，对于程序员来说，在进行网络程序编写时，需要首先根据逻辑的需要设计网络协议格式，然后遵循协议格式约定进行协议生成和解析代码的编写，最后使用网络编程技术实现整个网络编程的功能。

由于各种网络程序使用不同的协议格式，所以不同网络程序的客户端之间无法通用。

而对于常见协议的格式，例如 HTTP(Hyper Text Transfer Protocol，超文本传输协议)、FTP(File Transfer Protocol，文件传输协议)，SMTP(Simple Mail Transfer Protocol，简单邮件传输协议)等等，都有通用的规定，具体可以查阅相关的 RFC 文档。

最后，对于一种网络程序来说，网络协议格式是该程序最核心的技术秘密，因为一旦协议格式泄漏，则任何一个人都可以根据该格式进行客户端的编写，这样将影响服务器端的实现，也容易出现一些其它的影响。

6. 小结

关于网络编程基本的技术就介绍这么多，该部分介绍了网络编程的基础知识，以及 Java 语言对于网络编程的支持，网络编程的步骤等，并详细介绍了 TCP 方式网络编程和 UDP 方式网络编程在 Java 语言中的实现。

网络协议也是网络程序的核心，所以在实际开始进行网络编程时，设计一个良好的协议格式也是必须进行的工作。

三、 网络编程示例

“实践出真知”，所以在进行技术学习时，还是需要进行很多的练习，才可以体会技术的奥妙，下面通过两个简单的示例，演示网络编程的实际使用。

1. 质数判别示例

该示例实现的功能是质数判断，程序实现的功能为客户端程序接收用户输入的数字，然后将用户输入的内容发送给服务器端，服务器端判断客户端发送的数字是否是质数，并将判断的结果反馈给客户端，客户端根据服务器端的反馈显示判断结果。

质数的规则是：最小的质数是 2，只能被 1 和自身整除的自然数。当用户输入小于 2 的数字，以及输入的内容不是自然数时，都属于非法输入。

网络程序的功能都分为客户端程序和服务器端程序实现，下面先描述一下每个程序分别实现的功能：

1、客户端程序功能：

- 接收用户控制台输入
- 判断输入内容是否合法
- 按照协议格式生成发送数据
- 发送数据
- 接收服务器端反馈
- 解析服务器端反馈信息，并输出

2、服务器端程序功能：

- 接收客户端发送数据
- 按照协议格式解析数据
- 判断数字是否是质数
- 根据判断结果，生成协议数据
- 将数据反馈给客户端

分解好了网络程序的功能以后，就可以设计网络协议格式了，如果该程序的功能比较简单，所以设计出的协议格式也不复杂。

客户端发送协议格式：

- 将用户输入的数字转换为字符串，再将字符串转换为 byte 数组即可。
- 例如用户输入 16，则转换为字符串“16”，使用 getBytes 转换为 byte 数组。
- 客户端发送“quit”字符串代表结束连接。

服务器端发送协议格式：

- 反馈数据长度为 1 个字节。数字 0 代表是质数，1 代表不是质数，2 代表协议格式错误。
- 例如客户端发送数字 12，则反馈 1，发送 13 则反馈 0，发送 0 则反馈 2。

功能设计完成以后，就可以分别进行客户端和服务器端程序的编写了，在编写完成以后联合起来进行调试即可。

下面分别以 TCP 方式和 UDP 方式实现该程序，注意其实现上的差异。不管使用哪种方式实现，客户端都可以多次输入数据进行判断。对于 UDP 方式来说，不需要向服务器端发送 quit 字符串。

以 TCP 方式实现的客户端程序代码如下：

```
package example1;
import java.io.*;
import java.net.*;
/**
 * 以 TCP 方式实现的质数判断客户端程序
 */
```

```
public class TCPPrimeClient {
        static BufferedReader br;
        static Socket socket;
        static InputStream is;
        static OutputStream os;
        /**服务器 IP*/
        final static String HOST = "127.0.0.1";
        /**服务器端端口*/
        final static int PORT = 10005;
        public static void main(String[] args) {
                init(); //初始化
                while(true){
                        System.out.println("请输入数字：");
                        String input = readInput(); //读取输入
                        if(isQuit(input)){ //判读是否结束
                                byte[] b = "quit".getBytes();
                                send(b);
                                break; //结束程序
                        }
                        if(checkInput(input)){ //校验合法
                                //发送数据
                                send(input.getBytes());
                                //接收数据
                                byte[] data = receive();
                                //解析反馈数据
                                parse(data);
                        }else{
                                System.out.println("输入不合法，请重新输入！");
                        }
                }
                close(); //关闭流和连接
        }
        /**
         * 初始化
         */
        private static void init(){
                try {
                        br = new BufferedReader(
                                        new InputStreamReader(System.in));
                        socket = new Socket(HOST,PORT);
```

```
                    is = socket.getInputStream();
                    os = socket.getOutputStream();
            } catch (Exception e) {}
    }
    /**
     * 读取客户端输入
     */
    private static String readInput(){
            try {
                    return br.readLine();
            } catch (Exception e) {
                    return null;
            }
    }
    /**
     * 判断是否输入 quit
     * @param input 输入内容
     * @return true 代表结束，false 代表不结束
     */
    private static boolean isQuit(String input){
            if(input == null){
                    return false;
            }else{
                    if("quit".equalsIgnoreCase(input)){
                            return true;
                    }else{
                            return false;
                    }
            }
    }
    /**
     * 校验输入
     * @param input 用户输入内容
     * @return true 代表输入符合要求，false 代表不符合
     */
    private static boolean checkInput(String input){
            if(input == null){
                    return false;
            }
           try{
```

```
                int n = Integer.parseInt(input);
                if(n >= 2){
                        return true;
                }else{
                        return false;
                }
        }catch(Exception e){
                return false; //输入不是整数
        }
}
/**
 * 向服务器端发送数据
 * @param data 数据内容
 */
private static void send(byte[] data){
        try{
                os.write(data);
        }catch(Exception e){}
}
/**
 * 接收服务器端反馈
 * @return 反馈数据
 */
private static byte[] receive(){
        byte[] b = new byte[1024];
        try {
                int n = is.read(b);
                byte[] data = new byte[n];
                //复制有效数据
                System.arraycopy(b, 0, data, 0, n);
                return data;
        } catch (Exception e){}
        return null;
}
/**
 * 解析协议数据
 * @param data 协议数据
 */
private static void parse(byte[] data){
        if(data == null){
```

```
                        System.out.println("服务器端反馈数据不正确！");
                        return;
                }
                byte value = data[0]; //取第一个 byte
                //按照协议格式解析
                switch(value){
                case 0:
                        System.out.println("质数");
                        break;
                case 1:
                        System.out.println("不是质数");
                        break;
                case 2:
                        System.out.println("协议格式错误");
                        break;
                }
        }
        /**
         * 关闭流和连接
         */
        private static void close(){
                try{
                        br.close();
                        is.close();
                        os.close();
                        socket.close();
                }catch(Exception e){
                        e.printStackTrace();
                }
        }
}
```

在该代码中，将程序的功能使用方法进行组织，使得结构比较清晰，核心的逻辑流程在 main 方法中实现。

以 TCP 方式实现的服务器端的代码如下：

```
package example1;
import java.net.*;
/**
 * 以 TCP 方式实现的质数判别服务器端
 */
```

```
public class TCPPrimeServer {
	public static void main(String[] args) {
		final int PORT = 10005;
		ServerSocket ss = null;
		try {
			ss = new ServerSocket(PORT);
			System.out.println("服务器端已启动：");
			while(true){
				Socket s = ss.accept();
				new PrimeLogicThread(s);
			}
		} catch (Exception e) {}
		finally{
			try {
				ss.close();
			} catch (Exception e2) {}
		}
	}
}

package example1;
import java.io.*;
import java.net.*;
/**
 * 实现质数判别逻辑的线程
 */
public class PrimeLogicThread extends Thread {
	Socket socket;
	InputStream is;
	OutputStream os;
	public PrimeLogicThread(Socket socket){
		this.socket = socket;
		init();
		start();
	}
	/**
	 * 初始化
	 */
	private void init(){
		try{
```

```
                is = socket.getInputStream();
                os = socket.getOutputStream();
        }catch(Exception e){}
    }
    public void run(){
        while(true){
            //接收客户端反馈
            byte[] data = receive();
            //判断是否是退出
            if(isQuit(data)){
                break; //结束循环
            }
            //逻辑处理
            byte[] b = logic(data);
            //反馈数据
            send(b);
        }
        close();
    }
    /**
     * 接收客户端数据
     * @return 客户端发送的数据
     */
    private byte[] receive(){
        byte[] b = new byte[1024];
        try {
            int n = is.read(b);
            byte[] data = new byte[n];
            //复制有效数据
            System.arraycopy(b, 0, data, 0, n);
            return data;
        } catch (Exception e){}
        return null;
    }
    /**
     * 向客户端发送数据
     * @param data 数据内容
     */
    private void send(byte[] data){
        try{
```

```
                os.write(data);
            }catch(Exception e){}
        }
        /**
         * 判断是否是 quit
         * @return 是返回 true，否则返回 false
         */
        private boolean isQuit(byte[] data){
            if(data == null){
                return false;
            }else{
                String s = new String(data);
                if(s.equalsIgnoreCase("quit")){
                    return true;
                }else{
                    return false;
                }
            }
        }
        private byte[] logic(byte[] data){
            //反馈数组
            byte[] b = new byte[1];
            //校验参数
            if(data == null){
                b[0] = 2;
                return b;
            }
            try{
                //转换为数字
                String s = new String(data);
                int n = Integer.parseInt(s);
                //判断是否是质数
                if(n >= 2){
                    boolean flag = isPrime(n);
                    if(flag){
                        b[0] = 0;
                    }else{
                        b[0] = 1;
                    }
                }else{
```

```
                                b[0] = 2; //格式错误
                                System.out.println(n);
                        }
                }catch(Exception e){
                        e.printStackTrace();
                        b[0] = 2;
                }
                return b;
        }
        /**
         *
         * @param n
         * @return
         */
        private boolean isPrime(int n){
                boolean b = true;
                for(int i = 2;i <= Math.sqrt(n);i++){
                        if(n % i == 0){
                                b = false;
                                break;
                        }
                }
                return b;
        }
        /**
         * 关闭连接
         */
        private void close(){
                try {
                        is.close();
                        os.close();
                        socket.close();
                } catch (Exception e){}
        }
}
```

本示例使用的服务器端的结构和前面示例中的结构一致，只是逻辑线程的实现相对来说要复杂一些，在线程类中的 logic 方法中实现了服务器端逻辑，根据客户端发送过来的数据，判断是否是质数，然后根据判断结果按照协议格式要求，生成客户端反馈数据，实现服务器端要求的功能。

2. 猜数字小游戏

下面这个示例是一个猜数字的控制台小游戏。该游戏的规则是：当客户端第一次连接到服务器端时，服务器端生产一个【0，50】之间的随机数字，然后客户端输入数字来猜该数字，每次客户端输入数字以后，发送给服务器端，服务器端判断该客户端发送的数字和随机数字的关系，并反馈比较结果，客户端总共有 5 次猜的机会，猜中时提示猜中，当输入”quit”时结束程序。

和前面的示例类似，在进行网络程序开发时，首先需要分解一下功能的实现，觉得功能是在客户端程序中实现还是在服务器端程序中实现。区分的规则一般是：客户端 程序实现接收用户输入等界面功能，并实现一些基础的校验降低服务器端的压力，而将程序核心的逻辑以及数据存储等功能放在服务器端进行实现。遵循该原则划分 的客户端和服务器端功能如下所示。

客户端程序功能列表：

- 接收用户控制台输入
- 判断输入内容是否合法
- 按照协议格式发送数据
- 根据服务器端的反馈给出相应提示

服务器端程序功能列表：

- 接收客户端发送数据
- 按照协议格式解析数据
- 判断发送过来的数字和随机数字的关系
- 根据判断结果生产协议数据
- 将生产的数据反馈给客户端

在该示例中，实际使用的网络命令也只有两条，所以显得协议的格式比较简单。

其中客户端程序协议格式如下：

- 将用户输入的数字转换为字符串，然后转换为 byte 数组
- 发送“quit”字符串代表退出

其中服务器端程序协议格式如下：

- 反馈长度为 1 个字节，数字 0 代表相等(猜中)，1 代表大了，2 代表小了，其它数字代表错误。

实现该程序的代码比较多，下面分为客户端程序实现和服务器端程序实现分别进行列举。

客户端程序实现代码如下：

```
package guess;
import java.net.*;
import java.io.*;
/**
 * 猜数字客户端
 */
public class TCPClient {
```

```
public static void main(String[] args) {
        Socket socket = null;
        OutputStream os = null;
        InputStream is = null;
        BufferedReader br = null;
        byte[] data = new byte[2];
        try{
                //建立连接
                socket = new Socket("127.0.0.1",10001);
                //发送数据
                os= socket.getOutputStream();
                //读取反馈数据
                is = socket.getInputStream();
                //键盘输入流
                br = new BufferedReader(
                                new InputStreamReader(System.in));
                //多次输入
                while(true){
                        System.out.println("请输入数字：");
                        //接收输入
                        String s = br.readLine();
                        //结束条件
                        if(s.equals("quit")){
                                os.write("quit".getBytes());
                                break;
                        }
                        //校验输入是否合法
                        boolean b = true;
                        try{
                                Integer.parseInt(s);
                        }catch(Exception e){
                                b = false;
                        }
                        if(b){ //输入合法
                                //发送数据
                                os.write(s.getBytes());
                                //接收反馈
                                is.read(data);
                                //判断
                                switch(data[0]){
```

```
                                case 0:
                                        System.out.println("相等！祝贺你！");
                                        break;
                                case 1:
                                        System.out.println("大了！");
                                        break;
                                case 2:
                                        System.out.println("小了！");
                                        break;
                                default:
                                        System.out.println("其它错误！");
                                }
                                //提示猜的次数
                                System.out.println("你已经猜了" + data[1] + "次！
");
                                //判断次数是否达到 5 次
                                if(data[1] >= 5){
                                        System.out.println("你挂了！");
                                        //给服务器端线程关闭的机会
                                        os.write("quit".getBytes());
                                        //结束客户端程序
                                        break;
                                }
                        }else{ //输入错误
                                System.out.println("输入错误！");
                        }
                }
        }catch(Exception e){
                e.printStackTrace();
        }finally{
                try{
                        //关闭连接
                        br.close();
                        is.close();
                        os.close();
                        socket.close();
                }catch(Exception e){
                        e.printStackTrace();
                }
        }
```

```
}
    }
```

在该示例中，首先建立一个到 IP 地址为 127.0.0.1 的端口为 10001 的连接，然后进行各个流的初始化工作，将逻辑控制的代码放入在一个 while 循环中，这样可以在客户端多次进行输入。在循环内部，首先判断用户输入的是否为 quit 字符串，如果是则结束程序，如果输入不是 quit，则首先校验输入的是否是数字，如果不是数字则直接输出“输入错误！”并继续接收用户输入，如果是数字则发送给服务器端，并根据服务器端的反馈显示相应的提示信息。最后关闭流和连接，结束客户端程序。

服务器端程序的实现还是分为服务器控制程序和逻辑线程，实现的代码分别如下：

```
package guess;
import java.net.*;
/**
 * TCP 连接方式的服务器端
 * 实现功能：接收客户端的数据，判断数字关系
 */
public class TCPServer {
 public static void main(String[] args) {
        try{
                //监听端口
                ServerSocket ss = new ServerSocket(10001);
                System.out.println("服务器已启动：");
                //逻辑处理
                while(true){
                        //获得连接
                        Socket s = ss.accept();
                        //启动线程处理
                        new LogicThread(s);
                }
        }catch(Exception e){
                e.printStackTrace();
        }
 }
    }

package guess;
import java.net.*;
import java.io.*;
import java.util.*;
/**
```

```
 * 逻辑处理线程
 */
public class LogicThread extends Thread {
	Socket s;
	static Random r = new Random();
	public LogicThread(Socket s){
		this.s = s;
		start(); //启动线程
	}
	public void run(){
		//生成一个[0，50]的随机数
		int randomNumber = Math.abs(r.nextInt() % 51);
		//用户猜的次数
		int guessNumber = 0;
		InputStream is = null;
		OutputStream os = null;
		byte[] data = new byte[2];
		try{
			//获得输入流
			is = s.getInputStream();
			//获得输出流
			os = s.getOutputStream();
			while(true){ //多次处理
				//读取客户端发送的数据
				byte[] b = new byte[1024];
				int n = is.read(b);
				String send = new String(b,0,n);
				//结束判别
				if(send.equals("quit")){
					break;
				}
				//解析、判断
				try{
					int num = Integer.parseInt(send);
					//处理
					guessNumber++; //猜的次数增加 1
					data[1] = (byte)guessNumber;
					//判断
					if(num > randomNumber){
						data[0] = 1;
```

```
                                        }else if(num < randomNumber){
                                                data[0] = 2;
                                        }else{
                                                data[0] = 0;
                                                //如果猜对
                                                guessNumber = 0; //清零
                                                randomNumber =
Math.abs(r.nextInt() % 51);
                                        }
                                        //反馈给客户端
                                        os.write(data);
                                }catch(Exception e){ //数据格式错误
                                        data[0] = 3;
                                        data[1] = (byte)guessNumber;
                                        os.write(data); //发送错误标识
                                        break;
                                }
                                os.flush();   //强制发送
                        }
                }catch(Exception e){
                        e.printStackTrace();
                }finally{
                        try{
                                is.close();
                                os.close();
                                s.close();
                        }catch(Exception e){}
                }
        }
}
```

在该示例中，服务器端控制部分和前面的示例中一样。也是等待客户端连接，如果有客户端连接到达时，则启动新的线程去处理客户端连接。在逻辑线程中实现程序的 核心逻辑，首先当线程执行时生产一个随机数字，然后根据客户端发送过来的数据，判断客户端发送数字和随机数字的关系，然后反馈相应的数字的值，并记忆客户 端已经猜过的次数，当客户端猜中以后清零猜过的次数，使得客户端程序可以继续进行游戏。

总体来说，该程序示例的结构以及功能都与上一个程序比较类似，希望通过比较这两个程序，加深对于网络编程的认识，早日步入网络编程的大门。