Python2.5 学习笔记

阿左 [1]　　Nobody [2]

October 7, 2012

[1]感谢读者
[2]感谢国家

Contents

List of Figures

List of Tables

Abstract

Java TCP/IP

摘要

Java TCP/IP

Part I

TCP/IP 基本概念

Chapter 1

简介

如今，人们可以通过电脑来打电话，看电视，给朋友发送即时信息，与其他人玩游戏，甚至可以通过电脑买到你能想到的任何东西，包括从歌曲到 SUV。计算机程序能够通过互联网相互通信使这一切成为了可能。很难统计现在有多少个人电脑接入互联网，但可以肯定，这个数量增长得非常迅速，相信不久就能达到 10 亿。除此之外，新的应用程序每天在互联网上层出不穷。随着日益增加的互联网访问带宽，我们可以预见，互联网将会对人们将来的生活产生长远的影响。

那么程序是如何通过网络进行相互通信的呢？本书的目的就是通过在 Java 编程语言环境下，带领你进入对这个问题的解答之路。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）。

在我们开始探究套接字的细节之前，有必要向读者简单介绍计算机网络和通信协议的整体框架，以使读者能清楚我们的代码将应用的地方。本章的目的不是向读者介绍计算机网络和 TCP/IP 协议是如何工作的（已经有很多相关内容的教程），而是介绍一些基本的概念和术语。

1.1 计算机网络，分组报文和协议

计算机网络由一组通过通信信道相互连接的机器组成。我们把这些机器称为主机（hosts）和路由器（routers）。主机是指运行应用程序的计算机，这些应用程序包括网络浏览器（Web browser），即时通讯代理（IM agent），或者是文件共享程序。运行在主机上的应用程序才是计算机网络的真正" 用户"。路由器的作用是将信息从一个通信信道传递或转发（forward）到另一个通信信道。路由器上可能会运行一些程序，但大多数情况下它们是不运行应用程序的。基于本书的目的对通信信道（communication channel）进行解释：它是将字节序列从一个主机传输到另一个主机的一种手段，可能是有线电缆，如以太网（Ethernet），也可能是无线的，如 WiFi，或是其他方式的连接。

路由器非常重要，因为要想直接将所有不同主机连接起来是不可行的。相反，一些主机先得连接到路由器，路由器再连接到其他路由器，这样就形成了网络。这种布局使每个主机只需要用到数量相对较少的通信信道，大部分主机仅需要一条信道。在网络上相互传递信息的程序并不直接与路由器进行交互，它们基本上感觉不到路由器的存在。

这里的信息（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。

要实现一个有用的网络，必须解决大量各种各样的问题。为了使这些问题可管理和模块化，人们设计了不同的协议来解决不同类型的问题。TCP/IP 协议就是这样一组的解决方案，有时也被称为协议族（protocol suite）。它刚好是互联网所使用的协议，不过也能用在独立的专用网络中。本书以后所提到的网络（network），都是指任何使用了 TCP/IP 协议族的网络。TCP/IP 协议族主要协议有 IP 协议（互联网协议，Internet Protocol），TCP 协

议（传输控制协议，Transmission Control Protocol）UDP 协议和（用户数据报协议，User Datagram Protocol）。

事实证明将各种协议分层组织是一种非常有用的措施，TCP/IP 协议族，实际上其他所有协议族都是按这种方式组织的。图 1.1 展示了通信协议、应用程序和主机和路由器中的套接字 API（应用程序接口，Application Programming Interface）之间的关系，同时也展示了数据流从一个应用程序到另一个应用程序的过程（使用 TCP 协议）。标记为 TCP，UDP 和 IP 的方框分别代表了这些协议的实现，它们通常驻留在主机的操作系统中。应用程序通过套接字 API 对 UDP 协议和 TCP 协议所提供的服务进行访问。箭头描述了数据流从一个应用程序，经过 TCP 协议层和 IP 协议层，通过网络，再反向经过 IP 协议层和 TCP 协议层传输到另一端的应用程序。

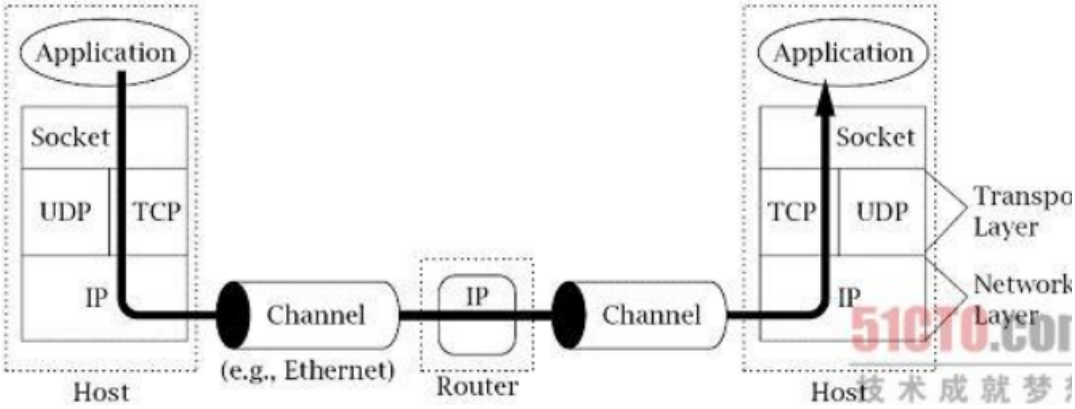

Figure 1.1: 一个 TCP/IP 网络

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。

在 TCP/IP 协议族中，底层由基础的通信信道构成，如以太网或调制解调器拨号连接。这些信道由网络层（network layer）使用，而网络层则完成将分组报文传输到它们的目的地址的工作（也就是路由器的功能）。TCP/IP 协议族中属于网络层的唯一协议是 IP 协议，它使两个主机间的一系列通信信道和路由器看起来像是一条单一的主机到主机的信道。

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。

1.2 关于地址

寄信的时候，要在表格中填上邮政服务能够理解的收信人的地址。在给别人打电话之前，必须给电话系统提供你所联系的人的电话号码。同样，一个程序要与另一个程序通信，就要给网络提供足够的信息，使其能够找到另一个程序。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，IPv4，即另一个版本是刚开始开发的版本 6，IPv6。即 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。

为了便于人们使用互联网地址（相对于程序内部的表示），两个版本的 IP 协议有不同的表示方法。IPv4 地址被表示为一组 4 个十进制数，每两个数字之间由圆点隔开（如：10.1.2.3），这种表示方法叫做点分形式（dotted-quad）。点分形式字符串中的 4 个数字代表了互联网地址的 4 个字节，也就是说，每个数字的范围是 0 到 255。

另一方面，IPv6 地址的 16 个字节由几组 16 进制的数字表示，这些 16 进制数之间由分号隔开（如：2000：fdb8：0000：0000：0001：00ab：853c：39a1）。每组数字分别代表了地址中的两个字节，并且每组开头的 0 可以省略，因此前面的例子中，第 5 组和第 6 组数字可以缩写为：1：ab:。甚至，只包含 0 的连续组可以全部省略（但在一个地址中只能这样做一次）。因此，前面的例子的完整地址可以表示为 2000：fdb8：：1：00ab：853c：39a1。

从技术角度来讲，每个互联网地址代表了一台主机与底层的通信信道的连接，换句话说，也就是一个网络接口（network interface）。主机可以有多个接口，这并不少见，例如一台主机同时连接了有线以太网（Ethernet）和无线网（WiFi）。由于每个这样的连接都属于唯一的一台主机，所以只要它连接到网络，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）。

TCP 或 UDP 协议中的端口号总与一个互联网地址相关联。回到前面我们作类比的例子，一个端口号就相当于指定街道上一栋大楼的某个房间号。邮政服务通过街道地址把信分发到一个邮箱，再由清空邮箱的人把这封信递送到这栋楼的正确房间中。或者考虑一个公司的内部电话系统：要与这个公司中的某个人通话，首先要拨打该公司的总机电话号码连接到其内部电话系统，然后再拨打你要找的那个人的分机号码。在上面的例子中，互联网地址就相对于街道地址或公司的总机电话号码，端口号就相当于房间号或分机号码。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。回环接口在测试中非常有用，因为发送给这个地址的报文能够立即返回到目标地址。而且每台主机上都有回环接口，即使当这台计算机没有其他接口 (也就是说没有连接到网络)，回环接口也能使用。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。如果你在拥有私有类型地址的计算机上试验本书的例子，并试图与另一台没有这类地址的主机进行通信，通常只有这台拥有私有类型地址的主机发起的通信才能成功。

相关的类型的地址包括本地链接 (link-local)，或称为" 自动配置" 地址。IPv4 中，这类地址由 169.254 开头，在 IPv6 中，前 16 位由 FE8 开头的地址是本地链接地址。这类地址只能用来在连接到同一网络的主机之间进行通信，路由器不会转发这类地址的信息。最后，另一类地址由多播 (multicast) 地址组成。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。我们将在

第 4 章中简要地对多播技术作进一步介绍。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。

1.3 关于名字

也许你更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。在本书第一版的写作期间，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（我们马上会对其进行更多的介绍），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，它将像 www.mkp.com 这样的域名映射到真实的互联网地址和其他信息上。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。

1.4 客户端和服务器

在前面的邮政和电话系统例子中，每次通信都是由发信方或打电话者发起，而另一方则通过发回反馈信或接听电话来对通信的发起者作出响应。互联网通信也与这个过程类似。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）。客户端和服务器这两个术语对典型的情况作出了描述，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。

客户端如何才能找到服务器的地址和端口号呢？通常情况，客户端知道服务器的名字，例如使用 URL（Universal Resource Locator，统一资源定位符）如 http：//www.mkp.com，再通过名字解析服务获取其相应的互联网地址。

获取服务器的端口号则是另一种情况。从原理上来讲，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。例如，端口号 21 被 FTP（File Transfer Protocol，文件传输协议）使用。当你运行 FTP 客户端应用程序时，它将默认通过这个端口号连接服务器。互联网的端口号授权机构维护了一个包含所有已约定使用的端口号列表（见 http：//www.iana.org/assignments/port-numbers）。

1.5 什么是套接字

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。

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，本书只涵盖了 TCP/IP 协议族的内容。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，但本书只对 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。随着进一步学习，你将了解到把一个套接字绑定到一个互联网地址上的多种方法。

图 1.2 描述了一个主机中，应用程序、套接字抽象层、协议、端口号之间的逻辑关系。

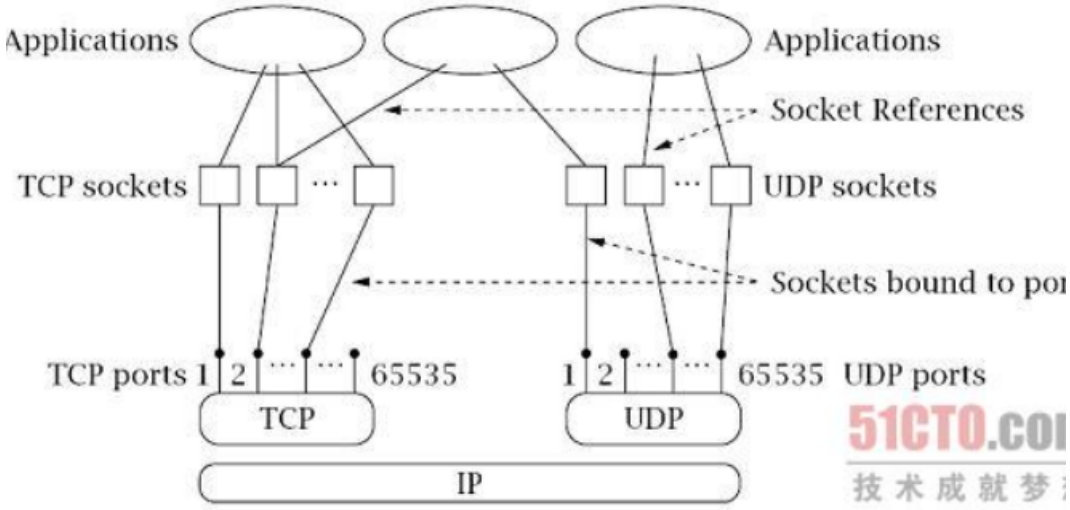

Figure 1.2: 套接字、协议、端口

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。

值得注意的是一个套接字抽象层可以被多个应用程序引用。每个使用了特定套接字的程序都可以通过那个套接字进行通信。前面已提到，每个端口都标识了一台主机上的一个应用程序。实际上，一个端口确定了一台主机上的一个套接字。从图 1.2 中我们可以看到，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。

Part II

Java API

Chapter 2

基本套接字

现在我们可以学习如何编写自己的套接字应用程序了。我们首先通过使用 InetAddress 类和 SocketAddress 类来示范 Java 应用程序如何识别网络主机。然后，举了一个使用 TCP 协议的客户端和服务器端例子来展示 Socket 类和 ServerSocket 类的用法。同样，我们举了一个使用 UDP 协议的客户端和服务器端例子来展示 DatagramSocket 类的用法。对于每个类对应的网络抽象，列举出了各个类中最重要的方法，根据这些方法的不同用途进行了分组，并简要描述了它们的功能。

2.1 套接字地址

回顾前面章节所讲的内容，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)。在后面的例子中，主机名必须被解析 (resolved) 成数字型地址才能用来进行通信。

InetAddress 类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。我们将通过一个示例程序来示范 InetAddress 类的用法。在这个例子中，首先打印出与本地主机关联的所有 IP 地址，包括 IPv4 和 IPv6，然后对于每个在命令行中指定的主机，打印出其相关的主机名和地址。为了获得本地主机地址，示例程序利用了 NetworkInterface 类的功能。前面已经讲过,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。NetworkInterface 类提供了访问主机所有接口的信息的功能。这个功能非常有用，比如当一个程序需要通知其他程序其 IP 地址时就会用到。

```
1  package ch02;
2
3  import java.net.Inet4Address;
4  import java.net.Inet6Address;
5  import java.net.InetAddress;
6  import java.net.NetworkInterface;
7  import java.net.SocketException;
8  import java.net.UnknownHostException;
9  import java.util.Enumeration;
10
11 public class InetAddressExample {
12
13   public static void main(String[] args) {
14     /*
15      * get the newwork interface and associate address for this host
16      */
17     try {
18       Enumeration<NetworkInterface> interfaceList =
19           NetworkInterface.getNetworkInterfaces();
20       if (null == interfaceList) {
21         System.out.println("--No interface found--");
22       } else {
23         while (interfaceList.hasMoreElements()) {
24           NetworkInterface iface = interfaceList.nextElement();
```

```
25            System.out.println("Interface " + iface.getName() + ":");
26            Enumeration<InetAddress> addrList =
27                iface.getInetAddresses();
28            if (!addrList.hasMoreElements()) {
29              System.out
30                  .println("\t(No addresses forthis interface)");
31            }
32            while (addrList.hasMoreElements()) {
33              InetAddress address = addrList.nextElement();
34              /*
35               * check address type
36               */
37              String addressType = "(?)";
38              if (address instanceof Inet4Address) {
39                addressType = "(v4)";
40              }
41              if (address instanceof Inet6Address) {
42                addressType = "(v6)";
43              }
44              System.out.println("\tAddress " + addressType //
45                  + ": " + address.getHostAddress());
46            }
47          }
48        }
49      } catch (SocketException e) {
50        System.out.println("Error getting network interface: "
51            + e.getMessage());
52      }
53
54      /*
55       * get name(s)/address(s) of hosts given on command line
56       */
57      for (String host : args) {
58        System.out.println(host + ": ");
```

```
59        try {
60          InetAddress[] addressList = InetAddress.getAllByName(host);
61          for (InetAddress address : addressList) {
62            System.out.println("\t" + address.getHostName() + "/"
63                + address.getHostAddress());
64          }
65        } catch (UnknownHostException e) {
66          System.out.println("\tUnable to find address for : " + host);
67        }
68      }
69
70    }
71
72  }
```

1. 获取主机的网络接口列表: 第 18-19 行

静态方法 getNetworkInterfaces() 返回一个列表, 其中包含了该主机每一个接口所对应的 NetworkInterface 类实例。

2. 空列表检测: 第 20-22 行

通常情况下, 即使主机没有任何其他网络连接, 回环接口也总是存在的。因此, 只要当一个主机根本没有网络子系统时, 列表检测才为空。

3. 获取并打印出列表中每个接口的地址: 第 23-47 行

打印接口名: 第 15 行 getName() 方法为接口返回一个本地名称。接口的本地名称通常由字母与数字的联合组成, 代表了接口的类型和具体实例, 如"lo0" 或"eth0"。

获取与接口相关联的地址: 第 26-27 行

getInetAddresses() 方法返回了另一个 Enumeration 类对象, 其中包含了 InetAddress 类的实例, 即该接口所关联的每一个地址。根据主机的不同配置, 这个地址列表可能只包含 IPv4 或 IPv6 地址, 或者是包含了两种类型地址的混合列表。

空列表检测: 第 28-31 行

列表的迭代, 打印出每个地址: 第 32-46 行

对每个地址实例进行检测以判断其属于哪个 IP 地址子类 (目前 InetAddress 的子类只有上面列出的那些, 但可以想像到, 将来也许还会有其他子类)。InetAddress 类的 getHostAddress() 方法返回一个字符串来代表主机的数字型地址。不同类型的地址对应了不同的格式:IPv4 是点分形式,IPv6 是冒号分隔的 16 进制形式。参考下文中的" 字符串表示法" 概要, 其对不同类型的 IP 地址格式进行了描述。

4. 捕获异常: 第 49-52 行

对 getNetworkInterfaces() 方法的调用将会抛出 SocketException 异常。

5. 获取从命令行输入的每个参数所对应的主机名和地址: 第 34-44 行

获取给定主机/地址的相关地址列表: 第 60 行

迭代列表, 打印出列表中的每一项: 第 61-64 行

对于列表中的每个主机, 我们通过调用 getHostName() 方法来打印主机名, 并把调用 getHostAddress() 方法所获得的数字型地址打印在主机名后面。

为了使用这个应用程序来获取本地主机信息、出版社网站 (www.mkp.com) 服务器信息、一个虚假地址信息 (blah.blah)、以及一个 IP 地址的信息, 需要在命令行中运行如下代码:

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

得到的结果为:

```
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
```

你也许已经注意到，一些 IPv6 地址带有%d 型式的后缀，其中 d 是一个数字。这样的地址在一个有限的范围内 (通常它们是本地链接)，其后缀表明了该地址所关联的特定范围。这就保证了列出的每个地址字符串都是唯一的。IPv6 的本地链接地址由 fe8 开头。

你可能还注意到，当程序解析 blah.blah 这个虚假地址时，会有一定的延迟。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。在本书的所有例子中，如果远程主机由名字指定，运行示例程序的主机必须配置为能够将名字解析成地址，否则示例程序将无法正确运行。如果能通过主机的名字 ping 到该主机 (如，在命令行窗口中执行命令"ping server.example.com")，那么在示例程序中就可以使用主机名。如果 ping 测试失败或示例程序挂起，可以尝试使用 IP 地址来定位主机，这就完全避免了从名字到地址的转换。(参见后文将要讨论的 InetAddress 类的 isReachable() 方法)

InetAddress 类中创建与访问实例方法：

```

static InetAddress [] getAllByName(String host);
static InetAddress getByName(String host);
static InetAddress getLocalHost();
/**
 * return byte array as ip address
 * size if 4 byte in ipv4
 * or size is 16 in ipv6
 */
```

这些静态工厂方法所返回的实例能够传递给另一个套接字方法来指定一个主机。这些方法的输入字符串可以是一个域名，如"skeezix" 或"farm.example.com"，也可以是一个代

表数字型地址的字符串。对于 IPv6 地址，第 1 章所提到的缩写形式同样适用。一个名字可能关联了多个数字地址,getAllByName() 方法用于返回一组与给定主机名相关联的所有地址的实例。

getAddress() 方法返回一个适当长度的字节数组，代表地址的二进制的形式。如果是一个 Inet4Address 实例，该数组长 4 个字节；如果是 Inet6Address 实例，则长 16 字节。返回的数组的第一个元素是该地址中最重要的字节。

我们已看到，一个 InetAddress 实例可以通过多种方式转换成字符串形式。

InetAddress 类中字符串显示方法：

```
1
2  /**
3   * return value format like:
4   *   hostname.example.com/192.0.2.127
5   *   or
6   *   never.example.net/2000::620:1a30:95b2
7   */
8  String toString();
9  String getHostAddress();
10 /**
11  * return host name
12  */
13 String getHostName();
14 /**
15  * return host name ()
16  */
```

上面这些方法返回主机名或数字型地址，或者以一定格式的字符串返回两者的联合形式。toString() 方法重写了 Object 类的方法，返回如"hostname.example.com/192.0.2.127"或"never.example.net/2000::620:1a30:95b2 " 形式的字符串。单一的数字型地址表示形式由 getHostAddress() 方法返回。对于 IPv6 地址，字符串中总是包含了完整的 8 组数字 (即显示地列出了 7 个":")，这样做是为了消除二义性。因为通常情况下，地址字符串后还会附有由另一个分号隔开的端口号，后面我们将看到这样的例子。而且，对于有范围限制的

IPv6 地址, 如本地链接地址, 还会在后面附有一个范围标识符 (scope identifier)。这只是一个用于消除二义性 (因为同样的本地链接地址能用于不同的链接中) 的本地标识符, 不是数据报文中所传输的地址的一部分。

最后两个方法只返回主机名, 它们的区别在于: 如果实例最初通过主机名创建,getHostName() 则直接返回这个名字, 没有解析的步骤; 否则,getHostName() 要通过系统配置的名字解析机制将地址解析成名字。另一方面,getCanonicalName() 方法总是尝试对地址进行解析, 以获取主机域名全称 (fully qualified domain name), 如"ns1.internat.net"或"bam.example.com"。注意, 如果不同名字映射到了同一地址, 该方法所返回的主机名可能与最初用于创建实例的主机名不同。如果名字解析失败, 两个方法都将返回数字型地址, 而且在发送任何消息之前, 都将用安全管理器进行许可检查。

InetAddress 类还支持地址属性的检查, 如判断其是否属于 1.2 节提到的" 特殊用途" 地址中的某一类, 以及检测其可达性, 即与主机进行报文交互的能力。

InetAddress 类中检查属性的方法:

```
1
2  boolean isAnyLocalAddress();
3  boolean isLinkLocalAddress();
4  boolean isLoopBackAddress();
5  // 是否为多播地址
6  boolean isMulticastAddress();
7  // 测试多播地址范围: global
8  boolean isMCGlobal();
9  // 测试多播地址范围: link local
10 boolean isMCLinkLocal();
11 // 测试多播地址范围: node local
12 boolean isMCNodeLocal();
13 // 测试多播地址范围: org local
14 boolean isMCOrgLocal();
15 // 测试多播地址范围: site local
16 boolean isMCSiteLocal();
17 boolean isReachable(int timeout);
```

这些方法检查一个地址是否属于某个特定类型。它们对 IPv4 地址和 IPv6 地址都适用。上述前三个方法分别检查地址实例是否属于" 任意" 本地地址, 本地链接地址, 以及回环地址 (匹配 127.*.*.* 或::1 的形式)。第 4 个方法检查其是否为一个多播地址 (见 4.3.2 节), 而 isMC...() 形式的方法检测多播地址的各种范围 (scopes)。(范围粗略地定义了到达该目的地址的数据报文从它的起始地址开始, 所能传递的最远距离。)

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 类提供了更多的方法, 其中有很多方法不属于本书的讨论范围。下面, 我们只对与我们所讨论的问题最有用的方法进行描述。

NetworkInterface 创建与获取的方法:

```
1 是否为多播地址测试多播地址范围: 测试多播地址范围: 测试多播地址范围: 测试多播地址范围: 测
    试多播地址范围:
2
3 static Enumeration<NetworkInterface> getNetworkInterfaces();
4 static NetworkInterface getByInetAddress(InetAddress addr);
5 static NetworkInterface getByName(InetAddress addr);
6 static Enumeration<InetAddress> getInetAddresses();
7 String getName();
```

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。

2.2 TCP 套接字

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。

我们从一个简单的客户端例子开始介绍。

2.2.1 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。(见 2.2.3 节)

3. 使用 Socket 类的 close() 方法关闭连接。

我们的第一个 TCP 应用程序叫 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。回馈服务器的功能只是简单地将收到的信息返回给客户端。在这个程序中, 要回馈的字符串以命令行参数的型式传递给我们的客户端。很多系统都包含了用于进行调试和测试的回馈服务程序。你也许可以使用 telnet 程序来检测你的系统上是否运行了标准的回馈服务程序 (如在命令行中输入"telnet server.example.com 7"), 或者继续阅读本书, 并运行下一节的服务器端示例程序。

```java
package ch02;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.net.SocketException;

public class TCPEchoClient {

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

    /*
     * Test for correct of args
     */
    if ((args.length < 2) || (args.length > 3)) {
      throw new IllegalArgumentException(
          "Parameter(s): <Server> <Word> [<Port>]");
    }

    // Server name or IP address
    String server = args[0];

    // Convert argument String to bytes
    // using the default character encoding
    byte[] data = args[1].getBytes();

    // get port, default port is 7
    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");
```

```
34
35      InputStream in = socket.getInputStream();
36      OutputStream out = socket.getOutputStream();
37
38      out.write(data); // Send the encoded string to the server
39
40      // Receive the same string back from the server
41      int totalBytesRcvd = 0; // Total bytes received so far
42      int bytesRcvd; // Bytes received in last read
43      while (totalBytesRcvd < data.length) {
44        if ((bytesRcvd = in.read(data, totalBytesRcvd, //
45            data.length - totalBytesRcvd)) == -1) {
46          throw new SocketException("Connection closed prematurely");
47        }
48        totalBytesRcvd += bytesRcvd;
49      } // data array is full
50
51      System.out.println("Received: " + new String(data));
52
53      socket.close(); // Close the socket and its streams
54    }
55  }
```

1. 应用程序设置与参数解析: 第 13-29 行

转换回馈字符串: 第 15 行 TCP 套接字发送和接收字节序列信息。String 类的 getBytes() 方法将返回代表该字符串的一个字节数组。(见 3.1 节讨论的字符编码)

确定回馈服务器的端口号: 第 29 行

默认端口号是 7。如果我们给出了第三个参数,Integer.parseInt() 方法就将第三个参数字符串转换成相应的整数, 并作为端口号。

2. 创建 TCP 套接字: 第 32 行

Socket 类的构造函数将创建一个套接字, 并将其连接到由名字或 IP 地址指定的服务器, 再将该套接字返回给程序。注意, 底层的 TCP 协议只能处理 IP 地址, 如果给出的是主机的

名字,Socket 类具体实现的时候会将其解析成相应的地址。若因某些原因连接失败, 构造函数将抛出一个 IOException 异常。

3. 获取套接字的输入输出流: 第 35-36 行

每个 Socket 实例都关联了一个 InputStream 和一个 OutputStream 对象。就像使用其他流一样, 我们通过将字节写入套接字的 OutputStream 来发送数据, 并通过从 InputStream 读取信息来接受数据。

4. 发送字符串到回馈服务器: 第 38 行

OutputStream 类的 write() 方法将指定的字节数组通过之前建立好的连接, 传送到指定的服务器。

5. 从回馈服务器接受回馈信息: 第 40-49 行

既然已经知道要从回馈服务器接收的字节数, 我们就能重复执行接收过程, 直到接收了与发送的字节数相等的信息。这个特殊型式的 read() 方法需要 3 个参数:

1) 接收数据的字节数组,

2) 接收的第一个字节应该放入数组的位置, 即字节偏移量,

3) 放入数组的最大字节数。

read() 方法在没有可读数据时会阻塞等待, 直到有新的数据可读, 然后读取指定的最大字节数, 并返回实际放入数组的字节数 (可能少于指定的最大字节数)。循环只是简单地将数据填入 data 字节数组, 直到接收的字节数与发送的字节数一样。如果 TCP 连接被另一端关闭,read() 方法返回 -1。对于客户端来说, 这表示服务器端提前关闭了套接字。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。

6. 打印回馈字符串: 第 51 行

要打印服务器的响应信息, 我们必须通过默认的字符编码将字节数组转换成一个字符串。

7. 关闭套接字: 第 53 行

当客户端接收到所有的回馈数据后，将关闭套接字。

我们可以使用以下两种方法来与一个名叫 server.example.com,IP 地址为 192.0.2.1 的回馈服务器进行通信。

Socket：创建

```
1 Socket(InetAddress remoteAddr, int remotePort);
2 Socket(String remoteHost, int remotePort);
3 Socket(InetAddress remoteAddr, int remotePort,
4     InetAddress localAddr, int localPort);
5 socket(String remoteHost, int remotePort,
6     Inetaddress localAddr, int localPort);
7 socket();
```

前四个构造函数在创建了一个 TCP 套接字后，先连接到 (connect) 指定的远程地址和端口号，再将其返回给程序。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，见下文)。

Socket：操作

```
1 void connect(SocketAddress distination);
2 void connect(SocketAddress destination, int timeout);
3 InputStream getInputStream();
4 OutputStream getOutputStream();
5 void close();
6 void shutdownInput();
7 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 (见下文介绍) 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。详情见 4.5 节。

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。由于本书是关于 TCP/IP 的，因此为了简便我们假设所有这些网络类的底层实现都与默认情况一致。

Socket：获取 / 检测属性

```
1 InetAddress getInetAddress();
2 int getPort();
3 InetAddress getLocalAddress();
4 int getLocalPort();
5 SocketAddress getRemoteSocketAddress();
6 SocketAddress getLocalSocketAddress();
```

这些方法返回套接字的相应属性。实际上，本书中所有返回 SocketAddress 的方法返回的都是 InetSocketAddress 实例，而 InetSocketAddress 中封装了一个 InetAddress 和一个端口号。

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，因此我们推迟到第 4.4 节才对它们进行介绍。

InetSocketAddress: 创建与访问

```
1 InetSocketAddress(InetAddress addr, int port);
2 InetSocketAddress(int port);
3 InetScoketAddress(String hostname, int port);
4
5 static InetSocketAddress createUnresolved(
6     String host, int port);
7
8 boolean isUnresolved();
```

```
9
10 InetAddress getAddres();
11 int getPort();
12 String getHostName();
13 String toString();
```

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, 而 createUnresolved() 静态方法允许在不对主机名进行解析情况下创建实例。如果在创建 InetSocketAddress 实例时没有对主机名进行解析, 或解析失败,isUnresolved() 方法将返回 true。get...() 系列方法提供了对指定属性的访问,getHostName() 方法将返回 InetSocketAddress 内部 InetAddress 所关联的主机名。toString() 方法重写了 Object 类的 toString() 方法, 返回一个包含了主机名、数字型地址 (如果已知) 和端口号的字符串。其中, 主机名与地址之间由'/'(斜线) 隔开, 地址和端口号之间由':'(冒号) 隔开。如果 InetSocketAddress 的主机名没有解析, 则冒号前只有创建实例时的主机名字符串。

2.2.2 TCP 服务端

服务器端的工作是建立一个通信终端并等待客户端的连接。典型的工作有:

1) 建立一个 ServerSocket 实例并指定到本地端口，侦听该端口收到的所有连接。

2) 重复执行以下步骤:

a) 调用 ServletSocket 的 accept() 方法获取下一个客户端的连接。基于新的客户端连接，创建一个 Socket() 实例，并由 accept() 方法返回。

b) 通过得到的 Socket 的输入输出流进行通信。

c) 通信完成，使用 Socket 的 close() 方法关闭连接。

服务器端的示例如下:

```
1 package ch02;
2
```

```
3  import java.io.IOException;
4  import java.io.InputStream;
5  import java.io.OutputStream;
6  import java.net.ServerSocket;
7  import java.net.Socket;
8  import java.net.SocketAddress;
9
10 public class TCPEchoServer {
11
12   private static final int BUFSIZE = 32; // Size of receive buffer
13
14   public static void main(String[] args) throws IOException {
15
16     if (args.length != 1) // Test for correct # of args
17       throw new IllegalArgumentException("Parameter(s): <Port>");
18
19     int servPort = Integer.parseInt(args[0]);
20
21     // Create a server socket to accept client connection requests
22     ServerSocket servSock = new ServerSocket(servPort);
23
24     int recvMsgSize; // Size of received message
25     byte[] receiveBuf = new byte[BUFSIZE]; // Receive buffer
26
27     while (true) { // Run forever, accepting and servicing connections
28       Socket clntSock = servSock.accept(); // Get client connection
29
30       SocketAddress clientAddress = clntSock.getRemoteSocketAddress();
31       System.out.println("Handling client at " + clientAddress);
32
33       InputStream in = clntSock.getInputStream();
34       OutputStream out = clntSock.getOutputStream();
35
36       // Receive until client closes connection, indicated by -1 return
```

```
37        while ((recvMsgSize = in.read(receiveBuf)) != -1) {
38          out.write(receiveBuf, 0, recvMsgSize);
39        }
40
41        clntSock.close(); // Close the socket. We are done with this client!
42      }
43      /* NOT REACHED */
44    }
45  }
```

ServerSocket 的构造函数有：

```
1  ServerSocket(int loaclPort);
2  ServerSocket(int localPort, int queueLimit);
3  ServerSocket(int localPort, int uqeueLimit, InetAddress localAddr);
4  ServerSocket();
```

注意参数中的队列长度不是一个严格限制，也不能用来控制客户端的总数。

如果指定了本地地址，这个地址就一定要是本主机的网络接口之一；如果没有指定，套接字会接受指向本主机任何 IP 地址的连接，这对于有多个接口的服务器非常有用。

没有参数的构造函数生产的实例在使用前，必须使用 bind() 方法为其绑定一个端口号。

ServerSocket 常用操作：

```
1  void bind(int prot);
2  void bind(int port, int queuelimit);
3  Socket accept();
4  void close();
```

bind() 方法为套接字关联一个本地的端口。如果该实例已经关联了一个端口，或指定的端口已经被战胜，则抛出 IOException 异常。

ServerSocket 取得属性：

```
1  InetAddress getInetAddress();
```

```
SocketAddress getLocalSocketAddress();
int getLocalPort();
```

2.2.3 输入输出流

OutputStream 操作：

```
abstract void write(int data);
void write(byte[] data);
void write(byte[] data, int offset, int length);
void flush();
void close();
```

输出一个字节的 write 方法只将整形参数的低 8 位输出。注意如果输出流上的输出在另一端还没有被 read() 方法读取时会产生阻塞。

InputStream 操作：

```
abstract int read();
int read(byte[] data);
int read(byte[] data, int offset, int length);
int available();
void close();
```

available() 方法作用是返回当前可读字节的总数。

Part III

其他扩展