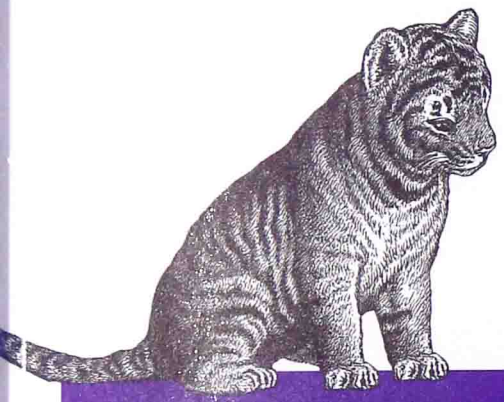
Learning Java

第4版
下册



Java学习指南

[美] *Patrick Niemeyer & Daniel Leuck* 著
李强 王建新 吴戈 译

O'REILLY®

人民邮电出版社
POSTS & TELECOM PRESS

O'REILLY®

Java学习指南（第4版）

（下册）

[美] Patrick Niemeyer
Daniel Leuck
著
李 强 王建新 吴 戈 译

人民邮电出版社
北京

封面介绍

本书的封面动物是一只孟加拉母虎和它的一群幼仔。孟加拉虎生活在南亚，但是在此遭到了大量捕杀而濒临灭绝，主要是因为虎骨具有药用价值。目前它们大多数生活在自然保护区或国家公园中。据估计现存的野生孟加拉虎不足 3000 只。

孟加拉虎通体呈红黄色并带有黑色、灰色或棕色的窄条纹，条纹通常为垂直方向。成年雄虎身长可达 9 英尺，体重约为 500 磅；它们是目前最大的猫科动物。孟加拉虎喜欢居住在稠密的灌木丛、很深的草地或者河边的灌木丛中。其最长寿命可以达到 26 岁，不过野生孟加拉虎往往只能活到 15 岁左右。

老虎通常在雨季之后受孕；怀孕 3 个半月后，大多数幼仔都于 2 月到 5 月之间出生。雌虎每两年或三年生一窝幼仔。小老虎出生时体重约为 3 磅，而且生来即带有条纹。一窝幼仔一般有 1 到 4 只，有时可多达 6 只，但是存活下来的往往只有两三只。幼虎会到 4 至 6 个月时才断奶，但是还要有两年依赖它们的母亲提供食物和保护。雌虎 3 至 4 年即长为成年虎，而雄虎则需要 4 至 5 年。

第 13 章

网络编程

网络可谓是 Java 的灵魂。Java 拥有诸多让人赞叹的新颖之处，而其中大多数均围绕于它能够用于构筑动态的网络化应用。由于 Java 的网络 API 已经成熟了，Java 也称为实现传统客户端端/服务器应用程序和服务的语言选择。在这一章中，我们将先对 java.net 包加以讨论，此包中包含了用于通信和处理网络资源的基本类（我们将在第 14 章完成这一讨论）。其后将会介绍 java.rmi 包，它提供了 Java 的本地的、高层及的高级远程方法调用工具。最后，基于第 12 章所介绍的内容，我们将完成对 java.nio 包的讨论，在实现大型服务器时 java.nio 包将极为高效。

java.net 的类可以划分为两类：套接字 API（Sockets API）和用于处理统一资源定位器（Uniform Resource Locator，URL）的工具。图 13-1 显示了 java.net 包。

Java 的套接字 API 提供了对标准网络协议的访问，这些协议可用于完成 Internet 上主机间的通信。套接字（Socket）是所有其他可移植网络通信的底层机制。套接字可谓是最低级的工具，可以将套接字用于网络上客户端和服务器之间（或对等应用之间）的各种通信，不过还必须实现自己的应用级协议来处理和解释相应的数据。还有一些更高级的网络化工具，如远程方法调用和其他分布式对象系统，它们均在套接字之上实现。

Java 远程方法调用（Remote Method Invocation，RMI）是一个功能强大的工具，它可以充分利用 Java 的对象串行化，并允许在远程机器上透明地处理对象，就如同这些对象是本地的一样。利用 RMI，可以很容易地编写分布式应用，其中客户端和服务器可以把相互的数据处理为完备的 Java 对象，而不是原始的流或数据包。

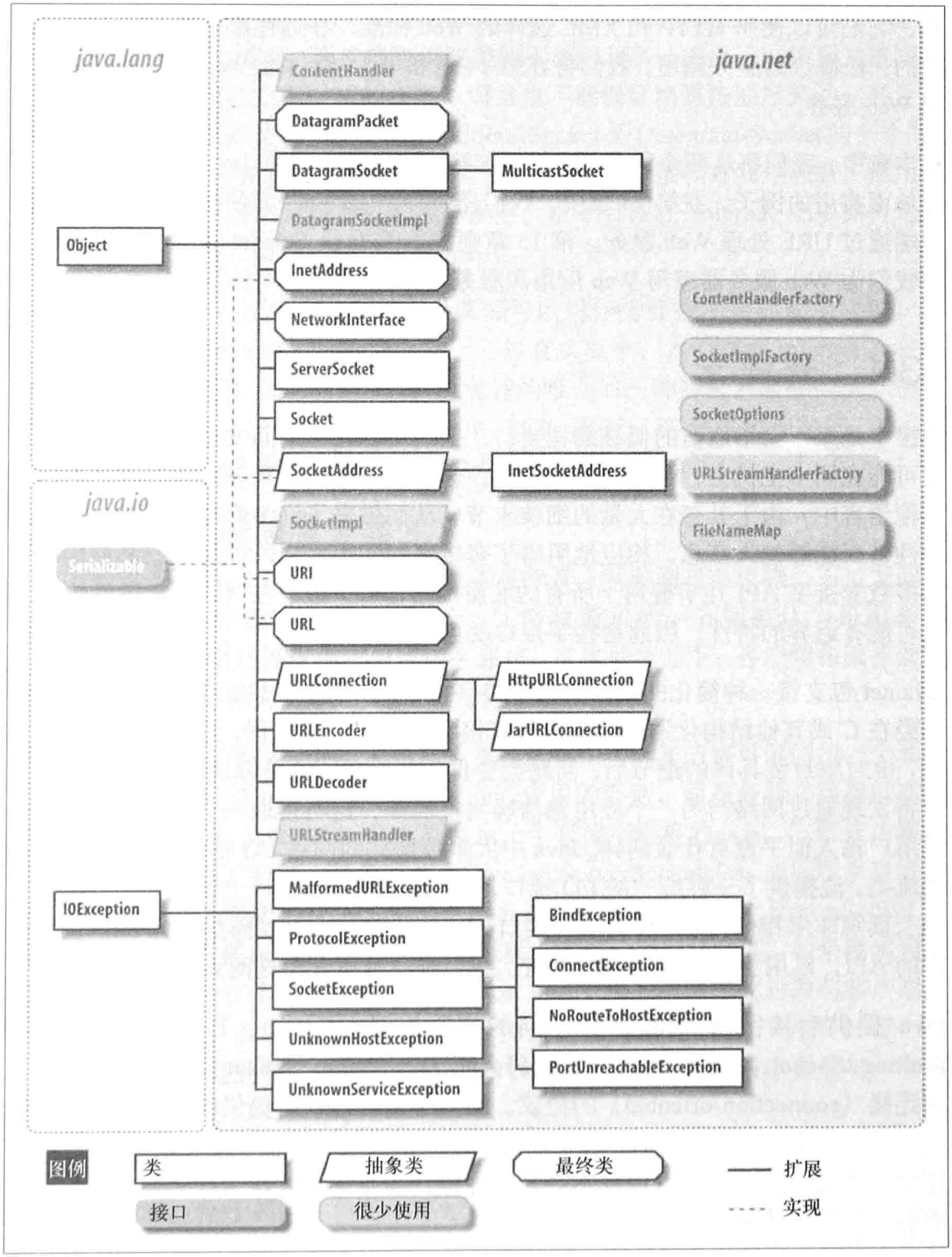


图 13-1　java.net 包,

RMI 只能在两个 Java 应用程序之间使用，相反，Web 服务则指的是较为通用的技

术，它们可以使用 HTTP 和 XML 这样的 Web 标准，在远程服务器上提供平台独立的、松耦合的服务调用。我们将在第 14 章和第 15 章讨论 Web 编程的时候，介绍 Web 服务。

在本章中，我们将从两个层次（即使用套接字和 RMI）提供 Java 网络编程的一些简单而实用的例子。在第 14 章中，我们还将介绍 java.net 包的另一半，即允许客户端通过 URL 处理 Web 服务。第 15 章则涵盖了 Java Servlet 及其工具，它们允许我们为 Web 服务器编写 Web 应用和服务。

13.1 套接字

套接字是用于网络通信的低级编程接口。它们可以在应用间发送数据流，这些应用可能在同一主机上，也可能并非如此。套接字可以溯源至 BSD UNIX，而且在其他语言中，由于其存在大量的细枝末节，从而带来了许多麻烦和困难，这一复杂性甚至能够使人窒息，相应地阻碍了套接字的使用。之所以如此，其原因在于大多数套接字 API 几乎要用于所有的底层网络协议。由于在网络上传输数据的协议可能有迥异的特性，因此套接字接口就可能极为复杂[1]。

java.net 包支持一种简化的面向对象套接字接口，从而使网络通信容易得多。如果已经在 C 或其他结构化语言中使用过套接字来完成网络编程，那么你会欣喜地发现，由对象封装具体的细节后，问题会变得何等简单。如果你是初次涉入套接字，也将发现通过网络与另一个应用通信确实很简单，这与读取一个文件或从一个终端用户输入似乎没有什么两样。Java 中大多数形式的网络 I/O 都使用第 12 章所述的流类。流提供了一种统一的 I/O 接口，从而使得在 Internet 上读或写与在本地系统上读写非常相似。除了面向流的接口，Java 网络 API 可以操作 Java NIO 面向缓存的 API，以用于高可扩展性的应用。我们将在本章介绍这两者。

Java 提供套接字来支持 3 种不同的底层协议：Socket、DatagramSocket 和 MulticastSocket。在第一节中，我们将介绍 Java 的基本 Socket 类，它使用一种面向连接（connection-oriented）的协议。面向连接的协议所提供的就等价于一个电话交谈；在建立一个连接后，两个应用可以来回发送数据流，即使没有人在谈话，连接仍然保留。此协议可保证不会丢失数据，而且无论发送了什么数据，它们都能够以发送的顺序到达接收方。

[1] 要了解对套接字的一般性讨论，请参见 Richard Stevens 所著的《Unix NetworkProgramming》（Prentice-Hall 出版）。

在下一节中，我们将分析 DatagramSocket 类，它使用的是一个无连接（connectionless）的协议。无连接协议更类似于邮政服务。应用可以相互发送短消息，但是不会提前建立端对端的连接，而且也不能保证消息按顺序到达。甚至连消息最终是否能够到达都无法保证。MulticastSocket 是 Datagram-Socket 的一个“变种”，它可以完成多路传送（也称多播），即同时向多个接收方发送数据。使用多路传送套接字非常类似于使用数据报套接字。由于目前在 Internet 上多路传送并没有得到广泛支持，因此在这里未做介绍。

再次指出，从理论上说，几乎所有协议都可以用作为套接字层的底层协议，如 Novell 的 IPX、Apple 的 AppleTalk 等。不过在实践中，人们对于 Internet 所关心的只有一种协议，而且这也是 Java 所支持的唯一的一种协议，即 Internet 协议（Internet Protocol，IP）。Socket 类采用 TCP 通信，这是面向连接的 IP，而 DatagramSocket 类则采用 UDP 通信，这是无连接的 IP。

13.1.1 客户端和服务器

在编写网络应用时，通常会谈到客户端和服务器。其差别日益模糊，不过发起通信的一方通常认为是客户端（client），而接收请求的一方则被认为是服务器（server）。如果两个对等应用使用套接字通信，在这种情况下，客户端和服务器的差别并不重要，但是为简单起见，我们仍采用以上定义。

对我们而言，客户端和服务器之间最显著的差别在于：客户端可以创建一个套接字，从而在任何时刻可以发起与一个服务器应用的会话，而服务器必须提前准备好监听到来的会话。java.net.Socket 类表示一个套接字连接的一端，它既作为客户端，又作为服务器。另外，服务器使用 java.net.ServerSocketclass 来监听来自客户端的新连接。在大多数情况下，相当于服务器的应用要创建一个 ServerSocket 对象，并等待，即在其 accept()方法调用中阻塞，直至一个连接到达。如果确实到来了一个连接，accept()方法会创建一个 Socket 对象，服务器将用此对象与客户端通信。服务器可以一次与多个客户端进行会话；在这种情况下，仍然仅有一个 ServerSocket，但是此服务器有多个 Socket 对象，每个 Socket 对象分别与一个客户端相关联，如图 13-2 所示。

在套接字层级，客户端需要两个信息来找到 Internet 上的一个服务器并与之连接，即一个主机名（用于找到主机的网络地址）以及一个端口号。端口号是一个标识符，用于区别同一主机上的多个客户端或服务器。服务器应用在等待连接时将在一个预置的端口监听。对于客户端要访问的服务，客户端需要选择为此服务所指定的端口号。如果将主机看作旅馆，而应用是旅客，那么端口就像是旅客的房间

号。如果一个人要拜访另一个人，他（她）就必须知道另一方的旅馆名和房间号。

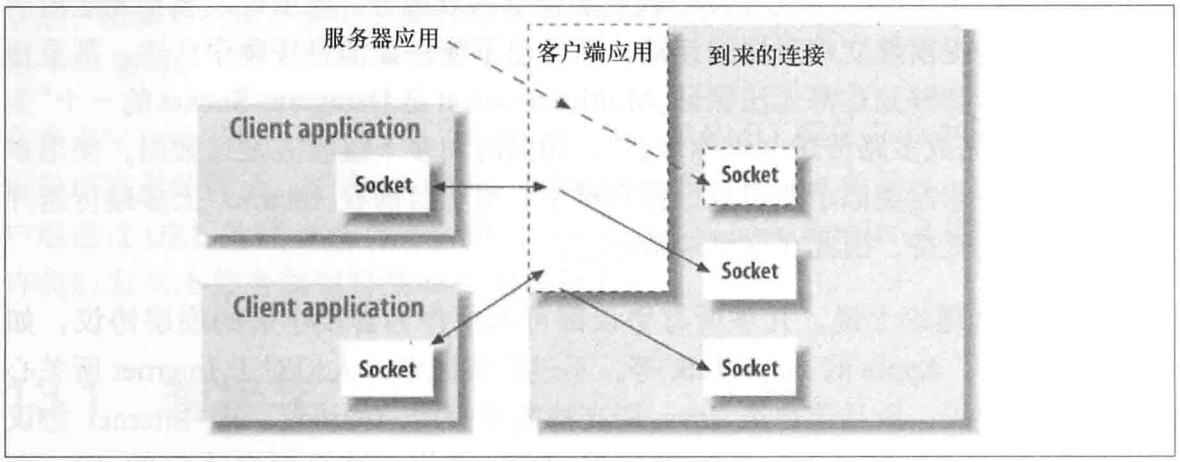


图 13-2　客户端和服务器，Socket 和 ServerSocket

客户端

客户端通过构造一个 Socket（在此要指定所需服务器的主机名和端口号）来打开与一个服务器的连接：

```
try {
    Socket sock = new Socket("wupost.wustl.edu", 25);
} catch ( UnknownHostException e ) {
    System.out.println("Can't find host.");
} catch ( IOException e ) {
    System.out.println("Error connecting to host.");
}
```

以上代码段试图与主机 wupost.wustl.edu 的端口 25（SMTP 邮件服务）建立一个 Socket 连接。客户端需要处理以下可能的情况，如无法解析主机名（UnknownHostException）以及可能无法与之连接（IOException）。构造函数也可以处理一个包含有主机 IP 地址的字符串：

```
Socket sock = new Socket("22.66.89.167", 25);
```

一旦建立了连接，输入和输出流则可使用 getInputStream()和 getOutputStream()方法得到。以下代码（有些随意）将基于流发送和接收一些数据：

```
try {
    Socket server = new Socket("foo.bar.com", 1234);
    InputStream in = server.getInputStream();
    OutputStream out = server.getOutputStream();

    // write a byte
    out.write(42);
```

```
        // write a newline or carriage return delimited string
        PrintWriter pout = new PrintWriter( out, true );
        pout.println("Hello!");

        // read a byte
        byte back = (byte)in.read();

        // read a newline or carriage return delimited string
        BufferedReader bin =
          new BufferedReader( new InputStreamReader( in ) );
        String response = bin.readLine();

        // send a serialized Java object
        ObjectOutputStream oout = new
        ObjectOutputStream( out );
        oout.writeObject( new java.util.Date() );
        oout.flush();

        server.close();
    }
    catch (IOException e ) { ... }
```

在这个例子中，客户端首先创建一个 Socket 从而与服务器通信。Socket 构造函数指定了服务器的主机名（foo.bar.com）以及一个预置的端口号（1234）。一旦建立了连接，客户端即使用 OutputStream 的 write()方法向服务器写一个字节。然后用一个 PrintWriter 包装 OutputStream，以便更容易地发送一个文本串。接下来，它将完成与之对应的操作：即使用 InputStream 的 read()方法从服务器读一个字节，再创建一个 BufferedReader，由此得到一个完整的文本串。最后，我们将完成一些真正有意思的工作，并将一个串行化 Java 对象发送至服务器，在此使用了一个 ObjectOutputStream（对于发送串行化对象，在本章后面还将深入探讨）。然后，客户端将使用 close()方法来终止连接。所有这些操作都有可能生成 IOException 异常，因此应用将在 catch 子句中处理这些异常情况。

服务器

建立了连接后，服务器应用则使用同类的 Socket 对象来完成服务器端的通信。不过，为了从客户端接受一个连接，首先必须要创建一个 ServerSocket，并绑定至正确的端口。下面再从服务器的角度来重新创建前面的会话：

```
// Meanwhile, on foo.bar.com...
try {
    ServerSocket listener = new ServerSocket( 1234 );

    while ( !finished ) {
        Socket client = listener.accept();  // wait for connection
```

```
        InputStream in = client.getInputStream();
        OutputStream out = client.getOutputStream();

        // read a byte
        byte someByte = (byte)in.read();

        // read a newline or carriage-return-delimited string
        BufferedReader bin =
          new BufferedReader( new InputStreamReader( in ) );
        String someString = bin.readLine();

        // write a byte
        out.write(43);

        // say goodbye
        PrintWriter pout = new PrintWriter( out, true );
        pout.println("Goodbye!");

        // read a serialized Java object
        ObjectInputStream oin = new ObjectInputStream( in );
        Date date = (Date)oin.readObject();

        client.close();
      }

    listener.close();
  }
  catch (IOException e ) { ... }
  catch (ClassNotFoundException e2 ) { ... }
```

首先，我们的服务器创建了一个连至端口 1234 的 ServerSocket。在某些系统中，对于应用可以使用哪些端口存在着一些规则。1024 以下的端口号通常为系统处理以及公认标准服务所保留，因此我们所选择的端口在此范围之外。ServerSocket 只创建一次；在此之后，对于到来的多个连接，都可予以接受。

接下来进入一个循环，在此等待 ServerSocket 的 accept()方法返回一个由客户端发起的活动 Socket 连接。当连接已经建立时，则完成会话的服务器端工作，其后关闭连接，并返回到循环顶部以等待另一个连接。最后，当服务器应用希望停止监听连接时，它将调用 ServerSocket 的 close()方法。

此服务器是单线程的；一次只能处理一个连接，只有在完成了当前连接之后它才会调用 accept()去监听一个新的连接。更为理想的服务器应当有一个循环来并发地接受连接，并将连接交由其自己的线程进行处理。关于实现多线程服务器，需要说明的内容还有很多，在本章后面我们将创建一个小型的 Web 服务器，它会为每个连接启动一个新的线程，另外还将创建一个稍有些复杂的 Web 服务器，此服务器将使用 NIO 包，从而仅用少量的线程即可处理多个连接。

套接字和安全性

前面的例子有一个假设，即客户端有权限与服务器连接，而且服务器允许在指定套接字上监听。如果你要编写一个一般的独立应用，那么往往可以满足这种要求。不过，不可信应用（例如，Web 浏览器中的 applet）则要在受制于一个安全策略的条件下运行，此安全策略可以施加任意的限制，指出应用能够（或不能够）与哪些主机通信，以及是否可以监听连接。

例如，对于大多数浏览器，它们对 applet 所施加的安全策略都有如下要求，即不可信 applet 只允许打开对应于其服务主机（为其提供服务的主机）的套接字连接。也就是说，它们只能与某个服务器会话，条件是其类文件正是由此服务器获得。不可信 applet 甚至不允许打开服务器套接字本身。就现在而言，这并不是说，不可信 applet 不能与其服务器协作从而与某个位置的其他实体通信。事实上，applet 的服务器可以运行一个代理，这样就能够使 applet 间接地与其希望的对象实现通信。此安全策略所要制止的是恶意 applet 在公司防火墙内“游荡”并试图与可信服务建立连接。它将安全性的工作负担均压到了始发服务器的肩头，而未交由客户机处理。将访问局限于始发服务器，这样就限制了“特洛伊木马”应用的使用，可以避免这种应用在客户端端做一些令人不快的事情（你应当不会利用代理向别人发送邮件炸弹吧，否则你会遭到斥责的）。

如果打算在默认安全管理器之下运行你的应用，应该知道默认的安全管理器不允许任何网络访问。因此，要进行网络连接，就必须修改策略文件，从而为你的代码授予适当的权限（请参见第 3 章了解更多细节）。以下策略文件段即设置了套接字权限，从而允许在任何非特权端口上向任何主机发起连接或者接受来自任何主机的连接：

```
grant {
  permission java.net.SocketPermission
    "*:1024-", "listen,accept,connect";
};
```

在启动 Java 解释器时，可以安装安全管理器，并使用以下文件（称之为 mysecurity.policy）：

```
% java -Djava.security.manager ↵

\-Djava.security.policy=mysecurity.policy MyApplication
```

13.1.2 DateAtHost 客户端

许多网络工作站都运行了一个简单的时间服务，它将在一个公认端口分配其本地时钟时间。

这可算是 NTP 的前身，即更为通用的网络时间协议（Network Time Protocol）。在下面的例子中（即 DateAtHost），我们将建立 java.util.Date 的一个特定子类，它由一个远程主机获取时间，而不是从本地时钟自行初始化（对于 Date 类的完整讨论请见第 11 章）。

DateAtHost 连接至时间服务（端口 37），并读取 4 个字节，它们表示远程主机上的时间。这 4 个字节有一个特定的规范，我们将对此解码以得到时间。以下是有关的代码：

```
//file: DateAtHost.java
import java.net.Socket;
import java.io.*;

public class DateAtHost extends java.util.Date {
    static int timePort = 37;
    // seconds from start of 20th century to Jan 1, 1970 00:00 GMT
    static final long offset = 2208988800L;

    public DateAtHost( String host ) throws IOException {
        this( host, timePort );
    }

    public DateAtHost( String host, int port ) throws IOException {
        Socket server = new Socket( host, port );
        DataInputStream din =
          new DataInputStream( server.getInputStream() );
        int time = din.readInt();
        server.close();

        setTime( (((1L << 32) + time) - offset) * 1000 );
    }
}
```

这就是全部，仅此而已。即使增加了一些“装饰”，这段代码也并不长。我们为 DateAtHost 提供了两个可能的构造函数。一般总希望使用第一个，它只需将远程主机名作为一个参数。第二个构造函数则要指定主机名和远程时间服务的端口号（如果时间服务在一个非标准端口上运行，则应使用第二个构造函数来指定可能的端口号）。第二个构造函数将完成建立连接并设置时间的工作。第一个构造函数则只是调用第二个构造函数（使用 this()构造），在此以默认端口作为参数。在此使用了一种很常见而且很有用的技术，即提供简化的构造函数，而它的工作只是基于默认参数调用其他构造函数；之所以在这里提供了两个构造函数，其原因即在于此。

第二个构造函数打开了一个对应远程主机上指定端口的套接字。它创建了一个 DataInputStream 来包装输入流，然后使用 readInt()方法读取一个 4 字节的整数。这些字节的顺序是正确的，这一点绝非偶然。Java 的 DataInputStream 和

DataOutputStream 类采用网络字节顺序（network byte order）来处理整数类型的字节，即以从最重要到最不重要的顺序加以处理。时间协议（及其他处理二进制数据的标准网络协议）也使用网络字节顺序，因此我们无需调用任何转换例程。如果使用一个非标准协议，特别是在与非 Java 客户端或服务器通信时，则可能需要显式的数据转换。在这种情况下，我们必须逐字节地读取，并且要完成一些重新排列的工作来得到 4 字节的值。在读取了字节后，套接字的工作就宣告结束，因此要将其关闭，从而终止与服务器的连接。最后，构造函数将调用 Date 的 setTime() 方法，并使用计算得到的时间值，从而对于对象余下的部分进行初始化。

时间值的 4 个字节被解释为一个整数，以表示自 20 世纪起始时刻开始以来的秒数。DateAtHost 将此转换为 Java 的绝对时间，即自 1970 年 1 月 1 日（C 和 Unix 随意选定的一个标准时间）以来的毫秒数。此转换首先要创建一个 long 值，这是与整数时间等价的无符号值。它将减去一个偏差以得到相对于起始时间（1970 年 1 月 1 日）的时间，而不是相对于世纪起始时刻的时间，然后再将其乘以 1000 从而转换为毫秒数。它再用此转换后的时间来自行初始化该对象。

DateAtHost 类可以处理由一个远程主机获取的时间，这与在本地主机上使用 Date 处理时间同样简单。唯一增加的开销在于，由于 DateAtHost 构造函数可能会抛出 IOException 异常，因此必须对此加以处理：

```
try {
    Date d = new DateAtHost( "someserver.net" );
    System.out.println( "The time over there is: " + d );
}
catch ( IOException e ) { ... }
```

这个例子获取了主机 sura.net 的时间，并打印其值。

13.1.3 TinyHttpd 服务器

你是否曾经希望编写你自己的 Web 服务器呢？倘若如此，你将非常幸运。因为在这一节中，我们将会构建一个 TinyHttpd，尽管它很小，但却是一个功能完备的 Web 服务器，正所谓“麻雀虽小，五脏俱全”。TinyHttpd 将在一个指定端口上监听，并对简单的 HTTP GET 请求提供服务。其形式如下：

```
GET /path/filename [ optional stuff ]
```

Web 浏览器对于从一个 Web 服务器获取得到的每个文档都会发出一个或多个这样的请求。在读到请求时，服务器将试图打开指定的文件，并发送其内容。如果此文档包含有一些引用，而这些引用指向图像或其他需要在线显示的项，那么浏览器将另增 GET 请求。为了实现最好的性能，TinyHttpd 将在其自己的线程中对各

个请求提供服务。因此，TinyHttpd 可以并发地为多个请求提供服务。

这个例子可以工作，但是这有些过于简化了。首先，它实现了 HTTP 协议非常旧的一个子集，因此，一些浏览器可能对其嗤之以鼻（在编写本书的时候，我在自己的 Mac 的 Safari 上测试了它，对于这个示例，它工作的很好）。要记住，在 Java 中，文件的路径名有些依赖于体系结构。这个例子在大多数系统中确实可以运行，但是若要让它在任何环境下均能工作则需要一些改进。可以编写稍微精巧一些的代码，即使用 Java 所提供的环境信息来自我调整，以适应于本地系统（第 12 章对此给出了一些提示）。

除非有一个防火墙或其他安全设施，否则下面这个例子将向外界提供你的主机上的文件，而未做任何保护。因此不要做此尝试。

现在不再赘述，以下将列出 TinyHttpd：

```
//file: TinyHttpd.java
import java.net.*;
import java.io.*;
import java.util.regex.*;
import java.util.concurrent.*;

public class TinyHttpd {
  public static void main( String argv[] ) throws IOException {
    Executor executor = Executors.newFixedThreadPool(3);
    ServerSocket ss = new ServerSocket( Integer.parseInt(argv[0]) );
    while ( true )
      executor.execute( new TinyHttpdConnection( ss.accept() ) );
  }
}

class TinyHttpdConnection implements Runnable {
  Socket client;
  TinyHttpdConnection ( Socket client ) throws SocketException {
    this.client = client;
  }
  public void run() {
    try {
      BufferedReader in = new BufferedReader(
        new InputStreamReader(client.getInputStream(), "8859_1" ) );
      OutputStream out = client.getOutputStream();
      PrintWriter pout = new PrintWriter(
        new OutputStreamWriter(out, "8859_1"), true );
      String request = in.readLine();
      System.out.println( "Request: "+request);

      Matcher get = Pattern.compile("GET /?(\\S*).*").matcher( request );
      if ( get.matches() ) {
        request = get.group(1);
        if ( request.endsWith("/") || request.equals("") )
```

```
            request = request + "index.html";
          try {
            FileInputStream fis = new FileInputStream ( request );
            byte [] data = new byte [ 64*1024 ];
            for(int read; (read = fis.read( data )) > -1; )
                out.write( data, 0, read );
            out.flush();
          } catch ( FileNotFoundException e ) {
            pout.println( "404 Object Not Found" ); }
        } else
          pout.println( "400 Bad Request" );
        client.close();
      } catch ( IOException e ) {
        System.out.println( "I/O error " + e ); }
    }
}
```

参照第 3 章所述，编译此 TinyHttpd，并将其置于类路径中。进入一个目录（其中包括一些有意思的文档），然后启动这一服务器，并将一个未用的端口号作为实参。例如：

```
% java TinyHttpd 1234
```

现在就可以使用 Web 浏览器从主机上获取文件了。必须在 URL 中指定所选择的端口号指。例如，如果主机名是 foo.bar.com，而且如上启动服务器，那么就应当采用以下形式引用一个文件：

```
http://foo.bar.com:1234/welcome.html
```

如果你在同一台机器上运行了服务器和 Web 浏览器，那么也可以如下引用：

```
http://localhost:1234/welcome.html
```

TinyHttpd 将寻找相对于其当前目录的文件，因此你提供的路径名必须相对于该位置。要获取一些文件（而不是一个文件），那么情况如何呢（不知你是否注意到，在获取一个 HTML 文件时，Web 浏览器将为文件中所包含的项（如图像等）自动生成更多的请求）？下面就来更深入地加以分析。

TinyHttpd 应用有两个类。公共的 TinyHttpd 类包含有这个独立应用的 main()方法。它先创建一个 ServerSocket，将其连至指定的端口。然后进入循环以等待客户端连接，并创建第二个类（TinyHttpdConnection）的实例，从而为各个请求提供服务。while 循环将等待 ServerSocket accept()方法为各个客户端连接返回一个新的 Socket。Socket 将作为一个参数传递从而构造一个对其加以处理的 TinyHttpdConnection 线程。我们使用一个 Executor 来服务所有的连接，它带有固定的线程池大小，即可容纳 3 个线程。

TinyHttpdConnection 是一个 Runnable 对象。对于每个连接，我们都要启动一个线程，线程的生命期足以处理一个客户端连接，然后“消亡”。所有奇妙之处均发生在 TinyHttpdConnection 的 run()方法体中。首先，我们获取一个 OutputStream 从而与客户端进行会话。第二行将从 InputStream 将 GET 请求读入变量 request。此请求是一个单行的（以换行符终止的）String，它形如前面所述的 GET 请求。为此，我们使用了一个 BufferedInputStream 来包装一个 InputStreamReader（稍后将对 InputStreamReader 做更多说明）。

接下来对请求的内容进行解析，从而抽取出一个文件名。在此我们使用了正则表达式 API（有关正则表达式和正则表达式 API 的全面介绍请见第 10 章）。此模式只是要寻找以下字符串，即“GET”后跟有一个可选的斜线，其后为非空白符组成的任意字符串。在最后，我们增加了“.*”，其目的是使此模式与整个输入匹配，从而可以使用 Matcher match()方法来检查整个请求对我们而言是否有意义。与文件名匹配的部分在一个捕获组中，即“(\\S*)”。这样我们就可以使用 Matcher group()方法来得到该文本。最后，我们还将查看所请求的文件名是否形如一个目录名（即以斜线结尾），或者是否为空。在这些情况下，要为其追加一个我们所熟知的默认文件名 index.html 以提供方便。

一旦有了文件名，就可以尝试打开所指定的文件，并使用一个大的字节数组来发送其内容。在此进入一个循环，每次对缓冲区做一次读取，并通过 OutputStream 写至客户端。如果无法解析请求，或者文件并不存在，则使用 PrintStream 来发送一条文本消息。然后返回一个标准的 HTTP 错误消息。最后要关闭套接字并从 run()返回，从而完成任务。

法国的 Web 服务器就说法语吗？

在 TinyHttpd 中，我们显式地为 BufferedReader 创建了 InputStreamReader，并为 PrintWriter 显式创建了 OutputStreamWriter。之所以如此是为了指定字符编码，从而在与 HTTP 协议消息的字节表示进行来回转换时使用（注意，我们并没有谈及将发送的文件体，对我们来说这只是一个原始字节流；在此我们要讨论的只是关于 GET 和响应消息）。如果并未指定编码，则会得到本地系统的默认字符编码。对于很多用途来说这可能是正确的，但是我们现在所讨论的是一个定义良好的国际化协议，因此应当更为特定。有关 HTTP 的 RFC 指定了 Web 客户端和服务器应当使用 ISO8859-1 字符编码。在构造 InputStreamReader 和 OutputStreamWriter 时，我们就显式地指定了这种编码。实际上，ISO8859-1 就是无格式的 ASCII，而且与 Unicode 的来回转换不会对 ASCII 值有任何影响，因此如果没有指定编码可能也不会陷入麻烦。但是至少要考虑到这种情况，这一点很重要，而且现在你

就面临这一境况。

置后台程序于掌控之中

TinyHttpd 存在一个重要的问题，即对于其提供的文件没有任何限制。只需稍微增加一些技巧，后台程序就会将文件系统中的任何文件“拱手”发送给客户端。如果施加一些限制，要求 TinyHttpd 只提供当前工作目录或一个子目录中的文件，那么将很不错，而且通常正是采用了这种做法。为此，一种简单的方法就是启动 Java 安全管理器（Java Security Manager）。一般地，对于通过网络下载的 Java 代码，安全管理器可以用于防止其做任何恶意的事情。不过，我们也可以将安全管理器用于应用中来限制对文件的访问。

可以使用一个简单的策略，例如本章前面所提供的策略；它允许服务器接受指定范围套接字上的连接。幸运的是，默认的文件访问安全策略所做的正应我们所需：它允许应用访问其当前工作目录和子目录中的文件。因此对于这种情况下所希望的文件保护，只需安装安全管理器就可以提供相应的保障（如果希望将服务器的范围扩展到其他定义良好的范畴，再增加额外的权限也将很简单）。

有了安全管理器，后台程序就无法访问当前目录及其子目录之外的内容了。如果要做此尝试，安全管理器就会抛出一个异常，并阻止访问相应文件。在这种情况下，应当由 TinyHttpd 捕获 SecurityException，并向 Web 浏览器返回一条适当的消息。应当在 FileNotFoundException 的 catch 子句后增加如下的 catch 子句。

```
...
} catch ( Security Exception e ) {
    pout.println("403 Forbidden");
}
```

改进的余地

TinyHttpd 仍有相当大的改进余地。从理论上说，它只是实现了 HTTP 协议（0.9 版）的一个过时的子集，在此服务器只接受 GET 请求，而且仅返回内容。所有现代服务器都采用 HTTP 1.0 或 1.1 通信，从而允许在 HTTP 请求和响应中有额外的元数据，另外还可接受特定的数据（如版本号、内容长度等）。HTTP 1.1 还允许在一个套接字连接上发送多个请求。

当然，真正的 Web 服务器还可以做所有其他工作。例如，正如大多数 Web 服务器所做的一样，你可能会考虑增加几行代码来读取目录，并生成链接的 HTML 链表。放手地尝试这个例子吧，你会从中得到颇多教益。