第 15 章 网络编程

历史上的网络编程都倾向于困难、复杂，而且极易出错。

程序员必须掌握与网络有关的大量细节，有时甚至要对硬件有深刻的认识。一般地，我们需要理解连网协议中不同的“层”（Layer）。而且对于每个连网库，一般都包含了数量众多的函数，分别涉及信息块的连接、打包和拆包；这些块的来回运输；以及握手等等。这是一项令人痛苦的工作。

但是，连网本身的概念并不是很难。我们想获得位于其他地方某台机器上的信息，并把它们移到这儿；或者相反。这与读写文件非常相似，只是文件存在于远程机器上，而且远程机器有权决定如何处理我们请求或者发送的数据。

Java 最出色的一个地方就是它的“无痛苦连网”概念。有关连网的基层细节已被尽可能地提取出去，并隐藏在 JVM 以及 Java 的本机安装系统里进行控制。我们使用的编程模型是一个文件的模型；事实上，网络连接（一个“套接字”）已被封装到系统对象里，所以可象对其他数据流那样采用同样的方法调用。除此以外，在我们处理另一个连网问题——同时控制多个网络连接——的时候，Java 内建的多线程机制也是十分方便的。

本章将用一系列易懂的例子解释 Java 的连网支持。

15.1 机器的标识

当然，为了分辨来自别处的一台机器，以及为了保证自己连接的是希望的那台机器，必须有一种机制能独一无二地标识出网络内的每台机器。早期网络只解决了如何在本地网络环境中为机器提供唯一的名字。但 Java 面向的是整个因特网，这要求用一种机制对来自世界各地的机器进行标识。为达到这个目的，我们采用了 IP（互联网地址）的概念。IP 以两种形式存在着：

(1) 大家最熟悉的 DNS（域名服务）形式。我自己的域名是 **bruceeckel.com。** 所以假定我在自己的域内有一台名为 Opus 的计算机，它的域名就可以是 Opus.bruceeckel.com。这正是大家向其他人发送电子函件时采用的名字，而且通常集成到一个万维网（WWW）地址里。

(2) 此外，亦可采用“四点”格式，亦即由点号（**.**）分隔的四组数字，比如 202.98.32.111。

不管哪种情况，IP 地址在内部都表达成一个由 32 个二进制位（bit）构成的数字（注释①），所以 IP 地址的每一组数字都不能超过 255。利用由 java.net 提供的 **static InetAddress.getByName()，** 我们可以让一个特定的 Java 对象表达上述任何一种形式的数字。结果是类型为 InetAddress 的一个对象，可用它构成一个“套接字”（Socket），大家在后面会见到这一点。

①：这意味着最多只能得到 40 亿左右的数字组合，全世界的人很快就会把它用光。但根据目前正在研究的新 IP 编址方案，它将采用 128 bit 的数字，这样得到的唯一性 IP 地址也许在几百年的时间里都不会用完。

作为运用 InetAddress.getByName()一个简单的例子，请考虑假设自己有一家拨号连接因特网服务提供者（ISP），那么会发生什么情况。每次拨号连接的时候，都会分配得到一个临时 IP 地址。但在连接期间，那个 IP 地址拥有与因特网上其他 IP 地址一样的有效性。如果有人按照你的 IP 地址连接你的机器，他们就有可能使用在你机器上运行的 Web 或者 FTP 服务器程序。当然这有个前提，对方必须准确地知道你目前分配到的 IP。由于每次拨号连接获得的 IP 都是随机的，怎样才能准确地掌握你的 IP 呢？

下面这个程序利用 InetAddress.getByName()来产生你的 IP 地址。为了让它运行起来，事先必须知道计算机的名字。该程序只在 Windows 95 中进行了测试，但大家可以依次进入自己的“开始”、“设置”、“控制面板”、“网络”，然后进入“标识”卡片。其中，“计算机名称”就是应在命令行输入的内容。

827 页程序

```
//: c15:WhoAmI.java
// Finds out your network address when
// you're connected to the Internet.
import java.net.*;

public class WhoAmI {
  public static void main(String[] args)
      throws Exception {
    if(args.length != 1) {
      System.err.println(
        "Usage: WhoAmI MachineName");
      System.exit(1);
    }
    InetAddress a =
      InetAddress.getByName(args[0]);
```

```
            System.out.println(a);
        }
    } ///:~
```

就我自己的情况来说，机器的名字叫作 **“Colossus”**（来自同名电影，“巨人”的意思。我在这台机器上有一个很大的硬盘）。所以一旦连通我的 ISP，就象下面这样执行程序：

java whoAmI Colossus

得到的结果象下面这个样子（当然，这个地址可能每次都是不同的）：

Colossus/202.98.41.151

假如我把这个地址告诉一位朋友，他就可以立即登录到我的个人 Web 服务器，只需指定目标地址 http://202.98.41.151 即可（当然，我此时不能断线）。有些时候，这是向其他人发送信息或者在自己的 Web 站点正式出台以前进行测试的一种方便手段。

15.1.1 服务器和客户机

网络最基本的精神就是让两台机器连接到一起，并相互“交谈”或者“沟通”。一旦两台机器都发现了对方，就可以展开一次令人愉快的双向对话。但它们怎样才能“发现”对方呢？这就象在游乐园里那样：一台机器不得不停留在一个地方，侦听其他机器说：“嘿，你在哪里呢？”

“停留在一个地方”的机器叫作“**服务器**”（Server）；到处“找人”的机器则叫作“**客户机**”（Client）或者“客户”。它们之间的区别只有在客户机试图同服务器连接的时候才显得非常明显。一旦连通，就变成了一种双向通信，谁来扮演服务器或者客户机便显得不那么重要了。

所以服务器的主要任务是侦听建立连接的请求，这是由我们创建的特定服务器对象完成的。而客户机的任务是试着与一台服务器建立连接，这是由我们创建的特定客户机对象完成的。一旦连接建好，那么无论在服务器端还是客户机端，连接只是魔术般地变成了一个 I/O 数据流对象。从这时开始，我们可以象读写一个普通的文件那样对待连接。所以一旦建好连接，我们只需象第 10 章那样使用自己熟悉的 I/O 命令即可。这正是 Java 连网最方便的一个地方。

1. 在没有网络的前提下测试程序

由于多种潜在的原因，我们可能没有一台客户机、服务器以及一个网络来测试自己做好的程序。我们也许是在一个课堂环境中进行练习，或者写出的是一个不十分可靠的网络应用，还能拿到网络上去。IP 的设计者注意到了这个问题，并建立了一个特殊的地址——**localhost**——来满足非网络环境中的测试要求。在 Java 中产生这个地址最一般的做法是：

InetAddress addr = InetAddress.getByName(null);

如果向 getByName()传递一个 null（空）值，就默认为使用 localhost。我们用 InetAddress 对特定的机器进行索引，而且必须在进行进一步的操作之前得到这个 InetAddress（互联网地址）。我们不可以操纵一个 InetAddress 的内容（但可把它打印出来，就象下一个例子要演示的那样）。创建 InetAddress 的唯一途径就是那个类的 **static（静态）成员方法** getByName()（这是最常用的）、getAllByName()或者 getLocalHost()。

为得到本地主机地址，亦可向其直接传递字串"localhost"：

InetAddress.getByName("localhost");

或者使用它的保留 IP 地址（四点形式），就象下面这样：

InetAddress.getByName("127.0.0.1");

这三种方法得到的结果是一样的。

15.1.2 端口：机器内独一无二的场所

有些时候，一个 IP 地址并不足以完整标识一个服务器。这是由于在一台物理性的机器中，往往运行着多个服务器（程序）。由 IP 表达的每台机器也包含了“端口”（Port）。我们设置一个客户机或者服务器的时候，必须选择一个无论客户机还是服务器都认可连接的端口。就象我们去拜会某人时，IP 地址是他居住的房子，而端口是他在的那个房间。

注意端口并不是机器上一个物理上存在的场所，而是一种**软件抽象**（主要是为了表述的方便）。客户程序知道如何通过机器的 IP 地址同它连接，但怎样才能同自己真正需要的那种服务连接呢（一般每个端口都运行着一种服务，一台机器可能提供了多种服务，比如 HTTP 和 FTP 等等）？端口编号在这里扮演了重要的角色，它是必需的一种二级定址措施。也就是说，我们请求一个特定的端口，便相当于请求与那个端口编号关联的服务。“报时”便是服务的一个典型例子。通常，每个服务都同一台特定服务器机器上的一个独一无二的端口编号关联在一起。客户程序必须事先知道自己要求的那项服务的运行端口号。

系统服务保留了使用端口 1 到端口 1024 的权力，所以不应让自己设计的服务占用这些以及其他任何已知正在使用的端口。本书的第一个例子将使用端口 8080（为追忆我的第一台机器使用的老式 8 位 Intel 8080 芯片，那是一部使用 CP/M 操作系统的机子）。

15.2 套接字

“套接字”或者“插座”（Socket）也是一种**软件形式的抽象**，用于表达两台机器间一个连接的“终端”。针对一个特定的连接，每台机器上都有一个“套接字”，可以想象它们之间有一条虚拟的“线缆”。线缆的每一端都插入一个“套接字”或者“插座”里。当然，机器之间的物理性硬件以及电缆连接都是完全未知的。抽象的基本宗旨是让我们尽可能不必知道那些细节。

在 Java 中，我们创建一个套接字，用它建立与其他机器的连接。从套接字得到的结果是一个 **InputStream** 以及 **OutputStream**（若使用恰当的转换器，则分别是 Reader 和 Writer），以便将连接作为一个 I/O 流对象对待。有两个基于数据流的套接字类：**ServerSocket**，服务器用它“侦听”进入的连接；以及 **Socket**，客户用它初始一次连接。一旦客户（程序）申请建立一个套接字连接，ServerSocket 就会返回（通过 **accept()**方法）一个对应的服务器端套接字，以便进行直接通信。从此时起，我们就得到了真正的“套接字－套接字”连接，可以用同样的方式对待连接的两端，因为它们本来就是相同的！此时可以利用 **getInputStream()**以及 **getOutputStream()**从每个套接字产生对应的 InputStream 和 OutputStream 对象。这些数据流必须封装到缓冲区内。可按第 10 章介绍的方法对类进行格式化，就象对待其他任何流对象那样。

对于 Java 库的命名机制，ServerSocket（服务器套接字）的使用无疑是容易产生混淆的又一个例证。大家可能认为 ServerSocket 最好叫作“ServerConnector”

（服务器连接器），或者其他什么名字，只是不要在其中安插一个“Socket”。也可能以为 ServerSocket 和 Socket 都应从一些通用的基础类继承。事实上，这两种类确实包含了几个通用的方法，但还不够资格把它们赋给一个通用的基础类。相反，ServerSocket 的主要任务是在那里耐心地等候其他机器同它连接，再返回一个实际的 Socket。这正是“ServerSocket”这个命名不恰当的地方，因为它的目标不是真的成为一个 Socket，而是在其他人同它连接的时候产生一个 Socket 对象。

然而，ServerSocket 确实会在主机上创建一个物理性的“服务器”或者侦听用的套接字。这个套接字会侦听进入的连接，然后利用 accept()方法返回一个“已建立”套接字（本地和远程端点均已定义）。容易混淆的地方是这两个套接字（侦听和已建立）都与相同的服务器套接字关联在一起。侦听套接字只能接收新的连接请求，不能接收实际的数据包。所以尽管 ServerSocket 对于编程并无太大的意义，但它确实是“物理性”的。

创建一个 ServerSocket 时，只需为其赋予一个端口编号。不必把一个 IP 地址分配它，因为它已经在自己代表的那台机器上了。但在创建一个 Socket 时，却必须同时赋予 IP 地址以及要连接的端口编号（另一方面，从 ServerSocket.accept()返回的 Socket 已经包含了所有这些信息）。

15.2.1 一个简单的服务器和客户机程序

这个例子将以最简单的方式运用套接字对服务器和客户机进行操作。服务器的全部工作就是等候建立一个连接，然后用那个连接产生的 Socket 创建一个 InputStream 以及一个 OutputStream。在这之后，它从 InputStream 读入的所有东西都会反馈给 OutputStream，直到接收到行中止（END）为止，最后关闭连接。

客户机连接与服务器的连接，然后创建一个 OutputStream。文本行通过 OutputStream 发送。客户机也会创建一个 InputStream，用它收听服务器说些什么（本例只不过是反馈回来的同样的字句）。

服务器与客户机（程序）都使用同样的端口号，而且客户机利用本地主机地址连接位于同一台机器中的服务器（程序），所以不必在一个物理性的网络里完成测试（在某些配置环境中，可能需要同真正的网络建立连接，否则程序不能工作——尽管实际并不通过那个网络通信）。

下面是服务器程序：

831-832 页程序

```
//: c15:JabberServer.java
// Very simple server that just
// echoes whatever the client sends.
import java.io.*;
import java.net.*;

public class JabberServer {
  // Choose a port outside of the range 1-1024:
  public static final int PORT = 8080;
  public static void main(String[] args)
      throws IOException {
    ServerSocket s = new ServerSocket(PORT);
```

```
      System.out.println("Started: " + s);
      try {
        // Blocks until a connection occurs:
        Socket socket = s.accept();
        try {
          System.out.println(
            "Connection accepted: "+ socket);
          BufferedReader in =
            new BufferedReader(
              new InputStreamReader(
                socket.getInputStream()));
          // Output is automatically flushed
          // by PrintWriter:
          PrintWriter out =
            new PrintWriter(
              new BufferedWriter(
                new OutputStreamWriter(
                  socket.getOutputStream())),true);
          while (true) {
            String str = in.readLine();
            if (str.equals("END")) break;
            System.out.println("Echoing: " + str);
            out.println(str);
          }
        // Always close the two sockets...
        } finally {
          System.out.println("closing...");
          socket.close();
        }
      } finally {
        s.close();
      }
    }
  } ///:~
```

可以看到，ServerSocket 需要的只是一个端口编号，不需要 IP 地址（因为它就在这台机器上运行）。调用 **accept()**时，方法会暂时陷入停顿状态（堵塞），直到某个客户尝试同它建立连接。换言之，尽管它在那里等候连接，但其他进程仍能正常运行（参考第 14 章）。建好一个连接以后，accept()就会返回一个 Socket 对象，它是那个连接的代表。

清除套接字的责任在这里得到了很艺术的处理。假如 ServerSocket 构建器失败，则程序简单地退出（注意必须保证 ServerSocket 的构建器在失败之后不会留下任何打开的网络套接字）。针对这种情况，main()会“掷”出一个 IOException 违例，所以不必使用一个 try 块。若 ServerSocket 构建器成功执行，则其他所有

方法调用都必须到一个 try-finally 代码块里寻求保护，以确保无论块以什么方式留下，ServerSocket 都能正确地关闭。

同样的道理也适用于由 accept()返回的 Socket。若 accept()失败，那么我们必须保证 Socket 不再存在或者含有任何资源，以便不必清除它们。但假若执行成功，则后续的语句必须进入一个 **try-finally** 块内，以保障在它们失败的情况下，Socket 仍能得到正确的清除。由于套接字使用了重要的非内存资源，所以在这里必须特别谨慎，必须自己动手将它们清除（Java 中没有提供“破坏器”来帮助我们做这件事情）。

无论 ServerSocket 还是由 accept()产生的 Socket 都打印到 System.out 里。这意味着它们的 toString 方法会得到自动调用。这样便产生了：

833 页中程序

```
ServerSocket[addr=0.0.0.0,PORT=0,localport=8080]
Socket[addr=127.0.0.1,PORT=1077,localport=8080]
```

大家不久就会看到它们如何与客户程序做的事情配合。

程序的下一部分看来似乎仅仅是打开文件，以便读取和写入，只是 InputStream 和 OutputStream 是从 Socket 对象创建的。利用两个“转换器”类 InputStreamReader 和 OutputStreamWriter，InputStream 和 OutputStream 对象已经分别转换成为 Java 1.1 的 Reader 和 Writer 对象。也可以直接使用 Java1.0 的 InputStream 和 OutputStream 类，但对输出来说，使用 Writer 方式具有明显的优势。这一优势是通过 **PrintWriter** 表现出来的，它有一个过载的构建器，能获取第二个参数——一个布尔值标志，指向是否在每一次 println()结束的时候自动刷新输出（但不适用于 **print()**语句）。每次写入了输出内容后（写进 out），它的缓冲区必须刷新，使信息能正式通过网络传递出去。对目前这个例子来说，刷新显得尤为重要，因为客户和服务器在采取下一步操作之前都要等待一行文本内容的到达。若刷新没有发生，那么信息不会进入网络，除非缓冲区满（溢出），这会为本例带来许多问题。

编写网络应用程序时，需要特别注意自动刷新机制的使用。每次刷新缓冲区时，必须创建和发出一个数据包（数据封）。就目前的情况来说，这正是我们所希望的，因为假如包内包含了还没有发出的文本行，服务器和客户机之间的相互“握手”就会停止。换句话说，一行的末尾就是一条消息的末尾。但在其他许多情况下，消息并不是用行分隔的，所以不如不用自动刷新机制，而用内建的缓冲区判决机制来决定何时发送一个数据包。这样一来，我们可以发出较大的数据包，而且处理进程也能加快。

注意和我们打开的几乎所有数据流一样，它们都要进行缓冲处理。本章末尾有一个练习，清楚展现了假如我们不对数据流进行缓冲，那么会得到什么样的后果（速度会变慢）。

无限 while 循环从 BufferedReader in 内读取文本行，并将信息写入 System.out，然后写入 PrintWriter.out。注意这可以是任何数据流，它们只是在表面上同网络连接。

客户程序发出包含了"END"的行后，程序会中止循环，并关闭 Socket。

下面是客户程序的源码：

834-835 页程序

```
//: c15:JabberClient.java
// Very simple client that just sends
// lines to the server and reads lines
// that the server sends.
import java.net.*;
import java.io.*;

public class JabberClient {
  public static void main(String[] args)
      throws IOException {
    // Passing null to getByName() produces the
    // special "Local Loopback" IP address, for
    // testing on one machine w/o a network:
    InetAddress addr =
      InetAddress.getByName(null);
    // Alternatively, you can use
    // the address or name:
    // InetAddress addr =
    //    InetAddress.getByName("127.0.0.1");
    // InetAddress addr =
    //    InetAddress.getByName("localhost");
    System.out.println("addr = " + addr);
    Socket socket =
      new Socket(addr, JabberServer.PORT);
    // Guard everything in a try-finally to make
    // sure that the socket is closed:
    try {
      System.out.println("socket = " + socket);
      BufferedReader in =
        new BufferedReader(
          new InputStreamReader(
            socket.getInputStream()));
      // Output is automatically flushed
      // by PrintWriter:
      PrintWriter out =
        new PrintWriter(
          new BufferedWriter(
            new OutputStreamWriter(
              socket.getOutputStream())),true);
      for(int i = 0; i < 10; i ++) {
        out.println("howdy " + i);
        String str = in.readLine();
        System.out.println(str);
      }
```

```
        out.println("END");
      } finally {
        System.out.println("closing...");
        socket.close();
      }
    }
} ///:~
```

在 main()中，大家可看到获得本地主机 IP 地址的 InetAddress 的三种途径：使用 null，使用 localhost，或者直接使用保留地址 127.0.0.1。当然，如果想通过网络同一台远程主机连接，也可以换用那台机器的 IP 地址。打印出 InetAddress addr 后（通过对 toString()方法的自动调用），结果如下：

```
localhost/127.0.0.1
```

通过向 getByName()传递一个 null，它会默认寻找 localhost，并生成特殊的保留地址 127.0.0.1。注意在名为 socket 的套接字创建时，同时使用了 InetAddress 以及端口号。打印这样的某个 Socket 对象时，为了真正理解它的含义，请记住一次独一无二的因特网连接是用下述四种数据标识的：clientHost（客户主机）、clientPortNumber（客户端口号）、serverHost（服务主机）以及 serverPortNumber（服务端口号）。服务程序启动后，会在本地主机（127.0.0.1）上建立为它分配的端口（8080）。一旦客户程序发出请求，机器上下一个可用的端口就会分配给它（这种情况下是 1077），这一行动也在与服务程序相同的机器（127.0.0.1）上进行。现在，为了使数据能在客户及服务程序之间来回传送，每一端都需要知道把数据发到哪里。所以在同一个“已知”服务程序连接的时候，客户会发出一个“返回地址”，使服务器程序知道将自己的数据发到哪儿。我们在服务器端的示范输出中可以体会到这一情况：

```
Socket[addr=127.0.0.1,port=1077,localport=8080]
```

这意味着服务器刚才已接受了来自 127.0.0.1 这台机器的端口 1077 的连接，同时监听自己的本地端口（8080）。而在客户端：

```
Socket[addr=localhost/127.0.0.1,PORT=8080,localport=1077]
```

这意味着客户已用自己的本地端口 1077 与 127.0.0.1 机器上的端口 8080 建立了 连接。

大家会注意到每次重新启动客户程序的时候，本地端口的编号都会增加。这个编号从 1025（刚好在系统保留的 1-1024 之外）开始，并会一直增加下去，除非我们重启机器。若重新启动机器，端口号仍然会从 1025 开始增值（在 Unix 机器中，一旦超过保留的套按字范围，数字就会再次从最小的可用数字开始）。

创建好 Socket 对象后，将其转换成 **BufferedReader** 和 **PrintWriter** 的过程便与在服务器中相同（同样地，两种情况下都要从一个 Socket 开始）。在这里，客户通过发出字串"howdy"，并在后面跟随一个数字，从而初始化通信。注意缓冲区必须再次刷新（这是自动发生的，通过传递给 PrintWriter 构建器的第二个参数）。若缓冲区没有刷新，那么整个会话（通信）都会被挂起，因为用于初始化的“howdy”永远不会发送出去（缓冲区不够满，不足以造成发送动作的自动进行）。从服务器返回的每一行都会写入 **System.out，** 以验证一切都在正常运转。为中止会话，需要发出一个"END"。若客户程序简单地挂起，那么服务器会“掷”出一个异常。

大家在这里可以看到我们采用了同样的措施来确保由 Socket 代表的网络资源得到正确的清除，这是用一个 **try-finally** 块实现的。

套接字建立了一个“专用”连接，它会一直持续到明确断开连接为止（专用连接也可能间接性地断开，前提是某一端或者中间的某条链路出现故障而崩溃）。这意味着参与连接的双方都被锁定在通信中，而且无论是否有数据传递，连接都会连续处于开放状态。从表面看，这似乎是一种合理的连网方式。然而，它也为网络带来了额外的开销。本章后面会介绍进行连网的另一种方式。采用那种方式，连接的建立只是暂时的。

15.3 服务多个客户

JabberServer 可以正常工作，但每次只能为一个客户程序提供服务。在典型的服务器中，我们希望同时能处理多个客户的请求。解决这个问题的关键就是多线程处理机制。而对于那些本身不支持多线程的语言，达到这个要求无疑是异常困难的。通过第 14 章的学习，大家已经知道 Java 已对多线程的处理进行了尽可能的简化。由于 Java 的线程处理方式非常直接，所以让服务器控制多名客户并不是件难事。

最基本的方法是在服务器（程序）里创建单个 ServerSocket，并调用 accept()来等候一个新连接。一旦 accept()返回，我们就取得结果获得的 Socket，并用它新建一个线程，令其只为那个特定的客户服务。然后再调用 accept()，等候下一次新的连接请求。

对于下面这段服务器代码，大家可发现它与 JabberServer.java 例子非常相似，只是为一个特定的客户提供服务的所有操作都已移入一个独立的线程类中：

837-839 页程序

```
//: c15:MultiJabberServer.java
// A server that uses multithreading
// to handle any number of clients.
import java.io.*;
import java.net.*;

class ServeOneJabber extends Thread {
  private Socket socket;
  private BufferedReader in;
  private PrintWriter out;
  public ServeOneJabber(Socket s)
      throws IOException {
    socket = s;
    in =
      new BufferedReader(
        new InputStreamReader(
          socket.getInputStream()));
    // Enable auto-flush:
    out =
      new PrintWriter(
        new BufferedWriter(
          new OutputStreamWriter(
            socket.getOutputStream())), true);
    // If any of the above calls throw an
    // exception, the caller is responsible for
    // closing the socket. Otherwise the thread
    // will close it.
    start(); // Calls run()
  }
  public void run() {
    try {
      while (true) {
        String str = in.readLine();
        if (str.equals("END")) break;
        System.out.println("Echoing: " + str);
        out.println(str);
      }
      System.out.println("closing...");
    } catch (IOException e) {
    } finally {
      try {
        socket.close();
      } catch(IOException e) {}
    }
  }
}

public class MultiJabberServer {
  static final int PORT = 8080;
  public static void main(String[] args)
      throws IOException {
    ServerSocket s = new ServerSocket(PORT);
    System.out.println("Server Started");
    try {
      while(true) {
        // Blocks until a connection occurs:
        Socket socket = s.accept();
        try {
          new ServeOneJabber(socket);
        } catch(IOException e) {
          // If it fails, close the socket,
          // otherwise the thread will close it:
          socket.close();
        }
    }
  } finally {
    s.close();
  }
  }
} ///:~
```

每次有新客户请求建立一个连接时，ServeOneJabber 线程都会取得由 accept()在 main()中生成的 Socket 对象。然后和往常一样，它创建一个 BufferedReader，并用 Socket 自动刷新 PrintWriter 对象。最后，它调用 Thread 的特殊方法 **start()**，令其进行线程的初始化，然后调用 **run()**。这里采取的操作与前例是一样的：从套接字读入某些东西，然后把它原样反馈回去，直到遇到一个特殊的"END"结束标志为止。

同样地，套接字的清除必须进行谨慎的设计。就目前这种情况来说，套接字是在 ServeOneJabber 外部创建的，所以清除工作可以"共享"。若 ServeOneJabber 构建器失败，那么只需向调用者"掷"出一个异常即可，然后由调用者负责线程的清除。但假如构建器成功，那么必须由 ServeOneJabber 对象负责线程的清除，这是在它的 run()里进行的。

请注意 MultiJabberServer 有多么简单。和以前一样，我们创建一个 ServerSocket，并调用 accept()允许一个新连接的建立。但这一次，accept()的返回值（一个套接字）将传递给用于 ServeOneJabber 的构建器，由它创建一个新线程，并对那个连接进行控制。连接中断后，线程便可简单地消失。

如果 ServerSocket 创建失败，则再一次通过 main()掷出异常。如果成功，则位于外层的 try-finally 代码块可以担保正确的清除。位于内层的 try-catch 块只负责防范 ServeOneJabber 构建器的失败；若构建器成功，则 ServeOneJabber 线程会将对应的套接字关掉。

为了证实服务器代码确实能为多名客户提供服务，下面这个程序将创建许多客户（使用线程），并同相同的服务器建立连接。每个线程的"存在时间"都是有限的。一旦到期，就留出空间以便创建一个新线程。允许创建的线程的最大数量是由 **final int maxthreads** 决定的。大家会注意到这个值非常关键，因为假如把它设得很大，线程便有可能耗尽资源，并产生不可预知的程序错误。

840-842 页程序

```
//: c15:MultiJabberClient.java
// Client that tests the MultiJabberServer
// by starting up multiple clients.
import java.net.*;
import java.io.*;

class JabberClientThread extends Thread {
  private Socket socket;
  private BufferedReader in;
  private PrintWriter out;
  private static int counter = 0;
private int id = counter++;
private static int threadcount = 0;
public static int threadCount() {
   return threadcount;
}
public JabberClientThread(InetAddress addr) {
   System.out.println("Making client " + id);
   threadcount++;
   try {
      socket =
         new Socket(addr, MultiJabberServer.PORT);
   } catch(IOException e) {
      // If the creation of the socket fails,
      // nothing needs to be cleaned up.
   }
   try {
      in =
         new BufferedReader(
            new InputStreamReader(
               socket.getInputStream()));
      // Enable auto-flush:
      out =
         new PrintWriter(
            new BufferedWriter(
               new OutputStreamWriter(
                  socket.getOutputStream())), true);
      start();
   } catch(IOException e) {
      // The socket should be closed on any
      // failures other than the socket
      // constructor:
      try {
         socket.close();
      } catch(IOException e2) {}
   }
   // Otherwise the socket will be closed by
   // the run() method of the thread.
}
public void run() {
   try {
      for(int i = 0; i < 25; i++) {
         out.println("Client " + id + ": " + i);
         String str = in.readLine();
         System.out.println(str);
```

```
          }
          out.println("END");
        } catch(IOException e) {
        } finally {
          // Always close it:
          try {
            socket.close();
          } catch(IOException e) {}
          threadcount--; // Ending this thread
        }
      }
    }

    public class MultiJabberClient {
      static final int MAX_THREADS = 40;
      public static void main(String[] args)
          throws IOException, InterruptedException {
        InetAddress addr =
          InetAddress.getByName(null);
        while(true) {
          if(JabberClientThread.threadCount()
             < MAX_THREADS)
            new JabberClientThread(addr);
          Thread.currentThread().sleep(100);
        }
      }
    } ///:~
```

JabberClientThread 构建器获取一个 InetAddress，并用它打开一个套接字。大家可能已看出了这样的一个套路：Socket 肯定用于创建某种 Reader 以及 / 或者 Writer（或者 InputStream 和 / 或 OutputStream）对象，这是运用 Socket 的唯一方式（当然，我们可考虑编写一、两个类，令其自动完成这些操作，避免大量重复的代码编写工作）。同样地，**start()**执行线程的初始化，并调用 run()。在这里，消息发送给服务器，而来自服务器的信息则在屏幕上回显出来。然而，线程的"存在时间"是有限的，最终都会结束。注意在套接字创建好以后，但在构建器完成之前，假若构建器失败，套接字会被清除。否则，为套接字调用 close()的责任便落到了 run()方法的头上。

threadcount 跟踪计算目前存在的 JabberClientThread 对象的数量。它将作为构建器的一部分增值，并在 run()退出时减值（run()退出意味着线程中止）。在 MultiJabberClient.main()中，大家可以看到线程的数量会得到检查。若数量太多，则多余的暂时不创建。方法随后进入"休眠"状态。这样一来，一旦部分线程最后被中止，多作的那些线程就可以创建了。大家可试验一下逐渐增大 MAX_THREADS，看看对于你使用的系统来说，建立多少线程（连接）才会使您的系统资源降低到危险程度。

15.4 数据报

大家迄今看到的例子使用的都是“传输控制协议”（TCP），亦称作“基于数据流的套接字”。根据该协议的设计宗旨，它具有高度的可靠性，而且能保证数据顺利抵达目的地。换言之，它允许重传那些由于各种原因半路“走失”的数据。而且收到字节的顺序与它们发出来时是一样的。当然，这种控制与可靠性需要我们付出一些代价：TCP 具有非常高的开销。

还有另一种协议，名为“用户数据报协议”（UDP），它并不刻意追求数据包会完全发送出去，也不能担保它们抵达的顺序与它们发出时一样。我们认为这是一种“不可靠协议”（TCP 当然是“可靠协议”）。听起来似乎很糟，但由于它的速度快得多，所以经常还是有用武之地的。对某些应用来说，比如声音信号的传输，如果少量数据包在半路上丢失了，那么用不着太在意，因为传输的速度显得更重要一些。大多数互联网游戏，如 Diablo，采用的也是 UDP 协议通信，因为网络通信的快慢是游戏是否流畅的决定性因素。也可以想想一台报时服务器，如果某条消息丢失了，那么也真的不必过份紧张。另外，有些应用也许能向服务器传回一条 UDP 消息，以便以后能够恢复。如果在适当的时间里没有响应，消息就会丢失。

Java 对数据报的支持与它对 TCP 套接字的支持大致相同，但也存在一个明显的区别。对数据报来说，我们在客户和服务器程序都可以放置一个 **DatagramSocket**（数据报套接字），但与 ServerSocket 不同，前者不会干巴巴地等待建立一个连接的请求。这是由于不再存在“连接”，取而代之的是一个数据报陈列出来。另一项本质的区别的是对 TCP 套接字来说，一旦我们建好了连接，便不再需要关心谁向谁“说话”——只需通过会话流来回传送数据即可。但对数据报来说，它的数据包必须知道自己来自何处，以及打算去哪里。这意味着我们必须知道每个数据报包的这些信息，否则信息就不能正常地传递。

DatagramSocket 用于收发数据包，而 **DatagramPacket** 包含了具体的信息。准备接收一个数据报时，只需提供一个缓冲区，以便安置接收到的数据。数据包抵达时，通过 DatagramSocket，作为信息起源地的因特网地址以及端口编号会自动得到初化。所以一个用于接收数据报的 DatagramPacket 构建器是：

DatagramPacket(buf, buf.length)

其中，**buf 是一个字节数组**。既然 buf 是个数组，大家可能会奇怪为什么构建器自己不能调查出数组的长度呢？实际上我也有同感，唯一能猜到的原因就是 C 风格的编程使然，那里的数组不能自己告诉我们它有多大。

可以重复使用数据报的接收代码，不必每次都建一个新的。每次用它的时候（再生），缓冲区内的数据都会被覆盖。

缓冲区的最大容量仅受限于允许的数据报包大小，这个限制位于比 64KB 稍小的地方。但在许多应用程序中，我们都宁愿它变得还要小一些，特别是在发送数据的时候。具体选择的数据包大小取决于应用程序的特定要求。

发出一个数据报时，DatagramPacket 不仅需要包含正式的数据，也要包含因特网地址以及端口号，以决定它的目的地。所以用于输出 DatagramPacket 的构建器是：

DatagramPacket(buf, length, inetAddress, port)

这一次，buf（一个字节数组）已经包含了我们想发出的数据。length 可以是

buf 的长度，但也可以更短一些，意味着我们只想发出那么多的字节。另两个参数分别代表数据包要到达的因特网地址以及目标机器的一个目标端口（注释②）。

②：我们认为 TCP 和 UDP 端口是相互独立的。也就是说，可以在端口 8080 同时运行一个 TCP 和 UDP 服务程序，两者之间不会产生冲突。

大家也许认为两个构建器创建了两个不同的对象：一个用于接收数据报，另一个用于发送它们。如果是好的面向对象的设计方案，会建议把它们创建成两个不同的类，而不是具有不同的行为的一个类（具体行为取决于我们如何构建对象）。这也许会成为一个严重的问题，但幸运的是，DatagramPacket 的使用相当简单，我们不需要在这个问题上纠缠不清。这一点在下例里将有很明确的说明。该例类似于前面针对 TCP 套接字的 MultiJabberServer 和 MultiJabberClient 例子。多个客户都会将数据报发给服务器，后者会将其反馈回最初发出消息的同样的客户。

为简化从一个 String 里创建 DatagramPacket 的工作（或者从 DatagramPacket 里创建 String），这个例子首先用到了一个工具类，名为 Dgram：

844-845 页程序

```
//: Dgram.java
// A utility class to convert back and forth
// Between Strings and DataGramPackets.
  import java.net.*;
  public class Dgram {
       public static DatagramPacket    toDatagram(
         String s, InetAddress destIA, int destPort) {
// Deprecated in Java 1.1, but it works:
byte[] buf = new byte[s.length() + 1];
s.getBytes(0, s.length(), buf, 0);
// The correct Java 1.1 approach, but it's
// Broken (it truncates the String):
// byte[] buf = s.getBytes();
return new DatagramPacket(buf, buf.length,
destIA, destPort);
}
public static String toString(DatagramPacket p){
// The Java 1.0 approach:
// return new String(p.getData(),
// 0, 0, p.getLength());
// The Java 1.1 approach:
return
new String(p.getData(), 0, p.getLength());
}
} ///:~
```

Dgram 的第一个方法采用一个 String、一个 InetAddress 以及一个端口号作为自己的参数，将 String 的内容复制到一个字节缓冲区，再将缓冲区传递进入 DatagramPacket 构建器，从而构建一个 DatagramPacket。注意缓冲区分配时的"+1"——这对防止截尾现象是非常重要的。String 的 getByte()方法属于一种特殊操作，能将一个字串包含的 char 复制进入一个字节缓冲。该方法现在已被“反对”使用；Java 1.1 有一个“更好”的办法来做这个工作，但在这里却被当作注释屏蔽掉了，因为它会截掉 String 的部分内容。所以尽管我们在 Java 1.1 下编译该程序时会得到一条“反对”消息，但它的行为仍然是正确无误的（这个错误应该在你读到这里的时候修正了）。

Dgram.toString()方法同时展示了 Java 1.0 的方法和 Java 1.1 的方法（两者是不同的，因为有一种新类型的 String 构建器）。

下面是用于数据报演示的服务器代码：

845-846 页程序

ChatterServer 创建了一个用来接收消息的 DatagramSocket（数据报套接字），而不是在我们每次准备接收一条新消息时都新建一个。这个单一的 DatagramSocket 可以重复使用。它有一个端口号，因为这属于服务器，客户必须确切知道自己把数据报发到哪个地址。尽管有一个端口号，但没有为它分配因特网地址，因为它就驻留在“这”台机器内，所以知道自己的因特网地址是什么（目前是默认的 localhost）。在无限 while 循环中，套接字被告知接收数据（receive()）。然后暂时挂起，直到一个数据报出现，再把它反馈回我们希望的接收人——DatagramPacket dp——里面。数据包（Packet）会被转换成一个字串，同时插入的还有数据包的起源因特网地址及套接字。这些信息会显示出来，然后添加一个额外的字串，指出自己已从服务器反馈回来了。

大家可能会觉得有点儿迷惑。正如大家会看到的那样，许多不同的因特网地址和端口号都可能是消息的起源地——换言之，客户程序可能驻留在任何一台机器里（就这一次演示来说，它们都驻留在 localhost 里，但每个客户使用的端口编号是不同的）。为了将一条消息送回它真正的始发客户，需要知道那个客户的因特网地址以及端口号。幸运的是，所有这些资料均已非常周到地封装到发出消息的 DatagramPacket 内部，所以我们要做的全部事情就是用 getAddress()和 getPort()把它们取出来。利用这些资料，可以构建 DatagramPacket echo——它通过与接收用的相同的套接字发送回来。除此以外，一旦套接字发出数据报，就会添加“这”台机器的因特网地址及端口信息，所以当客户接收消息时，它可以利用 getAddress()和 getPort()了解数据报来自何处。事实上，getAddress()和 getPort()唯一不能告诉我们数据报来自何处的前提是：我们创建一个待发送的数据报，并在正式发出之前调用了 getAddress()和 getPort()。到数据报正式发送的时候，这台机器的地址以及端口才会写入数据报。所以我们得到了运用数据报时一项重要的原则：不必跟踪一条消息的来源地！因为它肯定保存在数据报里。事实上，对程序来说，最可靠的做法是我们不要试图跟踪，而是无论如何都从目标数据报里提取出地址以及端口信息（就象这里做的那样）。

为测试服务器的运转是否正常，下面这程序将创建大量客户（线程），它们都会将数据报包发给服务器，并等候服务器把它们原样反馈回来。

847-849 页程序

ChatterClient 被创建成一个线程（Thread），所以可以用多个客户来“骚扰”服务器。从中可以看到，用于接收的 DatagramPacket 和用于 ChatterServer 的那个是相似的。在构建器中，创建 DatagramPacket 时没有附带任何参数（自变量），因为它不需要明确指出自己位于哪个特定编号的端口里。用于这个套接字的因特网地址将成为“这台机器”（比如 localhost），而且会自动分配端口编号，这从输出结果即可看出。同用于服务器的那个一样，这个 DatagramPacket 将同时用于发送和接收。

hostAddress 是我们想与之通信的那台机器的因特网地址。在程序中，如果需要创建一个准备传出去的 DatagramPacket，那么必须知道一个准确的因特网地址和端口号。可以肯定的是，主机必须位于一个已知的地址和端口号上，使客户能启动与主机的“会话”。

每个线程都有自己独一无二的标识号（尽管自动分配给线程的端口号是也会提供一个唯一的标识符）。在 run()中，我们创建了一个 String 消息，其中包含了线程的标识编号以及该线程准备发送的消息编号。我们用这个字串创建一个数据报，发到主机上的指定地址；端口编号则直接从 ChatterServer 内的一个常数取得。一旦消息发出，receive()就会暂时被“堵塞”起来，直到服务器回复了这条消息。与消息附在一起的所有信息使我们知道回到这个特定线程的东西正是从始发消息中投递出去的。在这个例子中，尽管是一种“不可靠”协议，但仍然能够检查数据报是否到去过了它们该去的地方（这在 localhost 和 LAN 环境中是成立的，但在非本地连接中却可能出现一些错误）。

运行该程序时，大家会发现每个线程都会结束。这意味着发送到服务器的每个数据报包都会回转，并反馈回正确的接收者。如果不是这样，一个或更多的线程就会挂起并进入“堵塞”状态，直到它们的输入被显露出来。

大家或许认为将文件从一台机器传到另一台的唯一正确方式是通过 TCP 套接字，因为它们是“可靠”的。然而，由于数据报的速度非常快，所以它才是一种更好的选择。我们只需将文件分割成多个数据报，并为每个包编号。接收机器会取得这些数据包，并重新“组装”它们；一个“标题包”会告诉机器应该接收多少个包，以及组装所需的另一些重要信息。如果一个包在半路“走丢”了，接收机器会返回一个数据报，告诉发送者重传。

15.5 一个 Web 应用

现在让我们想想如何创建一个应用，令其在真实的 Web 环境中运行，它将把 Java 的优势表现得淋漓尽致。这个应用的一部分是在 Web 服务器上运行的一个 Java 程序，另一部分则是一个“程序片”或“小应用程序”（Applet），从服务器下载至浏览器（即“客户”）。这个程序片从用户那里收集信息，并将其传回 Web 服务器上运行的应用程序。程序的任务非常简单：程序片会询问用户的 E-mail 地址，并在验证这个地址合格后（没有包含空格，而且有一个@符号），将该 E-mail 发送给 Web 服务器。服务器上运行的程序则会捕获传回的数据，检查一个包含了所有 E-mail 地址的数据文件。如果那个地址已包含在文件里，则向浏览器反馈一条消息，说明这一情况。该消息由程序片负责显示。若是一个新地址，则将

其置入列表，并通知程序片已成功添加了电子函件地址。

若采用传统方式来解决这个问题，我们要创建一个包含了文本字段及一个“提交”（Submit）按钮的 HTML 页。用户可在文本字段里键入自己喜欢的任何内容，并毫无阻碍地提交给服务器（在客户端不进行任何检查）。提交数据的同时，Web 页也会告诉服务器应对数据采取什么样的操作——知会“通用网关接口”（CGI）程序，收到这些数据后立即运行服务器。这种 CGI 程序通常是用 Perl 或 C 写的（有时也用 C++，但要求服务器支持），而且必须能控制一切可能出现的情况。它首先会检查数据，判断是否采用了正确的格式。若答案是否定的，则 CGI 程序必须创建一个 HTML 页，对遇到的问题进行描述。这个页会转交给服务器，再由服务器反馈回用户。用户看到出错提示后，必须再试一遍提交，直到通过为止。若数据正确，CGI 程序会打开数据文件，要么把电子函件地址加入文件，要么指出该地址已在数据文件里了。无论哪种情况，都必须格式化一个恰当的 HTML 页，以便服务器返回给用户。

作为 Java 程序员，上述解决问题的方法显得非常笨拙。而且很自然地，我们希望一切工作都用 Java 完成。首先，我们会用一个 Java 程序片负责客户端的数据有效性校验，避免数据在服务器和客户之间传来传去，浪费时间和带宽，同时减轻服务器额外构建 HTML 页的负担。然后跳过 Perl CGI 脚本，换成在服务器上运行一个 Java 应用。事实上，我们在这儿已完全跳过了 Web 服务器，仅仅需要从程序片到服务器上运行的 Java 应用之间建立一个连接即可。

正如大家不久就会体验到的那样，尽管看起来非常简单，但实际上有一些意想不到的问题使局面显得稍微有些复杂。用 Java 1.1 写程序片是最理想的，但实际上却经常行不通。到本书写作的时候，拥有 Java 1.1 能力的浏览器仍为数不多，而且即使这类浏览器现在非常流行，仍需考虑照顾一下那些升级缓慢的人。所以从安全的角度看，程序片代码最好只用 Java 1.0 编写。基于这一前提，我们不能用 JAR 文件来合并（压缩）程序片中的.class 文件。所以，我们应尽可能减少.class 文件的使用数量，以缩短下载时间。

好了，再来说说我用的 Web 服务器（写这个示范程序时用的就是它）。它确实支持 Java，但仅限于 Java 1.0！所以服务器应用也必须用 Java 1.0 编写。

15.5.1 服务器应用

现在讨论一下服务器应用（程序）的问题，我把它叫作 NameCollecor（名字收集器）。假如多名用户同时尝试提交他们的 E-mail 地址，那么会发生什么情况呢？若 NameCollector 使用 TCP/IP 套接字，那么必须运用早先介绍的多线程机制来实现对多个客户的并发控制。但所有这些线程都试图把数据写到同一个文件里，其中保存了所有 E-mail 地址。这便要求我们设立一种锁定机制，保证多个线程不会同时访问那个文件。一个“信号机”可在这里帮助我们达到目的，但或许还有一种更简单的方式。

如果我们换用数据报，就不必使用多线程了。用单个数据报即可“侦听”进入的所有数据报。一旦监视到有进入的消息，程序就会进行适当的处理，并将答复数据作为一个数据报传回原先发出请求的那名接收者。若数据报半路上丢失了，则用户会注意到没有答复数据传回，所以可以重新提交请求。

服务器应用收到一个数据报，并对它进行解读的时候，必须提取出其中的电子函件地址，并检查本机保存的数据文件，看看里面是否已经包含了那个地址（如果没有，则添加之）。所以我们现在遇到了一个新的问题。Java 1.0 似乎没有足够

的能力来方便地处理包含了电子函件地址的文件（Java 1.1 则不然）。但是，用 C 轻易就可以解决这个问题。因此，我们在这儿有机会学习将一个非 Java 程序同 Java 程序连接的最简便方式。程序使用的 Runtime 对象包含了一个名为 exec()的方法，它会独立机器上一个独立的程序，并返回一个 Process（进程）对象。我们可以取得一个 OutputStream，它同这个单独程序的标准输入连接在一起；并取得一个 InputStream，它则同标准输出连接到一起。要做的全部事情就是用任何语言写一个程序，只要它能从标准输入中取得自己的输入数据，并将输出结果写入标准输出即可。如果有些问题不能用 Java 简便与快速地解决（或者想利用原有代码，不想改写），就可以考虑采用这种方法。亦可使用 Java 的“固有方法”（Native Method），但那要求更多的技巧，大家可以参考一下附录 A。

1. C 程序

这个非 Java 应用是用 C 写成，因为 Java 不适合作 CGI 编程；起码启动的时间不能让人满意。它的任务是管理电子函件（E-mail）地址的一个列表。标准输入会接受一个 E-mail 地址，程序会检查列表中的名字，判断是否存在那个地址。若不存在，就将其加入，并报告操作成功。但假如名字已在列表里了，就需要指出这一点，避免重复加入。大家不必担心自己不能完全理解下列代码的含义。它仅仅是一个演示程序，告诉你如何用其他语言写一个程序，并从 Java 中调用它。在这里具体采用何种语言并不重要，只要能够从标准输入中读取数据，并能写入标准输出即可。

852-853 页程序

该程序假设 C 编译器能接受'//'样式注释（许多编译器都能，亦可换用一个 C++编译器来编译这个程序）。如果你的编译器不能接受，则简单地将那些注释删掉即可。

文件中的第一个函数检查我们作为第二个参数（指向一个 char 的指针）传递给它的名字是否已在文件中。在这儿，我们将文件作为一个 FILE 指针传递，它指向一个已打开的文件（文件是在 main()中打开的）。函数 fseek()在文件中遍历；我们在这儿用它移至文件开头。fgets()从文件 list 中读入一行内容，并将其置入缓冲区 lbuf——不会超过规定的缓冲区长度 BSIZE。所有这些工作都在一个 while 循环中进行，所以文件中的每一行都会读入。接下来，用 strchr()找到新行字符，以便将其删掉。最后，用 strcmp()比较我们传递给函数的名字与文件中的当前行。若找到一致的内容，strcmp()会返回 0。函数随后会退出，并返回一个 1，指出该名字已经在文件里了（注意这个函数找到相符内容后会立即返回，不会把时间浪费在检查列表剩余内容的上面）。如果找遍列表都没有发现相符的内容，则函数返回 0。

在 main()中，我们用 fopen()打开文件。第一个参数是文件名，第二个是打开文件的方式；a+表示“追加”，以及“打开”（或“创建”，假若文件尚不存在），以便到文件的末尾进行更新。fopen()函数返回的是一个 FILE 指针；若为 0，表示打开操作失败。此时需要用 perror()打印一条出错提示消息，并用 exit()中止程序运行。

如果文件成功打开，程序就会进入一个无限循环。调用 gets(buf)的函数会从标准输入中取出一行（记住标准输入会与 Java 程序连接到一起），并将其置入缓

冲区 buf 中。缓冲区的内容随后会简单地传递给 alreadyInList()函数，如内容已在列表中，printf()就会将那条消息发给标准输出（Java 程序正在监视它）。fflush()用于对输出缓冲区进行刷新。

如果名字不在列表中，就用 fseek()移到列表末尾，并用 fprintf()将名字“打印”到列表末尾。随后，用 printf()指出名字已成功加入列表（同样需要刷新标准输出），无限循环返回，继续等候一个新名字的进入。

记住一般不能先在自己的计算机上编译此程序，再把编译好的内容上载到 Web 服务器，因为那台机器使用的可能是不同类的处理器和操作系统。例如，我的 Web 服务器安装的是 Intel 的 CPU，但操作系统是 Linux，所以必须先下载源码，再用远程命令（通过 telnet）指挥 Linux 自带的 C 编译器，令其在服务器端编译好程序。

2. Java 程序

这个程序先启动上述的 C 程序，再建立必要的连接，以便同它“交谈”。随后，它创建一个数据报套接字，用它“监视”或者“侦听”来自程序片的数据报包。

854-956 页程序

NameCollector 中的第一个定义应该是大家所熟悉的：选定端口，创建一个数据报包，然后创建指向一个 DatagramSocket 的句柄。接下来的三个定义负责与 C 程序的连接：一个 Process 对象是 C 程序由 Java 程序启动之后返回的，而且那个 Process 对象产生了 InputStream 和 OutputStream，分别代表 C 程序的标准输出和标准输入。和 Java IO 一样，它们理所当然地需要“封装”起来，所以我们最后得到的是一个 PrintStream 和 DataInputStream。

这个程序的所有工作都是在构建器内进行的。为启动 C 程序，需要取得当前的 Runtime 对象。我们用它调用 exec()，再由后者返回 Process 对象。在 Process 对象中，大家可看到通过一简单的调用即可生成数据流：getOutputStream()和 getInputStream()。从这个时候开始，我们需要考虑的全部事情就是将数据传给数据流 nameList，并从 addResult 中取得结果。

和往常一样，我们将 DatagramSocket 同一个端口连接到一起。在无限 while 循环中，程序会调用 receive()——除非一个数据报到来，否则 receive()会一起处于“堵塞”状态。数据报出现以后，它的内容会提取到 String rcvd 里。我们首先将该字串两头的空格剔除（trim），再将其发给 C 程序。如下所示：

```
nameList.println(rcvd.trim());
```

之所以能这样编码，是因为 Java 的 exec()允许我们访问任何可执行模块，只要它能从标准输入中读，并能向标准输出中写。还有另一些方式可与非 Java 代码“交谈”，这将在附录 A 中讨论。

从 C 程序中捕获结果就显得稍微麻烦一些。我们必须调用 read()，并提供一个缓冲区，以便保存结果。read()的返回值是来自 C 程序的字节数。若这个值为-1，意味着某个地方出现了问题。否则，我们就将 resultBuf（结果缓冲区）转换成一个字串，然后同样清除多余的空格。随后，这个字串会象往常一样进入一个 DatagramPacket，并传回当初发出请求的那个同样的地址。注意发送方的地址也是我们接收到的 DatagramPacket 的一部分。

记住尽管 C 程序必须在 Web 服务器上编译，但 Java 程序的编译场所可以是任意的。这是由于不管使用的是什么硬件平台和操作系统，编译得到的字节码都是一样的。就就是 Java 的“跨平台”兼容能力。

15.5.2 NameSender 程序片

正如早先指出的那样，程序片必须用 Java 1.0 编写，使其能与绝大多数的浏览器适应。也正是由于这个原因，我们产生的类数量应尽可能地少。所以我们在这儿不考虑使用前面设计好的 Dgram 类，而将数据报的所有维护工作都转到代码行中进行。此外，程序片要用一个线程监视由服务器传回的响应信息，而非实现 Runnable 接口，用集成到程序片的一个独立线程来做这件事情。当然，这样做对代码的可读性不利，但却能产生一个单类（以及单个服务器请求）程序片：

858-860 页程序

程序片的 UI（用户界面）非常简单。它包含了一个 TestField（文本字段），以便我们键入一个电子函件地址；以及一个 Button（按钮），用于将地址发给服务器。两个 Label（标签）用于向用户报告状态信息。

到现在为止，大家已能判断出 DatagramSocket、InetAddress、缓冲区以及 DatagramPacket 都属于网络连接中比较麻烦的部分。最后，大家可看到 run()方法实现了线程部分，使程序片能够“侦听”由服务器传回的响应信息。

init()方法用大家熟悉的布局工具设置 GUI，然后创建 DatagramSocket，它将同时用于数据报的收发。

action()方法只负责监视我们是否按下了“发送”（send）按钮。记住，我们已被限制在 Java 1.0 上面，所以不能再用较灵活的内部类了。按钮按下以后，采取的第一项行动便是检查线程 pl，看看它是否为 null（空）。如果不为 null，表明有一个活动线程正在运行。消息首次发出时，会启动一个新线程，用它监视来自服务器的回应。所以假若有个线程正在运行，就意味着这并非用户第一次发送消息。pl 句柄被设为 null，同时中止原来的监视者（这是最合理的一种做法，因为 stop()已被 Java 1.2“反对”，这在前一章已解释过了）。

无论这是否按钮被第一次按下，I2 中的文字都会清除。

下一组语句将检查 E-mail 名字是否合格。String.indexOf()方法的作用是搜索其中的非法字符。如果找到一个，就把情况报告给用户。注意进行所有这些工作时，都不必涉及网络通信，所以速度非常快，而且不会影响带宽和服务器的性能。

名字校验通过以后，它会打包到一个数据报里，然后采用与前面那个数据报示例一样的方式发到主机地址和端口编号。第一个标签会发生变化，指出已成功发送出去。而且按钮上的文字也会改变，变成“重发”（resend）。这时会启动线程，第二个标签则会告诉我们程序片正在等候来自服务器的回应。

线程的 run()方法会利用 NameSender 中包含的 DatagramSocket 来接收数据（receive()），除非出现来自服务器的数据报包，否则 receive()会暂时处于“堵塞”或者“暂停”状态。结果得到的数据包会放进 NameSender 的 DatagramPacketdp 中。数据会从包中提取出来，并置入 NameSender 的第二个标签。随后，线程的执行将中断，成为一个“死”线程。若某段时间里没有收到来自服务器的回应，用户可能变得不耐烦，再次按下按钮。这样做会中断当前线程（数据发出以后，会再建一个新的）。由于用一个线程来监视回应数据，所以用户在监视期间仍然

可以自由使用 UI。

1. Web 页

当然，程序片必须放到一个 Web 页里。下面列出完整的 Web 页源码；稍微研究一下就可看出，我用它从自己开办的邮寄列表（Mailling List）里自动收集名字。

程序片标记（<applet>）的使用非常简单，和第 13 章展示的那一个并没有什么区别。

15.5.3 要注意的问题

前面采取的似乎是一种完美的方法。没有 CGI 编程，所以在服务器启动一个 CGI 程序时不会出现延迟。数据报方式似乎能产生非常快的响应。此外，一旦 Java 1.1 得到绝大多数人的采纳，服务器端的那一部分就可完全用 Java 编写（尽管利用标准输入和输出同一个非 Java 程序连接也非常容易）。

但必须注意到一些问题。其中一个特别容易忽略：由于 Java 应用在服务器上是连续运行的，而且会把大多数时间花在 Datagram.receive()方法的等候上面，这样便为 CPU 带来了额外的开销。至少，我在自己的服务器上便发现了这个问题。另一方面，那个服务器上不会发生其他更多的事情。而且假如我们使用一个任务更为繁重的服务器，启动程序用“nice”（一个 Unix 程序，用于防止进程贪吃 CPU 资源）或其他等价程序即可解决问题。在许多情况下，都有必要留意象这样的一些应用——一个堵塞的 receive()完全可能造成 CPU 的瘫痪。

第二个问题涉及防火墙。可将防火墙理解成自己的本地网与因特网之间的一道墙（实际是一个专用机器或防火墙软件）。它监视进出因特网的所有通信，确保这些通信不违背预设的规则。

防火墙显得多少有些保守，要求严格遵守所有规则。假如没有遵守，它们会无情地把它们拒之门外。例如，假设我们位于防火墙后面的一个网络中，开始用 Web 浏览器同因特网连接，防火墙要求所有传输都用可以接受的 http 端口同服务器连接，这个端口是 80。现在来了这个 Java 程序片 NameSender，它试图将一个数据报传到端口 8080，这是为了越过“受保护”的端口范围 0-1024 而设置的。防火墙很自然地把它想象成最坏的情况——有人使用病毒或者非法扫描端口——根本不允许传输的继续进行。

只要我们的客户建立的是与因特网的原始连接（比如通过典型的 ISP 接驳 Internet），就不会出现此类防火墙问题。但也可能有一些重要的客户隐藏在防火墙后，他们便不能使用我们设计的程序。

在学过有关 Java 的这么多东西以后，这是一件使人相当沮丧的事情，因为看来必须放弃在服务器上使用 Java，改为学习如何编写 C 或 Perl 脚本程序。但请大家不要绝望。

一个出色方案是由 Sun 公司提出的。如一切按计划进行，Web 服务器最终都装备“小服务程序”或者“服务程序片”（Servlet）。它们负责接收来自客户的请求（经过防火墙允许的 80 端口）。而且不再是启动一个 CGI 程序，它们会启动小服务程序。根据 Sun 的设想，这些小服务程序都是用 Java 编写的，而且只能在服务器上运行。运行这种小程序的服务器会自动启动它们，令其对客户的请求进行处理。这意味着我们的所有程序都可以用 Java 写成（100%纯咖啡）。这显然是一种非常吸引人的想法：一旦习惯了 Java，就不必换用其他语言在服务器上处

理客户请求。

由于只能在服务器上控制请求，所以小服务程序 API 没有提供 GUI 功能。这对 NameCollector.java 来说非常适合，它本来就不需要任何图形界面。

在本书写作时，java.sun.com 已提供了一个非常廉价的小服务程序专用服务器。Sun 鼓励其他 Web 服务器开发者为他们的服务器软件产品加入对小服务程序的支持。

15.6 Java 与 CGI 的沟通

Java 程序可向一个服务器发出一个 CGI 请求，这与 HTML 表单页没什么两样。而且和 HTML 页一样，这个请求既可以设为 GET（下载），亦可设为 POST（上传）。除此以外，Java 程序还可拦截 CGI 程序的输出，所以不必依赖程序来格式化一个新页，也不必在出错的时候强迫用户从一个页回转到另一个页。事实上，程序的外观可以做得跟以前的版本别无二致。

代码也要简单一些，毕竟用 CGI 也不是很难就能写出来（前提是真正地理解它）。所以在这一节里，我们准备办个 CGI 编程速成班。为解决常规问题，将用 C++创建一些 CGI 工具，以便我们编写一个能解决所有问题的 CGI 程序。这样做的好处是移植能力特别强——即将看到的例子能在支持 CGI 的任何系统上运行，而且不存在防火墙的问题。

这个例子也阐示了如何在程序片（Applet）和 CGI 程序之间建立连接，以便将其方便地改编到自己的项目中。

15.6.1 CGI 数据的编码

在这个版本中，我们将收集名字和电子函件地址，并用下述形式将其保存到文件中：

First Last <email@domain.com>;

这对任何 E-mail 程序来说都是一种非常方便的格式。由于只需收集两个字段，而且 CGI 为字段中的编码采用了一种特殊的格式，所以这里没有简便的方法。如果自己动手编制一个原始的 HTML 页，并加入下述代码行，即可正确地理解这一点：

865 页程序

上述代码创建了两个数据输入字段（区），名为 name 和 email。另外还有一个 submit（提交）按钮，用于收集数据，并将其发给 CGI 程序。Listmgr2.exe 是驻留在特殊程序目录中的一个可执行文件。在我们的 Web 服务器上，该目录一般都叫作“cgi-bin”（注释③）。如果在那个目录里找不到该程序，结果就无法出现。填好这个表单，然后按下提交按钮，即可在浏览器的 URL 地址窗口里看到象下面这样的内容：

http://www.myhome.com/cgi-bin/Listmgr2.exe?name=First+Last&email=email@domain.com&submit=Submit

③：在 Windows32 平台下，可利用与 Microsoft Office 97 或其他产品配套提供的 Microsoft Personal Web Server（微软个人 Web 服务器）进行测试。这是进行

试验的最好方法，因为不必正式连入网络，可在本地环境中完成测试（速度也非常快）。如果使用的是不同的平台，或者没有 Office 97 或者 FrontPage 98 那样的产品，可到网上找一个免费的 Web 服务器供自己测试。

当然，上述 URL 实际显示时是不会拆行的。从中可稍微看出如何对数据编码并传给 CGI。至少有一件事情能够肯定——空格是不允许的（因为它通常用于分隔命令行参数）。所有必需的空格都用“+”号替代，每个字段都包含了字段名（具体由 HTML 页决定），后面跟随一个“=”号以及正式的字段数据，最后用一个“&”结束。

到这时，大家也许会对“+”，“=”以及“&”的使用产生疑惑。假如必须在字段里使用这些字符，那么该如何声明呢？例如，我们可能使用“John & MarshaSmith”这个名字，其中的“&”代表“And”。事实上，它会编码成下面这个样子：

John+%26+Marsha+Smith

也就是说，特殊字符会转换成一个“%”，并在后面跟上它的十六进制 ASCII 编码。

幸运的是，Java 有一个工具来帮助我们进行这种编码。这是 **URLEncoder** 类的一个静态方法，名为 encode()。可用下述程序来试验这个方法：

866 页程序

该程序将获取一些命令行参数，把它们合并成一个由多个词构成的字串，各词之间用空格分隔（最后一个空格用 String.trim()剔除了）。随后对它们进行编码，并打印出来。

为调用一个 CGI 程序，程序片要做的全部事情就是从自己的字段或其他地方收集数据，将所有数据都编码成正确的 URL 样式，然后汇编到单独一个字串里。每个字段名后面都加上一个“=”符号，紧跟正式数据，再紧跟一个“&”。为构建完整的 CGI 命令，我们将这个字串置于 CGI 程序的 URL 以及一个“?”后。这是调用所有 CGI 程序的标准方法。大家马上就会看到，用一个程序片能够很轻松地完成所有这些编码与合并。

15.6.2 程序片

程序片实际要比 NameSender.java 简单一些。这部分是由于很容易即可发出一个 GET 请求。此外，也不必等候回复信息。现在有两个字段，而非一个，但大家会发现许多程序片都是熟悉的，请比较 NameSender.java。

867-869 页程序

CGI 程序（不久即可看到）的名字是 Listmgr2.exe。许多 Web 服务器都在 Unix 机器上运行（Linux 也越来越受到青睐）。根据传统，它们一般不为自己的可执行程序采用.exe 扩展名。但在 Unix 操作系统中，可以把自己的程序称呼为自己希望的任何东西。若使用的是.exe 扩展名，程序毋需任何修改即可通过 Unix 和 Win32 的运行测试。

和往常一样，程序片设置了自己的用户界面（这次是两个输入字段，不是一

个）。唯一显著的区别是在 action()方法内产生的。该方法的作用是对按钮按下事件进行控制。名字检查过以后，大家会发现下述代码行：

869-870 页程序

name 和 email 数据都是它们对应的文字框里提取出来，而且两端多余的空格都用 trim()剔去了。为了进入列表，email 名字被强制换成小写形式，以便能够准确地对比（防止基于大小写形式的错误判断）。来自每个字段的数据都编码为 URL 形式，随后采用与 HTML 页中一样的方式汇编 GET 字串（这样一来，我们可将 Java 程序片与现有的任何 CGI 程序结合使用，以满足常规的 HTML GET 请求）。

到这时，一些 Java 的魔力已经开始发挥作用了：如果想同任何 URL 连接，只需创建一个 URL 对象，并将地址传递给构建器即可。构建器会负责建立同服务器的连接（对 Web 服务器来说，所有连接行动都是根据作为 URL 使用的字串来判断的）。就目前这种情况来说，URL 指向的是当前 Web 站点的 cgi-bin 目录（当前 Web 站点的基础地址是用 getDocumentBase()设定的）。一旦 Web 服务器在 URL 中看到了一个“cgi-bin”，会接着希望在它后面跟随了 cgi-bin 目录内的某个程序的名字，那是我们要运行的目标程序。程序名后面是一个问号以及 CGI 程序会在 QUERY_STRING 环境变量中查找的一个参数字串（马上就要学到）。

我们发出任何形式的请求后，一般都会得到一个回应的 HTML 页。但若使用 Java 的 URL 对象，我们可以拦截自 CGI 程序传回的任何东西，只需从 URL 对象里取得一个 InputStream（输入数据流）即可。这是用 URL 对象的 openStream()方法实现，它要封装到一个 DataInputStream 里。随后就可以读取数据行，若 readLine()返回一个 null（空值），就表明 CGI 程序已结束了它的输出。

我们即将看到的 CGI 程序返回的仅仅是一行，它是用于标志成功与否（以及失败的具体原因）的一个字串。这一行会被捕获并置放第二个 Label 字段里，使用户看到具体发生了什么事情。

1. 从程序片里显示一个 Web 页

程序亦可将 CGI 程序的结果作为一个 Web 页显示出来，就象它们在普通 HTML 模式中运行那样。可用下述代码做到这一点：

getAppletContext().showDocument(u);

其中，u 代表 URL 对象。这是将我们重新定向于另一个 Web 页的一个简单例子。那个页凑巧是一个 CGI 程序的输出，但可以非常方便地进入一个原始的 HTML 页，所以可以构建这个程序片，令其产生一个由密码保护的网关，通过它进入自己 Web 站点的特殊部分：

871-872 页程序

URL 类的最大的特点就是有效地保护了我们的安全。可以同一个 Web 服务器建立连接，毋需知道幕后的任何东西。

15.6.3 用 C++写的 CGI 程序

经过前面的学习，大家应该能够根据例子用 ANSI C 为自己的服务器写出

CGI 程序。之所以选用 ANSI C，是因为它几乎随处可见，是最流行的 C 语言标准。当然，现在的 C++也非常流行了，特别是采用 GNU C++编译器（g++）形式的那一些（注释④）。可从网上许多地方免费下载 g++，而且可选用几乎所有平台的版本（通常与 Linux 那样的操作系统配套提供，且已预先安装好）。正如大家即将看到的那样，从 CGI 程序可获得面向对象程序设计的许多好处。

④：GNU 的全称是“Gnu's Not Unix”。这最早是由“自由软件基金会”（FSF）负责开发的一个项目，致力于用一个免费的版本取代原有的 Unix 操作系统。现在的 Linux 似乎正在做前人没有做到的事情。但 GNU 工具在 Linux 的开发中扮演了至关重要的角色。事实上，Linux 的整套软件包附带了数量非常多的 GNU 组件。

为避免第一次就提出过多的新概念，这个程序并未打算成为一个“纯”C++程序；有些代码是用普通 C 写成的——尽管还可选用 C++的一些替用形式。但这并不是个突出的问题，因为该程序用 C++制作最大的好处就是能够创建类。在解析 CGI 信息的时候，由于我们最关心的是字段的“名称／值”对，所以要用一个类（Pair）来代表单个名称／值对；另一个类（CGI_vector）则将 CGI 字串自动解析到它会容纳的 Pair 对象里（作为一个 vector），这样即可在有空的时候把每个 Pair（对）都取出来。

这个程序同时也非常有趣，因为它演示了 C++与 Java 相比的许多优缺点。大家会看到一些相似的东西；比如 class 关键字。访问控制使用的是完全相同的关键字 public 和 private，但用法却有所不同。它们控制的是一个块，而非单个方法或字段（也就是说，如果指定 private:，后续的每个定义都具有 private 属性，直到我们再指定 public:为止）。另外在创建一个类的时候，所有定义都自动默认为 private。

在这儿使用 C++的一个原因是要利用 C++“标准模板库”（STL）提供的便利。至少，STL 包含了一个 vector 类。这是一个 C++模板，可在编译期间进行配置，令其只容纳一种特定类型的对象（这里是 Pair 对象）。和 Java 的 Vector 不同，如果我们试图将除 Pair 对象之外的任何东西置入 vector，C++的 vector 模板都会造成一个编译期错误；而 Java 的 Vector 能够照单全收。而且从 vector 里取出什么东西的时候，它会自动成为一个 Pair 对象，毋需进行造型处理。所以检查在编译期进行，这使程序显得更为“健壮”。此外，程序的运行速度也可以加快，因为没有必要进行运行期间的造型。vector 也会过载 operator[]，所以可以利用非常方便的语法来提取 Pair 对象。vector 模板将在 CGI_vector 创建时使用；在那时，大家就可以体会到如此简短的一个定义居然蕴藏有那么巨大的能量。

若提到缺点，就一定不要忘记 Pair 在下列代码中定义时的复杂程度。与我们在 Java 代码中看到的相比，Pair 的方法定义要多得多。这是由于 C++的程序员必须提前知道如何用副本构建器控制复制过程，而且要用过载的 operator=完成赋值。正如第 12 章解释的那样，我们有时也要在 Java 中考虑同样的事情。但在 C++中，几乎一刻都不能放松对这些问题的关注。

这个项目首先创建一个可以重复使用的部分，由 C++头文件中的 Pair 和 CGI_vector 构成。从技术角度看，确实不应把这些东西都塞到一个头文件里。但就目前的例子来说，这样做不会造成任何方面的损害，而且更具有 Java 风格，所以大家阅读理解代码时要显得轻松一些：

873-877 页程序

在#include 语句后，可看到有一行是：

using namespace std;

C++中的“命名空间”（Namespace）解决了由 Java 的 package 负责的一个问题：将库名隐藏起来。std 命名空间引用的是标准 C++库，而 vector 就在这个库中，所以这一行是必需的。

Pair 类表面看异常简单，只是容纳了两个（private）字符指针而已——一个用于名字，另一个用于值。默认构建器将这两个指针简单地设为零。这是由于在 C++中，对象的内存不会自动置零。第二个构建器调用方法 decodeURLString()，在新分配的堆内存中生成一个解码过后的字串。这个内存区域必须由对象负责管理及清除，这与“破坏器”中见到的相同。name()和 value()方法为相关的字段产生只读指针。利用 empty()方法，我们查询 Pair 对象它的某个字段是否为空；返回的结果是一个 bool——C++内建的基本布尔数据类型。operator bool()使用的是 C++“运算符过载”的一种特殊形式。它允许我们控制自动类型转换。如果有一个名为 p 的 Pair 对象，而且在一个本来希望是布尔结果的表达式中使用，比如 if(p){//...，那么编译器能辨别出它有一个 Pair，而且需要的是个布尔值，所以自动调用 operator bool()，进行必要的转换。

接下来的三个方法属于常规编码，在 C++中创建类时必须用到它们。根据 C++类采用的所谓“经典形式”，我们必须定义必要的“原始”构建器，以及一个副本构建器和赋值运算符——operator=（以及破坏器，用于清除内存）。之所以要作这样的定义，是由于编译器会“默默”地调用它们。在对象传入、传出一个函数的时候，需要调用副本构建器；而在分配对象时，需要调用赋值运算符。只有真正掌握了副本构建器和赋值运算符的工作原理，才能在 C++里写出真正“健壮”的类，但这需要需要一个比较艰苦的过程（注释⑤）。

⑤：我的《Thinking in C++》（Prentice-Hall,1995）用了一整章的地方来讨论这个主题。若需更多的帮助，请务必看看那一章。

只要将一个对象按值传入或传出函数，就会自动调用副本构建器 Pair(const Pair&)。也就是说，对于准备为其制作一个完整副本的那个对象，我们不准备在函数框架中传递它的地址。这并不是 Java 提供的一个选项，由于我们只能传递句柄，所以在 Java 里没有所谓的副本构建器（如果想制作一个本地副本，可以“克隆”那个对象——使用 clone()，参见第 12 章）。类似地，如果在 Java 里分配一个句柄，它会简单地复制。但 C++中的赋值意味着整个对象都会复制。在副本构建器中，我们创建新的存储空间，并复制原始数据。但对于赋值运算符，我们必须在分配新存储空间之前释放老存储空间。我们要见到的也许是 C++类最复杂的一种情况，但那正是 Java 的支持者们论证 Java 比 C++简单得多的有力证据。在 Java 中，我们可以自由传递句柄，善后工作则由垃圾收集器负责，所以可以轻松许多。

但事情并没有完。Pair 类为 nm 和 val 使用的是 char*，最复杂的情况主要是围绕指针展开的。如果用较时髦的 C++ string 类来代替 char*，事情就要变得简单得多（当然，并不是所有编译器都提供了对 string 的支持）。那么，Pair 的第

一部分看起来就象下面这样：

878 页程序

（此外，对这个类 decodeURLString()会返回一个 string，而不是一个 char*）。我们不必定义副本构建器、operator=或者破坏器，因为编译器已帮我们做了，而且做得非常好。但即使有些事情是自动进行的，C++程序员也必须了解副本构建以及赋值的细节。

Pair 类剩下的部分由两个方法构成：decodeURLString()以及一个“帮助器”方法 translateHex()——将由 decodeURLString()使用。注意 translateHex()并不能防范用户的恶意输入，比如“%1H”。分配好足够的存储空间后（必须由破坏器释放），decodeURLString()就会其中遍历，将所有“+”都换成一个空格；将所有十六进制代码（以一个“%”打头）换成对应的字符。

CGI_vector 用于解析和容纳整个 CGI GET 命令。它是从 STL vector 里继承的，后者例示为容纳 Pair。C++中的继承是用一个冒号表示，在 Java 中则要用 extends。此外，继承默认为 private 属性，所以几乎肯定需要用到 public 关键字，就象这样做的那样。大家也会发现 CGI_vector 有一个副本构建器以及一个 operator=，但它们都声明成 private。这样做是为了防止编译器同步两个函数（如果不自己声明它们，两者就会同步）。但这同时也禁止了客户程序员按值或者通过赋值传递一个 CGI_vector。

CGI_vector 的工作是获取 QUERY_STRING，并把它解析成“名称／值”对，这需要在 Pair 的帮助下完成。它首先将字串复制到本地分配的内存，并用常数指针 start 跟踪起始地址（稍后会在破坏器中用于释放内存）。随后，它用自己的 nextPair()方法将字串解析成原始的“名称／值”对，各个对之间用一个“=”和“&”符号分隔。这些对由 nextPair()传递给 Pair 构建器，所以 nextPair()返回的是一个 Pair 对象。随后用 push_back()将该对象加入 vector。nextPair()遍历完整个 QUERY_STRING 后，会返回一个零值。

现在基本工具已定义好，它们可以简单地在一个 CGI 程序中使用，就象下面这样：

879-881 页程序

alreadyInList()函数与前一个版本几乎是完全相同的，只是它假定所有电子函件地址都在一个“<>”内。

在使用 GET 方法时（通过在 FORM 引导命令的 METHOD 标记内部设置，但这在这里由数据发送的方式控制），Web 服务器会收集位于“?”后面的所有信息，并把它们置入环境变量 QUERY_STRING（查询字串）里。所以为了读取那些信息，必须获得 QUERY_STRING 的值，这是用标准的 C 库函数 getnv()完成的。在 main()中，注意对 QUERY_STRING 的解析有多么容易：只需把它传递给用于 CGI_vector 对象的构建器（名为 query），剩下的所有工作都会自动进行。从这时开始，我们就可以从 query 中取出名称和值，把它们当作数组看待（这是由于 operator[]在 vector 里已经过载了）。在调试代码中，大家可看到这一切是如何运作的；调试代码封装在预处理器引导命令#if defined(DEBUG)和#endif(DEBUG)之间。

现在，我们迫切需要掌握一些与 CGI 有关的东西。CGI 程序用两个方式之一传递它们的输入：在 GET 执行期间通过 QUERY_STRING 传递（目前用的这种方式），或者在 POST 期间通过标准输入。但 CGI 程序通过标准输出发送自己的输出，这通常是用 C 程序的 printf()命令实现的。那么这个输出到哪里去了呢？它回到了 Web 服务器，由服务器决定该如何处理它。服务器作出决定的依据是 content-type（内容类型）头数据。这意味着假如 content-type 头不是它看到的第一件东西，就不知道该如何处理收到的数据。因此，我们无论如何也要使所有 CGI 程序都从 content-type 头开始输出。

在目前这种情况下，我们希望服务器将所有信息都直接反馈回客户程序（亦即我们的程序片，它们正在等候给自己的回复）。信息应该原封不动，所以 content-type 设为 text/plain（纯文本）。一旦服务器看到这个头，就会将所有字串都直接发还给客户。所以每个字串（三个用于出错条件，一个用于成功的加入）都会返回程序片。

我们用相同的代码添加电子函件名称（用户的姓名）。但在 CGI 脚本的情况下，并不存在无限循环——程序只是简单地响应，然后就中断。每次有一个 CGI 请求抵达时，程序都会启动，对那个请求作出反应，然后自行关闭。所以 CPU 不可能陷入空等待的尴尬境地，只有启动程序和打开文件时才存在性能上的隐患。Web 服务器对 CGI 请求进行控制时，它的开销会将这种隐患减轻到最低程度。

这种设计的另一个好处是由于 Pair 和 CGI_vector 都得到了定义，大多数工作都帮我们自动完成了，所以只需修改 main()即可轻松创建自己的 CGI 程序。尽管小服务程序（Servlet）最终会变得越来越流行，但为了创建快速的 CGI 程序，C++仍然显得非常方便。

15.6.4 POST 的概念

在许多应用程序中使用 GET 都没有问题。但是，GET 要求通过一个环境变量将自己的数据传递给 CGI 程序。但假如 GET 字串过长，有些 Web 服务器可能用光自己的环境空间（若字串长度超过 200 字符，就应开始关心这方面的问题）。CGI 为此提供了一个解决方案：POST。通过 POST，数据可以编码，并按与 GET 相同的方法连结起来。但 POST 利用标准输入将编码过后的查询字串传递给 CGI 程序。我们要做的全部事情就是判断查询字串的长度，而这个长度已在环境变量 CONTENT_LENGTH 中保存好了。一旦知道了长度，就可自由分配存储空间，并从标准输入中读入指定数量的字符。

对一个用来控制 POST 的 CGI 程序，由 CGITools.h 提供的 Pair 和 CGI_vector 均可不加丝毫改变地使用。下面这段程序揭示了写这样的一个 CGI 程序有多么简单。这个例子将采用“纯”C++，所以 studio.h 库被 iostream（IO 数据流）代替。对于 iostream，我们可以使用两个预先定义好的对象：cin，用于同标准输入连接；以及 cout，用于同标准输出连接。有几个办法可从 cin 中读入数据以及向 cout 中写入。但下面这个程序准备采用标准方法：用“<<”将信息发给 cout，并用一个成员函数（此时是 read()）从 cin 中读入数据：

883 页程序

getenv()函数返回指向一个字串的指针，那个字串指示着内容的长度。若指针

为零，表明 CONTENT_LENGTH 环境变量尚未设置，所以肯定某个地方出了问题。否则就必须用 ANSI C 库函数 atoi()将字串转换成一个整数。这个长度将与 new 一起运用，分配足够的存储空间，以便容纳查询字串（另加它的空中止符）。随后为 cin()调用 read()。read()函数需要取得指向目标缓冲区的一个指针以及要读入的字节数。随后用空字符（null）中止 query_str，指出已经抵达字串的末尾，这就叫作“空中止”。

到这个时候，我们得到的查询字串与 GET 查询字串已经没有什么区别，所以把它传递给用于CGI_vector的构建器。随后便和前例一样，我们可以自由vector内不同的字段。

为测试这个程序，必须把它编译到主机 Web 服务器的 cgi-bin 目录下。然后就可以写一个简单的 HTML 页进行测试，就象下面这样：

884 页程序

填好这个表单并提交出去以后，会得到一个简单的文本页，其中包含了解析出来的结果。从中可知道 CGI 程序是否在正常工作。

当然，用一个程序片来提交数据显得更有趣一些。然而，POST 数据的提交属于一个不同的过程。在用常规方式调用了 CGI 程序以后，必须另行建立与服务器的一个连接，以便将查询字串反馈给它。服务器随后会进行一番处理，再通过标准输入将查询字串反馈回 CGI 程序。

为建立与服务器的一个直接连接，必须取得自己创建的 URL，然后调用 openConnection()创建一个 URLConnection。但是，由于 URLConnection 一般不允许我们把数据发给它，所以必须很可笑地调用 setDoOutput(true)函数，同时调用的还包括 setDoInput(true)以及 setAllowUserInteraction(false)——注释⑥。最后，可调用 getOutputStream()来创建一个 OutputStream（输出数据流），并把它封装到一个 DataOutputStream 里，以便能按传统方式同它通信。下面列出的便是一个用于完成上述工作的程序片，必须在从它的各个字段里收集了数据之后再执行它：

885-887 页程序

⑥：我不得不说自己并没有真正理解这儿都发生了什么事情，这些概念都是从 Elliotte Rusty Harold 编著的《Java Network Programming》里得来的，该书由 O'Reilly 于 1997 年出版。他在书中提到了 Java 连网函数库中出现的许多令人迷惑的 Bug。所以一旦涉足这些领域，事情就不是编写代码，然后让它自己运行那么简单。一定要警惕潜在的陷阱！

信息发送到服务器后，我们调用 getInputStream()，并把返回值封装到一个 DataInputStream 里，以便自己能读取结果。要注意的一件事情是结果以文本行的形式显示在一个 TextArea（文本区域）中。为什么不简单地使用 getAppletContext().showDocument(u)呢？事实上，这正是那些陷阱中的一个。上述代码可以很好地工作，但假如试图换用 showDocument()，几乎一切都会停止运行。也就是说，showDocument()确实可以运行，但从 POSTtest 得到的返回结果是“Zero CONTENT_LENGTH”（内容长度为零）。所以不知道为什么原因，showDocument()阻止了 POST 查询向 CGI 程序的传递。我很难判断这到底是一个

在以后版本里会修复的 Bug，还是由于我的理解不够（我看过的书对此讲得都很模糊）。但无论在哪种情况下，只要能坚持在文本区域里观看自 CGI 程序返回的内容，上述程序片运行时就没有问题。

企业级程序设计概念（第二版）

Enterprise computing is about collecting and distributing information.

You do this by creating common repositories (single points of access) to that information, and allowing people to get at that information in multiple ways. So enterprise computing is creating and manipulating those common repositories, and providing ways for users to view and manipulate the information in those repositories.

In this chapter you'll see that there are a number of different ways to achieve this goal. The varied mechanics needed for the collection and distribution of information has to do with the variety of clients we must deal with (different application protocols) and the variety of repositories holding our enterprise data (relational databases, hierarchical databases, files, newsgroups and Email).

Since we can loosely define the any enterprise as a community of individuals working together let's leverage this definition and create a "Community Information System" (CIS) that will allow members of the community to collect and distribute information about what is going on within their group. This group could be 20 people or 20,000 but some of the services that may be needed would include the following:

Community Calendar/Schedule of events
Community directory (phone/email listings)
Community interest lists
General Announcements/Notices/News items
Lost & Found
Ticket sales to community events (this would justify the security issues)
Classified ads
An anti-spamming system to prevent spam harvesters—either you have a password, or it actually emails the information back to you, after verifying your email address shows that you're a community member.

So lets start by looking at an API set that provides a common interface to all those relational databases.

15.7 用 JDBC 连接数据库

据估算，将近一半的软件开发都要涉及客户（机）/ 服务器方面的操作。Java 为自己保证的一项出色能力就是构建与平台无关的客户机 / 服务器数据库应用。在 Java 1.1 中，这一保证通过 Java 数据库连接（JDBC）实现了。

数据库最主要的一个问题就是各家公司之间的规格大战。确实存在一种“标准”数据库语言，即“结构查询语言”（SQL-92），但通常都必须确切知道自己要和哪家数据库公司打交道，否则极易出问题，尽管存在所谓的“标准”。JDBC 是面向“与平台无关”设计的，所以在编程的时候不必关心自己要使用的是什么数据库产品。然而，从 JDBC 里仍有可能发出对某些数据库公司专用功能的调用，所以仍然不可任性妄为。

和 Java 中的许多 API 一样，JDBC 也做到了尽量的简化。我们发出的方法调用对应于从数据库收集数据时想当然的做法：同数据库连接，创建一个语句并执行查询，然后处理结果集。

为实现这一“与平台无关”的特点，JDBC 为我们提供了一个“驱动程序管理器”，它能动态维护数据库查询所需的所有驱动程序对象。所以假如要连接由三家公司开发的不同种类的数据库，就需要三个单独的驱动程序对象。驱动程序对象会在装载时由“驱动程序管理器”自动注册，并可用 Class.forName()强行装载。

为打开一个数据库，必须创建一个“数据库 URL”，它要指定下述三方面的内容：

(1) 用“jdbc”指出要使用 JDBC。

(2) “子协议”：驱动程序的名字或者一种数据库连接机制的名称。由于 JDBC 的设计从 ODBC 吸收了许多灵感，所以可以选用的第一种子协议就是“jdbc-odbc 桥”，它用“odbc”关键字即可指定。

(3) 数据库标识符：随使用的数据库驱动程序的不同而变化，但一般都提供了一个比较符合逻辑的名称，由数据库管理软件映射（对应）到保存了数据表的一个物理目录。为使自己的数据库标识符具有任何含义，必须用自己的数据库管理软件为自己喜欢的名字注册（注册的具体过程又随运行平台的不同而变化）。

所有这些信息都统一编译到一个字串里，即“数据库 URL”。举个例子来说，若想通过 ODBC 子协议同一个标识为“people”的数据库连接，相应的数据库 URL 可设为：

```
String dbUrl = "jdbc:odbc:people"
```

如果通过一个网络连接，数据库 URL 也需要包含对远程机器进行标识的信息。

准备好同数据库连接后，可调用静态方法 DriverManager.getConnection()，将数据库的 URL 以及进入那个数据库所需的用户名密码传递给它。得到的返回结果是一个 Connection 对象，利用它即可查询和操纵数据库。

下面这个例子将打开一个联络信息数据库，并根据命令行提供的参数查询一个人的姓（Last Name）。它只选择那些有 E-mail 地址的人的名字，然后列印出符合查询条件的所有人：

888-889 页程序

可以看到，数据库 URL 的创建过程与我们前面讲述的完全一样。在该例中，数据库未设密码保护，所以用户名和密码都是空串。

用 DriverManager.getConnection()建好连接后，接下来可根据结果 Connection 对象创建一个 Statement（语句）对象，这是用 createStatement()方法实现的。根据结果 Statement，我们可调用 executeQuery()，向其传递包含了 SQL-92 标准 SQL 语句的一个字串（不久就会看到如何自动创建这类语句，所以没必要在这里知道关于 SQL 更多的东西）。

executeQuery()方法会返回一个 ResultSet（结果集）对象，它与继承器非常相似：next()方法将继承器移至语句中的下一条记录；如果已抵达结果集的末尾，则返回 null。我们肯定能从 executeQuery()返回一个 ResultSet 对象，即使查询结果是个空集（也就是说，不会产生一个违例）。注意在试图读取任何记录数据之前，都必须调用一次 next()。若结果集为空，那么对 next()的这个首次调用就会返回 false。对于结果集中的每条记录，都可将字段名作为字串使用（当然还有其他方法），从而选择不同的字段。另外要注意的是字段名的大小写是无关紧要的——SQL 数据库不在乎这个问题。为决定返回的类型，可调用 getString()，getFloat()等等。到这个时候，我们已经用 Java 的原始格式得到了自己的数据库数据，接下去可用 Java 代码做自己想做的任何事情了。

15.7.1 让示例运行起来

就 JDBC 来说，代码本身是很容易理解的。最令人迷惑的部分是如何使它在自己特定的系统上运行起来。之所以会感到迷惑，是由于它要求我们掌握如何才能使 JDBC 驱动程序正确装载，以及如何用我们的数据库管理软件来设置一个数据库。

当然，具体的操作过程在不同的机器上也会有所区别。但这儿提供的在 32 位 Windows 环境下操作过程可有效帮助大家理解在其他平台上的操作。

1. 步骤 1：寻找 JDBC 驱动程序

上述程序包含了下面这条语句：

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

这似乎暗示着一个目录结构，但大家不要被它蒙骗了。在我手上这个 JDK 1.1 安装版本中，根本不存在叫作 JdbcOdbcDriver.class 的一个文件。所以假如在看了这个例子后去寻找它，那么必然会徒劳而返。另一些人提供的例子使用的是一个假名字，如“myDriver.ClassName”，但人们从字面上得不到任何帮助。事实上，上述用于装载 jdbc-odbc 驱动程序（实际是与 JDK 1.1 配套提供的唯一驱动）的语句在联机文档的多处地方均有出现（特别是在一个标记为“JDBC-ODBC Bridge Driver”的页内）。若上面的装载语句不能工作，那么它的名字可能已随着 Java 新版本的发布而改变了；此时应到联机文档里寻找新的表述方式。

若装载语句出错，会在这个时候得到一个违例。为了检验驱动程序装载语句是不是能正常工作，请将该语句后面直到 catch 从句之间的代码暂时设为注释。如果程序运行时未出现违例，表明驱动程序的装载是正确的。

2. 步骤 2：配置数据库

同样地，我们只限于在 32 位 Windows 环境中工作；您可能需要研究一下自己的操作系统，找出适合自己平台的配置方法。

首先打开控制面板。其中可能有两个图标都含有“ODBC”字样，必须选择那个“32 位 ODBC”，因为另一个是为了保持与 16 位软件的向后兼容而设置的，和 JDBC 混用没有任何结果。双击“32 位 ODBC”图标后，看到的应该是一个卡片式对话框，上面一排有多个卡片标签，其中包括“用户 DSN”、“系统 DSN”、“文件 DSN”等等。其中，“DSN”代表“数据源名称”（Data Source Name）。它们都与 JDBC-ODBC 桥有关，但设置数据库时唯一重要的地方“系统 DSN”。尽管如此，由于需要测试自己的配置以及创建查询，所以也需要在“文件 DSN”中设置自己的数据库。这样便可让 Microsoft Query 工具（与 Microsoft Office 配套提供）正确地找到数据库。注意一些软件公司也设计了自己的查询工具。

最有趣的数据库是我们已经使用过的一个。标准 ODBC 支持多种文件格式，其中包括由不同公司专用的一些格式，如 dBASE。然而，它也包括了简单的“逗号分隔 ASCII”格式，它几乎是每种数据工具都能够生成的。就目前的例子来说，我只选择自己的“people”数据库。这是我多年来一直在维护的一个数据库，中间使用了各种联络管理工具。我把它导出成为一个逗号分隔的 ASCII 文件（一般有个.csv 扩展名，用 Outlook Express 导出通信簿时亦可选用同样的文件格式）。在“文件 DSN”区域，我按下“添加”按钮，选择用于控制逗号分隔 ASCII 文件的文本驱动程序（Microsoft Text Driver），然后撤消对“使用当前目录”的选择，以便导出数据文件时可以自行指定目录。

大家会注意到在进行这些工作的时候，并没有实际指定一个文件，只是一个目录。那是因为数据库通常是由某个目录下的一系列文件构成的（尽管也可能采用其他形式）。每个文件一般都包含了单个“数据表”，而且 SQL 语句可以产生从数据库中多个表摘取出来的结果（这叫作“联合”，或者 join）只包含了单张表的数据库（就象目前这个）通常叫作“平面文件数据库”。对于大多数问题，如果已经超过了简单的数据存储与获取力所能及的范围，那么必须使用多个数据表。通过“联合”，从而获得希望的结果。我们把这些叫作“关系型”数据库。

3. 步骤 3：测试配置

为了对配置进行测试，需用一种方式核实数据库是否可由查询它的一个程序“见到”。当然，可以简单地运行上述的 JDBC 示范程序，并加入下述语句：

```
Connection c = DriverManager.getConnection(
dbUrl, user, password);
```

若掷出一个违例，表明你的配置有误。

然而，此时很有必要使用一个自动化的查询生成工具。我使用的是与 Microsoft Office 配套提供的 Microsoft Query，但你完全可以自行选择一个。查询工具必须知道数据库在什么地方，而 Microsoft Query 要求我进入 ODBC Administrator 的“文件 DSN”卡片，并在那里新添一个条目。同样指定文本驱动程序以及保存数据库的目录。虽然可将这个条目命名为自己喜欢的任何东西，但最好还是使用与“系统 DSN”中相同的名字。

做完这些工作后，再用查询工具创建一个新查询时，便会发现自己的数据库可以使用了。

4. 步骤 4：建立自己的 SQL 查询

我用 Microsoft Query 创建的查询不仅指出目标数据库存在且次序良好，也会自动生成 SQL 代码，以便将其插入我自己的 Java 程序。我希望这个查询能够检

查记录中是否存在与启动 Java 程序时在命令行键入的相同的“姓”（Last Name）。所以作为一个起点，我搜索自己的姓“Eckel”。另外，我希望只显示出有对应 E-mail 地址的那些名字。创建这个查询的步骤如下：

(1) 启动一个新查询，并使用查询向导（Query Wizard）。选择“people”数据库（等价于用适应的数据库 URL 打开数据库连接）。

(2) 选择数据库中的“people”表。从这张数据表中，选择 FIRST，LAST 和 EMAIL 列。

(3) 在“Filter Data”（过滤器数据库）下，选择 LAST，并选择“equals”（等于），加上参数 Eckel。点选“And”单选钮。

(4) 选择 EMAIL，并选中“Is not Null”（不为空）。

(5) 在“Sort By”下，选择 FIRST。

查询结果会向我们展示出是否能得到自己希望的东西。

现在可以按下 SQL 按钮。不需要我们任何方面的介入，正确的 SQL 代码会立即弹现出来，以便我们粘贴和复制。对于这个查询，相应的 SQL 代码如下：

893 页上程序

若查询比较复杂，手工编码极易出错。但利用一个查询工具，就可以交互式地测试自己的查询，并自动获得正确的代码。事实上，亲手为这些事情编码是难以让人接受的。

5. 步骤 5：在自己的查询中修改和粘贴

我们注意到上述代码与程序中使用的代码是有所区别的。那是由于查询工具对所有名字都进行了限定，即便涉及的仅有一个数据表（若真的涉及多个数据表，这种限定可避免来自不同表的同名数据列发生冲突）。由于这个查询只需要用到一个数据表，所以可考虑从大多数名字中删除“people”限定符，就象下面这样：

893 页下程序

此外，我们不希望“硬编码”这个程序，从而只能查找一个特定的名字。相反，它应该能查找我们在命令行动态提供的一个名字。所以还要进行必要的修改，并将 SQL 语句转换成一个动态生成的字串。如下所示：

893-894 页程序

SQL 还有一种方式可将名字插入一个查询，名为“程序”（Procedures），它的速度非常快。但对于我们的大多数实验性数据库操作，以及一些初级应用，用 Java 构建查询字串已经很不错了。

从这个例子可以看出，利用目前找得到的工具——特别是查询构建工具——涉及 SQL 及 JDBC 的数据库编程是非常简单和直观的。

15.7.2 查找程序的 GUI 版本

最好的方法是让查找程序一直保持运行，要查找什么东西时只需简单地切换到它，并键入要查找的名字即可。下面这个程序将查找程序作为一个

“application/applet”创建，且添加了名字自动填写功能，所以不必键入完整的姓，即可看到数据：

894-896 页程序

数据库的许多逻辑都是相同的，但大家可看到这里添加了一个 TextListener，用于监视在 TextField（文本字段）的输入。所以只要键入一个新字符，它首先就会试着查找数据库中的“姓”，并显示出与当前输入相符的第一条记录（将其置入 completion Label，并用它作为要查找的文本）。因此，只要我们键入了足够的字符，使程序能找到与之相符的唯一一条记录，就可以停手了。

15.7.3 JDBC API 为何如何复杂

阅览 JDBC 的联机帮助文档时，我们往往会产生畏难情绪。特别是 DatabaseMetaData 接口——与 Java 中看到的大多数接口相反，它的体积显得非常庞大——存在着数量众多的方法，比如 dataDefinitionCausesTransactionCommit()，getMaxColumnNameLength()，getMaxStatementLength()，storesMixedCaseQuotedIdentifiers()，supportsANSI92IntermediateSQL()，supportsLimitedOuterJoins()等等。它们有这儿有什么意义吗？

正如早先指出的那样，数据库起初一直处于一种混乱状态。这主要是由于各种数据库应用提出的要求造成的，所以数据库工具显得非常“强大”——换言之，“庞大”。只是近几年才涌现出了 SQL 的通用语言（常用的还有其他许多数据库语言）。但即便象 SQL 这样的“标准”，也存在无数的变种，所以 JDBC 必须提供一个巨大的 DatabaseMetaData 接口，使我们的代码能真正利用当前要连接的一种“标准”SQL 数据库的能力。简言之，我们可编写出简单的、能移植的 SQL。但如果想优化代码的执行速度，那么为了适应不同数据库类型的特点，我们的编写代码的麻烦就大了。

当然，这并不是 Java 的缺陷。数据库产品之间的差异是我们和 JDBC 都要面对的一个现实。但是，如果能编写通用的查询，而不必太关心性能，那么事情就要简单得多。即使必须对性能作一番调整，只要知道最终面向的平台，也不必针对每一种情况都编写不同的优化代码。

在 Sun 发布的 Java 1.1 产品中，配套提供了一系列电子文档，其中有对 JDBC 更全面的介绍。此外，在由 Hamilton Cattel 和 Fisher 编著、Addison-Wesley 于 1997 年出版的《JDBC Database Access with Java》中，也提供了有关这一主题的许多有用资料。同时，书店里也经常出现一些有关 JDBC 的新书。

服务器小程序—Servlets

现在我们理解了 JDBC，知道了它怎样用一个 API 抽象了 and how it abstracts all those different database backends into a common API set, let’s turn our attention to another common task - handling HTTP requests and responses. It is taken for granted in today’s technical environment that client access from the Internet or corporate intranets is a sure way to allow many users to access data and resources easily. This type of access is predicated on the clients utilizing the World Wide Web standards of Hypertext Markup Language (HTML) and Hypertext Transfer Protocol (HTTP). Wouldn’t it be nice to have an API set that abstracted out this commonly used area of client access? Welcome Java Servlets!

Traditionally, the way to handle a problem such as allowing an Internet client to update their personal data is to create an HTML page with a text field and a “submit” button. The user can type whatever he or she wants into the text field, and it will be submitted to the server without question. As it submits the data, the Web page also tells the server what to do with the data by mentioning the Common Gateway Interface (CGI) program that the server should run after receiving this data. This CGI program is typically written in either Perl or C (and sometimes C++, if the server supports it), and it must handle everything. First it looks at the data and decides whether it’s in the correct format. If not, the CGI program must create an HTML page to describe the problem; this page is handed to the server, which sends it back to the user. The user must then back up a page and try again. If the data is correct, the CGI program opens the data file and either adds the email address to the file or discovers that the address is already in the file. In both cases it must format an appropriate HTML page for the server to return to the user.

As Java programmers, this seems like an awkward way for us to solve the problem, and naturally, we’d like to do the whole thing in Java. First, we’ll use a Java applet to take care of data validation at the client site, without all that tedious Web traffic and page formatting. Then let’s skip the Perl CGI script in favor of a Java application running on the server. In fact, let’s skip the Web server altogether and simply make our own network connection from the applet to the Java application on the server!

As you’ll see, there are a number of issues that make this a more complicated problem than it seems. It would be ideal to write the applet but applet’s, while a proven technology with plenty of support, have been problematic in the Wild World Web where different browsers handle applet’s differently. In a corporate intranet where there is some level of standardization this seems possible in the short term but what happens with the next acquisition or merger? What happens when employees want to start working from home? That’s easy - things start to look a lot like the Internet - you can’t depend on anything especially how applets are implemented in your client’s browsers. So to be on the safe side, what we really want to do is deal with straight HTML and HTTP within our java server. The client knows nothing of the implementation, they are only aware that they can get at their data and perform their work without installing, upgrading or calling tech support.

Sun has delivered on this need. The Servlet API wraps up the HTTP protocol so we can put Java on the server side of our HTTP connection and deal with our client in HTML and HTTP. Servlets are completely server side java for the Web. The client sees nothing except HTTP and HTML.

基本的 Servlet

The architecture of the Servlet API is that of a classic service provider with a service( ) method through which all client requests will drive and life cycle methods init( ) and destroy( ).

```
public interface Servlet {
      public void init(ServletConfig config)
         throws ServletException;
      public ServletConfig getServletConfig();
      public void service(ServletRequest req,
         ServletResponse res)
         throws ServletException, IOException;
      public String getServletInfo();
      public void destroy();
}
```

getServletConfig( ) sole purpose is to return a ServletConfig object which contains initialization and startup parameters for this servlet and getServletInfo( ) returns a string containing information about the servlet, such as author, version, and copyright.

The **GenericServlet** class is a shell implementation of this interface, nothing more. The **HttpServlet** class is an extension of GenericServlet and is designed specifically to handle the HTTP protocol.

Although you can derive your servlets from GenericServlet, Sun recommends that all servlets derive from HttpServlet. This makes sense since your servlet is designed to work with a servlet engine that is satisfying clients requests from within a Web server. Why not utilize the built-in parsing capabilities for POST and GET that come with HttpServlet? We will be getting into this shortly.

The most wonderful attribute of the Servlet API is the auxiliary objects that come along with the HttpServlet class to support it. Look at the service( ) method in the Servlet interface. It has two parameters **ServletRequest and ServletResponse.** With the HttpServlet class these two object are extended for HTTP as well—HttpServletRequest and HttpServletResponse. Let’s take a closer look.

```
//: c15:servlets:ServletsRule.java
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class ServletsRule extends HttpServlet {
  int i = 0; // Servlet "persistence"
  public void service(HttpServletRequest req,
  HttpServletResponse res) throws IOException {
    PrintWriter out = res.getWriter();
    out.print("<HEAD><TITLE>");
    out.print("A server-side strategy");
    out.print("</TITLE></HEAD><BODY>");
    out.print("<h1>Servlets Rule! " + i++);
    out.print("</h1></BODY>");
    out.close();
  }
} ///:~
```

ServletsRule is about as simple as a servlet can get. But that is the beauty of it all - just think how much stuff is being handled for us! Once the servlet is initialized - its init( ) method has run to completion - can clients enter the service( ) method. In the service method our main responsibility is to interact with the HTTP request the client sent us and build a HTTP response based upon the attributes contained within the request. In ServletsRule we only manipulate the response object without looking at what the client may has sent us. We call the getWriter( ) method of the response object to get a PrintWriter object. The PrintWriter is used for writing character-based response data.

As we dive more deeply into the HttpRequest and HttpResponse objects you should notice that a greater understanding of HTTP and HTML would be helpful. Servlets are designed for Web server-side development and it shows. Now let’s scratch a little deeper by getting some HTML form data that was passed to the servlet in the request object.

```
//: c15:servlets:EchoForm.java
// Dumps the name-value pairs of any HTML form
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class EchoForm extends HttpServlet {
  public void doGet(HttpServletRequest req,
    HttpServletResponse res) throws IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    out.print("<h1>Your form contained:</h1>");
    Enumeration flds = req.getParameterNames();
    while(flds.hasMoreElements()) {
      String field = (String)flds.nextElement();
      String value = req.getParameter(field);
      out.print(field + " = " + value + "<br>");
    }
    out.close();
  }
  public void doPost(HttpServletRequest req,
    HttpServletResponse res) throws IOException {
    doGet(req, res);
  }
} ///:~
```

The EchoForm servlet parrots back to our client the request fields that have been sent to the servlet as part of the request object. Since the request fields will be placed into a response we must set up the response object with calls to setContentType( ) and getWriter( ). In an HttpResponse object the setContentType( ) methods sets the Content-Type HTTP header. This is most commonly "text/html".

Notice that in EchoForm the service( ) method has been replaced by doGet( ). This automatic HTTP method name parsing is a feature of HttpServlet. HTTP was designed for the Web and has been made more general than necessary. The first word on the full request line is simply the name of the method (command) to be executed on the Web page. The built-in methods are GET, POST, HEAD, PUT, DELETE, LINK and UNLINK. The HttpServlet class is written to parse these methods and let the programmer react differently for a GET than a POST or even a HEAD request method. If you don't care you can just override service( ). Since most HTTP methods are POST or GET many times you just implement one and direct the other to it.

In The HTTP request object has the potential to come with request parameters and generally does. The parameters are typically name-value pairs sent as part of its query string (for GET requests) or as encoded post data (for POST requests). EchoForm uses the request object's getParameterNames( ) method to loop through the parameter list. The getParameter( ) method is used to pull the value. The pair is then written to the PrintWriter of the response object with the appropriate HTML tags. The response object is sent to the client when the servlet is finished.

Servlets and Multithreading

Now that you understand the basics you should be realizing that servlets are excellent for server-side Web development. Elegant and straight forward, they do just about everything for you - right? Well, almost. Remember early we said all client requests drive through the service method? This is the well-used, high traffic corridor of the servlet and more than one client request may come through at the same time. The servlet engine has a pool of threads that it will dispatch to handle client requests. It is quite likely that two clients arriving at the same time could beprocessing through your service( ) or doGet( ) or doPost( ) methods at the same time. Therefore the service( ) methods and other methods called by HttpServlet.service( ) (e.g., doGet( ), doPost( ), doHead( ),etc.) need to be written in a thread-safe manner. Any common resources (files, databases) that will be used by your client requests will need to be synchronized.

ThreadServlet is a simple example that simply synchronizes around the threads sleep( ) method. This will hold up all threads until the allotted time (5000 ms) is all used up. When testing this you should start several browsers instances and hit this servlet as quickly as possible.

```
//: c15:servlets:ThreadServlet.java
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;

public class ThreadServlet extends HttpServlet {
  int i;
  public void service(HttpServletRequest req,
    HttpServletResponse res) throws IOException {
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    synchronized(this) {
      try {
        Thread.currentThread().sleep(5000);
      } catch(InterruptedException e) {}
    }
    out.print("<h1>Finished " + i++ + "</h1>");
    out.close();
  }
} ///:~
```

Handling Sessions with Servlets

The servlet API comes with more than just the classes that implement the servlet interface, GenericServlet and HttpServlet and the Request and Response objects. The design of HTTP is such that it is a 'sessionless' protocol. A great deal of effort has gone into mechanisms that will allow Web developers to track sessions. How could companies do e-commerce if you couldn't keep track of client and the items they have put into their shopping cart? You couldn't! This may be great for privacy advocates but it does little to help create robust, commerce driven Web sites.

There are several methods of session tracking but the most common method is with persistant 'cookies'. The term cookie sounds cute and could be perceived as a session tracking solution that was baked up in someone's garage. The fact is that cookies are an integral part of the Internet standards. The HTTP Working Group of the Internet Engineering Task Force has written cookies in the official standard in RFC 2109 (http://ds.internic.net/rfc/rfc2109.txt or check http://www.cookiecentral.com).

A cookie is nothing more than a small piece of information sent by a Web server to a browser. The browser stores the cookie locally and all calls to the server from that browser will contain the cookie as an identifier. The cookie therefore acts to uniquely identify the client with each hit of this Web server. It should be noted that clients can turn off the browsers ability to accept cookies. If your site still need to be able to session track this type of client then another method of session tracking (URL rewriting or hidden form fields) will have to be incorporated. The session tracking capabilities built into the Servlet API are designed around cookies.

The Cookie Class

The Servlet API (version 2.0 and up) provides the javax.servlet.http.Cookie class. This class incorporates all the HTTP header details and allows the setting of various cookie attributes. Using the cookie is simply a matter of creating it using the constructor and adding it to the response object. The constructor takes a cookie name as the first argument and a value as the second. Cookies are added to the response object before you send any content.

```
Cookie oreo = new Cookie("TIJava", "2000");
res.addCookie(cookie);
```

Cookies are then received by calling the getCookies( ) method of the

HttpServletRequest object which returns an array of cookie objects.

```
Cookie[] cookies = req.getCookies();
```

The Session Class

A session in the world of HTTP and the Internet is one or more page requests by a client to a Web site during a defined period of time. If I am buying my groceries on-line, I want a session to be confined to the period from when I first add an item to my shopping cart to the point where I checkout. Each item I add to the shopping cart will be a new connection in the HTTP world, they have no knowledge of previous connections or items in the shopping cart. The mechanics supplied by the Cookie specification allows us to perform 'session tracking'.

You should understand that a cookie is an object that encapsulates that small bit of information that will be stored on the client side. A Servlet Session object lives on the server side of the communication channel and its goal is to capture data about this client that would be useful as the client moves through and interacts with your Web site. This data may be pertinent for the present session, such as items in the shopping cart or it may be information you asked the client to enter such as authentication information entered when the client first entered your Web site and which should not have to be re-enter before a set time of inactivity.

The Session class of the Servlet API uses the Cookie class but really all the session object needs is a unique identifier stored on the client and passed to the server. Usually this is a cookie and that is the mechanism we will cover here. Web sites may also use the other types of session tracking but these mechanisms will be more difficult to implement as they are not encapsulated into the Servlet API.

Let's take a look at implementing session tracking with the Servlet API:

```
//: c15:servlets:SessionPeek.java
// Using the HttpSession class.
import java.io.*;
import java.util.Date;
import javax.servlet.*;
import javax.servlet.http.*;

public class SessionPeek extends HttpServlet {
  public void doGet(HttpServletRequest req,
  HttpServletResponse res)
```

```
throws ServletException, IOException {
   // Retrieve Seesion Object before any
   // output is sent to the client.
   HttpSession session = req.getSession();
   // Get the output stream
   ServletOutputStream out =
      res.getOutputStream();
   res.setContentType("text/html");
   out.println("<HEAD><TITLE> SessionPeek ");
   out.println(" </TITLE></HEAD><BODY>");
   out.println("<h1> SessionPeek </h1>");
   // A simple hit counter for this session.
   Integer ival = (Integer)
      session.getValue("sesspeek.cntr");
   if(ival==null)
      ival = new Integer(1);
   else
      ival = new Integer(ival.intValue() + 1);
   session.putValue("sesspeek.cntr", ival);
   out.println("You have hit this page <b>"
      + ival + "</b> times.<p>");
   // Session Data
   out.println("<h2>");
   out.println(" Saved Session Data </h2>");
   // loop through all data in the session
   // and spit is out.
   String[] sesNames = session.getValueNames();
   for(int i = 0; i < sesNames.length; i++) {
      String name = sesNames[i];
      String value =
         session.getValue(name).toString();
      out.println(name + " = " + value + "<br>");
   }
   // Session Statistics
   out.println("<h3> Session Statistics </h3>");
   out.println("Session ID: "
      + session.getId() + "<br>");
   out.println("New Session: " + session.isNew()
      + "<br>");
   out.println("Creation Time: "
      + session.getCreationTime());
   out.println("<I>(" +
      new Date(session.getCreationTime())
      + ")</I><br>");
```

```
        out.println("Last Accessed Time: " +
          session.getLastAccessedTime());
        out.println("<I>(" +
          new Date(session.getLastAccessedTime())
          + ")</I><br>");
        out.println("Session Inactive Interval: "
          + session.getMaxInactiveInterval());
        out.println("Session ID in Request: "
          + req.getRequestedSessionId() + "<br>");
        out.println("Is session id from Cookie: "
          + req.isRequestedSessionIdFromCookie()
          + "<br>");
        out.println("Is session id from URL: "
          + req.isRequestedSessionIdFromURL()
          + "<br>");
        out.println("Is session id valid: "
          + req.isRequestedSessionIdValid()
          + "<br>");
        out.println("</BODY>");
        out.close();
      }
      public String getServletInfo() {
        return "A session tracking servlet";
      }
    } ///:~
```

The very first thing we do when we enter the doGet( ) method is to call getSession( ) on the request object. getSession will return the session object associated with this request. Do not be misled into thinking that the session object is returned with the request. The session object does not travel across the network it lives on the server and is associated with a client and its requests.

getSession( ) now comes in two versions—no parameter as used here and getSession(boolean). getSession(true) is equivalent to getSession( ). The only reason for the boolean is to state whether you want to the session object created if it is not found. getSession(true) is the most likely call hence getSession( ).

The session object, if it is not new, will give us details about our client on their previous visits. If the session object is new then we will start with this visit to gather information about this client’s activities. Capturing this client information is done through the setAttribute( ) and getAttribute( ) methods of the session object.

```
java.lang.Object getAttribute(java.lang.String)
void setAttribute(java.lang.String name,
                  java.lang.Object value)
```

The session object uses a simple name-value pairing for loading information. The values must be derived from java.lang.Object and the name is a string. In ServletPeek we are keeping track of how many times the client has been back here during this session. This is done with an Integer object that is named sesspeek.cntr. You should notice in the if-else statement that if the name is not found we create a new Integer with value of 1, otherwise we create a new Integer with a value equal to the incremented value of the previously held Integer. We stick the new Integer into the session object and let the garbage collector handle the old Integer. You should realize that if you are using the same key the new object would overwrite the old one. Lastly, we use our incremented counter to display how many times the client has visited during this session.

Related to getAttribute( ) and setAttribute( ) is getAttributeNames( ). getAttributeNames( ) returns an enumeration of all the names of objects that are bound to the session object. This is quite handy and a small while loop has been added to SessionPeek to show this method in action.

This brings us to the question "Just how long does a session object hang around for?" The answer depends upon the servlet engine you are using although I think they usually default to 30 minutes (1800 seconds), which is what you should see from the ServletPeek call to getMaxInactiveInterval( ). We have tested this and found mixed results between servlet engines. Sometimes the session object can hang around overnight. I have never seen a case where the session object disappears before the Inactive Interval. You can try this by setting the Inactive Interval with setMaxInactiveInterval( ) to 5 seconds and see if your session object hangs around or if it is cleaned up at the appropriate time. This may be an attribute you will want to investigate while choosing a servlet engine.

Getting the Servlet examples to work

If you are not already working with an application server that handles Sun's Servlet and JSP technologies for you, then you may want to reevaluate your choice of application server or you may want to download the Tomcat implementation of Java Servlets and JSP's. This can be found at http://jakarta.apache.org.

First, you should follow the instructions for decompressing the version specific to your environment. This will install the Tomcat implementation into a directory structure under where you unzipped it. Second, edit the server.xml so that you have a new Web application. Lastly, you will then want to load the servlet examples into that new Web application directory where the Tomcat server can find them.

Java Server Pages

Servlets are found by the servlet engine looking for the .class file along a Web application path defined by the configuration of the Web server. That makes sense, no problem there. But what if we would like the Web server to compile the servlet at the time it is invoked, thereby insuring that the lastest and greatest code is delivered to the client. This is the nature of JavaServer Pages or JSPs.

You can think of JSPs as special Java tags inside the HTML page that will result in a servlet being generated, then compiled by the Web server at the time the client invokes that page. This allows the separation of the dynamic content and the static content in the HTML page. You get support for scriping and tags plus the reuse associated with those tags and the JSP components that provide the functionality. Essentially, your previously static HTML pages have become dynamic in some of it parts.

The structure of a JSP page is a cross between a servlet and an HTML page. The JSP page is a test-based document that describes how to process a request and create a response. The text based description of the page intermixes template data with some dynamic actions and leverages the Java platform. JavaServer Pages is a standard extension that is defined on top of the Servlet Standard Extension. JSP 1.1 uses class from Servlet 2.2 which relies on JRE 1.1.

Here’s an extremely simple JSP example that uses a standard Java library call to get the current time in milliseconds, which is then divided by 1000 to produce the time in seconds. Since a JSP expression (the <%=) is used, the result of the calculation is coerced into a String so it can be printed to the out object (which puts it in the Web page):

```
//:! c15:jsp:ShowSeconds.jsp
<html><body>
<H1>The time in seconds is:
<%= System.currentTimeMillis()/1000 %></H1>
</body></html>
///:~
```

There is a great deal more going on here than meets the eye. If we follow the route of the request and response we can get an idea how much many layers the request and response are moving through. The client creates the request for the JSP page and sends it off to the Web Server. The Web Server must be able to find the JSP page and forward the request to the page. The JSP page has associated with it a compiled component that is created from the Java code embedded within. This JSP component is the ultimate destination of the request and the creator of the response. The response then bubbles up the same path that the request followed picking up pieces to pass back to the client along the way.

This is important as Sun describes a JSP page as "a text-based document that describes how to process a request and create a response." That about a broad as you can get so let's dig deeper.

Basic operations

We know that the server automatically creates, compiles, loads and runs a special servlet to generate the page's content. The static portions of the HTML page are generated by the servlet using the equivalent of out.println( ) calls within the servlet. The dynamic portions are included directly into the servlet.

There is good and bad in everything and JSPs are no different. The downside to all this dynamism is poor performance for first time access. Try it—it is obvious. The first access is slow and subsequent accesses are excellent.

Implicit Objects

When we looked at servlets there were several objects already built into the API—response, request, session, etc. These objects are very conveniently built into the JSP specification and they provide the same robust foundation for manipulating HTTP and HTML in a Web application.

JSP writers have access to these implicit objects within the JSP page just as you would within a servlet. The implicit objects in a JSP are detailed in the table below. Each of the variables has a class or interface that is defined in the core Java technology or the Java Servlet API. Scope of each object can vary significantly. For example, a Session object would have a scope exceeding that of a page as it many span several client requests and pages and an application object would provide service to a group of jsp pages that together would represent a Web application.

Implicit variable

Of Type (javax.servlet)

Description

Scope

request

protocol dependent subtype of : HttpServletRequest.

The request that triggers the service invocation.

request

response

protocol dependent subtype of : HttpServletResponse.

The response to the request.

page

pageContext

jsp.PageContext

The page context encapsulates implementation-dependent features and provides convenience methods and namespace access for this JSP.

page

session

protocol dependent subtype of: http.HttpSession

The session object created for the requesting client. See Servlet Session object.

session

application

ServletContext

The servlet context obtained from the servlet configuration object (e.g., getServletConfig( ).getContext( )

app

out

jsp.JspWriter

The object that writes into the output stream.

page

config

ServletConfig

The ServletConfig for this JSP.

page

page

java.lang.Object

The instance of this page’s implementation class processing the current request.

page

JSP Scripting Elements

The implicit objects and the power of Java are all brought together with JSP actions. Actions affect the current out stream and use, modify or create objects. The actions to be performed will be determined by the details of the request object received by the JSP page. The JSP specification includes actions types that are standard and must be implemented by conforming engines. The syntax for action elements is based on XML.

The actions all start with Directives. Directives are messages to the JSP engine and the syntax is:

<%@ directive {attr=”value”}*%>

Directives do not produce any output into the current out stream but they are important in setting up your JSP pages attributes and dependencies with the JSP engine. As an example the line:

<%@ page language=”java” %>

says that the scripting language being used within the JSP page is Java. In fact the specification only describes the semantics of scripts for the language attribute equal to Java. This should give you an idea of the flexibility that is being built into the JSP technology. In the future, if you were to choose another language, say Python (a good scripting choice), then that language would have to support the Java Run-time Environment by exposing the Java technology object model to the script environment, especially the implicit variables defined above, JavaBeans properties, and public methods.

The most important directive is the page directive. It defines a number of page dependent attributes and communications these attributes to the JSP engine. These attributes include: language, extends, import, session, buffer, autoFlush, isThreadSafe, info and errorPage. For example:

<%@ page session=”true” import=”java.util.*” %>

This line indicates that the page requires participation in an (HTTP) session. Since we have not set the language directive the JSP engine defaults to java and the implicit script language variable named “session” is of type javax.servlet.http.HttpSession. If the directive had been false then the implicit variable “session” would be unavailable, the default is true.

The import attribute describes the types that are available to the scripting environment. This attribute is used just as it would be in the Java programming language i.e. a (comma separated) list of either a fully qualified Java type name denoting that type, or of a package named followed by the ".*" string denoting all the public types declared in that package. The import list is imported by the translated JSP page implementation and is available to the scripting environment. Again, this is currently only defined for when the value of the language directive is "java".

Once the directives have been used to set the scripting environment we can utilize the scripting language elements. JSP 1.1 has three scripting language elements—declarations, scriptlets, and expressions. A declaration will declare elements, a scriptlet is a statement fragment, and an expression is a complete language expression. In JSP each scripting element begins with a "<%". The exact syntax for each is:

```
<%! declaration %>
<%   scriptlet     %>
<%= expression    %>
```

White space is optional after "<%!", "<%", "<%=", and before "%>".

As mentioned earlier, all these tags are based upon XML. More accurately you could state that a JSP page could be mapped to a XML document and although this is a little touted section of the specification, I suspect you will be hearing more and more about this aspect of JSP as Java, XML, and server-side Java become more intertwined. I will not go into these mapping details here but you should be aware of them and if you need more details you should refer to the JSP specification. Therefore, you should realize that the XML equivalent syntax for the scripting elements above would be:

```
<jsp:declaration> declaration </jsp:declaration>
<jsp:scriptlet>     scriptlet     </jsp:scriptlet>
<jsp:expression>   expression   </jsp:expression>
```

Declarations are used to declare variables and methods in the scripting language used in a JSP page—Java at this time. The declaration should be a complete Java statement and should not produce any output in the current out stream. In the Hello.jsp example below the variables loadTime, loadDate and hitCount are all complete Java statements declares new variables and initializes them.

```
//:! c15:jsp:Hello.jsp
<%-- This JSP comment will not appear in the
generated html --%>
<%-- This is a JSP directive: --%>
<%@ page import="java.util.*" %>
```

```
<%-- These are declarations: --%>
<%!
    long loadTime= System.currentTimeMillis();
    Date loadDate = new Date();
    int hitCount = 0;
%>
<html><body>
<%-- The next several lines are the result of a
JSP expression inserted in the generated html;
the '=' indicates a JSP expression --%>
<H1>This page was loaded at <%= loadDate %> </H1>
<H1>Hello, world! It's <%= new Date() %></H1>
<H2>Here's an object: <%= new Object() %></H2>
<H2>This page has been up
<%= (System.currentTimeMillis()-loadTime)/1000 %>
seconds</H2>
<H3>Page has been accessed <%= ++hitCount %>
times since <%= loadDate %></H3>
<%-- A "scriptlet" which writes to the server
console. Note that a ';' is required: --%>
<%
   System.out.println("Goodbye");
   out.println("Cheerio");
%>
</body></html>
///:~
```

At the tail end of Hello.jsp is a scriptlet that writes “Goodbye” to the Web server console and “Cheerio” to the implicit out JspWriter object. Scriptlets can contain any code fragments that are valid Java statements. Scriptlets are executed at request-processing time. When all the scriptlets fragments in a given JSP are combined in the order they appear in the JSP page, they should yield a valid statement as defined by the Java programming language. Whether or not they produce any output into the out stream depends upon the actual code in the scriptlet. You should be careful as scriptlets can have side effects through their modification of the objects visible within them.

JSP expressions can found intermingled with the HTML in the middle section of Hello.jsp. Expressions are interesting because they must be complete Java statements, which are then evaluated. The result of the JSP expression is coerced to a java.lang.String which is emitted into the current implicit out JspWriter object. If the result of the expression cannot be coerced to a java.lang.String then a ClassCastException is thrown.

Extracting fields and values

```
//:! c15:jsp:DisplayFormData.jsp
<%-- Fetching the data from an HTML form. --%>
<%-- This JSP also generates the form. --%>
<%@ page import="java.util.*" %>
<html><body>
<H1>DisplayFormData</H1><H3>
<%
Enumeration f = request.getParameterNames();
if(!f.hasMoreElements()) { // No fields
%>
<form method="POST" action="DisplayFormData.jsp">
<%
  for(int i = 0; i < 10; i++) {
%>
      Field<%=i%>:
      <input type="text" size="20"
        name="Field<%=i%>" value="Value<%=i%>"><br>
<% } %>
  <INPUT TYPE=submit name=submit Value="Submit">
</form>
<% } %>
<%
Enumeration flds = request.getParameterNames();
while(flds.hasMoreElements()) {
  String field = (String)flds.nextElement();
  String value = request.getParameter(field);
%>
  <li><%= field %> = <%= value %></li>
<%
}
%></H3></body></html>
///:~
```

JSP Page Attributes and Scope

I have spent some time trying to get my code editor to view a .jsp page with syntax highlighting. This is bit more difficult than it would seem. Is a .jsp page Java or is it HTML? That's easy - it's both. So setting up my color coding was like melding the HTML section with the Java section. The real pint is that a .jsp page provides a new set of tags that allows you to separate the passive display code (HTML) from the dynamic programming code (Java).

There is no reason you can't have a whole block of code that performs some action that will provide content for you HTML. This action could be a database call or a call to some other resource. PageContext.jsp below calls getAttributeNamesInScope( ) method of pageContext to get all the attributes in the scope passed in (1 refers to page).

```
//:! c15:jsp:PageContext.jsp
<%--Viewing the attributes in the pageContext--%>
<%-- Note that you can include any amount of code
inside the scriptlet tags --%>
<%@ page import="java.util.*" %>
<html><body>
<%
  session.setAttribute("My dog", "Ralph");
  for(int scope = 1; scope <= 4; scope++) {
    out.println("<H3>Scope: " +
                  scope + "</H3><BR>");
    Enumeration e =
      pageContext.getAttributeNamesInScope(scope);
    while(e.hasMoreElements()) {
      out.println("\t<li>" +
        e.nextElement() + "</li>");
    }
  }
%>
<H4>End of list</H4>
</body></html>
///:~
```

The output looks like this:

```
Scope: 1
• javax.servlet.jsp.jspOut


• javax.servlet.jsp.jspPage
```

• javax.servlet.jsp.jspSession

• javax.servlet.jsp.jspApplication

• javax.servlet.jsp.jspPageContext

• javax.servlet.jsp.jspConfig

• javax.servlet.jsp.jspResponse

• javax.servlet.jsp.jspRequest

Scope: 2
• org.apache.tomcat.servlet.resolved

Scope: 3
• My dog

Scope: 4
• sun.servlet.workdir

• javax.servlet.context.tempdir

End of list
Scope 1 is the page scope and all objects reference available in this scope will be discarded upon completion of the current request by the page body. Scope 2 refers to the request scope and will be discarded upon completion of the current client request. As you can see I am using the Apache Tomcat implementation of the Servlets and JSP. (I am not sure what org.apache.tomcat.servlet.resolved is I will try to find out.) Scope 3 will be our session scope and the only object we have with session scope is the one that we added right before the for loop - "My dog". Scope 4 is the scope of our application and is based upon the ServletContext object. There is one ServletContext per "Web application" per Java Virtual Machine. (A "Web application" is a collection of servlets and content installed under a specific subset of the server's URL

namespace such as /catalog. In the Tomcat release this information is set via server.xml file.) At the application scope level we have to objects that represent paths for working directory and temporary directory.

Manipulating sessions in JSP

Let's take a closer look at sessions within the JSP model. The next example will exercise the session object a little bit and allow you to manipulate the amount of time before your session becomes invalid. First we must capture some information about this session object. I make a call to getID( ), getCreationTime( ) and getMaxInactiveInterval( ) and display these attributes about our session. When I first bring this session up in the Tomcat implementation the MaxInactiveInterval is 1800 seconds or 30 minutes. Now I know a bit about my session, so lets change its behavior by shortening the MaxInactiveInterval to 5 seconds. Now we should see some action. Next, I check to see if the object "My dog" is attached to the session object giving it session scope. The first time through this should be null but right afterwards we do create a String object "Ralph" and attach it to the session object by call setAttribute( ). Now Ralph should hang around for at least 5 seconds. The invalidate button at the bottom calls a second .jsp page SessionObject2.jsp that simply asks the session if it has the object tagged "My dog" then kills the session by calling invalidate( ) on the session object. "Ralph" is gone. The other button on the bottom of SessionObject.jsp is "Keep Around". This calls a third page, SessionObject3.jsp, that does NOT invalidate the session and you can see that "Ralph" in fact does hang around as long as your 5 second time interval does not expire. Try the refresh button on SessionObject.jsp or move back and forth between SessionObject and SessionObject3.jsp (Keep Around button) a couple of times using different intervals to get a feel for how long "My dog" stays around. (For those of you who have kids this is like the Tomagotchi pets - as long as you play with "Ralph" he will stick around otherwise he packs it up :-)

```
//:! c15:jsp:SessionObject.jsp
<%--Setting and getting session object values--%>
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<H3><li>This session was created at
<%= session.getCreationTime() %></li></H1>
<H3><li>MaxInactiveInterval=
  <%= session.getMaxInactiveInterval() %></li></H3>
<% session.setMaxInactiveInterval(5); %>
<H3><li>Reset MaxInactiveInterval=
  <%= session.getMaxInactiveInterval() %></li></H3>
<H2>If this session object "My dog" is still around <H2>
<H3><li>Session value for "My dog" =
<%=
      session.getAttribute("My dog")
%></li></H3>
<%-- Now add the session object "My dog" --%>
<%
      session.setAttribute("My dog",
                              new String("Ralph"));
%>
<H1>My dog's name is
<%= session.getAttribute("My dog") %></H1>
<%-- See if "My dog" wanders to another form --%>
<FORM TYPE=POST ACTION=SessionObject2.jsp>
<INPUT TYPE=submit name=submit Value="Invalidate">
</FORM>
<FORM TYPE=POST ACTION=SessionObject3.jsp>
<INPUT TYPE=submit name=submit Value="Keep Around">
</FORM>
</body></html>
///:~
//:! c15:jsp:SessionObject2.jsp
<%--The session object carries through--%>
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<H1>Session value for "My dog"
<%= session.getValue("My dog") %></H1>
<% session.invalidate(); %>
</body></html>
///:~
//:! c15:jsp:SessionObject3.jsp
<%--The session object carries through--%>
<html><body>
<H1>Session id: <%= session.getId() %></H1>
<H1>Session value for "My dog"
<%= session.getValue("My dog") %></H1>
<FORM TYPE=POST ACTION=SessionObject.jsp>
<INPUT TYPE=submit name=submit Value="Return">
</FORM>
</body></html>
///:~
```

Creating and modifying Cookies

```
//:! c15:jsp:Cookies.jsp
<%--This program has different behaviors under
  different browsers! --%>
<html><body>
```

```
<H1>Session id: <%= session.getId() %></H1>
<%
  Cookie[] cookies = request.getCookies();
  for(int i = 0; i < cookies.length; i++) { %>
Cookie name: <%= cookies[i].getName() %> <br>
value: <%= cookies[i].getValue() %><br>
Max age in seconds:
  <%= cookies[i].getMaxAge() %><br>
<% cookies[i].setMaxAge(3); %>
Max age in seconds:
  <%= cookies[i].getMaxAge() %><br>
<% response.addCookie(cookies[i]); %>
<% } %>
<%-- <% response.addCookie(
     new Cookie("Bob", "Car salesman")); %> --%>
</body></html>
///:~
```

RMI (Remote Method Invocation)

Traditional approaches to executing code on other machines across a network have been confusing as well as tedious and error-prone to implement. The nicest way to think about this problem is that some object happens to live on another machine, and you can send a message to that object and get a result as if the object lived on your local machine. This simplification is exactly what Java 1.1 Remote Method Invocation (RMI) allows you to do. This section walks you through the steps necessary to create your own RMI objects.

Remote interfaces

RMI makes heavy use of interfaces. When you want to create a remote object, you mask the underlying implementation by passing around an interface. Thus, when the client gets a reference to a remote object, what they really get is an interface reference, which happens to connect to some local stub code that talks across the network. But you don’t think about this, you just send messages via your interface reference.

When you create a remote interface, you must follow these guidelines:

The remote interface must be public (it cannot have “package access,” that is, it cannot be “friendly”). Otherwise, a client will get an error when attempting to load a remote object that implements the remote interface.

The remote interface must extend the interface java.rmi.Remote.

Each method in the remote interface must declare java.rmi.RemoteException in its throws clause in addition to any application-specific exceptions.

A remote object passed as an argument or return value (either directly or embedded within a local object) must be declared as the remote interface, not the implementation class.

Here’s a simple remote interface that represents an accurate time service:

```
//: c15:rmi:PerfectTimeI.java
// The PerfectTime remote interface.
package c15.rmi;
import java.rmi.*;

interface PerfectTimeI extends Remote {
  long getPerfectTime() throws RemoteException;
} ///:~
```

It looks like any other interface except that it extends Remote and all of its methods throw RemoteException. Remember that an interface and all of its methods are automatically public.

Implementing the remote interface

The server must contain a class that extends UnicastRemoteObject and implements the remote interface. This class can also have additional methods, but only the methods in the remote interface will be available to the client, of course, since the client will get only a reference to the interface, not the class that implements it.

You must explicitly define the constructor for the remote object even if you’re only defining a default constructor that calls the base-class constructor. You must write it out since it must throw RemoteException.

Here’s the implementation of the remote interface PerfectTimeI:

```
//: c15:rmi:PerfectTime.java
// The implementation of
// the PerfectTime remote object.
package c15.rmi;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.net.*;

public class PerfectTime
    extends UnicastRemoteObject
    implements PerfectTimeI {
  // Implementation of the interface:
  public long getPerfectTime()
      throws RemoteException {
    return System.currentTimeMillis();
  }
  // Must implement constructor
  // to throw RemoteException:
  public PerfectTime() throws RemoteException {
    // super(); // Called automatically
  }
  // Registration for RMI serving:
  public static void main(String[] args) {
    System.setSecurityManager(
      new RMISecurityManager());
    try {
      PerfectTime pt = new PerfectTime();
      Naming.bind(
        "//peppy:2005/PerfectTime", pt);
      System.out.println("Ready to do time");
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
} ///:~
```

Here, main( ) handles all the details of setting up the server. When you're serving RMI objects, at some point in your program you must:

Create and install a security manager that supports RMI. The only one available for RMI as part of the Java distribution is RMISecurityManager.

Create one or more instances of a remote object. Here, you can see the creation of the PerfectTime object.

Register at least one of the remote objects with the RMI remote object registry for bootstrapping purposes. One remote object can have methods that produce references to other remote objects. This allows you to set it up so the client must go to the registry only once, to get the first remote object.

Setting up the registry

Here, you see a call to the static method Naming.bind( ). However, this call requires that the registry be running as a separate process on the computer. The name of the registry server is rmiregistry, and under 32-bit Windows you say:

```
start rmiregistry
```

to start it in the background. On Unix, it is:

```
rmiregistry &
```

Like many network programs, the rmiregistry is located at the IP address of whatever machine started it up, but it must also be listening at a port. If you invoke the rmiregistry as above, with no argument, the registry's port will default to 1099. If you want it to be at some other port, you add an argument on the command line to specify the port. For this example, the port will be located at 2005, so the rmiregistry should be started like this under 32-bit Windows:

```
start rmiregistry 2005
```

or for Unix:

```
rmiregistry 2005 &
```

The information about the port must also be given to the bind( ) command, as well as the IP address of the machine where the registry is located. But this brings up what can be a frustrating problem if you're expecting to test RMI programs locally the way the network programs have been tested so far in this chapter. In the JDK 1.1.1 release, there are a couple of problems:[69]

localhost does not work with RMI. Thus, to experiment with RMI on a single machine, you must provide the name of the machine. To find out the name of your machine under 32-bit Windows, go to the control panel and select "Network." Select the "Identification" tab, and you'll see your computer name. In my case, I called my computer "Peppy." It appears that capitalization is ignored.

RMI will not work unless your computer has an active TCP/IP connection, even if all your components are just talking to each other on the local machine. This means that you must connect to your Internet service provider before trying to run the program or you'll get some obscure exception messages.

With all this in mind, the bind( ) command becomes:

```
Naming.bind("//peppy:2005/PerfectTime", pt);
```

If you are using the default port 1099, you don't need to specify a port, so you could say:

```
Naming.bind("//peppy/PerfectTime", pt);
```

You should be able to perform local testing by leaving off the IP address and using only the identifier:

```
Naming.bind("PerfectTime", pt);
```

The name for the service is arbitrary; it happens to be PerfectTime here, just like the name of the class, but you could call it anything you want. The important thing is that it's a unique name in the registry that the client knows to look for to procure the remote object. If the name is already in the registry, you'll get an AlreadyBoundException. To prevent this, you can always use rebind( ) instead of bind( ), since rebind( ) either adds a new entry or replaces the one that's already there.

Even though main( ) exits, your object has been created and registered so it's kept alive by the registry, waiting for a client to come along and request it. As long as the rmiregistry is running and you don't call Naming.unbind( ) on your name, the object will be there. For this reason, when you're developing your code you need to shut down the rmiregistry and restart it when you compile a new version of your remote object.

You aren't forced to start up rmiregistry as an external process. If you know that your application is the only one that's going to use the registry, you can start it up inside your program with the line:

```
LocateRegistry.createRegistry(2005);
```

Like before, 2005 is the port number we happen to be using in this example. This is the equivalent of running rmiregistry 2005 from a command line, but it can often be more convenient when you're developing RMI code since it eliminates the extra steps of starting and stopping the registry. Once you've executed this code, you can bind( ) using Naming as before.

Creating stubs and skeletons

If you compile and run PerfectTime.java, it won't work even if you have the rmiregistry running correctly. That's because the framework for RMI isn't all there yet.

You must first create the stubs and skeletons that provide the network connection operations and allow you to pretend that the remote object is just another local object on your machine.

What's going on behind the scenes is complex. Any objects that you pass into or return from a remote object must implement Serializable (if you want to pass remote references instead of the entire objects, the object arguments can implement Remote), so you can imagine that the stubs and skeletons are automatically performing serialization and deserialization as they "marshal" all of the arguments across the network and return the result. Fortunately, you don't have to know any of this, but you do have to create the stubs and skeletons. This is a simple process: you invoke the rmic tool on your compiled code, and it creates the necessary files. So the only requirement is that another step be added to your compilation process.

The rmic tool is particular about packages and classpaths, however. PerfectTime.java is in the package c15.rmi, and even if you invoke rmic in the same directory in which PerfectTime.class is located, rmic won't find the file, since it searches the classpath. So you must specify the location off the class path, like so:

```
rmic c15.rmi.PerfectTime
```

You don't have to be in the directory containing PerfectTime.class when you execute this command, but the results will be placed in the current directory.

When rmic runs successfully, you'll have two new classes in the directory:

```
PerfectTime_Stub.class
PerfectTime_Skel.class
```

corresponding to the stub and skeleton. Now you're ready to get the server and client to talk to each other.

Using the remote object

The whole point of RMI is to make the use of remote objects simple. The only extra thing that you must do in your client program is to look up and fetch the remote interface from the server. From then on, it's just regular Java programming: sending messages to objects. Here's the program that uses PerfectTime:

```
//: c15:rmi:DisplayPerfectTime.java
```

```
// Uses remote object PerfectTime.
package c15.rmi;
import java.rmi.*;
import java.rmi.registry.*;

public class DisplayPerfectTime {
  public static void main(String[] args) {
    System.setSecurityManager(
      new RMISecurityManager());
    try {
      PerfectTimeI t =
        (PerfectTimeI)Naming.lookup(
          "//peppy:2005/PerfectTime");
      for(int i = 0; i < 10; i++)
        System.out.println("Perfect time = " +
          t.getPerfectTime());
    } catch(Exception e) {
      e.printStackTrace();
    }
  }
} ///:~
```

The ID string is the same as the one used to register the object with Naming, and the first part represents the URL and port number. Since you're using a URL, you can also specify a machine on the Internet.

What comes back from Naming.lookup( ) must be cast to the remote interface, not to the class. If you use the class instead, you'll get an exception.

You can see in the method call

t.getPerfectTime( )

that once you have a reference to the remote object, programming with it is indistinguishable from programming with a local object (with one difference: remote methods throw RemoteException).

Introduction to CORBA

In large, distributed applications, your needs might not be satisfied by the preceding approaches. For example, you might want to interface with legacy datastores, or you might need services from a server object regardless of its physical location. These situations require some form of Remote Procedure Call (RPC), and possibly language independence. This is where CORBA can help.

CORBA is not a language feature; it's an integration technology. It's a specification that vendors can follow to implement CORBA-compliant integration products. CORBA is part of the OMG's effort to define a standard framework for distributed, language-independent object interoperability.

CORBA supplies the ability to make remote procedure calls into Java objects and non-Java objects, and to interface with legacy systems in a location-transparent way. Java adds networking support and a nice object-oriented language for building graphical and non-graphical applications. The Java and OMG object model map nicely to each other; for example, both Java and CORBA implement the interface concept and a reference object model.

CORBA Fundamentals

The object interoperability specification developed by the OMG is commonly referred to as the Object Management Architecture (OMA). The OMA defines two components: the Core Object Model and the OMA Reference Architecture. The Core Object Model states the basic concepts of object, interface, operation, and so on. (CORBA is a refinement of the Core Object Model.) The OMA Reference Architecture defines an underlying infrastructure of services and mechanisms that allow objects to interoperate. The OMA Reference Architecture includes the Object Request Broker (ORB), Object Services (also known as CORBAservices), and common facilities.

The ORB is the communication bus by which objects can request services from other objects, regardless of their physical location. This means that what looks like a method call in the client code is actually a complex operation. First, a connection with the server object must exist, and to create a connection the ORB must know where the server implementation code resides. Once the connection is established, the method arguments must be marshaled, i.e. converted in a binary stream to be sent across a network. Other information that must be sent are the server machine name, the server process, and the identity of the server object inside that process. Finally, this information is sent through a low-level wire protocol, the information is decoded on the server side, and the call is executed. The ORB hides all of this complexity from the programmer and makes the operation almost as simple as calling a method on local object.

There is no specification for how an ORB Core should be implemented, but to provide a basic compatibility among different vendors' ORBs, the OMG defines a set of services that are accessible through standard interfaces.

CORBA Interface Definition Language (IDL)

CORBA is designed for language transparency: a client object can call methods on a server object of different class, regardless of the language they are implemented with. Of course, the client object must know the names and signatures of methods that the server object exposes. This is where IDL comes in. The CORBA IDL is a language-neutral way to specify data types, attributes, operations, interfaces, and more. The IDL syntax is similar to the C++ or Java syntax. The following table shows the correspondence between some of the concepts common to three languages that can be specified through CORBA IDL:

CORBA IDL	Java	C++
Module	Package	Namespace
Interface	Interface	Pure abstract class
Method	Method	Member function

The inheritance concept is supported as well, using the colon operator as in C++. The programmer writes an IDL description of the attributes, methods, and interfaces that will be implemented and used by the server and clients. The IDL is then compiled by a vendor-provided IDL/Java compiler, which reads the IDL source and generates Java code.

The IDL compiler is an extremely useful tool: it doesn't just generate a Java source equivalent of the IDL, it also generates the code that will be used to marshal method arguments and to make remote calls. This code, called the stub and skeleton code, is organized in multiple Java source files and is usually part of the same Java package.

The naming service

The naming service is one of the fundamental CORBA services. A CORBA object is accessed through a reference, a piece of information that's not meaningful for the human reader. But references can be assigned programmer-defined, string names. This operation is known as stringifying the reference, and one of the OMA components, the Naming Service, is devoted to performing string-to-object and object-to-string conversion and mapping. Since the Naming Service acts as a telephone directory that both servers and clients can consult and manipulate, it runs as a separate process. Creating an object-to-string mapping is called binding an object, and removing the mapping is called unbinding. Getting an object reference passing a string is called resolving the name.

For example, on startup, a server application could create a server object, bind the object into the name service, and then wait for clients to make requests. A client first obtains a server object reference, resolving the string name, and then can make calls into the server using the reference.

Again, the Naming Service specification is part of CORBA, but the application that implements it is provided by the ORB vendor. The way you get access to the Naming Service functionality can vary from vendor to vendor.

An example

The code shown here will not be elaborate because different ORBs have different ways to access CORBA services, so examples are vendor specific. (The example below uses JavaIDL, a free product from Sun that comes with a light-weight ORB, a naming service, and an IDL-to-Java compiler.) In addition, since Java is young and still evolving, not all CORBA features are present in the various Java/CORBA products.

We want to implement a server, running on some machine, that can be queried for the exact time. We also want to implement a client that asks for the exact time. In this case we'll be implementing both programs in Java, but we could also use two different languages (which often happens in real situations).

Writing the IDL source

The first step is to write an IDL description of the services provided. This is usually done by the server programmer, who is then free to implement the server in any language in which a CORBA IDL compiler exists. The IDL file is distributed to the client side programmer and becomes the bridge between languages.

The example below shows the IDL description of our ExactTime server:

```
//: c15:corba:ExactTime.idl
//# You must install idltojava.exe from
//# java.sun.com and adjust the settings to use
//# your local C preprocessor in order to compile
//# This file. See docs at java.sun.com.
module remotetime {
    interface ExactTime {
        string getTime();
    };
}; ///:~
```

This is a declaration of the ExactTime interface inside the remotetime namespace. The interface is made up of one single method that gives back the current time in string format.

Creating stubs and skeletons

The second step is to compile the IDL to create the Java stub and skeleton code that we'll use for implementing the client and the server. The tool that comes with the JavaIDL product is idltojava:

```
idltojava remotetime.idl
```

This will automatically generate code for both the stub and the skeleton. Idltojava generates a Java package named after the IDL module, remotetime, and the generated

Java files are put in the remotetime subdirectory. _ExactTimeImplBase.java is the skeleton that we'll use to implement the server object, and _ExactTimeStub.java will be used for the client. There are Java representations of the IDL interface in ExactTime.java and a couple of other support files used, for example, to facilitate access to the naming service operations.

Implementing the server and the client

Below you can see the code for the server side. The server object implementation is in the ExactTimeServer class. The RemoteTimeServer is the application that creates a server object, registers it with the ORB, gives a name to the object reference, and then sits quietly waiting for client requests.

```
//: c15:corba:RemoteTimeServer.java
import remotetime.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import java.util.*;
import java.text.*;

// Server object implementation
class ExactTimeServer extends _ExactTimeImplBase {
  public String getTime(){
    return DateFormat.
        getTimeInstance(DateFormat.FULL).
          format(new Date(
              System.currentTimeMillis()));
  }
}

// Remote application implementation
public class RemoteTimeServer {
  public static void main(String[] args) {
    try {
      // ORB creation and initialization:
      ORB orb = ORB.init(args, null);
      // Create the server object and register it:
      ExactTimeServer timeServerObjRef =
        new ExactTimeServer();
      orb.connect(timeServerObjRef);
      // Get the root naming context:
      org.omg.CORBA.Object objRef =
        orb.resolve_initial_references(
          "NameService");
      NamingContext ncRef =
      NamingContextHelper.narrow(objRef);
      // Assign a string name to the
      // object reference (binding):
      NameComponent nc =
        new NameComponent("ExactTime", "");
      NameComponent[] path = { nc };
      ncRef.rebind(path, timeServerObjRef);
      // Wait for client requests:
      java.lang.Object sync =
        new java.lang.Object();
      synchronized(sync){
        sync.wait();
      }
    }
    catch (Exception e) {
      System.out.println(
         "Remote Time server error: " + e);
      e.printStackTrace(System.out);
    }
  }
} ///:~
```

As you can see, implementing the server object is simple; it's a regular Java class that inherits from the skeleton code generated by the IDL compiler. Things get a bit more complicated when it comes to interacting with the ORB and other CORBA services.

Some CORBA services

This is a short description of what the JavaIDL-related code is doing (primarily ignoring the part of the CORBA code that is vendor dependent). The first line in main( ) starts up the ORB, and of course, this is because our server object will need to interact with it. Right after the ORB initialization, a server object is created. Actually, the right term would be a transient servant object: an object that receives requests from clients, and whose lifetime is the same as the process that creates it. Once the transient servant object is created, it is registered with the ORB, which means that the ORB knows of its existence and can now forward requests to it.

Up to this point, all we have is timeServerObjRef, an object reference that is known only inside the current server process. The next step will be to assign a stringified name to this servant object; clients will use that name to locate the servant object. We accomplish this operation using the Naming Service. First, we need an object reference to the Naming Service; the call to resolve_initial_references( ) takes the stringified object reference of the Naming Service that is “NameService,” in JavaIDL, and returns an object reference. This is cast to a specific NamingContext reference using the narrow( ) method. We can use now the naming services.

To bind the servant object with a stringified object reference, we first create a NameComponent object, initialized with “ExactTime,” the name string we want to bind to the servant object. Then, using the rebind( ) method, the stringified reference is bound to the object reference. We use rebind( ) to assign a reference, even if it already exists, whereas bind( ) raises an exception if the reference already exists. A name is made up in CORBA by a sequence of NameContexts—that’s why we use an array to bind the name to the object reference.

The servant object is finally ready for use by clients. At this point, the server process enters a wait state. Again, this is because it is a transient servant, so its lifetime is confined to the server process. JavaIDL does not currently support persistent objects—objects that survive the execution of the process that creates them.

Now that we have an idea of what the server code is doing, let’s look at the client code:

```
//: c15:corba:RemoteTimeClient.java
import remotetime.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class RemoteTimeClient {
  public static void main(String[] args) {
    try {
      // ORB creation and initialization:
      ORB orb = ORB.init(args, null);
      // Get the root naming context:
      org.omg.CORBA.Object objRef =
        orb.resolve_initial_references(
          "NameService");
      NamingContext ncRef =
        NamingContextHelper.narrow(objRef);
      // Get (resolve) the stringified object
      // reference for the time server:
      NameComponent nc =
        new NameComponent("ExactTime", "");
      NameComponent[] path = { nc };
      ExactTime timeObjRef =
        ExactTimeHelper.narrow(
          ncRef.resolve(path));
      // Make requests to the server object:
      String exactTime = timeObjRef.getTime();
      System.out.println(exactTime);
    } catch (Exception e) {
      System.out.println(
         "Remote Time server error: " + e);
      e.printStackTrace(System.out);
    }
  }
} ///:~
```

The first few lines do the same as they do in the server process: the ORB is initialized and a reference to the naming service server is resolved. Next, we need an object reference for the servant object, so we pass the stringified object reference to the resolve( ) method, and we cast the result into an ExactTime interface reference using the narrow( ) method. Finally, we call getTime( ).

Activating the name service process

Finally we have a server and a client application ready to interoperate. You've seen that both need the naming service to bind and resolve stringified object references. You must start the naming service process before running either the server or the client. In JavaIDL, the naming service is a Java application that comes with the product package, but it can be different with other products. The JavaIDL naming service runs inside an instance of the JVM and listens by default to network port 900.

Activating the server and the client

Now you are ready to start your server and client application (in this order, since our server is transient). If everything is set up correctly, what you'll get is a single output line on the client console window, giving you the current time. Of course, this might be not very exciting by itself, but you should take one thing into account: even if they are on the same physical machine, the client and the server application are running inside different virtual machines and they can communicate via an underlying integration layer, the ORB and the Naming Service.

This is a simple example, designed to work without a network, but an ORB is usually configured for location transparency. When the server and the client are on different machines, the ORB can resolve remote stringified references using a component known as the Implementation Repository. Although the Implementation Repository is part of CORBA, there is almost no specification, so it differs from vendor to vendor.

As you can see, there is much more to CORBA than what has been covered here, but you should get the basic idea. If you want more information about CORBA, the place to start is the OMG Web site, at http://www.omg.org. There you'll find documentation, white papers, proceedings, and references to other CORBA sources and products.

Java Applets and CORBA

Java applets can act as CORBA clients. This way, an applet can access remote information and services exposed as CORBA objects. But an applet can connect only with the server from which it was downloaded, so all the CORBA objects the applet interacts with must be on that server. This is the opposite of what CORBA tries to do: give you complete location transparency.

This is an issue of network security. If you're on an intranet, one solution is to loosen the security restrictions on the browser. Or, set up a firewall policy for connecting with external servers.

Some Java ORB products offer proprietary solutions to this problem. For example, some implement what is called HTTP Tunneling, while others have their special firewall features.

This is too complex a topic to be covered in an appendix, but it is definitely something you should be aware of.

CORBA vs. RMI

You saw that one of the main CORBA features is RPC support, which allows your local objects to call methods in remote objects. Of course, there already is a native Java feature that does exactly the same thing: RMI (see Chapter 15). While RMI makes RPC possible between Java objects, CORBA makes RPC possible between objects implemented in any language. It's a big difference.

However, RMI can be used to call services on remote, non-Java code. All you need is some kind of wrapper Java object around the non-Java code on the server side. The wrapper object connects externally to Java clients via RMI, and internally connects to the non-Java code using one of the techniques shown above, such as JNI or J/Direct.

This approach requires you to write a kind of integration layer, which is exactly what CORBA does for you, but then you don't need a third-party ORB.

Enterprise Java Beans

[70] By now, you've been introduced to CORBA and RMI. But could you imagine trying to develop a large-scale application using CORBA and/or RMI? Your manager has asked you to develop a multi-tiered application to view and update records in a database through a Web interface. You sit back and think of what that really means. Sure, you can write a database application using JDBC, a Web interface using JSP/Servlets, and a distributed system using CORBA/RMI. But what extra considerations must you make when developing a distributed object based system rather than just knowing API's? Here are the issues:

Performance: The new distributed objects that you are creating are going to have to perform, as they potentially could service many clients at a time. So you'll want to think of optimization techniques such as caching, pooling of resources (e.g., JDBC database connections). You'll also have to manage the lifecycle of your distributed object.

Scalability: The distributed objects must also be scalable. Scalable in a distributed application sense means that the number of instances of your distributed object can be increased and moved over to a different machine without the modification of any code. Take for example a system that you develop internally as a small lookup of clients inside your organization from a database. The application works well when you use it, but your manager has seen it and has said "Robert, that is an excellent system, get it on our public Web-site now!!!"-Will my distributed object be able to handle the load of a potentially limitless demand?

Security: Does my distributed object manage the authorization of the clients that accesses it? Can I add new users and roles to it without recompilation?

Distributed Transactions: Can my distributed object reference distributed-transactions transparently? Can I update my Oracle and Sybase databases simultaneously within the same transaction and roll them both back if a certain criteria is not met?

Reusability: Have I created my distributed object so that I can move it into another vendors' application server? Can I resell my distributed object (component) to somebody else? Can I buy somebody else's component and use it without having to recompile and 'hack it into shape'?

Availability: If one of the machines in my system was to go down, are my clients able to automatically fail-over to back up copies of my objects running on other machines?

As you can see from the above, the considerations that a developer must make when developing a distributed system and we haven't even mentioned solving the problem that we were originally trying to solve!

So you now have your list of extra problems that you must solve. So how do you go about doing it? Surely somebody must have done this before? Could I use some well-known design patterns to help me solve these problems? Then an idea flashes in your head... "I could create a framework that handles all of these issues and write my components on top of the framework!".... This is where Enterprise JavaBeans comes into the picture.

Sun, along with other leading distributed object vendors (listed at URL HERE!) realized that sooner or later every development team would be reinventing the wheel. So they created the Enterprise JavaBeans specification (EJB). EJB is a specification for a server-side component model that tackles all of the considerations mentioned above using a defined, standard approach that allows developers to create components-which are actually called Enterprise JavaBeans (EJB's)-that are isolated from low-level 'plumbing' code and focus solely on providing business logic. Because EJB's are defined as a standard they can be used without being vendor dependent.

What’s defined in the EJB specification?

The Enterprise JavaBeans specification, currently at version 1.1 - public release 2 - defines a server side component model. It defines 6 roles that are used to perform the tasks in development and deployment as well as defining the components of the system.

Roles

The EJB specification defines roles that are used during in the development, deployment and running of a distributed system. Vendors, administrators and developers play the various roles. They allow the partitioning of technical and domain knowledge. This allows the vendor to provide a technically sound framework and the developers to create domain specific components e.g., an Accounting component. The same party can perform one or many roles. The roles defined in the EJB specification have been summarized in the following table:

Role	Responsibility
Enterprise Bean Provider	The developer who is responsible for creating reusable EJB components. These components are packaged into a special jar file (ejb-jar file).
Application Assembler	Create and assemble applications from a collection of ejb-jar files. This includes writing applications that utilize the collection of EJB's (e.g., Servlets, JSP, Swing etc. etc.)
Deployer	The Deployer's role of the to take the collection of ejb-jar files from the Assembler and/or Bean Provider and deploy them into a run-time environment - one or many EJB Container(s).
EJB Container/Server Provider	Provide a run-time environment and tools that are used to deploy, administer and "run" EJB components.
System Administrator	

Over see the most important goal of the entire system - That it is up and running. Management of a distributed application can consist of many different components and services all configured and interacting together correctly.

Components of EJB

EJB components are reusable business logic. EJB components adhere to strict standards and design patterns as defined in the EJB specification. This allows the components to be portable and also allow other services—such as security, caching and distributed transactions - to be performed on the components' behalf. An Enterprise Bean Provider is responsible for developing EJB components. The internals of an EJB component are covered in - "What makes up an EJB component?"

EJB Container

The EJB Container is a run-time environment that contains -or runs—EJB components and provides a set of standard services to the components. The EJB Containers responsibilities are tightly defined by the specification to allow for vendor neutrality. The EJB container provides the low-level "plumbing" of EJB, including distributed transactions, security, life cycle management of beans, caching, threading and session management. The EJB Container Provider is responsible for providing an **EJB Container.**

EJB Server

An EJB Server is defined as an Application Server that contains and runs 1 or more EJB Containers. that both the Container and Server are the same vendor. The EJB Server Provider is responsible for providing an EJB Server. The specification suggests and you can assume for this introduction, that the EJB Container and EJB Server are the same.

Java Naming and Directory Interface (JNDI)

Java Naming and Directory Interface (JNDI) is used in Enterprise JavaBeans as the naming service for EJB Components on the network and other container services such as transactions. JNDI maps very closely to other naming and directory standards such as CORBA CosNaming and can actually be implemeted as a wrapper on top of it.

Java Transaction API / Java Transaction Service (JTA/JTS)

JTA/JTS is used in Enterprise JavaBeans as the transactional API. An Enterprise Bean Provider can use the JTS to create transactional code although the EJB Container commonly implements transactions in EJB on the EJB components' behalf. The Deployer can define the transactional attributes of an EJB component at deployment time. The EJB Container is responsible for handling the transaction whether it is local or distributed. The JTS specification is the Java mapping to the CORBA OTS (Object Transaction Service)

CORBA and RMI/IIOP

The EJB specification defines interoperablility with CORBA. The 1.1 specification quotes "The Enterprise JavaBeans architecture will be compatible with the CORBA protocols." CORBA interoperability is achieved through the mapping of EJB services such as JTS and JNDI to corresponding CORBA services and the implementation of RMI on top of the CORBA protocol IIOP.

Use of CORBA and RMI/IIOP in Enterprise JavaBeans is implemented in the EJB Container and is the responsibility of the EJB Container provider. Use of CORBA and RMI/IIOP in the EJB Container is hidden from the EJB Component itself. This means that the Enterprise Bean Provider can write their EJB Component and deploy it into any EJB Container without any regard of which communication protocol is being used.

The components and available services of EJB.

What makes up an EJB component?

Enterprise Bean

The Enterprise Bean is a Java class that the Enterprise Bean Provider develops. It implements an EnterpriseBean interface (more detail in a later section) and provides the implementation of the business methods that the component is to perform. The class does not implement any authorization/authentication code, multithreading, transactional code.

Home Interface

Every Enterprise Bean that is created must have an associated Home interface. The Home interface is used as a factory for your EJB. Clients use the Home interface to find an instance of your EJB or create a new instance of your EJB.

Remote Interface

The Remote interface is a Java Interface that reflects the methods of your Enterprise Bean that you wish to expose to the outside world. The Remote interface plays a similar role to a CORBA IDL interface.

Deployment Descriptor

The Deployment Descriptor is an XML file that contains information about your EJB. Using XML allows the Deployer to easily change attributes about your EJB. The configurable attributes defined in the Deployment Descriptor include:

The Home and Remote interface names that are required by your EJB

The name to publish into JNDI for your EJB's Home interface

Transactional attributes for each method of your EJB

Access Control Lists for authentication

EJB-Jar File

The EJB-Jar file is a normal java jar file that contains your EJB, Home and Remote interfaces, as well as the Deployment Descriptor.

How does EJB work?

Now that we have our EJB-Jar file containing our Bean, Home and Remote interfaces and Deployment Descriptor, Let's take a look at how all of these pieces fit together and why Home and Remote interfaces are needed and how the EJB Container uses them.

Who implements the Home and Remote Interfaces?

The EJB Container implements the Home and Remote interfaces that are in our EJB-Jar file. Because the EJB Container implements the Home interface that - as mentioned earlier - provides methods to create and find your EJB. This means that the EJB Container is responsible for the lifecycle management of your EJB. This level of indirection allows for optimizations to occur. For example 5 clients simultaneously request to create an EJB through a Home Interface, the EJB Container could create only one and share that EJB between all 5 clients. This is achieved through the

Remote Interface, which is again implemented by the EJB Container. The implemented Remote object plays the role of a proxy object to the EJB.

The following diagram show the level of indirection achieved by this approach and that all calls to the EJB are ‘proxied’ through the EJB Container via the Home and Remote interfaces. This level of indirection is also the reason why the EJB Container can control security and transactional behavior.

Types of EJB’s

There should be one question in your head from the previous section, “Surely sharing the same EJB between clients can improve performance, but what If I want to maintain state on my server?”

The Enterprise JavaBeans specification defines different types of EJB’s that have different characteristics and exhibit different behavior. Two categories of EJB’s have been defined in the specification. Session Beans and Entity Beans, and each of these categories has variations. A hierarchy of the various types of EJB components is shown in the following figure.

Session Beans

Session Beans are used to represent Use-Cases or Workflow on behalf of a client. They represent operations on persistent data, not persistent data itself. There are two types of Session Beans, Stateless and Stateful. All Session Beans must implement the javax.ejb.SessionBean interface. The EJB Container governs the life of a Session Bean. If the EJB Container crashes, data for all Stateful Session Beans could be lost. Some high-end EJB Containers provide recovery for Stateful Session Beans.

Stateless Session Beans

Stateless Session Beans are the simplest type of EJB component to implement. They do not maintain any conversational state with clients between method invocations so they are easily reusable on the server side and because they can be cached, they scale well on demand. When using Stateless Session Beans, all state must be stored outside of the EJB.

Stateful Session Beans

Stateful Session Beans - as you could probably guess - maintain state between invocations. They have a 1 to 1 logical mapping to a client and can maintain state within themselves. The EJB Container is responsible for pooling and caching of Stateful Session Beans, which is achieved through Passivation and Activation.

Entity Beans

Entity Beans are components that represent persistent data and behavior of this data. Entity Beans can be shared amongst multiple clients, the same as data in a database. The EJB Container is responsible for caching Entity Beans and for maintaining the integrity of the Entity Beans. The life of an Entity Bean outlives the EJB Container, so if an EJB Container crashes, the Entity Bean is expected to still be available when the EJB Container becomes available.

There are two types of Entity Beans, those that have Bean-Managed persistence and Container Managed persistence.

Container Managed Persistence (CMP)

A CMP Entity Bean has its' persistence implemented on its behalf by the EJB Container. Through attributes specified in the Deployment Descriptor, the EJB Container will map the Entity Bean's attributes to some persistent store (usually -but not always—a database). CMP reduces the time to develop and dramatically reduces the amount of code required for the EJB.

Bean Managed Persistence (BMP)

A BMP Entity Bean has its' persistence implemented by the Enterprise Bean Provider. The Enterprise Bean Provider is responsible for implementing the logic required to create a new EJB, update some attributes of the EJBS, delete an EJB and find an EJB from persistent store. This usually involves writing JDBC code to interact with a database or other persistent store. With BMP, the developer is in full control of how the Entity Bean persistence is managed.

BMP also gives flexibility where a CMP implementation may not be available e.g., if you wanted to create an EJB that wrapped some code on an existing mainframe system, you could write your persistence using CORBA.

How do I put the 'E' in my existing JavaBeans?

There is much confusion about the relationship between the JavaBeans component model and the Enterprise JavaBeans specification. Whilst both the JavaBeans and Enterprise JavaBeans specifications share the same objectives in promoting reuse and portability of Java code between development and deployment tools with the use of standard design patterns, the motives behind each specification are geared to solve different problems.

The standards defined in the JavaBeans component model are designed for creating reusable components that are typically used in IDE development tools and are commonly, although not exclusively visual components.

The Enterprise JavaBeans specification defines a component model for developing server side java code. Because EJB's can potentially run on many different server-side platforms -including mainframes that do not have visual displays - An EJB cannot make use of the java.awt package.

Developing an Enterprise Java Bean

We will now implement the "Perfect Time" example from the previous RMI section as an Enterprise JavaBean component. The example will be a simple Stateless Session Bean. Enterprise JavaBean components will consist of at least one class and two interfaces.

The first interface defined is the Remote Interface to our Enterprise JavaBean component. When you create a Remote interface for an EJB , you must follow these guidelines:

The remote interface must be public.

The remote interface must extend the interface javax.ejb.EJBObject.

Each method in the remote interface must declare java.rmi.RemoteException in its throws clause in addition to any application-specific exceptions.

Any object passed as an argument or return value (either directly or embedded within a local object) must be a valid RMI-IIOP data type (this includes other EJB objects)

Here is the simple remote interface for our PerfectTime EJB:

```
//: c15:ejb:PerfectTime.java
//# You must install the J2EE Java Enterprise
//# Edition from java.sun.com and add j2ee.jar
//# To your CLASSPATH in order to compile
//# This file. See details at java.sun.com.
// Remote Interface of PerfectTimeBean
import java.rmi.*;
import javax.ejb.*;

public interface PerfectTime extends EJBObject {
  public long getPerfectTime()
    throws RemoteException;
} ///:~
```

The second interface defined is the Home Interface to our Enterprise JavaBean component. The Home interface is the factory where our component will be created. The Home interface can define create or finder methods. Create methods create instances of EJB's, finder methods locate existing EJB's and are used for Entity Beans only. When you create a Home interface for an EJB , you must follow these guidelines:

The Home interface must be public.

The Home interface must extend the interface javax.ejb.EJBHome.

Each method in the Home interface must declare java.rmi.RemoteException in its throws clause as well as a javax.ejb.CreateException

The return value of a create method must be a Remote Interface

The return value of a finder method (Entity Beans only) must be a Remote Interface or java.util.Enumeration or java.util.Collection

Any object passed as an argument (either directly or embedded within a local object) must be a valid RMI-IIOP data type (this includes other EJB objects)

The standard naming convention for Home interfaces is to take the Remote interface name and append "Home" to the end. Following is the Home interface for our PerfectTime EJB:

```
//: c15:ejb:PerfectTimeHome.java
// Home Interface of PerfectTimeBean.
import java.rmi.*;
import javax.ejb.*;

public interface PerfectTimeHome extends EJBHome {
  public PerfectTime create()
    throws CreateException, RemoteException;
} ///:~
```

Now that we have defined the interfaces of our component, we can now implement the business logic behind it. When you create your EJB implementation class, you must follow these guidelines, (note that you should consult the EJB specification for a complete list of guidelines when developing Enterprise JavaBeans):

The class must be public.

The class must implement an EJB interface (either javax.ejb.SessionBean or javax.ejb.EntityBean).

The class should define methods that map directly to the methods in the Remote interface, Note that the class does not implement the Remote interface, it mirrors the methods in the Remote interface but does NOT throw java.rmi.RemoteException.

Define one or more ejbCreate( ) methods to initialize your EJB.

The return value and arguments of all methods must be valid RMI-IIOP data types.

```
//: c15:ejb:PerfectTimeBean.java
// Simple Stateless Session Bean
// that returns current system time.
import java.rmi.*;
import javax.ejb.*;

public class PerfectTimeBean
   implements SessionBean {
   private SessionContext sessionContext;
   //return current time
   public long getPerfectTime() {
        return System.currentTimeMillis();
   }
   // EJB methods
   public void
   ejbCreate() throws CreateException {}
   public void ejbRemove() {}
   public void ejbActivate() {}
   public void ejbPassivate() {}
   public void
   setSessionContext(SessionContext ctx) {
      sessionContext = ctx;
   }
}///:~
```

Notice that the EJB methods (ejbCreate( ), ejbRemove( ), ejbActivate( ), ejbPassivate( )) are all empty. These methods are invoked by the EJB Container and are used to control the state of our component. As this is a simple example, we can leave these empty. The setSessionContext( ) method passes a javax.ejb.SessionContext object which contains information about context that the component is in, such as the current transaction and security information.

After we have created our Enterprise Java Bean, we then need to create a Deployment Descriptor. In EJB 1.1, the Deployment Descriptor is an XML file that describes the EJB component. The Deployment Descriptor should be stored in a file called ejb-jar.xml.

```
<?xml version="1.0" encoding="Cp1252"?>
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 1.1//EN' 'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>

<ejb-jar>
  <description>example for chapter 15</description>
  <display-name></display-name>
  <small-icon></small-icon>
  <large-icon></large-icon>
  <enterprise-beans>
    <session>
      <ejb-name>PerfectTime</ejb-name>
      <home>PerfectTimeHome</home>
      <remote>PerfectTime</remote>
      <ejb-class>PerfectTimeBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  <ejb-client-jar></ejb-client-jar>
</ejb-jar>
```

Inside the <session> tag of our deployment descriptor we can see that our Component, the Remote interface and Home interfaces are begin defined. Deployment Descriptors can easily be automatically generated inside tools such as JBuilder.

Along with the standard ejb-jar.xml deployment descriptor the EJB 1.1 specification states that any vendor specific tags should be stored in a separate file, this is to achieve high portability between components and different brands EJB containers.

Now that we have created our component, and defined it’s composition in the Deployment Descriptor we then need to archive the files inside a standard Java Archive (JAR) file. The Deployment Descriptors should be placed inside the /META-INF sub-directory of the Jar file.

Once we have defined our EJB component in the Deployment Descriptor, the Deployer should then deploy the EJB component into the EJB Container. At the time of writing, this process was quite "gui intensive" and specific to each individual EJB Container, so we decided not to document the entire deployment process in this overview. Every EJB Container, however will have a documented process for deploying an EJB.

Because an EJB component is a distributed object, the deployment process should also create some client stubs for calling the EJB component. These classes whould be placed on the classpath of the client application. Because EJB components can be implemented on top of RMI-IIOP (CORBA) or RMI-JRMP, the stubs generated could vary between EJB Containers, nevertheless they are generated classes.

When a client program wishes to invoke an EJB, it must lookup the EJB component inside JNDI and obtain a reference to the Home interface of the EJB component. The Home Interface can then be invoked to create an instance of the EJB, which can then be invoked.

In this example the Client program is a simple Java program, but you should remember that it could just as easily be a Servlet, a JSP even a CORBA or RMI distributed object.

The PerfectTimeClient code is as follows.

```
//: c15:ejb:PerfectTimeClient.java
// Client program for PerfectTimeBean

public class PerfectTimeClient {
public static void
main(String[] args) throws Exception {
  // Get a JNDI context using the
  // JNDI Naming service:
  javax.naming.Context context =
    new javax.naming.InitialContext();
  // Look up the home interface in the
  // JNDI Naming service:
  Object ref = context.lookup("perfectTime");
```

```
    // Cast the remote object to the home interface:
    PerfectTimeHome home = (PerfectTimeHome)
      javax.rmi.PortableRemoteObject.narrow(
        ref, PerfectTimeHome.class);
    // Create a remote object from the home interface:
    PerfectTime pt = home.create();
    // Invoke  getPerfectTime()
    System.out.println(
      "Perfect Time EJB invoked, time is: " +
      pt.getPerfectTime() );
    }
  } ///:~
```

In Summary

The Enterprise JavaBeans specification - although intially seems very daunting - is a dramatic step forward in the standardization and simplification or distributed object computing. It is a major piece of the Java 2, Enterprise Edition Platform and is receiving much support from the Distributed Object community. Many tools are currently available or will be in the near future to help accelerate the development of EJB components.

This overview was aimed at giving you a brief tour as to what EJB is all about. For more information about the Enterprise JavaBeans specification you should see the Official Enterprise JavaBeans Home Page at http://java.sun.com/products/ejb/. Here you can download the latest specification as well as the Java 2, Enterprise Edition Reference Implementation, which you can use to develop and deploy your own EJB components.

Jini: distributed services

This section[71] gives an overview of Sun Microsystems's Jini technology. It describes some Jini nuts and bolts and shows how Jini's architecture helps to raise the level of abstraction in distributed systems programming, effectively turning network programming into object-oriented programming.

Jini in context

Traditionally, operating systems have been designed with the assumption that a computer will have a processor, some memory, and a disk. When you boot a computer, the first thing it does is look for a disk. If it doesn't find a disk, it can't function as a computer. Increasingly, however, computers are appearing in a different guise: as embedded devices that have a processor, some memory, and a network connection—but no disk. The first thing a cellphone does when you boot it up, for example, is look for the telephone network. If it doesn't find the network, it can't function as a cellphone. This trend in the hardware environment, from disk-centric to network-centric, will affect how we organize our software—and that's where Jini comes in.

Jini is an attempt to rethink computer architecture, given the rising importance of the network and the proliferation of processors in devices that have no disk drive. These devices, which will come from many different vendors, will need to interact over a network. The network itself will be very dynamic—devices and services will be added and removed regularly. Jini provides mechanisms to enable smooth adding, removal, and finding of devices and services on the network. In addition, Jini provides a programming model that makes it easier for programmers to get their devices talking to each other.

Building on top of Java, object serialization, and RMI (which enable objects to move around the network from virtual machine to virtual machine) Jini attempts to extend the benefits of object-oriented programming to the network. Instead of requiring device vendors to agree on the network protocols through which their devices can interact, Jini enables the devices to talk to each other through interfaces to objects.

What is Jini?

Jini is a set of APIs and network protocols that can help you build and deploy distributed systems that are organized as federations of services. A service can be anything that sits on the network and is ready to perform a useful function. Hardware devices, software, communications channels—even human users themselves—can be services. A Jini-enabled disk drive, for example, could offer a “storage” service. A Jini-enabled printer could offer a “printing” service. A federation of services, then, is a set of services, currently available on the network, that a client (meaning a program, service, or user) can bring together to help it accomplish some goal.

To perform a task, a client enlists the help of services. For example, a client program might upload pictures from the image storage service in a digital camera, download the pictures to a persistent storage service offered by a disk drive, and send a page of thumbnail-sized versions of the images to the printing service of a color printer. In this example, the client program builds a distributed system consisting of itself, the image storage service, the persistent storage service, and the color-printing service. The client and services of this distributed system work together to perform the task: to offload and store images from a digital camera and print a page of thumbnails.

The idea behind the word federation is that the Jini view of the network doesn't involve a central controlling authority. Because no one service is in charge, the set of all services available on the network form a federation—a group composed of equal peers. Instead of a central authority, Jini's run-time infrastructure merely provides a way for clients and services to find each other (via a lookup service, which stores a directory of currently available services). After services locate each other, they are on their own. The client and its enlisted services perform their task independently of the Jini run-time infrastructure. If the Jini lookup service crashes, any distributed systems brought together via the lookup service before it crashed can continue their work. Jini even includes a network protocol that clients can use to find services in the absence of a lookup service.

How Jini works

Jini defines a run-time infrastructure that resides on the network and provides mechanisms that enable you to add, remove, locate, and access services. The run-time infrastructure resides in three places: in lookup services that sit on the network, in the service providers (such as Jini-enabled devices), and in clients. Lookup services are the central organizing mechanism for Jini-based systems. When new services become available on the network, they register themselves with a lookup service. When clients wish to locate a service to assist with some task, they consult a lookup service.

The run-time infrastructure uses one network-level protocol, called discovery, and two object-level protocols, called join and lookup. Discovery enables clients and services to locate lookup services. Join enables a service to register itself in a lookup service. Lookup enables a client to query for services that can help accomplish its goals.

The discovery process

Discovery works like this: Imagine you have a Jini-enabled disk drive that offers a persistent storage service. As soon as you connect the drive to the network, it broadcasts a presence announcement by dropping a multicast packet onto a well-known port. Included in the presence announcement is an IP address and port number where the disk drive can be contacted by a lookup service.

Lookup services monitor the well-known port for presence announcement packets. When a lookup service receives a presence announcement, it opens and inspects the packet. The packet contains information that enables the lookup service to determine whether or not it should contact the sender of the packet. If so, it contacts the sender directly by making a TCP connection to the IP address and port number extracted from the packet. Using RMI, the lookup service sends an object, called a service registrar, across the network to the originator of the packet. The purpose of the service registrar object is to facilitate further communication with the lookup service. By invoking methods on this object, the sender of the announcement packet can perform join and lookup on the lookup service. In the case of the disk drive, the lookup service would make a TCP connection to the disk drive and would send it a service registrar object, through which the disk drive would then register its persistent storage service via the join process.

The join process

Once a service provider has a service registrar object, the end product of discovery, it is ready to do a join—to become part of the federation of services that are registered in the lookup service. To do a join, the service provider invokes the register( ) method on the service registrar object, passing as a parameter an object called a service item, a bundle of objects that describe the service. The register( ) method sends a copy of the service item up to the lookup service, where the service item is stored. Once this has completed, the service provider has finished the join process: its service has become registered in the lookup service.

The service item is a container for several objects, including an object called a service object, which clients can use to interact with the service. The service item can also include any number of attributes, which can be any object. Some potential attributes are icons, classes that provide GUIs for the service, and objects that give more information about the service.

Service objects usually implement one or more interfaces through which clients interact with the service. For example, a lookup service is a Jini service, and its service object is the service registrar. The register( ) method invoked by service providers during join is declared in the ServiceRegistrar interface (a member of the net.jini.core.lookup package), which all service registrar objects implement. Clients and service providers talk to the lookup service through the service registrar object by invoking methods declared in the ServiceRegistrar interface. Likewise, a disk drive would provide a service object that implemented some well-known storage service interface. Clients would look up and interact with the disk drive by this storage service interface.

The lookup process

Once a service has registered with a lookup service via the join process, that service is available for use by clients who query that lookup service. To build a distributed system of services that will work together to perform some task, a client must locate and enlist the help of the individual services. To find a service, clients query lookup services via a process called lookup.

To perform a lookup, a client invokes the lookup( ) method on a service registrar object. (A client, like a service provider, gets a service registrar through the previously-described process of discovery.) The client passes as an argument to lookup( ) a service template, an object that serves as search criteria for the query. The service template can include a reference to an array of Class objects. These Class objects indicate to the lookup service the Java type (or types) of the service object desired by the client. The service template can also include a service ID, which uniquely identifies a service, and attributes, which must exactly match the attributes uploaded by the service provider in the service item. The service template can also contain wildcards for any of these fields. A wildcard in the service ID field, for example, will match any service ID. The lookup( ) method sends the service template to the lookup service, which performs the query and sends back zero to any matching service objects. The client gets a reference to the matching service objects as the return value of the lookup( ) method.

In the general case, a client looks up a service by Java type, usually an interface. For example, if a client needed to use a printer, it would compose a service template that included a Class object for a well-known interface to printer services. All printer services would implement this well-known interface. The lookup service would return a service object (or objects) that implemented this interface. Attributes can be included in the service template to narrow the number of matches for such a type-based search. The client would use the printer service by invoking methods from the well-known printer service interface on the service object.

Separation of interface and implementation

Jini's architecture brings object-oriented programming to the network by enabling network services to take advantage of one of the fundamentals of objects: the separation of interface and implementation. For example, a service object can grant clients access to the service in many ways. The object can actually represent the entire service, which is downloaded to the client during lookup and then executed locally. Alternatively, the service object can serve merely as a proxy to a remote server. Then when the client invokes methods on the service object, it sends the requests across the network to the server, which does the real work. A third option is for the local service object and a remote server to each do part of the work.

One important consequence of Jini's architecture is that the network protocol used to communicate between a proxy service object and a remote server does not need to be known to the client. As illustrated in the figure below, the network protocol is part of the service's implementation. This protocol is a private matter decided upon by the developer of the service. The client can communicate with the service via this private protocol because the service injects some of its own code (the service object) into the client's address space. The injected service object could communicate with the service via RMI, CORBA, DCOM, some home-brewed protocol built on top of sockets and streams, or anything else. The client simply doesn't need to care about network protocols, because it can talk to the well-known interface that the service object implements. The service object takes care of any necessary communication on the network.

The client talks to the service through a well-known interface

Different implementations of the same service interface can use completely different approaches and network protocols. A service can use specialized hardware to fulfill client requests, or it can do all its work in software. In fact, the implementation approach taken by a single service can evolve over time. The client can be sure it has a service object that understands the current implementation of the service, because the client receives the service object (by way of the lookup service) from the service provider itself. To the client, a service looks like the well-known interface, regardless of how the service is implemented.

Abstracting distributed systems

Jini attempts to raise the level of abstraction for distributed systems programming, from the network protocol level to the object interface level. In the emerging proliferation of embedded devices connected to networks, many pieces of a distributed system may come from different vendors. Jini makes it unnecessary for vendors of devices to agree on network level protocols that allow their devices to interact. Instead, vendors must agree on Java interfaces through which their devices can interact. The processes of discovery, join, and lookup, provided by the Jini run-time infrastructure, will enable devices to locate each other on the network. Once they locate each other, devices will be able to communicate with each other through Java interfaces.

JavaSpaces

Summary

There’s actually a lot more to networking than can be covered in this introductory treatment. Java networking also provides fairly extensive support for URLs, including protocol handlers for different types of content that can be discovered at an Internet site. You can find other Java networking features fully and carefully described in Java Network Programming by Elliotte Rusty Harold (O’Reilly, 1997).

Exercises

Compile and run the JabberServer and JabberClient programs in this chapter. Now edit the files to remove all of the buffering for the input and output, then compile and run them again to observe the results.

Create a server that asks for a password, then opens a file and sends the file over the network connection. Create a client that connects to this server, gives the appropriate password, then captures and saves the file. Test the pair of programs on your machine using the localhost (the local loopback IP address 127.0.0.1 produced by calling InetAddress.getByName(null)).

Modify the server in Exercise 2 so that it uses multithreading to handle multiple clients.

Modify JabberClient so that output flushing doesn’t occur and observe the effect.

Modify MultiJabberServer so that it uses thread pooling. Instead of throwing away a thread each time a client disconnects, the thread should put itself into an “available pool” of threads. When a new client wants to connect, the server will look in the available pool for a thread to handle the request, and if one isn’t available, make a new one. This way the number of threads necessary will naturally grow to the required quantity. The value of thread pooling is that it doesn’t require the overhead of creating and destroying a new thread for each new client.

Build on ShowHTML.java to produce an applet that is a password-protected gateway to a particular portion of your Web site.

(More challenging) Create a client/server pair of programs that use datagrams to transmit a file from one machine to the other. (See the description at the end of the datagram section of this chapter.)

(More challenging) Take the VLookup.java program and modify it so that when you click on the resulting name it automatically takes that name and copies it to the clipboard (so you can simply paste it into your email). You’ll need to look back at the IO stream chapter to remember how to use the Java 1.1 clipboard.

----

[68] This means a maximum of just over four billion numbers, which is rapidly running out. The new standard for IP addresses will use a 128-bit number, which should produce enough unique IP addresses for the foreseeable future.

[69] Many brain cells died in agony to discover this information.

[70] This section was contributed by Robert Castaneda, with help from Dave

Bartlett.

[71] This section was contributed by Bill Venners (www.artima.com).

[Previous Chapter] [Short TOC] [Table of Contents] [Index] [Next Chapter]

Last Update:03/13/2000

15.8 远程方法

为通过网络执行其他机器上的代码，传统的方法不仅难以学习和掌握，也极易出错。思考这个问题最佳的方式是：某些对象正好位于另一台机器，我们可向它们发送一条消息，并获得返回结果，就象那些对象位于自己的本地机器一样。Java 1.1 的“远程方法调用”（RMI）采用的正是这种抽象。本节将引导大家经历一些必要的步骤，创建自己的 RMI 对象。

15.8.1 远程接口概念

RMI 对接口有着强烈的依赖。在需要创建一个远程对象的时候，我们通过传递一个接口来隐藏基层的实施细节。所以客户得到远程对象的一个句柄时，它们真正得到的是接口句柄。这个句柄正好同一些本地的根代码连接，由后者负责通过网络通信。但我们并不关心这些事情，只需通过自己的接口句柄发送消息即可。

创建一个远程接口时，必须遵守下列规则：

(1) 远程接口必须为 public 属性（不能有“包访问”；也就是说，它不能是“友好的”）。否则，一旦客户试图装载一个实现了远程接口的远程对象，就会得到一个错误。

(2) 远程接口必须扩展接口 java.rmi.Remote。

(3) 除与应用程序本身有关的异常之外，远程接口中的每个方法都必须在自己的 **throws** 从句中声明 **java.rmi.RemoteException**。

(4) 作为参数或返回值传递的一个远程对象（不管是直接的，还是在本地对象中嵌入）必须声明为远程接口，不可声明为实施类。

下面是一个简单的远程接口示例，它代表的是一个精确计时服务：

898 页程序

```
//: c15:rmi:PerfectTimeI.java
// The PerfectTime remote interface.
package c15.rmi;
import java.rmi.*;

interface PerfectTimeI extends Remote {
  long getPerfectTime() throws RemoteException;
} ///:~
```

它表面上与其他接口是类似的，只是对 Remote 进行了扩展，而且它的所有方法都会“掷”出 RemoteException（远程异常）。记住接口和它所有的方法都是 **public** 的。

15.8.2 远程接口的实施

服务器必须包含一个扩展了 **UnicastRemoteObject** 的类，并实现远程接口。这个类也可以含有附加的方法，但客户只能使用远程接口中的方法。这是显然的，因为客户得到的只是指向接口的一个句柄，而非实现它的那个类。

必须为远程对象明确定义构建器，即使只准备定义一个默认构建器，用它调用基础类构建器。必须把它明确地编写出来，因为它必须"掷"出 RemoteException 违例。

下面列出远程接口 PerfectTime 的实施过程：

899-900 页程序

```
//: c15:rmi:PerfectTime.java
// The implementation of
// the PerfectTime remote object.
package c15.rmi;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.net.*;

public class PerfectTime
    extends UnicastRemoteObject
    implements PerfectTimeI {
  // Implementation of the interface:
  public long getPerfectTime()
      throws RemoteException {
    return System.currentTimeMillis();
  }
  // Must implement constructor
  // to throw RemoteException:
  public PerfectTime() throws RemoteException {
    // super(); // Called automatically
  }
  // Registration for RMI serving:
  public static void main(String[] args) {
    System.setSecurityManager(
      new RMISecurityManager());
    try {
      PerfectTime pt = new PerfectTime();
      Naming.bind(
        "//peppy:2005/PerfectTime", pt);
      System.out.println("Ready to do time");
    } catch(Exception e) {
      e.printStackTrace();
    }
    }
} ///:~
```

在这里，main()控制着设置服务器的全部细节。保存 RMI 对象时，必须在程序的某个地方采取下述操作：

(1) 创建和安装一个安全管理器，令其支持 RMI。作为 Java 发行包的一部分，适用于 RMI 唯一一个是 RMISecurityManager。

(2) 创建远程对象的一个或多个实例。在这里，大家可看到创建的是 PerfectTime 对象。

(3) 向 RMI 远程对象注册表注册至少一个远程对象。一个远程对象拥有的方法可生成指向其他远程对象的句柄。这样一来，客户只需到注册表里访问一次，得到第一个远程对象即可。

1. 设置注册表

在这儿，大家可看到对静态方法 Naming.bind()的一个调用。然而，这个调用要求注册表作为计算机上的一个独立进程运行。注册表服务器的名字是 rmiregistry。在 32 位 Windows 环境中，可使用：

start rmiregistry

令其在后台运行。在 Unix 中，使用：

rmiregistry &

和许多网络程序一样，rmiregistry 位于机器启动它所在的某个 IP 地址处，但它也必须监视一个端口。如果象上面那样调用 rmiregistry，不使用参数，注册表的端口就会默认为 1099。若希望它位于其他某个端口，只需在命令行添加一个参数，指定那个端口编号即可。对这个例子来说，端口将位于 2005，所以 rmiregistry 应该象下面这样启动（对于 32 位 Windows）：

start rmiregistry 2005

对于 Unix，则使用下述命令：

rmiregistry 2005 &

与端口有关的信息必须传送给 bind()命令，同时传送的还有注册表所在的那台机器的 IP 地址。但假若我们想在本地测试 RMI 程序，就象本章的网络程序一直测试的那样，这样做就会带来问题。在 JDK 1.1.1 版本中，存在着下述两方面的问题（注释⑦）：

(1) localhost 不能随 RMI 工作。所以为了在单独一台机器上完成对 RMI 的测试，必须提供机器的名字。为了在 32 位 Windows 环境中调查自己机器的名字，可进入控制面板，选择“网络”，选择“标识”卡片，其中列出了计算机的名字。就我自己的情况来说，我的机器叫作“Colossus”（因为我用几个大容量的硬盘保存各种不同的开发系统——Clossus 是“巨人”的意思）。似乎大写形式会被忽略。

(2) 除非计算机有一个活动的 TCP/IP 连接，否则 RMI 不能工作，即使所有组件都只需要在本地机器里互相通信。这意味着在试图运行程序之前，必须连接到自己的 ISP（因特网服务提供者），否则会得到一些含义模糊的违例消息。

⑦：为找出这些信息，我不知损伤了多少个脑细胞。

考虑到这些因素，bind()命令变成了下面这个样子：

Naming.bind("//colossus:2005/PerfectTime", pt);

若使用默认端口 1099，就没有必要指定一个端口，所以可以使用：

Naming.bind("//colossus/PerfectTime", pt);

在 JDK 未来的版本中（1.1 之后），一旦改正了 localhost 的问题，就能正常地进行本地测试，去掉 IP 地址，只使用标识符：

Naming.bind("PerfectTime", pt);

服务名是任意的；它在这里正好为 PerfectTime，和类名一样，但你可以根据情况任意修改。最重要的是确保它在注册表里是个独一无二的名字，以便客户正常地获取远程对象。若这个名字已在注册表里了，就会得到一个 AlreadyBoundException 违例。为防止这个问题，可考虑坚持使用 rebind()，放弃 bind()。这是由于 rebind()要么会添加一个新条目，要么将同名的条目替换掉。

尽管 main()退出，我们的对象已经创建并注册，所以会由注册表一直保持活动状态，等候客户到达并发出对它的请求。只要 rmiregistry 处于运行状态，而且我们没有为名字调用 Naming.unbind()方法，对象就肯定位于那个地方。考虑到这个原因，在我们设计自己的代码时，需要先关闭 rmiregistry，并在编译远程对象的一个新版本时重新启动它。

并不一定要将 rmiregistry 作为一个外部进程启动。若事前知道自己的是要求用以注册表的唯一一个应用，就可在程序内部启动它，使用下述代码：

LocateRegistry.createRegistry(2005);

和前面一样，2005 代表我们在这个例子里选用的端口号。这等价于在命令行执行 rmiregistry 2005。但在设计 RMI 代码时，这种做法往往显得更加方便，因为它取消了启动和中止注册表所需的额外步骤。一旦执行完这个代码，就可象以前一样使用 Naming 进行“绑定”——bind()。

15.8.3 创建根与干

若编译和运行 PerfectTime.java，即使 rmiregistry 正确运行，它也无法工作。这是由于 RMI 的框架尚未就位。首先必须创建根和干，以便提供网络连接操作，并使我们将远程对象伪装成自己机器内的某个本地对象。

所有这些幕后的工作都是相当复杂的。我们从远程对象传入、传出的任何对象都必须“implement Serializable”（如果想传递远程引用，而非整个对象，对象的参数就可以“implement Remote”）。因此可以想象，当根和干通过网络“汇集”所有参数并返回结果的时候，会自动进行序列化以及数据的重新装配。幸运的是，我们根本没必要了解这些方面的任何细节，但根和干却是必须创建的。一个简单的过程如下：在编译好的代码中调用 rmic，它会创建必需的一些文件。所以唯一要做的事情就是为编译过程新添一个步骤。

然而，rmic 工具与特定的包和类路径有很大的关联。PerfectTime.java 位于包 c15.Ptime 中，即使我们调用与 PerfectTime.class 同一目录内的 rmic，rmic 都无法找到文件。这是由于它搜索的是类路径。因此，我们必须同时指定类路径，就象下面这样：

rmic c15.PTime.PerfectTime

执行这个命令时，并不一定非要在包含了 PerfectTime.class 的目录中，但结果会置于当前目录。

若 rmic 成功运行，目录里就会多出两个新类：

PerfectTime_Stub.class

PerfectTime_Skel.class

它们分别对应根（Stub）和干（Skeleton）。现在，我们已准备好让服务器与客户互相沟通了。

15.8.4 使用远程对象

RMI 全部的宗旨就是尽可能简化远程对象的使用。我们在客户程序中要做的唯一一件额外的事情就是查找并从服务器取回远程接口。自此以后，剩下的事情就是普通的 Java 编程：将消息发给对象。下面是使用 PerfectTime 的程序：

903-904 页程序

ID 字串与那个用 Naming 注册对象的那个字串是相同的，第一部分指出了 URL 和端口号。由于我们准备使用一个 URL，所以也可以指定因特网上的一台机器。

从 Naming.lookup()返回的必须造型到远程接口，而不是到类。若换用类，会得到一个违例提示。

在下述方法调用中：

t.getPerfectTime( )

我们可看到一旦获得远程对象的句柄，用它进行的编程与用本地对象的编程是非常相似（仅有一个区别：远程方法会“掷”出一个 RemoteException 违例）。

15.8.5 RMI 的替选方案

RMI 只是一种创建特殊对象的方式，它创建的对象可通过网络发布。它最大的优点就是提供了一种“纯 Java”方案，但假如已经有许多用其他语言编写的代码，则 RMI 可能无法满足我们的要求。目前，两种最具竞争力的替选方案是微软的 **DCOM**（根据微软的计划，它最终会移植到除 Windows 以外的其他平台）以及 **CORBA**。CORBA 自 Java 1.1 便开始支持，是一种全新设计的概念，面向跨平台应用。在由 Orfali 和 Harkey 编著的**《Client/Server Programming with Java and CORBA》**一书中（John Wiley&Sons 1997 年出版），大家可获得对 Java 中的分布式对象的全面介绍（该书似乎对 CORBA 似乎有些偏见）。为 CORBA 赋予一个较公正的对待的一本书是由 Andreas Vogel 和 Keith Duddy 编写的**《Java Programming with CORBA》，** John Wiley&Sons 于 1997 年出版。

15.9 总结

由于篇幅所限，还有其他许多涉及连网的概念没有介绍给大家。Java 也为 URL 提供了相当全面的支持，包括为因特网上不同类型的客户提供协议控制器等等。

除此以外，一种正在逐步流行的技术叫作 **Servlet Server**。它是一种因特网服务器应用，通过 Java 控制客户请求，而非使用以前那种速度很慢、且相当麻烦的 CGI（通用网关接口）协议。这意味着为了在服务器那一端提供服务，我们可以用 Java 编程，不必使用自己不熟悉的其他语言。由于 Java 具有优秀的移植能力，所以不必关心具体容纳这个服务器是什么平台。

所有这些以及其他特性都在**《Java Network Programming》**一书中得到了详细讲述。该书由 Elliotte Rusty Harold 编著，O'Reilly 于 1997 年出版。

15.10 练习

(1) 编译和运行本章中的 JabberServer 和 JabberClient 程序。接着编辑一下程序，删去为输入和输出设计的所有缓冲机制，然后再次编译和运行，观察一下结果。

(2) 创建一个服务器，用它请求用户输入密码，然后打开一个文件，并将文件通过网络连接传送出去。创建一个同该服务器连接的客户，为其分配适当的密码，然后捕获和保存文件。在自己的机器上用 localhost（通过调用 InetAddress.getByName(null)生成本地 IP 地址 127.0.0.1）测试这两个程序。

(3) 修改练习 2 中的程序，令其用多线程机制对多个客户进行控制。

(4) 修改 JabberClient，禁止输出刷新，并观察结果。

(5) 以 ShowHTML.java 为基础，创建一个程序片，令其成为对自己 Web 站点的特定部分进行密码保护的大门。

(6) （可能有些难度）创建一对客户 / 服务器程序，利用数据报（Datagram）将一个文件从一台机器传到另一台（参见本章数据报小节末尾的叙述）。

(7) （可能有些难度）对 VLookup.java 程序作一番修改，使我们能点击得到的结果名字，然后程序会自动取得那个名字，并把它复制到剪贴板（以便我们方便地粘贴到自己的 E-mail）。可能要回过头去研究一下 I/O 数据流的那一章，回忆该如何使用 Java 1.1 剪贴板。