附录 A 使用非 JAVA 代码

JAVA 语言及其标准 API（应用程序编程接口）应付应用程序的编写已绰绰有余。但在某些情况下，还是必须使用非 JAVA 编码。例如，我们有时要访问操作系统的专用特性，与特殊的硬件设备打交道，重复使用现有的非 Java 接口，或者要使用“对时间敏感”的代码段，等等。与非 Java 代码的沟通要求获得编译器和“虚拟机”的专门支持，并需附加的工具将 Java 代码映射成非 Java 代码（也有一个简单方法：在第 15 章的“一个 Web 应用”小节中，有个例子解释了如何利用标准输入输出同非 Java 代码连接）。目前，不同的开发商为我们提供了不同的方案：Java 1.1 有“Java 固有接口”（Java Native Interface，JNI），网景提出了自己的“Java 运行期接口”（Java Runtime Interface）计划，而微软提供了 J/Direct、“本源接口”（Raw Native Interface，RNI）以及 Java/COM 集成方案。

各开发商在这个问题上所持的不同态度对程序员是非常不利的。若 Java 应用必须调用固有方法，则程序员或许要实现固有方法的不同版本——具体由应用程序运行的平台决定。程序员也许实际需要不同版本的 Java 代码，以及不同的 Java 虚拟机。

另一个方案是 CORBA（通用对象请求代理结构），这是由 OMG（对象管理组，一家非赢利性的公司协会）开发的一种集成技术。CORBA 并非任何语言的一部分，只是实现通用通信总线及服务的一种规范。利用它可在由不同语言实现的对象之间实现“相互操作”的能力。这种通信总线的名字叫作 ORB（对象请求代理），是由其他开发商实现的一种产品，但并不属于 Java 语言规范的一部分。

本附录将对 JNI，J/DIRECT，RNI，JAVA/COM 集成和 CORBA 进行概述。但不会作更深层次的探讨，甚至有时还假定读者已对相关的概念和技术有了一定程度的认识。但到最后，大家应该能够自行比较不同的方法，并根据自己要解决的问题挑选出最恰当的一种。

A.1 Java 固有接口

JNI 是一种包容极广的编程接口，允许我们从 Java 应用程序里调用固有方法。它是在 Java 1.1 里新增的，维持着与 Java 1.0 的相应特性——“固有方法接口”（NMI）——某种程度的兼容。NMI 设计上一些特点使其未获所有虚拟机的支持。考虑到这个原因，Java 语言将来的版本可能不再提供对 NMI 的支持，这儿也不准备讨论它。

目前，JNI 只能与用 C 或 C++写成的固有方法打交道。利用 JNI，我们的固有方法可以：

■创建、检查及更新 Java 对象（包括数组和字串）

■调用 Java 方法

■俘获和丢弃“异常”

■装载类并获取类信息

■进行运行期类型检查

所以，原来在 Java 中能对类及对象做的几乎所有事情在固有方法中同样可以做到。

A.1.1 调用固有方法

我们先从一个简单的例子开始：一个 Java 程序调用固有方法，后者再调用 Win32 的 API 函数 MessageBox()，显示出一个图形化的文本框。这个例子稍后也会与 J/Direct 一志使用。若您的平台不是 Win32，只需将包含了下述内容的 C 头：

#include <windows.h>

替换成：

#include <stdio.h>

并将对 MessageBox()的调用换成调用 printf()即可。

第一步是写出对固有方法及它的自变量进行声明的 Java 代码：

999 页程序

在固有方法声明的后面，跟随有一个 static 代码块，它会调用 System.loadLibrary()（可在任何时候调用它，但这样做更恰当）System.loadLibrary()将一个 DLL 载入内存，并建立同它的链接。DLL 必须位于您的系统路径，或者在包含了 Java 类文件的目录中。根据具体的平台，JVM 会自动添加适当的文件扩展名。

1. C 头文件生成器：javah

现在编译您的 Java 源文件，并对编译出来的.class 文件运行 javah。javah 是在 1.0 版里提供的，但由于我们要使用 Java 1.1 JNI，所以必须指定-jni 参数：

javah -jni ShowMsgBox

javah 会读入类文件，并为每个固有方法声明在 C 或 C++头文件里生成一个函数原型。下面是输出结果——ShowMsgBox.h 源文件（为符合本书的要求，稍微进行了一下修改）：

1000 页程序

从“#ifdef_cplusplus”这个预处理引导命令可以看出，该文件既可由 C 编译器编译，亦可由 C++编译器编译。第一个#include 命令包括 jni.h——一个头文件，作用之一是定义在文件其余部分用到的类型；JNIEXPORT 和 JNICALL 是一些宏，它们进行了适当的扩充，以便与那些不同平台专用的引导命令配合；JNIEnv，jobject 以及 jstring 则是 JNI 数据类型定义。

2. 名称管理和函数签名

JNI 统一了固有方法的命名规则；这一点是非常重要的，因为它属于虚拟机将 Java 调用与固有方法链接起来的机制的一部分。从根本上说，所有固有方法都要以一个“Java”起头，后面跟随 Java 方法的名字；下划线字符则作为分隔符使用。若 Java 固有方法“过载”（即命名重复），那么也把函数签名追加到名字后面。在原型前面的注释里，大家可看到固有的签名。欲了解命名规则和固有方法签名更详细的情况，请参考相应的 JNI 文档。

3. 实现自己的 DLL

此时，我们要做的全部事情就是写一个 C 或 C++源文件，在其中包含由 javah 生成的头文件；并实现固有方法；然后编译它，生成一个动态链接库。这一部分的工作是与平台有关的，所以我假定读者已经知道如何创建一个 DLL。通过调用一个 Win32 API，下面的代码实现了固有方法。随后，它会编译和链接到一个名为 MsgImpl.dll 的文件里：

1001 页程序

若对 Win32 没有兴趣，只需跳过 MessageBox()调用；最有趣的部分是它周围的代码。传递到固有方法内部的自变量是返回 Java 的大门。第一个自变量是类型 JNIEnv 的，其中包含了回调 JVM 需要的所有挂钩（下一节再详细讲述）。由于方法的类型不同，第二个自变量也有自己不同的含义。对于象上例那样的非 static 方法（也叫作实例方法），第二个自变量等价于 C++的“this”指针，并类似于 Java 的“this”：都引用了调用固有方法的那个对象。对于 static 方法，它是对特定 Class 对象的一个引用，方法就是在那个 Class 对象里实现的。

剩余的自变量代表传递到固有方法调用里的 Java 对象。主类型也是以这种形式传递的，但它们进行的“按值”传递。

在后面的小节里，我们准备讲述如何从一个固有方法的内部访问和控制 JVM，同时对上述代码进行更详尽的解释。

A.1.2 访问 JNI 函数：JNIEnv 自变量

利用 JNI 函数，程序员可从一个固有方法的内部与 JVM 打交道。正如大家在前面的例子中看到的那样，每个 JNI 固有方法都会接收一个特殊的自变量作为自己的第一个参数：JNIEnv 自变量——它是指向类型为 JNIEnv_的一个特殊 JNI 数据结构的指针。JNI 数据结构的一个元素是指向由 JVM 生成的一个数组的指针；该数组的每个元素都是指向一个 JNI 函数的指针。可从固有方法的内部发出对 JNI 函数的调用，做法是撤消对这些指针的引用（具体的操作实际很简单）。每种 JVM 都以自己的方式实现了 JNI 函数，但它们的地址肯定位于预先定义好的偏移处。

利用 JNIEnv 自变量，程序员可访问一系列函数。这些函数可划分为下述类别：

■获取版本信息
■进行类和对象操作
■控制对 Java 对象的全局和局部引用
■访问实例字段和静态字段
■调用实例方法和静态方法
■执行字串和数组操作
■产生和控制 Java 异常

JNI 函数的数量相当多，这里不再详述。相反，我会向大家揭示使用这些函数时背后的一些基本原理。欲了解更详细的情况，请参阅自己所用编译器的 JNI 文档。

若观察一下 jni.h 头文件，就会发现在#ifdef _cplusplus 预处理器条件的内部，当由 C++编译器编译时，JNIEnv_结构被定义成一个类。这个类包含了大量内嵌函数。通过一种简单而且熟悉的语法，这些函数让我们可以从容访问 JNI 函数。例如，前例包含了下面这行代码：
(*jEnv)->ReleaseStringUTFChars(jEnv, jMsg,msg);
它在 C++里可改写成下面这个样子：
jEnv->ReleaseStringUTFChars(jMsg,msg);
大家可注意到自己不再需要同时撤消对 jEnv 的两个引用，相同的指针不再作为第一个参数传递给 JNI 函数调用。在这些例子剩下的地方，我会使用 C++风格的代码。

1. 访问 Java 字串
作为访问 JNI 函数的一个例子，请思考上述的代码。在这里，我们利用 JNIEnv 的自变量 jEnv 来访问一个 Java 字串。Java 字串采取的是 Unicode 格式，所以假若收到这样一个字串，并想把它传给一个非 Unicode 函数（如 printf()），首先必须用 JNI 函数 GetStringUTFChars()将其转换成 ASCII 字符。该函数能接收一个 Java 字串，然后把它转换成 UTF-8 字符（用 8 位宽度容纳 ASCII 值，或用 16 位宽度容纳 Unicode；若原始字串的内容完全由 ASCII 构成，那么结果字串也是 ASCII）。
GetStringUTFChars 是 JNIEnv 间接指向的那个结构里的一个字段，而这个字段又是指向一个函数的指针。为访问 JNI 函数，我们用传统的 C 语法来调用一个函数（通过指针）。利用上述形式可实现对所有 JNI 函数的访问。

A.1.3 传递和使用 Java 对象
在前例中，我们将一个字串传递给固有方法。事实上，亦可将自己创建的 Java 对象传递给固有方法。
在我们的固有方法内部，可访问已收到的那些对象的字段及方法。
为传递对象，声明固有方法时要采用原始的 Java 语法。如下例所示，MyJavaClass 有一个 public（公共）字段，以及一个 public 方法。UseObjects 类声明了一个固有方法，用于接收 MyJavaClass 类的一个对象。为调查固有方法是否能控制自己的自变量，我们设置了自变量的 public 字段，调用固有方法，然后打印出 public 字段的值。

1004 页上程序

编译好代码，并将.class 文件传递给 javah 后，就可以实现固有方法。在下面这个例子中，一旦取得字段和方法 ID，就会通过 JNI 函数访问它们。

1004-1005 页程序

除第一个自变量外，C++函数会接收一个 jobject，它代表 Java 对象引用“固有”的那一面——那个引用是我们从 Java 代码里传递的。我们简单地读取 aValue，把它打印出来，改变这个值，调用对象的 divByTwo()方法，再将值重新打印一遍。
为访问一个字段或方法，首先必须获取它的标识符。利用适当的 JNI 函数，可方便地取得类对象、元素名以及签名信息。这些函数会返回一个标识符，利用它可访问对应的元素。尽管这一方式显得有些曲折，但我们的固有方法确实对 Java 对象的内部布局一无所知。因此，它必须通过由 JVM 返回的索引访问字段和方法。这样一来，不同的 JVM 就可实现不同的内部对象布局，同时不会对固有方法造成影响。

若运行 Java 程序，就会发现从 Java 那一侧传来的对象是由我们的固有方法处理的。但传递的到底是什么呢？是指针，还是 Java 引用？而且垃圾收集器在固有方法调用期间又在做什么呢？

垃圾收集器会在固有方法执行期间持续运行，但在一次固有方法调用期间，我们的对象可保证不会被当作“垃圾”收集去。为确保这一点，事先创建了“局部引用”，并在固有方法调用之后立即清除。由于它们的“生命期”与调用过程息息相关，所以能够保证对象在固有方法调用期间的有效性。

由于这些引用会在每次函数调用的时候创建和破坏，所以不可在 static 变量中制作固有方法的局部副本（本地拷贝）。若希望一个引用在函数存在期间持续有效，就需要一个全局引用。全局引用不是由 JVM 创建的，但通过调用特定的 JNI 函数，程序员可将局部引用扩展为全局引用。创建一个全局引用时，需对引用对象的“生存时间”负责。全局引用（以及它引用的对象）会一直留在内存里，直到用特定的 JNI 函数明确释放了这个引用。它类似于 C 的 malloc()和 free()。

A.1.4 JNI 和 Java 异常

利用 JNI，可丢弃、捕捉、打印以及重新丢弃 Java 异常，就象在一个 Java 程序里那样。但对程序员来说，需自行调用专用的 JNI 函数，以便对异常进行处理。下面列出用于异常处理的一些 JNI 函数：

■Throw()：丢弃一个现有的异常对象；在固有方法中用于重新丢弃一个异常。

■ThrowNew()：生成一个新的异常对象，并将其丢弃。

■ExceptionOccurred()：判断一个异常是否已被丢弃，但尚未清除。

■ExceptionDescribe()：打印一个异常和堆栈跟踪信息。

■ExceptionClear()：清除一个待决的异常。

■FatalError()：造成一个严重错误，不返回。

在所有这些函数中，最不能忽视的就是 ExceptionOccurred()和 ExceptionClear()。大多数 JNI 函数都能产生异常，而且没有象在 Java 的 try 块内的那种语言特性可供利用。所以在每一次 JNI 函数调用之后，都必须调用 ExceptionOccurred()，了解异常是否已被丢弃。若侦测到一个异常，可选择对其加以控制（可能时还要重新丢弃它）。然而，必须确保异常最终被清除。这可以在自己的函数中用 ExceptionClear()来实现；若异常被重新丢弃，也可能在其他某些函数中进行。但无论如何，这一工作是必不可少的。

我们必须保证异常被彻底清除。否则，假若在一个异常待决的情况下调用一个 JNI 函数，获得的结果往往是无法预知的。也有少数几个 JNI 函数可在异常时安全调用；当然，它们都是专门的异常控制函数。

A.1.5 JNI 和线程处理

由于 Java 是一种多线程语言，几个线程可能同时发出对一个固有方法的调用（若另一个线程发出调用，固有方法可能在运行期间暂停）。此时，完全要由程序员来保证固有调用在多线程的环境中安全进行。例如，要防范用一种未进行监视的方法修改共享数据。此时，我们主要有两个选择：将固有方法声明为“同步”，或在固有方法内部采取其他某些策略，确保数据处理正确地并发进行。

此外，绝对不要通过线程传递 JNIEnv，因为它指向的内部结构是在“每线程”的基础上分配的，而且包含了只对那些特定的线程才有意义的信息。

A.1.6 使用现成代码

为实现 JNI 固有方法，最简单的方法就是在一个 Java 类里编写固有方法的原型，编译那个类，再通过 javah 运行.class 文件。但假若我们已有一个大型的、早已存在的代码库，而且想从 Java 里调用它们，此时又该如何是好呢？不可将 DLL 中的所有函数更名，使其符合 JNI 命名规则，这种方案是不可行的。最好的方法是在原来的代码库“外面”写一个封装 DLL。Java 代码会调用新 DLL 里的函数，后者再调用原始的 DLL 函数。这个方法并非仅仅是一种解决方案；大多数情况下，我们甚至必须这样做，因为必须面向对象引用调用 JNI 函数，否则无法使用它们。

A.2 微软的解决方案

到本书完稿时为止，微软仍未提供对 JNI 的支持，只是用自己的专利方法提供了对非 Java 代码调用的支持。这一支持内建到编译器 Microsoft JVM 以及外部工具中。只有程序用 Microsoft Java 编译器编译，而且只有在 Microsoft Java 虚拟机（JVM）上运行的时候，本节讲述的特性才会有效。若计划在因特网上发行自己的应用，或者本单位的内联网建立在不同平台的基础上，就可能成为一个严重的问题。

微软与 Win32 代码的接口为我们提供了连接 Win32 的三种途径：

(1) J/Direct：方便调用 Win32 DLL 函数的一种途径，具有某些限制。

(2) 本原接口（RNI）：可调用 Win32 DLL 函数，但必须自行解决“垃圾收集”问题。

(3) Java/COM 集成：可从 Java 里直接揭示或调用 COM 服务。

后续的小节将分别探讨这三种技术。

写作本书的时候，这些特性均通过了 Microsoft SDK for Java 2.0 beta 2 的支持。可从微软公司的 Web 站点下载这个开发平台（要经历一个痛苦的选择过程，他们叫作“Active Setup”）。Java SDK 是一套命令行工具的集合，但编译引擎可轻易嵌入 Developer Studio 环境，以便我们用 Visual J++ 1.1 来编译 Java 1.1 代码。

A.3 J/Direct

J/Direct 是调用 Win32 DLL 函数最简单的方式。它的主要设计目标是与 Win32API 打交道，但完全可用它调用其他任何 API。但是，尽管这一特性非常方便，但它同时也造成了某些限制，且降低了性能（与 RNI 相比）。但 J/Direct 也有一些明显的优点。首先，除希望调用的那个 DLL 里的代码之外，没有必要再编写额外的非 Java 代码，换言之，我们不需要一个封装器或者代理／存根 DLL。其次，函数自变量与标准数据类型之间实现了自动转换。若必须传递用户自定义的数据类型，那么 J/Direct 可能不按我们的希望工作。第三，就象下例展示的那样，它非常简单和直接。只需少数几行，这个例子便能调用 Win32 API 函数 MessageBox()，它能弹出一个小的模态窗口，并带有一个标题、一条消息、一个可选的图标以及几个按钮。

1008 页程序

令人震惊的是，这里便是我们利用 J/Direct 调用 Win32 DLL 函数所需的全部代码。其中的关键是位于示范代码底部的 MessageBox()声明之前的@dll.import 引导命令。它表面上看是一条注释，但实际并非如此。它的作用是告诉编译器：引导命令下面的函数是在 USER32 DLL 里实现的，而且应相应地调用。我们要做的全部事情就是提供与 DLL 内实现的函数相符的一个原型，并调用函数。但是毋需在 Java 版本里手工键入需要的每一个 Win32 API 函数，一个 Microsoft Java 包会帮我们做这件事情（很快就会详细解释）。为了让这个例子正常工作，函数必须“按名称”由 DLL 导出。但是，也可以用@dll.import 引导命令“按顺序”链接。

举个例子来说，我们可指定函数在 DLL 里的入口位置。稍后还会具体讲述@dll.import 引导命令的特性。
用非 Java 代码进行链接的一个重要问题就是函数参数的自动配置。正如大家看到的那样，MessageBox()的 Java 声明采用了两个字串自变量，但原来的 C 方案则采用了两个 char 指针。编译器会帮助我们自动转换标准数据类型，同时遵照本章后一节要讲述的规则。
最好，大家或许已注意到了 main()声明中的 UnsatisfiedLinkError 异常。在运行期的时候，一旦链接程序不能从非 Java 函数里解析出符号，就会触发这一异常（事件）。这可能是由多方面的原因造成的：.dll 文件未找到；不是一个有效的 DLL；或者 J/Direct 未获您所使用的虚拟机的支持。为了使 DLL 能被找到，它必须位于 Windows 或 Windows\System 目录下，位于由 PATH 环境变量列出的一个目录中，或者位于和.class 文件相同的目录。J/Direct 获得了 Microsoft Java 编译器 1.02.4213 版本及更高版本的支持，也获得了 Microsoft JVM 4.79.2164 及更高版本的支持。为了解自己编译器的版本号，请在命令行下运行 JVC，不要加任何参数。为了解 JVM 的版本号，请找到 msjava.dll 的图标，并利用右键弹出菜单观察它的属性。

A.3.1 @dll.import 引导命令
作为使用 J/Direct 唯一的途径，@dll.import 引导命令相当灵活。它提供了为数众多的修改符，可用它们自定义同非 Java 代码建立链接关系的方式。它亦可应用于类内的一些方法，或应用于整个类。也就是说，我们在那个类内声明的所有方法都是在相同的 DLL 里实现的。下面让我们具体研究一下这些特性。

1. 别名处理和按顺序链接
为了使@dll.import 引导命令能象上面显示的那样工作，DLL 内的函数必须按名字导出。然而，我们有时想使用与 DLL 里原始名字不同的一个名字（别名处理），否则函数就可能按编号（比如按顺序）导出，而不是按名字导出。下面这个例子声明了 FinestraDiMessaggio()（用意大利语说的“MessageBox”）。正如大家看到的那样，使用的语法是非常简单的。

1010 页上程序

下面这个例子展示了如何同 DLL 里并非按名字导出的一个函数建立链接，那个函数事实是按它们在 DLL 里的位置导出的。这个例子假设有一个名为 MYMATH 的 DLL，这个 DLL 在位置编号 3 处包含了一个函数。那个函数获取两个整数作为自变量，并返回两个整数的和。

1010 页下程序

可以看出，唯一的差异就是 entrypoint 自变量的形式。

2. 将@dll.import 应用于整个类
@dll.import 引导命令可应用于整个类。也就是说，那个类的所有方法都是在相同的 DLL 里实现的，并具有相同的链接属性。引导命令不会由子类继承；考虑到这个原因，而且由于 DLL 里的函数是自然的 static 函数，所以更佳的设计方案是将 API 函数封装到一个独立的类里，如下所示：

1011 页程序

由于 MessageBeep()和 MessageBox()函数已在不同的类里被声明成 static 函数，所以必须在调用它们时规定作用域。大家也许认为必须用上述的方法将所有 Win32 API（函数、常数和数据类型）都映射成 Java 类。但幸运的是，根本不必这样做。

A.3.2 com.ms.win32 包

Win32 API 的体积相当庞大——包含了数以千计的函数、常数以及数据类型。当然，我们并不想将每个 Win32 API 函数都写成对应 Java 形式。微软考虑到了这个问题，发行了一个 Java 包，可通过 J/Direct 将 Win32 API 映射成 Java 类。这个包的名字叫作 com.ms.win32。安装 Java SDK 2.0 时，若在安装选项中进行了相应的设置，这个包就会安装到我们的类路径中。这个包由大量 Java 类构成，它们完整再现了 Win32 API 的常数、数据类型以及函数。包容能力最大的三个类是 User32.class，Kernel.class 以及 Gdi32.class。它们包含的是 Win32 API 的核心内容。为使用它们，只需在自己的 Java 代码里导入即可。前面的 ShowMsgBox 示例可用 com.ms.win32 改写成下面这个样子（这里也考虑到了用更恰当的方式使用 UnsatisfiedLinkError）：

1012 页程序

Java 包是在第一行导入的。现在，可在不进行其他声明的前提下调用 MessageBeep()和 MessageBox()函数。在 MessageBeep()里，我们可看到包导入时也声明了 Win32 常数。这些常数是在大量 Java 接口里定义的，全部命名为 winx（x 代表欲使用之常数的首字母）。
写作本书时，com.ms.win32 包的开发仍未正式完成，但已可堪使用。

A.3.3 汇集

“汇集”（Marshaling）是指将一个函数自变量从它原始的二进制形式转换成与语言无关的某种形式，再将这种通用形式转换成适合调用函数采用的二进制格式。在前面的例子中，我们调用了 MessageBox()函数，并向它传递了两个字串。MessageBox()是个 C 函数，而且 Java 字串的二进制布局与 C 字串并不相同。但尽管如此，自变量仍获得了正确的传递。这是由于在调用 C 代码前，J/Direct 已帮我们考虑到了将 Java 字串转换成 C 字串的问题。这种情况适合所有标准的 Java 类型。下面这张表格总结了简单数据类型的默认对应关系：

Java C

byte BYTE 或 CHAR
short SHORT 或 WORD
int INT，UINT，LONG，ULONG 或 DWORD
char TCHAR
long __int64
float Float
double Double
boolean BOOL
String LPCTSTR（只允许在 OLE 模式中作为返回值）
byte[] BYTE *
short[] WORD *
char[] TCHAR *

int[] DWORD *

这个列表还可继续下去，但已很能说明问题了。大多数情况下，我们不必关心与简单数据类型之间的转换问题。但一旦必须传递用户自定义类型的自变量，情况就立即变得不同了。例如，可能需要传递一个结构化的、用户自定义的数据类型，或者需要把一个指针传给原始内存区域。在这些情况下，有一些特殊的编译引导命令标记一个 Java 类，使其能作为一个指针传给结构（@dll.struct 引导命令）。欲知使用这些关键字的细节，请参考产品文档。

A.3.4 编写回调函数

有些 Win32 API 函数要求将一个函数指针作为自己的参数使用。Windows API 函数随后就可以调用自变量函数（通常是在以后发生特定的事件时）。这一技术就叫作“回调函数”。回调函数的例子包括窗口进程以及我们在打印过程中设置的回调（为后台打印程序提供回调函数的地址，使其能更新状态，并在必要的时候中止打印）。

另一个例子是 API 函数 EnumWindows()，它能枚举目前系统内所有顶级窗口。EnumWindows()要求获取一个函数指针作为自己的参数，然后搜索由 Windows 内部维护的一个列表。对于列表内的每个窗口，它都会调用回调函数，将窗口句柄作为一个自变量传给回调。

为了在 Java 里达到同样的目的，必须使用 com.ms.dll 包里的 Callback 类。我们从 Callback 里继承，并取消 callback()。这个方法只能接近 int 参数，并会返回 int 或 void。方法签名和具体的实施取决于使用这个回调的 Windows API 函数。

现在，我们要进行的全部工作就是创建这个 Callback 衍生类的一个实例，并将其作为函数指针传递给 API 函数。随后，J/Direct 会帮助我们自动完成剩余的工作。

下面这个例子调用了 Win32 API 函数 EnumWindows()；EnumWindowsProc 类里的 callback()方法会获取每个顶级窗口的句柄，获取标题文字，并将其打印到控制台窗口。

1014 页程序

对 sleep()的调用允许窗口进程在 main()退出前完成。

A.3.5 其他 J/Direct 特性

通过@dll.import 引导命令内的修改符（标记），还可用到 J/Direct 的另两项特性。第一项是对 OLE 函数的简化访问，第二项是选择 API 函数的 ANSI 及 Unicode 版本。

根据约定，所有 OLE 函数都会返回类型为 HRESULT 的一个值，它是由 COM 定义的一个结构化整数值。若在 COM 那一级编写程序，并希望从一个 OLE 函数里返回某些不同的东西，就必须将一个特殊的指针传递给它——该指针指向函数即将在其中填充数据的那个内存区域。但在 Java 中，我们没有指针可用；此外，这种方法并不简练。利用 J/Direct，我们可在@dll.import 引导命令里使用 ole 修改符，从而方便地调用 OLE 函数。标记为 ole 函数的一个固有方法会从 Java 形式的方法签名（通过它决定返回类型）自动转换成 COM 形式的函数。

第二项特性是选择 ANSI 或者 Unicode 字串控制方法。对字串进行控制的大多数 Win32 API 函数都提供了两个版本。例如，假设我们观察由 USER32.DLL 导出的符号，那么不会找到一个 MessageBox()函数，相反会看到 MessageBoxA()和 MessageBoxW()函数——分别是该函数的 ANSI 和 Unicode 版本。如果在@dll.import 引导命令里不规定想调用哪个版本，JVM 就会试着自行判断。但这一操作会在程序执行时花费较长的时间。所以，我们一般可用 ansi，unicode 或 auto 修改符硬性规定。

欲了解这些特性更详细的情况，请参考微软公司提供的技术文档。

A.4 本原接口（RNI）

同 J/Direct 相比，RNI 是一种比非 Java 代码复杂得多的接口；但它的功能也十分强大。RNI 比 J/Direct 更接近于 JVM，这也使我们能写出更有效的代码，能处理固有方法中的 Java 对象，而且能实现与 JVM 内部运行机制更紧密的集成。

RNI 在概念上类似 Sun 公司的 JNI。考虑到这个原因，而且由于该产品尚未正式完工，所以我只在这里指出它们之间的主要差异。欲了解更详细的情况，请参考微软公司的文档。

JNI 和 RNI 之间存在几方面引人注目的差异。下面列出的是由 msjavah 生成的 C 头文件（微软提供的 msjavah 在功能上相当于 Sun 的 javah），应用于前面在 JNI 例子里使用的 Java 类文件 ShowMsgBox。

1016 页程序

除可读性较差外，代码里还隐藏着一些技术性问题，待我一一道来。

在 RNI 中，固有方法的程序员知道对象的二进制布局。这样便允许我们直接访问自己希望的信息；我们不必象在 JNI 里那样获得一个字段或方法标识符。但由于并非所有虚拟机都需要将相同的二进制布局应用于自己的对象，所以上面的固有方法只能在 Microsoft JVM 下运行。

在 JNI 中，通过 JNIEnv 自变量可访问大量函数，以便同 JVM 打交道。在 RNI 中，用于控制 JVM 运作的函数变成了可直接调用。它们中的某一些（如控制异常的那一个）类似于它们的 JNI“兄弟”。但大多数 RNI 函数都有与 JNI 中不同的名字和用途。

JNI 和 RNI 最重大的一个区别是“垃圾收集”的模型。在 JNI 中，垃圾收集在固有方法执行期间遵守与 Java 代码执行时相同的规则。而在 RNI 中，要由程序员在固有方法活动期间自行负责“垃圾收集器”器的启动与中止。默认情况下，垃圾收集器在进入固有方法前处于不活动状态；这样一来，程序员就可假定准备使用的对象用不着在那个时间段内进行垃圾收集。然而一旦固有方法准备长时间执行，程序员就应考虑激活垃圾收集器——通过调用 GCEnable()这个 RNI 函数（GC 是“Garbage Collector”的缩写，即“垃圾收集”）。

也存在与全局句柄特性类似的机制——程序员可利用可保证特定的对象在 GC 活动期间不至于被当作“垃圾”收掉。概念是类似的，但名称有所差异——在 RNI 中，人们把它叫作 GCFrames。

A.4.1 RNI 总结

RNI 与 Microsoft JVM 紧密集成这一事实既是它的优点，也是它的缺点。RNI 比 JNI 复杂得多，但它也为我们提供了对 JVM 内部活动的高度控制；其中包括垃圾收集。此外，它显然针对速度进行了优化，采纳了 C 程序员熟悉的一些折衷方案和技术。但除了微软的 JVM 之外，它并不适于其他 JVM。

A.5 Java/COM 集成

COM（以前称为 OLE）代表微软公司的“组件对象模型”（Component Object Model），它是所有 ActiveX 技术（包括 ActiveX 控件、Automation 以及 ActiveX 文档）的基础。但 COM 还包含了更多的东西。它是一种特殊的规范，按照它开发出来的组件对象可通过操作系统的专门特性实现“相互操作”。在实际应用中，为 Win32 系统开发的所有新软件都与 COM 有

着一定的关系——操作系统通过 COM 对象揭示出自己的一些特性。由其他厂商开发的组件也可以建立在 COM 的基础上，我们能创建和注册自己的 COM 组件。通过这样或那样的形式，如果我们想编写 Win32 代码，那么必须和 COM 打交道。在这里，我们将仅仅重述 COM 编程的基本概念，而且假定读者已掌握了 COM 服务器（能为 COM 客户提供服务的任何 COM 对象）以及 COM 客户（能从 COM 服务器那里申请服务的一个 COM 对象）的概念。本节将尽可能地使叙述变得简单。工具实际的功能要强大得多，而且我们可通过更高级的途径来使用它们。但这也要求对 COM 有着更深刻的认识，那已经超出了本附录的范围。如果您对这个功能强大、但与不同平台有关的特性感兴趣，应该研究 COM 和微软公司的文档资料，仔细阅读有关 Java/COM 集成的那部分内容。如果想获得更多的资料，向您推荐 Dale Rogerson 编著的《Inside COM》，该书由 Microsoft Press 于 1997 年出版。

由于 COM 是所有新型 Win32 应用程序的结构核心，所以通过 Java 代码使用（或揭示）COM 服务的能力就显得尤为重要。Java/COM 集成无疑是 Microsoft Java 编译器以及虚拟机最有趣的特性。Java 和 COM 在它们的模型上是如此相似，所以这个集成在概念上是相当直观的，而且在技术上也能轻松实现无缝结合——为访问 COM，几乎不需要编写任何特殊的代码。大多数技术细节都是由编译器和／或虚拟机控制的。最终的结果便是 Java 程序员可象对待原始 Java 对象那样对待 COM 对象。而且 COM 客户可象使用其他 COM 服务器那样使用由 Java 实现的 COM 服务器。在这里提醒大家，尽管我使用的是通用术语“COM”，但根据扩展，完全可用 Java 实现一个 ActiveX Automation 服务器，亦可在 Java 程序中使用一个 ActiveX 控件。

Java 和 COM 最引人注目的相似之处就是 COM 接口与 Java 的“interface”关键字的关系。这是接近完美的一种相符，因为：

■COM 对象揭示出了接口（也只有接口）

■COM 接口本身并不具备实施方案；要由揭示出接口的那个 COM 对象负责它的实施

■COM 接口是对语义上相关的一组函数的说明；不会揭示出任何数据

■COM 类将 COM 接口组合到了一起。Java 类可实现任意数量的 Java 接口。

■COM 有一个引用对象模型；程序员永远不可能“拥有”一个对象，只能获得对对象一个或多个接口的引用。Java 也有一个引用对象模型——对一个对象的引用可“造型”成对它的某个接口的引用。

■COM 对象在内存里的“生存时间”取决于使用对象的客户数量；若这个数量变成零，对象就会将自己从内存中删去。在 Java 中，一个对象的生存时间也由客户的数量决定。若不再有对那个对象的引用，对象就会等候垃圾收集器的处理。

Java 与 COM 之间这种紧密的对应关系不仅使 Java 程序员可以方便地访问 COM 特性，也使 Java 成为编写 COM 代码的一种有效语言。COM 是与语言无关的，但 COM 开发事实上采用的语言是 C++和 Visual Basic。同 Java 相比，C++在进行 COM 开发时显得更加强大，并可生成更有效的代码，只是它很难使用。Visual Basic 比 Java 简单得多，但它距离基础操作系统太远了，而且它的对象模型并未实现与 COM 很好的对应（映射）关系。Java 是两者之间一种很好的折衷方案。

接下来，让我们对 COM 开发的一些关键问题进行讨论。编写 Java/COM 客户和服务器时，这些问题是首先需要弄清楚的。

A.5.1 COM 基础

COM 是一种二进制规范，致力于实施可相互操作的对象。例如，COM 认为一个对象的二进制布局必须能够调用另一个 COM 对象里的服务。由于是对二进制布局的一种描述，所以

只要某种语言能生成这样的一种布局，就可通过它实现 COM 对象。通常，程序员不必关注象这样的一些低级细节，因为编译器可自动生成正确的布局。例如，假设您的程序是用 C++写的，那么大多数编译器都能生成符合 COM 规范的一张虚拟函数表格。对那些不生成可执行代码的语言，比如 VB 和 Java，在运行期则会自动挂接到 COM。
COM 库也提供了几个基本的函数，比如用于创建对象或查找系统中一个已注册 COM 类的函数。
一个组件对象模型的基本目标包括：
■让对象调用其他对象里的服务
■允许新类型对象（或更新对象）无缝插入环境
第一点正是面向对象程序设计要解决的问题：我们有一个客户对象，它能向一个服务器对象发出请求。在这种情况下，“客户”和“服务器”这两个术语是在常规意义上使用的，并非指一些特定的硬件配置。对于任何面向对象的语言，第一个目标都是很容易达到的——只要您的代码是一个完整的代码块，同时实现了服务器对象代码以及客户对象代码。若改变了客户和服务器对象相互间的沟通形式，只需简单地重新编译和链接一遍即可。重新启动应用程序时，它就会自动采用组件的最新版本。
但假若应用程序由一些未在自己控制之下的组件对象构成，情况就会变得迥然有异——我们不能控制它们的源码，而且它们的更新可能完全独立于我们的应用程序进行。例如，当我们在自己的程序里使用由其他厂商开发的 ActiveX 控件时，就会面临这一情况。控件会安装到我们的系统里，我们的程序能够（在运行期）定位服务器代码，激活对象，同它建立链接，然后使用它。以后，我们可安装控件的新版本，我们的应用程序应该仍然能够运行；即使在最糟的情况下，它也应礼貌地报告一条出错消息，比如“控件未找到”等等；一般不会莫名其妙地挂起或死机。
在这些情况下，我们的组件是在独立的可执行代码文件里实现的：DLL 或 EXE。若服务器对象在一个独立的可执行代码文件里实现，就需要由操作系统提供的一个标准方法，从而激活这些对象。当然，我们并不想在自己的代码里使用 DLL 或 EXE 的物理名称及位置，因为这些参数可能经常发生变化。此时，我们想使用的是由操作系统维护的一些标识符。另外，我们的应用程序需要对服务器展示出来的服务进行的一个描述。下面这两个小节将分别讨论这两个问题。

1. GUID 和注册表
COM 采用结构化的整数值（长度为 128 位）唯一性地标识系统中注册的 COM 项目。这些数字的正式名称叫作 GUID（Globally Unique IDentifier，全局唯一标识符），可由特殊的工具生成。此外，这些数字可以保证在“任何空间和时间”里独一无二，没有重复。在空间，是由于数字生成器会读取网卡的 ID 号码；在时间，是由于同时会用到系统的日期和时间。可用 GUID 标识 COM 类（此时叫作 CLSID）或者 COM 接口（IID）。尽管名字不同，但基本概念与二进制结构都是相同的。GUID 亦可在其他环境中使用，这里不再赘述。
GUID 以及相关的信息都保存在 Windows 注册表中，或者说保存在“注册数据库”（Registration Database）中。这是一种分级式的数据库，内建于操作系统中，容纳了与系统软硬件配置有关的大量信息。对于 COM，注册表会跟踪系统内安装的组件，比如它们的 CLSID、实现它们的可执行文件的名字及位置以及其他大量细节。其中一个比较重要的细节是组件的 ProgID；ProgID 在概念上类似于 GUID，因为它们都标识着一个 COM 组件。区别在于 GUID 是一个二进制的、通过算法生成的值。而 ProgID 则是由程序员定义的字串值。ProgID 是随同一个 CLSID 分配的。
我们说一个 COM 组件已在系统内注册，最起码的一个条件就是它的 CLSID 和它的执行文

件已存在于注册表中（ProgID 通常也已就位）。在后面的例子里，我们主要任务就是注册与使用 COM 组件。
注册表的一项重要特点就是它作为客户和服务器对象之间的一个去耦层使用。利用注册表内保存的一些信息，客户会激活服务器；其中一项信息是服务器执行模块的物理位置。若这个位置发生了变动，注册表内的信息就会相应地更新。但这个更新过程对于客户来说是“透明”或者看不见的。后者只需直接使用 ProgID 或 CLSID 即可。换句话说，注册表使服务器代码的位置透明成为了可能。随着 DCOM（分布式 COM）的引入，在本地机器上运行的一个服务器甚至可移到网络中的一台远程机器，整个过程甚至不会引起客户对它的丝毫注意（大多数情况下如此）。

2. 类型库
由于 COM 具有动态链接的能力，同时由于客户和服务器代码可以分开独立发展，所以客户随时都要动态侦测由服务器展示出来的服务。这些服务是用“类型库”（Type Library）中一种二进制的、与语言无关的形式描述的（就象接口和方法签名）。它既可以是一个独立的文件（通常采用.TLB 扩展名），也可以是链接到执行程序内部的一种 Win32 资源。运行期间，客户会利用类型库的信息调用服务器中的函数。
我们可以写一个 Microsoft Interface Definition Language（微软接口定义语言，MIDL）源文件，用 MIDL 编译器编译它，从而生成一个.TLB 文件。MIDL 语言的作用是对 COM 类、接口以及方法进行描述。它在名称、语法以及用途上都类似 OMB/CORBA IDL。然而，Java 程序员不必使用 MIDL。后面还会讲到另一种不同的 Microsoft 工具，它能读入 Java 类文件，并能生成一个类型库。

3. COM:HRESULT 中的函数返回代码
由服务器展示出来的 COM 函数会返回一个值，采用预先定义好的 HRESULT 类型。HRESULT 代表一个包含了三个字段的整数。这样便可使用多个失败和成功代码，同时还可以使用其他信息。由于 COM 函数返回的是一个 HRESULT，所以不能用返回值从函数调用里取回原始数据。若必须返回数据，可传递指向一个内存区域的指针，函数将在那个区域里填充数据。我们把这称为“外部参数”。作为 Java/COM 程序员，我们不必过于关注这个问题，因为虚拟机会帮助我们自动照管一切。这个问题将在后续的小节里讲述。

A.5.2 MS Java/COM 集成
同 C++/COM 程序员相比，Microsoft Java 编译器、虚拟机以及各式各样的工具极大简化了 Java/COM 程序员的工作。编译器有特殊的引导命令和包，可将 Java 类当作 COM 类对待。但在大多数情况下，我们只需依赖 Microsoft JVM 为 COM 提供的支持，同时利用两个有力的外部工具。
Microsoft Java Virtual Machine（JVM）在 COM 和 Java 对象之间扮演了一座桥梁的角色。若将 Java 对象创建成一个 COM 服务器，那么我们的对象仍然会在 JVM 内部运行。Microsoft JVM 是作为一个 DLL 实现的，它向操作系统展示出了 COM 接口。在内部，JVM 将对这些 COM 接口的函数调用映射成 Java 对象中的方法调用。当然，JVM 必须知道哪个 Java 类文件对应于服务器执行模块；之所以能够找出这方面的信息，是由于我们事前已用 Javareg 在 Windows 注册表内注册了类文件。Javareg 是与 Microsoft Java SDK 配套提供的一个工具程序，能读入一个 Java 类文件，生成相应的类型库以及一个 GUID，并可将类注册到系统内。亦可用 Javareg 注册远程服务器。例如，可用它注册在不同机器上运行的一个服务器。
如果想写一个 Java/COM 客户，必须经历一系列不同的步骤。Java/COM“客户”是一些特殊

的 Java 代码，它们想激活和使用系统内注册的一个 COM 服务器。同样地，虚拟机会与 COM 服务器沟通，并将它提供的服务作为 Java 类内的各种方法展示（揭示）出来。另一个 Microsoft 工具是 jactivex，它能读取一个类型库，并生成相应的 Java 源文件，在其中包含特殊的编译器引导命令。生成的源文件属于我们在指定类型库之后命名的一个包的一部分。下一步是在自己的 COM 客户 Java 源文件中导入那个包。
接下来让我们讨论两个例子。

A.5.3 用 Java 设计 COM 服务器
本节将介绍 ActiveX 控件、Automation 服务器或者其他任何符合 COM 规范的服务器的开发过程。下面这个例子实现了一个简单的 Automation 服务器，它能执行整数加法。我们用 setAddend()方法设置 addend 的值。每次调用 sum()方法的时候，addend 就会添加到当前 result 里。我们用 getResult()获得 result 值，并用 clear()重新设置值。用于实现这一行为的 Java 类是非常简单的：

1023 页上程序

为了将这个 Java 类作为一个 COM 对象使用，我们将 Javareg 工具应用于编译好的 Adder.class 文件。这个工具提供了一系列选项；在这种情况下，我们指定 Java 类文件名（"Adder"），想为这个服务器在注册表里置入的 ProgID（"JavaAdder.Adder.1"），以及想为即将生成的类型库指定的名字（"JavaAdder.tlb"）。由于尚未给出 CLSID，所以 Javareg 会自动生成一个。若我们再次对同样的服务器调用 Javareg，就会直接使用现成的 CLSID。

```
javareg /register
/class:Adder /progid:JavaAdder.Adder.1
/typelib:JavaAdder.tlb
```

Javareg 也会将新服务器注册到 Windows 注册表。此时，我们必须记住将 Adder.class 复制到 Windows\Java\trustlib 目录。考虑到安全方面的原因（特别是涉及程序片调用 COM 服务的问题），只有在 COM 服务器已安装到 trustlib 目录的前提下，这些服务器才会被激活。
现在，我们已在自己的系统中安装了一个新的 Automation 服务器。为进行测试，我们需要一个 Automation 控制器，而 Automation 控制器就是 Visual Basic（VB）。在下面，大家会看到几行 VB 代码。按照 VB 的格式，我设置了一个文本框，用它从用户那里接收要相加的值。并用一个标签显示结果，用两个下推按钮分别调用 sum()和 clear()方法。最开始，我们声明了一个名为 Adder 的对象变量。在 Form_Load 子例程中（在窗体首次显示时载入），会调用 Adder 自动服务器的一个新实例，并对窗体的文本字段进行初始化。一旦用户按下“Sum”或者“Clear”按钮，就会调用服务器中对应的方法。

1024 页程序

注意，这段代码根本不知道服务器是用 Java 实现的。
运行这个程序并调用了 CreateObject()函数以后，就会在 Windows 注册表里搜索指定的 ProgID。在与 ProgID 有关的信息中，最重要的是 Java 类文件的名字。作为一个响应，会启动 Java 虚拟机，而且在 JVM 内部调用 Java 对象的实例。从那个时候开始，JVM 就会自动接管客户和服务器代码之间的交流。

A.5.4 用 Java 设计 COM 客户

现在，让我们转到另一侧，并用 Java 开发一个 COM 客户。这个程序会调用系统已安装的 COM 服务器内的服务。就目前这个例子来说，我们使用的是在前一个例子里为服务器实现的一个客户。尽管代码在 Java 程序员的眼中看起来比较熟悉，但在幕后发生的一切却并不寻常。本例使用了用 Java 写成的一个服务器，但它可应用于系统内安装的任何 ActiveX 控件、ActiveX Automation 服务器或者 ActiveX 组件——只要我们有一个类型库。

首先，我们将 Jactivex 工具应用于服务器的类型库。Jactivex 有一系列选项和开关可供选择。但它最基本的形式是读取一个类型库，并生成 Java 源文件。这个源文件保存于我们的 windows/java/trustlib 目录中。通过下面这行代码，它应用于为外部 COM Automation 服务器生成的类型库：

```
jactivex /javatlb JavaAdder.tlb
```

Jactivex 完成以后，我们再来看看自己的 windows/java/trustlib 目录。此时可在其中看到一个新的子目录，名为 javaadder。这个目录包含了用于新包的源文件。这是在 Java 里与类型库的功能差不多的一个库。这些文件需要使用 Microsoft 编译器的专用引导命令：@com。jactivex 生成多个文件的原因是 COM 使用多个实体来描述一个 COM 服务器（另一个原因是我没有对 MIDL 文件和 Java/COM 工具的使用进行细致的调整）。

名为 Adder.java 的文件等价于 MIDL 文件中的一个 coclass 引导命令：它是对一个 COM 类的声明。其他文件则是由服务器揭示出来的 COM 接口的 Java 等价物。这些接口（比如 Adder_DispatchDefault.java）都属于“遣送”（Dispatch）接口，属于 Automation 控制器与 Automation 服务器之间的沟通机制的一部分。Java/COM 集成特性也支持双接口的实现与使用。但是，IDispatch 和双接口的问题已超出了本附录的范围。

在下面，大家可看到对应的客户代码。第一行只是导入由 jactivex 生成的包。然后创建并使用 COM Automation 服务器的一个实例，就象它是一个原始的 Java 类那样。请注意行内的类型模型，其中“例示”了 COM 对象（即生成并调用它的一个实例）。这与 COM 对象模型是一致的。在 COM 中，程序员永远不会得到对整个对象的一个引用。相反，他们只能拥有对类内实现的一个或多个接口的引用。

“例示”Adder 类的一个 Java 对象以后，就相当于指示 COM 激活服务器，并创建这个 COM 对象的一个实例。但我们随后必须指定自己想使用哪个接口，在由服务器实现的接口中挑选一个。这正是类型模型完成的工作。这儿使用的是“默认遣送”接口，它是 Automation 控制器用于同一个 Automation 服务器通信的标准接口。欲了解这方面的细节，请参考由 Ibid 编著的《Inside COM》。请注意激活服务器并选择一个 COM 接口是多么容易！

1026 页程序

现在，我们可以编译它，并开始运行程序。

1. com.ms.com 包

com.ms.com 包为 COM 的开发定义了数量众多的类。它支持 GUID 的使用——Variant（变体）和 SafeArray Automation（安全数组自动）类型——能与 ActiveX 控件在一个较深的层次打交道，并可控制 COM 异常。

由于篇幅有限，这里不可能涉及所有这些主题。但我想着重强调一下 COM 异常的问题。根

据规范，几乎所有 COM 函数都会返回一个 HRESULT 值，它告诉我们函数调用是否成功，以及失败的原因。但若观察服务器和客户代码中的 Java 方法签名，就会发现没有 HRESULT。相反，我们用函数返回值从一些函数那里取回数据。“虚拟机”（VM）会将 Java 风格的函数调用转换成 COM 风格的函数调用，甚至包括返回参数。但假若我们在服务器里调用的一个函数在 COM 这一级失败，又会在虚拟机里出现什么事情呢？在这种情况下，JVM 会认为 HRESULT 值标志着一次失败，并会产生类 com.ms.com.ComFailException 的一个固有 Java 异常。这样一来，我们就可用 Java 异常控制机制来管理 COM 错误，而不是检查函数的返回值。
如欲深入了解这个包内包含的类，请参考微软公司的产品文档。

A.5.5 ActiveX/Beans 集成

Java/COM 集成一个有趣的结果就是 ActiveX/Beans 的集成。也就是说，Java Bean 可包含到象 VB 或任何一种 Microsoft Office 产品那样的 ActiveX 容器里。而一个 ActiveX 控件可包含到象 Sun BeanBox 这样的 Beans 容器里。Microsoft JVM 会帮助我们考虑到所有的细节。一个 ActiveX 控件仅仅是一个 COM 服务器，它展示了预先定义好的、请求的接口。Bean 只是一个特殊的 Java 类，它遵循特定的编程风格。但在写作本书的时候，这一集成仍然不能算作完美。例如，虚拟机不能将 JavaBeans 事件映射成为 COM 事件模型。若希望从 ActiveX 容器内部的一个 Bean 里对事件加以控制，Bean 必须通过低级技术拦截象鼠标行动这类的系统事件，不能采用标准的 JavaBeans 委托事件模型。
抛开这个问题不管，ActiveX/Beans 集成仍然是非常有趣的。由于牵涉的概念与工具与上面讨论的完全相同，所以请参阅您的 Microsoft 文档，了解进一步的细节。

A.5.6 固有方法与程序片的注意事项

固有方法为我们带来了安全问题的一些考虑。若您的 Java 代码发出对一个固有方法的调用，就相当于将控制权传递到了虚拟机“体系”的外面。固有方法拥有对操作系统的完全访问权限！当然，如果由自己编写固有方法，这正是我们所希望的。但这对程序片来说却是不可接受的——至少不能默许这样做。我们不想看到从因特网远程服务器下载回来的一个程序片自由自在地操作文件系统以及机器的其他敏感区域，除非特别允许它这样做。为了用 J/Direct，RNI 和 COM 集成防止此类情况的发生，只有受到信任（委托）的 Java 代码才有权发出对固有方法的调用。根据程序片的具体使用，必须满足不同的条件才可放行。例如，使用 J/Direct 的一个程序片必须拥有数字化签名，指出自己受到完全信任。在写作本书的时候，并不是所有这些安全机制都已实现（对于 Microsoft SDK for Java，beta 2 版本）。所以当新版本出现以后，请务必留意它的文档说明。

A.6 CORBA

在大型的分布式应用中，我们的某些要求并非前面讲述的方法能够满足的。举个例子来说，我们可能想同以前遗留下来的数据仓库打交道，或者需要从一个服务器对象里获取服务，无论它的物理位置在哪里。在这些情况下，都要求某种形式的“远程过程调用”（RPC），而且可能要求与语言无关。此时，CORBA 可为我们提供很大的帮助。
CORBA 并非一种语言特性，而是一种集成技术。它代表着一种具体的规范，各个开发商通过遵守这一规范，可设计出符合 CORBA 标准的集成产品。CORBA 规范是由 OMG 开发出来的。这家非赢利性的机构致力于定义一个标准框架，从而实现分布式、与语言无关对象的相互操作。
利用 CORBA，我们可实现对 Java 对象以及非 Java 对象的远程调用，并可与传统的系统进

行沟通——采用一种“位置透明”的形式。Java 增添了连网支持，是一种优秀的“面向对象”程序设计语言，可构建出图形化和非图形化的应用（程序）。Java 和 OMG 对象模型存在着很好的对应关系；例如，无论 Java 还是 CORBA 都实现了“接口”的概念，并且都拥有一个引用（参考）对象模型。

A.6.1 CORBA 基础

由 OMG 制订的对象相互操作规范通常称为“对象管理体系”（ObjectManagement Architecture，OMA）。OMA 定义了两个组件：“核心对象模型”（Core Object Model）和“OMA 参考体系”（OMA Reference Model）。OMA 参考体系定义了一套基层服务结构及机制，实现了对象相互间进行操作的能力。OMA 参考体系包括“对象请求代理”（Object Request Broker，ORB）、“对象服务”（Object Services，也称作 CORBAservices）以及一些通用机制。

ORB 是对象间相互请求的一条通信总线。进行请求时，毋需关心对方的物理位置在哪里。这意味着在客户代码中看起来象一次方案调用的过程实际是非常复杂的一次操作。首先，必须存在与服务器对象的一条连接途径。而且为了创建一个连接，ORB 必须知道具体实现服务器的代码存放在哪里。建好连接后，必须对方法自变量进行“汇集”。例如，将它们转换到一个二进制流里，以便通过网络传送。必须传递的其他信息包括服务器的机器名称、服务器进程以及对那个进程内的服务器对象进行标识的信息等等。最后，这些信息通过一种低级线路协议传递，信息在服务器那一端解码，最后正式执行调用。ORB 将所有这些复杂的操作都从程序员眼前隐藏起来了，并使程序员的工作几乎和与调用本地对象的方法一样简单。并没有硬性规定应如何实现 ORB 核心，但为了在不同开发商的 ORB 之间实现一种基本的兼容，OMG 定义了一系列服务，它们可通过标准接口访问。

1. CORBA 接口定义语言（IDL）

CORBA 是面向语言的透明而设计的：一个客户对象可调用属于不同类的服务器对象方法，无论对方是用何种语言实现的。当然，客户对象事先必须知道由服务器对象揭示的方法名称及签名。这时便要用到 IDL。CORBA IDL 是一种与语言无关的设计方法，可用它指定数据类型、属性、操作、接口以及更多的东西。IDL 的语法类似于 C++或 Java 语法。下面这张表格为大家总结了三种语言一些通用概念，并展示了它们的对应关系。

CORBA IDL Java C++

模块（Module） 包（Package） 命名空间（Namespace）
接口（Interface） 接口（Interface） 纯抽象类（Pure abstract class）
方法（Method） 方法（Method） 成员函数（Member function）

继承概念也获得了支持——就象 C++那样，同样使用冒号运算符。针对需要由服务器和客户实现和使用的属性、方法以及接口，程序员要写出一个 IDL 描述。随后，IDL 会由一个由厂商提供的 IDL/Java 编译器进行编译，后者会读取 IDL 源码，并生成相应的 Java 代码。IDL 编译器是一个相当有用的工具：它不仅生成与 IDL 等价的 Java 源码，也会生成用于汇集方法自变量的代码，并可发出远程调用。我们将这种代码称为“根干”（Stub and Skeleton）代码，它组织成多个 Java 源文件，而且通常属于同一个 Java 包的一部分。

2. 命名服务

命名服务属于 CORBA 基本服务之一。CORBA 对象是通过一个引用访问的。尽管引用信息

用我们的眼睛来看没什么意义，但可为引用分配由程序员定义的字串名。这一操作叫作“引用的字串化”。一个叫作“命名服务”（Naming Service）的 OMA 组件专门用于执行“字串到对象”以及“对象到字串”转换及映射。由于命名服务扮演了服务器和客户都能查询和操作的一个电话本的角色，所以它作为一个独立的进程运行。创建“对象到字串”映射的过程叫作“绑定一个对象”；删除映射关系的过程叫作“取消绑定”；而让对象引用传递一个字串的过程叫作“解析名称”。
比如在启动的时候，服务器应用可创建一个服务器对象，将对象同命名服务绑定起来，然后等候客户发出请求。客户首先获得一个服务器引用，解析出字串名，然后通过引用发出对服务器的调用。
同样地，“命名服务”规范也属于 CORBA 的一部分，但实现它的应用程序是由 ORB 厂商（开发商）提供的。由于厂商不同，我们访问命名服务的方式也可能有所区别。

A.6.2 一个例子
这儿显示的代码可能并不详尽，因为不同的 ORB 有不同的方法来访问 CORBA 服务，所以无论什么例子都要取决于具体的厂商（下例使用了 JavaIDL，这是 Sun 公司的一个免费产品。它配套提供了一个简化版本的 ORB、一个命名服务以及一个“IDL→Java”编译器）。除此之外，由于 Java 仍处在发展初期，所以在不同的 Java/CORBA 产品里并不是包含了所有 CORBA 特性。
我们希望实现一个服务器，令其在一些机器上运行，其他机器能向它查询正确的时间。我们也希望实现一个客户，令其请求正确的时间。在这种情况下，我们让两个程序都用 Java 实现。但在实际应用中，往往分别采用不同的语言。

1. 编写 IDL 源码
第一步是为提供的服务编写一个 IDL 描述。这通常是由服务器程序员完成的。随后，程序员就可用任何语言实现服务器，只需那种语言里存在着一个 CORBA IDL 编译器。
IDL 文件已分发给客户端的程序员，并成为两种语言间的桥梁。
下面这个例子展示了时间服务器的 IDL 描述情况：

1031 页上程序

这是对 RemoteTime 命名空间内的 ExactTime 接口的一个声明。该接口由单独一个方法构成，它以字串格式返回当前时间。

2. 创建根干
第二步是编译 IDL，创建 Java 根干代码。我们将利用这些代码实现客户和服务器。与 JavaIDL 产品配套提供的工具是 idltojava：
idltojava -fserver -fclient RemoteTime.idl
其中两个标记告诉 idltojava 同时为根和干生成代码。idltojava 会生成一个 Java 包，它在 IDL 模块、RemoteTime 以及生成的 Java 文件置入 RemoteTime 子目录后命名。_ExactTimeImplBase.java 代表我们用于实现服务器对象的“干”；而_ExactTimeStub.java 将用于客户。在 ExactTime.java 中，用 Java 方式表示了 IDL 接口。此外还包含了用到的其他支持文件，例如用于简化访问命名服务的文件。

3. 实现服务器和客户

大家在下面看到的是服务器端使用的代码。服务器对象是在 ExactTimeServer 类里实现的。RemoteTimeServer 这个应用的作用是：创建一个服务器对象，通过 ORB 为其注册，指定对象引用时采用的名称，然后“安静”地等候客户发出请求。

1031-1033 页程序

正如大家看到的那样，服务器对象的实现是非常简单的；它是一个普通的 Java 类，从 IDL 编译器生成的“干”代码中继承而来。但在与 ORB 以及其他 CORBA 服务进行联系的时候，情况却变得稍微有些复杂。

4. 一些 CORBA 服务

这里要简单介绍一下 JavaIDL 相关代码所做的工作（注意暂时忽略了 CORBA 代码与不同厂商有关这一事实）。main()的第一行代码用于启动 ORB。而且理所当然，这正是服务器对象需要同它进行沟通的原因。就在 ORB 初始化以后，紧接着就创建了一个服务器对象。实际上，它正式名称应该是“短期服务对象”：从客户那里接收请求，“生存时间”与创建它的进程是相同的。创建好短期服务对象后，就会通过 ORB 对其进行注册。这意味着 ORB 已知道它的存在，可将请求转发给它。

到目前为止，我们拥有的全部东西就是一个 timeServerObjRef——只有在当前服务器进程里才有效的一个对象引用。下一步是为这个服务对象分配一个字串形式的名字。客户会根据那个名字寻找服务对象。我们通过命名服务（Naming Service）完成这一操作。首先，我们需要对命名服务的一个对象引用。通过调用 resolve_initial_references()，可获得对命名服务的字串式对象引用（在 JavaIDL 中是“NameService”），并将这个引用返回。这是对采用 narrow()方法的一个特定 NamingContext 引用的模型。我们现在可开始使用命名服务了。

为了将服务对象同一个字串形式的对象引用绑定在一起，我们首先创建一个 NameComponent 对象，用“ExactTime”进行初始化。“ExactTime”是我们想用于绑定服务对象的名称字串。随后使用 rebind()方法，这是受限于对象引用的字串化引用。我们用 rebind()分配一个引用——即使它已经存在。而假若引用已经存在，那么 bind()会造成一个异常。在 CORBA 中，名称由一系列 NameContext 构成——这便是我们为什么要用一个数组将名称与对象引用绑定起来的原因。

服务对象最好准备好由客户使用。此时，服务器进程会进入一种等候状态。同样地，由于它是一种“短期服务”，所以生存时间要受服务器进程的限制。JavaIDL 目前尚未提供对“持久对象”（只要创建它们的进程保持运行状态，对象就会一直存在下去）的支持。

现在，我们已对服务器代码的工作有了一定的认识。接下来看看客户代码：

1034 页程序

前几行所做的工作与它们在服务器进程里是一样的：ORB 获得初始化，并解析出对命名服务的一个引用。

接下来，我们需要用到服务对象的一个对象引用，所以将字串形式的对象引用直接传递给 resolve()方法，并用 narrow()方法将结果造型到 ExactTime 接口引用里。最后调用 getTime()。

5. 激活名称服务进程

现在，我们已分别获得了一个服务器和一个客户应用，它们已作好相互间进行沟通的准备。大家知道两者都需要利用命名服务绑定和解析字串形式的对象引用。在运行服务或者客户之

前，我们必须启动命名服务进程。在 JavaIDL 中，命名服务属于一个 Java 应用，是随产品配套提供的。但它可能与其他产品有所不同。JavaIDL 命名服务在 JVM 的一个实例里运行，并（默认）监视网络端口 900。

6. 激活服务器与客户

现在，我们已准备好启动服务器和客户应用（之所以按这一顺序，是由于服务器的存在是“短期”的）。若各个方面都设置无误，那么获得的就是在客户控制台窗口内的一行输出文字，提醒我们当前的时间是多少。当然，这一结果本身并没有什么令人兴奋的。但应注意一个问题：即使都处在同一台机器上，客户和服务器应用仍然运行于不同的虚拟机内。它们之间的通信是通过一个基本的集成层进行的——即 ORB 与命名服务的集成。

这只是一个简单的例子，面向非网络环境设计。但通常将 ORB 配置成“与位置无关”。若服务器与客户分别位于不同的机器上，那么 ORB 可用一个名为“安装库”（Implementation Repository）的组件解析出远程字串式引用。尽管“安装库”属于 CORBA 的一部分，但它几乎没有具体的规格，所以各厂商的实现方式是不尽相同的。

正如大家看到的那样，CORBA 还有许多方面的问题未在这儿进行详细讲述。但通过以上的介绍，应已对其有一个基本的认识。若想获得 CORBA 更详细的资料，最传真的起点莫过于 OMB Web 站点，地址是 http://www.omg.org。这个地方提供了丰富的文档资料、白页、程序以及对其他 CORBA 资源和产品的链接。

A.6.3 Java 程序片和 CORBA

Java 程序片可扮演一名 CORBA 客户的角色。这样一来，程序片就可访问由 CORBA 对象揭示的远程信息和服务。但程序片只能同最初下载它的那个服务器连接，所以程序片与它沟通的所有 CORBA 对象都必须位于那台服务器上。这与 CORBA 的宗旨是相悖的：它许诺可以实现“位置的透明”，或者“与位置无关”。

将 Java 程序片作为 CORBA 客户使用时，也会带来一些安全方面的问题。如果您在内联网中，一个办法是放宽对浏览器的安全限制。或者设置一道防火墙，以便建立与外部服务器安全连接。

针对这一问题，有些 Java ORB 产品专门提供了自己的解决方案。例如，有些产品实现了一种名为“HTTP 通道”（HTTP Tunneling）的技术，另一些则提供了特别的防火墙功能。

作为放到附录中的内容，所有这些主题都显得太复杂了。但它们确实是需要重点注意的问题。

A.6.4 比较 CORBA 与 RMI

我们已经知道，CORBA 的一项主要特性就是对 RPC（远程过程调用）的支持。利用这一技术，我们的本地对象可调用位置远程对象内的方法。当然，目前已有一项固有的 Java 特性可以做完全相同的事情：RMI（参考第 15 章）。尽管 RMI 使 Java 对象之间进行 RPC 调用成为可能，但 CORBA 能在用任何语言编制的对象之间进行 RPC。这显然是一项很大的区别。然而，可通过 RMI 调用远程、非 Java 代码的服务。我们需要的全部东西就是位于服务器那一端的、某种形式的封装 Java 对象，它将非 Java 代码“包裹”于其中。封装对象通过 RMI 同 Java 客户建立外部连接，并于内部建立与非 Java 代码的连接——采用前面讲到的某种技术，如 JNI 或 J/Direct。

使用这种方法时，要求我们编写某种类型的“集成层”——这其实正是 CORBA 帮我们做的事情。但是这样做以后，就不再需要其他厂商开发的 ORB 了。

A.7 总结

我们在这个附录讨论的都是从一个 Java 应用里调用非 Java 代码最基本的技术。每种技术都有自己的优缺点。但目前最主要的问题是并非所有这些特性都能在所有 JVM 中找到。因此，即使一个 Java 程序能调用位于特定平台上的固有方法，仍有可能不适用于安装了不同 JVM 的另一种平台。

Sun 公司提供的 JNI 具有灵活、简单（尽管它要求对 JVM 内核进行大量控制）、功能强大以及通用于大多数 JVM 的优点。到本书完稿时为止，微软仍未提供对 JNI 的支持，而是提供了自己的 J/Direct（调用 Win32 DLL 函数的一种简便方法）和 RNI（特别适合编写高效率的代码，但要求对 JVM 内核有很深入的理解）。微软也提供了自己的专利 Java/COM 集成方案。这一方案具有很强大的功能，且将 Java 变成了编写 COM 服务器和客户的有效语言。只有微软公司的编译器和 JVM 能提供对 J/Direct、RNI 以及 Java/COM 的支持。

我们最后研究的是 CORBA，它使我们的 Java 对象可与其他对象沟通——无论它们的物理位置在哪里，也无论是用何种语言实现的。CORBA 与前面提到的所有技术都不同，因为它并未集成到 Java 语言里，而是采用了其他厂商（第三方）的集成技术，并要求我们购买其他厂商提供的 ORB。CORBA 是一种有趣和通用的方案，但如果只是想发出对操作系统的调用，它也许并非一种最佳方案。