The Java Native Interface Programmer's Guide and Specification
Java 本地接口(JNI)编程指南和规范

序言

这本书涉及了"Java"本地接口(JNI)。如果你对下列情况感兴趣，这本书将对你有用:
.整合带有例如"C"或 C++"语言编写的传统代码的 Java 应用程序。
.用例如"C"或 C++"语言编写的已存在的程序来实现 Java 虚拟机。
.实现一个 Java 虚拟机(Java virtual machine)
.理解在语言互操作性上的技术说明，特别是怎样处理例如垃圾收集和多线程的特性。

首先，这本书是为开发者写的。在"JNI"各种特性上的丰富的各种讨论，和在怎样有效的使用"JNI"的有帮助的提示后，你将能发现很容易按部就班的开始使用"JNI"。"JNI"在 1997 年初初始发布。这本书总结了，在"Sun"微系统(Sun Microsystem)上工程师和还有在技术交流社区中大量的开发者，获得的两年经验。

第二，这本书体现了各种"JNI"特性的设计基本原理。不仅学术界感兴趣这个，而且十分透彻的实际理解也是高效使用 JNI 的先决条件。

第三，这个书的一部分是为"Java 2"平台的"JNI"定义规范。JNI 编程可以使用这个规范当作参考说明书。Java 虚拟机的实现必须按照规范来一致实现。

关于这个规范的评论或关于"JNI"的问题请发送到我们的地址邮件:jni@java.sun.com。为了最新的"Java 2"平台,或最新的"Java 2 SDK release"。请访问我们的网站 <<<http://java.sun.com>>>。为关于"Java Series"的更新信息包括这本的勘误表和将要出版书的预览，请访问<<<http://java.sun.com/Series>>>。

"JNI"的设计引来了在"Sun Microsystems"和 Java 技术授权之间的一些列争论。"JNI"是来自"Netscape"的 JRI(Java Runtime Interface)的部分进化而来，"JNI"是"Warren Harris"设计的。来自 Java 技术授权公司的许多人积极地参与了设计的讨论。他们包括 Russ Arun(Microsoft), Patrick Beard(Apple),Simon Nash(IBM),Ken Root(Intel),Ian Ellision-Taylor(Microsoft), and Mike Toutoghi(Microsoft)。

"JNI"的设计也大量地得益于 Sun 内部设计评论，这评论来自 Dave Bowen， James Gosling, Peter Kessler, Tim Lindholm, Mark Reinhold, Derek White and Frank Yellin。Dave Brown, Dave Connelly, James McIlree, Benjamin Renaud, andTom Rodrigues 对"JNI"在"Java 2 SDK 1.2"上的增强做出了有意义的贡献。在俄罗斯新西伯利亚(Novosibirsk)的兼容性测试的 Carla Schroer 的团队为"JNI"写了兼容性测试程序。在这过程中，他们发现了原始规范不清楚或不完整的地方。

"JNI"技术没有 Dave Bowen, Larry Abrahams, Dick Neiss, Jon Kannegaard,and Alan Baraz 的管理支持将不能被开发和部署。我得到来自我的经理 Dave Bowen 的强有力地支持和鼓励来写这本书。

Tim Lindholm，《The Java Virtual Machine Specification》的作者，在"JNI"被设计时，正主导 Java 虚拟机开发。Tim 在虚拟机和本机接口上做了引领性的工作，提倡"JNI"的使用和为这书增加了严密性和清晰度。为这本书的封面的厨房和餐厅的艺术设计，他也提供初始的草图。

这本书得益于许多同事的帮助。Anand Palaniswamy 写了第十章关于一般陷阱和缺陷(on common traps and pitfalls)的部分。Janet Keonig 细心地预读初始的草稿和贡献了许多哦有用的意见。Beth Stearns 根据在线的 JNI 指南写了第二章的草稿。

我从 Craig J.Bordelon, Michael Brundage, Mary Dageforde, Joshua Engel and Elliott Hughes 处得到关于这本书草稿有价值的评论。

Lisa Friendly, The Java Series 的编者，有助于这本书的编写和出版。Ken Arnold, The Java Programming Language 得到作者，首先提出了 JNI 书的编写。我要感谢在整个过程中 Mike hedrikson 和 Marina Lang 给的帮助和耐心在 Addison-Wesley 出版社。Diance 监督了生产流程从复制，编辑和最后的打印。

在过去的几年里，我和一群在 Sun Micorsystems 上的 Java 软件中有才能和奉献的人一起，有特权的工作，特别是 original，HotSpot and Sun Labs 虚拟机团队成员。这本书献给他们

第一部分，介绍和指南
(Part One: Introduction and Tutorial)

第一章 介绍

"JNI"是"Java"平台的一个强大的功能。使用"JNI"的应用程序能能混合用例如"C"和"C++"语言编写的本地代码(native code)，和用 Java 编程语言编写的代码。"JNI"允许编程人员，在不丢弃在传统编码上的投入，来利用 Java 平台功能。因为"JNI"是"Java"平台的一部分,编程人员立马解决互操作的问题，同时解决和 Java 平台的所有的实现一起工作的问题。

这本书是"JNI"的编程指导和参考说明。书包含三个部分:
.第二章通过简单的例子，介绍"JNI"。这个说明，是给不熟悉"JNI"的初始使用者的。
.第三章到第十章构建了编程者的指南，这给出了大量 JNI 功能的全面介绍。我们通过一系列短而详细描述的例子来展示"JNI"不同的功能,和展示被证明在"JNI"编程中有用的技术。
.第十一章到第十三章为所有的 JNI 类型和函数，展示明确的规范。这些章也被组织为参考说明服务。

这本书尝试吸引广泛的用户，他们对 JNI 不同需求的。这指南和编程引导是为初始编程者，但有经验的开发者和 JNI 的实现者可以找到更有用的参考章节在这指南和编程引导中。大多

数读者将可能是开发者，他们用"JNI"来写应用程序。这本书的"你"术语将暗指用"JNI"编程的开发者(program with the JNI)。相反是"JNI"实现者(JNI implementors)或使用"JNI"写的应用程序的终端用户。

这本书假设你有"Java，C 和 C++"编程语言的基本知识。如果没有，你可以参考许多有用的优秀的书本:Ken Arnold and James Gosling(Addison-Wesley,1998)写的"The Java Programming Language, Second Edittion", Brian Kernighan and Dennis Ritchie（Prentice Hall,1988)写的"The C Programming Language, Second Edition"和 Bjarne Stroustrup(Addison-Wesley 1997)写的"The C++ Programming Language, Third Edition".

这章剩余的部分介绍 JNI 的背景，作用和演化。

1.1 The Java Platform and Host Environment(Java 平台和主机环境)

这本书时间用 Java 编程语言和用本地编程语言(C,C++等(etc.))来写应用程序.让我们先为这些语言，明确正确的编程环境区域。

"Java"平台的编程环境包含"Java"虚拟机(VM)和 Java 应用程序编程的接口(Java Application Programming Interface(API))。"Java"应用程序是用"Java"编程语言编写的，被编译成一个独立于机器(machine-independent)二进制类格式.一个类在任何 Java 虚拟机上执行实现。Java 的 API 包含预定义的类集合。Java"平台的任何实现被假设支持 Java 编程语言，虚拟机和"API"。

主机环境(host environment)术语代表主机操作系统，本地库组，和 CPU 指令集。用本地变成语言(native programming languages)如"C"和"C++"编写本地应用程序(Native application)，被编译特定主机的二进制编码，和被连接到本地库。本地应用程序和本地库是典型依赖于特定主机环境。为一个操作系统建立的一个"C"应用程序，例如，典型不能呢工作在另一操作系统上。

"Java"平台被一般的配置在主机环境的上面。例如，"JRE"(Java Runtime Enviroment)是"Sun"产品，它支持"Java"平台运行在例如"Solaris"和"Windows"的存在操作系统上。"Java"平台提供一组特性，应用程序能依赖独立于底下的主机环境。

1.2 Role of the JNI(JNI 的作用)

当"Java"平台被配置在主机环境的顶层上，平台对于允许"Java"应用程序，带有用其他语言编写地本机编码工作，是需要或必须的。编程者已经开始采用 Java 平台来建立用"C"和"C++"编写的传统的应用程序。因为在遗留代码上存在投资价值，然而，"Java"应用程序将和"C"和"C++"代码共存许多年。

JNI 的一个强大的特性是允许你来利用"Java"平台,但利用的代码仍然用其他语言来编写。作为"Java"虚拟机(Jave virtual machine)执行的一部分，"JNI"是一个双向接口，允许 Java 应用程序调用本地代码，反之亦然(vice versa)。Figure 1.1(图 1.1）说明"JNI"的作用。

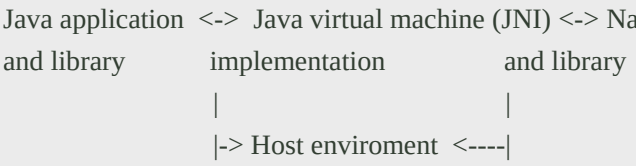


Figure 1.1 Role of JNI

"JNI"被设计为处理你需要联合"Java"应用程序和本地代码的情况。做为一个双向接口，"JNI"能支持两种本地编码:本地库(native libraries)和本地应用程序(native application)。
.你能使用"JNI"来编写，允许 Java 应用程序来调用在本地库中实现的函数的本地方法(native method)。"Java"应用程序，通过和他们调用在 Java 编程语言中实现的方法一样办法，调用本地方法。然而幕后，本地方法是用另一种语言实现的和位于本地库中。
."JNI"支持一个调用接口(invocation interface),这个接口允许你嵌入一个 Java 虚拟机执行到你本地应用程序中。本地应用程序能和执行 Java 虚拟机的本地库链接，同样因此用调用接口来执行用 Java 编程语言编写的软件控件。例如，一个用"C"写的"web"浏览器能在嵌入式 Java 虚拟机执行中执行下载"applets"。

1.3 Implications of Using the JNI(使用 JNI 的影响)

记住一旦一个应用程序使用了 JNI，它冒险失去了 Java 平台的两个好处。

首先,依赖"JNI"的"Java"应用程序不再可以在多种主机环境中运行。即使用 Java 编程语言编写的应程序的部分可移植到多种主机环境,它将需要重新编译用本地编程语言编写的应用程序的部分。

其次，Java 编程语言的类型安全和可靠，然而本地语言例如"C"或"C++"是没有的。因此，当你用 JNI 来写应用程序时你必须额外地细心。一个有恶意的本地方法可能破坏整个应用程序。因为这个原因，Java 应用程序在调用 JNI 特性前，接受了安全检查。

作为一般规则，你应该构建应用程序时，在尽量少的类中定义本地方法。这就意味着在本地代码和应用程序剩余代码之间有一个明显的隔离。

1.4 When to Use the JNI(什么时间使用 JNI）

在你从事一个使用"JNI"的工程前，后退一步调查一下是不是有根合理的替代方案是值得的。

像上一章提到的，当和严格用"Java"编程语言写的应用程序比时，使用"JNI"的应用程序有固有的缺点。例如，你失去 Java 编程语言的类型安全保证。

大量的可选方案也允许"Java"应用程序和其他语言编写的代码互操作的。例如:
."Java"应用程序通过一个"TCP/IP"链接和其他进程间通信机制(other inter-process communication（IPC) mechanisms)来和本地应用程序交流。
.一个"Java"应用程序可以通过"JDBC API"连接到原来的数据库上。
.一个"Java"应用程序可以利用分布式对象技术例如"Java IDL API"

这些替代方案的一般特征是"Java"应用程序和本地代码放在不同的进程中(在一些案例中在不同的机器上).进程隔离提供了一个重要的好处。进程提供了地址空间的保护能够最大限度的错误隔离，一个奔溃的本地应用程序不会立马终止通过"TCP/IP"和它通讯的"Java"应用程序。

然而有时，你可以发现对于"Java"应用程序必须和驻留在一样进程中(resides in the same process)的本地代码交流。这是"JNI"很有用的时候。思考，例如，下面的情况:
."Java API"可能不支持应用需要的某个依赖主机的特性。例如:一个应用程序可能需要执行特别的文件操作，但 Java API 不支持。然而通过另一个进程来操作文件是繁琐和低效的。
.你可能需要访问一个存在的本地库，不愿意花费在不同进程间的数据的拷贝和复制的开销。在同一个进程中载入本地库是更有效率的。
.横跨多个进程的应用程序可能导致不可接受的内存占用。如果这些进程需要驻留在同一个客户端机器上，这是真确的。载入一个本地库到存在本机进程中，应用程序需要比启动一个新的进程并载入这个库到新进程中需要的更少的系统资源。
.你可能需要在一个低级语言中执行一小段时间要求严格的代码(实时反应的代码)(time-critical code),例如汇编。如果一个"3D"增强应用程序消耗大量的时间在图形绘制中，你可以发现必须用汇编代码写图形库的核心部分来达到最好性能。

总之，如果你的 Java 应用程序必须和驻留在一个进程中的本地代码互操作，就得用"JNI".

1.5 Evolution of the JNI(JNI 的演化)
从"Java"平台的很早的时候，对于"Java"应用程序对和本地代码互操作的需要的就被认识。"Java"平台的第一版本，"JDK(Java Development Kit) release 1.0", 就包含一个本地方法接口。它允许Java应用程序调用其他语言如"C"和"C++"编写的函数。许多第三方的应用程序和Java类库的实现(例如，包扩 java.lang,java.io and java.net),都依赖本地方法接口来访问在底层主机环境中的特性。

不幸的是，在"JDK release 1.0"中的本地方法接口发现两个主要的问题：
.首先，本地代码，用 C 结构体的成员作为对象，来访问域。然而，Java 虚拟机规范没有定义对象在内存中怎样布置。如果一个 Java 虚拟机的实现以一种方法布局这些对象，不同于本地方法接口的设定方法，你必须重编译本地方法库。
.其次，在"JDK release 1.0"中本地方法的接口依赖于一个保守的垃圾收集器，因为本地方法

在虚拟机中得到对象的直接指针。任何使用更先进的垃圾收集算法的虚拟机的实现，不可能支持在"JDK release 1.0"中的本地方法接口。

设计"JNI"来克服这些问题。一个接口能被在不同主机环境中的所有的 Java 虚拟机的实现支持。有"JNI":
.每个虚拟机的实现者能支持很大体积的本地代码。
.开发工具商不必处理不同种类的本地接口。
.最重要的，应用程序编程者可以写一个版本的本地代码，这个版本将能在不同的 Java 虚拟机实现上运行。

"JNI"首先在"JDK release 1.1"中被支持。然而， "JDK release 1.1"内在仍然使用老风格的本地方法(和在"JDK release 1.0"中一样)来执行"Java APIs".在"Java 2 SDK release 1.2"中不再是这样(以前被称为"JDK release 1.2").本地方法被重写， 为了确定"JNI"标准。

"JNI"是被所有 Java 虚拟机实现支持的本机接口。从"JDK release 1.1"开始，你应该用"JNI"编程。老风格本地方法接口仍然在"Java 2 SDK release 1.2"中被支持，但将在未来高级的 Java 虚拟机实现中不被支持。

"Java 2 SDK relase 1.2"包含了大量的 JNI 增强性能。这个增强是向后兼容的。"JNI"的将来的演化将兼容保留所有的二进制。

1.6 Example Programs(例子程序)

这本书包含大量例子程序，它们来证明"JNI"特性。典型的例子吃呢供需包含多段用"Java"编程语言和"C"或"C++"本地编程语言编写的代码片段。有时本地编码参考了在 Solaris 和 Win32 系统中的本机特定特性。我们也显示了怎样使用 JDK 和 Java 2SDK release 带有的命令行工具(例如"javah")来建立"JNI"程序。

记住"JNI"的使用被限制在指定的主机环境或指定的应用开发工具中。这本书关心的是写代码，不是在使用工具编译和运行编码上。和"JDK"和"Java 2 SDK release"捆绑的命令行工具是很原始的。第三方的工具可以提供改善的方法编译使用 JNI 的应用程序。我们鼓励你参考和你选择的开发工具绑定的"JNI"相关的文档。

你能下载这本书中例子的源代码，和这本书的最新更新，来自下面 web 地址:
<<<http://java.sun.com/docs/books/jni>>>

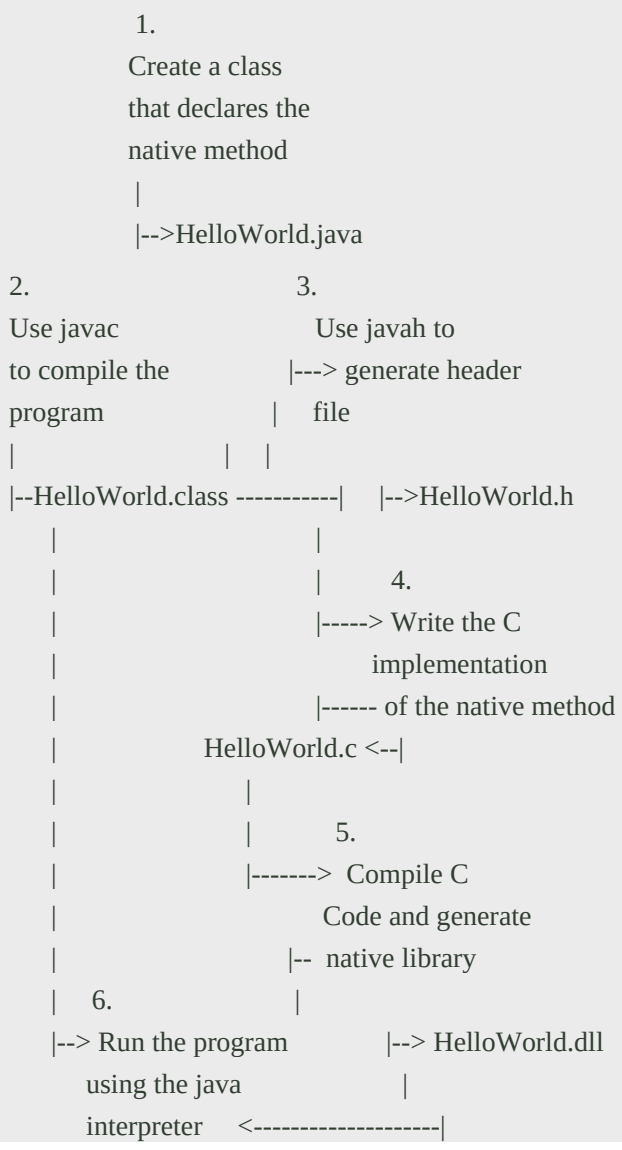
第二章 开始

这章通过一个使用"Java Native Interface(JNI)"的一个简单例子来引导你。我们将携一个"Java"应用程序,它调用一个"C"函数来打印"Hello World!"。

2.1 概要

"Figure 2.1"(图 2.1)说明这个进程为使用"JDK"或"Java 2 SDK releases"来写的一个简单的 Java 应用程序，它调用一个"C"函数打印"Hello World!"。这个过程包含以下几步:
1.创建一个类(HelloWorld.java)宣布一个本地方法。
2.用"javac"来编译"HelloWorld"源代码文件，产生"class"文件"HelloWorld.class"."javac"编译器是"JDK"或"Java 2 SDK releases"提供的。
3.使用"javah -jni"来产生一个"C"头文件(HelloWorld.h)，它包含函数的原型为本地方法的实现。"javah"工具是"JDK"或"Java 2 SDK releases"提供的。
4.写"C"的本地方法的实现(HelloWorld.c)。
5.编译"C"的实现为一个本地库，创建"HelloWorld.dll"或"libHelloWorld.so"文件。使用在本机环境中可用的"C"编译器和链接器。
6.使用"java runtime interpreter"(java 实时解释器)来运行"HelloWorld"程序。"class"文件(HelloWorld.class)和本地库(HelloWorld.dll or libHelloWorld.so)被实时的载入。

这章剩余的部分解释这些步骤的细节。



```
                    |
                    |--> "Hello World!"
```

Figure 2.1 Steps in Writing and Running the "Hello World" Program

2.2 申明本地方法(Declare the Native Method)

你通过用"Java"编程语言写下面的程序开始。这个程序定义一个类，名字为“HelloWorld"，包含一个本地方法"print"。

```
class HelloWorld{
 private native void print() ;
 public static void main(String[] args){
  new HelloWorld().print() ;
 }
 static{
  System.loadLibrary("HelloWorld");
 }
}
```

"HelloWorld"类定义开始申明了"print"本地方法。"main"方法实例化"HelloWorld"类，同时调用了"print"本地方法为这个实例。这个类定义的最后部分是一个静态初始化，来载入包含"print"本地方法实现的本地库。

在本地方法如"print"的声明和用 Java 编程语言声明的规则的方法之间有两个不同。本地方法的声明必须包含"native"修饰字符。这个"native"修饰字符指明这个方法用另一种语言来实现。本地方法声明是用分号终止的，声明终止符号，因为在这类自身中本地方法没有实现。我们将在单独的"C"文件中实现"print"方法。

在本地方法"print"被调用前，实现"print"的本地库必须被载入。这个例子中，我们在"HelloWorld"类的静态初始化中载入本地库。Java 虚拟机自动地运行静态初始化，在"HelloWorld"类中调用任何方法前，因此保证在"print"本地方法被调用前，本地库被载入。

我们定义了一个"main"方法能运行"HelloWorld"类。"HelloWorld.main"用和它调用一般规定方法一样方法调用本地方法"print"。

"System.loadLibrary"需要一个库名字，定位和这个名字相应的本地库，载入这个本地库到应用程序。我们将在这本书后面讨论详细地载入过程。现在简单地记住为了"System.loadLibrary("HelloWorld")"成功，我们必须创建在"Win32"上名叫"HelloWorld.dll"，或在"Solaris"上名叫"libHelloWorld.so"的本地库。

2.3 编译"HelloWorld"类

你已经定义了"HelloWorld"类后，保存源代码在"HelloWorld.java"文件中。然后用"JDK"或"Java 2 SDK release"自带的"javac"编译器编译源代码文件:

```
javac HelloWorld.java
```

这个命令将在当前的目录中产生一个"HelloWorld.class"文件。

2.4 创建本地方法的头文件

下一步我们将使用"javah"工具来产生一个"JNI"格式头文件，当用"C"语言来实现本地方法时这头文件是有用的。你能在 HelloWorld 类上运行"javah"，如下:

```
javah -jni HelloWorld
```

头文件名字是一个带有".h"的类名，".h"扩展它的结尾。上面显示的命令产生了一个名为"HelloWorld.h"的文件。我们这儿将不再完整地列出产生的头文件。头文件最重要的部分是 Java_HelloWorld_print 的函数原型，它是"C"函数实现了"HelloWorld.print"方法:

```
JNIEXPORT void JNICALL
Java_HelloWorld_print(JNIEnv *, jobject) ;
```

因为现在，忽略"JNIEXPORT"和"JNICALL"宏。我们可能已经注意到，即使虽然对应地本地方法的声明不接受参数，但是本地方法 C 的实现接受两个参数。对于每个本地方法的实现的第一个参数都是一个"JNIEnv"接口的指针第二个参数是"HelloWorld"对象自身的一个参考(在"C++"中就像"this"指针)。我们将在这本书的后面讨论怎样使用"JNIEnv"接口指针和"jobject"参数，但这个简单的例子忽略这两个参数。

2.5 写本地方法的实现

通过"javah"产生的"JNI"风格头文件帮助你为本地方法来写"C"或"C++"的实现。你写的函数必须按照在产生头文件中的详细定义的原型来。你能实现"HelloWorld.print"方法在"HelloWorld.c"的"C"文件中。如下(as follows):

```
#include <jni.h>
#include <stdio.h>
#include "HelloWorld.h"
JNIEXPORT void JNICALL
Java_HelloWorld_print(JNIEnv *env, jobject obj)
{
 printf("Hello World!\n") ;
 return ;
}
```

这个本地方法的实现是简单的。它使用"printf"函数来显示一个字符串"Hello World!"，然后放回。对于前面提到的俩那个参数，"JNIEnv"指针和对象(object)的参考被忽略。

这个"C"程序包含三个头文件:
.jni.h ---- 这个头文件提供了本地编码需要调用的"JNI"接口的信息。当写本地方法时，你必须总是包含这个头文件在你的"C"或"C++"源代码中。
.stdio.h ----上面的代码片段总是包含"stdio.h",因为它使用了"printf"函数。
.HelloWorld.h ---- 你用"javah"产生的头文件。它为 Java_HelloWorld_print 函数包含了"C/C++"原型。

2.6 编译 C 源代码和创建一本地库

记住当你在"HelloWorld.java"文件中创建"HelloWorld"类的时候，你包含了一行代码，它载入一个本地库到程序中:
System.loadLibrary("HelloWorld") ;

现在所有需要的"C"源代码都写了，你需要编译"HelloWorld.c"和建立本地库。

不同的操作系统支持不同的方法来建立本地库。在 Solaris，下面的 1 命令构建一个叫"libHelloWorld.so"的共享库：
cc -G -I/java/include -I/java/include/solaris HelloWorld.c -o libHelloWorld.so

"-G"可选命令"C"编译器来产生一个共享库来替代一个规定的"Solaris"的可执行文件。因为本书的篇幅的限制，我们把命令行打断成两行。你需要输入命令在单独一行，或放置这命令在一个脚本文件中。在 Win32 中，下面命令，来使用"Microsoft Visual C++"编译器,构建一个动态链接库(DLL)(dynamic link library）"HelloWorld.dll"：
cl -Ic:\java\include -Ic:\java\include\win32 -MD -LD HelloWorld.c -FeHelloWorld.dll

"-MD"可选项保证"HelloWorld.dll"被用"Win32"多线程"C"库来链接。"-LD"选项命令"C"编译器来产生一个"DLL"来替代一个规定的 Win32 可执行的文件。当然，在"Solaris"和"Win32"上，你需要输入包含的目录，它对应在你自己机器上安装目录。

2.7 运行程序

在这时，你为运行程序准备了两个控件。"class"文件(HelloWorld.class)调用本地方法，和本地库(HelloWorld.dll)实现本地方法。

因为"HelloWorld"类包含它自己的"main"方法，你可以在"Solaris"或"Win32"上运行，如下(as follows):
java HelloWorld

你应该看到如下输出:
Hello World!

重要的是为你程序运行时正确地设置本地库路径。这个本地库路径是在 Java 虚拟机搜索的列表目录中，当载入本地库时。如果你没有真确的建立本地库路径，你将看到和下面相识的一个错误:

```
java.lang.UnsatisfiedLinkError: no HelloWorld in library path
        at java.lang.Runtime.loadLibrary(Runtime.java)
        at java.lang.System.loadLibrary(System.java)
        at HelloWorld.main(HelloWorld.java)
```

确保本地库驻留在本地库路径中的一个目录中。如果你在"Solaris"系统上运行，"LD_LIBRARY_PATH"环境变量被用来定义本地库的路径。确保它包含了包含"libHelloWorld.so"文件的目录名字。如果"libHelloWorld.so"文件是在当前目录中。你可以发布如下两个命令在标准的 shell(sh)或 KornShell(ksh)来建立"LD_LIBRARY_PATH"恰当的环境变量:

```
LD_LIBRARY_PATH=.
export LD_LIBRARY_PATH
```

在 C shell(csh or tcsh)中同样的命令是像下面:

```
setenv LD_LIBRARY_PATH .
```

如果你运行在"Window 95"或"Windows NT"机器，你确保"HelloWorld.dll"是在当期目录下，或在你 PATH 环境变量的列表的目录中。

在"Java 2 JDK 1.2 release"中，你也能在"java 命令行"上详细说明本地库路径作为一个系统属性如下:

```
java -Djava.library.path=. HelloWorld
```

"-D"命令行选项设置一个 Java 平台系统属性。设置"java.library.path"属性为"."命令 Java virtual machine(Java 虚拟机)来在当前目录中搜索本地库。

第二部分：编程者的指南
(Part Two: Programmer's Guide)

第三章 基本类型，字符串和数组

(Basic Types, Strings, and Arrays)

当面对带有本地代码的 Java 的应用程序时，程序员问的最通常的问之一，是在 Java 编程语言中的数据类型怎样对映到本地编程语言例如"C"和"C++"中的数据类型。在上一章节中出现的"Hello World!"例子中，我们没有传递任何参数到本地方法中，本地方法没有放回任何结果。本地方法简单地答应了一个消息和放回。

实际上，大多数程序将需要传递参数给本地方法，和也从本地方法接受结果。在这章节中，我们将描述怎样转换数据类型在用 Java 编程语言写的代码和实现本地方法的本地代码中。我们将从基本的类型开始，如整型(intergers)和普通的对象类型，如字符串(stirngs)和数组(arrays).我们推迟任意对象的彻底解决到下一章，在下一章我们将解释本地代码能怎样访问域和调用方法。

3.1 一个简单本地方法(A Simple Native Method)

让我们从一个简单的例子开始，这个例子不同于在上一章的"HelloWorld"程序。这个例子程序，"Prompt.java",是一个打印字符串，等待用户输入，然后返回一行用户的输入的本地方法。这个程序的源代码如下(as follows):

```
class Prompt{
 // native method that prints a prompt and read a line
 private native String getLine(String prompt) ;

 public static void main(String args[]){
  Prompt p = new Prompt() ;
  String input = p.getLine("Type a line: ") ;
  System.out.println("User typed:"+input) ;
 }
 static {
  System.loadLibrary("Prompt") ;
 }
}
```

"Prompt.main"调用本地方法"Prompt.getLine"来得到用户的输入。静态初始化调用了"System.loadLibrary"方法来在载入一个本地库"Prompt"。

3.1.1 为实现本地方法的 C 原型(C Prototype for Implementing the Native Method)

"Prompt.getLine"方法能用下面的"C"函数来实现:

```
JNIEXPORT jstring JNICALL
Java_Prompt_getLine(JNIEnv *env, jobject this, jstring prompt);
```

你能用"javah"工具来产生一个包含上面函数原型的同文件。"JNIEXPORT"和"JNICALL"宏(被定义在"jni.h"头文件里)确保这个函数从本地库中导出和"C"编译器产生带有为这个函数正确调用约定的代码。C 函数的名字被格式为连接"Java_"前缀，类名和方法名。11.3 部分包含了一个更精确的怎样格式化 C 函数名字的描述。

3.1.2 本地方法参数(Native Method Arguments)

在 2.4 部分中被简单的讨论，本地方法实现如"Java_Prompt_getLine"接受两个标准参数，除了参数在本地方法中被声明外。第一参数，"JNIEnv"接口指针，指向一个包含指向函数表的地方。在函数表中每个条目都指向一个"JNI"函数.本地方法总是通过这些"JNI"函数的一个来访问在 Java 虚拟器中的数据结构。"Figure 3.1"说明"JNIEnv"接口指针。

```
JNIEnv *
  |
  |--->  pointer              ---->  Pointer ---> an interface function
         (Internal virtual           Pointer ---> an interface function
         machine data                Pointer ---> an interface function
         structures)                 ...
```

Figure 3.1 the JNIEnv Interface Pointer

对于本地方法是一个静态或是一个实例方法，第二个参数不同。对一个实例化的本地方法的第二个参数，是一个关于被调用方法的对象的参考，类似于在"C++"中的"this"指针。对于一个静态的本地方法的第二个参数，是一个定义这个方法的类的参考。我们的例子，"Java_Prompt_getLine",实现了一个实例化的本地方法。因此这个"jobject"参数是对象(object)自己的参考。

3.1.3 类型的映射(Mapping of Types)

在本地方法声明中参数类型有对应的在本地编程语言中的类型。"JNI"定义了一套"C"和"C++"类型来对应在"Java"编程语言中的类型。

在"Java"编程语言中的两种类型:基本来型如"int,float",和"char";和参考类型如"classes","instances"和"arrays".在"Java"编程语言中，strings 是"java.lang.String"类的一个实例。

"JNI"不同地对待基本类型和参考类型。基本类型的映射是简单易懂的。例如，在"Java"编程语言中的"int"类型映射到"C/C++"的"jint"类型(定义在"jni.h"作为一个有符号 32bit 整型),同时在"Java"编程语言中的"float"类映射到"C++"的"jfloat"类型(定义在"jni.h"作为一个有符号 32bit 浮点数),12.1.1 部分包含了在"JNI"中定义的所有基本类型的定义。

"JNI"传递"objects"到本地方法作为不透明的引用(opaque references)。不透明的引用是一个 C 指针类型，引用了在 Java 虚拟机中的内部数据结构。然而，内部数据结构的精确安排，对编程者是隐藏的。本地代码必须通过恰当的"JNI"函数处理下面的对象(objects)，"JNI"函数通

过"JNIEnv"接口指针是可用的。例如，为"java.lang.String"对应的"JNI"类型是"jstring"。一个"jstring"引用(reference)的确切值是和本地代码无关的。本地代码调用"JNI"函数例如"GetStringUTFChars"来访问一个 string 的内容。

所有的"JNI"引用都是类型 jobject。为了方便和增强类型的安全，"JNI"定义了一组引用类，它们概念上为"jobject"的子类型("subtypes").（A 是 B 的子类，A 的实例也是 B 的实例。)这些子类对应在"Java"编程语言中常用地引用类型。例如，"jstring"指示"strings";"jobjextArray"指示一组"objects"。12.1.2 部分包含"JNI"引用类型和其他关系的资料性的完整列表。

3.2 访问 Strings(Accessing Stirngs)

"Java_Prompt_getLine"函数接受"prompt"的一个"jstring"类型作为参数."jstring"类型表示在 Java 虚拟机中的"strings"，同时和规定的“C string"类型不同(指向字符的指针，char *).你不能用一个一般"C string"来作为一个"jstring"来使用。下面的代码，如果运行，它们不能产生想要的结果。事实上，它将很可能使 Java 虚拟机奔溃。

```
JNIEXPORT jstring JNICALL
Java_Prompt_getLine(JNIEnv *env, jobjext obj, jstring prompt）
{

 printf("%s", prompt) ;
 ......
}
```

3.2.1 转换到本地字串(Converting to Native Strings)

你的本地方法代码必须使用恰当的"JNI"函数来转化"jstirng objects"为"C/C++ strings"。"JNI"支持转换到或从"Unicode"和"UTF-8"的"strings"。"Unicode strings"表示字符是 16bit 的值，而"UTF-8 strings"使用了一编码规则，它享受兼容了"7-bit ASCII strings"。"UTF-8 strings"有想"C strings"空符号结尾，即使他们包含非"ASCII"（non-ASCII)字符.所有 7-bit ASCII 字符的值在 1 到 127 之间，在 UTF-8 编码中任然保留了。一个"byte"的最高"bit"被设置为 1,标记一个多"byte"编码"16-bit Unicode"的值的开始。

"Java_Prompt_getLine"函数调用"JNI"函数"GetStringUTFChars"来阅读"string"的内容。通过"JNIEnv"的接口指针 m"GetStringUTFChars"函数是能被调用的。它转换了作为一个"Unicode"序列通过 Java 虚拟机的实现来表示"jstring"的引用到用"UTF8" 格式表示的一个”C string"。如果你确定一个原始的字符只包含"7-bit ASCII"字符， 你可以传递转换字符串到到规定"C"库函数例如"printf"。(我们讨论怎样处理"non-ASII string" 在 8.2 部分。)

```
JNIEXPORT jstring JNICALL
Java_Prompt_getLine(JNIEvn *env, jobjext obj, jstring prompt)
{
 char buf[128] ;
 const jbyte *str ;
```

```
str = (*env)->GetStringUTFChars(env, prompt , NULL) ;
    if (NULL == str){
 return NULL;
 }
 printf("%s", str) ;
 (*env)->ReleaseStringUTFChars(env, prompt, str) ;
 scanf("%s", buf) ;
 return (*env)->NewStringUTF(env, buf) ;
}
```

不要忘记检查"GetStringUTFChars"的返回值。因为 Java 虚拟机实现需要分配空间来存储"UTF-8 string"，这有有机会内配失败。当这样的事发生时，"GetStringUTFChars"返回"NULL",同时通过"JNI"抛出一个异常,它不同于在 Java 编程语言中抛出的一个异常。通过"JNI"抛出的一个未决的异常不自动地改变在本地"C"代码中的控制流程。替代是，我们需要发布一个清楚的返回声明来在"C"函数中跳过剩余的语句。在"Java_Prompt_getLine"返回后，在"Prompt.main"中抛出这个异常，"Prompt.getLine"的本地方法的调用者。

3.2.2 释放本地字符资源(Freeing Native String Resources)

当你本地代码结束使用通过"GetStringUTFChars"得到的"UTF-8 string"时，它调用"ReleaseStringUTFChars"。调用"ReleaseStringUTFChars"指明本地方法不再需要这"GetStringUTFChars"返回的"UTF-8 string"了;因此"UTF-8 string"占有的内存将被释放。没有调用"ReleaseStringUTFChars"将导致内存泄漏，这将可能最终导致内存的耗尽。

3.2.3 构建新的字符串(Constructing New Strings)

在本地方法中通过调用"JNI"函数"NewStringUTF"，你能构建一个新的"java.lang.String"实例。"NewStringUTF"函数使用一个带有"UTF-8"格式的"C string"，同时构建一个"java.lang.String"实例。最新被构建的"java.lang.String"实例表现为和被给的"UTF-8 C string"一样的"Unicode"字符序列。

如果虚拟机没有内存分配来构建"java.lang.String"实例，"NewStringUTF"抛出一个"OutofMemoryError"异常，同时返回"NULL"。在这个例子中，我们不需要检查这个来放回值，因为本地方法过后立即返回了。如果"NewStringUTF"失败，"OutOfMemoryError"异常将在"Prompt.main"方法中被抛出，来说明本地方法的调用。。如果"NewStringUTF"成功，它返回一个"JNI"的最新构建的"java.lang.String"实例的引用。这最新实例被"Prompt.getLine"放回,然后赋给在"Prompt.main"中的"input"局部变量。

3.2.4 其他的 JNI 字符串函数(Other JNI String Functions)

"JNI"支持大量的其他字符串相关的函数(string-related functions),除了前面介绍的"GetStringUTFChars"，"ReleaseStringUTFChars"和"NewStringUTF"函数。

"GetStringChars"和"ReleaseStringChars"获得字符串使用"Unicode"格式。例如，在操作系统支持 Unicode 为本地字符串格式的时候，这些函数有用。

"UTF-8 string"总是以"\0"字符结尾，Unicode 字符不是。为在一个"jstring"引用中得到 Unicode 字符的个数，JNI 程序员能调用"GetStringLength"。为得到用"UTF-8"格式表示的一个"jstring"需要的"bytes"数,"JNI"程序员可以调用 ASCII C 函数 strlen 在"GetStringUTFChars"的结果上，或直接地调用"JNI"函数"GetStringUTFLength"在"jstring"引用上。

"GetStringChars"和"GetStringUTFChar"的第三个参数的需要额外的解释:

```
const jchar *
GetStringChars(JNIEnv *env, jstring str, jboolean *isCopy) ;
```

当来自"GetStringChars"返回时，通过"isCopy"被设置为"JNI_TRUE",返回的字符串是个在原始"java.lang.String"实例的字符串的一个复制，内存定位指向它。通过"isCopy"被设置为"JNI_FALSE",返回字符串时一个直接指向在原始的"java.lang.String"实例中的字符串，内存定位指向它。当通过"isCopy"被设置为"JNI_FALSE"，内存定位指向的字符串时，本地代码必须不能修改返回的字符串的内容。违反了这个规则将引起原始"java.lang.String"实例也被修改。这破坏了"java.lang.String"永恒不可变性。

你最常传递"NULL"作为"isCopy"参数，因为你不关心 Java 虚拟机是否返回在"java.lang.String"实例中的一个复制字符串和直接指向原始字符串。

一般不可能预测虚拟机将是不是复制字串对一个给定的"java.lang.String"实例。因此程序员必须假设函数如"GetStringChars"可以使用对应的空间和时间到在"java.lang.String"实例中的大量字符上。在一个典型 Java 虚拟器实现中，垃圾收集器搬迁堆上的对象。一旦直接指向一个"java.lang.String"实例的指针被传递给本地代码，垃圾收集器再不能搬移这"java.lang.String"实例了。换另一种说法，虚拟器必须固定住"java.lang.String"实例。因为过多地固定导致内存碎片，虚拟器实现可以，为灭个单独地"GetStringChars"调用，酌情地决定是复制字串还是固定实例。

不要忘记调用"ReleaseStringChars",当你不再需要访问来自"GetStringChars"返回的"string"元素的时候。"ReleaseStringChars"调用是必须的，无论"GetStringChars"设置"* isCopy"为"JNI_TRUE"或"JNI_FALSE"。"ReleaseStringChars"释放副本，或解除实例的固定，这依赖"GetStringChars"是返回一个副本还是指向实例。

3.2.5 在"Java 2 SDK Release 1.2"中新建 JNI 字符串函数 (New JNI String Functions in Java 2 SDK Release 1.2)

为了增加虚拟器能够返回直接指向在一个"java.lang.String"实例中的字符串的可能，"Java 2 SDK release 1.2"介绍了一对新函数，"Get/ReleaseStringCritical"。表面上，它们表现和"Get/ReleaseStringChars"函数相似。如果可能，在那两个函数中返回字符串的指针；否则产生一个副本。然而，这些函数怎样使用要有充分地限制。

你必须把这对函数中的代码放在一个排斥关键区域(critical region)中运行对待.在这个排斥关键域中，本地代码不应该调用任意"JNI"函数，或者任何可以引起当前线程阻塞和等待另一运行在 Java 虚拟器中的个线程的本地函数。例如，当前线程不应该等待在一个 I/O 流上的输入，这 I/O 是被另一个线程写入的。

这些限制使虚拟器能够暂停垃圾收集，当本地代码通过"GetStringCritical"来获得直接指向"string"的指针时。当垃圾收集暂停时，任何其他引发垃圾收集的线程也将本阻塞。在"Get/ReleaseStringCritical"对之间的本地代码，不应该出现阻塞调用或在 Java 虚拟器中分配新对象。否则，虚拟器可能锁死。思考下面的情况:
.另一线程出发垃圾收集不能进展下去，指导当前线程结束阻塞调用和使垃圾收集再能够。
.然而，当前线程不能进行下去，因为这阻塞调用需要获得一个已经被另一个等待执行垃圾收集的线程持有的锁。

交替的多对"GetStringCritical"和"ReleaseStringCritical"函数的使用时安全的。例如:

```
jchar *s1, *s2 ;
s1=（*env)->GetStringCritical(env, jstrl) ;
if ( s1 == NULL) {
 ....
}
.....
(*env)->ReleaseStringCritical(env, jstr1,s1) ;
(*env)->ReleaseStringCritical(env, jstr2,s2) ;
```

"Get/ReleaseStringCritical"对不需要严格的以一栈顺序来嵌套。我们不该忘记检查它们的返回值，阻止为内存状态可能是"NULL",因为如果 VM 内部用不同格式来表示数组，"GetStringCritical"可以分配一个"buffer"同时复制一个数组副本。例如，Java 虚拟器(VM)可以不连续地保存数组。在这个例子中，"GetStringCritical"必须复制在"jstring"实例中的说有的字符为了返回本地代码一个连续的字符数组。

为避免锁死，你必须确信本，在调用"GetStringCritical"后和在调用"ReleaseStringCritical"前，地代码没有调用任何 JNI 函数。在排斥关键域中被允许调用的"JNI"函数是重载的"Get/ReleaseStringCritical"和"Get/ReleasePrimitiveArrayCritical"调用。

"JNI"不支持"GetStringUTFCritical"和"ReleaseStringUTFCritical"函数。这样函数可能需要虚拟器来复制一份"string",因为虚拟器实现大都在内部用"Unicode"编码表示"strings"。

对于"Java 2 SDK release 1.2"的其他附加是"GetStringRegion"和"GetStringUTFRegion"。这些函数复制"string"元素到预分配的"buffer"中。"Prompt.getLine"方法可以使用"GetStringUTFRegion"来实现的如下:

```
JNIEXPORT jstring JNICALL
Java_Prompt_getLine(JNIEnv *env, jobject obj, jstring prompt）
{

 char outbuf[128], inbuf[128] ;
 int len = (*env)->GetStringLength(env, prompt) ;
 (*env)->GetStringUTFRegion(env, prompt, 0, len, outbuf) ;
 printf("%s", outbuf) ;
 scanf("%s", inbuf) ;
 return (*env)->NewStringUTF(env, inbuf) ;
}
```

"GetStringUTFRegion"函数使用一个开始索引和长度，这两个备用来计算 Unicode 字符的个数。函数也执行了边际检查，和如果必要放出"StringIndexOutOfBoundsException"。在上面代码中，我们从自己的"string"引用中得到长度，因此确保不会索引溢出。(然而上面的代码缺少必要检查来确保"prompt string"的字符少于 128。)

这个代码是比"GetStringUTFChars"稍微更简单的使用。因为"GetStringUTFRegion"不执行内存的分配，我们不需要检查可能没有内存的情况。(再说一次，上面的代码缺少必要检查来确保"prompt string"的字符少于 128。)

3.2.6 "JNI String"函数的总结(Summary of JNI String Functions)

Table 3.1 摘要所有字符串相关的"JNI"函数。"Java 2 SDK 1.2 release"增加一写新的函数，来增强对某些字符串操作的执行。这些增加的函数不支持新的操作，而是带来性能的改善。

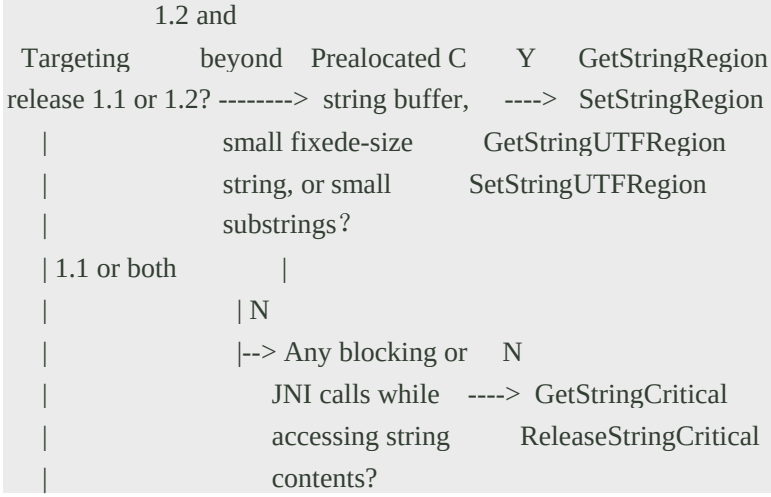
Table 3.1 Summary of JNI String Functions

JNI Function	Description	since
GetStringChars RetleaseStringChars	Obtains or release a pointer to the contents of a string in Unicode format. May return a copy of the string.	JDK1.1
GetStringUTFChars ReleaseStringUTFChars	Obtains or release a pointer to the contents of a string in UTF-8 format.	JDK1.1

	May return a copy of the string.	
GetStringLength	Returns the number of Unicode characters in the string.	JDK1.1
GetStringUTFLength	Returns the number of bytes needed (not including the trailing 0) to represent a string in the UTF-8 Format.	JDK1.1
NewString	Create a java.lang.String instance that contains the same sequence of characters as the given Unicode C string	JDK1.1
NewStringUTF	Create a java.lang.String instance that contains the same sequence of characters as the given UTF-8 encoded C string.	JDK1.1
GetStringCritical ReleaseStringCritical	Obtains a pointer to the contents of a string in Unicode format. May return a copy of theh string.Native code must not block between a pair of Get/ ReleaseStringCritical calls.	Java 2 SDK1.2
GetStringRegion SetStringRegion	Copies the content of a string to or from a preallocated C buffer in the Unicode format.	Java 2 SDK1.2
GetStringUTFRegion SetStringUTFRegion	Copies the content of a string to or from a preallocated C buffer in the UTF-8 format.	Java 2 SDK1.2

3.2.7 在字符串函数中选择(Choosing among the String Functions)

Figure 3.2 说明一个程序员可以怎样在"JDK release 1.1"和"Java 2 SDK release 1.2"上的字符串相关的函数中选择函数。



```
    |                           |
    |                           |Y
    |--> GetStringChars       <---------|
        ReleaseStringChars
        GetStringUTFChars
        ReleaseStringUTFChars
```
Figure 3.2 Choosing among the JNI String Functions

如果你目标是 1.1 或要 1.1 和 1.2 版本，没有选择除了"Get/ReleaseStringChars"和"Get/ReleaseStringUTFChars"。

如果你在 Java 2 SDK release 1.2 或更高的版本上编程，你需要复制字串的内容到一个已经存在"C buffer"中,请使用"GetStringRegion"或"GetStringUTFRegion"。

对于少的固定大小的字符串，"Get/SetStringRegion"和"Get/SetStringUTFRegion"几乎总是被执行函数，因为 C buffer 能被很容易在"C"栈来分配。复制在字符串中很少的字符开销可以忽略不计。

"Get/SetStringRegion"和"Get/SetStringUTFRegion"的一个好处就是它们不用执行内存的分配，因此不会产生不期望的没有内存的异常。不必没有异常检测，如果你确信没有索引溢出发生。

"Get/SetStringRegion"和"Get/SetStringUTFRegion"的另一好处你能指定开始的索引和复制的字符数目。如果本地代码只需要访问在长字符中的一部分，这函数很适合。

"GetStringCritical"必须特别小心使用。你必须保证在你通过"GetStringCritical"来得到一个指针时，本地代码在 Java 虚拟器中不能分配新的对象或执行另一个阻塞调用，这会引起系统的锁死。

这又个例子，示范使用"GetStringCritical"的精细说明。下面的代码获得"string"的内容和调用"fprintf"韩素来写出字符到文件句柄"fd"中:

```
const char *c_str =  （*env)->GetStringCritical(env, j_str, 0) ;
if ( c_str == NULL ){
  ...
}
fprintf(fd, "%s\n", c_str) ;
(*env)->ReleaseStringCritical(env, j_str, c_str) ;
```

带有上面代码的问题是，当当前线程使收集垃圾无效时，对于写文件句柄不总是安全的。例如,假设另一个线程"T"等待从文件 fd 句柄中读数据。让我们更远的设想，操作系统缓冲的建

立，是通过"fprintf"调用等待直到线程"T"从"fd"文件中完成读取所有未决的数据后的方法的。我们已经创建了一个有可能锁死情况:如果线程"T"不能分配足够内存来作为缓冲服务于从文件句柄中读数据，它必须请求一个垃圾收集，垃圾收集请求将被阻塞直到当前线程执行"ReleaseStringCritical",而这不会发生知道"fprintf"调用返回后。然而"fprintf"调用是等待的，因为线程"T"去完成从文件句柄中读数据。

下面代码，虽然相似于上面的例子，可几乎确定没有死锁:

```
const char *c_str =  （*env)->GetStringCritical(env, j_str, 0) ;
if ( c_str == NULL ){
  ...
}
DrawString(C_str) ;
(*env)->ReleaseStringCritical(env, j_str, c_str) ;
```

"DrawString"是一个系统调用，直接写字符到屏幕上。除非屏幕显示驱动也是一个在一样虚拟机中运行的 Java 应用程序。这个"DrawString"函数将不被阻塞指明第等待垃圾收集的发生。

总之，你需要考虑所有的可能阻塞行为在"Get/ReleaseStringCritical"成对的调用之间。

3.3 访问数组(Accessing Array)

"JNI"对待基本的数组和对象数组是不同地。基础数组包含原始是基本类型的例如"int"和"boolean"。对象数组(Object arrays)包含元素是应用类型例如 class 实例和其他数组。例如，在下面用 Java 编程语言下的代码片段中:

```
int[] iarr;
float[] farr;
object[] oarr ;
int[][] arr2 ;
```

"iarr"和"farr"是基本数组，然而"oarr"和"arr2"是对象数组。

在请求"JNI"函数使用的本地方法中，访问基本数组相似于用它们对"strings"访问。让我们看一个简单的例子。下面的代码调用一个本地方法"sumArray"来添加一个"int"数组的内容。

```
class IntArray{
 private native int sumArray(int[] arr) ;
 public static void main(String[] args){
  IntArray p = new IntArray() ;
  int arr[] = new int[10] ;
  for ( int i = 0 ; i < 10 ; i++ ){
   arr[i] = i ;
```

```
 }
 int sum = p.sumArray(arr) ;
 System.out.println("Sum = "+sum) ;
}
static {
 System.loadLibrary("IntArray") ;
}
}
```

3.3.1 在"C"中访问数组(Accessing Arrays in C)

数组通过"jarray"引用类型表示，和它的"subtypes"(子类)例如"jintArray"。就象"jstring"不是一个"C string"类型,"jarray"也不是一个"C"数组类型。你不能实现"Java_IntArray_sumArray"本地方法来非直接访问一个"jarray"引用。下面"C"代码是不合逻辑和不能产生需要的结果的:

```
JNIEXPORT jint JNICALL
Java_IntArray_sumArray(JNIEnv *env, jobject obj, jintArray arr)
{
jint i , sum = 0 ;
 for ( i = 0 ; i < 10 ; i++ ){
  sum += arr[i] ;
 }
}
```

你必须替代使用恰当"JNI"函数来访问基本数组(primitive aray)元素，就象在下面正确的例子中显示的:

```
JNIEXPORT jint JNICALL
Java_IntArray_sumArray(JNIEnv *env, jobject obj, jintArray arr)
{
 jint buf[10] ;
 int i , sum = 0 ;
 (*env)->GetIntArrayRegion(env, arr, 0, 10, buf) ;
 for ( i = 0 ; i < 10 ; i++ ){
  sum += arr[i] ;
 }
 return sum ;
}
```

3.3.2 方问基本类型数组(Accessing Arrays of Primitive Types)

前面的例子使用"GetIntArrayRegion"函数来复制所有的 integer 数组中的元素到一个"C"buffer(buf）。这第三个参数死元素的开始索引，同时第四个参数是被复制元素的数目。一旦元素在"C buffer"中,我们能在本地代码中访问它们。不需要异常检测，因为我们知道在我们例子中数组的长度是 10，因此没有索引的溢出。

"JNI"支持一个对应的"SetIntArrayRegion"函数，它允许本地代码来修改"int"类型的数组元素。其他基本类的数组(例如"boolean, short and float")也有类似的支持函数。

"JNI"支持一系列"Get/Release<Type>ArrayElements"函数(例如，包含"Get/ReleaseIntArrayElements"）,它们允许本地代码来得到一个直接指向基本类型数组元素的指针。因为底层的垃圾收集器可能不支持固定，虚拟器可能返回一个指向原始基本类型数组的一个副本。我们能够使用"GetIntArrayElements"来重写在 3.3.1 部分的本地方法的实现，如下:

```
JNIEXPORT jint JNICALL
Java_IntArray_sumArray(JNIEnv *env, jobjext obj, jintArray arr)
{
 jint *carr ;
 jint i, sum =0 ;
 carr = (*env)->GetIntArrayElements(env, arr, NULL ) ;
 if ( carr == NULL){
  return 0 ;
 }
 for (i = 0 ; i < 10 ; i++ ){
  sum += carr[i] ;
 }
 (*env)->ReleaseIntArrayElements(env, arr, carr, 0) ;
 return sum ;
}
```

"GetArrayLength"函数得到在基本类型或"object"类型数组中的元素的个数。当数组被第一次分配空间时，决定了一个数组的固定长度。

"Java 2 SDK release 1.2"介绍了"Get/ReleasePrimitiveArrayCritical"函数。这些函数允许虚拟器来使垃圾收集停止， 当本地代码访问基本类型数组的能容时候。编程者必须有和在使用"Get/ReleaseStringCritical"函数时同样的注意点。在"Get/ReleasePrimitiveArrayCritical"函数对之间，本地代码必须没有调用任意的"JNI"函数，或执行任何的阻塞操作，这些可以引起应用的死锁。

3.3.3 "JNI"基本类型数组函数概要(Summary of JNI Primitive Array Functions)

Table 3.2 是和基本类型数组相关的所有"JNI"函数的一个概要。"Java 2 SDK release 1.2"添加一些新的函数来增强对某些数组操作的执行。添加函数不是支持新的操作，而是带来执行性能的改善。

Table 3.2 Summary of JNI Primitive Array Functions

JNI Function	Description	since
Get<Type>ArrayRegion Set<Type>ArrayRegion	Copies the contents of primitive arrays to or from a preallicated buffer.	JDK1.1
Get<Type>ArrayElements Release<Type>ArrayElements	Obtains a pointer to the contents of a primitive array.May return a copy of the array.	JDK1.1
GetArrayLength	Returns the number of elements in the array.	JDK1.1
New<Type>Array	Creates an array with the given length.	JDK1.1
GetPrimitiveArrayCritical ReleasePrimitiveArrayCritical	Obtain or releases a pointer to the contents of a primitive array. May disable garbage collection, or return a copy of the array.	Java 2 SDK1.2

3.3.4 在基本类型数组函数中选择(Choosing among the Primitive Array Functions)

Figure 3.3 说明一个编程者可以怎样在"JDK release 1.1"和"Java 2 SDK release 1.2"的访问基本类的数组的 JNI 函数中选择函数：

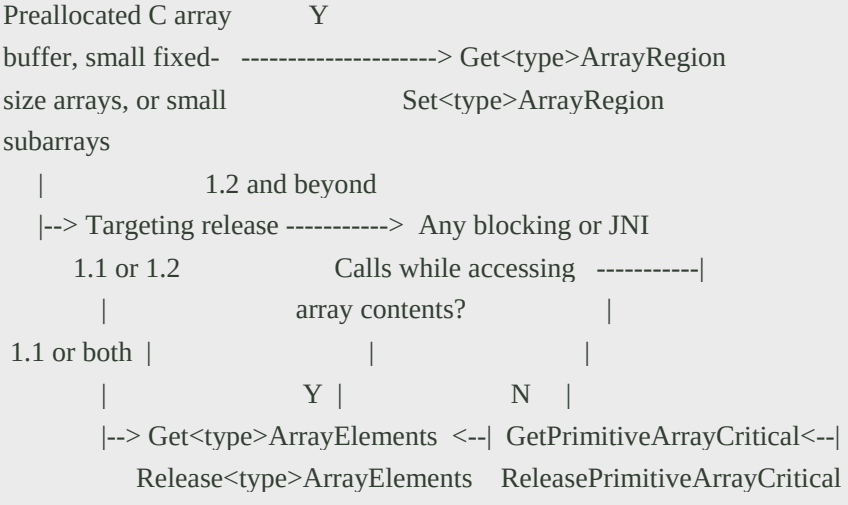


Figure 3.3 Choosing among Primitive Array Functions

如果你需要复制到或来自一个预分配的"C buffer"，就使用"Get/Set<Type>ArrayRegion"系列函数。这些函数执行了边界的检查，和在需要时产生"ArrayIndexOutOfBoundsException"异

常。在 3.3.1 部分中本地方法实现使用了"GetIntArrayRegion"来复制出自一个 jarray 引用的 10 个元素。

对于小的固定大小的数组，"Get/Set<Type>ArrayRegion"几乎总是首选函数，因为"C buffer"能在"C stack"上很容易地分配。复制一个少量数组元素的开销可以忽略。

"Get/Set<Type>ArrayRegion"函数允许你指定开始的索引和元素的个数，因此如果本地代码需要访问在一个大数组总的一部分元素，这是最好的函数。

如果你没有预分配的"C buffer"，基本类型的数组来决定大小，同时本地代码在得到数组元素的指针时没有调用阻塞调用，我们使用在"Java 2 SDK release 1.2"中的"Get/ReleasePrimitiveArrayCritical"函数。就象"Get/ReleaseStringCritical"函数，"Get/ReleasePrimitiveArrayCritical"函数必须格外小心使用，为了避免死锁。

使用"Get/Release<type>ArrayElements"系列函数总是安全的。虚拟器返回一个直接指向数组元素的指针，或返回一个缓冲保存这个数组元素的副本。

3.3.5 访问 Objects 类型数组(Accessing Arrays of Objects)

"JNI"提供单独一对函数来访问"Objects"类型数组。"GetObjectArrayElement"函数返回被给的一个索引的元素，然而"SetObjectArrayElement"函数更新被给的一个索引的元素。不象基本类型数组情况，你不能立马得到所有"objext"类型的元素或复制多个"object"类型的元素。

字符串(Stings)和数组(arrays)都是类型的引用。你使用"Get/SetObjectArrayElement"来访问字符串和数组的数组。

下面的例子调用一个本地方法来创建一个二维 int 类型数组，然后打印这个数组的内容。

```
class ObjectArrayTest{
 private static native int[][] initInt2DArray(int size) ;
 public static void main(String[] args){
  int[][] i2arr = initInt2DArray(3) ;
  for (int i = 0 ; i < 3 ; i++ ){
   for (int j = 0 ; j < 3 ; j++ ){
    System.out.print(" "+ i2arr[i][j]) ;
   }
  }
 }
 static{
  System.loadLibrary("ObjectArrayTest") ;
```

```
 }
}
```

这个静态本地方法"initInt2DArray"创建了一个被给大小的二维数组。这个本地方法分配和初始化了这二维的数组，如下:

```
JNIEXPORT jobjectArray JNICALL
Java_ObjextArrayTest_initInt2DArray(JNIEnv *env, jclass cls, int size)
{
 jobjectArray result;
 int i ;
 jclass intArrCls =  （*env)->FindClass(env, "[I"）;
 if (intArrCls == NULL）｛
  return NULL ;
 }
 result =  （*env)->NewObjectArray(env, size, intArrCls, NULL) ;
 if (result == NULL ){
  return NULL ;
 }
 for( i = 0 ; i < size ; i++）｛
  jint tmp[256] ;
   jint j ;
  jintArray iarr = (*env)->NewIntArray(env, size) ;
  if ( iarr == NULL){
   return NULL ;
  }
  for (j = 0 ; j < size ; j++ ){
   tmp[j] = i + j ;
  }
  (*env)->SetIntArrayRegion（env, iarr, 0, size, tmp) ;
  (*env)->SetObjectArrayElement(env, result, i, iarr) ;
  (*env)->DeleteLocalRef(env, iarr) ;
 }
 return result ;
}
```

"newInt2DArray"方法首先调用了"JNI"函数"FindClass"来得到一个二维 int 类型数组的元素类型的引用。"FindClass"的"[I"参数是"JNI class descriptor(描述符)",对应在 Java 编程语言中的"int[]"类型。如果类型载入失败，"FindClass"返回"NULL"同时抛出一个异常（例如，在一个错误类型文件或者没有存储空间的情况)

下一个"NewObjextArray"函数分配一个数组，数组元素类型被用"intArrCls"类型引用来指示。"NewObjectArray"函数只分配了第一个维度，我们剩下构建二维来填充数组元素任务。Java 虚拟机没有为多维数组指定特定的数据结构。一个二维的数组是一个简答的数组的数组。

创建二维的代码是相当简单的。"NewIntArray"分配了各个数组的元素，同时"SetIntArrayRegion"复制"temp[] buffer"的内容到新分配的一维数组中。在完成"SetObjectArrayElement"调用，第 i 个一维数组的第 j 个元素值是"i+j"。

运行"ObjectArrayTest.main"方法产生如下输出:

0 1 2
1 2 3
2 3 4

在循环最后的"DeleteLocalRef"调用确保虚拟器不会用光被用来持有 JNI 参考例如"iarr"的内存。5.2.1 部分详细解释什么时候和为什么你需要调用"DeleteLocalRef"。

第四章 成员和方法(CHAPTER 4 Fields and Methods)

现在你知道"JNI"怎样让本地代码访问基本类型和类型引用例如"strings"和"arrays"，下一步将是学习怎样和在任意对象(objects）中的成员(fields)和方法(methods)来交互。除了访问成员域，还包含从本地代码中调用用 Java 编程语言实现的方法，一般被认为从本地代码中执行回调(performing callbacks)。

我们将通过介绍支持成员域访问和方法回调的 JNI 函数开始。这章节的后面，我们将讨论怎样通过使用一个简单但有效的缓冲技术(simple buf effective caching technique)来更有效的做如此操作。在最后章节，我们将讨论调用本地方法的性能特点，和从本地方法中访问成员域和调用方法的性能特点。

4.1 访问成员域(Accessing Fields)

Java 编程语言支持两种类型的成员域。一个类的每个实例有这个类型的实例成员域(the instance fields of the class)的自己副本，还有一个类的所有实例共享这个类的静态成员域(static fields of the class)。

JNI 提供函数，本地代码能使用用它来得到和设置在对象(objects)中的实例成员域和在类中的静态成员域。让我们首先看一个例子程序，这程序说明怎样从一个本地方法的实现中访问实例成员域。

```
class InstanceFieldAccess{
 private String s ;
```

```
private native void accessField() ;
public staic void main(String args[]){
 InstanceFieldAccess c = new InstanceFieldAccess() ;
 c.s = "abc" ;
 c.accessField() ;
 System.out.println("In Java:") ;
 System.out.println("  c.s = ""+c.s+""") ;
}
static{
 System.loadLibrary("InstaceFieldAccess") ;
}
}
```

"InstanceFieldAccess"类定义一个实例成员域"s"。"main"方法创建了一个对象(object),设置实例成员域，然后调用本地方法"InstaceFieldAccess.accessField"。正如我们将要很快看到的，本地方法答应出实例成员域存在的值，然后设置成员域为一个新的值。在本地方法返回后，程序又打印这个成员域值，证明这个成员域值研究真正地改变了。

"InstanceFieldAccess"本地方法的实现。

```
JNIEXPORT void JNICALL
Java_InstanceFieldAccess_accessField(JNIEnv *env, jobject obj)
{
 jfieldID fid ;
 jstring jstr ;
 const char *str ;

 jclass cl = (*env)->GetObjectClass(env, obj) ;
 printf("In C:\n") ;


 fid = (*env)->GetFieldID(env, cls, "s", "Ljava/lang/String;") ;

 if( fid == NULL ){
  return ;
 }

 jstr = (*env)->GetObjectField(env, obj, fid) ;
 str = (*env)->GetStringUTFChars(env, jstr, NULL) ;
 if( str == NULL ){
  return ;
 }
```

```
 printf(" c.s = "%s"\n", str ) ;
 (*env)->ReleaseStringUTFChars(env, jstr, str) ;

 jstr = (*env)->NewStringUTF(env, "123") ;
 if( jstr == NULL ){
  return ;
 }
 (*env)->SetObjectField(env, obj, fid, jstr) ;
}
```

运行带有"InstanceFieldAccess"本地库的"InstanceFieldAccess"类产生下面的输出:

```
In C:
  c.s = "abc"
In Java:
  c.s = "123"
```

4.1.1 访问一个实例成员域的成果(Procedure for Accessing an Instance Field)

为了访问一个实例域，本地方法分两步处理。首先，他调用"GetFieldID"从类引用，成员域的名字，和成员域描述符，来得到成员域(field)ID:

```
fid =  （*env)->GetFieldID(env, cls, "s", "Ljava/lang/String;"）  ;
```

这个例子代码通过调用"GetObjectClass"在实例引用对象(obj)上得到类的引用，"obj"被作为第二个参数传递给本地方法实现。

一旦你已经得到成员域"ID",你能够传递对象引用(object reference)和成员域 ID(Field ID)给相应的实例域访问函数:

```
jstr = (*env)->GetObjectField(env, obj, fid) ;
```

因为"strings"和"arrays"是个特别的对象类型，我们使用"GetObjectField"来访问的这个实例域是一个字符串(string)。除了"Get/SetObjectField","JNI"也支持其他函数例如"GetIntField"和"SetFloatField"来访问基本类型的实例成员域。

4.1.2 成员域的描述符(Field Descriptors)

你可能已经注意到在前面的部分中，我们使用一个特殊的编码"C"字符串"Ljava/lang/String"来表示一个成员域在"Java"编程语言中的类型。这些"C strings"被称为"JNI"成语域描述符(JNI field descriptors)。

成员域声明的类型决定了这个"string"的内容。例如，你用"I"来代表一个"int"成员域，用"F"来代表一个"float"域，用"D"来代表一个"double"成员域，用"Z"来代表一个"boolean"成员域，等等(and so on)。

一个类型引用的描述符，例如"java.lang.String",带有字符"L"开始，它被 JNI 类描述符跟着，同时以一个分号(semicolon）结束。在完全合格的类名字中的"."分割符被变成在"JNI"类描述符中的"/"。因此。你为类型为"java.lang.String"的成员域建立了成员域描述符,如下:
"Ljava/lang/String;"

对于数组类型的描述符包含"["字符，被数组的组成类型的描述符跟着。例如,"[I"是"int[]"成员域类型的描述符。12.3.3 部分包含成员域描述符的详细信息，同时在 Java 程序语言中它们匹配的类型。

你能使用"javap"工具(随带"JDK"或"Java 2 SDK releases"发布)来从类的文件产生成员域描述。一般地，"javap"打印出在给出的类中的方法和成员类型。如果你指定了"-s"可选项(和为导出私有成员的"-p"可选项),"javap"打印 JNI 描述发来替代:
javap -s -p InstanceFieldAccess

这给你输出包含成员域"s"的"JNI"描述符：
...
s Ljava/lang/String
...

使用"javap"工具帮助消除错误，错误可能发生来自手工的"JNI"描述符字符串。

4.1.3 访问静态成员域(Accessing Static Fields)

访问静态成员域和访问实例成员域是相似的。让我们看一个"InstanceFieldAccess"例子的小的变化:

```
class StaticFieldAccess{
 private static int si ;
 private native void accessField() ;
 public static void main(String args[]){
  StaticFieldAccess c = new StaticFieldAccess() ;
  StaticFieldAccess.si = 100 ;
  c.accessField() ;
  System.out.println("In Java:") ;
  System.out.println("  StaticFieldAccess.si ="+si) ;
 }
 static{
  System.loadLibrary("StaticFieldAccess") ;
```

```
 }
}
```

"StaticFieldAccess"类包含一个静态整数成员域"si"。"StaticFieldAccess.main"方法创建了这个对象(object），初始化了静态成员域，然后调用了本地方法"StaticFieldAccess.accessField"。很快我们将看到，本地方法打印出静态成员域的存在的值，然后设置这个成语域为一个新值。为了验证成员域被真正地改变，在本地方法返回后，程序又打印了静态成员域的值。

"StaticFieldAccess.accessField"本地方法的实现。

```
JNIEXPORT void JNICALL
Java_StaticFieldAccess_accessField(JNIEnv *env, jobject obj)
{
 jfieldID fid ;
 jint si ;

 jclass cls =  （*env)->GetObjectClass(env, obj) ;
 printf("In C:\n") ;


 fid = (*env)->GetStaticFieldID(env, cls, "si", "I") ;
 if( fid == NULL ){
  return ;
 }

 si = (*env)->GetStaticIntField(env, cls, fid) ;
 printf("  StaticFieldAccess.si = %d\n", si) ;
 (*env)->SetStaticIntField(env, cls, fid, 200) ;
}
```

运行带有本地库的程序产生下面的输出:

```
In C:
   StaticFieldAccess.si = 100
In Java:
   StaticFieldAccess.si = 200
```

你怎样访问一个静态成员域和怎样访问一个实例成员域之间有两个不同:
1.你调用"GetStaticFieldID"来得到静态成员域，相对应"GetFieldID"来得到实例成员域。"GetStaticFieldID"和"GetFieldID"有一样的返回类型"jfieldID"。
2.一旦你已经得到静态成员域"ID",你传递了类的引用，相对于一个对象引用， 来给适当静态成员域访问函数。

4.2 调用方法(Calling Methods）

在"Java"编程语言中有几种类型的方法。实例方法必须在一个类的具体实例上被调用，然而静态方法可以不依赖任何实例被调用。我们将推迟构建(constructors)的讨论到下一部分中。

"JNI"支持一整套函数来允许你执行来自本地代码的回调。这个下面程序例子包含一个本地方法，反过来调用了用 Java 编程语言实现的一个实例方法。

```
class InstanceMethodCall{
 private native void nativeMethod() ;
 private void callback(){
  System.out.println("In Java") ;
 }
 public static void main(String args[]){
  InstanceMethodCall c= new InstanceMethodCall() ;
  c.nativeMethod() ;
 }
 static{
  System.loadLibrary("InstanceMethodCall") ;
 }
}
```

这儿是本地方法的实现:

```
JNIEXPORT void JNICALL
Java_InstanceMethodCall_nativeMethod(JNIEnv *env, jobject obj)
{
 jclass cls = (*env)->GetObjectClass(env, obj）  ;
 jmethod mid =  （*env)->GetMethodID(env, cls, "callback", "()V") ;
 if(mid == NULL){
  return;
 }
 printf("In C\n") ;
 (*env)->CallVoidMethod(env, obj, mid) ;
}
```

运行上面程序产生下面的输出:

```
In C
In Java
```

4.2.1 调用实例方法(Calling Instance Methods)

"Java_InstanceMethodCall_nativeMethod"实现说明了两步来请求调用一个实例方法:

.本地方法首先调用"JNI"函数"GetMethodID"。"GetMethodID"为在被给的类中方法执行一个

查询。查询是依靠名字和方法的类型描述符。如果方法没有，"GetMethodID"返回"NULL"。在这点，从本地代码的一个立马返回产生一个"NoSuchMethodError"异常，在调用"InstanceMethodCall.nativeMethod"的代码中被抛出。
.然后本地代码调用"CallVoidMethod"。"CallVoidMethod"调用一个实例方法，它返回一个 void 类型。你传递一个对象(object)，方法 ID 和实际参数(虽然在上面的例子中没有)给"CallVoidMethod"。

除了"CallVoidMethod"函数，"JNI"也支持带有其他类型返回的方法调用函数。例如，如果你回调方法返回一个"int"类型的值,你的本地方法应该使用"CallIntMethod"。类似地，你能使用"CallObjectMethod"来调用方法，放回一个对象(objects），它包含"java.lang.String"实例和数组。

你最好能使用"Call<Type)Method"系列函数来调用接口方法。你必须从接口类型来得到方法"ID"。例如，下面的代码片段调用"Runnable.run"方法在一"java.lang.Thread"实例上:

```
jobject thd = ... ;
jmethodID mid ;
jclass runnableIntf =  （*env)->FindClass(env, "java/lang/Runnable") ;
if ( runnableIntf == NULL ){
 ....
}
mid =  （*env)->GetMethodID（env, runnableintf, "run", "()V") ;
if (mid == NULL ){
 ....
}
(*env)->CallVoidMethod(env, thd, mid) ;
...
```

我们在 3.3.5 中了解了 FindClass 函数返回一个名字类的引用。这儿总是得到一个命名的接口的应用。

4.2.2 构造方法描述符

"JNI"是用的构造描述符字串来表示方法类型，和怎样表示类型成员域用一样方法。一个方法描述符联合一个方法的参数类型和返回类型。首先参数类型出现在，同时被一堆括号(parentheses)包围。参数类型被有序的列表，它们按照在方法声明中顺序出现。在多个参数类型之间没有分割符。如果一个方法没有参数，用一对空的括号来表示。在为参数类型的右闭括号后，立即放置方法的放回类型。

例如，"(I)V"表示带有一个"int"类型参数和返回值为"void"的一个方法。"()D"表示一个没有参数和返回"double"类型的一个方法。不要让"C"函数原型例如"int f(void)"误导你认为"(V)I"一个有效方法描述符。要使用"()I"来替代。

方法描述符可以包含类描述符.例如，方法:
native private String getLine(String) ;

有下面的描述符:
"(Ljava/lang/String;）Ljava/lang/String;"

数组类型的描述符使用"["字符开始，数组元素类型的描述符来跟着。例如，方法描述符:
public static void main(String[] args) ;
如下:
"([Ljava/lang/string;)V"

12.3.4 部分给出一个完整描述怎样来构成"JNI"方法的描述符。你能使用"javap"工具来打印出"JNI"方法描述符。例如，通过运行:
javap -s -p InstanceMethodCall

你得到下面输出:
...
private callback ()V
public static main([Ljava/lang/String;)V
private native nativeMethod ()V
...

"-s"标记通知"javap"来输出"JNI"描述符字串，而不是像它们在 Java 编程语言中出现的类型。
"-p"标记引起"javap"来包含这类的私有成员的信息在它的输出中。

4.2.3 调用静态方法(Calling Static Methods)

前面的例子示范本地代码怎样调用一个实例方法。类似地，你能从本地代码执行静态方法的回调，通过下面步骤:
.用"GetStaticMethodID"来获得方法"ID"，对应于"GetMethodID"。
.传递类，方法"ID"和参数给系列静态方法调用函数的一个:"CallStaticVoidMethod","CallStaticBooleanMethod",等等(and so on)。

在允许你调用的静态方法的函数和允许你调用实例方法的函数之间有一个关键的不同点。前者(former)调用一个类的引用(class reference)作为第二个参数，然而后者(latter)使用一个对象应用(object reference）作为第二个参数。例如，你传递类的引用到"CallStaticVoidMethod"，但是传递一个对象应用到"CallVoidMethod"。

在 Java 编程语言层，你能够使用两种可选语法，调用在类"Cls"中的一个静态方法"f":"Cls.f"或者"obj.f"，这儿"obj"是一个"Cls"的实例。(然而，后者是被推荐的编程格式)在"JNI"中，你必须总是指明类的引用，在从本地代码中调用静态方法时。

让我么看一个例子，它从本地代码中使用了一个静态方法的回调。它是前面(earlier)"InstanceMethodCall"例子的一个小变化:

```
class StaticMethodCall{
 private native void nativeMethod() ;
 private static void callback(){
  System.out.println("In Java") ;
 }
 public static void main(String args[]){
  StaticMethodCall c = new StaticMethodCall() ;
  c.nativeMethod() ;
 }
 static{
  System.loadLibrary("StaticMethodCall") ;
 }
}
```

这儿是本地方法的实现:

```
JNIEXPORT void JNICALL
Java_StaticMehtodCall_nativeMethod(JNIEnv *env, jobject obj)
{
 jclass cls = (*env)->GetObjectClass(env, obj) ;
 jmethodID mid = (*env)->GetStaticMethodID(env, cls, "callback", "()V") ;
 if( mid == NULL ){
  return ;
 }
 printf("In C\n") ;
 (*env)->CallStaticVoidMethod(env, cls, mid) ;
}
```

确信你传递"cls"(用粗体高亮),相对于"obj"，来给"CallStaticVoidMethod"。运行上面的程序产生下面期待输出:

```
In C
In Java
```

4.2.4 调用一个超类的实例方法(Calling Instance Methods of a Superclass)

你能调用实例方法，它被定义在一个超类中，但是在这个类中已经被重载了，这对象属于这类。"JNI"提供一套"CallNonvirtual<Type>Method"函数来为这个目的。为调用被定义在一个超类中的一个实例方法，你如下做:
.使用"GetMethodID"来从超类的引用来得到方法"ID"。
.传递对象(object),超类(superclass),方法"ID"和参数给系列非虚拟调用函数(nonvirtual invocation functions)的一个，例如"CallNovirtualVoidMethod", "CallNovirtualBooleanMethod"等等(and so on)

相对地很少你将需要调用一个超类的实例方法。这个工具和调用一个重载超类方法类似，说"f"吧，在 Java 编程语言总用下面结构:
super.f();

"CallNovirtualVoidMethod"也能被用来调用构造器(constructors),像下一部分将要说明的。

4.3 调用构造器(Invoking Constructors)

在"JNI"中，按照下面几步，构造器可以被调用,类似于为调用实例方法。为获得一个构造器的方法"ID",传递"<init>"作为方法名字和在方法描述符中"V"作为返回类型。然后，你能通过传递方法 ID 给"JNI"函数来调用构造器，例如 NewObject。接下来的代码实现了和"JNI"函数"NewString"一样的功能，他构建一个 java.lang.String 的对象，从来自一个"C buffer"中存储的 Unicode 字符串(Unicode characters):

```
jstring MyNewString(JNIEnv *env, jchar *Chars, jint len)
{
 jclass stringClass ;
 jmethodID cid ;
 jcharArray elemArr ;
 jstring result ;
 stringClass = (*env)->FindClass(env, "java/lang/String") ;
 if (stringClass == NULL ){
  return NULL ;
 }

 cid = (*env)->GetMethodID(env, stringClass, "<init>", "([c)V") ;
 if( cid == NULL ){
  return NULL ;
 }


 elemArr = (*env)->NewCharAray(env, len) ;
 if (elemArr == NULL ){
  return NULL ;
```

```
 }
 (*env)->SetCharArrayRegion(env, elemArr, 0, len, chars) ;

 result = (*env)->NewObject(env, stringClass, cid, elemArr) ;

 (*env)->DeleteLocalRef(env, elemArr) ;
 (*env)->DeleteLocalref(env, stringClass) ;
 return result ;
}
```

这个函数是相当复杂需要细致的解释。首先，"FindClass"返回一个"java.lang.String"类的引用。下一步，"GetMethodID"为字符串构造器(string constuctor）返回方法"ID",String(char[] chars）。然后，我们调用"NewCharArray"来分配一个字符数组来得到所有字符串元素。"JNI"函数"NewObject"函数使用被构建类型引用，构建器的方法"ID"和需要传递给构建起的参数来做参数。

"DeleteLocalRef"调用允许虚拟机来释放被"elemArr"和"stringClass"局部引用来使用的资源。5.2.1 部分将提供一个详细描述关于你什么时候和为什么应该调用"DeleteLocalRef"。

"Strings"是个对象。这个例子进一步指明这点。然而，例子也提出一个问题。给出我们能使用其他"JNI"函数实现一样的功能，为什么 JNI 提供内建函数例如 NewString?原因是内建字符串函数(built-in string functions)更高效于从自本地代码中调用"java.lang.String"API。"String"是最常被使用的对象类型(type of objects），在"JNI"中值得特别支持。

也可能使用"CallNonvirtualVoidMethod"函数来调用构造器。这种情况中，首先，本地代码通过"AllocObject"函数来创建一个没有初始化的对象。上面是单独 NewObject 调用:

```
result =  （*env)->NewObject(env, stringClass, cid , elemArr) ;
```

可以通过一个"AllocObject"调用跟着通过一个"CallNovirtualvoidMethod"调用来替代它:

```
result = (*env)->AllocObject(env, stringClass) ;
if（result）{
 (*env)->CallNovirtualVoidMethod(env, result, stringClass, cid, elemArr）  ;

 if( (*env)->ExceptionCheck(env) ){
  (*env)->DeleteLocalRef(env, result) ;
  result = NULL ;
 }
}
```

"AllocObject"创建一个没初始化的对象，同时不需小心被使用，所以一个构造器在每个对象撒谎那个最多调用一次。本地代码不应该多次地调用一个构造器在同一个对象上。

有时候，你可以发现首先分配一个没有初始化的对象和而后某时调用构造器是有用的。然而，大多数情况中你应该使用"NewObject",同时避免更容易产生错误的 AllocObject/CallNovirtualVoidMethod 对。

4.4 缓冲成员域和方法 ID(Caching Field and Method IDs)

得到成员域和方法"ID"需要基于名字和成员域或方法的描述符的符号的查找。符号的查找是相对费时的。在这部分，我们介绍一中技术，被用来减少这个开销。

这个方法是为计算成员域和方法"ID",同时缓冲它们为以后再次使用。对于缓冲成员域和方法"ID"这儿有两种方法，依赖于是否在成员域或方法 ID 的使用点时执行缓冲，或在定义成员域或方法的类的静态初始化中来执行缓冲。

4.4.1 使用时缓冲(Caching at the Point of Use)

成员域和方法 ID 可以被缓冲，在本地代码访问成员域值或执行方法回调的地方。"Java_InstanceFieldAccess_accessField"函数的下面实现，缓冲成员域的 ID 到静态变量中，所以在"InstanceFieldAccess.accessField"方法的每次调用时不需要被再次计算。

```
JNIEXPORT void JNICALL
Java_InstanceFieldAccess_accessField(JNIEnv *env,jobject obj)
{
 static jfieldID fid_s = NULL ;
 jstring jstr ;
 const char *str ;

 if (fid_s == NULL ){
  fid_s = (*env)->GetFieldID(env, cls, "s", "Ljava/lang/String;") ;
  if( fid_s == NULL) {
   return ;
  }
 }
 printf("In C:\n") ;
 jstr = (*env)->GetObjectField(env, obj, fid_s) ;
 str = (*env)->GetStringUTFChar(env, jstr, NULL) ;
 if ( str == NULL ){
  return ;
 }
 printf("  c.s = "%s"\n", str) ;
 (*env)->ReleaseStringUTFChars(env, jstr, str) ;
```

```
 jstr = (*env)->NewStringUTF(env, "123") ;
 if( jstr == NULL){
  return ;
 }
 (*env)->SetObjectField(env, obj, fid_s, jstr) ;
}
```

高亮的静态变量"fid_s"存储前一次计算的成员域 ID 为"InstanceFieldAccess.s"。静态变量被初始化为"NULL"。当"InstanceFieldAccess.accessField"方法第一次调用时，它计算成员域的 ID 同时为以后的使缓冲在静态变量中。

你可能注意到在上面代码中一个明显的判断条件。多个线程可以在相同的时间调用"InstanceFieldAccess.accessField"方法，同时并发地计算一样的成员域 ID.一个线程可以通过另一个线程重写静态变量"fid_s".幸运地，虽然这个判断条件在躲线程中导致重复操作，但没有坏处。这个被多线程计算为了在同一个类中的同样的成员域的成员域 ID 将一定是同样的。

按照同样的注意，我们也可以为在较早"MyNewString"例子中的"java.lang.String"的构造器，缓冲方法"ID"：

```
jstring
MyNewString(JNIEnv *env, jchar *char, jint len)
{
 jclass stringClass ;
 static jmethodID cid = NULL ;
 jcharArray elemArr ;
 jstring result ;
 stringClass = (*env)->FindClass(env, "java/lang/String") ;
 if (stringClass == NULL ){
  return NULL ;
 }

 if ( cid == NULL ){

  cid = (*env)->GetMethodID(env, stringClass, "<init>", "([c)V") ;
  if( cid == NULL ){
   return NULL ;
  }
 }


 elemArr = (*env)->NewCharAray(env, len) ;
 if (elemArr == NULL ){
```

```
 return NULL ;
 }
 (*env)->SetCharArrayRegion(env, elemArr, 0, len, chars) ;

 result = (*env)->NewObject(env, stringClass, cid, elemArr) ;

 (*env)->DeleteLocalRef(env, elemArr) ;
 (*env)->DeleteLocalref(env, stringClass) ;
 return result ;
}
```

我们为"java.lang.String"的构造器来计算方法 ID，当"MyNewString"第一次被调用时。高亮的静态变量"cid"缓冲了结果。

4.4.2 在定义类的初始化中缓冲(Caching in the Defining Class's Initializer)

当我们在使用的地方缓冲一个成员域或一个方法"ID"，我们必须引入一个检查来侦测是不是"ID"已经有缓冲了。这个方法不但在加速路上(on the fast path)当"ID"已经缓冲的时候，导致一个小的性能缺陷，而且也可能导致多次缓冲和检查的。例如，入股多个本地方法都请求访问同一成员域，然后它们都需要一个检查来计算和缓冲相应的成员域"ID"。

在许多情况中，在应用程序有机会调用本地方法前，通过一个本地方法的请求来更方便的初始化成员域和方法 ID。在调用在这个类的任何方法前，虚拟机总是执行这个类的静态初始化。因此对于计算和缓冲成员域或方法 ID 的一个适合的地方是在定义了成员域或方法的类的静态初始化中。

例如，为缓冲"InstanceMethodCall.callback"的方法 ID，我们引入一个全新本地方法"initIDs", 在"InstanceMethodCall"类的静态初始化中调用:

```
class InstanceMethodCall{
 private static native void initIDs() ;
 private native void nativeMethod() ;
 private void callback(){
  System.out.println("In Java") ;
 }
 public static void main(String args[]){
  InstancMethodCall c = new InstanceMethodCall() ;
  c.nativeMethod() ;
 }
 static{
```

```
 System.loadLibrary("InstanceMethodCall") ;
 initIDs() ;
 }
}
```

对比在 4.2 部分中的原始代码，上面的程序包含两个额外的行(用粗体字的高亮)。"initIDs"的实现仅仅地计算和缓冲了"InstanceMethodCall.callback"的方法 ID：

```
jmethodID MID_InstanceMethodCall_callback ;

JNIEXPORT void JNICALL
Java_InstanceMethodCall_initIDs(JNIEnv *env, jclass cls)
{
 MID_InstanceMethodCall_callback =
  (*env)->GetMethodID(env, cls, "callback", "()V") ;
}
```

这虚拟机运行静态初始化，同时在执行在"InstanceMethodCall"类中的任何其他方法(例如"nativeMethod"或"main"前，轮到调用"initIDs"方法。在全局变量中缓冲了方法"ID"时，"InstanceMethodCall.nativeMethod"的本地方法的实现不再需要执行一个符号查找了:

```
JNIEXPORT void JNICALL
Java_InstanceMethodCall_nativeMethod(JNIEnv *env, jobject obj)
{
 printf("In C\n") ;
 (*env)->CallVoidMethod(env, obj, MID_InstanceMethodCall_callback) ;
}
```

4.4.3 在两种缓冲 ID 的方法间的比较(Comparison between the Two Approaches to Caching IDs)

如果"JNI"编程者不可以控制定义成员域或方法的类的源代码，在使用时缓冲"IDs"是个有效合理的解决方法。例如，在"MyNewString"例子中，我们不能注入一个客户的"initIDs"本地方法到"java.lang.String"类中，为了预计算和缓冲"java.lang.String"的构造器的方法"ID"。

当和在定义类的静态初始化中缓冲比较时，在使用时缓冲有许多缺点。

.像前面解释，在使用时的缓冲在执行快速路径中时需要一个检查，同时也可能需要多个相同成员域或方法 ID 的初始化和检查。

.方法和成员域"IDs"是有效的直到类被载出。如果你在使用时缓冲了成员域和方法"IDs"，在你本地代码依赖缓冲 ID 的值时，你必须确保定义的类不被载出和重载入。(下一章将显示你怎样通过创建那个使用"JNI"的类的一个引用，能保持一个不被载出。)另一方面，如果定义的类的静态初始化中做了缓冲，缓冲 IDs 将自动地被计算，当类被载出和后面又载入时。

因此，如果可行的话，最好是在它们定义类的静态初始化中来缓冲成员域和方法"IDs"。

4.5 JNI 成员域和方法操作的性能(Performance of JNI Field and Method Operations)

学习怎样缓冲成员域和方啊 ID 来增强性能后，你可能惊奇:使用"JNI"访问成员域和调用方法的性能特点是什么啊？从本地代码(一个本地/Java 回调)执行一个回调的花费和调用一个本地(一个 Java/本地调用)的花费比较，还有和调用一般方法(一个 Java/Java 的调用)的花费比较怎养?

让我们通过比较 Java/native 调用和 Java/Java 调用的花费开始。Java/native 调用时潜在地比 Java/Java 调用慢，因为下面的原因:
.在虚拟机器实现中，本地方法更可能比 Java/Java 调用使用一个不同的调用协定。做为一个结果，虚拟机器必须执行额外的操作来建立参数和构建栈(build argument and set up the stack frame)，在进入一个本地方法入口点前。
.它对于对于虚拟机器一般是内联的方法调用。内联 Java/native 调用比内联 Java/Java 调用困难许多。

我们估计，一个典型的虚拟机可能执行一个 Java/native 调用大约慢两到三倍于执行一个 Java/Java 调用。因为一个 Java/Java 调用只需要几个周期，增加的开销可以忽略不计，除非本地方法执行简单操作。也可能建立的虚拟机的实现，Java/native 调用的性能接近或等于 Java/Java 调用的性能。(例如，这样的虚拟机的实现可能采用了"JNI"调用协议像内部 Java/Java 调用协议。)

一个 native/Java 回调的性能特征是技术上类似于一个 Java/native 调用。因此，native/Java 回调的开销也可能是大约两到三被于 Java/Java 调用。然而，事实上，native/Java 回调相对少见。虚拟机的实现通常不优化回调的性能。在写的许多产品的时候，虚拟机的实现就是如此，一个 native/Java 回调的开销比一个 Java/Java 调用要高十倍一样多。

用"JNI"来访问成员域的开销依赖于通过"JNIEnv."调用的开销，而不是直接地取对象(object)的值,本地方法必须执行一个 C 函数调用来取对象的值。这个函数调用时必须的，因为它隔离本地代码于被虚拟机实现来维护的内部对象表示。"JNI"成员域访问的开销是典型的可以忽略不计的，因为一个函数调用只有几个周期。

第五章 局部和全局引用

"JNI"公开了实例和数组类型(such as jobject, jclass, jstring, and jarray)作为不透明的应用。本地代码不能直接地查看一个不透明引用的指针的内容。作为替代，使用"JNI"函数来访问被一个不透明引用指向的数据结构。通过处理不透明的引用，你不必担心内部对象(object)布置，这布置是依赖于一个特定的 Java 虚拟机的实现。然而，你做的是需要了解在"JNI"中不同类别的引用:

."JNI"支持三种类型不透明引用:局部引用，全局引用和弱全局引用。
.局部和全局引用有不同的生命周期(lifetimes)。局部引用会自动释放，然而全局和弱全局引用保持有效一直到它们被程序员释放。
.一个局部或全局的引用保持了引用的对象(referenced object)不被垃圾收集掉。另一方面，一个弱全局引用允许引用的对象被垃圾收集。
.不是所有的引用能被用在所有地方的(in all contexts).例如，在创建引用返回值的本地方法后，使用一个局部引用是不合法的。

在这章中，我们将详细的讨论这些问题。恰当地管理"JNI"引用对于写可靠和节省空间的代码是至关重要的。

5.1 局部和全局引用(Local and Global References)

什么是局部和全局引用? 它们有什么不同?我们将用一系列的例子来说明局部和全局引用。

5.1.1 局部引用(Local Reference)

大多数"JNI"函数都创建局部引用。例如，"JNI"函数"NewObject"创建一个新的实例，同时返回一个局部引用到那个实例。

一个局部引用，只在创建它的本地方法的动态上下文中，同时只在本地方法的一个调用中，是有效的。所有的在一个本地方法执行期间创建的局部引用将被释放，一旦本地方法返回。

你不必写本地方法，在一个静态变量中存储的一个局部引用的，和期望在后来的调用中使用一样的引用。例如，下面的程序，是在 4.4.1 部分中的"MyNewString"函数的修改版本，不正确的使用了局部引用。

```
jstring MyNewString(JNIEnv *env, jchar *chars, jint len)
{
 static  jclass stringClass = NULL ;
 jmethodID cid ;
 jcharArray elemArr ;
 jstring result;
 if( stringClass == NULL ){
  stringClass = (*env)->FindClass(env, "java/lagn/String") ;
  if ( stringClass == NULL ){
   return NULL ;
  }
 }

 cid = (*env)->GetMethodID(env, stringClass, "<init>", "([C)V") ;
```

```
 ...
 elemArr = (*env)->NewCharArray(env, len) ;
 ...
 result = (*env)->NewObject(env, stringClass, cid, elemArr) ;
 (*env)->DeleteLocalRef(env, elemArr) ;
 return result ;
}
```

这儿我们已经省略和我们讨论不直接相关的代码行。在一个静态变量中缓冲"stringClass"的目的可能是为了消除再次调用下面函数的开销:

```
FindClass(env, "java/lang/String") ;
```

这是一个不正确的方法，因为"FindClass"返回的是"java.lang.String"类对象的一个局部引用。为了了解这是一个什么问题，假设"C.f"的本地方法的实现调用了"MyNewString"：

```
JNIEXPORT jstring JNICALL
Java_C_f(JNIEnv *env, jobject this)
{
 char *c_str = ... ;
 ....
 return MyNewString(c_str) ;
}
```

在本地方法"C.f"返回后，虚拟机释放所有在"Java_C_f"执行期间创建的局部引用。这些被释放的局部引用包含存储在"stringClass"变量中的类对象的局部引用。然后将来，"MyNewString"调用将尝试使用一个无效局部应用，可能导致内存的破坏或系统奔溃。例如，一个如下(such as the following)代码片段，做两个"C.f"的调用，引起"MyNewString"使用了无效的局部引用:

```
...
...= C.f() // The first call is perhaps OK
...= C.f() // This would use an invalid local reference.
...
```

有两种方法能使一个局部引用无效。像前面解释的，虚拟器在本地方法返回后，自动地释放在本地方法执行期间创建的所有的局部引用。另外(In addition),程序员可以明确地管理局部引用的生命周期(lifetime),使用例如"DeleteLocalRef"的"JNI"函数。

如果虚拟器在本地方法返回后，能自动地释放局部引用，为什么你还需要明确地删除局部引用?一个局部引用保持一个引用对象阻止垃圾收集，直到局部引用是无效的。例如，在"MyNewString"中的"DeleteLoacalRef"调用，允许中间的"array"对象 elemArr 能立即被垃圾收集。否则虚拟机将只能在调用"MyNewString"的本地方法(例如上面 C.f)返回后，释放"elemArr"对象。

一个局部引用在它销毁前，可以通过多个本地函数本传递。例如，"MyNewString"返回的一个通过"NewObject"创建的"string"引用。然后它将上传到"MyNewString"的调用者来决定是否释放"MyNewString"返回的局部引用。在"Java_C_f"例子中，C.f 又返回"MyNewString"的结果作为本地方法的返回值。虚拟机从"Java_C_f"函数得到局部引用后，它传递底层的"string"对象到"C.f"的调用者，然后销毁了通过"JNI"函数"NewObject"原始创建的局部引用。

局部应用也只在创建它们的线程中有效。一个在一个线程中创建的局部引用不能在另一线程中被使用。对于一个本地方法，存储一个局部引用在一个全局变量中同时期望另一线程使用这个局部引用，是一个编程错误。

5.1.2 全局引用(Global References)

你能在一个本地方法的多次调用中使用一个全局引用。一个全局引用能在多个线程中(across muliple threads)被使用,同时保持有效直到编程者释放它。像一个局部引用，一个全局引用确保了引用对象将不被垃圾收集。

和被大部分"JNI"函数创建的局部引用不一样，全局引用只被一个"JNI"函数"NewGlobalRef"创建。"MyNewString"的下面的版本说明怎样使用一个全局引用。我们高亮在下面代码和在上一部分中错误缓冲一个局部引用的代码之间的不同:

```
jstring MyNewString(JNIEnv *env, jchar *chars, jint len)
{
 static jclass stringClass = NULL ;
 if ( stringClass == NULL ){
  jclass LocalRefCls = (*env)->FindClass(env, "java/lang/String") ;
  if (localRefCls == NULL ){
   return NULL ;
  }

  stringClass = (*env)->NewGlobalRef(env, localRefCls) ;


  (*env)->DeleteLocalRef(env, localRefClas) ;

  if( stringclass == NULL ){
   return NULL ;
  }
 }
 ...
}
```

这个修改版本传递来自"FindClass"的局部引用到"NewGlobalRef",来创建一个"Java.lang.String"类对象的的全局引用。我们在删除"localRefCls"后，检查"NewGlobalRef"是否成功地创建"string 类(stringClass)"，因为无论那种情况局部引用"localRefCls"需要被删除。

5.1.3 弱全局引用(Weak Global Reference)

弱全局引用是在"Java 2 SDK release 1.2"中新出现的。它们使用"NewGlobalWeakRef"来创建和使用"DeleteGlobalWeakRef"来释放。像全局引用一样，弱全局引用在本地方法调用中和在不同的线程中，保持有效。但和全局引用不一样，弱全局引用不能保持底层对象不被垃圾收集。

"MyNewString"例子显示怎样缓冲一个"java.lang.String"类的全局引用。"MyNewString"例子也可以选择使用一个弱全局引用来存储缓冲的"java.lang.String"类。无论我们使用的是一个全局引用还是一个弱全局引用，因为"java.lang.String"是一个系统类，所以将不会被垃圾收集。

当本地方法代码缓冲了一个引用不必保持底层的对象不被垃圾收集，弱全局引用变的十分有用。例如，假设一个本地方法"mypkg.MyCls.f"需要缓冲一个"mypkg.MyCls2"类的引用。在一个弱全局引用中，缓冲了这个类仍然允许"mypkg.MyCls2"类被载出:

```
JNIEXPORT void JNICALL
Java_mypkg_MyCls_f(JNIEnv *env, jobject self)
{
 static jclass myCls2 = NULL;
 if( myCls2 == NULL ){
  jclass myCls2Local = (*env)->FindClass(env, "mypkg/MyCls2") ;
  if( myCls2Local == NULL ){
   return ;
  }
  myCls2 = NewWeakGlobalRef(env, myCls2Local) ;
  if( myCls2 == NULL ){
   reurn ;
  }
 }
 ...
}
```

我假设"MyCls"和"MyCls2"有一样的生命周期。(例如，它们可以被同样的类载入器载入。)因此我们不用思考这种情况，当"MyCls2"被载出而后再载入时，然而 MyCls 和他的本地方法实现"Java_mypkg_MyCls"一直保持使用。如果这发生，我们必须检查缓冲的弱全局引用是否任然指向一个活着的类对象，或指向一个已经被垃圾回收的类对象。下一部分将解释怎样在弱全局引用上执行如此检查。

5.1.4 比较引用(Comparing Reference)

给出两个局部，全局，弱全局引用，你使用"IsSameObject"函数，能检查它们是否参考一样使对象。例如:

(*env)->IsSameObject(env, obj1, obj2)

如果 obj1 和 obj2 参考了一样对象，返回"JNI_TRUE"(或 1)；否则返回"JNI_FALSE"(或 0)。

在 Java 虚拟机中，一个在"JNI"中的"NULL"引用参考"null"对象。如果"obj"是一个局部或一个全局的参考，你可以使用

(*env)->IsSameObject(env, obj, NULL)

或

obj == NULL

来决定是否"obj"参考"null"对象。

对于弱全局引用的规则则有所不同。"NULL"弱引用参考"null"对象。然而，"IsSameObject"为弱全局引用有指定用途。你能使用"IsSameObject"来确定一个"non-NULL"弱全局引用是否指向一个活着的对象(object)。假设"wobj"是一个"non-NULL"弱全局引用。下面调用:

(*env)->IsSameObject(env, wobj, NULL)

如果"wobj"指向已经被收集的对象，返回"JNI_TRUE"，然而如果"wobj"指向一个活着的对象，返回"JNI_FALSE"。

5.2 释放引用(Freeing References)

每个"JNI"引用都自己消耗一定量的内存，除了被引用对象使用的内存外。做为一个"JNI"程序员，你应该知道你的程序在某个时间使用的引用的数目。特别地，你应该知道你的程序在她的执行期间的任何时候创建的局部引用的个数上限，即使这些局部引用最终将被虚拟机自动地释放。然而短暂地，过分的引用创建，可能导致内存的耗尽。

5.2.1 释放局部引用(Freeing Local References)

在大多数情况，当执行一个本地方法时，你不必担心释放局部引用。当本地方法放回到调用者时，Java 虚拟器为你释放它们。然而，有些时候，你，"JNI"编程者，应该明确地释放局部引用为避免过分的内存使用。考虑一下下面的情况:

.你需要创建大量的局部引用在一个单独的本地方法调用中。这可能导致"JNI"内部的局部引用表的一个溢出。好的方法是适当地删除些将不需要的局部引用。例如，在下面程序片段中，本地代码遍历(iterate through)了一个潜在的大量的"string"数组。每次迭代后，本地代码应该明确释放对字符元素的局部引用，想下面:

for (i = 0 ; i < len ; i++){

```
 jstring jstr =  （*env)->GetObjectArrayElement(env, arr, i) ;
 ...
 (*env)->DeleteLocalRef(env, jstr) ;
}
```

.你需要写个工具函数被从不清楚的上下文中调用。在 4.3 部分中显示"MyNewString"例子说明"DeleteMyNewSring"使用为了删除恰当局部引用在一个工具函数中。否则，在"MyNewString"函数的每次调用后，两个局部引用将保持被分配。

.你的本地方法不再返回。例如，一个本地方法也可进入一个无终止的事件派遣循环。释放在循环中创建的局部引用至关重要(crucial),所以他们不会无限期的积累，导致内存泄漏。

.你本地方法访问一个很大的对象，因此(thereby)创建了一个这个对象局部引用。然而本地方法在返回调用者前，执行额外的计算。即使对象在本地方法剩余处不再本使用，这个很大对象的局部引用将阻止垃圾搜集这对象，直到本地方法返回。例如，在下面的程序片段中，因为事前(beforehand),这有个清楚的"DeleteLocalRef"的调用,垃圾收集器可以释放被"lref"指向的对象，在函数内部执行一个很长的计算时:

```
JNIEXPORT void JNICALL
Java_pkg_Cls_func(JNIEnv *env, jobject this )
{
 lref = ...
 ...
 (*env)->DeleteLocalRef(env, lref) ;
 lengthyComputation() ;
 return ;
}
```

5.2.2 管理局部引用在"Java 2 SDK Release 1.2"（Managing Local References in Java 2 SDK Release 1.2)

"Java 2 SDK Release 1.2"提供另外一套函数来管理局部引用的生命周期。这些函数是"EnsureLocalCapacity","NewLocalRef","PushLocalFrame",和"PopLocalFrame"。

"JNI"说明指示虚拟器自动地确保每个本地方法能创建至少 16 个局部引用。经验显示这提供了足够容量用于在 Java 虚拟器中对象，为大部分主要不包含复杂指令的本地方法。然而，如果这儿需要创建额外的局部引用，一个本地方法可以使用一个"EnsureLocalCapacity"调用来确保满足局部引用数目的空间是用的(available)。例如，前面例子的一个小的变化，预留足够的容量为在循环处理期间创建的所有局部引用，如果有充足内存可用:

```
if ((*env)->EnsureLocalCapacity(env, len))<0 ){
 ...
}
for ( i = 0 ; i < len ; i++ ){
 jstring jstr = (*env)->GetObjectArrayElement(env, arr, i) ;
```

```
...

}
```

当然，上面的版本和适当删除局部引用的前面的版本消耗大量内存很像。

可选则地，"Push/PopLocalFrame"函数允许程序员来创建嵌入域的局部引用。例如，我们也可以重写一样的例子，如下:

```
#define N_REFS ...
for ( i = 0 ; i < len ; i++ ) {
 if((*env)->PushLocalFrame(env, N_REFS) <0){
  ...
 }
 jstr =  （*env)->GetObjectArrayElement(env, arr，i) ;
 ...
 (*env)->PopLocalFrame(env, NULL) ;
}
```

"PushLocalFrame"为指定数目的局部引用创建一个新的域。"PopLocalFrame"销毁最顶层的域，释放在这个域的所有局部引用。

使用"Push/PopLocalFrame"函数的好处是他们可以管理局部引用的生命周期，而不必担心在执行期间被创建的每个单独的局部引用。在上面例子中，如果处理"jstr"的计算创建额外的局部引用，这些局部引用将在"PopLocalFrame"返回时被释放。

在你写希望返回一个局部引用的有效函数时，"NewLocalRef"函数是有用的。我们将在 5.3 部分中示范"NewLocalRef"函数的使用。

本地代码可以创建局部引用超过默认的 16 的容量或在"PushLocalFrame"或"EnsureLocalCapacity"调用中保留的容量。虚拟器实现将尝试分配局部引用需要的内存。然而，不保证(no guaratee）内存将可用。若果分配内存失败，虚拟器退出。你应该为局部引用来保留足够的内存，同时恰当地释放局部引用来避免这种不期望虚拟器退出。

"Java 2 SDK release 1.2"支持一个行命令选项"-verbose:jni"。当这个选项使能时，虚拟器实现报告过多的局部引用创建，超过保留的容量。

5.2.3 释放全局引用(Freeing Global References)

当你的本地代码不再需要访问全局引用时，你应该调用"DeleteGlobalRef"。如果你调用这个

函数失败，"Java"虚拟器将不能垃圾收集这对应的对象，即使在系统的任何地方当对象不再使用的时候。

当你的本地代码不再需要访问一个弱全局引用，你应该调用"DeleteWeakGlobalRef"。如果你调用这个函数失败，"Java"虚拟器任然将能垃圾收回底层对象，但将不能收回(reclaim)被弱全局应用自己消耗的内存。

5.3 管理引用的规则(Rules for Managing References)

基于在前面部分我们已经涵盖的，我们现在准备通过规则去管理在本地代码中的"JNI"引用。目标(objective)是消除(eliminate)不必内存使用和对象保持(object retention）。

通常(in general),这有两种类型的本地代码:直接实现本地代码的函数和在任意背景(arbitrary contexts)中使用的有效函数(utility functions)。

当写直接地实现本地方法的函数时，你需要小心过多局部引用在循环中被创建和不需要局部引用被在本地方法没有返回时创建。对于虚拟器，能接受(acceptable）预留达 16 个局部引用使用，在本地方法返回后被删除。本地方法调用必须不引起全局或弱全局引用的累积，因为全局和弱全局引用在本地方法返回后，不自动地被释放。

当写本地有效函数，你必须在任何执行过程上通过函数，小心不泄漏任何局部引用。因为一个有效的函数可以在一个不可预计情况（unanticipated context)中被重复调用，任何不必的应用创建可以引起内存的溢出。

.当调用一个返回一个基本类型的有效函数时，它必定对累积额外的局部，全局或弱全局引用没有副作用(hava the side effect of)。
.当调用一个放回一个应用类型的有效函数时，它必须不累积额外的局部，全局或弱全局引用，除作为结果返回的引用。

对于一个有效的函数，创建一些全局或弱全局引用为缓冲的目的是能接受的，因为只在第一次调用创建这些引用。

如果一个有效的函数返回一个引用，你应该说明这个函数的返回引用部分的类型。在有些时候不应该返回一个局部引用，和不应该在任何时候返回一个全局引用。调用者需要知道被有效函数返回的引用类型，为了正确地管理它自己 JNI 引用。例如，下面的代码重复地调用一个有效函数"GetInfoString"。我们必须知道被"GetInfoString"返回的引用的类型，为在每个迭代后能够正确地释放返回的"JNI"引用。

```
while(JNI_TRUE){
 jstring infoString = GetInfoString(info) ;
 ...
```

```
 ???
}
```

在"Java 2 SDK release 1.2"中，"NewLocalRef"函数某些时候，对于确保一个有效函数总是返回一个局部引用很有用。为了说明，让我对"MyNewString"函数做另外的改变(有些人为(somewhat contrived))。下面的版本缓冲一个濒繁被请求的字串(叫"CommonString"）在全局引用中:

```
jstring
MyNewString(JNIEnv *env, jchar *chars, jint len)
{
 static jstring result ;

 if( wstrncmp("CommonString", chars, len) == ){

  static jstring cachedString = NULL;
  if ( cachedString == NULL ){

   jstring cachedStringLocal = ... ;

   cachedString =
    (*env)->NewGlobalRef(env, cachedStringLocal) ;
  }
  return (*env)->NewLocalRef(env, cachedString) ;
 }
 ...
 return result ;
}
```

一般路径返回一个作为一个局部引用的字符串。想前面解释的，我们必须保存缓冲字符串到一个全局引用中，为了它在多个本地方法调用中和来自多个线程调用能被访问。高亮行创建了一个新的局部引用，来指向和缓冲全局引用一样的对象。对于它的调用者做为协议的一部分，"MyNewString"总是返回一个局部引用。

"Push/PopLocalFrame"函数对于管理局部引用的生命周期特别便利(especially convenient)。如果你调用"PushLocalFrame"在一个本地函数的入口上，在本地函数返回前，调用"PopLocalFrame"确保在本地方法执行中创建的所有局部引用将被释放。
"Push/PopLocalFrame"函数是高效的(effcient)。强烈鼓励你使用他们。

如果你在你的函数入口上调用"PushLocalFrame"，记住在所有的函数退出路径上调用"PopLocalFrame"。例如，下面函数有一个"PushLocalFrame"调用，但需要多个"PopLocalFrame"调用:

```
jobject f(JNIEnv *env, ...)
{
 jobject result;
 if( (*env)->PushLocalFrame(env, 10) < 0 ){

  return NULL ;
 }
 ...
 result = ... ;
 if (...){

  result = (*env)->PopLocalFrame(env, result) ;
  return result ;
 }
 ...
 result = (*env)->PopLocalFrame(env, result) ;

 return result ;
}
```

没有正确的调用"PopLocalFrame"可能导致一个未定义行为，例如虚拟器的奔溃。

上面例子也说明指定的"PopLocalFrame"的第二个参数为什么有时有用。"result"局部引用在通过"PushLocalFrame"来构建新的"Frame"中被初始创建。"PopLocalFrame"转换他的第二个参数"result"为一个在前一个"frame"中的新的局部引用，在弹出最顶层的"frame"前。

第六章 异常（CHAPTER 6 Exceptions)

在调用 JNI 函数后，在本地代码为可能出现的错误做检查中，我们遇到的许多情况。这章探讨本地代码怎样侦测和修复这些错误情况。

我们将关注作为"JNI"函数调用的结果的发生的错误，不是在本地代码中发生的任何错误(arbitrary errors)。如果一个本地调用操作系统功能，这只能简单使用记录文本的方法来在系统调用中可能的失败。另一方面，如果本地方法调用一个"Java API"方法的回调函数,使用在这章中详细描述的几步来恰当地检查和修复在方法执行中发生的可能的一场。

6.1 概要(Overview)

通过一系列例子，我们介绍"JNI"异常处理函数。

6.1.1 在本地方法中缓冲和抛出异常

下面的程序显示怎样声明一个抛出一个异常的本地方法。"CatchThrow"类声明一个"doit"本地方法，同时指定他抛出一个"IllegalArgumentException":

```
class CathcThrow{
 private native void doit()
  thows IllegalArgumentException;
 private void callback() throw NulllPointerException{
  throw new NullPointerException（"CathThrow.callback") ;
 }
 public static void main(String args[]){
  CatchThrow c = new CatchThrow() ;
  try{
   c.doit() ;
  }catch(Exception e){
   System.out.println("In Java: \n\t"+e) ;
  }
 }
 static{
  System.loadLibrary("CatchThrow") ;
 }
}
```

"CatchThrow.main"方法调用本地函数"doit",实现如下:

```
JNIEXPORT void JNICALL
Java_CatchThrow_doit(JNIEnv *env, jobject obj）
{
 jthrowable exc ;
 jclass cls =  （*env)->GetObjectClass(env, obj) ;
 jmethodID mid =
  （*env)->GetMethodID(env, cls, "callback", "()V") ;
 if ( mid == NULL ){
  return ;
 }
 (*env)->CallVoidMethod(env, obj, mid) ;
 exc = (*env)->ExceptionOccurred(env) ;
 if (exc){

  jclass newExcCls ;
  (*env)->ExceptionDescribe(env) ;
  (*env)->ExceptionClear(env) ;
  newExcCls = (*env)->FindClass(env,
    "java/lang/IllegalArgumentException");
```

```
 if( newExcCls == NULL ){

  return ;
 }
 (*env)->ThrowNew(env, newExcCls, "thrown from C code") ;
}
}
```

运行带本地库程序，如下输出:

```
java.lang.NullPointerException:
  at CatchThrow.callback(CatchThrow.java)
  at CatchThrow.dot(Native Method)
  at CatchThrow.main(CatchThrow.java)
In Java:
 java.lang.IllegalArgumentException: thrown from C code
```

这个回调方法抛出一个"NullPointerExcetption"。当"CallVoidMethod"返回控制给本地方法时，本地代码将通过"JNI"函数"ExceptionOccurred"侦测到这个异常.在我们例子中，当一个异常被侦测到，本地代码输出一个通过调用"ExceptionDescribe"的异常的详细信息，使用"ExceptionClear"来清除异常，同时抛出一个"IllegalArgumentException"来替代。

通过"JNI"(例如，通过调用"ThrowNew")产生的一个未觉的异常，不会立即中断本地方法的执行。这和怎样处理在"Java"编程语言中的异常行为(exception behave)不一样。当一个异常在"Java"中被抛出时，虚拟器自动地改变控制流程到最近套装的"try/catch"声明来匹配异常类型。然后虚拟器清除未决异常和执行异常处理。对照相反(In contrast),"JNI"编程者必须清晰地实现控制流程，在一个异常已经发生后。

6.1.2 一个工具函数(A Utility Function)

抛出一个包含首次发现的异常类的异常，然后调用了"ThrowNew"函数。为了简单化这个任务，我们能写一个工具函数来抛出这个命名的异常:

```
void JNU_ThrowByName(JNIEnv *env), const char *name, const char *msg)
{
 jclass cls =  （*env)->FindClass(env, name) ;

 if (cls != NULL){
  (*env)->ThrowNew(env, cls, msg) ;
 }

 (*env)->DeleteLocalRef(env, cls) ;
}
```

在这本书中，"JNU"前缀代表"JNI Utilities"（"JNI"工具)。"JNU_ThrowByName"首席使用"FindClass"函数发现异常类型。如果"FindClass"失败(返回 NULL),虚拟器必须抛出一个异常(例如"NoClassDefFoundError"）。在这情况中，"JNU_ThrowByName"没有尝试抛出另一个异常。如果"FindClass"成功，我们抛出调用"ThrowNew"产生的命名异常。当"JNU_ThrowByName"放回，保证有个未决的异常，虽然这个未决异常不一定是名字参数指定的异常。我们保证了删除在这个函数创建的异常类型的局部引用。传地 NULL 给"DeleteLocalRef"是一个空操作，如果"FindClass"失败返回一个 NULL，这是一个正确的行为。

6.2 恰当地异常处理

"JNI"编程者必须必须预知(foresee)可能的异常情况，同时写代码来检查和处理这些情况。适当地异常处理有时冗长乏味(tedious)，但为了产生强壮的应用程序，它是必须的。

6.2.1 检查异常（Checking for Exceptions）

有两种方法检查是否有错误发生。
1.大多"JNI"函数使用一个清晰的(distinct)返回值(例如 NULL)来指示一个错误发生。错误返回值也暗示(implies）在当前线程有个未解决的异常。(在返回值中编码错误条件是在 C 语言中常见的做法)
下面的例子说明使用 NULL 值，做为在错误检查中"GetFieldID"的返回值。这例子包含两个部分:一个类窗口(class Window）定义大量实例成员域(handle,length and width）和一个本地方法缓冲这些域的域 ID。即使这些域存在于"Window"类中，我们任然需要检查可能从"GetFileID"返回的错误，因为虚拟器可能不能分配需要描述一个域 ID(represent a field ID）的内存。

```
public class Window{
 long handle ;
 int length ;
 int width ;
 static native void initIDs() ;
 static {
  initIDs() ;
 }
}

jfieldID FID_Window_handle ;
jfieldID FID_Window_length ;
jfieldID FID_Window_width ;
JNIEXPORT void JNICALL
Java_Window_initIDs(JNIEnv *env, jclass classWindow)
{
 FID_Window_handle =
```

```
  （*env)->GetFieldID(env, classWindow, "handle","J") ;
 if( FID_Window_handle == NULL ){
  return ;
 }
 FID_Window_length =
  （*env)->GetFieldID(env, classWindow, "length","I") ;
 if( FID_Window_handle == NULL ){
  return ;
 }
 FID_Window_width =
  （*env)->GetFieldID(env, classWindow, "width","I") ;

}
```

2.当使用一个"JNI"函数的返回值不能标记一个错误发生时，本地代码必须依赖产生异常来做错误检查。在当前线程中执行一个未决异常检测的"JNI"函数是"ExceptionOccurred"。("ExceptionCheck"被添加到"Java 2 SDK release 1.2"中。)例如，"JNI"函数"CallIntMethod"不能编码错误情况在返回值中。错误情况返回值的典型选择(Typical choices），例如"NULL"和"-1",不能工作， 因为他们可能是被调方法返回的合法值。思考一个"Fraction"类，这个类的"floor"f 方法返回"fraction"的值的整数部分，同时一些本地代码调用了这个方法。

```
public class Fraction{
 // details such as constructors omitted
 int over , under ;
 public int  floor{
  return Math.floor((double)over/under) ;
 }
}


void f(JNIEnv *env, jobject fraction)
{
 jint floor = (*env)->CallIntMethoed(env, fraction, MID_Fraction_floor) ;


 if( (*env)->ExceptionCheck(env)){
  return ;
 }
 ...
}
```

当"JNI"函数返回一个清晰的错误代码时，本地代码任然可以明显地通过调用,例如"ExceptionCheck",来检查异常。然而作为替代，它是更有效的检查明确的错误返回值。如果一个"JNI"函数返回它的错误值，在当前线程中的一个后续的"ExcptionCheck"调用保证返回"JNI_TRUE"。

6.2.2 处理异常(Handign Exceptions)

通过两种方式，本地代码可以处理一个未决的异常:
.本地方法实现能选择立即返回，引起异常在调用者中处理它。
.本地代码通过调用"ExceptionClear"能清理异常，然后执行它自己的异常处理代码。

在调用任何后续的 JNI 函数前，检查，处理和清楚一个未决的异常是非常重要(extremely important)。调用带有一个未决异常的大多数"JNI"函数(带有一个你没有清楚地清理异常)可能导致不期望的结果。当在当前线程中有一个未决的异常时，你能安全调用的 JNI 函数很少。11.8.2 部分详细说明(specify)这些"JNI"函数的完整的清单。一般说，当这儿有一个未决的异常，你能调用被设计来处理异常的"JNI"函数，同时调用"JNI"函数来释放各种通过"JNI"暴露的虚拟器资源。

当异常发生的时候，释放资源经常是必须的。在下面的例子中，本地方法首先通过"GetStringChars"调用来获得字符串的内容。如果一个后续(subsequent)操作失败，需调用"ReleaseStringChars"：

```
JNIEXPORT void JNICALL
Java_pkg_Cls_f(JNIEnv *env, jclass cls, jstring jstr)
{
 const jchar *cstr = (*env)->GetStringChars(env, jstr) ;
 if ( c_str == NULL ){
  return  ;
 }
 ....
 if( ...){
  (*env)->ReleaseStringChars(env, jstr, Cstr) ;
  return ;
 }
 ...

 (*env)->ReleaseStringChars(env, jstr, cstr) ;
}
```

当有个未决的异常，"ReleaseStringChars"的第一次被调用。本地方法的实现释放字符资源和事后立即返回没有首先清理异常。

6.2.3 在工具函数中的异常

写工具函数的编程者应该是花费特别的注意力在确保异常传播(propage)给本地方法的调用者.特别是(In particular),我们强调下面两个问题(issue):
.更合适地，工具函数应该提供一个特别的返回值来指示一个异常发生。者简化了调用者检查未决异常的任务。

.另外(In addition),工具函数在异常处理代码中应该按照规则(5.3 部分)来管理局部引用。
为了说明，让我们介绍个工具函数，它是执行一个基于一个实例方法的名字和描述的回调。

```
jvalue
JNU_CallMethodByName(JNIEnv *env,
   jboolean *hasException,
   jobject obj,
   const char *name,
   const char *descriptor,....)
{
 va_list args;
 jclass clazz ;
 jmethodID mid ;
 jvalue result;

 if ( (*env)->EnsureLocalCapacity(env, 2) == JNI_OK) {
  clazz = (*env)->GetObjectClass(env, obj) ;
  mid = (*env)->GetMethodID(env, clazz, name, descriptor) ;
  if( mid ){
   const char *p = descriptor ;

   while( *p != ')') p++ ;
   / * skip ')' */
   p++ ;
   va_start(args, descriptor) ;
   switch(*p) {
   case 'V'
    (*env)->CallVoidMethod(env, obj, mid, args) ;
    break ;
   case '[':
   case 'L':
    result.l = (*env)->CallObjectMethodV(
       env, obj, mid, args) ;
    break ;
   case 'Z':
    result.z = (*env)->CallBooleanMethodV(
       env, obj, mid, args) ;
    break ;
   case 'B':
    result.b = (*env)->CallByteMethodV(
       env, obj, mid, args) ;
    break ;
   case 'C':
    result.b = (*env)->CallCharMethodV(
       env, obj, mid, args) ;
    break ;
```

```
  case 'S':
   result.s = (*env)->CallShortMethodV(
      env, obj, mid, args) ;
   break ;
  case 'I':
   result.i = (*env)->CallIntMethodV(
      env, obj, mid, args) ;
   break ;
  case 'J':
   result.j = (*env)->CallLongMethodV(
      env, obj, mid, args) ;
   break ;
  case 'F':
   result.f = (*env)->CallFloatMethodV(
      env, obj, mid, args) ;
   break ;
  case 'D':
   result.d = (*env)->CallDoubleMethodV(
      env, obj, mid, args) ;
   break ;
  default:
   (*env)->FatalError(env, "illegal descriptor") ;
   break ;
  }
  va_end(args) ;
 }
 (*env)->DeleteLocalRef(env, clazz) ;
}
if ( hasException){
 *hasException = (*env)->ExceptionCheck(env) ;
}
return result ;
}
```

在其他参数中，"JNU_CallMethodByName"有个指向一个"jboolean"的指针。如果所有事都成功，这个"jboolean"被设置为"JNI_FALSE"。如果一个异常在这个函数的执行期间的任何位置发生，这个"jboolean"被设置为"JNI_TRUE"。这给"JNU_CallMethodByName"的调用者一个容易的方法来检测可能的异常。

"JNU_CallMethodByName"首先确信它能创建两个局部引用:一个类的引用和另一个为了从方法调用返回的结果。下一部，从"object"(对象)得到类的引用和查询方法 ID。依靠返回类型，"switch"声明分发到对应的"JNI"方法调用函数。在回调返回后，如果"hasException"不是 NULL,我们调用 ExceptionCheck 来检查未决异常。

"ExceptionCheck"函数是新加的在"Java 2 SDK release 1.2"中。相似的函数是"ExceptionOccurred"函数。不同的是"ExceptionCheck"没有返回异常对象的引用，但当有个未决的异常时返回"JNI_TRUE"和当没有未决异常时返回"JNI_FALSE"。当本地代码只需要知道是否有一个异常，但不需要得到这个异常对象的引用时，"ExceptionCheck"简化了局部引用管理。在"JDK release 1.1"中，前面的代码必须被写，如下:

```
if（hasException){
 jthrowable exc = (*env)->ExceptionOccurred(env) ;
 *hasException = exc != NULL;
 (*env)->DeleteLocalRef(env, exc) ;
}
```

这额外的"DeleteLocalRef"调用是必须的，为了删除对异常对象的几部引用。

使用"JNU_CallMethodByName"函数，我们能重写"InstanceMethodCall.nativeMethod"的实现，在 4.2 部分，如下(as follows)：

```
JNIEXPORT void JNICALL
Java_InstanceMehtodCall_nativeMethod(JNIEnv *env, jobject obj)
{
 printf("In C\n") ;
 JNU_CallMethodByName(env, NULL, obj, "callback","()V") ;
}
```

在"JNU_CallMethodByName"调用后，我们不需要要检查异常，因为本地方法后来立即返回(return immediately afterwards)。

第七章 调用接口

这章告诉你怎样能嵌入一个"Java"虚拟器到你的本地应用程序中。一个 Java 虚拟器实现是典型作为一个本地库的运用。本地应用程序能针对这个库链接和使用载入 Java 虚拟机的调用接口。真正地，在"JDK"或"Java 2 SDK release"中得标准的启动器命令(java)仅仅是一个链接到"Java"虚拟器上的简单 C 程序。这个启动器解析命令行参数，载入虚拟器，和通过调用接口来运行"Java"应用程序。

7.1 创建 Java 虚拟器

为了说明调用接口，让我们看一个"C"程序,它载入一个"Java"虚拟器和调用定义的"Prog.main"方法，如下：

```
public class Prog{
 public static void main(String[] args){
  System.out.println("Hello World" + args[0]) ;
```

```
 }
}
```

下面的"C"程序，"invoke.c"，载入一个"Java"虚拟器和调用"Prog.main"。

```
#include <jni.h>

#define PATH_SEPERATOR ';'
#define PATH_CLASSPATH '.'

main(){
 JNIEnv *env ;
 JavaVM *jvm ;
 jint res ;
 jclass cls ;
 jmethodID mid ;
 jstring jstr ;
 jclass stringClass ;
 jobjectArray args ;

#ifdef JNI_VERSIO_1_2
 JavaVMInitArgs vm_args ;
 JavaVMOption options[1] ;
 options[0].optionString = "-Djava.class.path="USERCLASSPATH ;
 vm_args.version = 0x00010002 ;
 vm_args.options = options ;
 vm_args.ignoreUnrecognized= JNI_TRUE ;

 res = JNI_CreateJavaVM(&jvm, (Void **)&env, &vm_args) ;
#else
 JDK1_1InitArgs vm_args ;
 char classpath[1024] ;
 vm_args.version = 0x00010001 ;
 JNI_GetDefaultJavaVMInitArgs(&vm_args) ;

 sprintf(classpath, "%s%c%s",
  vm_args.classpath, PATH_SEPERATOR, USER_CLASSPATH) ;
 vm_args.classpath = classpath ;

 res = JNI_CreateJavaVM(&jvm, &env, &vm_args) ;
#endif

 if ( res < 0 ){
  fprintf(stderr, "Can't create Java VM\n") ;
  exit(1) ;
 }
 cls = (*env)->FindClass(env, "Prog") ;
```

```
if ( cls == NULL ){
 goto destroy ;
}
mid = (*env)->GetStaticMethodID(env, cls, "main", "([Ljava/lang/String;)V") ;
if ( mid == NULL ){
 goto destory ;
}

jstr = (*env)->NewStringUTF(env, " From C!") ;
if( jstr == NULL ){
 goto destory ;
}
stringClass = (*env)->FindClass(env,"java/lang/String") ;
args = (*env)->NewObjectArray(env, 1, stringClass, jstr) ;
if( args == NULL ){
 goto destory ;
}
(*env)->CallStaticVoidMethod(env, cls, mid, args) ;
destroy:
 if( (*env)->ExceptionOccurred(env) ){
  (*env)->ExceptionDescribe(env) ;
 }
 (*jvm)->DestroyJavaVM(jvm) ;
}
```

这代码条件编译一个初始化结构"JDK1_1InitArgs"，这结构明确虚拟器在"JDK release 1.1"上实现。"Java 2 SDK release 1.2"任然支持"JDK1_1InitArgs",虽然它介绍一个通用(general purpose)虚拟器初始化机构叫做"JavaVMInitArgs"。这个"JNI_VERSION_1_2"常数定义在"Java 2 SDK release 1.2"中，但不在"JDK release 1.1"中。

当目标是"1.1 release"时，"C"代码从调用"JNI_GetDefaultJavaVMInitArgs"得到虚拟器设定开始。"JNI_GetDefaultJavaVMInitArgs"返回值包含堆的大小，栈大小，默认类路经等等(and so on)在"vm_args"参数中。然后我们追加"Prog.class"所在的目录到"vm_args.classpath"结尾。

当目标是"1.2 release"时，"C"代码创建了一个"JavaVMInitArgs"结构体。虚拟器初始参数被存储在"JavaVMOption"数组中。你能设置一般选项(例如(e.g.) -Djava.class.path=。）和实现的特别选项(例如(e.g.)"-Xmx64")来指示相应的"Java"命令行选项。设置"ignoreUnrecognized"域为"JNI_TRUE"命令虚拟器忽略不认识的特别实现选项。

在建立起虚拟器初始化结构后，"C"程序调用"JNI_CreateJavaVM"来载入和初始化"Java"虚拟器。这"JNI_CreateJavaVM"函数填入两个返回值:

.一个接口指针，"jvm",指向最新创建的"Java"虚拟器。
.为了当前线程的"JNIEnv"接口指针"env"。通过"env"接口指针，再调用本地代码访问"JNI"函数。

当"JNI_CreateJavaVM"函数成功返回时，当前本地线程已经引导(bootstrap)自己进入"Java"虚拟器。在这方面，就象运行一个本地方法。因此，在其它事中，能做出"JNI"调用来调用"Prog.main"方法。

最终(Eventually)，程序调用"DestroyJavaVM"函数来载出 Java 虚拟器。（不幸地(Unfortunately)，你不能载出 Java 虚拟器实现在"JDK release 1.1 or Java 2 SDK release 1.2"。"DestoryJavaVM"总是返回一个错误在这些版本(releases)中。）

运行上面程序产品:
Hello World from C!

7.2 链接本地应用程序和"Java"虚拟器（Linking Native Applications with the Java Virtual Machine)

调用接口请求你来链接程序例如"invoke.c"和一个"Java"虚拟器实现。你怎样和 Java 虚拟器链接，依赖于是否本地应用倾向于被布置到一个特别的虚拟器实现，或它被设计在来自不同厂商的不同虚拟器的实现上工作。

7.2.1 和一个知名的 Java 虚拟器链接

你可能决定你的本地应用程序将只被布置在一个特殊虚拟器实现上。在这种情况，你能链接本地应用程序到实现虚拟器的本地库上。例如，"Solaris 的 JDK 1.1 release"上，你能使用以下命令来编译和链接"invoke.c"：
cc -I<jni.h dir> -L<libjava.so dir> -lthread -ljava invoke.c

"-lthread"选项表明我们使用 Java 虚拟器实现带有本地线程支持(8.1.5 部分)。"-ljava"选项指明"libjava.so"是"Solaris"共享库，这共享库实现了"Java"虚拟器。

在 Win32 上带有的"Microsoft Visual C++"编译器，命令行来编译和链接同样代码和"JDK 1.1 release"：
cl -I<jni.h dir> -MD invoke.c -link <javai.lib dir>\javai.lib
(其中 jni.h dir 指的是 jni.h 的目录)

当然，你需要提供正确的头文件和库目录，它们目录是对应于在你机器上 JDK 安装目录。"-MD"选项确保你的本地应用程序被链接到"Win32"多线程"C"库上，同样的"C"库被在"JDK

1.1 and Java 2 SDK 1.2 releases"中的 Java 虚拟器使用。"cl"命令参考了"javai.lib"文件啊，在 Win32 上是"JDK release 1.1"导入的,是为了关于函数接口调用的链接信息，例如在虚拟器中的"JNI_CreateJavaVM"实现。在运行时被用的实际"JDK1.1"虚拟器实现包含在一个独立的动态链接库的"javai.dll"文件中。于此相反(In constrast),同样的 Solaris 系统的共享库文件(.so 文件)也是在链接和运行时备用。

对于"Java 2 SDK release 1.2",虚拟器库名字在"Solaris"变为"libjvm.so",同时在 Win32 上变为"jvm.lib"和"jvm.dll"。总得来说，不同的供应商可以命名他们的不同的虚拟器实现。

一旦编译(compilation)和链接(linking)完成，你能从行命令运行可执行的(executable)文件(resulting)。你可能得到一个系统没有发现一个共享库或一个动态链接库的错误。在"Solaris"上，如果这个错误消息指示系统没有发现共享库"libjava.so"(或者"libjvm.so"在"Java 2 SDK release 1.2"上)，然后你需要添加目录包含虚拟器库的目录到你的"LD_LIBRARY_PATH"变量中。在 Win32 系统，这个错误可能指示找不到动态链接库"javai.dll"(或"jvm.dll"在"Java 2 SDK release 1.2"中)。如果是这种情况，添加包含"DLL"的目录到你的 PATH 环境变量中。

7.2.2 和知名的 Java 虚拟器链接

如果应用程序倾向于和来自不同供应商的虚拟器的实现一起工作，你就不能链接本地应用程序和一个指定的实现了一个虚拟器的库。因为"JNI"不能详细说明实现一个"Java"虚拟器的本地库的名字，你应该准备使用不同名字发布的 Java 虚拟器实现。例如，在 Win32 上，虚拟器在 JDK release 1.1 中被作为"javai.dll"发布，在"Java 2 SDK release 1.2"中作为"jvm.dll"发布。

解决方法是使用运行时动态链接到(use run-time dynamic linking to)载入的指定的应用程序需要的虚拟器库。然后，虚拟器库的名字能被使用一种应用程序指定的方法来配置。例如，下面的"Win32"代码找到被给一个虚拟器库的路径上的函数"JNI_CreateJavaVM"入口地址:

```
void *JNU_FindCreateJavaVM(char *vmlibpath)
{
 HINSTANCE hVM = LoadLibrary(vmlibpath) ;
 if ( hVM == NULL ){
  return NULL ;
 }
 return GetProcAddress(hVM, "JNI_CreateJavaVM") ;
}
```

"LoadLibrary"和"GetProcAddress"都是在 Win32 上的动态链接的 API 函数。虽然"LoadLibrary"能接受实现"Java"虚拟器的本地库的名字(例如"jvm"）或路径(例如"C:\\jdk1.2\\jre\\bin\\classic\\jvm.dll")，最好是你传递一个本地库的绝对路径给"JNU_FindCreateJavaVM"函数。依赖于"LoadLibrary"来搜索"jvm.dll"文件，使你的应用程序很容易变化配置，例如添加到"PATH"环境变量。

"Solaris"本版是相似的:

```
void *JNU_FindCreateJavaaVM(char *vmlibpath)
{
 void *libVM = dlopen(vmlibpath, RTLD_LAZY);
 if( libVM == NULL ){
  return NULL ;
 }
 return dlsym(libVM, "JNI_CreateJavaVM") ;
}
```

"dlopen"和"dlsym"函数在“Solaris"上来支持动态链接的共享库。

7.3 附加本地线程(Attaching Native Threads)

假设你有个多线程的应用程序例如一个用"C"写的"web server"。当 HTTP 请求到来时，Web 服务创建多个本地线程来处理并发的"HTTP"请求。我们想嵌入(embed)一个 Java 虚拟器在这个服务中，所以同时多线程能执行在虚拟器上的操作，如在"Figure 7.1"中的说明。

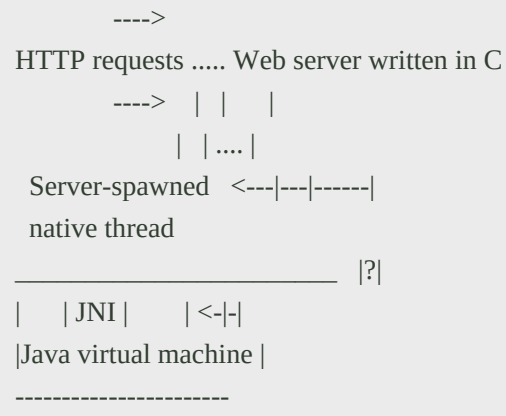


Figure 7.1 Embedding th Java virtual machine in a web server

服务器孵化的本地方法可能其生命比 Java 虚拟器还要短。因此，我们需要一个方法来附加一个本地线程到一个正在运行的 Java 虚拟器上，在这个被附加的本地线程上执行了"JNI"调用，然后在不破坏其他附加线程情况下从虚拟器分离本地线程。

接下来的例子，"attach.c"，说明怎样附加本地线程到一个使用调用接口的虚拟器。这个程序使用"Win32"线程"API"来写的。相似的版本能被写为"Solaris"和其他操作系统。

```
#include <windows.h>
#include <jni.h>
```

```
JavaVM *jvm ;
#define PATH_SEPERATOR ';'
#define PATH_CLASSPATH '.'
void thread_fun(void *arg)
{
 jint res ;
 jclass cls ;
 jmethodID mid ;
 jstring jstr ;
 jclass stringClass ;
 jobjectArray args ;
 JNIEnv *env
 char buf[100] ;
 int threadNm = (int)arg ;

#ifdef JNI_VERSION_1_2
 res = (*jvm)->AttachCurrentThread(jvm, (void**)&env, NULL ) ;
#else
 res = (*jvm)->AttachCurrentThread(jvm, &env, NULL) ;
#endif
 if( res < 0 ){
  fprintf(stderr, "Attach failed\n") ;
  retrn ;
 }
 cls = (*env)->FindClass(env, "Prog") ;
 if ( cls == NULL ){
  goto detach
 }
 mid = (*env)->GetStaticMethodID(env, cls, "main", "([Ljava/lang/String;)V") ;
 if ( mid == NULL ){
  goto detach ;
 }

 sprintf(buf, " from Thread %d", threadNum) ;
 jstr = (*env)->NewStringUTF(env, buf) ;
 if( jstr == NULL ){
  goto detach ;
 }

 stringClass = (*env)->FindClass(env, "java/lang/String") ;
 args = (*env)->NewObjectArray(env, 1, stringClass, jstr) ;
 if ( args == NULL ){
```

```
 goto detach ;
 }

 (*env)->CallStaticVoidMethod(env, cls, mid , args) ;
detach:
 if( (*env)->ExceptionOccurred(env)){
  (*env)->ExceptionDescribe(env) ;
 }
 (*jvm)->DetachCurrentThread(jvm) ;
}
main(){
 JNIEnv *env ;
 int i ;
 jint res ;
#ifdef JNI_VERSION_1_2
 JavaVMInitArgs vm_args ;
 JavaVMOption options[1] ;

 options[0].option.String = "-Djava.class.path="USER_CLASSPATH ;
 vm_args.version = 0x00010002 ;
 vm_args.options = options ;
 vm_args.nOptions = 1 ;
 vm_args.ignoreUnrecognized = TRUE ;

 res = JNI_CreateJavaVM(&jvm, (void**)&env, &vm_args) ;
#else
 JDK1_1InitArgs vm_args ;
 char classpatch[1024] ;

 vm_args.version = 0x00010001 ;
 JNI_GetDefaultJavaVMInitArgs(&vm_args) ;

 sprintf(classpath, "%s%c%s",
  vm_args.classpath, PATH_SEPARATOR, USER_CLASSPATH) ;
 vm_args.classpath = classpath ;

 res = JNI_CreateJavaVM(&jvm, &env, &vm_args) ;
#endif
 if( res < 0 ){
  fprintf(stderr, "Can't create Java VM\n") ;
  exit(1) ;
 }
```

```
 for ( i = 0 ; i < 5 ; i++ )

  _beginthread(thread_fun, 0, (void *) i ) ;
 sleep(1000) ;
 (*jvm)->DestroyJavaVM(jvm) ;
}
```

"attach.c"程序是一个"invoke.c"的变种。不是在主线程中调用"Prog.main"函数，而是本地代码启动了五个线程。一旦产生了线程，然后等待线程们都开始，再调用"DestroyJavaVM"。每个产生的线程都附加自己到"Java"虚拟器上,调用"Prog.main"方法， 同时最后在虚拟器终止前从虚拟器分离自己。在所有五个线程终止后，"DestroyJavaVM"返回。我们忽略"DestroyJavaVM"的返回值，因为在"JDK release 1.1 and Java 2 SDK release 1.2"中这个函数不能完整被执行。

"JNI_AttachCurrentThread"把 NULL 作为它的第三个参数。"Java 2 SDK release 1.2"介绍了"JNI_ThreadAttachArgs"机构体。允许你指定额外的参数，例如你想附加的线程组。"JNI_ThreadAttachArgs"机构体的细节作为"JNI_AttachCurrentThread"的定义的一部分被详细描述在 13.2 部分(section)。

当程序执行函数"DetachCurrentThread",它释放所有属于当前线程的局部引用。

运行程序产生下面输出:

```
Hello World from thread 1
Hello World from thread 0
Hello World from thread 4
Hello World from thread 2
Hello World from thread 3
```

输出的精确(exact)顺序将可能不同，依赖于在线程安排中的随机因素。

第八章 JNI 的附加功能(Additional JNI Features)

我们已经讨论了 JNI 被使用来写本地方法和嵌入一个 Java 虚拟器实现到一个本地应用程序中的功能。这章介绍 JNI 剩余的功能。

8.1 JNI 和线程(JNI and Threads)

Java 虚拟器支持控制并发的在一样地址空间中执行的多线程。这并发性引起一个复杂(complexity)的程度，这在一个单线程环境中是没有的。多线程可以访问同一个对象，同一个文件描述符--简而言之，一样的共享资源--同一个时间。

为最有效的理解(get the most out of)这部分，你应该了解多线程编程的概念。你应该知道怎样写使用多线程的 Java 应用程序和怎样对共享资源的同步访问。一个好的关于用 Java 编程语言编写多线程的参考书是"Concurrent Programming in Java, Design Principles and Patterns，by Doug Lea(Addsion-Wesley 1997)"

8.1.1 限制(Constraints)

当写本地方法为在都线程环境中运行时，我们必须在思想上记住某些局限。通过理解这些局限和使用这些局限编程，你的本地方法将安全执行，无论多少线程同时地(simultaneously）执行一个给定本地方法。例如:
.一个"JNIEnv"指针只在和其关联的线程中有效。你不必传递这个指针从一个线程到另一个线程，或者在多线程中缓冲和使用它。在从同一个线程的并发调用中，"Java"虚拟器传递给一个本地方法同样的"JNIEnv"指针，但当从不同线程中调用那个本地方法时，传递不同的"JNIEnv"指针。避免在一个线程缓冲"JNIEnv"指针和在另一线程中使用这个指针的一般性错误。
.局部引用只在创建它们的线程中是有效的。你不必传递局部引用从一个线程到另一个线程。你应该总是转换局部引用为全局引用，任何时候多线程可以使用一样的引用成为可能。

8.1.2 监视入口和出口(Monitor Entry and Exit)

监视是在 Java 平台上的基本同步机制(primitive sychronization mechanism)。每个对象能被动态地关联到一个监视上。"JNI"允许你使用监视来同步，因此在 Java 编程语言中，实现了等同于(equivalent to)一个同步块(synchronized block)的功能(functionality):

```
synchronized(obj){
 ....  // synchronized block
}
```

Java 虚拟器保证了在线程执行块中的任何语句前，线程获得和"obj"对象关联的监视。这确保在给定的任何时候这儿至多有一个线程拥有监视和在同步块中执行。当一个线程等待另一个线程退出监视时，这线程阻塞。

在 JNI 引用上，本地代码能使用"JNI"函数来实现一样的同步。你能使用"MonitorEnter"函数来进入监视和"MonitorExit"函数来退出监视:

```
if( (*env)->MonitorEnter(env,obj) != JNI_OK){
 ...
}
...
if( (*env)->MonitorExit(env, obj) != JNI_OK){
 ...
}
```

执行以上代码，在执行在同步块中的任何代码前，一个线程首先必须进入和"obj"关联的监视。"MonitorEnter"操作使用一个"jobject"来作为一个参数，同时如果另一线程已经进入了和这个"jobject"关联的监视，这个线程被阻塞。当当前线程没有拥有这个监视时，调用"MonitorExit"导致一个错误同时产生一个"IllegalMonitorStateException"。以上代码包含一对匹配的"MonitorEnter"和"MonitorExit"调用，然而我任然需要检查可能的错误。例如，若果底层线程实现不能分配必须的资源来执行监视操作，监视操作就可能失败。

"MonitorEnter"和"MonitorExit"能运行在"jclass, jstring, and jarray"类型上，这是"jobject"引用的特殊类型。

记得用恰当(appropriate)数目的"MonitorExit"调用来匹配一个"MonitorEnter",特别是在处理错误和异常的代码中:

```
if( (*env)->MonitorEnter(env, obj) != JNI_OK) ...;
...
if ( (*env)->ExceptionOccurred(env) ){
 ....

 if ( (*env)->MonitorExit(env, obj) != JNI_OK) ... ;
}
...
if ( (*env)->MonitorExit(env, obj) != JNI_OK) ... ;
```

调用"MonitorExit"的失败将几乎可能导致死锁。通过比较上面"C"代码片度和这部分开始处的代码片段，你能认识到用 Java 编程语言编程比用 JNI 更容易。因此，在 Java 编程语言中表示同步结构更可行(preferable)。例如，如果一个静态本地方法需要进入和它的定义类关联的监视，你应该定义一个静态变量来同步本地方法，而不是在本地方法中执行 JNI 层(JNI-level)同步监视(monitor sychronization)。

8.1.3 监视等待和通知(Monitor Wait and Notify)

"Java API"包含一些其他的方法，对于线程的同步是有用的。它们是"Object.wait, Object.notify and Object.notifyAll"。没有 JNI 函数被提供来直接对应这些方法，因为监视等带和通知操作没有和监视进入和退出操作一样的关键性能(performance cirtical)。本地代码可以用 JNI 方法调用机制来调用在 Java API 中对应的方法来替代:

```
static jmethodID MID_Object_wait ;
static jmehtodID MID_Object_notify ;
static jmethodID MID_Object_notifyAll ;
void
JNU_MonitorWait(JNIEnv *env, jobject object, jlong timeout)
```

```
{
 (*env)->CallLongMethod(env, object, MID_Object_wait, timeout) ;
}
void
JNU_MonitorNotify(JNIEnv *env, jobject object)
{
 (*env)->CallVoidMethod(env, object, MID_Object_notify) ;
}
void
JNU_MonitorNotifyAll(JNIEnv *env, jobject object)
{
 (*env)->CallVoidMethod(env, object, MID_Object_notifyAll) ;
}
```

我们假设对"Object.wait, Object.notify, and Object.notify"的方法"IDs"已经被得到在其他地方同时缓冲到全局变量中。像在 Java 编程语言中一样，只在拥有和 jobject 参数关联的监视时，你能调用上面监视相关的函数。

8.1.4 在任意上下文中获取一个指向"JNIEnv"指针 (Obtaining a JNIEnv Pointer in Arbitrary Contexts)

我们较早地解释一个"JNIEnv"指针只在它关联的线程中有效。一般地，这对于本地方法不是一个问题，因为他们从虚拟器得到"JNIEnv"指针作为第一个参数。然而，有时候(Occasionally)，它可能对一小段本地代码时必须的，它不能从虚拟器被直接调用来得到属于当前线程的"JNIEnv"接口指针。例如，本地代码可以是一个被操作系统调用的"callback"函数，在这种情况中"JNIEnv"指针将可能不是作为一个参数的变量。

通过"AttachCurrentThread"函数的接口调用,你能为当前线程获得"JNIEnv"指针:

```
JavaVM *jvm
f()
{
 JNIEnv *env ;
 (*jvm)->AttachCurrentThread(jvm, (void **)&env, NULL ) ;
 ....
}
```

当当前线程已经被附加到虚拟器上的时候，"AttachCurrentThread"返回属于当前线程的"JNIEnv"的接口指针。

这儿有许多方法来获得"JavaVM"指针：通过在创建虚拟器的时候记录它；通过使用"JNI_GetCreateJavaVMs"来查询被创建的虚拟器；通过在一个一般的本地方法中调用"JNI"函数"GetJavaVM"；或者通过定义一个"JNI_OnLoad"处理。和"JNIEnv"指针不一样，在多线程中"JavaVM"指针保持有效，因此它能被缓冲在一个全局变量中。

"Java 2 SDK release 1.2"提供一个新的调用接口函数"GetEnv"，所以你能检查当前线程是不是被附加到虚拟器上，同时如果如此，返回属于当前线程的"JNIEnv"指针。如果当期线程已经附加到虚拟器上，"GetEnv"和"AttachCurrentThread"有一样的功能。

8.1.5 匹配线程模型(Matching the Thread Models)

假设(suppose)运行在多线程中的本地代码访问一个全局资源。本地代码应该使用"JNI"函数"MonitorEner and MonitorExit"，或者使用在主机环境中的本地线程同步原语(例如"mutex_lock"在 Solaris 系统上)?相似地，如果本地代码需要创建新的线程，它应该创建一个"java.lang.Thread"对象和通过"JNI"来执行"Thread.start"的回调，或者它应该在本机环境中使用本地线程创建原语(creation primitive)(例如"thr_create"在 Solaris 系统上)?

答案是如果"Java"虚拟器实现支持一种线程模型，且线程模型匹配了通过本地代码的虚拟器，那时所有这些方法(approach)都能工作。线程模型(thread model)指示(dictat)系统怎样实现必要的线程操作，例如时序安排(scheduling)，上下文的切换(context switching)，同步(sychronization)，和在系统调用中的阻塞(blocking)。另一方面，在一个用户的线程模型中，应用程序代码实现了线程的操作。例如，在 Solaris 系统上被"JDK"和"Java 2 SDK releases"导出的"Green thread"模型使用"ASCI C"函数"setjmp"和"longjmp"来实现上下文的切换。

许多模型的操作系统(例如"Solaris"和"Win32")都支持本地线程模型。不幸地，一些操作系统任然缺少本地线程的支持。替代地，在这些操作系统上有一个或多个用户线程包。

如果你严格地用 Java 编程语言来写应用程序，你不需要担心虚拟器实现的底层线程模型。"Java"平台能被移植到任何支持线程原语请求设置的主机环境。大多本地和用户线程包提供必需的线程原语，为实现一个"Java"虚拟器。

另一方面，JNI 编程者必须注意线程模型。如果"Java"虚拟器实现和本地代码有不同的线程和同步概念(a different notion of threading and sychronization），使用本地代码应用程序可能无法正常执行(function properly)。例如，一个本地方法能在它自己线程模型中的一个同步操作中被阻塞，但"Java"虚拟器运行在一个不同的线程模型中，可能不知道执行本地方法的线程被阻塞了。因为没有其他线程将本安排执行，所以应用程序锁死。

如果本地代码和"Java"虚拟器实现使用一样的线程模型，这线程模型匹配。如果"Java"虚拟器实现使用本地线程支持，本地代码能自由地调用在主机环境中的相关线程原语。如果"Java"虚拟器实现是基于一个用户线程包，本地代码因该链接到一样的用户线程包上或依赖

于非线程操作。后者可能比你想象的更难实现：大多 C 库调用(例如，"I/O"和内存分配功能)在下面(underneath)执行了线程同步。除非本地代码执行纯粹的计算和不使用库的调用，它很可能间接地(indirectly)使用了线程原语。

大多虚拟器实现只支持一个特别的线程模式为"JNI"本地代码.支持本地线程的实现是最灵活的，因此当本地线程可用时，它是在被给定主机环境的首选。可能严格地限制，依赖于一个特别的用户线程包的虚拟器实现，做为它们能操作本地代码的类型。

一些虚拟器实现可以支持大量的不同的线程模型。一个更灵活(more flexible)类型的虚拟器实现可以允许你提供一个自定义的线程模式实现为虚拟器的内部使用，因而确保虚拟器实现能带了你的本地代码工作。在在可能请求本地代码的工程上着手处理(embark)前，你应该参考(consult)说明你的虚拟器实现的线程模式限制的文档。

8.2 写国际化的代码(Writing Internationalized Code)

必须特别关心写的代码能很好地工作在多个地区。"JNI"给程序员完整的访问 Java 平台的国际化的功能。我们将使用字符串转换(string conversion）作为一个例子，因为文件的名字和消息在许多地域中可以包含非 ASCII 字符。

"Java"虚拟器用"Unicode"格式来表示字符串(string)。虽然一些本地平台(例如 Window NT)也提供"Unicode"的支持，但大多使用本地指定的编码来表示字符串。

不能使用"GetStringUTFChars"和"GetStringUTFRegion"函数来做在"jstring"和本地指定字符串之间的转换，除非在平台上本地的编码是"UTF-8"。"UTF-8"字符串是有用的，当表示名字和描述符(例如"GetMethodID"的参数)来被传递给"JNI"函数时,但表示本地指定编码的字符串例如文件名字是不恰当的。

8.2.1 从本地字符串创建"jstring"(Create jstring from Native Strings)

使用"string(byte[] bytes)"构造器(constructor)来转换一个本地字符串为一个"jstring"。下面工具函数创建一个"jstring"从一个本地指定的本地"C"字符串（a local-specific native C string)：

```
jstring JNU_NewStringNative(JNIEnv *env, const char *str)
{
 jstring result ;
 jbyteArray bytes = 0 ;
 int len ;
 if( (*env)->EnsureLocalCapacity(env, 2） < 0 ){
  return NULL ;
 }
```

```
 len = strlen(str) ;
 bytes = (*env)->NewByteArray(env, len) ;
 if ( bytes != NULL ){
  (*env)->SetByteArrayRegion(env, bytes, 0, len, (jbyte *)str) ;
  result = (*env)->NewObject(env, Classjava_lang_String,
       MID_String_init, bytes) ;
  (*env)->DeleteLocalRef(env, bytes) ;
  return result ;
 }

 reurn NULL ;
}
```

这个函数创建一个"byte"数组，复制本地"C"字符串到"byte array"，和最终调用 String(byte[] bytes)构造函数来创建"jstring object"的结果。"Class_java_lang_String"是对于"java.lang.String"类的一个全局引用,同时"MID_String_init"是字符构造器的方法"ID"。因为这是一个工具函数，我们确保删除"byte array"的局部引用，这局部引用被创建来临时储存字符串的。

如果你需要在"JDK release 1.1"中使用这个函数，删除"EnsureLocalCapacity"的调用。

8.2.2 转换"jstrings"到本地字符串(Translating1 jstrings to Native Strings)

使用"String.getBytes"方法来转换一个"jstring"为恰当的本地编码。下面工具函数转换一个"jstring"为一个区域指定(locale-specific)的本地"C"字符串:

```
char *JNU_GetStringNativeChars(JNIEnv *env, jstring jstr)
{
 jbyteArray bytes = 0 ;
 jthrowable exc ;
 char *result = 0 ;
 if( (*env)->EnsureLocalCapacity(env, 2) < 0 ){
  return 0 ;
 }
 bytes = (*env)->CallObjectMethod(env, jstr, MID_String_getBytes) ;

 exc = (*env)->ExceptionOcurrend(env) ;
 if( !exc){
  jint len = (*env)->GetArrayLength(env, bytes) ;
  result = (char *)malloc(len+1) ;
  if ( result == 0 ){
   JNU_ThrowByName(env, "java/lang/OutOfMemoryError", 0 ) ;
```

```
  (*env)->DeleteLocalRef(env, bytes) ;
  return 0 ;
 }
 (*env)->GetByteArrayRegion(env, bytes, 0, len, (jbyte *)result) ;
 result[len] = 0 ;
}
else{
 (*env)->DeleteLocalRef(env, exc) ;
}
(*env)->DeleteLocalRef(env, bytes) ;

 return result ;
}
```

这个函数传递"java.lang.String"引用到"String.getBytes"方法，然后复制"byte array"的元素到一个最新分配的"C"数组。"MID_String_getBytes"是"String.getBytes"方法的预先计算的(precomputed)方法 ID。因为这是个工具函数，我们确保删除"byte array"的局部引用和异常对象。记住(Keep in mind)删除一个异常对象的"JNI"引用不会清除未决的异常。

再次，如果你需要使用这个函数在 JDK release 1.1 中，删除"EnsureLocalCapacity"的调用。

8.3 注册本地方法(Registering Native Methods)

在一个应用程序执行一个本地方法前，它通过一个两步处理(a two-step process)来载入包含本地方法实现的本地库和然后(and then）链接到本地方法实现上:

1."System.loadLibrary"定位和载入命名的本地库。例如，"System.loadLibrary("foo")"可以在 Win32 平台载入"foo.dll"。

2.虚拟器在载入的一个本地库中定位本地方法实现本地方法实现。例如，一个"Foo.g"本地犯法调用请求定位和链接本地函数"Java_Foo_g",它驻留在"foo.dll"中。

这章将介绍另一个方法来完成第二步。替代依赖虚拟器来搜索本地方法在已经载入的本地库中，"JNI"编程者能手动地(manually）链接本地函数，通过注册带有一个类引用，方法名字和方法描述符的一个函数指针:

```
JNINativeMethod nm;
nm.name = "g";

nm.signature = "()V" ;
nm.fnPtr = g_impl ;
(*env)->RegisterNatives(env, cls, *nm, 1) ;
```

上面代码注册本地函数"g_impl"做为"Foo.g"本地方法的实现:
void JNICALL g_impl(JNIEnv *env, jobject self) ;

这个本地函数"g_impl"不需要按照"JNI"命名转换，因为只涉及函数指针，不需要从库中导出(因此这儿不需要用 JNIEXPORT 来声明函数)。然而，本地函数"g_impl"任然，跟在"JNICALL"调用的转换后。

"RegisterNatives"函数对于许多用途是很有用的：
.有时候积极地(eagerly)注册大量的本地方法的实现更方便和高效(more convenient and more effcient)，相对于让虚拟器懒洋洋地(lazily)链接这些入口。
.你可以在一个方法中调用"RegisterNatives"多次，允许本地方法实现在运行时被更新。
.在一个本地应用程序嵌入一个虚拟器实现和需要链接一个定义在本地应用程序中的本地方法实现，"RegisterNatives"是非常(particularly)有用的。虚拟器应该不能自动地找到这个本地方法实现，因为它只能在本地库中搜索，不会在应用程序自身中搜索。

8.4 载入和载出处理程序(Load and Unload Handlers)

载入和载出处理程序允许本地库导出两个函数:一个在"System.loadLibrary"载入本地库时调用，另一个在虚拟器载出本地库时调用。这个特征在"Java 2 SDK release 1.2"中被加入。

8.4.1 JNI_OnLoad 处理程序(The JNI_OnLoad Handler）

当"System.loadLibrary"载入一个本地库时，虚拟器在本地库中搜索下面导出入口:
JNIEXPORT jint JINCALL JNI_OnLoad(JavaVM *jvm, void *reserved) ;

在"JNI_OnLoad"的实现中，你能调用任何"JNI"函数。"JNI_OnLoad"处理程序的典型使用时缓冲"JavaVM"指针，类的引用，或者方法和域的"IDs",像在下面例子中显示的(as shown in the following example):

```
JavaVM *cached_jvm ;
jclass Class_C ;
jmethodID MID_C_g ;
JNIEXPORT jint JNICALL
JNI_OnLoad(JavaVM *jvm, void *reserved)
{
 JNIEnv *env ;
 jclass cls ;

 cached_jvm = jvm  ;
 if（  (*jvm)->GetEnv(jvm, (void **)&env, JNI_VERSION_1_2)){
  return JNI_ERR ;
 }
```

```
cls = (*env)->FindClass(env, "C") ;
if( cls == NULL ){
 return JNI_ERR ;
}

Class_C = (*env)->NewWeakGlobalRef(env, cls) ;
if( Class_C == NULL ){
 return JNI_ERR ;
}

MID_C_g = (*env)->GetMethodID(env, cls, "g","()V") ;
if(MID_C_g == NULL ){
 return JNI_ERR ;
}
return JNI_VERSION_1_2 ;
}
```

"JNI_OnLoad"函数首先缓冲"JavaVM"指针到全局变量"cached_jvm"中。然后通过调用"GetEnv",来获得"JNIEnv"指针。最后载入"C class"，缓冲这个类引用，和计算了"C.g"的方法"ID"。"JNI_OnLoad"函数在错误时返回"JNI_ERR"(12.4 部分)，否则放回被本地库需要的"JNIEnv"版本号"JNI_VERSION_1_2"。

我们将在下一部分中解释为什么我们缓冲"C class"在一个弱全局引用中来替代一个全局引用。

被给一个缓冲"JavaVM"接口指针，对于实现一个允许本地代码来得到当前线程的"JNIEnv"接口指针的工具是微不足道的。

```
JNIEnv *JNU_GetEnv()
{
 JNIEnv *evn ;
 (*cached_jvm)->GetEnv(cached_jvm, (void **)&env, JNI_VERSION_1_2) ;
 return env ;
}
```

8.4.2 JNI_OnUnload 处理程序(The JNI_OnUnload Handler)

直觉地(intuitively),当虚拟器载出一个"JNI"本地库时，虚拟器调用"JNI_OnUnload"处理程序。然而(However),这不够精确(precise enough)。什么时候虚拟器决定它载出一个本地库?哪个线程运行"JNI_OnUnload"处理程序？

载出本地库的规则是如下(as follows):
.虚拟器关联每个本地库使用"class C"的类载入器"L"调用了"System.loadLibrary"函数。
.在它决定类载入器"L"不在是一个活的对象(a live object)后，虚拟器调用"JNI_OnUnload"处理，同时载出本地库。因为一个类载入器查看了这个虚拟器定义的所有的类，这暗示(imply)类 C 也能被载出。
."JNI_OnUnload"处理程序在最后运行，且被"java.lang.System.runFinalization"同步地调用或者被虚拟器同步地调用。

"JNI_OnUnload"处理程序的定义清除了在上一部分中"JNI_OnLoad"分配的资源:

```
JNIEXPORT void JNICALL
JNI_OnUnload(JavaVM *jvm, void *reserved)
{
 JNIEnv *env ;
 if((*jvm)->GetEnv(jvm, (void **)&env, JNI_VERSION_1_2)){
  return ;
 }

 (*env)->DeleteWeakGlobalRef(env, Class_C) ;
 return ;
}
```

"JNI_OnUnload"函数删除了"C class"在"JNI_OnLoad"处理程序中创建的弱全局引用。我们不需要删除方法 ID "MID_C_g"，因为在载出"class C"定义时，虚拟器自动地回收代表"C"的方法 IDs 的需要的资源。

我们现在准备解释为什么我们缓冲"C class"到一个弱全局应用替代一个全局应用。一个全局引用应该保持"C"活跃，转而它应该保持"C"的类载入器活跃(alive)。由于(Given that)本地库是关联在"C"的类载入器"L"上，本地库不应该被载出和"JNI_OnUnload"不应该被调用。

"JNI_OnUnload"处理程序在最后(in a finalizer)运行。相反(In contrast)，"JNI_OnLoad"处理程序在发起(initiate）"System.loadLibrary"调用的线程中运行。因为"JNI_OnUnload"在一个未知的线程上下文中运行，为了避免可能死锁，在"JNI_OnUnload"中你应该避免复杂的同步和锁操作。"JNI_OnUnload"处理程序典型地运行简单的任务例如释放被本地库分配的资源。

在类载入器载入库和被这个类载入器定义的所有类不在活跃时，"JNI_OnUnload"处理程序运行。"JNI_OnUnload"处理程序不得以任何方式(in any way)使用这些类。在上面"JNI_OnUnload"定义中，你不得执行任何操作，操作假设"Class_C"任然指向一个有效的类。在这个例子中"DeleteWeakGlobalRef"调用释放弱全局引用自己，但不以任何方式操作引用的"class C"。

总之(In summary)，当写"JNI_OnUnload"处理程序时，你应该小心。避免复杂的可能导致死锁的锁操作。记住当"JNI_OnUnload"处理程序被调用时，类已经被载出了。

8.5 反馈支持(Reflection Support)

一般地，反馈提到在运行时操作语言级的构造。例如，反馈允许你在运行(at run time)时发现任何类对象，系列域和在类中定义的方法的名字。在 Java 编程语言级通过"java.lang.reflect"包来提供反馈的支持，和在"java.lang.Object"和"java.lang.Class"类中的一些方法一样。虽然你总能调用对应的"Java API"来执行(carry out)反馈操作，"JNI"提供下面函数使来自本地代码的频繁的反馈操作更有效和方便。

."GetSuperclass"返回一个被给类引用的父类。

."IsAssignableFrom"检查一个类的实体是否能被用，当另一个类的事例期待使用时。

."GetObjectClass"返回被给"jobject"引用的类。

."IsInstanceOf"检查一个"jobject"对象是否是一个被给类的实体。

."FromReflectedField and ToReflectedField"允许本地代码在域"ID"和"java.lang.reflect.Field"对象之间转换。他们是在"Java 2 SDK release 1.2"中新增的。

."FromReflectedMethod and ToReflectedMethod"允许本地代码在方法"IDs","java.lang.reflect.Method objects"和"java.lang.reflect.Constructor objects"之间转换。他们是在"Java 2 SDK release 1.2"中新增的。

8.6 JNI 在 C++中的编程(JNI Programming in C++）

"JNI"对于"C++"编程者表示一个稍微(slightly）简单接口。"jni.h"文件包含一系列定义，所以C++编程者能写，例如:

```
jclass cls = env->FindClass("java/lang/String");
```

替代在"C"中:

```
jclass cls = (*env)->FindClass(env, "java/lang/String");
```

在"env"上额外级别的间接寻址，和"FindClass"的"env"参数的对编程者的隐藏。"C++"编译器内联"C++"成员函数调用来等同于 C 成员函数调用(同行(counterparts）；结果的编码是一样的。在"C"或"C++"中使用"JNI"之间没有内在的(inherent)性能差别。

此外(In addition),"jni.h"文件也定义一些列空的"C++"类来强制在不同的"jobject"子类型中子类化联系:

```
// JNI reference type defined in C++
class _jobject{} ;
class _jclass: public _jobject{} ;
class _jstring: public _jobject{} ;
...
typedef _jobject * jobject ;
typedef _jclass* jclass ;
typedef _jstring* jstring ;
...
```

"C++"编译器能在编译时发现你是否传递，例如，一个"jobject"给"GetMethdodID"：

```
// ERROR: pass jobject as a jclass :
jobject obj = env->NewObject(...) ;
jmethodID mid = env->GetMethodID（obj, "foo", "()V") ;
```

因为"GetMethodID"希望一个"jclass"引用，"C++"编译将给出一个错误消息。在"C"类型定义的 JNI 中，jclass 是和 jobject 一样的:

```
typedef jobject jclass ;
```

因此，一个 C 编译器不能检查你错误地(mistakenly)传递一个"jobject"替代"jclass"。

在 C++中加入类型的层次(type hirrarchy)有时额外的转换(casting)成必要(necessitate）。在"C"中,你能从一个字符串数组中得到一个字符串，赋结果到一个"jstring"：

```
jstring jstr = (*env)->GetObjectArrayElement(env, arr, i) ;
```

然而， 在"C++"中你需要插入一个清晰的转换:

```
jstring jstr = (jstring)env->GetObjectArrayElement(arr, i) ;
```

第九章 利用存在的本地库(Leveraging Existing Native Libraries)

一个 JNI 的应用程序是写利用在存在本地库中代码的本地方法。在这章中，一个典型的方法(approach)是生成一个包装一些列本地函数的类库。

这章首先讨论最易懂的(straightforward)写封装类(wrapper classer)的方法--一对一的映射。然后我们介绍一个技术，共享存根(shared stubs),简化了些封装类的任务。

一对一的映射和共享存根都是封装本地函数的技术。这章的最后，我们将讨论怎样使用"peer classes"来封装本地数据结构。

这章中描述的方法直接公开了一个使用本地方法的本地库，因此使一个应用调用依赖于这样本地库的这样的本地方法是有缺点。如此一个应用可能只能运行在提供本地库的操作系统上。一个更好的方法是宣布一个独立于操作系统的本地方法。仅仅实现这些本地方法的本地函数直接地使用本地库，限制了对这些本地函数移植的需要。包含本地方法声明的应用程序不需要被移植。

9.1 一对一映射(One-to-One Mapping)

让我们开始一个简单的例子。假设我们想写个封装类来导出在标准"C"库的"atol"函数:

```
long atol (const char *str) ;
```

这个"atol"函数解析一个字符串和返回被字符串代表的十进制的值。实际上(in practice)可能没有原因要定义如此一个本地方法，因为"Integer.parseInt"方法，"Java API"的一部分，提供同样的功能(equivalent functionality)。例如，估算"atol（"100")"，结果是整数 100(result in the integer value 100)。我们定义一个封装类如下:

```
public class C{
 public static native int atol(String str) ;
 ...
}
```

为说明在"C++"中"JNI"编程的好处，我们将在这章中使用"C++"来实现本地方法。"C.atol"本地方法的"C++"实现如下：

```
JNIEXPORT jint JNICALL
Java_C_atol(JNIEnc *env, jclass, jstring str)
{
 const char *cstr = env->GetStringUTFChar(str, 0) ;
 if( cstr == NULL ){
  return 0 ;
 }
 int result = atol(cstr) ;
 env->ReleaseStringUTFChars(str, cstr) ;
 return result ;
}
```

这个实现是很简单的。我们使用"GetStringUTFChars"来转换"Unicode string"，因为十进制数是"ASCII"码的字符(ASCII characters)。

现在让我们测试一个复杂的例子，它调用一个传递结构体指针的 C 函数。假设我们想写个公开来自"Win32 API"的"CreateFile"函数的封装类：

```
typedef void *HANDLE ;
typedef long DWORD ;
typedef struct {...}SECURITY_ATTRIBUTES ;
HANDLE CreateFile(
 const char *fileName,   // file name
 DWORD desiredAccess,   // access(read-write) mode
 DWORD shareMode,    // share mode
 SECURITY_ATTRIBUTES *attrs,  // security attributes
 DWORD creationDistribution,  // how to create
```

```
 DWORD flagsAndAttributes,  // file attributes
 HANDLE templateFile   // file with attr. to copy
) ;
```

"CreateFile"函数支持大量的"Win32"规定特征，在平台无关的"Java File API"中不可用。例如，"CreateFile"函数可以被用来指定特殊的访问模式和文件属于，来打开Win32 命名管道(Win32 named pips)和来处理串口通讯(serial port communications)。

我们在这本书中将不讨论更多的"CreateFile"函数的细节。关注于"CreateFile"可以怎样被映射到一个本地函数，函数被定义在一个叫"Win32"的封装类中:

```
public class Win32{
 public static native int CreateFile(
  String fileName,    // file name
  int desiredAccess,   // access(read-write) mode
  int shareMode,    // share mode
  int[] secAttrs,    // security attributes
  int creationDistribution,  // how to create
  int flagsAndAttributes,   // file attributes
  int templateFile    // file with attr. to copy
 ) ;
 ...
}
```

从"char"指针类型到"String"的映射是明显的。我们映射本地"Win32"类型"long(DWORD)"到在"Java"编程中的"int"。"Win32"类型"HANDLE",一个不透明的"32-bit"指针类型，也被映射为"int"。

因为在内存中怎样安排成员域的潜在的不同，我们不能映射"C"结构到在"Java"编程语言中的类。作为替代，我们使用一个数组来存储"C"结构的"SECURITY_ATTIBUTES"的内容。调用者也可以传递 NULL 作为"seAttrs"来指定默认的"Win32"安全属性。我们将不讨论"SECURITY_ATTRIBUTES"结构的内容或者怎样在一个"int"数组中编码。

一个"C++"上面本地方法的实现如下:

```
JNIEXPORT jint JNICALL Java_Win32_CreateFile(
 JNIEnv *env,
 jclass cls,
 jstring fileNmae,   // file name
 jint desiredAccess,  // access (read-write) mode
 jint shareMode,   // share mode
 jintArray secAttrs,  // security attributes
 jint createDistribution, // how to create
 jint flagsAndAttributes, // file attrbutes
 jint templateFile)  // file with attr. to copy
```

```
{
 jint result = 0 ;
 jint *cSecAttrs = NULL ;
 if (secAttrs){
  cSecAttrs = env->GetIntArrayElements(secAttrs, 0) ;
  if( cSecAttrs == NULL） {
   return 0 ;
  }
 }
 char *cFileName = JNU_GetStringNativeChars(env, fileName);
 if (cFileName) {

  result = (jint)CreateFile(cFileName,
     desiredAccess,
     shareMode,
     (SECURITY_ATTRIBUTES *)cSecAttrs,
     creationDistribution,
     flagsAndAttributes,
     (HANDLE)templateFile);
  free(cFileName);
 }

 if (secAttrs) {
  env->ReleaseIntArrayElements(secAttrs, cSecAttrs, 0);
 }
 return result;
}
```

首先，我们转化存储在"int"数组中的安全属性到一个"jint"数组。如果"setAttrs"参数是一个"NULL"引用，我们传递一个"NULL"作为安全属性到"Win32 CreateFile"函数。下一步，我们调用工具函数"JNU_GetStringNativeChars"(8.2 节)来获得文件名，使用本地描述的"C"字符串。一旦我们已经转换了安全属性和文件名，我们传递转换的结果和剩余参数给"Win32 CreateFile"函数。

我们关心异常的检查和虚拟器资源的释放(例如 cSetAttrs)。

"C.atol"和"Win32.CreateFile"的例子示范了一个封装类合格本地方法的一般方法.每个本地函数(例如，"CreateFile")映射一个单独的本地存根(stub)函数(例如，"Java_Win32_CreateFile"),它再依次地映射到一个单独的本地的方法定义（例如，Win32.CreateFile）。在一对一映射，存根的(stub)函数有两个目的(serve two purpose)：
1.存根函数使本地函数的参数传递的协定适合"Java"虚拟器的期望。虚拟器期望本地方法实现为一个被给地命名协定和接受俩个额外的参数("JNIEnv"指针和"this"指针)。

2.存根函数在"Java"编程语言类型和本地类型之间转换。例如，"Java_Win32_CreateFile"函数转换"jstring"文件名为一个本地描述的"C"字符串。

9.2 共享存根(Share stubs)

一对一的映射方法(approach)需要你来写一个存根函数为你想要打包的没个本地函数。当你面对(be faced with)为大量本地函数写分装的类的任务时，这变的冗长乏味。在这章，我们介绍共享存根的内容和示范共享存根可以怎样被用来简化(simplify）写封装类的任务。

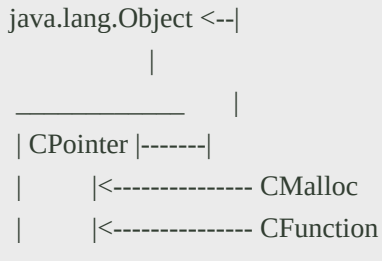
一个共享存根是一个本地函数，这函数分配给其他本地函数。共享存根负责转换来自调用者提供的参数类型到本地函数能够接受的参数类型。

我们将马上介绍一个共享存根类"CFunction",但首先让我们显示它能怎样简化"C.atol"方法的实现:

```
public class C{
 private static CFunction c_atol =
  new CFunction("msvcrt.dll",  // native library name
    "atol",   // C function name
    "C") ;   // calling convention
 public static int atol(string str){
  return c_atol.callInt(new Object[]{str}) ;
 }
 ...
}
```

"C.atol"不在是一个本地方法(因而不在需要一个存根函数)。相反，使用"CFunction"类来定义"C.atol"函数。"CFunction"类内部实现一个共享存根。静态变量"C.c_atol"存储一个"CFunction"对象,对象对应在"msvcrt.dll"库中的"C"函数"atol"(在"Win32"上的多线程"C"库)。"CFunction"构造器调用也指定"atol"追寻"C"调用的协定(11.4 部分)。一旦"c_atol"域被初始化，"C.atol"方法的调用只需要通过"c_atol.callInt",共享存根，来重新发送。

"CFunction"类属于一个类继承，我们将建立和简单地使用:



"CFunction"类的实例表示一个"C"函数的一个指针。"CFunction"是一个 CPointer 的子类，"CPointer"表示任意的"C"指针:

```
public class CFunction extends CPointer{
 public CFunction(String lib,  // native library name
   String fname,  // C function name
   String conv){  // calling convention
  ...
 }
 public native int callInt(Object[] args) ;
 ...
}
```

"callInt"方法是用一个"java.lang.Object"数组作为它的参数。它检查(inspect)在数组中的元素类型，转换它们(例如，从"jstring"到"char *"),同时作为参数传递它们给底层的"C"函数。然后，"callInt"方法返回底层"C"函数的结果作为一个"int"。"CFunction"类可能定义了方法例如"callFloat or callDouble"来处理带有其他返回类型的"C"函数。

"CPointer"类定义如下:

```
public abstract class CPointer{
 public native void copyIn(
   int bOff,  // offset from a C pointer
   int[] buf,  // source data
   int off,  // offset into source
   int len);  // number of elements to be copied
 public native void copyOut(...) ;
 ...
}
```

"CPointer"是一个抽象的类支持"C pointers"的任意的访问。例如，"copyIn"的方法，复制大量元素从一个"int"数组到被"C pointer"指向的位置。这个方法应该小心使用（be used with care),因为它能很容易被用来破坏(corrupt)在地址空间中的任意内存位置。本地方法例如"CPointer.copyIn"和在"C"中直接的指针处理(direct pointer manipulation)一样不安全。

"CMalloc"是"CPointer"的一个子类，"CMalloc"指向一个用"malloc"在"C"堆上分配(allocated in the C heap using malloc)的一个内存块：

```
pubic class CMalloc extends CPointer{
 public CMalloc(int size) throws OutOfMemoryError{...}
 public native void free() ;
 ...
}
```

"CMalloc"构造器在"C"堆上分配给定大小的内存块。"CMalloc.free"方法释放这个内存块。

装备(Equipped with)了"CFunction and CMalloc classes"，我们能在实现"Win32.CreateFile"如下:

```
public class Win32{
 private static CFunction c_CreateFile =
  new CFunction("kernel32.dll",  // native library name
    "CreateFileA",  // native function
    "JNI") ;   // calling convention

 public static int CreateFile(
  String filename,   // file name
  int desiredAccess,  // access (read-write) mode
  int shareMode,   // share mode
  int[] secAttrs,   // security attributes
  int creationDistribution, // how to create
  int flagAndAttributes,  // file attributes
  int templateFile)   // file with attr. to copy
 {
  CMalloc cSecAttrs = null ;

  if( secAttrs ! = NULL ){
   cSecAttrs = new CMalloc(secAttrs.length*4) ;
   cSecAttrs.copyIn(0, secAttrs, 0, secAttrs.length) ;
  }
  try{
   return c_CreateFile.callInt(new Object[]{
      filename,
      new Integer(desiredAccess),
      new Integer(shareMode),
      cSecAttrs,
      new Integer(creationDistribution),
      new Integer(flagAndAttributes),
      new Integer(templateFile)} ) ;
  }
  finally{
   if( secAttrs != NULL ){
    cSecAttrs.free() ;
   }
  }
 }
 ...
}
```

在一个静态变量中我们缓冲了"CFunction"对象。"Win32 API CreateFile"被作为"CreateFileA"从"kernel32.dll"库中导出。另一个导出入口"CreateFileW"，使用一个"Unicode"编码的字符串作为文件名参数。这个函数遵守了"JNI"调用协定，这是标准的"Win32"调用的协定(stdcall)。

"Win32.CreateFile"实现首先在"C"堆上分配一个内存块，这块足够大能暂时保存安全属性值。然后，封包所有参数到一个数组中和通过共享分发器来调用底层的"C"函数"CreateFileA"。最后"Win32.CreateFile"方法释放被用来保存安全属性的"C"内存块。我们在最后一行中调用"cSecAttrs.free"来保证临时使用的"C"内存被释放,即使"c_CreateFile.callInt"调用产生一个异常。

9.3 一对一映射与共享存根相对(One-to-One Mapping versus Shared Stubs)

一对一的映射和共享存根是两种为本地库建立封装类的方法。每个方法有它自己优点。

共享存根的主要优点是程序员不需要写在本地代码中写大量的存根函数。一旦一个共享存根实现例如"CFunction"是有用的，程序员可以不用写本地代码的一个单独行来建立封装类。

然而(however)，共享存根必须小心使用。使用共享存根，程序员实质地是在用"Java"编程语言来写"C"代码。这破坏(defeat)了"Java"编程语言的类型安全。在使用共享存根中错误可能导致破坏内存(corrupted memory)和应用程序的奔溃。

一对一映射的优点是典型地在转换数据类型中更有效率，数据是在"Java"虚拟器和本地代码(native code)之间传递。另一方面，共享存根(Shared stubs),能处理最多(at most)预定的(predetermined)一组参数类型，同时不能实现(achieve)最佳的性能(optimal performance)为这些参数类型。"CFunction.callInt"的调用者总是必须创建一个"Ineger"对象来为每个"int"参数。对于共享存根设计(scheme),这增加了空间和时间开销。

实际，你需要平衡性能，可移植性(portability)和短期生产率(short-term productivity）。共享存根可能适合利用固有地不可移植的本地代码，本地代码能忍受(tolerate)一点稍微的性能的下降(degradation),然而一对一映射应该在最佳性能是必须或可移植性问题的地方被使用。

9.4 共享存根的实现(Implementation of Shared Stubs)

到目前为止(so far),我们已经对待"CFunction, CPointer, and CMalloc"类为黑盒子。这章描述怎样使用基础的"JNI"特性来实现它们。

9.4.1 CPointer 类(The CPointer Class)

我们首先看看"CPointer"类，因为它是"CFunction and CMalloc"的父类(superclass)。"CPointer"的抽象类包含一个"64-bit"成员域，"peer",存储底层的"C"指针:

```
public abstract class CPointer{
 protected long peer;
 public native void copyIn(int bOff, int[] buf, int off, int len) ;
 public native void copyOut(...) ;
 ...
}
```

本地方法例如"copyIn"的"C++"实现是简单的:

```
JNIEXPORT void JNICALL
Java_CPointer_copyIn_I_3III(JNIEnv *env, jobject self,
    jint boff, jintArray arr, jint off, jint len)
{
 long peer = env->GetLongField(self, FID_CPointer_peer) ;
 env->GetIntArrayRegion（arr, off, len, (jint *）peer+boff) ;
}
```

"FID_CPointer_peer"是预算的成员域 ID(precomputed field ID)为"CPointer.peer"。这个本地方法的实现使用长名字编码设计来解决和重载的"copyIn"本地方法的实现的冲突，为在"CPointer"类中其他数组类型。

9.4.2 CMalloc 类(The CMalloc Class)

"CMalloc"类增加两个本地方法，方法被用来分配和释放"C"的内存块:

```
public class CMalloc extends CPointer{
 private static native long malloc(int size) ;
 public CMalloc(int size）throws OutOfMemoryError{
  peer = Malloc(size) ;
  if (peer == 0){
   throw new OutOfMemoryError();
  }
 }
 public native void free() ;
 ...
}
```

"CMalloc"构造器调用一个本地方法(native methhod)"CMalloc.malloc",同时如果"CMalloc.malloc"对返回一个最新在"C"堆上分配内存块失败，抛出一个"OutOfMemoryError"的错误。我们能够实现"CMalloc.malloc"和"CMalloc.free"方法如下:

```
JNIEXPORT jlong JNICALL
Java_CMalloc_malloc(JNIEnv *env, jclass cls, jintsize)
{
 return （jlong)malloc(size) ;
}
JNIEXPORT void JNICALL
Java_CMalloc_free(JNIEnv *env, jclass self)
{
 long  peer = env->GetLognField(self, FID_CPointer_peer) ;
 free((void *)peer) ;
}
```

9.4.3 CFunction 类(The CFunction Class)

"CFunction"类的实现需要使用在操作系统中的动态链接支持和 CPU 特定的汇编代码。下面呈现的实现是特别面向"Win32/Intel x86"环境。一旦你理解在实现"CFunction"类后面的规则，你能按一样的步骤在其他平台上实现它。

"CFunction"类被定义如下:

```
pubic class CFunction extends CPointer{
 private static final int CONV_C = 0 ;
 private static final int CONV_JNI = 1 ;
 private int conv ;
 private native long find(String lib, String fname) ;

 public CFunction(String lib,    // native library name
   String fname,  // C function name
   String conv){  // Calling convention
  if( conv.equals("C"））{
   conv = CONV_C ;
  }else if (conv.equals("JNI")){
   conv = CONV_JNI ;
  }else{
   throw new IllegalArgumentException("bad calling convention") ;
  }
  peer = find(lib, fname) ;
 }
 pubic native int callInt(Object[] args) ;
 ...
}
```

"CFunction"类声明一个私有成员域"conv",被用来存储"C"函数调用的协定。"CFunction.find"

本地方法实现(be implemeneted)如下(as follows)：

```
JNIEXPORT jlong JNICALL
Java_CFunction_find(JNIEnv *env, jobject self, jstring lib, jstring fun)
{
 void *handle ;
 void *func ;
 char *libname ;
 char *funname ;

 if( (libname = JNU_GetStringNativeChars(env, lib)) ){
  if( (funname = JNU_GetStringNativeChars(env, fun)) ){
   if ( (handle = LoadLibrary(libname)) ){
    if ( !(func = GetProcAddress(handle, funname)) ){
     JNU_ThrowByName(env, "java/lang/UnsatisfiedLinkError", funname) ;
    }
   }else{
    JNU_ThrowByName(env, "java/lang/UnsatisfiedLinkError", libname) ;
   }
   free(funname) ;
  }
  free(libname) ;
 }
 return (jlong)func ;
}
```

"CFunciton.find"转换库名哈函数名到局部指定的"C"字符串，然后调用"Win32 API"函数"LoadLibrary"和"GetProcAddress"来定位"C"函数在命名的本地库中。

"callInt"方法，实现如下，进行了重分配到底层"C"函数的主要任务:

```
JNIEXPORT jint JNICALL
Java_CFunction_callInt(JNIEnv *env, jobject self,
    jobjectArray arr)
{
#define MAX_NARGS 32
 jint ires ;
 int nargs, nwords ;
 jboolean is_string[MAX_NARGS] ;
 word_t args[MAX_NARGS] ;
 nargs = env->GetArrayLength(arr) ;
 if( nargs > MAX_NARGS ){
  JNU_ThrowByName(env,
    "java/lang/IllegalArgumentException",
    "too many arguments") ;
```

```
 return 0 ;
}
// convert arguments
for ( nwords = 0 ; nwords < nargs ; nwords++ ){
 is_string[nwords] = JNI_FALSE ;
 jobject arg = env->GetObjectArrayElement(arr, nwords) ;
 if ( arg == NULL ){
  args[nwords].p = NULL ;
 }else if ( env->isInstatnceof(arg, Class_Integer)){
  args[nwords].i =
   env->GetIntField(arg, FID_Interger_value) ;
 }else if ( env->IsInstanceOf(arg, Class_Float)){
  args[nwords].f =
   env->GetFloatField(arg, FID_Float_value) ;
 }else if ( env->IsInstanceOf(arg, Class_CPointer) ){
  args[nwords].p = (void *)
   env->GetLongField(arg, FID_CPointer_peer) ;
 }else if ( env->IsInstanceOf(arg, Class_String) ){
  char *cstr =
   JNU_GetStringNativeChars(env, (jstring)arg) ;
  if ( (args[nwords.p = cstr) == NULL) {
   goto cleanup ;   // error thrown
  }
  is_string[nwords] = JNI_TRUE ;
 }else{
  JNU_ThrowByName(env,
    "java/lang/IllegalArgumentException",
    "unrecognized argument type") ;
  goto cleanup ;
 }
 env->DeleteLocalRef(arg) ;
}
void *func =
 (void *)env->GetLongField(self, FID_CPointer_peer) ;
int conv = env->GetIntField(self, FID_CFunction_conv) ;
// now transfer control to func.
ires = asm_dispatch(func, nwords, args, conv) ;
cleanup:
 // free all the native string we have created
 for( int i = 0 ; i < nwords ; i++ ){
  if ( is_string[i] ){
   free(args[i].p) ;
  }
```

```
 }
 return ires ;
}
```

我们假设我们已经建立大量的全局变量为获得恰当的类引用和成员域"IDs"。例如，全局变量"FID_CPointer_peer"为了"CPointer.peer"而获得成员域"ID"，和全局变量"Class_String"是一个"java.lang.String"类对象的一个全局引用。"word_t"类型表示一个机器字(a machine word），定义如下:

```
typedef union{
 jint i ;
 jfloat f ;
 void *p
}word_t ;
```

"Java_CFunction_callInt"函数遍历(iterate through)参数数组,同时检查了每个参数类型:
.如果元素是一个"NULL"引用，他是作为一个"NULL"指针传递给"C"函数。
.如果元素是"java.lang.Integer"类的实例，"integer"值被得到和传递给"C"函数。
.如果元素是"java.lang.Float"类的实例，"floating-point"值被得到和传递给"C"函数。
.如果参数是"java.lang.String"类的实例，转换它为一个本地指定的(locale-specific)"C"字符串和传递它给"C"函数。
.另外(otherwise)，一个"IllegalArgumentException"异常被抛出。

我们在参数转换期间细心地检查可能的错误，同时在从"Java_CFunction_callInt"函数返回前，释放全部的为"C"字符串分配的临时存储。

从临时缓冲器"args"到"C"函数的传递参数的代码需要直接地处理 C 的堆栈。用内部的汇编来写:

```
int asm_dispatch( void *func,  // pointer to the C function
   int nwords,  // number of words in args array
   word_t *args, // start of the argument data
   int conv)  // call convention 0: C
      //                1: JNI
{
 __asm{
  mov esi, args
  mov edx nwords
  // Word address -> byte address
  shl edx, 2
  sub edx, 4
  jc args_done
 // push the last argument first
 args_loop:
```

```
 mov eax, DWORD PTR[esi+edx]
 push eax
 sub edx, 4
 jge SHORT args_loop
args_done:
 call func
 // check for calling converntion
 mov edx, conv
 or edx, edx
 jnz jni_call
 // pop the arguments
 mov edx, nwords
 shl edx, 2
 add esp, edx
jni_call:
 // done, return value in eax
 }
}
```

这个汇编函数复制参数到"C"的堆栈，然后，重分配到"C"函数"func".在"func"返回后，"asm_dispatch"函数检查"func"的调用协定(convention)。如果"func"允许"C"调用协定，"asm_dispatch"弹出传递给"func"的参数。如果"func"允许"JNI"调用协定，"asm_dispatch"不弹出参数；"func"在"asm_dispatch"返回前弹出参数。

9.5 Peer 类(Peer Classes)

一对一的映射和共享存根两都解决了封装本地函数的问题。在构建共享存根实现的期间，我们也遇到(encounter)封装数据结构体的问题。重看"CPointer"类的定义:

```
public abstract class CPointer{
 protected long peer;
 public native void copyIn(int boff, int[] buf,
    int off, int leng) ;
 public native void copyOut(...) ;
 ...
}
```

它包含一个"64-bit"的"peer"成员域，它参考本地数据结构(在这个情况中，在"C"地址空间中的一块内存)。"CPointer"的子类赋予这个"peer"成员域指定的意思(specific meanings)。"CMalloc"类，例如，使用"peer"成员域来指向在"C"堆上的一块(chunck)内存:

```
  |   peer  --|--------------------> |   memory in the C heap  |
```

An instance of the
CMalloc class

直接对应到本地数据结构的类，例如"CPointer and CMalloc"，被称为"peer classes"。你能构建"peer classes"为各种不同的本地数据结构，包括(including)，例如(for example):
.文件描述符(file descriptors)
.埠描述符(socket descriptors）
.窗口或其他图形用户接口控件(windows or other graphics user interface components)

9.5.1 在 Java 平台中的 Peer classes(Peer Classes in the Java Platform)

当前"JDK"和"Java 2 SDK release“内部使用"peer classes"来实现了"java.io，java.net and java.awt"包。例如，"java.io.FileDescroptor"类的实例，包含一个私有的成员域"fd",它代表一个本地文件描述符:

```
// Implemention of the java.io.FileDescriptor class
public final class FileDescriptor{
 private int fd ;
 ...
}
```

假设你想执行一个"Java"平台"API"不支持的文件操作。你可以试探(tempt)使用"JNI"来找出一个"java.io.FileDescriptor"实例的底层本地文件描述符。一旦你知道它的名字和类型，"JNI"允许你访问一个私有成员域。然后，你可以认为你能够直接地在那个文件操作符上执行本地文件操作。然而，这个方法有两个问题:
.首先，你依赖于一个存储了本地文件描述符在一个私有成员域"fd"中的"java.io.FileDescriptor"实现。然而，不保证，来自"Sun"的将来的实现或第三方的"java.io.FileDescriptor"类的实现将仍然使用同样的使用成员域名"fd"为本地文件描述符。假设"peer"成员域的名字的本地代码可能在一个不同实现的"Java"平台上不能工作。
.第二，你在本地文件描述上的直接地执行的操作可能破坏(disrupt)"peer class"的内部一致性(internal consistency)。例如,"java.io.FileDescriptor"实例维持一个内部状态指示，是否底层本地文件描述已经被关闭。如果你使用本地代码来迂回跳过(bypass）"peer class"，同时关闭了底层的文件描述符，在"java.io.FileDescriptor"实例中维持的状体将不再(no longer)和本地文件描述符的真实状态一致了(be consistent with)。"Peer class"实现假设他们有独家访问底层的本地数据结构。

客服这些问题的唯一方法是定义你自己的"peer classed"来封装本地数据结构。在上面例子中，你能定义你自己文件描述符"peer class"来支持操作请求集(the required set of operations)。这个方法不是让你用你自己的"peer classes"来实现"Java API classes".例如，你不能传递你自己文件描述符实例到期望一个"java.io.FileDescriptor"实例的方法。然而，你能容易定义你自己

"peer class"来实现在"Java API"上的一个标准接口.这有一个强壮的参数为设计"APIs"基于接口，而不是类(classes）。

9.5.2 释放本地数据结构

"Peer classes"用 Java 编程语言定义;因此"peer classes"实例自动被垃圾收集(garbage collected)。然而，你需要保证，底层的本地数据结构也将被释放。

回想"CMalloc"类包含一个"free"方法来明确地释放分配(malloc'ed)的"C"内存:

```
public class CMalloc extends CPointer{
 public native void free() ;
 ...
}
```

你必须记住调用"free"在"CMalloc"类的实例上；否则一个"CMalloc"实例可以被垃圾收集，但它对应的"malloc'ed"的"C"内存将不被收回(reclaim)。

一些程序员喜欢在"peer classes"中放置一个终止函数(finalizer)例如"CMalloc":

```
public class CMalloc extends CPointer{
 public native synchronized void free() ;
 protected void finalize(){
  free() ;
 }
 ...
}
```

在虚拟器垃圾收集一个"CMalloc"实例前，虚拟器调用"finalize"方法。即使你忘记调用"free"函数，"finalize"方法为你释放"malloc'ed"的"C"内存。

你需要做个小改变在"CMalloc.free"的本地方法的实现来计数(account for)它被多次(multiple times)调用的可能(possibility)。你也需要使"CMalloc.free"为一个同步的方法来避免线程竞争情况(thread race conditions):

```
JNIEXPORT void JNICALL
Java_CMalloc_free(JNIEnv *env, jobject self)
{
 long peer = env->GetLongField(self, FID_CPointer_peer) ;
 if( peer == 0 ){
  return ;
 }
```

```
 free( (void*)peer) ;
 peer = 0 ;
 env->SetLongField(self, FID_CPointer_peer, peer) ;
}
```

我们用两句来设置"peer"成员域:

```
peer = 0 ;
env->SetLongField(self, FID_CPointer_peer, peer) ;
```

替代一句:

```
env->SetLongField(self, FID_CPointer_peer, 0) ;
```

因为"C++"编译器将把文字"0"认为一个"32-bit"的整数(integer)，而不是(as opposed to)一个"64-bit"整数。一些"C++"编译器允许你来指定"64-bit"整数文字，但使用"64-bit"文字将不能移植。

定义一个"finalize"方法是一个适合的安全装置，但你不应该依赖"finalizers"作为释放本地数据结构的中心方法(sole means）。原因是本地数据结构可能消耗太多的资源比它们的"peer"实例。"Java"虚拟器可能不垃圾收集和完成对"peer classes"的实例的最快的释放他们本地副本。

定义一个终结器也有性能的影响(performance consequences)。典型地带有"finalizers"的类型的实例的创建和收回都要慢于不带"finalizers"的类型的实例的创建和收回。

如果你也能确保你人工地(manually)释放本地数据结果为"peer classes"，你不必定义一个终结器(finalizer)。然而，你应该确保在所有的执行路径中(in all paths of execution)都释放本地数据结构；否则你可能已经创建了一个资源泄漏。在使用一个"peer"实例的处理期间，特别注意可能的异常抛出。在"finally"语句中，总是释放本地数据结构:

```
CMalloc cptr = new CMalloc(10) ;
try{
 ... // use cptr
}finally{
 cptr.free() ;
}
```

即使一个异常发生在"try"块中，"finally"语句确保"cptr"被释放。

9.5.3 Peer 实例的返回点(Backpointers to Peer Instances)

我们已经显示典型的"peer classes"包含一个参考底层本地数据结构的私有成员域。在一些例

子中，最好也包括一个参考来自本地数据结构的"peer class"的实例(from the native data structure to instances of the peer class)。例如，这发生在，当本地代码需要初始化在"peer class"中的实例方法的回调。

假设我们建立一个叫做"KeyInput"的假设(hypothetical)用户接口控件。当用户按了一个键时，"KeyInput"的本地"C++"控件，"key_input",接受一个事件，做来自操作系统的一个Key_pressed"的"C++"函数调用。"key_input"的"C++"控件通过调用在"KeyInput"实例上的"keyPressed"方法来报告操作系统事件到"KeyInput"实例。在下图中，箭头指示怎样通过一个用户按键产生一个按键事件和怎样从"key_input"的"C++"控件到"KeyInput peer"实例的传播(propagate):

```
| KeyInput(){ |<------|     | key_pressed(){ |<------- User key press
| ...         |       |-----| ...            |
| }           |             | }              |
KeyInput instance           key_input C++ component
```

"KeyInput peer class"被定义如下:

```
class KeyInput{
 private long peer ;
 private native long create() ;
 private native void destroy(long peer) ;

 public KeyInput(){
  peer = create() ;
 }
 pulbic destroy(){
  destroy(peer) ;
 }
 private void keyPressed(int key){
  ...
 }
}
```

"create"本地方法实现分配一个"C++"机构体"key_input"的实例。"C++"机构体和"C++"类相似，唯一的不同是所有的成员默认都是公有的而不是(as opposed to)私有的。在这个例子中，我们使用一个"C++"机构体替代一个"C++"类主要地为避免和在"Java"编程语言中类产生混淆(avoid confusion with)。

```
// C++ structure, native counterpart of KeyInput
struct key_input{
 jobject back_ptr;   // back pointer to peer instance
 int key_pressed(int key) ; // called by the operating system
};
JNIEXPORT jlong JNICALL
Java_KeyInput_create(JNIEnv *env, jobject self)
```

```
{
 key_input *cpp_obj = new key_input() ;
 cpp_obj->back_ptr = env->NewGlobalRef(self) ;
 return (jlong)cpp_obj ;
}
JNIEXPORT void JNICALL
Java_KeyInput_destroy(JNIEnv *env, jobject self, jlong peer)
{
 key_input *cpp_obj = (key_input *)peer ;
 env->DeleteGlobalRef(cpp_obj->back_ptr) ;
 delete cpp_obj ;
 return ;
}
```

"create"本地方法分配"C++"结构体和初始化它的"back_ptr"成员域为"KeyInput peer"实例的全局引用。"destroy"本地方法删除了"peer"实例的全局引用和被"peer"实例引用的"C++"的结构体。"KeyInput"构建函数调用了"create"本地方法来建立在一个"peer"实例和她的本地副本之间的连接:

```
|     |<------------------------|         |
| peer | JNI global reference    | back_ptr |
|     |                         |         |
|     |------------------------>|         |
|     |   64-bit long           |         |
KeyInput instance               C++ key_input structure
```

当用户按一个键，操作系统调用"C++"成员函数"key_input::key_pressed"。这个成员函数通过调用在"KeyInput peer"实例上的"keyPressed"方法的调用来响应事件。

```
// return 0 on success, -1 on failure
int key_input::key_pressed(int key)
{
 jboolean has_exception ;
 JNIEnv *env = JNU_GetEnv() ;
 JNU_CallmethodByName(env,
    &has_exception,
    java_peer,
    "keyPressed"
    "()V",
    key) ;
 if ( has_exception){
  env->ExceptionClear() ;
  return -1 ;
 }else{
```

```
 return 0 ;
 }
}
```

在回调后，"key_press"成员函数清除所有异常，同时使用-1 放回代码来返回错误情况到操作系统。引用了在 6.2.3 和 8.4.1 部分中分别地(respectively)"JNU_CallMethodByName and JNU_GetEnv"的工具函数的定义。

在结束(concluding)这部分前，让我们讨论最后一个问题。假设(Suppose)你添加一个"finalize"方法在"KeyInput"类中为避免潜在的内存泄漏(potential memory leaks)：

```
class KeyInput{
 ...
 public synchronize destroy(){
  if ( peer != 0 ){
   destroy(peer) ;
   peer = 0 ;
  }
 }
 protect void finalize(){
  destory() ;
 }
}
```

"destroy"方法检查是否"peer"成员域是 0，同时在调用重载的"destory"本地方法后，设置"peer"成员域为 0。"destroy"被定义为一个同步方法(synchronized method)来避免竞争情况(race conditions)。

然而，以上代码将不能像你期望的工作。虚拟器将不能垃圾收集任何 KeyInput 实例，除非你明确地调用"destroy"。"KeyInput"构造函数创建一个"JNI"全局的"KeyInput"实例的引用。全局引用阻止了"KeyInput"实例被垃圾收集。你能通过使用一个弱全局应用替代一个全局应用来克服这个问题。

```
JNIEXPORT jlong JNICALL
Java_KeyInput_create(JNIEnv *env, jobject self)
{
 key_input *cpp_obj = new key_input() ;
 cpp_obj->back_ptr = env->NewWeakGlobalRef(self) ;
 return (jlong)cpp_obj ;
}
JNIEXPORT void JNICALL
Java_KeyInput_destroy(JNIEnv *env, jobject self, jlong peer)
{
 key_input *cpp_obj = (key_input *)peer ;
```

```
env->DeleteWeakGlobalRef(cpp_obj->back_ptr) ;
delete cpp_obj ;
return ;
}
```

第十章 陷阱和缺陷(Traps and Pitfalls)

为了突出在前面章节涉及的重要技术，本章涉及 JNI 程序员通常所犯的大量错误。这儿描述的每个错误都已在真实世界的工程中发生。

10.1 错误检查(Error Checking)

当写本地方法时，最通常的错误是忘记检查是否一个错误情况已经发生。不象"Java"编程语言，本地语言不会提供标准的异常机制。"JNI"不能依赖任何特殊的本地异常机制(例如"C++"异常)。因此，在每个可能产生一个异常的"JNI"函数调用后，编程者被要求执行清楚地检查。并非所有的"JNI"函数都引发异常，但大多数可能。异常检查是繁琐(tedious)，但必须确保使用本地方法的应用程序健壮(robust)。

错误检查的繁琐，更强调需要限制本地代码到那些需要使用"JNI"的应用程序的良好定义的子集中。

10.2 传送无效参数给"JNI"函数(Passing Invalid Arguments to JNI Functions)

"JNI"函数不会尝试检查(detect)和恢复(recorer from)无效参数。如果你传递"NULL"或(jobject)"oxFFFFFFFF"给一个希望得到一个引用的"JNI"函数，结果行为是不可预期的(be undefined)。实际上，这可能导致不正确的结果或虚拟器的奔溃。"Java 2 SDK release 1.2"提供你一个命令行选项"-Xcheck:jni"(provide you with)。这选项命令虚拟器来侦测和报告许多传递无效逻辑参数给"JNI"函数的本地代码的情况，但不是全部。对于参数有效性的检查招致大量的开销，因此默认是不开启的。

在"C and C++"库中，不检查参数的有效性是通常实际情况。使用库的代码负责(be responsible for)确保所有传递给库函数的参数是有效的。然而,如果你习惯"Java"编程语言，你可能必须在"JNI"编程中，调整以适合安全缺失的这特定方面。

10.3 混淆"jclass"和"jobject"(Confusing jclass withc jobject)

当第一次使用"JNI",实例的引用("jobject"类型的值)和类引用("jclass"类型的值）之间不同可能是混淆的。

实例的引用对应"java.lang.Object"或它的子类的一种的实例和数组。类引用对应"java.lang.Class"实例，代表类的类型(class types)。

一个操作例如"GetFieldID"是一个类操作，它得到的是一个"jclass"，因为它得到来自一个类的成员域的描述符。相反(In constrast)，"GetIntField"是一个实例操作，得到一个"jobject"，因为它得到来自一个实例的一个成员域的值。"jobject"和实例操作相关和"jclass"和类型操作相关是一致贯彻整个"JNI"函数的，因此很容易记住类型操作和实例操作的区别(be distinct from)。

10.4 截短"jboolean"参数(Truncating jboolean Argumnets)

一个"jboolean"是一个"8-bit"无符号"C"类型，它能存储 0 到 255 的值。0 值对应常数"JNI_FALSE",1 到 255 的值对应"JNI_TRUE"。但"32-bit or 16-bit"大于 255 的值，它的低"8 bits"是 0 时，造成(pose)一个问题。

假设你已经定义一个函数"print",有一个为"jboolean"类型的参数"condition":

```
void print(jboolean condition)
{

 if ( condition ){
  printf("true\n") ;
 }else {
  printf("false\n") ;
 }
}
```

前面的定义没有任何错误。然而，下面看是无害的(innocent-looking)"print"调用将产生一个稍微(somewhat)的不期望的结果:

```
int n = 256 ;
print(n) ;
```

我们传递一个非零值(256)到"print"，期望它代表"true"。但因为超过低 8 位的所有"bits"被截去，参数是 0。程序打印"false",不是期望的(contrary to expectations)。

当强制(coerce)整数类型(integral types),例如 int,为"jboolean"类型时，"thumb"的一个好规则是总是在整数类型上评估条件，因此(thereby）避免不留意的错误(inadvertent errors)在强制期间。你能重写"print"的调用如下(as follow):

```
n = 256 ;
print（n? JNI_TRUE: JNI_FALSE) ;
```

10.5 "Java"应用和本地代码之间的边界线(Boundaries between Java Application and Native Code)

当设计一个被本地代码支持的"Java"应用程序时，一个通常的问题是"在本地代码中是什么和怎样做"。本地代码和用"Java"编程语言写的应用程序的剩余部分之间的边界是应用的特殊部分，但这有些一般地适应的规则(applicable principles):

.保持边界简单(simple)。在"Java"虚拟器和本地代码间的来回的(go back and forth)复杂控制流可能调试和维护困难。如此控制流也是阻碍了(get in the way of)通过高性能虚拟器实现来实现优化。例如，对于虚拟器的实现，在"Java"编程语言中定义内联方法比在"C and C++"中定义内联的本地方法(inline native methods)更容易。

.在本地代码方保持代码尽力少(minimal)。这儿有如此做(do so)的强制原因(compelling reason)。本地代码不能移植和类型不安全。在本地代码中错误检查是麻烦的(10.1 部分)。好的软件工程保持这样的部分为最小。

.保持本地带啊摹独立(isolated)。实际上，这可能意味着所有本地方法是在一样的包里或一样的类中，独立于应用程序其他部分。包含必须的本地方法的这个包或类对于应用程序编程端口层("porting layer")。

"JNI"提供访问虚拟器的功能(functionality)例如类的载入(class loading)，对象创建(object creation)，成员域访问(field access),方法调用(method calls),线程同步(thread synchronization)，等等(and so forth)。当事实上用"Java"编程语言更简单的完成同样的任务时，有时在本地代码中，尝试(tempt)用"Java"虚拟器的功能表达复杂的交互(express complex inerations)。下面的例子显示使用本地代码的"Java"编程为什么是不好做法(bad practice）。认识一个简单声明，它用"Java"程序语言创建一个新的线程:

```
new JobThread().start();
```

同样的声明也能使用本地代码来写:

```
aThreadObject =  （*env)->NewObject(env, Class_JobThread, MID_Thread_init) ;
if(aThreadObject == NULL){
 ...
}
(*env)->CallVoidMethod(env, aThreadObject, MID_Thread_start) ;
if( (*env)->ExceptionOccurred(env) ){
 ...
}
```

尽管实际上我们忽略了为错误检查需要的代码行，但本地代码比用"Java"编程语言写的同样实现复杂还是太多了。

通常可取的(preferable)是用"Java"编程语言来定义有一个辅助的(auxiliary)方法，同时用本地代码调用这个辅助方法的一个回调，而不是写个本地代码操作"Java"虚拟器的一个复杂片段。

10.6 混淆 IDs 和引用(Confusing IDs with References)

"JNI"揭示对象(objects)为引用。类，字符串和数组(as references, Classes, strings, and arrays)是引用的特别类型。"JNI"揭示方法和成员域为 IDs。一个 ID 不是一个引用。不能认为一个类的引用(class reference)是一个"Class ID",或者一个方法 ID 是一个"方法应用(method reference)"。

引用是被本地代码明确管理的虚拟器资源。例如，"JNI"函数"DeleteLocalRef"，允许本地代码来删除一个局部引用。相反(in contrast)，成员域和方法"IDs"被虚拟器管理，和保持有效直到他们定义的类被载出。在虚拟器载出定义的类前，本地代码不能明确地删除一个成员域或方法 ID。

本地代码可以创建多个引用，来查阅同一个对象(object）。例如，一个全局和一个局部引用，可以查阅同一个对象。相反，一个唯一成员域或方法"ID"被导出(be derived)为一个成员域或一个方法的定义。如果"class A"定义方法"f"，同时"class B"从"A"继承(inherit)"f"，在下面代码中，两个"GetMethodID"调用总是返回一样的结果:

```
jmethodID MID_A_f =  （*env)->GetMethodID(env, A, "f", "()V") ;
jmethodID MID_B_f =  （*env)->GetMethodID(env, B, "f", "()V") ;
```

10.7 缓冲成员域和方法 IDs(Caching Field and Method IDs)

本地代码通过指定的成员域或方法的名字和类型描述符作为字符串，从虚拟器得到成员域或方法"IDs"(4.1 部分,4.2 部分)。成员域和方法查看使用的名字和类型字符串是很慢的。通常为解决这问题(pay off)来缓冲"IDs"。在本地代码中，缓冲成员域和方法 ID 的失败是一种常见的性能问题。

在一些例子中，缓冲"IDs"比一个性能提高会更多。缓冲一个"ID"可能必须确保真确的成员域或方法被本地代码访问。下面例子说明缓冲一个成员域 ID 的失败怎样导致一个狡猾的问题(subtle bug)：

```
class C {
 private int i ;
 native void f() ;
}
```

假设本地方法"f"需要获得在"C"类实例中的成员域"i"的值。没有缓冲一个 ID 的一个直接的(straightforward)实现，用三步来完成这个:1)得到对象的类型(class)；2)为来自类型的引用的"i",查到成员域的 ID；3)访问俄成员域的值，基于对象的引用和成员域"ID"：

```
// No field IDs cached
```

```
JNIEXPORT void JNICALL
Java_C_f(JNIEnv *env, jobject this){
 jclass cls = (*env)->GetObjectClass(env, this) ;
 ...
 jfieldID fid = (*env)->GetFieldID (env, cls, "i", "I") ;
 ...
 ival = (*env)->GetIntField(env, this, fid) ;
 ...
}
```

这个代码运行的很好，直到我们定义另一个类"D"为"C"的资料(subclass),同时声明了一个私有成员域也是"i"名字在"D"类中:

```
// Trouble in the absence of ID caching
class D extends C{
 private int i ;
 D(){
  f() ; // inherited from C
 }
}
```

当"D"的构造器(D's constructor）叫做"C.f"，本地方法受到"D"的实例作为"this"参数， "cls"指的是"D class"，同时"fid"代表"D.i"。在本地方法的最后，"ival"包含了"D.i"的值，替代了"C.i"。当实现本地方法"C.f"时候，这可能不是你期望的。

解决方法是计算和缓冲成员域"ID"，当你确定你有个"C"的类引用，而不是 D 的时。来自缓冲 ID 的后续访问(subsequent access)也将指向正确的成员域"C.i"。这儿正确的版本:

```
// Version that caches IDs in static initializers
class C {
 private int i ;
 native void f() ;
 private static native void initIDs() ;
 static{
  initIDs() ;// Call an initializing navtive method
 }
}
```

修改本地代码是(The modified native code is):

```
static jfieldID FID_C_i ;
JNIEXPORT void JNICALL
Java_C_initIDs(JNIEnv *env, jclass cls){
```

```
 FID_C_i = (*env)->GetFieldID(env, cls, "i", "I") ;
}
JNIEXPORT void JNICALL
Java_C_f(JNIEnv *env, jobject this){
 ival = (*env)->GetIntField(env, this, FID_C_i) ;
 ...
}
```

成员域 ID 是在"C"的静态初始化函数中被计算和缓冲的。这保证"C.i"的成员域 ID 将被缓冲，因此本地方法实现"Java_C_f"将取得"C.i"的值，独立于"this"对象的实际类型。

缓冲对于一些方法调用也是需要的。如果我们稍微地改变上面的例子，使每个类型"C"和"D"都有自己定义的私有方法"g"，"f"需要缓冲"C.g"的方法"ID"来避免意外(accidentally)调用"D.g"方法。为调用正确的虚拟方法，缓冲是不需要的。被定义的虚拟方法，动态绑定到方法调用的实例上。因此你能安全地使用"JNU_CallMehtodByName"工具函数(6.2.3 部分)来调用虚拟方法。然而，前面例子告诉我们，为什么我们不定义一个类似的"JNU_GetFieldByName"工具函数。

10.8 Unicode 字符串的结束(Terminating Unicode Strings)

从"GetStringChars or GetStringCritical"得到 Unicode 字符串是没有"NULL"结束的(NULL-terminated)。调用"GetStringLength"来发现"16-bit Unicode"字符个数在字符串中。一些系统操作，例如 Windows NT，期望两个拖尾的为 0 的 byte 值(two trailing 0 byte value)来结束"Unicode"字符串。你不能传递"GetStringChars"的结果到期望一个"Unicode"字符串的"Windows NT API"。你必须做字符串的另外复制和插入两个拖尾的为 0 的 byte 值。

10.9 违反访问控制规则(Violating Access Control Rules)

"JNI"不能强制"class, field, and method"访问控制限制，限制就是在"Java"编程语言层通过修饰符的使用例如"private"和"final"来表示的。写本地代码访问或修改一个对象的成员域是可能的，即使在"Java"编程语言层如此做(do so)将导致一个"IllagalAccessException"。"JNI"的放任(permissiveness)是有意的(conscious)设计决定，给了本地代码访问和修改任何在堆上的内存地址。

迂回过(bypass)源代码语言层访问检查的本地代码，可以在程序执行上有不受欢迎的(undesirable)影响。例如，一个矛盾(inconsistency)被创建，如果在运行时编译的(just-in-time(JIT))编译器内联访问这成员域后一个本地方法修改了一个"final"成员域。类似地(Similarly),本地方法应该不能修改不可变的(immutable)对象例如在"java.lang.String or java.lang.Integer"的实例中的成员域。如此做(就是修改了)可以导致在"Java"平台实现中，不变量的破损。

10.10 漠视国际化(Disregarding Internationalization）

在"Java"虚拟器中字符串包含"Unicode"字符，但是(whereas）本地字符串是典型的一个本地特定的编码。使用工具函数例如"JNU_NewStringNative"(8.2.1 部分)和"JNU_GetStringNativeChars"(8.2.2 部分)来在"Unicode jstrings"和底层主机环境的本地指定的本地字符串之间转化。特别注意(pay special attention)消息字符串和文件名字，他们是典型地国际化的(internationalized)。如果一个本地方法得到一个文件名字作为一个"jstring",在文件名传递给一个"C"库函数前，文件名字必须转化为本地字符串。

下面本地方法，"MyFile.open",打开一个文件和返回一个文件描述符为它的结果:

```
JNIEXPORT jint JNICALL
Java_MyFile_open(JNIEnv *env, jobject seld, jstring name, jint mode)
{
 jint result;
 char *cname = JNU_GetStringNativeChars(env, name) ;
 if( cname == NULL ){
  return 0 ;
 }
 result = open(cname, mode) ;
 free(cname) ;
 return result ;
}
```

我们使用"JNU_GetSringNativeChars"函数来转化"jstring"参数，因为"open"系统调用希望文件名字是本地指定编码的。

10.11 保留虚拟器资源(Retaining Virtual Machine Resources)

在本地方法中一个通常的错误是忘记释放虚拟器的资源。程序员在错误发生时执行的代码过程(in code paths)中，需要特别地(particularly)细心。下面代码片段，在 6.2.2 部分的一个例子的稍微地修改，错失一个"ReleaseSringChars"调用:

```
JNIEXPORT void JNICALL
Java_pkg_Cls_f(JNIEnv *env, jclass cls, jstring jstr)
{
 const jchar *cstr =
  (*env)->GetStringChars(env, jstr, NULL) ;
 if( cstr == NULL){
  return ;
 }
 ...
```

```
 if( ...){

  return ;
 }
 ...

  （*env)->ReleaseStringChars(env, jstr, cstr) ;
}
```

忘记调用"ReleaseStringChars"函数可能引起"jstirng"对象被无限期地(indefinitely)保留(be pinned)，导致内存碎片(fragmentation),或者"C"语言的副本被无限期地保留(be retained),一个内存泄漏。

无论"GetStringChars"得到一个字符串的副本有没有，这儿必须有个对应的"ReleaseStringChars"调用。下面代码没有正确的释放虚拟器的资源(fail to release virtual machine resources properly)：

```
JNIEXPORT void JNICALLL
Java_pkg_Cls_f(JNIEnv *env, jclass cls, jstring jstr)
{
 jboolean isCopy ;
 const jchar *cstr = (*env)->GetStringChars(env, jstr, &isCopy) ;

 if( cstr == NULL ){
  return ;
 }
 ...

 if( isCopy){
  (*env)->ReleaseStringChars(env, jstr, cstr) ;
 }
}
```

当"isCopy"是"JNI_FALSE"时，"ReleaseStringChars"的调用也是需要的，以便虚拟器释放"jstring elements"。

10.12 过度的局部引用创建(Excessive Local Reference Creation)

过度局部引用的创建导致程序不必要的保留内存。一个不必要的局部引用浪费为被引用对象的内存和为引用自己的内存。

特别注意长时间运行(long-running)本地方法,在循环和工具函数中创建得分局部引用。在"Java 2 SDK release 1.2"中利用新的"Push/PopLocalFrame"函数来管理局部引用更有效率。参考 5.2.1 和 5.2.2 部分为这个问题的更多的细节的讨论。

在"Java 2 SDK release 1.2"中你能指定"-verbaose: jni"选项来请求虚拟器来侦测和报告过度局部引用创建。假设你带有这个选项运行一个类"Foo"：

```
%java -verbose:jni Foo
```

同时输出包含下面:

```
***ALART: JNI local ref creation excedded capacity
    (creating: 17,, limit: 16).
 at Baz.g(Native method)
 at Bar.f(Compiled method)
 at Foo.main(Compiled method)
```

本地方法实现"Baz.g"不正确的管理局部引用是可能的。

10.13 使用无效的局部引用(Using Invalid Local Reference)

局部引用只在一个本地方法的单次调用(invocation)中是有效的。当实现方法的本地函数返回后，在一个本地方法调用中创建的局部引用自动被释放。本地代码不应该存储一个局部引用到一个全局变量中，同时期望在这个本地方法的以后调用(in later invocations of the native method)中使用它。

局部引用只在创建它们的线程中是有效的。你不应该传递一个局部引用从一个线程到另一线程。当必须在线程之间传递一个引用时，要创建一个全局引用。

10.14 在线程间使用"JNIEnv"（Using the JNIEnv across Threads)

"JNIEnv"指针，作为第一个参数传递给每一个本地方法，只能在和它关联的线程中被使用。从一个线程得到缓冲的"JNIEnv"接口指针，同时在另一个线程中使用这个指针，是错误的。8.1.4 部分解释你能怎样为当前线程得到"JNIEnv"接口指针。

10.15 不匹配的线程模式(Mismatched Thread Models）

"JNI"才运行，只有主机的本地代码和"Java"虚拟器实现共享一样线程的模式(8.1.5 部分)。例如，程序员不能附加本地平台线程到一个使用一个用户线程包实现的嵌入式 Java 虚拟器。

在"Solaris"上，"Sun"附带一个基于一个名为"Green threads"的用户线程包的虚拟器实现。如果你的本地代码依赖于"Solaris"本地线程支持，它将不能和一个基于"Green thread"(Green-thread-based)的 Java 虚拟器实现一起工作。你需要一个虚拟器实现，它被设计来和"Solaris"本地线程一起工作的。在"Solaris"上支持本地线程的"JDK release 1.1"需要单独下载。本地线程支持是捆绑到"Solaris Java 2 SDK release 1.2"。

"Sun"的 Win32 的虚拟器实现默认是支持本地线程的，同时能容易嵌入到本地 Win32 应用程序中。

第三部分：规范
(Part Three: Specification)

第十一章"JNI"设计概要
(Overview of the JNI Design)

这章给出了"JNI"设计的概要。如果有需要，我们还提供底层技术的动机(technical motivation)。设计概要作为(serve as)主要的"JNI"概念的规范，例如"JNIEnv"接口指针，局部和全局应用，和成员域和方法"IDs"。技术的动机旨在(aim at)帮助读者来理解各种设计的取舍(trade-offs). 在有些时候(On a few occasions)，我们将讨论怎样实现某个特征。这样的讨论的目的不是表现一个实际实现的策略，而是替代来(instead to）澄清(charify）微妙的语义(subtle semantic)问题。

一个桥接不同语言的编程接口的概念不是新的。例如，"C"程序典型地能调用使用例如"FORTRAN"和汇编语言编写的函数。相似地，编程语言例如"LISP"和"Smalltalk"的实现支持各种其他语言的接口。

"JNI"解决了一个问题，类似于(similar to)通过被其他语言支持的互操作机制来解决的问题。然而，在"JNI"和在许多其他语言中的互操作机制之间有一明显的不同。"JNI"不是为一个特别的"Java"虚拟器的实现设计的。而是(Rather)，一个本地接口能过被每个"Java"虚拟器的实现支持的。我们将进一步详细描述这个，作为我们描述"JNI"设计的目标。

11.1 设计目标(Design Goals)

"JNI"设计的最重要目标是确保它在一个给定的主机环境上不同的"Java"虚拟器实现中提供二进制兼容性。同样的本地库的二进制不需要在编译(without the need for recompileation)将运行在一个给定的主机环境上的不同的虚拟器实现上。

为了实现这个目标，"JNI"设计不能做任何关于"Java"虚拟器实现的内部细节的假设。因为"Java"虚拟器实现技术是快速发展的(evolve）,我们必须小心地避免引进任何限制(introducing any constraints), 限制可能干扰(interfere with）在未来先进的实现技术。

"JNI"设计的第二个目的是效率(efficiency)。为了支持时间紧要的代码，"JNI"要实加(impose)尽可能少开销(as little overhead as possible)。然而，我们将清楚我们首先目标，需要实现独立，有时需要我们采取(adopt)一个比我们否则可能的稍微地(slightly）低效的设计。我们在效率和实现的独立之间实现妥协(strike a compromise)。

最后，"JNI"必须功能完整(functionally complete)。必须导出足够"Java"虚拟器功能(functionality)来使本地方法和应用程序能完成有用的任务。

被一个给定的"Java"虚拟器实现支持的唯一的本地编程接口不是"JNI"的目标。标准接口有益于编程者载入他们的本地代码库到不同的"Java"虚拟器实现。然而，在一些例子中，一个较低层(lower-level)实现特定(implementation-specific)接口可以实现较高的性能。在另一些例子中，编程者可以使用一个较高层的(higher-level)接口来构建软件的控件。

11.2 载入本地库(Loading Native Libraries）

当一个应用能调用一个本地方法前，虚拟器必须定位和载入一个本地库，它包含本地方法的实现。

12.1.1 类的载入(Class Loaders)

本地库被类载入器(by class loaders)定位(be located)。类载入器在"Java"虚拟器中有许多使用，例如，包括载入类文件，定义类和接口，在软件控件中提供名字空间的分隔，解决在不同类和接口中的符号引用，和最终的定位本地库。我们假设你对类载入器有基本地理解，我们将不去详细描述怎样在"Java"虚拟器中载入和链接它们的类。你能够发现关于类载入器的更多细节在"Dynamic Class Loading in the Java Virtual Machine"论文中，是"Sheng Liang and Gilad Bracha"写的，发表(published)在 1998 年的关于面向对象的编程系统，语言和应用程序(Object Oriented Programming Systems, Languages, and Applications(OOPSLA)的 ACM 会议(Conference)的会议录中。

类载入器提供名字空间的隔离，被需要在一样虚拟器的一个实例中运行多个控件(例如载入来自不同 Web 站点的"applets")。通过映射类或接口名字到在"Java"虚拟器中被表示为对象的实际类或接口类型的过程，一个类载入器保持一个隔离名字空间。每个类或接口类型和它定义的载入器相关，载入器初始时读类文件和定义类或接口对象。只有当他们有一样的名字和一样的定义载入器时，两个类或接口类型是一样的。例如，在图 11.1 中，类载入器"L1"和"L2"每个定义了一个名字为"C"的类。这两个名为"C"的类是不一样的。事实，他们包含两个不同的"f"方法，方法有不同(distinct)返回类型。

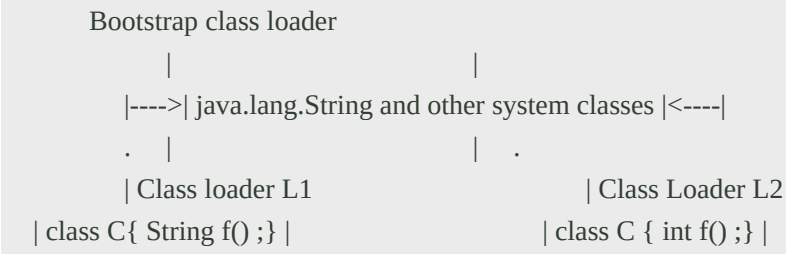


Figure 11.1 Two classes of the same name loaded by different class loaders

在上图中点线代表在类载入器之间的委托惯性(delegation）。一个类载入器可能请求另一个类载入器来载入它代表的一个类或接口。例如，"L1"和"L2"代表(delegate）为系统类"java.lang.String"的引导类载入器(the bootstrap system classe).代表允许系统类被在所有载入器中共享。这是必需的，因为类的安全性会被破坏(be violated），例如，如果应用程序和系统代码对类型"java.lang.String"是什么有不同的概念(notions）。

11.2.2 类载入器和本地库(Class Loaders and Native Libraries)

现在假设在两个"C"类中的方法"f"是本地方法。虚拟器使用"C_f"名字来定位两个 C.f 方法的本地实现。为确保每个 C 类链接到正确的本地函数上，每个类载入器必须它自己的一套本地库，像在图 11.2 中的显示(as shown in Figure 11.2)。

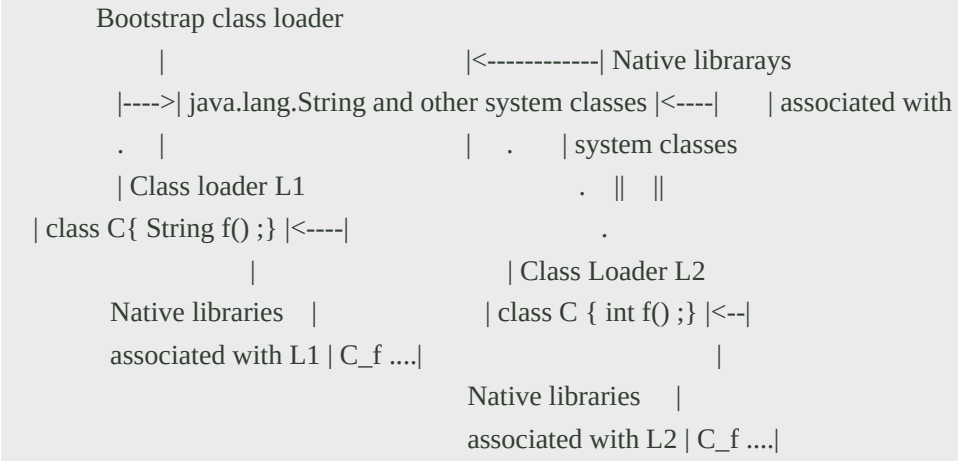


Figure 11.2 Associating native libraries with class loaders

因为每个类载入器维持一套本地库，所以编程者可以使用功能一个单一库来保存所有的被任何类请求的本地方法，只要这些类有一样定义的载入器。

当它们对应的类载入器被垃圾收集时，本地库将自动地被虚拟器载出。

11.2.3 定位本地库(Locating Native Libraries)

通过"System.loadLibrary"方法来载入本地库。在下面例子中，"Cls"类的静态初始化载入一个特定平台的本地库，在其中本地方法"f"被定义:

```
package pkg ;
class Cls{
 native double f(int i, String s) ;
 static{
  System.loadLibrary("mypkg") ;
 }
}
```

"System.loadLibrary"的参数是一个程序员选择的库名字。软件开发者负责选择本地库名字，最小化(minimize)名字冲突的机会(the chance of name clashes)。虚拟器按照一个标准的,而不是特定的主机环境的,协议(convention)来转换库名字到一个本地库名字。例如，"Solaris"操作系统转换"mypkg"名字到"libmypkg.so"，然而"Win32"操作系统转换一样的名字"mypkg"到"mypkg.dll"。

当"Java"虚拟器启动，它构建一个目录列表，被使用来为应用程序类定位的本地库。这列表的内容依赖于主机环境和虚拟器实现。例如，在"Win32 JDK"或"Java 2 SDK releases"下，目录列表包含"Windows"的系统目录，当前工作目录，和在"PATH"环境变量中的实体。在"Solaris JDK or Java 2 SDK releases"下，目录列表包含在"LD_LIBRARY_PATH"环境变量中的实体。

如果它载入命名的本地库失败(fail to load the named native library），"System.loadLibrary"抛出一个"UnsatisfiedLinkError"。如果一个较早的"System.loadLibrary"调用已经载入一样本地库，"System.loadLibrary"安静地完成。如果底层操作系统不支持动态链接，所有本地方法必须被预链接到虚拟器上。在这个例子中，虚拟器完成"System.loadLibrary"调用，但不实际地载入库。

为每个类载入器，虚拟器内部地保持了一个载入的本地库的列表。下面三步决定哪个类载入器应该被连接到一个最新载入本地库上:

1.确定"System.loadLibrary"的直接调用者(immediate caller)
2.识别定义调用者的类
3.得到调用者类定义的载入器。

在下面例子中，本地库"foo"将和"C"定义的载入器相关:

```
class C {
 static{
  System.loadLibrary("foo"）;
```

```
 }
}
```

"Java 2 SDK release 1.2"导入了一个新的"ClassLoader.findLibrary"方法，来允许编程者指定一个自定义的库载入规则，规则详细说明了一个给定的类载入器。"ClassLoader.findLibrary"方法有一个独立于平台的库名(例如(such as)"mypkg")为参数,同时：
.或者返回"null"来命令虚拟器，使用默认的库搜索路劲。
.或者返回一个依赖于主机环境的库文件的绝对路径(例如"c:\\mylibs\\mypkg.dll")。

"ClassLoader.findLibrary"通常和在"Java 2 SDK release 1.2"中添加的另一种方法"System.mapLibraryName"被一起使用的。"System.mapLibraryName"映射独立于平台的库名(例如"mypkg")到依赖平台的库文件名字(例如"mypkg.dll")。

在"Java 2 SDK release 1.2",通过设置"java.library.path"特性(property)，你能重载默认的库搜索路径。例如，下面的命令行启动(start up)一个程序"Foo",它需要在"c:\mylibs"目录中载入一个本地库:

```
java -Djava.library.path=c:\mylibs Foo
```

11.2.4 一个类型安全性限制(A Type Safety Restriction)

虚拟器不允许一个给定的本地库被不止一个类载入器载入。通过多个类载入器，尝试载入一样的本地库导致一个"UnsatisfiedLinkError"异常被抛出。这个限制的目的是确保，基于类载入器的名字空间隔离被保存在本地库中。没有这个限制，通过本地方法，比较容易错误地混用来自不同的类载入器的类和接口。考虑一个本地方法"Foo.f"在一个全局引用中缓冲它自己定义的类:

```
JNIEXPORT void JNICALL
Java_Foo_f(JNIEnv *env, jobject self)
{
 static jclass cachedFooClass ;

 if ( cachedFooClass == NULL ){
  jclass fooClass = (*env)->FindClass(env, "Foo") ;
  if( fooClass == NULL ){
   return ;
  }
  cachedFooClass = (*env)->NewGlobalRef(env, fooClass) ;
  if( cachedFooClass == NULL){
   return ;
  }
  assert( (*env)->IsInstatnceOf(env, self, cachedFooClass)) ;
  ...
```

```
}
}
```

我们期望断言是成功的，因为"Foo.f"是一个实例方法和"self"是"Foo"的实例。然而，这个断言能够失败，如果两个不同的"Foo"类被类载入器"L1"和"L2"载入，同时两个"Foo"类都链接到前面的"Foo.f"实现。为第一次调用"f"方法的"Foo"类，"CachedFooClass"全局引用将被创建。另一个"Foo"类的"f"方法的后来调用将引起断言失败。

在类载入器中，"JDK release 1.1"没有正确地强制本地方法隔离。这意味着对于在不同类载入器中两个类可能链接到一样本地方法。当前面例子显示时，在"JDK release 1.1"中这个方法导致下面两个问题:
.一个类可能错误地(miskakenly)链接到本地库，在一个不同类载入器中，它被一个类用一样名字载入。
.本地方法可能容易混淆来自不同类载入器的类。这破坏了类载入器提供的名字空间的间隔，同时倒是类安全问题。

11.2.3 载出本地库(Unloading Native Libraries)

在虚拟器垃圾收集(garbage collect)和本地库相关的类载入器，虚拟器载出一个本地库。因为类使用它们定义的载入器，这暗示虚拟器也已经载出了类，这个类有静态初始化调用"System.loadLibrary"和载入本地库(11.2.2 部分)。

11.3 链接本地方法(Linking Native Methods)

在第一次调用每个本地方法前，虚拟器尝试链接每个本地方法。一个本地方法"f"能被链接的最早的时间是一个方法"g"的第一次调用，那儿是来自"g"到"f"的方法体的一个引用。虚拟器实现不应该太早地链接一个本地方法。因为实现本地方法的本地库不能被载入，如此做可能导致不能预期的链接错误。

链接一个本地方法包括(involve)下面步骤:
.确定定义本地方法的类的类载入器。
.搜索和这个类载入器相关的本地库集来定位实现本地方法的本地函数。
.建立内部数据结构，使本地方法的所有将来调用将直接地跳到本地函数。

通过连接(concatenate)下面的控件，虚拟器从本地方法的名字中推得(deduce)本地函数的名字:
.前缀"Java_"
.一个编码的完全(fully)合格(qualified)类名字
.一个下划线隔开
.一个编码的方法名字
.为重载本地方法，双下划线跟随编码参数描述符

通过和定义的载入器相关的所有本地库，虚拟器搜索一个恰当名字的本地函数。为每个本地库，虚拟器首先须寻找短名字，是没有参数描述符的名字。然后，寻找带有参数描述符的长名字。只当一个本地函数被用另一个本地方法重载时，程序员需要使用长名字。然而，如果本地方法是对非本地方法的重载，这不是一个问题。后者(非本地方法)不是驻留在本地库中。

在下面的例子中，本地方法"g"不必使用长名字来链接，因为另一个方法"g"不是一个本地方法。

```
class Clsl{
 int g(int i){ ... } // regular method
 native int g(double d) ;
}
```

"JNI"采用(adopt)一个简单的名字编码方法来确保所有"Unicode"字符转化为(translate into)有效的"C"函数名。下划线(underscore)字符隔开完全合格的类名字的控件。因为一个名字或类型描述符没有一个数字的开始，我们能使用_0,...,_9 来转义序列(escape sequences),如下说明(as illustrated below):

Escape Sequence(转义序列)	Denotes（指示）
_0xxxx	a Unicode charater XXXX(一个"Unicode"字符 "xxxx"值)
_1	the charater "_"（下划线字符）
_2	the charater ";" in descriptors(在描述符中的分号字符)
_3	the charater "[" in descriptors(在描述符中的[字符)

如果匹配一个编码本地方法名字的本地函数出现在多个本地库中，在第一个载入的本地库中的函数被链接到本地方法。如果没有函数匹配本地方法名字，一个"UnsatisfiedLinkError"被抛出。

编程者也能调用"JNI"函数"RegisterNatives"来注册本地方法链接到一个类。"RegisterNatives"函数对于静态链接函数是特别有用的。

11.4 调用协议(Calling Conventions)

调用协议决定一个本地函数怎样接受参数和返回结果。在不同本地语言中，或相同语言的不同实现中，没有标准调用协议。例如对于不同的 C++编译器一般产生允许不同调用协议的代码。

如果可能，需要"Java"虚拟器和广泛不同的本地调用协议互操作是困难的。在一个给定主机环境上使用一个指定的标准调用协定来写"JNI"请求本地方法。例如，"JNI"在"UNIX"上按照"C"调用协定，同时在"Win32"上按照"stdcall"协定。

当程序员需要调用使用(follow)不同调用协定的函数时，他们必须写存根(stub)函数，来使适应(adapt)恰当的本地语言的函数的"JNI"调用协定。

11.5 "JNIEnv"接口指针(The JNIEnv Interface Pointer)

通过"JNIEnv"接口到处的不同函数的调用，本地代码来访问虚拟器功能(functionality)。

11.5.1 "JNIEnv"接口指针的组织(Organization of the JNIEnv Interface Pointer)

一个"JNIEnv"接口指针是一个线程局部数据的指针，它又包含一个函数表的值指针。每个接口函数是在表中一个预定义偏移上。"JNIEnv"接口的组织像一个"C++"虚拟函数表，同时也像一个"Microsoft COM"接口。图 11.3，说明"JNIEnv"接口指针的集。

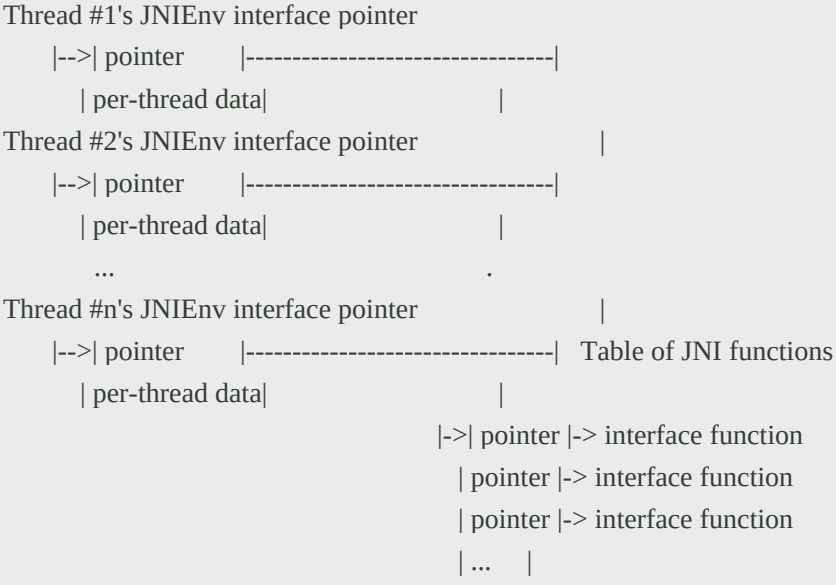


Figure 11.3 Thread Loacl JNIEnv Interface Pointers

实现一个本地方法的函数接受"JNIEnv"接口指针作为它们的第一个参数。保证虚拟器传递一样的接口指针给从一样线程调用的本地方法实现函数。然而，一个本地方法可能被来自不同线程的调用，因此可能被传地不同"JNIEnv"接口指针。虽然接口指针是线程局部的(thread-local)，双重间接的"JNI"函数表被共享在多线程中。

"JNIEnv"接口指针看做一个线程局部结构的原因是一些平台没有对线程局部存储访问的有效支持。通过绕过(pass around)一个线程局部指针，在虚拟器中的"JNI"实现能够避免许多线程局部存储访问操作，否则必须执行这些操作。

因为"JNIEnv"接口指针是线程局部的，本地代码不该在另一个线程中使用属于这个线程的"JNIEnv"接口指针。本地代码可以使用"JNIEnv"指针作为一个线程 ID, 线程 ID 对于线程的生命期是保持唯一的。

11.5.2 一个接口指针的好处(Benefits of an Interface Pointer)

这儿是使用一个接口指针的一些好处，是相对于(as opposed to)硬函数实体的(hardwired function entries):
.最重要地，因为"JNI"函数表是作为一个参数传递给每个本地方法，本地库不必和一个"Java"虚拟器的特殊实现链接。这是关键，因为不同卖主可以命名不同的虚拟器的实现。每个库自包含，是为一样的本地二进制库(the same native library binary)和在一个给定主机环境上来自不同卖主的虚拟器实现一起运行的先决条件(a prerequisite)。

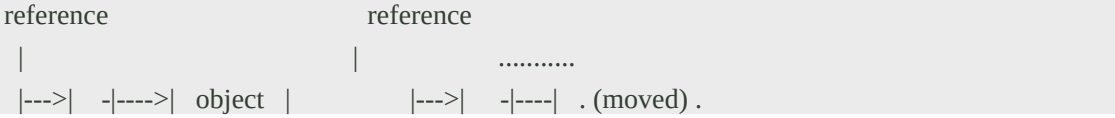
.其次，通过不适用硬函数实体，虚拟器实现可以选择提供多个版本的"JNI"函数表。例如，虚拟器实现可以支持两个"JNI"函数表：一个执行彻底的(thorough)不合逻辑的参数检测，同时适合调试；另一个执行最少的被"JNI"规则请求的检测，同时因此更有效率。"Java 2 SDK release 1.2"支持一个"-Xcheck"："jni"选项(option)选择地(optionally)为"JNI"函数打开(turn on）额外的检测。

.最后，多个"JNI"函数表可以支持将来的类似"JNIEnv"接口的多个版本。虽然我们不能预见(foresee)需要做的，"Java"平台的将来版本能支持一个新的"JNI"函数表，除了被在"1.1 and 1.2 releases"中的"JNIEnv"接口指向的函数表。"Java 2 SDK release 1.2"导入一个"JNI_Onload"函数，它是被一个本地函数定义的来指示(indicate)被本地函数需要的"JNI"函数表的版本。"Java"虚拟器的将来能够同时支持多个版本的"JNI"函数表，同时传递正确版本到依赖他们需要的单独的(individual)本地库。

11.6 传递数据(Passing Data)

基本数据类型，例如整数(intergers），字符(charaters)，等等(and so on),被在"Java"虚拟器和本地代码之间复制。在一方面上的对象通过引用传递。每个引用包含一个直接的指向底层对象的指针。对象的指针(The pointer to object）不能被本地代码直接使用。来自本地代码视角，引用是透明的(references are opaque)。

传递引用，替代直接的对象直接指针，使虚拟器能用更灵活的(flexible)方法来管理对象。图11.4,说明(illustrate）一个引用如此灵活。当本地代码持有一个引用时，虚拟器可以执行一个垃圾收集，导致一个对象被从记忆体的一块地方复制到另一块地发。虚拟器能自动地更新引用的内容，使即使对象已经移动了，引用任然是有效的。



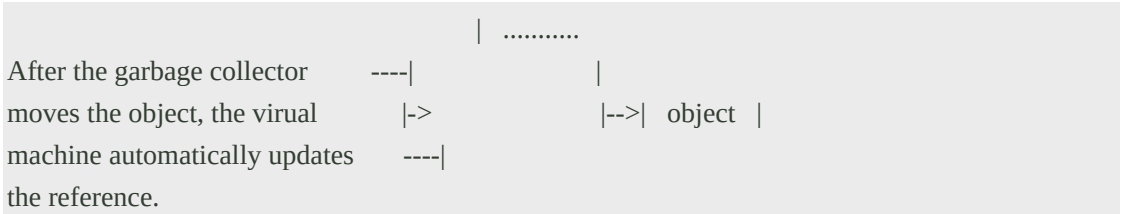


Figure 11.4 Relocating an Object while Native Code Holds a Reference(重新部署一个对象，当本地代码有一个引用时)

11.6.1 全局和局部引用(Global and Local Reference)

"JNI"为本地代码创建两种类型的对象引用：局部和全局引用。局部引用对于一个本地方法调用的期间(duration)是有效的,同时在本地方法返回后自动释放。全局引用保持有效，直到他们被明显地释放。

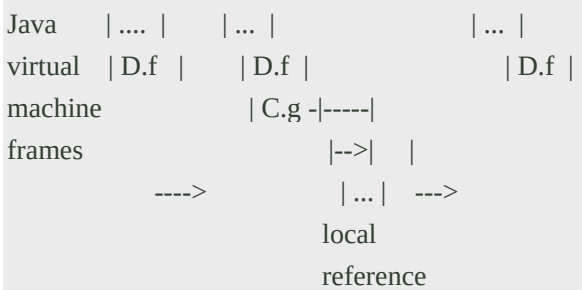
对象作为局部引用传递给本地方法。大多"JNI"函数返回局部引用。"JNI"允许编程者从局部引用来创建全局引用。把对象作为参数的"JNI"函数接受全局和局部引用。一个本地方法可以返回一个局部或一个全局引用到虚拟器作为它的结果。

局部引用只创建它们的线程中有效。本地代码不该传递局部引用从一个线程到另一线程(Frome one thread to another)。

在"JNI"中一个"NULL"引用相当于在"Java"虚拟器中的"null object"。一个值为非空的局部或全局引用不是一个"null object"。

11.6.2 实现局部引用(Implementing Local References)

为实现局部引用，为每个从虚拟器到一个本地方法的过渡控制，"Java"虚拟器创建一个登记(registry)。一个登记映射不能移动局部引用的对象指针。在登记中对象不能被垃圾收集。被传递给本地方法的所有对象，包括作为"JNI"函数调用的结果被返回的对象，被自动地加入登记。在方法返回后登记被删除，允许它的实体被垃圾收集。图 11.5 说明，局部引用登记怎样被创建和被删除。"Java"虚拟器框架对应的本地方法包含一个指向局部引用的登记。一个方法"D.f"调用本地方法"C.g"。"C.g"是被"C"函数"Java_C_g"实现的。在进入"Java_C_g"前，虚拟器创建一个局部引用登记，同时在"Java_C_g"返回后，删除局部引用登记。



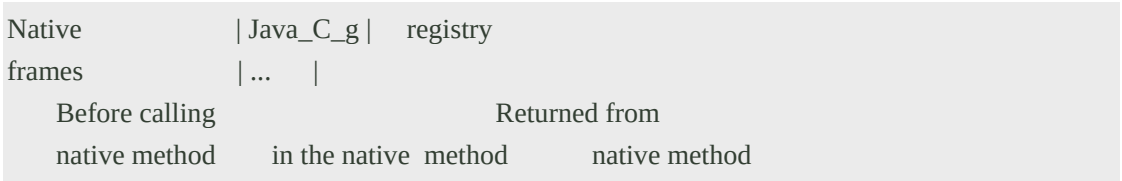


Figure 11.5 Creating and Deleting a Local Reference Registry

有不同的方法来实现一个登记，例如使用一个栈，一个表，一个链接列，或一个哈希(hash)表。虽然引用计数可以被用来避免在登记中有相同的(duplicated)实体(entries)，一个"JNI"的实现无责任来(be obliged to)侦测和去除(collapse)重复实体(duplicate entries)。

局部引用不能通过保守地扫描本地栈来被忠实地实现。本地代码可以存储局部引用到全局或"C"的堆数据结构中。

11.6.3 弱全局引用(Weak Global Reference)

"Java 2 SDK release 1.2"导入一个新的全局引用类：弱全局引用。不像一般全局引用，一个弱全局引用允许一个引用对象被垃圾收集。在底层对象的垃圾收集后，一个弱引用被清除。本地代码能够通过使用"IsSameObject"来比较引用和 NULL 来检测一个弱引用是不是被清除。

11.7 访问对象(Accessing Objects)

"JNI"提供为引用到对象提供了一套丰富的访问函数。这意味着无论虚拟器内部怎样表示对象，都有一样本地函数方法实现。这是至关重要的(crucial)设计决定(decision)，能使"JNI"被任何虚拟器实现来支持。

通过透明的(opaque)引用使用访问函数的开销(overhead)高于直接访问"C"数据结构的开销。我们相信，大多数情况(in most cases)，本地方法执行平凡的任务，掩盖了外部函数调用的费用。

11.7.1 访问基本类型数组(Accessing Primitive Arrays)

然而，对于重复访问在巨大对象中的基本类型数据类型的值，例如整型数组和字符串，函数调用开销是不能接受的。考虑一下被用来执行向量和矩阵(vector and matrix)计算的本地方法。通过一个整数数组的迭代(iterate)同时用一个函数调来返回每一个元素是非常(grossly)效率低下的(inefficient)。

一个解决方法导入一个寄托("pinning"）的概念，使本地方法能够请求虚拟器不移动一个数组的内容。然后，本地方法得到一个指向元素组的指针。然而，这个方法(approach)有两个影响(implication)：
.垃圾收集必修支持寄托(pinning)。在许多实现中，寄托(pinning)是不受欢迎的(undesirable)，因为它使垃圾收集算法(algorithms)变复杂(complicate)，同时导致内存碎片(fragmentation)。

.虚拟器必须连续地(contiguously)安排(lay out)基本类型数组在内存中。虽然(Although)这是对于大多数基本类型数组的正常(natural)实现，但"boolean"数组能像封包或解包一样被实现。一个封包的"boolean"数组使用一个"bit"为每个元素，然后(whereas)一个解包"boolean"数组典型地使用一个"byte"为每一个元素。因此，依赖(rely on)"boolean array"的特别安排的本地代码将不能被移植(portable)。

"JNI"采用一个妥协(compromise)方法来解决(address)上面两个问题。

首先，"JNI"提供一套函数(例如，GetIntArrayRegion and SetIntArrayRegion)来在一段基本类型数组和一个本地内存缓存之间复制基本类型数组元素。如果本地方法需要访问在一个大的数组中的一部分元素，或者如果本地方法需要复制一份这个数组，使用这些函数。

其次，编程者能使用另一套函数(例如，"GetIntArrayElements")来尝试得到一个"pinned"版本的数组元素。然而，依赖于虚拟器的实现，这些函数可以引起存储的分配和复制。事实上这些函数是否复制数组，依赖于虚拟器实现如下(as follows):
.如果垃圾收集支持"pinning"，数组的安排是和一样类型的本地数组的安排一样，然而不需要复制。
.否则，数组被复制到一个不能移动的内存块(例如，在"C"堆上)同时执行必须的格式化转化。返回一个副本的指针。

本地代码调用第三套函数(例如，"ReleaseIntArrayElements")来通知虚拟器，本地代码不再需要访问数组元素。当这个发生时，虚拟器"unpin"数组或者调和带有不可移动的副本的原始数组，同时释放副本。

这个方法提供了灵活性。一个垃圾收集算法能独立决定为数组的复制(copying)或"pinning"。在一个特别实现的设计(sheme)下，垃圾收集可能复制小的数组，但"pin"大的数组。

最后，"Java 2 SDK release 1.2"导入两个新函数:"GetPrimitiveArrayCritical and ReleasePrimitiveArrayCritical"。这些函数能以类似的方法，例如，"GetIntArrayElements and ReleaseIntArrayElements"，来被使用。然而，在它使用"GetPrimitiveArrayCritical"来得到数组元素的一个指针后，和在它使用"ReleasePrimitiveArrayCritical"来释放指针前，在本地代码上有重要的限制。在一个限制域内(Inside a "critical region")，本地代码不应该运行一个不定期的时间片段(即定时函数)，不该调用任意的"JNI"函数，同时不该在虚拟器上执行可以引起当前线程阻塞和等待另一个线程的操作。给出这些限制，虚拟器能够临时地使垃圾收集无效，当让本地代码直接访问数组元素时。因为不需要"pinning"支持，"GetPrimitiveArrayCritical"更适合(be more likely to）返回一个直接的基本类型数组元素的指针，比，例如，"GetIntArrayElements"。

"JNI"实现必须确保，运行在多线程中的本地方法能同时地(simultaneously)访问一样数组。例如，"JNI"可以为每个"pinned（被固定）"数组保持一个内部计数器，使一个线程不会"unpin"

一个被另一线程还"pinned"的数组。注意"JNI"不需要锁住基本类型数组为一个本地方法独占的访问。同时更新来自另一个线程的数组是允许的，虽然这导致不确定的结果(nondeterministic result)。

11.7.2 成员域和方法(Fields and Methods)

"JNI"允许本地代码访问在"Java"编程语言中定义的成员域和方法。"JNI"通过他们的符号名字和类型描述符来标记方法和成员域。一个两步处理分解出从它的名字和描述符定位成员域或方法的成本。例如，读一个整数实例成员域"i"在类"cls"中，首先本地代码得到一个成员域ID，如下:

```
jfieldID fid = env->GetFieldID(env, cls, "i", "I") ;
```

然后本地代码能重复地(repeatedly)使用成员域 ID，没有查询成员域的成本，如下:

```
jint value = env->GetIntField(env, obj, fid) ;
```

一个成员域或方法 ID 保持有效，直到虚拟器载出定义对应的成员域或方法的这个类或接口。这个类或接口被载出后，方法或成员域 ID 变的无效。

编程者能从拥有对应成员域或方法的类或接口得到一个成语域或方法 ID。成员域或方法能被定义在它自己的类或接口中或继承来自父类或父接口。"The Java Virtual Machine Specification"包含了解决成员域和方法的正确的(precise)规则。如果两个类或接口定义了一样的成员域或方法，"JNI"实现一定从这两个类或接口为一个给定名字和描述符，得到一样的成员域 ID 或方法 ID。例如，如果"B"定义成员域"fld"，同时"C"从"B"继承了"fld"，然后编程者保证从类"B"和"C"中为成员域名"fld"得到一样成员域 ID。

"JNI"不会在成员域和方法 IDs 内部怎样实现上做任何限制。

注意你需要成员域名字和成员域描述符来从一个给定类或接口中得到一个成员域 ID。这可能看似不必，因为成员域不能在"Java"编程语言中被重载。然而，在一个类文件中重载成员域，和在"Java"虚拟器上运行如此类文件，是合法的。因此，"JNI"能够处理合法类文件，但这不是为"Java"编程语言的一个编译器产生的。

如果编程者已经知道方法或成员域的名字和类型，编程者只能使用"JNI"来调用方法或访问成员域。相比之下(In comparison)，"the Java Core Reflection API"允许编程者来决定在一个被给的类或接口中的成员域和方法的设置。同时在本地代码中有时能够反映类或接口的类型。"Java 2 SDK release 1.2"提供新的"JNI"函数，他们被设计成和存在的"Java Core Reflect API"一起工作。新的函数包含一对在"JNI"成员域 IDs 和"java.lang.reflect.Field"类的实例之间转换函数，和另一对在"JNI"方法 IDs 和"java.lang.reflect.Method"类的实力之间转换的函数。

11.8 错误和异常(Errors and Exceptions)

在"JNI"程序中出现的错误不同于在"Java"虚拟器实现中发生的异常。编程者错误是"JNI"函数的错误使用(misuses of JNI functions）引起的。例如，编程者可能错误地传递一个对象引用，替代了一个类引用，给"GetFieldID"。例如，通过通过"JNI"当本地代码尝试分配一个对象时，发生内存不够的情况时，虚拟器异常产生。

11.8.1 对于编程的错误不检查(No Checking for Programming Errors)

"JNI"函数不检查编程的错误。传递不合法的参数给"JNI"函数导致(result in)未定义的行为。这样设计决定的原因(The reason for this design decision)如下(as follows)：
.强制"JNI"函数来检查所有可能错误条件，在所有(特别正确)本地方法上降低了性能(degrade the performance)。
.在许多情况，没有足够运行类型信息来执行如此检查。

大多"C"库函数没有防止(guard against)编程错误。例如，"printf"函数，当它收到一个无效的地址，通常触发(trigger）一个运行错误，替代了返回一个错误代码。强制"C"库函数来检查(check for)所有可能错误情况，可能会(would likely)导致如此检查重复，先(once)在用户代码中和然后(then)再在库中。

虽然"JNI"规范(specification)不要求虚拟器来检查编程的错误(errors)，虚拟器实现被鼓励来提供检查一般的错误(mistakes）。例如，一个虚拟器可以在"JNI"函数表的调试版本的中执行更多的检查(11.5.2 部分)。

11.8.2 "Java"虚拟器异常(Java Virtual Machine Exceptions)

"JNI"在本地编程语言中不依赖域异常处理机制(exception handling mechanisms)。通过调用"Throw"或"ThrowNew",本地代码可以引起"Java"虚拟器抛出一个异常。在当前线程中记录一个悬而未决的异常。不像在"Java"编程语言中抛出的异常，在本地代码中抛出的异常不会立即使当前执行中断(disrupt)。

在本地语言中没有标准的异常处理机制。因此(Thus)，在每个能潜在抛出一个异常的操作后，"JNI"编程者被期望检查和处理异常。"JNI"编程者可以用两种方法处理异常:
.在初始化本地方法调用的代码中，本地方法可以选择立即返回，产生异常被抛出。
.本地代码可以通过调用"ExceptionClear"来清除异常，然后执行它自己的异常处理代码。

在调用任何后续(subsequence)"JNI"函数前，最重要是检查，处理和清除一个悬而未决的异常。在带有一个悬而未决异常调用大多"JNI"函数，导致未定义的结果。下面是，在有一个悬而未决(pending)异常时，能过安全地调用的"JNI"函数的完成列表:
ExceptionOccurred
ExceptionDescribe
ExceptionClear
ExceptionCheck

ReleaseStringChars
ReleaseStringUTFchars
ReleaseStringCritical
Release<Type>ArrayElements
ReleasePrimitiveArrayCritical
DeleteLocalRef
DeleteGlobalRef
DeleteWeakGlobalRef
MonitorExit

开始的四个函数是直接地和异常处理相关的。剩下的函数通常是通过"JNI"导出的，在其中他们释放各种虚拟器资源。在异常发生时，经常必须能过释放资源。

11.8.3 异步异常(Asynchronous Exceptions)

在另一个线程中通过调用"Thread.stop"，一个线程可能产生一个异步异常。一个异步异常不会影响在当前线程中的本地代码的执行，直到:
.本地代码调用一个"JNI"函数，它能产生同步异常，或
.本地代码使用"ExceptionOccurred"来明确地(explicitly)检查同步或异步异常。

只有这些能产生潜在地同步异常的"JNI"函数，检查异步异常。

本地方法可以在必要的地方(in necessary places)(例如在没有异常检查的紧密循环中),插入"ExceptionOccurred"检查，来确保当前线程相应(respond to)异步异常在一个合理的时间段内。

产生(generate)异步异常的"Java thread API"，"Thread.stop"，在"Java 2 SDK release 1.2"中已经过时了(be deprecated)。编程者被强烈阻止(be discouraged from)使用"Thread.stop"，因为它一般导致不可信任的(unreliable)编程。这对于"JNI"代码是一个特别的问题。例如，许多今天写的"JNI"库不细心按照规则来检查在这章中描述的异步异常。

第十二章"JNI"类型

这章详细说明(specify)被"JNI"定义的标准数据类型。在引用这些类型前，"C and C++"代码应该包含头文件"jni.h"。

12.1 基本的引用类型(Primitive and Reference Types)

"JNI"定义 C/C++类型的集合对应在"Java"编程语言中基本的引用类型。

12.1.1 基本类型(Primitive Types)

下面的表详细描述在"Java"编程语言中基本类型和在"JNI"中的对应的类型。像在"Java"编程语言中它们的配对物，在"JNI"中所有的基本类型有定义好的(well-defined)大小(sizes)。

Java Language Type	Native Type	Description
boolean	jboolean	unsigned 8 bits
byte	jbyte	signed 8 bits
char	jchar	unsigned 16 bits
short	jshort	signed 16 bits
int	jint	signed 32 bits
long	jlong	signed 64 bits
float	jfloat	32 bits
double	jdouble	64 bits

"jsize"整数类型被用来描述基数的指标和大小(cardinal indices and sizes):

```
typedef jint jsize ;
```

12.1.2 引用的类型(Reference Types)

"JNI"包含一组引用类型，对应在"Java"编程中不同类别的引用类型。在继承关系中组织"JNI"引用类型显示如下(show below)

```
jobject                    (all objects)
  |-- jclass               (java.lang.Class instances)
  |-- jstring              (java.lang.String instances)
  |-- jarray               (arrays)
  |     |-- jobjectArray      (Object[])
  |     |-- jbooleanArray     (boolean[])
  |     |-- jbyteArray        (byte[])
  |     |-- jcharArray        (char[])
  |     |-- jshortArray       (short[])
  |     |-- jintArray         (int[])
  |     |-- jlongArray        (long[])
  |     |-- jfloatArray       (float[])
  |     |-- jdoubleArray      (double[])
  |-- jthrowable           (java.lang.Throwable objects)
```

当在"C"编程语言中使用引用类型时，所有其他 JNI 引用类型被定义为和"jobject"一样的。例如:

```
typedef jobject jclass;
```

当在"C++"编程语言中使用引用类型时，"JNI"介绍了一组空类集合来表带子类的关系在各种引用类之间:

```
class _jobject{} ;
class _jclass: public _jobject{} ;
class _jthrowable：public _jobject{} ;
class _jstring：public _jobject{} ;
class _jarray：public _jobject{} ;
class _jbooleanArray：public _jarray{} ;
class _jbyteArray：public _jarray{} ;
class _jcharArray：public _jarray{} ;
class _jshortArray：public _jarray{} ;
class _jintArray：public _jarray{} ;
class _jlongArray：public _jarray{} ;
class _jfloatArray：public _jarray{} ;
class _jdoubleArray：public _jarray{} ;
class _jobjectArray：public _jarray{} ;

typedef _jobject *jobject ;
typedef _jclass *jclass ;
typedef _jthrowable *jthrowable ;
typedef _jstring *jstring ;
typedef _jarray *jarray ;
typedef _jbooleanArray *jbooleanArray ;
typedef _jbyteArray *jbyteArray ;
typedef _jcharArray *jcharArray ;
typedef _jshortArray *jshortArray ;
typedef _jintArray *jintArray ;
typedef _jlongArray *jlongArray ;
typedef _jfloatArray *jfloatArray ;
typedef _jdoubleArray *jdoubleArray ;
typedef _jobjectArray *jobjectArray ;
```

12.1.3 "jvalue"类型(The jvalue Type)

"jvalue"类型是一个引用类型和基本类型的联合体。定义如下:

```
typedef union jvalue{
 jboolean z ;
 jbyte b ;
 jchar c ;
 jshort s ;
 jint i ;
```

```
 jlong j ;
 jfloat f ;
 jdouble d ;
 jobject l ;
}jvalue ;
```

12.2 成员域和方法 IDs(Field and Method IDs)

方法和成员域 IDs 是规则的"C"指针类型：

```
struct _jfieldID ;
typedef struct _jfieldID *jfieldID ;
struct _jmethodID ;
typedef struct _jmethodID *jmethodID ;
```

12.3 字符串格式(String Formats)

"JNI"使用"C"字符串来表示类名字，成员域和方法名字，和成员域和方法的描述。这些字符串是"UTF-8"格式。

12.3.1 UTF-8 字符串(UTF-8 Strings)

"UTF-8"字符串被编码，使包含非空"ASCII"码的字符串序列能够被每个字符只适用一个"byte"来表示，但是要表示高达"16 bits"的字符串。在"\u0001"到"\u007f"之间的所有字符都是用单一"byte"来表示的，如下:

```
|0| bits 6-0 |
```

在这个"byte"中， 数据的 7 个"bits"给出了字符代表的值。

空(nulll)字符("\u0000")和在"\u0080"到"\u07ff"之间的字符值是用一对"bytes"， x 和 y，来表示的，如下(as follows):

```
x: |1|1|0| bits 10-6 |    y:|1|0| bits 5-0 |
```

"byts"用((x&1f)<<6)+(y&0x3f))的值来表示字符。

在"\u0800"到"\uffff"之间的字符用三个 byts， x，y 和 z，来代表:

```
x:|1|1|1|0| bits 15-12 |   y:|1|0| bits 11-6 |   z:|1|0| bits 5-0 |
```

三个"bytes"用((x&0xf)<<12) + (y&0x3f)<<6 + (z&0x3f))的值来表示。

这个格式和标准的"UTF-8"格式之间的有两个不同。首先，空(null byte)字符(byte 0)是使用两个 byte(two-byte)格式而不是一个 byte(one-byte)格式来编码的。这意味"JNI UTF-8"字符串没有嵌入空（0x0000 无效)。第二，只有"one-byte, two-byte and three-byte"三种格式被用。"JNI"不认识更长的"UTF-8"格式。

12.3.2 类的描述(Class Descriptors)

一个类的描述表示一个类或一个接口的名字。通过用"/"字符替代"."字符(by substituting the "." charater with the "/" charater)，从用"The Java Langugage Specification"定义的一个完全合格(fully qualified)类或接口来得到它。例如，"java.lang.String"的类的描述符是:
"java/lang/String"

数组类(array classes)是使用后跟元素类型的成员域描述符的"["字符表示的。"int[]"的类描述符是:
"[I"

和"double[][][]"的类描述符是：
"[[[D"

12.3.3 成员域描述符(Field Descriptors)

Field Descriptor	Java Language Type
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double

引用类型的成员域描述符从"L"字符开始的(begin with)，后面跟着类描述符，和以";"字符结束(terminated with）。

数组类型(array types)的成员域描述符的构成，跟数组类(array classes)的类描述符一样的规则。下面(The following are)一些例子，为引用类型成员域描述符和他们"Java"编程语言类型副本。

Field Descriptor	Java Language Type
"Ljava/lang/String;"	String
"[I"	int[]
"[Ljava/lang/Object;"	Object[]

12.3.4 方法描述符(Method Descriptors)

通过在一对括号(parenttheses)中放置所有参数类型的成员域描述符，同时被返回类型成员域描述符跟随，来构成方法描述符。在参数类型之间没有空格或其它隔离字符。"V"被用来指示(denote)"void"方法返回类型。构造器使用"V"作为它们的返回类型，和使用"<init>"作为它们的方法名字。

这儿是"JNI"方法描述符的例子和它们对应的方法和构造器类型。

```
Method Descriptor                Java Language Type
"()Ljava/lang/String;"           String f() ;
"(ILjava/lang/Class;)J"           long f(int i, Class c);
"([B)V"                        String(byte[] bytes) ;
```

12.4 常量(Constants)

"JNIEXPORT"和"JNICALL"是宏，被用来指明"JNI"函数和本地方法实现的调用和链接约定。在函数返回类型前面，程序员必须放置"JNIEXPORT"宏，同时在函数名字和返回类型之间放置"JNICALL"。例如：

```
JNIEXPORT jint JNICALL
Java_pkg_Cls_f(JNIEnc *env, jobject this) ;
```

是一个"C"函数的原型，实现了"pkg.Cls.f",然而：

```
jint (JNICALL *f_ptr)(JNIEnv *env, jobject this) ;
```

是函数指针变量，它能被赋予"Java_pkg_Cls_f"函数。

"JNI_FALSE"和"JNI_TRUE"是个常数，为"jboolean"类型定义的。

```
#define JNI_FALSE 0
#define JNI_TRUE 1
```

"JNI_OK"表示"JNI"函数的成功返回值，同时"JNI_ERR"有时被用来表示错误的情况。

```
#define JNI_OK 0
#define JNI_ERR (-1)
```

不是所有的错误情况都用"JNI_ERR"来表示，因为"JNI"规范(specification)当前没有包含一组标准的错误编码。"JNI"函数在成功时返回"JNI_OK"，同时在失败时返回一个负数。

下面两个常数被用在释放原始数组的本地副本的函数中。这个函数的一个例子是"ReleaseIntArrayElements"。"JNI_COMMIT"强制本地数组被复制回在"Java"虚拟器中的原始数组中。"JNI_ABORT"释放为本地数组分配的空间，不用复制回新的内容。

```
#define JNI_COMMIT 1
#define JNI_ABORT 2
```

"Java 2 SDK release 1.2"介绍两个常量表示"JNI"版本号(version numbers）。

```
#define JNI_VERSION_1_1 0X00010001
#define JNI_VERSION_1_2 0X00010002
```

通过执行下面的编译条件，一个本地应用程序可以决定是否编译 1.1 或 1.2 版本的"jni.h"文件:

```
#ifdef JNI_VERSION_1_2

#else

#endif
```

下面的常量代表"GetEnv"函数返回的详细的错误代码，它是"JavaVM"接口的一部分:

```
#define JNI_EDETACHED (-2)
#define JNI_EVERSION (-3)
```