前言

翻译初衷，记录 JNI 编程经验以备后查，并奢望以 JNI 为蓝本，写一本更深入的关于虚拟机的书。真做起来，才发现以现有水平只能仰望这个目标，要达到它，还需要几年积累。

本书没有采用逐字逐句的翻译，更多采用意译，请大家在阅读时多参考原著；对于书中夹杂的评论，如有伤观感，请大家见谅。

现在有无数优秀的开源项目，以前高深莫测的技术(虚拟机、编译器、操作系统、协议栈和 IDE...)， 我们终于有机会一探究竟了，真令人兴奋。我们学习，我们参与，希望有一天我们中国人也能创一门牛技术。

感谢 Die...ken 的审稿，他严谨和认真的态度，深感敬佩；哥们儿祝你：天天开心，早结连理。

感谢老婆。老婆读书时，看见别人写的书总会感谢太太云云，煞是羡慕，总追问：你什么时候写书感谢我？难！翻译都这么费劲，写书就不知猴年马月了，在这儿感谢一下，糊弄糊弄得了。

do.chuan@gmail.com

Preface

本书涵盖了 Java Native Interface(JNI)的内容，将探讨以下问题：

- 在一个 Java 项目中集成一个 C/C++库
- 在一个用 C/C++开发的项目中，嵌入 JavaVM
- 实现 Java VM
- 语言互操作性问题，特别是互操作过程中的垃圾回收(GC, garbage collection)和并发编程(multithreading)

译注：
JNI(Java Native Interface)是 SUN 定义的一套标准接口，如 Dalvik, Apache Harmony 项目...等 Java 虚拟机，都会实现 JNI 接口，供本地(C/C++)应用与 Java VM 互调。
JNI: 可以供 Java 代码调用本地代码，本地代码也可以调用 Java 代码，即上文列出的第 4 条内容：语言互操作；所以，这是一套完善而功能强大的接口。
可能有朋友听说过 KNI，那是 J2ME VM(CLDC)中定义的一套东西，不如 JNI 强大。
此外，因为 C/C++在系统编程领域的地位，只要打通了与 C/C++的接口，就等于是天堑变通途。

首先，通过本书，你会很容易的掌握 JNI 开发，并能了解到方方面面的关于 JNI 的知识。本书详尽的叙述，会带给你你很多如何高效使用 JNI 的启示。JNI 自 1997 年初发布以来，Sun 的工程师们和 Java 社区使用 JNI 的经验造就了本书。

第二，本书介绍了 JNI 的设计原理。这些原理，不仅会使学术界感兴趣，也是高效使用 JNI 的前提。

第三，本书的某些部分是 Java 2 平台规范的最终版本。 JNI 程序员可以此书作为规范的参考手册，Java 虚拟机实现者必须遵循规范，以保证各平台实现的一致性。

(...几段不重要，未翻译...)

CHAPTER 1
Introduction

JNI 是 Java 平台中的一个强大特性。

应用程序可以通过 JNI 把 C/C++代码集成进 Java 程序中。通过 JNI，开发者在利用 Java 平台强大功能的同时，又不必放弃对原有代码的投资；因为 JNI 是 Java 平台定义的规范接口，当程序员向 Java 代码集成本地库时，只要在一个平台中解决了语言互操作问题，就可以把该解决方案比较容易的移植到其他 Java 平台中。

译注：

比如为 Dalvik 添加了一个本地库，也可以把这个本地库很容易的移植到 J2SE 和 Apache Harmony 中，因为在 Java 与 C/C++互操作方面，大家都遵循一套 API 接口，即 JNI。

本书由下列三个部分组成：

- Chapter 2 通过简单示例介绍了 JNI 编程
- Chapter 3-10，对 JNI 各方面特性和功能做介绍，并给出示例(译者：重要)
- Chapters 11-13, 罗列 JNI 所有的数据类型的定义

(...几段不重要，未翻译...)

1.1 The Java Platform and Host Environment

因本书覆盖了 Java 和本地(C, C++, etc...)编程语言，让我们首先理一理这些编程语言的适用领域。

Java 平台(Java Platform)的组成：Java VM 和 Java API. Java 应用程序使用 Java 语言开发，然后编译成与平台无关的字节码(.class 文件)。 Java API 由一组预定义的类组成。任何组织实现的 Java 平台都要支持：Java 编程语言，虚拟机，和 API(译者：Sun 对 Java 语言、虚拟机和 API 有明确规范)。

平台环境：操作系统，一组本机库，和 CPU 指令集。本地应用程序，通常依赖于一个特定的平台环境，用 C、C++等语言开发，并被编译成平台相关的二进制指令，目标二进制代码在不同 OS 间一般不具有可移植性。

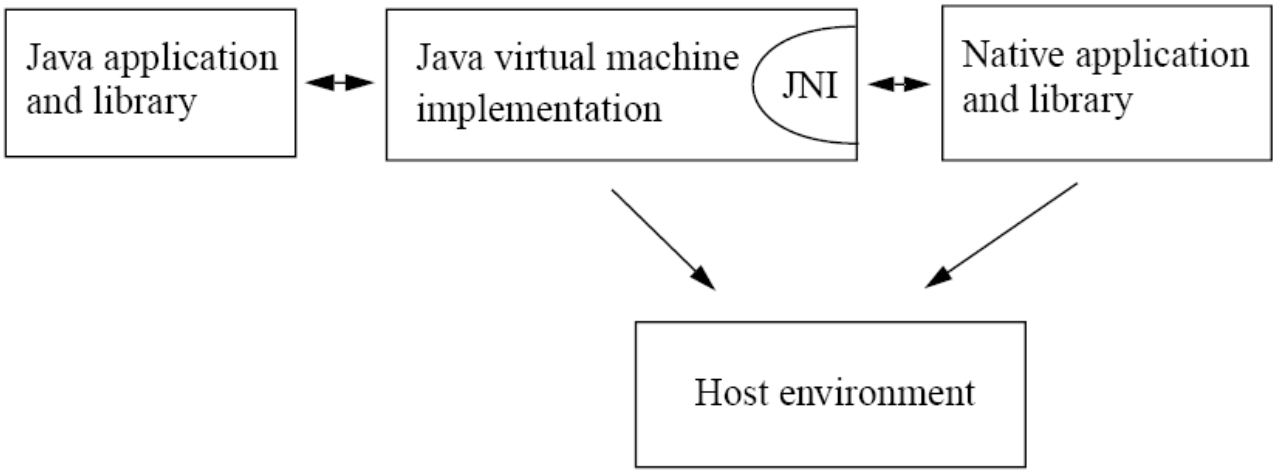
Java 平台(Java VM 和 Java API)一般在某个平台下开发。 比如，Sun 的 Java Runtime Environment(JRE)支持类 Unix 和 Windows 平台. Java 平台做的所有努力，都为了使程序更具可移植性。

1.2 Role of the JNI

当 Java 平台部署到本地系统中，有必要做到让 Java 程序与本地代码协同工作。 部分是由于遗留代码(保护原有的投资)的问题(一些效率敏感的代码用 C 实现，但现在 JavaVM 的执行效率完全可信赖)，工程师们很早就开始以 C/C++为基础构建 Java 应用，所以，C/C++代码将长时间的与 Java 应用共存。

JNI 让你在利用强大 Java 平台的同时，使你仍然可以用其他语言写程序。 作为 JavaVM 的一部分，JNI 是一套双向的接口，允许 Java 与本地代码间的互操作。

如图 1.1 所示



作为双向接口，JNI 支持两种类型本地代码：本地库和本地应用。

- 用本地代码实现 Java 中定义的 native method 接口，使 Java 调用本地代码
- 通过 JNI 你可以把 Java VM 嵌到一个应用程序中，此时 Java 平台作为应用程序的增强，使其可以调用 Java 类库

比如，在浏览器中运行 Applet, 当浏览器遇到"Applet"标签，浏览器就会把标签中的内容交给 Java VM 解释执行，这个实现，就是典型的把 JavaVM 嵌入 Browser 中。

译注:

JNI 不只是一套接口，还是一套使用规则。 Java 语言有"native"关键字，声明哪些方法

是用本地代码实现的. 翻译的时候，对于"native method"，根据上下文意思做了不同处理，当 native method 指代 Java 中用"native"关键字修饰的那些方法时，不翻译；而当代码用 C/C++实现的部分翻译成了本地代码。

上述，在应用中嵌入 Java VM 的方法，是用最少的力量，为应用做最强扩展的不二选择，这时你的应用程序可以自由使用 Java API 的所有功能；大家有兴趣可以读一读浏览器是怎么扩展 Applet 的，或者读一读 Palm WebOS 的东西。

译者最近一年都在做这件事，对这个强大的功能，印象特别深刻. 我们整个小组做了两个平台的扩展，设计、编码、测试和 debug 用了近一年半时间，代码量在 14000 行左右，做完扩展后，平台功能空前增强。我感觉做软件，难得不在编码，难在开始的设计和后期的测试、调试和优化，并最终商用，这就要求最终产品是一个强大而稳定的平台，达到此目标是个旷日持久的事. 看看 Java，Windows，Linux，Qt，WebKit 发展了多少年?

向所有软件工程师致敬!

1.3 Implications of Using the JNI

请记住，当 Java 程序集成了本地代码，它将丢掉 Java 的一些好处。

首先，脱离 Java 后，可移植性问题你要自己解决，且需重新在其他平台编译链接本地库。

第二，要小心处理 JNI 编程中各方面问题和来自 C/C++语言本身的细节性问题，处理不当，应用将崩溃。

一般性原则：做好应用程序架构，使 native methods 定义在尽可能少的几个类里。

译注:

学习 JNI 编程是个漫长的实践过程，会碰到无数问题。

用 C/C++编程，常见问题有内存泄露，指针越界...，此外使用了 JNI，还要面对 JavaVM 的问题：

- 在本地代码中 new 一个 Java 对象后期望在本地代码中维护此对象的引用，如何避免被 GC?
- Java 面向对象语言的封装性被破坏了，Java 类中任何方法和属性对 JNI 都是可见的，不管它是 public 的，还是 private/protected/package 的
- 对 LocalRef/GlobalRef 管理不善，会引发 Table Overflow Exception，导致应用崩溃
- 从 JNI 调用 Java 的过程不是很直观，往往几行 Java 代码能搞定的事情，用 JNI 实现却要几百行

虽然，有这样多问题，逃避不了，你就认了吧。 经过一段时间的实践，当你能熟练处理这些问题时，就会，眉头一皱，文思泉涌，指尖飞舞，瞬间几百行代码诞生了，一个 make 全部编译通过，这时的你肯定已经对 JNI 上瘾了.....

1.4 When to Use the JNI

当你准备在项目中使用 JNI 之前，请先考虑一下是否有其他更合适的方案。 上节有关 JNI 缺点的介绍，应该引起你足够的重视。

这里介绍几个不通过 JNI 与其他语言交互的技术：

- IPC 或者 通过 TCP/IP 网络方案 （ Android ASE)
- 数据库方面，可以使用 JDBC
- 使用 Java 的分布式对象技术: Java IDL API

译注:
IPC 与 TCP/IP 是常用的基于协议的信息交换方案. 可以参考 Android 上的 Binder 和 ASE(Android Script Environment)。

一典型的解决方案是，Java 程序与本地代码分别运行在不同的进程中. 采用进程分置最大的好处是：一个进程的崩溃，不会立即影响到另一个进程。

但是，把 Java 代码与本地代码置于一个进程有时是必要的。 如下：

- Java API 可能不支某些平台相关的功能。比如，应用程序执行中要使用 Java API 不支持的文件类型，而如果使用跨进程操作方式，即繁琐又低效
- 避免进程间低效的数据拷贝操作
- 多进程的派生：耗时、耗资源(内存)
- 用本地代码或汇编代码重写 Java 中低效方法

总之，如果 Java 必须与驻留同进程的本地代码交互，请使用 JNI。

译注:
写代码是技巧和艺术，看你想在设计上下多大功夫. 比如: Chrome，是多进程的典范，她的简洁、高效，令人叹服。

1.5 Evolution of the JNI

关于 Java 应用程序如何与本地代码互操作的问题，在 Java 平台早期就被提了出来. JDK1.0 包括了一套与本地代码交互的接口。 当时许多 Java 方法和库都依赖本地方法实现(如 java.io, java.net)。

但是，JDK release 1.0 有两个主要问题：

- Java 虚拟机规范未定义对象布局，本地代码访问对象的成员是通过访问 C 结构的成员实现的
- 本地代码可以得到对象在内存中的地址，所以，本地方法是 GC 相关的

为解决上述问题对 JNI 做了重新设计，让这套接口在所有平台都容易得到支持。

- 虚拟机实现者通过 JNI 支持大量的本地代码
- 工具开发商不用处理不同种类的本地接口
- 所有 JNI 开发者面对的是操作 JavaVM 的规范 API

JNI 的首次支持是在 JDK release 1.1，但 1.1 内部 Java 与本地代码的交互仍然使用原始方式(JDK 1.0). 但这种局面，没有持续很久，在 Java 2 SDK release 1.2 中 Java 层与本地代码的交互部分用 JNI 重写了。

作为 JavaVM 规范的一部分，Java 层与本地代码的交互，都应通过 JNI 实现。

1.6 Example Programs

本书注重 JNI 编程，不涉及如何通过第三方工具简化该过程。

(译者：不重要，未翻译)。

请从官网下载本书的示例代码：http://java.sun.com/docs/books/jni/

CHAPTER 2

Getting Started

本章用Hello World示例带你领略JNI编程。

2.1 Overview

准备过程：

1. 创建一个类(HelloWorld.java)
2. 使用 javac 编译该类
3. 利用 javah -jni 产生头文件
4. 用本地代码实现头文件中定义的方法
5. Run

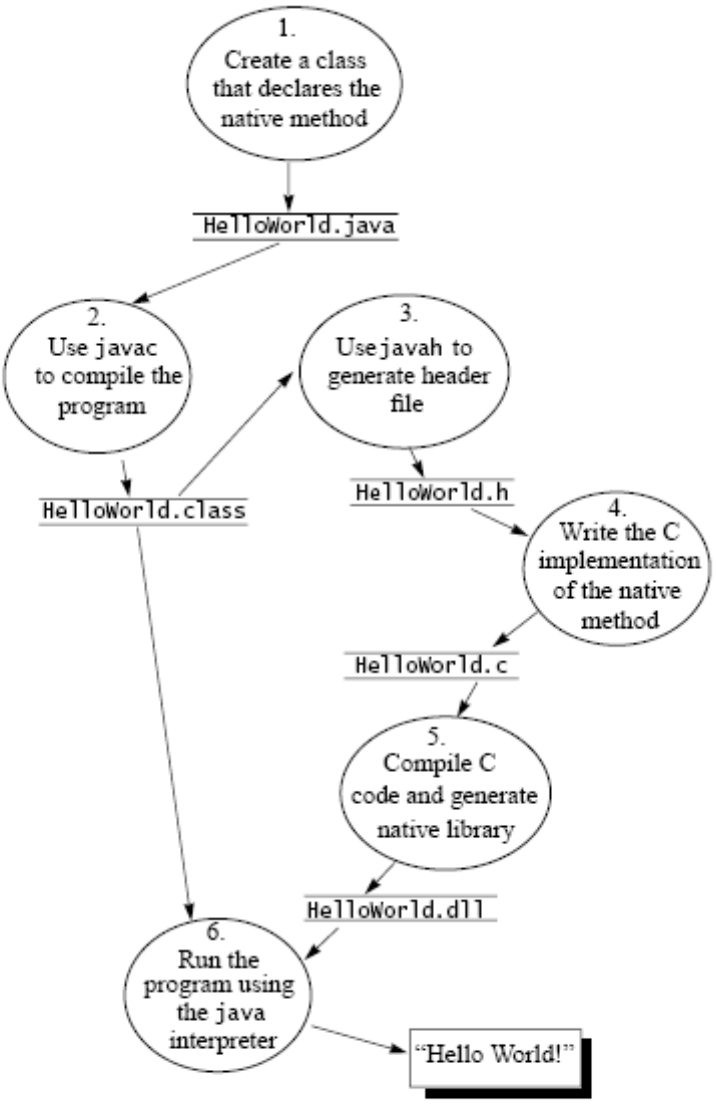


Figure 2.1 Steps in Writing and Running the "Hello World" Program

译注:

在一个特定环境中,写本地实现的过程是不同的(如 Android)。

javah 主要是生成头文件和函数签名(每个方法和成员都有签名，后有详细介绍)，通过 javah 学习如何正确的写法。

注意：如上述 HelloWorld.java，编译后的文件为 HelloWorld.class, 用

$javah HelloWorld

来产生头文件，不要带末尾的".class"。

2.2 Declare the Native Method

HelloWorld.java

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

    static {
      System.loadLibrary("HelloWorld");
    }
}
```

HelloWrold 类首先声明了一个 private native print 方法. static 那几行是本地库。

在 Java 代码中声明本地方法必须有"native"标识符，native 修饰的方法，在 Java 代码中只作为声明存在。

在调用本地方法前，必须首先装载含有该方法的本地库. 如 HelloWorld.java 中所示，置于 static 块中，在 Java VM 初始化一个类时，首先执行这部分代码，这可保证调用本地方法前，装载了本地库。

装载库的机制，后有介绍。

2.3 Compile the HelloWorld Class

$javac HelloWorld.java

2.4 Create the Native Method Header File

$javah -jni HelloWorld

译者："-jni"为默认参数，可有可无.

上述命令，生成 HelloWorld.h 文件. 关键部分如下：

JNIEXPORT void JNICALL

Java_HelloWorld_print (JNIEnv *, jobject);

现在，请先忽略两个宏：JNIEXPORT 和 JNICALL。 你会发现，该函数声明，接受两个参数，而对应的 Java 代码对该函数的声明没有参数。第一个参数是指向 JNIEnv 结构的指针； 第二个参数，为 HelloWorld 对象自身，即 this 指针。

译注：

JNIEnv 是 JNI 核心数据之一，地位非常崇高，所有对 JNI 的调用都要通过此结构。

2.5 Write the Native Method Implementation

必须根据 javah 生成的本地函数声明实现函数，如下：

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

请注意："jni.h"文件必须被包含，该文件定义了 JNI 所有的函数声明和数据类型。

2.6 Compile the C Source and Create a Native Library

请注意，生成的本地库的名字，必须与 System.loadLibrary("HelloWorld")；待装载库的名字相同。

Solaris:

$cc -G -I/java/include -I/java/include/solaris HelloWorld.c -o libHelloWorld.so

-G: 生成共享库

Win:

$cl -Ic:\java\include -Ic:\java\include\win32 -MD -LD HelloWorld.c -FeHelloWorld.dll

-MD：保证与 Win32 多线程 C 库连接(译者：Win 上分静态、动态、动态多线程...C 库)

-LD: 生成动态链接库

2.7 Run the Program

Solaris or Win:

$java HelloWorld

输出：

Hello World!

运行前，必须保证连接器，能找到待装载的库，不然，将抛如下异常：

```
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)
```

如，Solaris, 通过 sh 或 ksh shell：

$LD_LIBRARY_PATH=.

$export LD_LIBRARY_PATH

C shell：

$setenv LD_LIBRARY_PATH .

在 Win 上，请保证待装载库在当前位置，或在 PATH 环境变量中。

你也可以如下：

java -Djava.library.path=. HelloWorld

-D：设置 Java 平台的系统属性。 此时 JavaVM 可以在当前位置找到该库。

CHAPTER 3

Basic Types, Strings, and Arrays

JNI 编程中常被提到的问题是，Java 语言中的数据类型是如何映射到 c/c++本地语言中的。

实际编程中，向函数传参和函数返回值是很普遍的事情。 本章将介绍这方面技术，我们从基本类型(如 int)和一般对象(如 String 和 Array)开始介绍. 其他内容将放在下一章介绍。

译注：

JavaVM 规范中称 int，char，byte 等为 primitive types，译者平时叫惯了基本类型，所以翻译时延用了习惯，不知合适否。

3.1 A Simple Native Method

扩充 HelloWorld.java，该例是先打印一串字符，然后等待用户的输入， 如下：

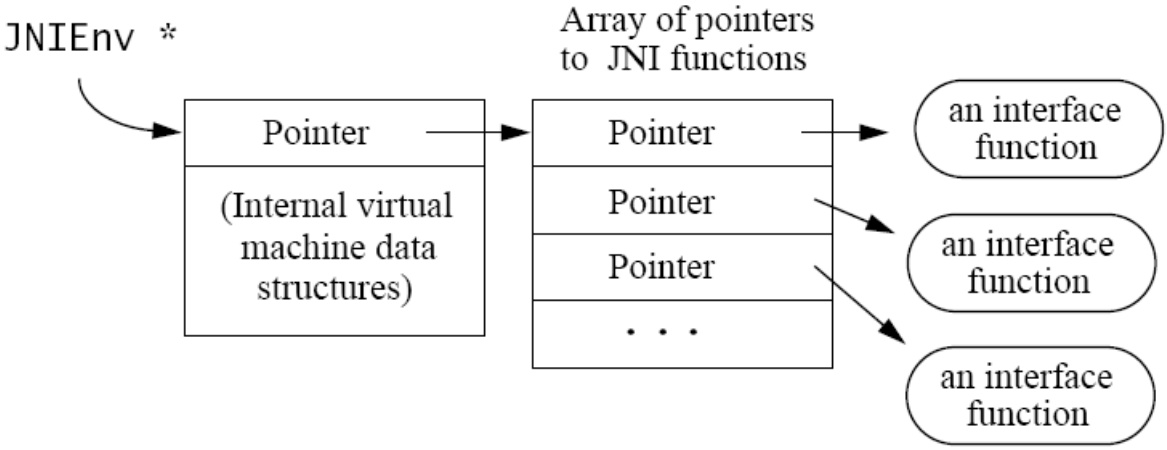
```
class Prompt {
    // native method that prints a prompt and reads a line
    private native String getLine(String prompt);
    public static void main(String args[]) {
        Prompt p = new Prompt();
        String input = p.getLine("Type a line: ");
        System.out.println("User typed: " + input);
        }

    static {
        System.loadLibrary("Prompt");
    }
}
```

Prompt.getLine 方法的 C 声明如下：

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

3.1.2 Native Method Arguments

Java_Prompt_getLine 接收 3 个参数: JNIEnv 结构包括 JNI 函数表。



第二个参数的意义取决于该方法是静态还是实例方法(static or an instance method)。当本地方法作为一个实例方法时，第二个参数相当于对象本身，即 this. 当本地方法作为一个静态方法时，指向所在类. 在本例中，Java_Prompt_getLine 是一个本地实例方法实现，所以 jobject 指向对象本身。

> *译注：*
> Java 语言中类与对象的联系与区别，概念很清晰，但在 JNI 和 VM 中，有一些问题需要说明，后有专门文章阐述。

3.1.3 Mapping of Types

在 native method 中声明的参数类型，在 JNI 中都有对应的类型.

在 Java 中有两类数据类型：primitive types，如，int, float, char；另一种为 reference types，如，类，实例，数组。

> *译者：*
> 数组，不管是对象数组还是基本类型数组，都作为 reference types 存在，并有专门的 JNI 方法取数组中每个元素.

Java 与 JNI 基本类型的映射很直接，如下:

Java	Native(jni.h)
boolean	jboolean
byte	jbyte
char	jchar
short	jshort
int	jint
long	jlong
float	jfloat

double	jdouble

相比基本类型，对象类型的传递要复杂很多。 Java 层对象作为 opaque references(指针)传递到 JNI 层。 Opaque references 是一种 C 的指针类型，它指向 JavaVM 内部数据结构。使用这种指针的目的是：不希望 JNI 用户了解 JavaVM 内部数据结构。对 Opaque reference 所指结构的操作，都要通过 JNI 方法进行. 比如，"java.lang.String"对象，JNI 层对应的类型为 jstring，对该 opaque reference 的操作要通过 JNIEnv->GetStringUTFChars 进行。

译注：

一定要按这种原则编程，千万不要为了效率或容易的取到某个值，绕过 JNI，直接操作 opaque reference.

JNI 是一套完善接口，所有需求都能满足。

在 JNI 中对象的基类即为 jobject. 为方便起见，还定义了 jstring，jclass, jobjectArray 等结构，他们都继承自 jobject。

3.2 Accessing Strings

如下使用方式是错误的，因为 jstring 不同于 C 中的 char *类型。

```
JNIEXPORT jstring JNICALL
Java_Prompt_getLine(JNIEnv *env, jobject obj, jstring prompt)
{
  /* ERROR: incorrect use of jstring as a char* pointer */
  printf("%s", prompt);
  ...
}
```

3.2.1 Converting to Native Strings

使用对应的 JNI 函数把 jstring 转成 C/C++字串。JNI 支持 Unicode/UTF-8 字符编码互转。Unicode 以 16-bits 值编码；UTF-8 是一种以字节为单位变长格式的字符编码，并与 7-bits ASCII 码兼容。UTF-8 字串与 C 字串一样，以 NULL('\0')做结束符，当 UTF-8 包含非 ASCII 码字符时，以'\0'做结束符的规则不变。7-bit ASCII 字符的取值范围在 1-127 之间，这些字符的值域与 UTF-8 中相同。当最高位被设置时，表示多字节编码。

如下，调用 GetStringUTFChars，把一个 Unicode 字串转成 UTF-8 格式字串，如果你确定字串只包含 7-bit ASCII 字符。这个字串可以使用 C 库中的相关函数,如 printf.

如何操作 non-ASCII 字符，后面有介绍。

```
JNIEXPORT jstring JNICALL
```

```
Java_Prompt_getLine(JNIEnv *env, jobject obj, jstring prompt)
{
  char buf[128];
  const jbyte *str;
  str = (*env)->GetStringUTFChars(env, prompt, NULL);
  if (str == NULL) {
    return NULL; /* OutOfMemoryError already thrown */
  }
  printf("%s", str);
  (*env)->ReleaseStringUTFChars(env, prompt, str);
  /* We assume here that the user does not type more than
   * 127 characters */
  scanf("%127s", buf);
  return (*env)->NewStringUTF(env, buf);
}
```

记得检测 GetStringUTFChars 的返回值，因为调用该函数会有内存分配操作，失败后，该函数返回 NULL，并抛 OutOfMemoryError 异常。

如何处理异常，后面会有介绍。JNI 处理异常，不同于 Java 中的 try...catch。在 JNI 中，发生异常，不会改变代码执行轨迹，所以，当返回 NULL，要及时返回，或马上处理异常。

3.2.2 Freeing Native String Resources

调用 ReleaseStringUTFChars 释放 GetStringUTFChars 中分配的内存(Unicode -> UTF-8 转换的原因)。

3.2.3 Constructing New Strings

使用 JNIEnv->NewStringUTF 构造 java.lang.String；如果此时没有足够的内存，NewStringUTF 将抛 OutOfMemoryError 异常，同时返回 NULL。

3.2.4 Other JNI String Functions

除了 GetStringUTFChars, ReleaseStringUTFChars, 和 NewStringUTF, JNI 还支持其他操作 String 的函数供使用。

GetStringChars 是有 Java 内部 Unicode 到本地 UTF-8 的转换函数，可以调用 GetStringLength，获得以 Unicode 编码的字串长度。也可以使用 strlen 计算 GetStringUTFChars 的返回值，得到字串长度。

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

上述声明中，有 isCopy 参数，当该值为 JNI_TRUE，将返回 str 的一个拷贝；为 JNI_FALSE 将直接指向 str 的内容。 注意：当 isCopy 为 JNI_FALSE，不要修改返回值，不然将改变 java.lang.String 的不可变语义。

一般会把 isCopy 设为 NULL，不关心 Java VM 对返回的指针是否直接指向 java.lang.String 的内容。

一般不能预知 VM 是否会拷贝 java.lang.String 的内容，程序员应该假设 GetStringChars 会为 java.lang.String 分配内存。在 JavaVM 的实现中，垃圾回收机制会移动对象，并为对象重新配置内存。一但 java.lang.String 占用的内存暂时无法被 GC 重新配置，将产生内存碎片，过多的内存碎片，会更频繁的出现内存不足的假象。

记住在调用 GetStringChars 之后，要调用 ReleaseStringChars 做释放，不管在调用 GetStringChars 时为 isCopy 赋值 JNI_TRUE 还是 JNI_FALSE，因不同 JavaVM 实现的原因，ReleaseStringChars 可能释放内存，也可能释放一个内存占用标记(isCopy 参数的作用，从 GetStringChars 返回一个指针，该指针直接指向 String 的内容，为了避免该指针指向的内容被 GC，要对该内存做锁定标记)。

3.2.5 New JNI String Function in Java 2 SDK Release 1.2

为尽可能的避免内存分配，返回指向 java.lang.String 内容的指针，Java 2 SDK release 1.2 提供了：Get/RleaseStringCritical. 这对函数有严格的使用原则。

当使用这对函数时，这对函数间的代码应被当做临界区(critical region). 在该代码区，不要调用任何会阻塞当前线程和分配对象的 JNI 函数，如 IO 之类的操作。

上述原则，可以避免 JavaVM 执行 GC。因为在执行 Get/ReleaseStringCritical 区的代码时，GC 被禁用了，如果因某些原因在其他线程中引发了 JavaVM 执行 GC 操作，VM 有死锁的危险：当前线程 A 进入 Get/RelaseStringCritical 区，禁用了 GC，如果其他线程 B 中有 GC 请求，因 A 线程禁用了 GC，所以 B 线程被阻塞了；而此时，如果 B 线程被阻塞时已经获得了一个 A 线程执行后续工作时需要的锁；死锁发生了。

可以嵌套调用 GetStringCritical:

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

```
    (*env)->ReleaseStringCritical(env, jstr1, s
    ... /* error handling */
}
...     /* use s1 and s2 */
(*env)->ReleaseStringCritical(env, jstr1, s1);
(*env)->ReleaseStringCritical(env, jstr2, s2);
```

GetStringCritical 因 VM 实现的原因，会涉及内存操作，所以我们需要检查返回指. 比如，对于 java.lang.String 来说，VM 内部并不是连续存储的，所以 GetStringCritical 要返回一个连续的字符数组，必然要有内存操作。

为避免死锁，此时应尽量避免调用其他 JNI 方法，只允许调用 GetStringCritical/ReleaseStringCritical,Get/ReleasePrimitiveArrayCritical 因 VM 内部 Unicode 编码的缘故，所以 Get/ReleaseStringUTFCritical 这种涉及 Unicode->UTF8 转换要分配内存的函数不支持。

GetStringRegion/GetStringUTFRegion，向准备好的缓冲区赋值，如下:

```
JNIEXPORT jstring JNICALL
Java_Prompt_getLine(JNIEnv *env, jobject obj, jstring prompt)
{
    /*assumethepromptstringanduserinputhaslessthan128
        characters */
    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 有两个参数，starting index 和 length, 这两个参数以 Unicode 编码计算. 该函数会做边界检查，所以可能抛出 StringIndexOutOfBoundsException。

因为该函数不涉及内存操作，所以较 GetStringUTFChars 使用要简单。

译注:

有两个函数：GetStringLength/GetStringUTFLength，前者是 Unicode 编码长度，后者是 UTF 编码长度。

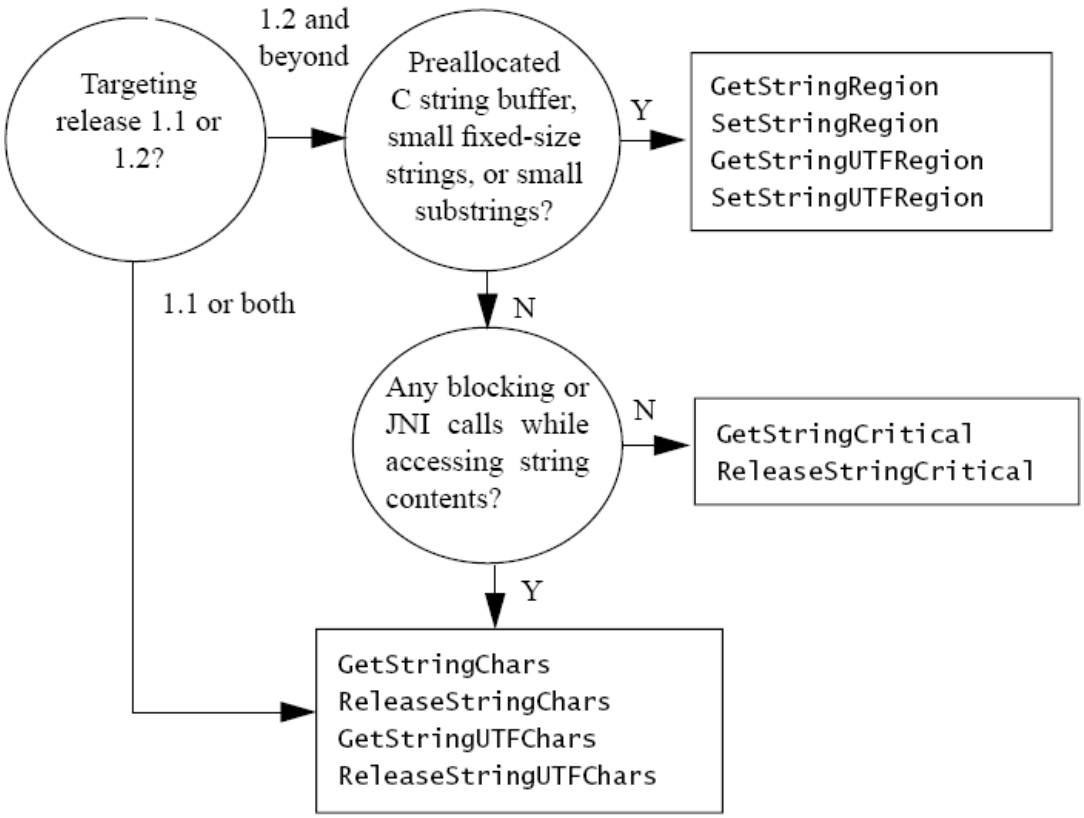
GetStringUTFRegion 很有用，因为你不能修改 GetStringUTFChars 返回值，所以需要另外 malloc/strcpy 之后，再操作返回值，耗时费力，不如直接使用 GetStringUTFRegion 来的简洁、高效。

3.2.6 Summary of JNI String Functions

JNI Function	Description	Since
GetStringChars ReleaseStringChars	Obtains or releases a pointer to the contents of a string in Unicode format.May return a copy of the string.	JDK1.1
GetStringUTFChars ReleaseStringUTFChars	Obtains or releases a pointer to the contents of a string in UTF-8 format. May return a copy of the string.	JDK1.1
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	Creates a java.lang.String instance that contains the same sequence of characters as the given Unicode C string.	JDK1.1
NewStringUTF	Creates 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 the string. Native code must not block between a pair of Get/ ReleaseStringCritical calls.	Java 2 SDK1.2
GetStringRegion SetStringRegion	Copies the contents 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 Choosing among the String Functions

该表给出了选择字符串函数的策略：



如果你使用 JDK 1.1 或 JDK 1.2，你只能使用 Get/ReleaseStringChars 和 Get/ReleaseStringUTFChars。

对于小尺寸字串的操作，首选 Get/SetStringRegion 和 Get/SetStringUTFRegion，因为栈上空间分配，开销要小的多；而且没有内存分配，就不会有 out-of-memory exception。如果你要操作一个字串的子集，本套函数的 starting index 和 length 正合要求。

GetStringCritical 必须非常小心使用。你必须确保不分配新对象和任何阻塞系统的操作，以避免发生死锁。如下，因调用 fprintf，该 c 函数要执行 IO 操作，所以是不安全的。

```
/* This is not safe! */
const char *c_str = (*env)->GetStringCritical(env, j_str, 0);
if (c_str == NULL) {
  ... /* error handling */
}
fprintf(fd, "%s\n", c_str);
(*env)->ReleaseStringCritical(env, j_str, c_str);
```

上述代码，不安全的原因：当前线程执行了 GetStringCritical 后将禁用 GC. 假设，T 线程正等待从 fd 读取数据. 进一步假设，调用 fprintf 时使用的系统缓存将等待 T 读取完毕后设置. 我们制造了一个死锁情景：如果 T 在读取数据时有内存分配需求，可能使

JavaVM 执行 GC. 而此时的 GC 请求将被阻塞，直到当前线程执行 ReleaseStringCritical，不幸的时，这个操作必须等 fprintf 调用完毕后才会执行。此时，死锁发生。

所以，当你调用 Get/RleaseStringCritical 要时刻警惕死锁。

3.3 Accessing Arrays

JNI 对每种数据类型的数组都有对应的函数。

```
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 Accessing Arrays in C

如下直接操作数组是错误的：

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

如下操作正确：

```
JNIEXPORT jint JNICALL
```

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

JNI 中数组的基类为 jarray，其他如 jintArray 都继承自 jarray。

3.3.2 Accessing Arrays of Primitive Types

上节示例中，使用 GetIntArrayRegion 拷贝数组内容到 buf 中，这里没有做越界异常检测，因为知道数组有 10 个，参数 3 为待拷贝数组的起始位置，参数 4 为拷贝元素的个数。

JNI 支持 SetIntArrayRegion 允许重新设置数组一个区域的值，其他基本类型(boolean, short, 和 float)也有对应的支持。

JNI 支持通过 Get/Release<Type>ArrayElemetns 返回 Java 数组的一个拷贝(实现优良的 VM，会返回指向 Java 数组的一个直接的指针，并标记该内存区域，不允许被 GC)。

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

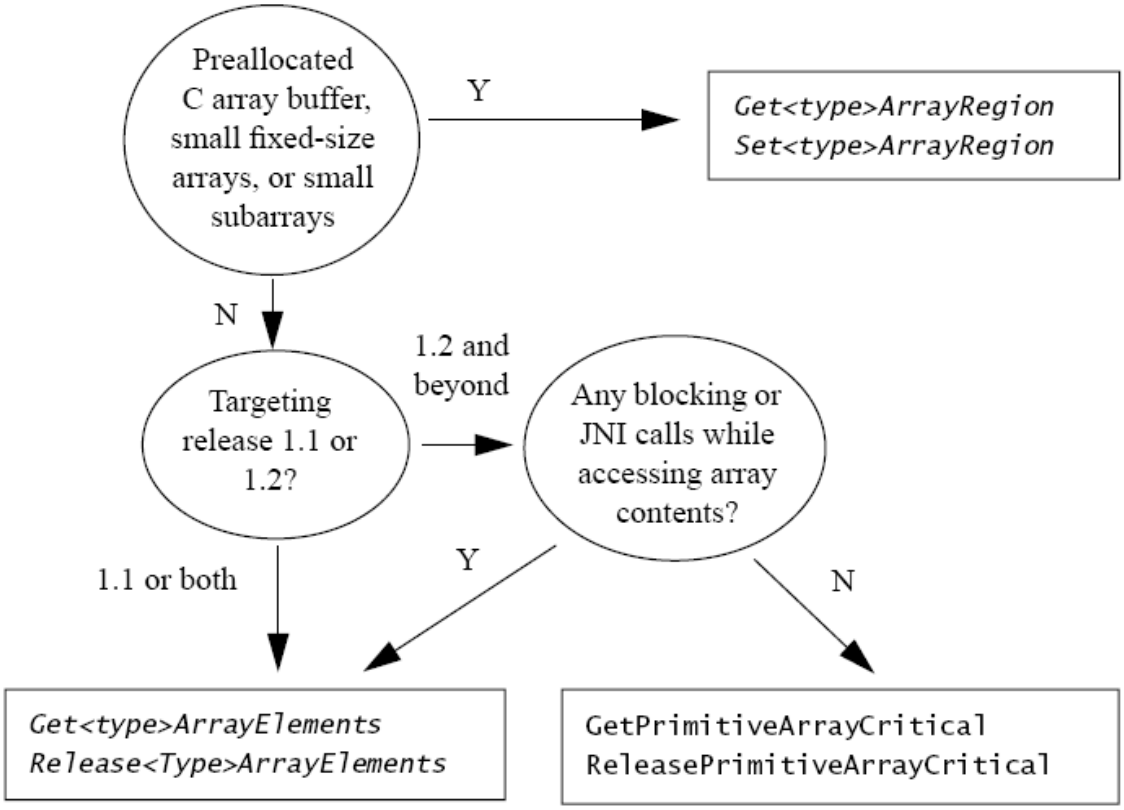
GetArrayLength 返回数组元素个数。

Java 2 SDK release 1.2 支持 Get/ReleasePrimitiveArrayCritical，该套函数的使用原则与上述 String 部分相同。

3.3.3 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 preallocated C 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`	Obtains 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 Choosing among the Primitive Array Functions



使用原则，与上述 String 部分相同，请阅读原文或回顾前面的内容。

3.3.5 Accessing Arrays of Objects

对于对象数组的访问，使用 Get/SetObjectArrayElement，对象数组只提供针对数组的每个元素的 Get/Set，不提供类似 Region 的区域性操作。

如下，二维数组示例，Java 部分

```
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]);
      }
      System.out.println();
    }
  }

  static {
    System.loadLibrary("ObjectArrayTest");
  }
}
```

JNI 部分：

```
JNIEXPORT jobjectArray JNICALL
Java_ObjectArrayTest_initInt2DArray(JNIEnv *env, jclass cls, int
size)
{
  jobjectArray result;
  int i;
  jclass intArrCls = (*env)->FindClass(env, "[I");

  if (intArrCls == NULL) {
    return NULL; /* exception thrown */
  }

  result = (*env)->NewObjectArray(env, size, intArrCls, NULL);
  if (result == NULL) {
    return NULL; /* out of memory error thrown */
  }

  for (i = 0; i < size; i++) {
    jint tmp[256]; /* make sure it is large enough! */
    int j;
```

```
    jintArray iarr = (*env)->NewIntArray(env, size);
    if (iarr == NULL) {
      return NULL; /* out of memory error thrown */
    }
    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 FindClass 获得一个一维 int 数组. "[I"作为 JNI 类描述符等价于 Java int[]声明。FindClass 当装载类失败，返回 NULL(可能是没找到类或内存不足)。

译注:

类描述符，也可以叫做"类签名"。签名的作用：为了准确描述一件事物. Java Vm 定义了类签名，方法签名；其中方法签名是为了支持方法重载。

FindClass 返回 NULL 的原因：

- 提供了错误的类描述符
- 无法在当前 ClassLoader 上下文中找到类

解决办法：

- 认真检查类描述符是否正确
- 以"/"作为包分隔符，即类描述符的形式为"xxx/xxx/xxx"，而非"xxx.xxx.xxx"，也可简单记忆为"/"用在本地形式(或虚拟机)中；"."分隔符，用在 Java(Java Programming Language)环境中；并且类描述符末尾没有".java"，如 FindClass("java/lang/String")而非 FindClass("java/lang/String.java")
- 构造 ClassLoader，并利用 Class.forName(String name, boolean initialize, ClassLoader loader)装载类

其中第三个解决办法比较复杂，涉及到 Java 的双亲委派模型，类与对象相容性判定等问题，将有专门文章阐述。

然后调用 NewObjectArray 分配一个对象数组。注意，"基本类型数组"这是个整体的概念，它是一个对象。后面我们要填充它。

注意，DeleteLocalRef 是释放局部对象引用。

译注:

Java 中有许多引用的概念，我们只关心 GlobalRef 和 LocalRef 两种。JNI 编程很复杂，建议不要引入更多复杂的东西，正确、高效的实现功能就可以了。比如对引用来说，最好

不要在JNI中考虑：虚引用和影子引用等复杂的东西。

GlobalRef: 当你需要在JNI层维护一个Java对象的引用，而避免该对象被垃圾回收时，使用NewGlobalRef告诉VM不要回收此对象，当本地代码最终结束该对象的引用时，用DeleteGlobalRef释放之。

LocalRef: 每个被创建的Java对象，首先会被加入一个LocalRef Table，这个Table大小是有限的，当超出限制，VM会报LocalRef Overflow Exception，然后崩溃. 这个问题是JNI编程中经常碰到的问题，请引起高度警惕，在JNI中及时通过DeleteLocalRef释放对象的LocalRef. 又，JNI中提供了一套函数：Push/PopLocalFrame，因为LocalRef Table大小是固定的，这套函数只是执行类似函数调用时，执行的压栈操作，在LocalRef Table中预留一部分供当前函数使用，当你在JNI中产生大量对象时，虚拟机仍然会因LocalRef Overflow Exception崩溃，所以使用该套函数你要对LocalRef使用量有准确估计。

CHAPTER 4

Fields and Methods

本章介绍如何访问对象成员，如何从本地代码调用 Java 方法，即以 callback 方式从本地代码调用 Java 代码；最后介绍一些优化技术。

4.1 Accessing Fields

Java 语言支持两种成员(field)：(static)静态成员和实例成员. 在 JNI 获取和赋值成员的方法是不同的.

译者:

Java 层的 field 和 method，不管它是 public，还是 package、private 和 protected，从 JNI 都可以访问到，Java 面向语言的封装性不见了。

```
Java:
class InstanceFieldAccess {
  private String s;
  private native void accessField();
  public static 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("InstanceFieldAccess");
  }
}

JNI:
JNIEXPORT void JNICALL
Java_InstanceFieldAccess_accessField(JNIEnv *env, jobject obj)
{
  jfieldID fid; /* store the field ID */
  jstring jstr;
  const char *str;
  /* Get a reference to obj's class */
```

```
  jclass cls = (*env)->GetObjectClass(env, obj);
  printf("In C:\n");
  /* Look for the instance field s in cls */
  fid = (*env)->GetFieldID(env, cls, "s", "Ljava/lang/String;");

  if (fid == NULL) {
    return; /* failed to find the field */
  }
  /* Read the instance field s */
  jstr = (*env)->GetObjectField(env, obj, fid);
  str = (*env)->GetStringUTFChars(env, jstr, NULL);
  if (str == NULL) {
    return; /* out of memory */
  }
  printf(" c.s = \"%s\"\n", str);
  (*env)->ReleaseStringUTFChars(env, jstr, str);
  /* Create a new string and overwrite the instance field */
  jstr = (*env)->NewStringUTF(env, "123");
  if (jstr == NULL) {
    return; /* out of memory */
  }
  (*env)->SetObjectField(env, obj, fid, jstr);
}
```

输出：

In C:

c.s = "abc"

In Java:

c.s = "123"

4.1.1 Procedure for Accessing an Instance Field

访问对象成员分两步，首先通过 GetFieldID 得到对象成员 ID，如下：

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

示例代码，通过 GetObjectClass 从 obj 对象得到 cls.

这时，通过在对象上调用下述方法获得成员的值：

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

示例中要得到的是一个对象类型，所以用 GetObjectField. 此外 JNI 还提供 Get/SetIntField，Get/SetFloatField 访问不同类型成员。

译者：

通过 JNI 方法访问对象的成员，JNI 对应的函数命名非常有规律，即 Get/Set<Return Value Type>Field。

4.1.2 Field Descriptors

此章主要讲述签名问题，较繁琐，可以总结如下：

Type Signature	Java Type
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double
L fully-qualified-class ;	fully-qualified-class
[type	type[]
(arg-types) ret-type	method type

如下 Java 方法:

```
long f (int n, String s, int[] arr);
```

```
signature:  "(ILjava/lang/String;[I)J"
```

签名是一种用参数个数和类型区分同名方法的手段，即解决方法重载问题。

其中要特别注意的是：

1. 类描述符开头的'L'与结尾的';'必须要有
2. 数组描述符，开头的'['必须有.
3. 方法描述符规则: "(各参数描述符)返回值描述符"，其中参数描述符间没有任何分隔符号

描述符很重要，请烂熟于心. 写 JNI，对于错误的签名一定要特别敏感，此时编译器帮不上忙，执行 make 前仔细检查你的代码。

4.1.3 Accessing Static Fields

静态成员访问与实例成员类似。

```
Java:
```

```
class StaticFielcdAccess {
  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");
  }
}
```

JNI:

```
JNIEXPORT void JNICALL
Java_StaticFieldAccess_accessField(JNIEnv *env, jobject obj)
{
  jfieldID fid; /* store the field ID */
  jint si;
  /* Get a reference to obj's class */
  jclass cls = (*env)->GetObjectClass(env, obj);
  printf("In C:\n");
  /* Look for the static field si in cls */
  fid = (*env)->GetStaticFieldID(env, cls, "si", "I");
  if (fid == NULL) {
    return; /* field not found */
  }
  /* Access the static field si */
  si = (*env)->GetStaticIntField(env, cls, fid);
  printf(" StaticFieldAccess.si = %d\n", si);
  (*env)->SetStaticIntField(env, cls, fid, 200);
}
```

输出：

In C:

StaticFieldAccess.si = 100

In Java:

StaticFieldAccess.si = 200

请阅读上述代码，不再叙述。

4.2 Calling Methods

Java 中有三类方法：实例方法、静态方法和构造方法。

```
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");
  }
}


JNIEXPORT void JNICALL
Java_InstanceMethodCall_nativeMethod(JNIEnv *env, jobject obj)
{
  jclass cls = (*env)->GetObjectClass(env, obj);
  jmethodID mid = (*env)->GetMethodID(env, cls, "callback", "()V");
  if (mid == NULL) {
    return; /* method not found */
  }
  printf("In C\n");
  (*env)->CallVoidMethod(env, obj, mid);
}
```

输出：

```
In C
In Java
```

4.2.1 Calling Instance Methods

如上节示例，回调 Java 方法分两步：

- 首先通过 GetMethodID 在给定类中查询方法. 查询基于方法名称和签名
- 本地方法调用 CallVoidMethod，该方法表明被调 Java 方法的返回值为 void

译者:
从 JNI 调用实例方法命名规则：Call<Return Value Type>Method

4.2.2 Formaing the Method Descriptor

一个方法描述(签名)由各参数类型签名和返回值签名构成. 参数签名在前，并用小括号括起. 具体描述请参照上文 4.1.2

4.2.3 Calling Static Methods

同实例方法，回调 Java 静态方法分两步：

- 首先通过 GetStaticMethodID 在给定类中查找方法
- 通过 CallStatic<ReturnValueType>Method 调用

静态方法与实例方法的不同，前者传入参数为 jclass，后者为 jobject

4.2.4 Calling Instance Methods of a Superclass

调用被子类覆盖的父类方法: JNI 支持用 CallNonvirtual<Type>Method 满足这类需求:

- GetMethodID 获得 method ID
- 调用 CallNonvirtualVoidMethod, CallNonvirtualBooleanMethod

上述，等价于如下 Java 语言的方式：

super.f();

CallNonvirtualVoidMethod 可以调用构造函数

4.3 Invoking Constructors

你可以像调用实例方法一样，调用构造方法，只是此时构造函数的名称叫做"<init>". 如下构造 java.lang.String 对象(JNI 为了方便有个对应的 NewString 做下面所有工作，这里只是做示例展示):

```
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; /* exception thrown */
  }
  /* Get the method ID for the String(char[]) constructor */
  cid = (*env)->GetMethodID(env, stringClass, "<init>", "([C)V");
  if (cid == NULL) {
    return NULL; /* exception thrown */
  }
  /* Create a char[] that holds the string characters */
  elemArr = (*env)->NewCharArray(env, len);
  if (elemArr == NULL) {
    return NULL; /* exception thrown */
  }
  (*env)->SetCharArrayRegion(env, elemArr, 0, len, chars);
  /* Construct a java.lang.String object */
  result = (*env)->NewObject(env, stringClass, cid, elemArr);
  /* Free local references */
  (*env)->DeleteLocalRef(env, elemArr);
  (*env)->DeleteLocalRef(env, stringClass);
  return result;
}
```

首先，FindClass 找到 java.lang.String 的 jclass. 接下来，用 GetMethodID 找到构造函数 String(char[] chars)的 MethodID. 此时用 NewCharArray 分配一个 Char 数组对象。NewObject 调用构造函数。

用 DeleteLocalRef 释放资源。

注意 NewString 是个常用函数，所以在 JNI 中直接被支持了，并且该函数的实现要比我们实现的高效。

也可使用 CallNonvirtualVoidMehtod 调用构造函数. 如下代码:

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

可被替换为：

```
  result = (*env)->AllocObject(env, stringClass);
  if (result) {
    (*env)->CallNonvirtualVoidMethod(env, result, stringClass, cid,
elemArr);
    /* we need to check for possible exceptions */
    if ((*env)->ExceptionCheck(env)) {
      (*env)->DeleteLocalRef(env, result);
      result = NULL;
    }
  }
```

AllocObject 创建一个未初始化的对象，该函数必须在每个对象上被调用一次而且只能是一次。

有时你会发现先创建未初始化对象再调用构造函数的方法是有用的。

4.4 Caching Field and Method IDs

获得 field 与 method IDs，需要做基于名称和签名的符号表查询，此过程可以被优化。

基本想法是：只在第一次使用 ID 时查询，然后缓存该值. 有两个缓存时机：首次使用和初始化类时。

4.4.1 Caching at the Point of Use

如下，首次使用时，缓存的局部静态变量中，避免每次调用计算。

```
JNIEXPORT void JNICALL
Java_InstanceFieldAccess_accessField(JNIEnv *env, jobject obj)
{
  static jfieldID fid_s = NULL; /* cached field ID for s */
  jclass cls = (*env)->GetObjectClass(env, obj);
  jstring jstr;
  const char *str;
  if (fid_s == NULL) {
    fid_s = (*env)->GetFieldID(env, cls, "s",
"Ljava/lang/String;");
    if (fid_s == NULL) {
      return; /* exception already thrown */
    }
  }
  printf("In C:\n");
  jstr = (*env)->GetObjectField(env, obj, fid_s);
  str = (*env)->GetStringUTFChars(env, jstr, NULL);
  if (str == NULL) {
    return; /* out of memory */
  }
  printf(" c.s = \"%s\"\n", str);
  (*env)->ReleaseStringUTFChars(env, jstr, str);
  jstr = (*env)->NewStringUTF(env, "123");
  if (jstr == NULL) {
    return; /* out of memory */
  }
```

```
  (*env)->SetObjectField(env, obj, fid_s, jstr);
}
```

如上，静态变量 fid_s 保存了 InstanceFieldAccess.s 的 filed ID。初始化阶段静态变量被赋值为 NULL。第一调用 InstanceFieldAccess.accessField 时，缓存 fieldID 以待后用。

你可能会发现上述代码有个竞争条件，当多个线程同时访问此函数时，可能会同时计算一个 field ID. 没关系，此处的竞争是无害的，因为即使在多个线程中同时计算该 field ID，各线程中的计算结果都是一样的。

构造函数的 MethodID 也可被缓存，如下：

```
jstring
MyNewString(JNIEnv *env, jchar *chars, jint len)
{
    jclass stringClass;
    jcharArray elemArr;
    static jmethodID cid = NULL;
    jstring result;
    stringClass = (*env)->FindClass(env, "java/lang/String");
    if (stringClass == NULL) {
        return NULL; /* exception thrown */
    }
    /* Note that cid is a static variable */
    if (cid == NULL) {
        /* Get the method ID for the String constructor */
        cid = (*env)->GetMethodID(env, stringClass,
                "<init>", "([C)V");
        if (cid == NULL) {
            return NULL; /* exception thrown */
        }
    }
    /* Create a char[] that holds the string characters */
    elemArr = (*env)->NewCharArray(env, len);
    if (elemArr == NULL) {
        return NULL; /* exception thrown */
    }
    (*env)->SetCharArrayRegion(env, elemArr, 0, len, chars);
    /* Construct a java.lang.String object */
    result = (*env)->NewObject(env, stringClass, cid, elemArr);
    /* Free local references */
    (*env)->DeleteLocalRef(env, elemArr);
    (*env)->DeleteLocalRef(env, stringClass);
    return result;
}
```

4.4.2 Caching in the Defining Class's Initializer

上述第一次使用缓存的方式，每次都有与 NULL 的判断，并且可能有一个无害的竞争条件。

而初始化类时，同时初始化 JNI 层对该类成员的缓存，可以弥补上述缺憾，如下 initIDs:

Java 代码：

```
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[]) {
        InstanceMethodCall c = new InstanceMethodCall();
        c.nativeMethod();
    }
    static {
        System.loadLibrary("InstanceMethodCall");
        initIDs();
    }
}
```

JNI 代码：

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

译注：

还可以改进上述缓存策略的初始化时机，第一种方法的缺陷文中已经提了，而第二种需要在 Java 代码主动调用 JNI 作缓存。

改进：可以在你的项目中加一套 Hash 表，封装 FindClass，GetMethodID，GetFieldID 等函数，查询的所有操作，都对 Hash 表操作，如首次 FindClass 一个类，这时可以把一个类的所有成员缓存到 Hash 表中，用名字+签名做键值。

译者所做项目引入了这个优化，项目的执行效率有 100 倍的提高；当时还做过两个权衡：

1. 用一个 Hash 表，还是每个类一个 Hash 表
2. 首次 FindClass 类时，一次缓存所有的成员，还是用时缓存

最终做的选择是：为了降低冲突，每个类一个 Hash 表，并且一次缓存一个类的所有成员。

当然，没有尽善尽美的优化策略，我们做到这个层次，已经达到预期目标，没有继续深入。

4.4.3 Comparison between the Two Approaches to Caching IDs

在对 Java 源码无改动权时使用时缓存是一种合理的解决方案. 但有许多弊端：

- 无害的竞争条件和重复与 NULL 比较
- 在类没被卸载时，MethodID 和 FieldID 一直有效. 所以你必须保证：当你的 JNI 代码依赖这些缓存值的声明周期内，该类不会被卸载。而与另一种优化策略，连同类的初始化缓存 Method/Field ID，每当类再次被装载，缓存值会被更新

所以，有条件的话，更安全的优化策略是：连同类的初始化缓存 Method/Field ID。

4.5 Performance of JNI Field and Method Operations

在学习了如何缓存 field 和 method ID 的优化技术后，你可能会想：影响 JNI 回调性能的关键性因素是什么？在效率方面，JNI/Java 与 Java/JNI 和 Java/Java 间对比，是怎样的？

这要看具体 VM 实现的 JNI 效率. 很难给出一个普适的性能关键指标. 取而代之，我们将分析在访问类成员时的固有性能损失。

首先比较 Java/native 和 Java/Java，前者因下述原因可能会比后者慢：

- Java/native 与 Java/Java 的调用约定不同. 所以，VM 必须在调用前，对参数和调用栈做特殊准备
- 常用的优化技术是内联. 相比 Java/Java 调用，Java/native 创建内联方法很难

粗略估计：执行一个 Java/native 调用要比 Java/Java 调用慢 2-3 倍. 也可能有一些 VM 实现，Java/native 调用性能与 Java/Java 相当。(此种虚拟机，Java/native 使用 Java/Java 相同的调用约定)。

native/Java 调用效率可能与 Java/Java 有 10 倍的差距，因为 VM 一般不会做 Callback 的优化。

对于 field 的访问，将没什么不同，只是通过 JNI 访问某对象结构中某个位置的值。

译注：
上述只是学术考虑. 用好缓存的优化策略，完全可以让项目工作的绝对出色。

CHAPTER 5

Local and Global References

JNI 把 instance 和 array 类型的指针对外公布为 opaque reference. 本地代码不直接操作指针，而是通过 JNI 函数，所以本地代码不用关心内存布局. 关于 reference，这里还有更丰富的东西有待介绍：

- JNI 支持三种类型的 opaque reference：local references, global references, 和 weak global references
- Local 和 Global 引用有不同的生命周期. Local Ref 在 native method 执行完毕后被 JavaVM 自动释放，而 GlobalRef,WeakRef 在程序员主动释放前一直有效
- 各种引用都有使用范围. 如 LocalRef 只能在当前线程的 native method 中使用

本章将详细讲述不同类型 Ref 的使用方法，正确管理 JNI 引用是程序健壮、空间占用少的关键。

5.1 Local and Global References

LocalRef 与 GlobalRef 的差异，将用几个示例说明：

大部分 JNI 函数都会创建 LocalRef，如 NewObject 创建一个实例，并返回一个指向该实例的 LocalRef。

LocalRef 只在本线程的 native method 中有效. 一但 native method 返回，LocalRef 将被释放。不要缓存一个 LocalRef，并企图在下次进入该 JNI 方法时使用，如下:

```
/* This code is illegal */
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/lang/String");
        if (stringClass == NULL) {
            return NULL; /* exception thrown */
        }
    }
```

```
    /* It is wrong to use the cached stringClass here,
       because it may be invalid. */
    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;
}
```

上述代码，企图重复使用FindClass(env, "java/lang/String")的返回值，这种方式不对，因为FindClass返回的是一个LocalRef. 请设想以下代码:

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

如下，两次调用f这个本地方法.

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

第一次调用可能正确，而第二次将引用一个无效位置，因为第二次企图使用存在静态变量中的LocalRef。

有两种方式让LocalRef无效，一，native method返回，JavaVM自动释放LocalRef；二，用DeleteLocalRef主动释放。

既然LocalRef会被JavaVM自动释放，为什么还要有DeleteLocalRef？因为LocalRef是阻止引用被GC，但当你在本地代码中操作大量对象时，而LocalRefTable又是有限的，及时调用DeleteLocalRef，会释放LocalRef在LocalRefTable中所占位置并使对象及时得到回收。

LocalRef只在创建该对象的线程中有效，企图把LocalRef存到全局变量中供其他线程使用的做法是错误的。

译注:

注意这里的提到的native method返回，返回是指回到Java层，如果从一个本地函数返

回到另一个本地函数，LocalRef 是有效的。

5.1.2 Global References

释放 GlobalRef 前，你可以在多个本地方法调用过程和多线程中使用 GlobalRef 所引对象。与 LocalRef 类似，GlobalRef 的作用：防止对象被 GC(garbage collected, 垃圾回收)。

GlobalRef 与 LocalRef 不同的是，LocalRef 一般自动创建(返回值为 jobject/jclass 等 JNI 函数)，而 GlobalRef 必须通过 NewGlobalRef 由程序员主动创建。如下：

```
  /* This code is OK */
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; /* exception thrown */
        }
        /* Create a global reference */
        stringClass = (*env)->NewGlobalRef(env, localRefCls);
        /* The local reference is no longer useful */
        (*env)->DeleteLocalRef(env, localRefCls);
        /* Is the global reference created successfully? */
        if (stringClass == NULL) {
            return NULL; /* out of memory exception thrown */
        }
    }

    ...
}
```

该例做了修改，当 stringClass 为 NULL 时，我们创建了 java.lang.String 的 GlboalRef，并删除了对应的 LocalRef，以待下次再进入此方法时，使用 stringClass。

5.1.3 Weak Global References

Weak Global Ref 用 NewGlobalWeakRef 于 DeleteGlobalWeakRef 进行创建和删除，多个本地方法调用过程中和多线程上下文中使用的特性与 GlobalRef 相同，但该类型的引用不保证不被 GC。

前述示例 MyNewString 中，对 java.lang.String 声明 GlobalRef 或 GlobalWeakRef 效果相同，因为 java.lang.String 是一个系统类不会被 GC。

Weak Global Ref 使用在允许被 GC 的场合，如内存紧张时。

```
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; /* can't find class */
        }
        myCls2 = NewWeakGlobalRef(env, myCls2Local);
        if (myCls2 == NULL) {
            return; /* out of memory */
        }
    }
    ... /* use myCls2 */
}
```

我们假设，MyCls 与 MyCls2 有同样的生命周期(并被同样的 Class Loader 装载)，类似 MyCls 被卸载而 MyCls2 没被卸载的情况不考虑。如果发生这种情况，我们还需要检测 myCls2 是否还执行的对象仍然有效。

5.1.4 Comparing Reference

有两个对象，用如下方法比较相容性:

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

如果相容，返回 JNI_TRUE, 否则返回 JNI_FALSE。

与 NULL 的比较，LocalRef 与 GlobalRef 语义显然，前提是释放了两个引用，程序员重新为相应变量做了 NULL 初始化。

但对于 Weak Global Ref 来说，需要使用下述代码判定：

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

因为，对于一个 Weak Global Ref 来说可能指向已经被 GC 的无效对象。

译注:
上述的判断，都是假设所有的类和对象都是在一个 Class Loader 下被装载的. 关于 ClassLoader 的议题后有专门文章.

5.2 Freeing Reference

每个 JNI 引用都会引用表中的一个位置. 作为一个 JNI 程序员，你应该清楚程序某阶段中使用的引用数量。 如 LocalRef，如果你疏于 DeleteLocalRef 的话，在 JavaVM 运行限制内你的应用程序工作正常，在极端情况会崩溃。

5.2.1 Freeing Local References

译注:
本章没有翻译.。
本章讲了很多关于 LocalRef 的释放原则，译者认为：考虑何时释放/何时不释放的问题，不如认真审查代码，严堵每个泄露环节，尽最大努力提高程序的稳定性。 就像内存分配一样，虽然进程结束后，OS 自动释放该进程分配的所有内存，但对于期望长期稳定运行的系统来说，我们希望杜绝内存泄露。

5.2.2 Managing Local References in Java 2 SDK Release 1.2

译注:
本章没有翻译。
由于 LocalRef Table 大小是固定的，这套函数只是执行类似函数调用时，执行的压栈操作，并在执行 PopLocalFrame 后执行类似退栈操作, 在 LocalRef Table 中预留一部分供当前函数使用，当你在 JNI 中产生大量对象时，虚拟机仍然会因 LocalRef Overflow Exception 崩溃。
具体原则仍如上述，严堵每个泄露环节；如果你能准确估计 LocalRef 用量，可以使用 Push/PopLocalFrame。

5.2.3 Freeing Global References

当不再使用 GlobalRef 所指对象，及时调用 DeleteGlobalRef 释放对象. 否则，GC 将不回收该对象。

对于 DeleteWeakGlobalRef 来说，不使用 WeakGlobalRef 时，也要及时释放，因为即使 GC 会回收该对象内容，WeakGlobalRef 在 Table 中的位置还占用着，即和尚都跑了，庙还在。

译注:
综上，不管何种类型引用，在不使用所引用对象后，及时调用对应指针类型的释放函数。

5.3 Rules for Managing References

现在我们归纳一下管理 JNI 引用的原则. 看看如何减少内存使用、有效使用对象。

有两类本地函数：功能函数和工具函数。

当写 native method 的实现时，要认真处理循环中产生的 LocalRef. VM 规范中规定每个本地方法至少要支持 16 个 LocalRef 供自由使用并在本地方法返回后回收. 本地方法绝对不能滥用 GlobalRef 和 WeakGlobalRef，因为此类型引用不会被自动回收。

工具函数，对LocalRef的使用更要提起警惕，因为该类函数调用上下文不确定，而且会被重复调用，每个代码路径都要保证不存在LocalRef泄露。

由于某些缓存机制，可以在工具函数中创建GlobalRef, WeakGlobalRef。

当工具函数返回对象时，要严格遵守引用约定，让调用者在决定是否释放时能作出准确判断，如下：

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

  ... /* process infoString */

  ???
  /*
   * we need to call DeleteLocalRef, DeleteGlobalRef,
   * or DeleteWeakGlobalRef depending on the type of
   * reference returned by GetInfoString.
   */
}
```

JNI方法NewLocalRef总保证返回一个LocalRef，如下：

```
jstring
MyNewString(JNIEnv *env, jchar *chars, jint len)
{
    static jstring result;
    /* wstrncmp compares two Unicode strings */
    if (wstrncmp("CommonString", chars, len) == 0) {
        /* refers to the global ref caching "CommonString" */
        static jstring cachedString = NULL;
        if (cachedString == NULL) {
            /* create cachedString for the first time */
            jstring cachedStringLocal = ... ;
            /* cache the result in a global reference */
            cachedString =
                (*env)->NewGlobalRef(env, cachedStringLocal);
        }
        return (*env)->NewLocalRef(env, cachedString);
    }

    ...
    /* create the string as a local reference and store in
       result as a local reference */
```

```
    return result;
}
```

Push/PopLocalFrame 常被用来管理 LocalRef. 在进入本地方法时，调用一次 PushLocalFrame，并在本地方法结束时调用 PopLocalFrame. 此对方法执行效率非常高，建议使用这对方法。

译注：

你只要对当前上下文内使用的对象数量有准确估计，建议使用这对方法，在这对方法间，不必调用 DeleteLocalRef，只要该上下文结尾处调用 PopLocalFrame 会一次性释放所有 LocalRef。

一定保证该上下文出口只有一个，或每个 return 语句都做严格检查是否调用了 PopLocalFrame。

```
jobject f(JNIEnv *env, ...)
{
    jobject result;
    if ((*env)->PushLocalFrame(env, 10) < 0) {
        /* frame not pushed, no PopLocalFrame needed */
        return NULL;
    }
    ...
    result = ...;
    if (...) {
        /* remember to pop local frame before return */
        result = (*env)->PopLocalFrame(env, result);
        return result;
    }
    ...
    result = (*env)->PopLocalFrame(env, result);
    /* normal return */
    return result;
}
```

忘记调用 PopLocalFrame 可能会使 VM 崩溃。

CHAPTER 6

Exceptions

我们已经碰到在调用 JNI 方法时出现异常的情况. 本章将介绍如何检查并处理异常。

本章只关注在调用 JNI 方法或 Java 方法时出现异常的处理办法(Java 异常)，不涉及本地代码本身(如本地代码中的除 0 错)或调用系统函数出现异常的处理方法。

6.1.1 Caching and Throwing Exceptions in Native Code

如下 Java 代码展示如何声明 JNI 可能抛出的异常。

```
class CatchThrow {
    private native void doit()
        throws IllegalArgumentException;
    private void callback() throws NullPointerException {
        throw new NullPointerException("CatchThrow.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");
    }
}
```

JNI 代码：

```
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) {
        /* We don't do much with the exception, except that
           we print a debug message for it, clear it, and
           throw a new exception. */
        jclass newExcCls;
        (*env)->ExceptionDescribe(env);
        (*env)->ExceptionClear(env);
        newExcCls = (*env)->FindClass(env,
                "java/lang/IllegalArgumentException");
        if (newExcCls == NULL) {
            /* Unable to find the exception class, give up. */
            return;
        }
        (*env)->ThrowNew(env, newExcCls, "thrown from C code");
    }
}
```

输出：

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

callback 方法抛出 NullPointerException. 当 CallVoidMethod 把控制权返回给本地代码，本地代码调用 ExceptionOccurred 检查是否有异常发生. 我们的处理方式是，当有异常发生，调用 ExceptionDescribe 打印调用堆栈，然后用 ExceptionClear 清空异常,最后重新抛出 IllegalArgumentException。

译者：

ExceptionOccurred 返回一个 jobject，注意结束处理时调用 DeleteLocalRef 删除该返回值。JNI 中还有一个 ExceptionCheck，只是返回一个 jboolean 的布尔值，更适合检查异常是否发生。

在 JNI 中产生的异常(通过调用 ThrowNew)，与 Java 语言中异常发生的行为不同，JNI 中当前代码路径不会立即改变。在 Java 中发生异常，VM 自动把控制权转向 try/catch 中匹配

的异常类型处理块。VM 首先清空异常队列，然后执行异常处理块。相反，JNI 中必须显式处理 VM 的处理方式。

6.1.2 A Utility Function

JNI 中抛异常很经典：找异常类，调用 ThrowNew 抛出之；所以，可以写一个工具函数。

```
void
JNU_ThrowByName(JNIEnv *env, const char *name, const char *msg)
{
    jclass cls = (*env)->FindClass(env, name);
    /* if cls is NULL, an exception has already been thrown */
    if (cls != NULL) {
        (*env)->ThrowNew(env, cls, msg);
    }
    /* free the local ref */
    (*env)->DeleteLocalRef(env, cls);
}
```

本书中，JNU 前缀表示 JNI Utilities. JNU_ThrowByName 首先通过 FindClass 找到异常类。如果 FindClass 找类失败，将返回 NULL，并抛出 NoClassDefFoundError 异常。此情况，JNU_ThrowByName 将保留该异常，然后返回. 如果 FindClass 成功, 将调用 ThrowNew 抛出异常。所以不管哪种情况，调用该函数后，当前的 JNIEnv 环境里总有个异常。

6.2 Proper Exception Handling

JNI 程序员应对所有可能的异常做处理，这个要求虽然苛刻，但这是健壮软件的保证。

6.2.1 Checking for Exception

有两种方式检查是否有异常发生。

1. 大多数 JNI 函数用显式方式表明当前线程是否有异常发生。

下述代码判断 GetFieldID 返回是否为 NULL 以检查是否发生异常：

```
/* a class in the Java programming language */
public class Window {
    long handle;
    int length;
    int width;
    static native void initIDs();
    static {
```

```
        initIDs();
    }
}
```

JNI 代码：

```
/* C code that implements Window.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) { /* important check. */
        return; /* error occurred. */
    }
    FID_Window_length =
        (*env)->GetFieldID(env, classWindow, "length", "I");
    if (FID_Window_length == NULL) { /* important check. */
        return; /* error occurred. */
    }
    FID_Window_width =
        (*env)->GetFieldID(env, classWindow, "width", "I");
    /* no checks necessary; we are about to return anyway */
}
```

2. 如果返回值不能表明是否有异常发生，需要用 JNI 提供的 ExceptionOccurred 检查但前线程是否有未处理异常。

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

JNI 代码：

```
/* Native code that calls Fraction.floor. Assume method ID
   MID_Fraction_floor has been initialized elsewhere. */
```

```
void f(JNIEnv *env, jobject fraction)
{
    jint floor = (*env)->CallIntMethod(env, fraction,
           MID_Fraction_floor);
    /* important: check if an exception was raised */
    if ((*env)->ExceptionCheck(env)) {
        return;
    }
    ... /* use floor */
}
```

6.2.2 Handling Exception

本地代码以两种方式处理异常：

- 本地代码可以立即返回，并在调用者中处理异常
- 本地代码可以 ExceptionClear 清空异常，然后自己做重新抛出等策略

继续执行后续代码前，必须严格按着：检查->处理->清除的逻辑处理异常. 如果没有预先清空异常就调用一个 JNI 方法，行为不可预料。有一些函数可以在未清空异常前调用，但只局限于很少的几个，而且多是异常处理 JNI 函数。

在异常发生后，及时释放资源很重要，如下异常发生时对 String 的处理：

```
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 (...) { /* exception occurred */
        (*env)->ReleaseStringChars(env, jstr, cstr);
        return;
    }
    ...
        /* normal return */
    (*env)->ReleaseStringChars(env, jstr, cstr);
}
```

6.2.3 Exceptions in Utility Functions

写工具函数对异常的处理要很注意，一定是针对调用者的，即无累积异常行为，一般原则如下：

- 工具函数应该提供返回值告诉调用者释放发生异常，以简化调用者的处理
- 工具函数要注意处理LocalRef

如下示例：

```
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;
            /* skip over argument types to find out the
               return type */
            while (*p != ')') p++;
            /* skip ')' */
            p++;
            va_start(args, descriptor);
            switch (*p) {
            case 'V':
                (*env)->CallVoidMethodV(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.c = (*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");
        }
        va_end(args);
    }
    (*env)->DeleteLocalRef(env, clazz);
  }
  if (hasException) {
    *hasException = (*env)->ExceptionCheck(env);
  }
  return result;
}
```

JNU_CallMethodByName，通过对*hasException 指针赋值，来表明是否有异常发生。

CHAPTER 7

The Invocation Interface

本章介绍如何在你的应用中嵌入一个 JavaVM。JavaVM 被普遍的实现为一个库。本地应用可以链接该库，并通过 invocation interface 装载 JavaVM. 命令行的 java 命令，实现原理与上述类似。

7.1 Creating the Java Virtual Machcine

为演示如何通过 invocation interface 装载 JavaVM 并执行 Java 代码，如下示例:

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

C 代码:
#include <jni.h>
#define PATH_SEPARATOR ';' /* define it to be ':' on Solaris */
#define USER_CLASSPATH "." /* where Prog.class is */
main() {
    JNIEnv *env;
    JavaVM *jvm;
    jint res;
    jclass cls;
    jmethodID mid;
    jstring jstr;
    jclass stringClass;
    jobjectArray args;
#ifdef JNI_VERSION_1_2
    JavaVMInitArgs vm_args;
    JavaVMOption options[1];
    options[0].optionString =
        "-Djava.class.path=" USER_CLASSPATH;
    vm_args.version = 0x00010002;
    vm_args.options = options;
    vm_args.nOptions = 1;
    vm_args.ignoreUnrecognized = JNI_TRUE;
```

```
    /* Create the Java VM */
    res = JNI_CreateJavaVM(&jvm, (void**)&env, &vm_args);
#else
    JDK1_1InitArgs vm_args;
    char classpath[1024];
    vm_args.version = 0x00010001;
    JNI_GetDefaultJavaVMInitArgs(&vm_args);
    /* Append USER_CLASSPATH to the default system class path */
    sprintf(classpath, "%s%c%s",
           vm_args.classpath, PATH_SEPARATOR, USER_CLASSPATH);
    vm_args.classpath = classpath;
    /* Create the Java VM */
    res = JNI_CreateJavaVM(&jvm, &env, &vm_args);
#endif /* JNI_VERSION_1_2 */
    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 destroy;
    }
    jstr = (*env)->NewStringUTF(env, " from C!");
    if (jstr == NULL) {
        goto destroy;
    }
    stringClass = (*env)->FindClass(env, "java/lang/String");
    args = (*env)->NewObjectArray(env, 1, stringClass, jstr);
    if (args == NULL) {
        goto destroy;
    }
    (*env)->CallStaticVoidMethod(env, cls, mid, args);
destroy:
    if ((*env)->ExceptionOccurred(env)) {
        (*env)->ExceptionDescribe(env);
    }
    (*jvm)->DestroyJavaVM(jvm);
}
```

该代码用宏区分不同版本 JavaVM 处理参数的代码。

当为 1.2 release 版本 JavaVM，C 代码创建了 JavaVMInitArgs 结构. VM 初始化参数存在 JavaVMOption 数组中. 并设置 ignoreUnrecognized 为 JNI_TRUE 使虚拟机忽略不支持的参数。

当 JavaVM 初始化完毕, 调用 JNI_CreateJavaVM 装载 VM. 该函数填充了两个结构：

- 指向新建 JavaVM 的 jvm
- 指向当前线程的 JNIEnv

JNI_CreateJavaVM 调用完毕，一切准备工作都做好了，就可以调用 JNI 方法了。

最后，调用 DestroyJavaVM 卸载 JavaVM。

输出：

Hello World from C!

7.2 Linking Native Applications with the Java Virtual Machine

如何链接，取决于你想把本地代码与特定 VM 链接，还是与任意厂商的 VM 链接.

7.2.1 Linking with a Known Java Virtual Machine

当你打算与特定虚拟机链接时，可以直接指定 VM 库路径.

$cc -I<jni.h dir> -L<libjava.so dir> -lthread -ljava invoke.c

-lthread：使用本地线程支持

-ljava: 链接 Solaris 共享库

Win32 与 JDK1.1 release 链接：

$cl -I<jni.h dir> -MD invoke.c -link <javai.lib dir>\javai.lib

-MD: Win32 多线程 C 库

Win32 与 JDK1.2 release 链接：上述的库要换为 jvm.lib 或 jvm.dll

链接完毕，如果执行出错，Log 显示找不到库，请设置 LD_LIBRARY_PATH(类 Unix 平台)，或 PATH 环境变量(Win32)。

7.2.2 Linking with Unknown Java Virtual Machines

如果本地应用期望与多个版本/厂商 VM 共同工作，你就不能连接特定虚拟机。比如 JDK release 1.1 的 JavaVM 动态库为 javai.dll,而 1.2 的动态库为 jvm.dll。

我们可以使用执行期装载动态库的方式解决，如下：

```
/* Win32 version */
```

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

LoadLibrary 和 GetProcAddress 为 Win32 API 可以接受"jvm"(PATH 环境变量中定义全路径)或"C:\\jdk1.2\\jre\\bin\\classic\\jvm.dll"的绝对路径形式。

类 Unix 如下：

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

dlopen/dlsym 为动态链接库函数。

7.3 Attaching Native Threads

假设你有一个多线程 Web Server，每来一个请求，将起一个线程为之服务，并在每个线程中调用 JNI/Java 方法。

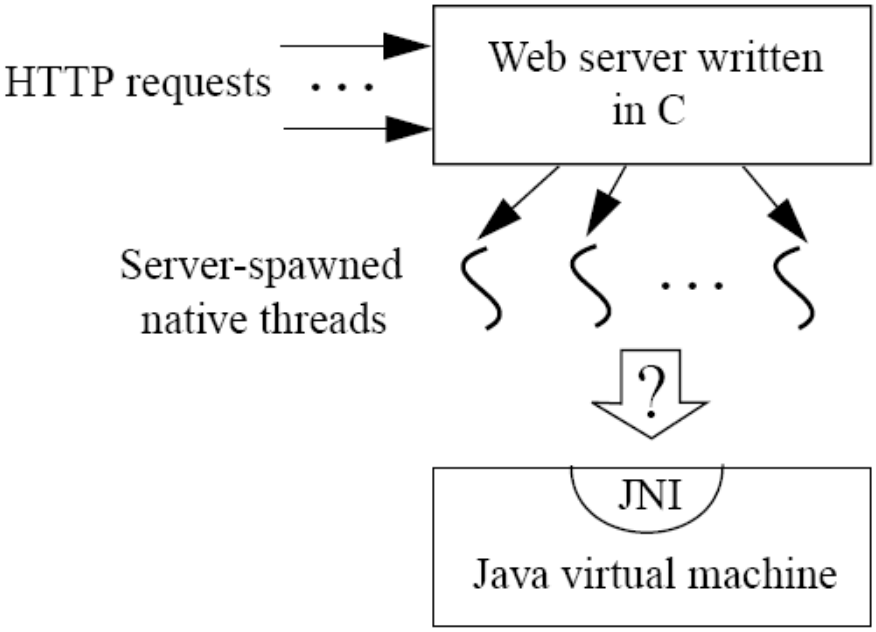


Figure 7.1 Embedding the Java virtual machine in a web server

派生的线程较 JavaVM 生命期要短，我们可以 attach 一个本地线程到 JavaVM，在线程结束时从 JavaVM detach 该线程。

译者:
attach/detach 至今也没有找到一个恰当的中文词描绘。

如下示例：

```
/* Note: This program only works on Win32 */
#include <windows.h>
#include <jni.h>
JavaVM *jvm; /* The virtual machine instance */
#define PATH_SEPARATOR ';'
#define USER_CLASSPATH "." /* where Prog.class is */
void thread_fun(void *arg)
{
    jint res;
    jclass cls;
    jmethodID mid;
    jstring jstr;
    jclass stringClass;
    jobjectArray args;
    JNIEnv *env;
    char buf[100];
    int threadNum = (int)arg;
    /* Pass NULL as the third argument */
```

```
#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");
        return;
    }
    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].optionString =
        "-Djava.class.path=" USER_CLASSPATH;
    vm_args.version = 0x00010002;
    vm_args.options = options;
    vm_args.nOptions = 1;
    vm_args.ignoreUnrecognized = TRUE;
    /* Create the Java VM */
    res = JNI_CreateJavaVM(&jvm, (void**)&env, &vm_args);
#else
    JDK1_1InitArgs vm_args;
    char classpath[1024];
    vm_args.version = 0x00010001;
    JNI_GetDefaultJavaVMInitArgs(&vm_args);
    /* Append USER_CLASSPATH to the default system class path */
    sprintf(classpath, "%s%c%s",
            vm_args.classpath, PATH_SEPARATOR, USER_CLASSPATH);
    vm_args.classpath = classpath;
    /* Create the Java VM */
    res = JNI_CreateJavaVM(&jvm, &env, &vm_args);
#endif /* JNI_VERSION_1_2 */
    if (res < 0) {
        fprintf(stderr, "Can't create Java VM\n");
        exit(1);
    }
    for (i = 0; i < 5; i++)
        /* We pass the thread number to every thread */
        _beginthread(thread_fun, 0, (void *)i);
    Sleep(1000); /* wait for threads to start */
    (*jvm)->DestroyJavaVM(jvm);
}
```

此示例，会分别在 5 个线程中调用 Prog.main。 当派生完毕，主线程等待各子线程启动，该函数会在 5 个 thread 结束后返回。

译者：

该示例只为简单展示 Attach/detach 机制，具体实现线程机制，请不要用 Sleep 之类的函数在主线程中等待子线程启动，应该使用信号量。

JNI_AttachCurrentThread 第三个参数为 NULL，Java 2 SDK release 1.2 中提供 JNI_ThreadAttachArgs 结构，允许传参，如准备把当前线程 attach 到那个 thread group。

当线程执行完毕，调用 DetachCurrentThread 释放所有当前线程的 LocalRef。

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
```

在不同平台输出可能不同。

CHAPTER 8

Additional JNI Features

上一章介绍了应用程序中嵌入 JavaVM 和多线程应用的写法，本章介绍其他特性。

8.1 JNI and Threads

JavaVM 支持多线程编程。

关于并发编程议题已超出本书范围，请阅读<Concurrent Programming in Java, Design Principles and Patterns> by Doug Lea(Addison-Wesley, 1997) (中译本，<并发编程实践>)。

8.1.1 Constraints

这里有几个在多线程环境中必须遵守的原则，这些原则可以让你的代码安全的在多线程环境下运行。

- JNIEnv 结构与线程绑定的，绝对不能在多线程中共享 JNIEnv 结构
- LocalRef 与本线程绑定的，绝对不能在多线程中共享 LocalRef

8.1.2 Monitor Entry and Exit

monitor(监视器)是 Java 平台原生(语言级别，语言本身支持的)的同步机制。每个对象都带一个 monitor。

如下 Java 代码的同步机制，JNI 提供等价机制达到同步的目的。

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

```
}
```

JavaVM 保证只有一个线程持有同步块锁。当另一个线程执行到同步块时将一直阻塞，直到先进入同步块的那个线程离开该块(释放锁)。

JNI 代码可用如下机制，得到同样语义：

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

  ... /* synchronized block */

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

MonitorEnter/MonitorExit 一定要配对调用。不然，将导致死锁。

8.1.3 Monitor Wait and Notify

为了同步需要，Java 提供了 Object.wait/Object.notify/Object.notifyAll。JNI 不提供这些对应的 JNI 实现，但你可以回调 Java 方法.

```
/* precomputed method IDs */
static jmethodID MID_Object_wait;
static jmethodID MID_Object_notify;
static jmethodID MID_Object_notifyAll;

void
JNU_MonitorWait(JNIEnv *env, jobject object, jlong timeout)
{
    (*env)->CallVoidMethod(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);
}
```

8.1.4 Obtaining a JNIEnv Pointer in Arbitrary Contexts

我们前面已经做了说明：JNIEnv 与线程绑定。对于 native method 这不是问题，因为第一个参数即为 JNIEnv。但某些函数不是在 native method 下被调用的，没有 JNIEnv 结构，比如供 OS 回调的函数。

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

如上，只要 jvm 被赋值，就可通过 AttachCurrentThread 得到 JNIEnv。JavaVM 结构可以在多线程间共享，有以下两个 jvm 的赋值时机：

- JNI_GetCreatedJavaVM 创建虚拟机时
- JNI_OnLoad

此外，只要在该线程中事先调用过 AttachCurrentThread 后，JNIEnv->GetEnv 可以返回 JNIEnv。