1 - 简介

本章介绍 *Java 本地接口*（Java Native Interface，JNI）。JNI 是本地编程接口。它使得在 Java 虚拟机（VM）内部运行的 Java 代码能够与用其它编程语言（如 C、C++ 和汇编语言）编写的应用程序和库进行互操作。

JNI 最重要的好处是它没有对底层 Java 虚拟机的实现施加任何限制。因此，Java 虚拟机厂商可以在不影响虚拟机其它部分的情况下添加对 JNI 的支持。程序员只需编写一种版本的本地应用程序或库，就能够与所有支持 JNI 的 Java 虚拟机协同工作。

本章论及以下主题：

Java 本地接口概述

尽管可以完全用 Java 编写应用程序，但是有时单独用 Java 不能满足应用程序的需要。程序员使用 JNI 来编写 *Java 本地方法*，可以处理那些不能完全用 Java 编写应用程序的情况。

以下示例说明了何时需要使用 Java 本地方法：

- 标准 Java 类库不支持与平台相关的应用程序所需的功能。
- 已经拥有了一个用另一种语言编写的库，而又希望通过 JNI 使 Java 代码能够访问该库。
- 想用低级语言（如汇编语言）实现一小段时限代码。

通过用 JNI 编程，可以将本地方法用于：

- 创建、检查及更新 Java 对象（包括数组和字符串）。
- 调用 Java 方法。
- 捕捉和抛出异常。
- 加载类和获得类信息。

- 执行运行时类型检查。

也可以与*调用 API* 一起使用 JNI，以允许任意本地应用程序嵌入到 Java 虚拟机中。这样使得程序员能够轻易地让已有应用程序支持 Java，而不必与虚拟机源代码相链接。

背景

目前，不同厂商的虚拟机提供了不同的本地方法接口。这些不同的接口使程序员不得不在给定平台上编写、维护和分发多种版本的本地方法库。

下面简要分析一下部分已有本地方法接口，例如：

- JDK 1.0 本地方法接口
- Netscape 的 Java 运行时接口
- Microsoft 的原始本地接口和 Java/COM 接口

JDK 1.0 本地方法接口

JDK 1.0 附带有本地方法接口。遗憾的是，有两点原因使得该接口不适合于其它 Java 虚拟机。

第一，平台相关代码将 Java 对象中的域作为 C 结构的成员来进行访问。但是，*Java 语言规范*没有规定在内存中对象是如何布局的。如果 Java 虚拟机在内存中布局对象的方式有所不同，程序员就不得不重新编译本地方法库。

第二，JDK 1.0 的本地方法接口依赖于保守的垃圾收集器。例如，无限制地使用 unhand 宏使得有必要以保守方式扫描本地堆栈。

Java 运行时接口

Netscape 建议使用 Java 运行时接口 (JRI)，它是 Java 虚拟机所提供服务的通用接口。JRI 的设计融入了可移植性---它几乎没有对底层 Java 虚拟机的实现细节作任何假设。JRI 提出了各种各样的问题，包括本地方法、调试、反射、嵌入（调用）等等。

原始本地接口和 Java/COM 接口

Microsoft Java 虚拟机支持两种本地方法接口。在低一级，它提供了高效的原始本地接口 (RNI)。RNI 提供了与 JDK 本地方法接口有高度源代码级的向后兼容性，尽管它们之间还有一个主要区别，即平台相关代码必须用 RNI 函数来与垃圾收集器进行显式的交互，而不是依赖于保守的垃圾收集。

在高一级，Microsoft 的 Java/COM 接口为 Java 虚拟机提供了与语言无关的标准二进制接口。Java 代码可以象使用 Java 对象一样来使用 COM 对象。Java 类也可以作为 COM 类显示给系统的其余部分。

目标

我们认为统一的，经过细致考虑的标准接口能够向每个用户提供以下好处：

- 每个虚拟机厂商都可以支持更多的平台相关代码。
- 工具构造器不必维护不同的本地方法接口。
- 应用程序设计人员可以只编写一种版本的平台相关代码就能够在不同的虚拟机上运行。

获得标准本地方法接口的最佳途径是联合所有对 Java 虚拟机有兴趣的当事方。因此，我们在 Java 获得许可方之间组织了一系列研讨会，对设计统一的本地方法接口进行了讨论。从研讨会可以明确地看出标准本地方法接口必须满足以下要求：

- 二进制兼容性 - 主要的目标是在给定平台上的所有 Java 虚拟机实现之间实现本地方法库的二进制兼容性。对于给定平台，程序员只需要维护一种版本的本地方法库。
- 效率 - 若要支持时限代码，本地方法接口必须增加一点系统开销。所有已知的用于确保虚拟机无关性（因而具有二进制兼容性）的技术都会占用一定的系统开销。我们必须在效率与虚拟机无关性之间进行某种折衷。
- 功能 - 接口必须显示足够的 Java 虚拟机内部情况以使本地方法能够完成有用的任务。

Java 本地接口方法

我们希望采用一种已有的方法作为标准接口，因为这样程序员（程序员不得不学习在不同虚拟机中的多种接口）的工作负担最轻。遗憾的是，已有解决方案中没有任何方案能够完全地满足我们的目标。

Netscape 的 JRI 最接近于我们所设想的可移植本地方法接口，因而我们采用它作为设计起点。熟悉 JRI 的读者将会注意到在 API 命名规则、方法和域 ID 的使用、局部和全局引用的使用，等等中的相似点。虽然我们进行了最大的努力，但是 JNI 并不具有对 JRI 的二进制兼容性，不过虚拟机既可以支持 JRI，又可以支持 JNI。

Microsoft 的 RNI 是对 JDK 1.0 的改进，因为它可以解决使用非保守的垃圾收集器的本地方法的问题。然而，RNI 不适合用作与虚拟机无关的本地方法接口。与 JDK 类似，RNI 本地方法将 Java 对象作为 C 结构来访问。这将导致两个问题：

- RNI 将内部 Java 对象的布局暴露给了平台相关代码。
- 将 Java 对象作为 C 结构直接进行访问使得不可能有效地加入“写屏障”，写屏障是高级的垃圾收集算法所必需的。

作为二进制标准，COM 确保了不同虚拟机之间的完全二进制兼容性。调用 COM 方法只要求间接调用，而这几乎不会占用系统开销。另外，COM 对象对动态链接库解决版本问题的方式也有很大的改进。

然而，有几个因素阻碍了将 COM 用作标准 Java 本地方法接口：

- 第一，Java/COM 接口缺少某些必需功能，例如访问私有域和抛出普通异常。
- 第二，Java/COM 接口自动为 Java 对象提供标准的 IUnknown 和 IDispatch COM 接口，因而平台相关代码能够访问公有方法和域。遗憾的是，IDispatch 接口不能处理重载的 Java 方法，而且在匹配方法名称时不区别大小写。另外，通过 IDispatch 接口暴露的所有 Java 方法被打包在一起来执行动态类型检查和强制转换。这是因为 IDispatch 接口的设计只考虑到了弱类型的语言（例如 Basic）。
- 第三，COM 允许软件组件（包括完全成熟的应用程序）一起工作，而不是处理单个低层函数。我们认为将所有 Java 类或低层本地方法都当作软件组件是不恰当的。
- 第四，在 UNIX 平台上由于缺少对 COM 的支持，所以阻碍了直接采用 COM。

虽然我们没有将 Java 对象作为 COM 对象暴露给平台相关代码，但是 JNI 接口自身与 COM 具有二进制兼容性。我们采用与 COM 一样的跳转表和调用约定。*这意味着，一旦具有对 COM 的跨平台支持，JNI 就能成为 Java 虚拟机的 COM 接口。*

我们认为 JNI 不应该是给定 Java 虚拟机所支持的唯一的本地方法接口。标准接口的好处在于程序员可以将自己的平台相关代码库加载到不同的 Java 虚拟机上。在某些情况下，程序员可能不得不使用低层且与虚拟机有关的接口来获得较高的效率。但在其它情况下，程序员可能使用高层接口来建立软件组件。实际上，我们希望随着 Java 环境和组件软件技术发展得越来越成熟，本地方法将变得越来越不重要。

利用 JNI 编程

本地方法程序设计人员应开始利用 JNI 进行编程。利用 JNI 编程隔离了一些未知条件，例如终端用户可能正在运行的厂商的虚拟机。遵守 JNI 标准是本地库能在给定 Java 虚拟机上运行的最好保证。例如，虽然 JDK 1.1 将继续支持 JDK 1.0 中所实现的旧式的本地方法接口，但是可以肯定的是 JDK 的未来版本将停止支持旧式的本地方法接口。依赖于旧式接口的本地方法将不得不重新编写。

如果您正在实现 Java 虚拟机，则应该实现 JNI。我们（Javasoft 和获得许可方）尽力确保 JNI 不会占用虚拟机实现的系统开销或施加任何限制，包括对象表示，垃圾收集机制等。如果您遇到了我们可能忽视了的问题，请告知我们。

JDK 1.1.2 中的变化

为了更好地支持 Java 运行时环境 (JRE)，在 JDK 1.1.2 中对调用 API 在几个方面作了扩展。这些变化没有破坏任何已有代码，JNI 本地方法接口也没有改变。

- JDK1_1InitArgs 结构中的 reserved0 域已被重新命名为 version。JDK1_1InitArgs 结构保存 JNI_CreateJavaVM 的初始化参数。JNI_GetDefaultJavaVMInitArgs 和 JNI_CreateJavaVM 的调用者必须将版本域设置为 0x00010001。JNI_GetDefaultJavaVMInitArgs 被更改为返回 jint，用于表示是否支持所请求的版本。
- JDK1_1InitArgs 结构中的 reserved1 域已被重新命名为 properties。这是一个 NULL-终结的字符串数组。每个字符串具有以下格式：

  name=value

  表示系统属性（该功能对应于 Java 命令行中的 -D 选项）。

- 在 JDK 1.1.1 中，调用 DestroyJavaVM 的线程必须是虚拟机中的唯一用户线程。JDK 1.1.2 放松了这一限制。如果调用 DestroyJavaVM 时有多

个用户线程，则虚拟机将等待直到当前线程成为唯一的用户线程，然后销毁自己。

2 - 设计概述

本章着重讨论 JNI 中的主要设计问题，其中的大部分问题都与本地方法有关。调用 API 的设计将在 第 5 章 “调用 API” 中讨论。

JNI 接口函数和指针

平台相关代码是通过调用 JNI 函数来访问 Java 虚拟机功能的。JNI 函数可通过*接口指针*来获得。接口指针是指针的指针，它指向一个指针数组，而指针数组中的每个元素又指向一个接口函数。每个接口函数都处在数组的某个预定偏移量中。图 2-1 说明了接口指针的组织结构。

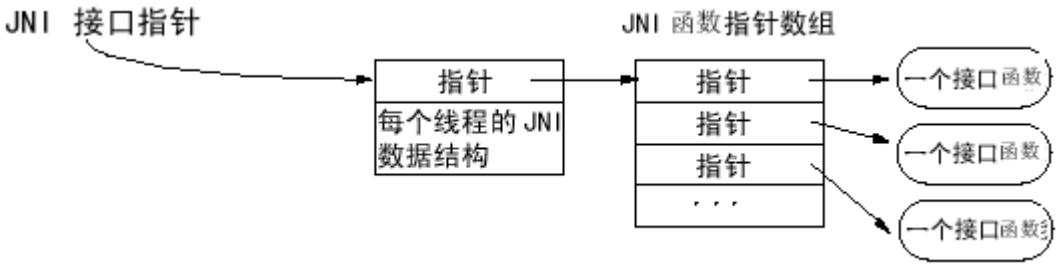


图 2-1 接口指针

JNI 接口的组织类似于 C++ 虚拟函数表或 COM 接口。使用接口表而不使用硬性编入的函数表的好处是使 JNI 名字空间与平台相关代码分开。虚拟机可以很容易地提供多个版本的 JNI 函数表。例如，虚拟机可支持以下两个 JNI 函数表：

- 一个表对非法参数进行全面检查，适用于调试程序；
- 另一个表只进行 JNI 规范所要求的最小程度的检查，因此效率较高。

JNI 接口指针只在当前线程中有效。因此，本地方法不能将接口指针从一个线程传递到另一个线程中。实现 JNI 的虚拟机可将本地线程的数据分配和储存在 JNI 接口指针所指向的区域中。

本地方法将 JNI 接口指针当作参数来接受。虚拟机在从相同的 Java 线程中对本地方法进行多次调用时，保证传递给该本地方法的接口指针是相同的。但是，一个本地方法可被不同的 Java 线程所调用，因此可以接受不同的 JNI 接口指针。

加载和链接本地方法

对本地方法的加载通过 System.loadLibrary 方法实现。下例中，类初始化方法加载了一个与平台有关的本地库，在该本地库中给出了本地方法 f 的定义：

```
package pkg;
class Cls {
     native double f(int i, String s);
     static {
         System.loadLibrary("pkg_Cls");
     }
}
```

System.loadLibrary 的参数是程序员任意选取的库名。系统按照标准的但与平台有关的处理方法将该库名转换为本地库名。例如，Solaris 系统将名称 pkg_Cls 转换为 libpkg_Cls.so，而 Win32 系统将相同的名称 pkg_Cls 转换为 pkg_Cls.dll。

程序员可用单个库来存放任意数量的类所需的所有本地方法，只要这些类将被相同的类加载器所加载。虚拟机在其内部为每个类加载器保护其所加载的本地库清单。提供者应该尽量选择能够避免名称冲突的本地库名。

如果底层操作系统不支持动态链接，则必须事先将所有的本地方法链接到虚拟机上。这种情况下，虚拟机实际上不需要加载库即可完成 System.loadLibrary 调用。

程序员还可调用 JNI 函数 RegisterNatives() 来注册与类关联的本地方法。在与静态链接的函数一起使用时，RegisterNatives() 函数将特别有用。

解析本地方法名

动态链接程序是根据项的名称来解析各项的。本地方法名由以下几部分串接而成：

- 前缀 Java_
- mangled 全限定的类名
- 下划线（“_”）分隔符
- mangled 方法名
- 对于重载的本地方法，加上两个下划线（“__”），后跟 mangled 参数签名

虚拟机将为本地库中的方法查找匹配的方法名。它首先查找短名（没有参数签名的名称），然后再查找带参数签名的长名称。只有当某个本地方法被另一个本 地方法重载时程序员才有必要使用长名。但如果本地方法的名称与非本地方法的名称相同，则不会有问题。因为非本地方法（Java 方法）并不放在本地库中。

下例中，不必用长名来链接本地方法 g，因为另一个方法 g 不是本地方法，因而它并不在本地库中。

```
class Cls1 {
  int g(int i);
  native int g(double d);
}
```

我们采取简单的名字搅乱方案，以保证所有的 Unicode 字符都能被转换为有效的 C 函数名。我们用下划线（“_”） 字符来代替全限定的类名中的斜杠（“/”）。**由于名称或类型描述符从来不会以数字打头，我们用 _0、...、_9 来代替转义字符序列，如表 2-1 所示：**

表 2-1 Unicode 字符转换

转义字符序列	表示
_0XXXX	Unicode 字符 XXXX。
_1	字符**“_”**
_2	签名中的字符“;”
_3	签名中的字符“[”

本地方法和接口 API 都要遵守给定平台上的库调用标准约定。例如，UNIX 系统使用 C 调用约定，而 Win32 系统使用 __stdcall。

本地方法的参数

JNI 接口指针是本地方法的第一个参数。其类型是 *JNIEnv*。第二个参数随本地方法是静态还是非静态而有所不同。非静态本地方法的第二个参数是对对象的引用，而静态本地方法的第二个参数是对其 Java 类的引用。

其余的参数对应于通常 Java 方法的参数。本地方法调用利用返回值将结果传回调用程序中。第 3 章 “JNI 的类型和数据结构” 将描述 Java 类型和 C 类型之间的映射。

代码示例 2-1 说明了如何用 C 函数来实现本地方法 f。对本地方法 f 的声明如下：

```
package pkg;
class Cls {
     native double f(int i, String s);
     ...
}
```

具有长 mangled 名称 Java_pkg_Cls_f_ILjava_lang_String_2 的 C 函数实现本地方法 f：

代码示例 2-1： 用 C 实现本地方法

```
jdouble Java_pkg_Cls_f__ILjava_lang_String_2 (
     JNIEnv *env,          /* 接口指针 */
     jobject obj,          /* “this”指针 */
     jint i,               /* 第一个参数 */
     jstring s)            /* 第二个参数 */
{
     /* 取得 Java 字符串的 C 版本 */
     const char *str = (*env)->GetStringUTFChars(env, s, 0);
     /* 处理该字符串 */
     ...
     /* 至此完成对 str 的处理 */
     (*env)->ReleaseStringUTFChars(env, s, str);
     return ...
}
```

注意，我们总是用接口指针 *env* 来操作 Java 对象。可用 C++ 将此代码写得稍微简洁一些，如代码示例 2-2 所示：

代码示例 2-2： 用 C++ 实现本地方法

```
extern "C" /* 指定 C 调用约定 */
jdouble Java_pkg_Cls_f__ILjava_lang_String_2 (
     JNIEnv *env,          /* 接口指针 */
     jobject obj,          /* “this”指针 */
     jint i,               /* 第一个参数 */
     jstring s)            /* 第二个参数  */
{
     const char *str = env->GetStringUTFChars(s, 0);
     ...
     env->ReleaseStringUTFChars(s, str);
     return ...
}
```

使用 C++ 后，源代码变得更为直接，且接口指针参数消失。但是，C++ 的内在机制与 C 的完全一样。在 C++ 中，JNI 函数被定义为内联成员函数，它们将扩展为相应的 C 对应函数。

引用 Java 对象

基本类型（如整型、字符型等）在 Java 和平台相关代码之间直接进行复制。而 Java 对象由引用来传递。虚拟机必须跟踪传到平台相关代码中的对象，以使这些对象不会被垃圾收集器释放。反之，平台相关代码必须能用某种方式通知虚拟机它不再需要那些对象，同时，垃圾收集器必须能够移走被平台相关代码引用过的对象。

全局和局部引用

JNI 将平台相关代码使用的对象引用分成两类：*局部引用*和*全局引用*。局部引用在本地方法调用期间有效，并在本地方法返回后被自动释放掉。全局引用将一直有效，直到被显式释放。

对象是被作为局部引用传递给本地方法的，由 JNI 函数返回的所有 Java 对象也都是局部引用。JNI 允许程序员从局部引用创建全局引用。要求 Java 对象的 JNI 函数既可接受全局引用也可接受局部引用。本地方法将局部引用或全局引用作为结果返回。

大多数情况下，程序员应该依靠虚拟机在本地方法返回后释放所有局部引用。但是，有时程序员必须显式释放某个局部引用。例如，考虑以下的情形：

- 本地方法要访问一个大型 Java 对象，于是创建了对该 Java 对象的局部引用。然后，本地方法要在返回调用程序之前执行其它计算。对这个大型 Java 对象的局部引用将防止该对象被当作垃圾收集，即使在剩余的运算中并不再需要该对象。
- 本 地方法创建了大量的局部引用，但这些局部引用并不是要同时使用。由于虚拟机需要一定的空间来跟踪每个局部引用，创建太多的局部引用将可能使系统耗尽内存。 例如，本地方法要在一个大型对象数组中循环，把取回的元素作为局部引用，并在每次迭代时对一个元素进行操作。每次迭代后，程序员不再需要对该数组元素的局 部引用。

JNI 允许程序员在本地方法内的任何地方对局部引用进行手工删除。为确保程序员可以手工释放局部引用，JNI 函数将不能创建额外的局部引用，除非是这些 JNI 函数要作为结果返回的引用。

局部引用仅在创建它们的线程中有效。本地方法不能将局部引用从一个线程传递到另一个线程中。

实现局部引用

为了实现局部引用，Java 虚拟机为每个从 Java 到本地方法的控制转换都创建了注册服务程序。注册服务程序将不可移动的局部引用映射为 Java 对象，并防止这些对象被当作垃圾收集。所有传给本地方法的 Java 对象（包括那些作为 JNI 函数调用结果返回的对象）将被自动添加到注册服务程序中。本地方法返回后，注册服务程序将被删除，其中的所有项都可以被当作垃圾来收集。

可用各种不同的方法来实现注册服务程序，例如，使用表、链接列表或 hash 表来实现。虽然引用计数可用来避免注册服务程序中有重复的项，但 JNI 实现不是必须检测和消除重复的项。

注意，以保守方式扫描本地堆栈并不能如实地实现局部引用。平台相关代码可将局部引用储存在全局或堆数据结构中。

访问 Java 对象

JNI 提供了一大批用来访问全局引用和局部引用的函数。这意味着无论虚拟机在内部如何表示 Java 对象，相同的本地方法实现都能工作。这就是为什么 JNI 可被各种各样的虚拟机实现所支持的关键原因。

通过不透明的引用来使用访问函数的开销比直接访问 C 数据结构的开销来得高。我们相信，大多数情况下，Java 程序员使用本地方法是为了完成一些重要任务，此时这种接口的开销不是首要问题。

访问基本类型数组

对于含有大量基本数据类型（如整数数组和字符串）的 Java 对象来说，这种开销将高得不可接受 （考虑一下用于执行矢量和矩阵运算的本地方法的情形便知）。对 Java 数组进行迭代并且要通过函数调用取回数组的每个元素，其效率是非常低的。

一个解决办法是引入“钉住”概念，以使本地方法能够要求虚拟机钉住数组内容。而后，该本地方法将接受指向数值元素的直接指针。但是，这种方法包含以下两个前提：

- 垃圾收集器必须支持钉住。
- 虚拟机必须在内存中连续存放基本类型数组。虽然大多数基本类型数组都是连续存放的，但布尔数组可以压缩或不压缩存储。因此，依赖于布尔数组确切存储方式的本地方法将是不可移植的。

我们将采取折衷方法来克服上述两个问题。

首先，我们提供了一套函数，用于在 Java 数组的一部分和本地内存缓冲之间复制基本类型数组元素。这些函数只有在本地方法只需访问大型数组中的一小部分元素时才使用。

其次，程序员可用另一套函数来取回数组元素的受约束版本。记住，这些函数可能要求 Java 虚拟机分配存储空间和进行复制。虚拟机实现将决定这些函数是否真正复制该数组，如下所示：

- 如果垃圾收集器支持钉住，且数组的布局符合本地方法的要求，则不需要进行复制。
- 否则，该数组将被复制到不可移动的内存块中（例如，复制到 C 堆中），并进行必要的格式转换，然后返回指向该副本的指针。

最后，接口提供了一些函数，用以通知虚拟机本地方法已不再需要访问这些数组元素。当调用这些函数时，系统或者释放数组，或者在原始数组与其不可移动副本之间进行协调并将副本释放。

这种处理方法具有灵活性。垃圾收集器的算法可对每个给定的数组分别作出复制或钉住的决定。例如，垃圾收集器可能复制小型对象而钉住大型对象。

JNI 实现必须确保多个线程中运行的本地方法可同时访问同一数组。例如，JNI 可以为每个被钉住的数组保留一个内部计数器，以便某个线程不会解开同时被另一个线程钉住的数组。注意，JNI 不必将基本类型数组锁住以专供某个本地方法访问。同时从不同的线程对 Java 数组进行更新将导致不确定的结果。

访问域和方法

JNI 允许本地方法访问 Java 对象的域或调用其方法。JNI 用符号名称和类型签名来识别方法和域。从名称和签名来定位域或对象的过程可分为两步。例如，为调用类 *cls* 中的 f 方法，平台相关代码首先要获得方法 ID，如下所示：

```
jmethodID mid =

 env->GetMethodID(cls, "f", "(ILjava/lang/String;)D");
```

然后，平台相关代码可重复使用该方法 ID 而无须再查找该方法，如下所示：

```
jdouble result = env->CallDoubleMethod(obj, mid, 10, str);
```

域 ID 或方法 ID 并不能防止虚拟机卸载生成该 ID 的类。该类被卸载之后，该方法 ID 或域 ID 亦变成无效。因此，如果平台相关代码要长时间使用某个方法 ID 或域 ID，则它必须确保：

- 保留对所涉及类的活引用，或
- 重新计算该方法 ID 或域 ID。

JNI 对域 ID 和方法 ID 的内部实现并不施加任何限制。

报告编程错误

JNI 不检查诸如传递 NULL 指针或非法参数类型之类的编程错误。非法的参数类型包括诸如要用 Java 类对象时却用了普通 Java 对象这样的错误。JNI 不检查这些编程错误的理由如下：

- 强迫 JNI 函数去检查所有可能的错误情况将降低正常（正确）的本地方法的性能。
- 在许多情况下，没有足够的运行时的类型信息可供这种检查使用。

大多数 C 库函数对编程错误不进行防范。例如，printf() 函数在接到一个无效地址时通常是引起运行错而不是返回错误代码。强迫 C 库函数检查所有可能的错误情况将有可能引起这种检查被重复进行--先是在用户代码中进行，然后又在库函数中再次进行。

程序员不得将非法指针或错误类型的参数传递给 JNI 函数。否则，可能产生意想不到的后果，包括可能使系统状态受损或使虚拟机崩溃。

Java 异常

JNI 允许本地方法抛出任何 Java 异常。本地方法也可以处理突出的 Java 异常。未被处理的 Java 异常将被传回虚拟机中。

异常和错误代码

一些 JNI 函数使用 Java 异常机制来报告错误情况。大多数情况下，JNI 函数通过返回错误代码*并*抛出 Java 异常来报告错误情况。错误代码通常是特殊的返回值（如 NULL），这种特殊的返回值在正常返回值范围之外。因此，程序员可以：

- 快速检查上一个 JNI 调用所返回的值以确定是否出错，并
- 通过调用函数 ExceptionOccurred() 来获得异常对象，它含有对错误情况的更详细说明。

在以下两种情况中，程序员需要先查出异常，然后才能检查错误代码：

- 调用 Java 方法的 JNI 函数返回该 Java 方法的结果。程序员必须调用 ExceptionOccurred() 以检查在执行 Java 方法期间可能发生的异常。
- 某些用于访问 JNI 数组的函数并不返回错误代码，但可能会抛出 ArrayIndexOutOfBoundsException 或 ArrayStoreException。

在所有其它情况下，返回值如果不是错误代码值就可确保没有抛出异常。

异步异常

在多个线程的情况下，当前线程以外的其它线程可能会抛出异步异常。异步异常并不立即影响当前线程中平台相关代码的执行，直到出现下列情况：

- 该平台相关代码调用某个有可能抛出同步异常的 JNI 函数，或者
- 该平台相关代码用 ExceptionOccurred() 显式检查同步异常或异步异常。

注意，只有那些有可能抛出同步异常的 JNI 函数才检查异步异常。

本地方法应在必要的地方（例如，在一个没有其它异常检查的紧密循环中）插入 ExceptionOccurred() 检查以确保当前线程可在适当时间内对异步异常作出响应。

异常的处理

可用两种方法来处理平台相关代码中的异常：

- 本地方法可选择立即返回，使异常在启动该本地方法调用的 Java 代码中抛出。
- 平台相关代码可通过调用 ExceptionClear() 来清除异常，然后执行自己的异常处理代码。

抛出了某个异常之后，平台相关代码必须先清除异常，然后才能进行其它的 JNI 调用。当有待定异常时，只有以下这些 JNI 函数可被安全地调用：ExceptionOccurred()、ExceptionDescribe() 和 ExceptionClear()。ExceptionDescribe() 函数将打印有关待定异常的调试消息。

---

3 - JNI 的类型和数据结构

本章讨论 JNI 如何将 Java 类型映射到本地 C 类型。

基本类型

表 3-1 描述 Java 基本类型及其与计算机相关的本地等效类型。

表 3-1 基本类型和本地等效类型

Java 类型	本地类型	说明
boolean	jboolean	无符号，8 位
byte	jbyte	无符号，8 位
char	jchar	无符号，16 位
short	jshort	有符号，16 位
int	jint	有符号，32 位
long	jlong	有符号，64 位
float	jfloat	32 位
double	jdouble	64 位
void	void	N/A

为了使用方便，特提供以下定义。

```
#define JNI_FALSE  0
#define JNI_TRUE   1
```

jsize 整数类型用于描述主要指数和大小：

```
typedef jint jsize;
```

引用类型

JNI 包含了很多对应于不同 Java 对象的引用类型。JNI 引用类型的组织层次如图 3-1 所示。

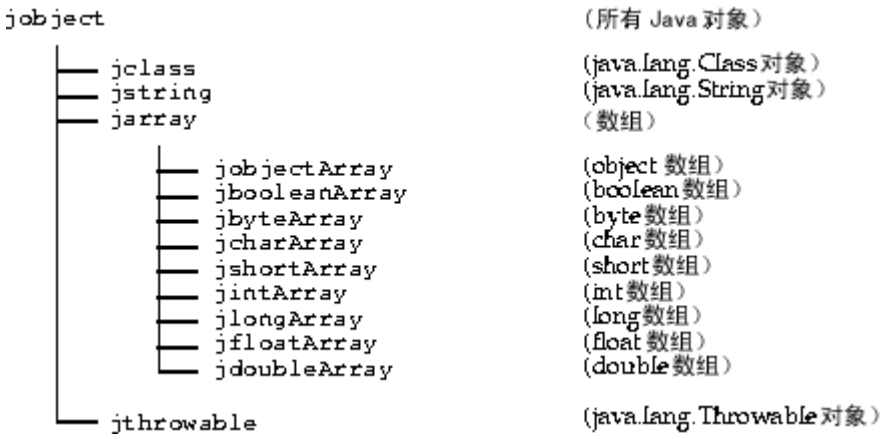


图 3-1 引用类型层次

在 C 中，所有其它 JNI 引用类型都被定义为与 jobject 一样。例如：

```
typedef jobject jclass;
```

在 C++ 中，JNI 引入了虚构类以加强子类关系。例如：

```
class _jobject {};
class _jclass : public _jobject {};
...
typedef _jobject *jobject;
typedef _jclass *jclass;
```

域 ID 和方法 ID

方法 ID 和域 ID 是常规的 C 指针类型：

```
struct _jfieldID;               /*不透明结构 */
typedef struct _jfieldID *jfieldID;   /* 域 ID */

struct _jmethodID;              /* 不透明结构 */
typedef struct _jmethodID *jmethodID; /* 方法 ID */
```

值类型

jvalue 联合类型在参数数组中用作单元类型。其声明方式如下：

```
typedef union jvalue {
    jboolean z;
    jbyte    b;
    jchar    c;
    jshort   s;
    jint     i;
    jlong    j;
    jfloat   f;
    jdouble  d;
    jobject  l;
} jvalue;
```

类型签名

JNI 使用 Java 虚拟机的类型签名表述。表 3-2 列出了这些类型签名。

表 3-2 Java 虚拟机类型签名

类型签名	Java 类型
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double
L *fully-qualified-class* **;**	全限定的类
[*type*	*type*[]
(*arg-types* **)** *ret-type*	方法类型

例如，Java 方法：

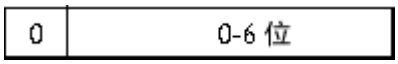
```
long f (int n, String s, int[] arr);
```

具有以下类型签名：

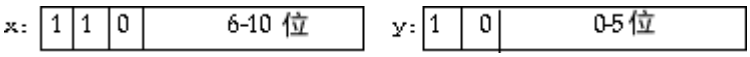
```
(ILjava/lang/String;[I)J
```

UTF-8 字符串

JNI 用 UTF-8 字符串来表示各种字符串类型。UTF-8 字符串和 Java 虚拟机所使用的一样。UTF-8 字符串的编码方式使得仅包含非空 ASCII 字符的字符序列能够按每字符一个字节表示，但是最多只能表示 16 位的字符。所有在 \u0001 到 \u007F 范围内的字符都用单字节表示，如下所示：

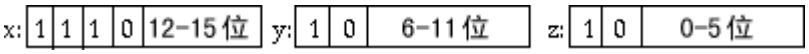


字节中的七位数据确定了所表示字符的值。空字符 (\u000) 和 \u0080 到 \u07FF 范围内的字符用一对字节表示， 即 *x* 和 *y*，如下所示：



值为 ((x&0x1f)<<6)+(y&0x3f) 的字符需用两个字节表示。

\u0800 到 \uFFFF 范围内的字符用三个字节表示，即 *x*，*y*，和 *z*：



值为 ((x&0xf)<<12)+(y&0x3f)<<6)+(z&0x3f) 的字符需用三个字节表示。

此格式与“标准” UTF-8 格式之间有两个区别。第一，空字节 (byte)0 使用双字节格式进行编码，而不是单字节格式。这意味着 Java 虚拟机的 UTF-8 字符串不可能有嵌入的空值。第二，只使用单字节、双字节和三字节格式。Java 虚拟机不能识别更长的 UTF-8 格式。

---

4 - JNI 函数

本章为 JNI 函数提供参考信息。其中列出了全部 JNI 函数，同时也给出了 JNI 函数表的准确布局。

注意：“必须”一词用于约束 JNI 编程人员。例如，当说明某个 JNI 函数*必须*接收非空对象时，就应确保不要向该 JNI 函数传递 NULL。这时，JNI 实现将无需在该 JNI 函数中执行 NULL 指针检查。

本章的部分资料改编自 Netscape 的 JRI 文档。

该参考资料按用法对函数进行组织。参考部分按下列函数区域进行组织：

接口函数表

每个函数均可通过 *JNIEnv* 参数以固定偏移量进行访问。*JNIEnv* 的类型是一个指针，指向存储全部 JNI 函数指针的结构。其定义如下：

注意：前三项留作将来与 COM 兼容。此外，我们在函数表开头部分也留出来多个 NULL 项，从而可将将来与类有关的 JNI 操作添加到 FindClass 后面，而非函数表的末尾。

注意，函数表可在所有 JNI 接口指针间共享。

代码示例 4-1

```
const struct JNINativeInterface ... = {
    NULL,
    NULL,
    NULL,
    NULL,
    GetVersion,

    DefineClass,
    FindClass,
    NULL,
    NULL,
    NULL,
    GetSuperclass,
    IsAssignableFrom,
    NULL,

    Throw,
    ThrowNew,
    ExceptionOccurred,
    ExceptionDescribe,
    ExceptionClear,
    FatalError,
    NULL,
    NULL,

    NewGlobalRef,
    DeleteGlobalRef,
    DeleteLocalRef,
    IsSameObject,
    NULL,
    NULL,

    AllocObject,
    NewObject,
    NewObjectV,
    NewObjectA,
```

GetObjectClass,
IsInstanceOf,

GetMethodID,

CallObjectMethod,
CallObjectMethodV,
CallObjectMethodA,
CallBooleanMethod,
CallBooleanMethodV,
CallBooleanMethodA,
CallByteMethod,
CallByteMethodV,
CallByteMethodA,
CallCharMethod,
CallCharMethodV,
CallCharMethodA,
CallShortMethod,
CallShortMethodV,
CallShortMethodA,
CallIntMethod,
CallIntMethodV,
CallIntMethodA,
CallLongMethod,
CallLongMethodV,
CallLongMethodA,
CallFloatMethod,
CallFloatMethodV,
CallFloatMethodA,
CallDoubleMethod,
CallDoubleMethodV,
CallDoubleMethodA,
CallVoidMethod,
CallVoidMethodV,
CallVoidMethodA,

CallNonvirtualObjectMethod,

CallNonvirtualObjectMethodV,
CallNonvirtualObjectMethodA,
CallNonvirtualBooleanMethod,
CallNonvirtualBooleanMethodV,
CallNonvirtualBooleanMethodA,
CallNonvirtualByteMethod,
CallNonvirtualByteMethodV,
CallNonvirtualByteMethodA,
CallNonvirtualCharMethod,
CallNonvirtualCharMethodV,
CallNonvirtualCharMethodA,
CallNonvirtualShortMethod,
CallNonvirtualShortMethodV,
CallNonvirtualShortMethodA,
CallNonvirtualIntMethod,
CallNonvirtualIntMethodV,
CallNonvirtualIntMethodA,
CallNonvirtualLongMethod,
CallNonvirtualLongMethodV,
CallNonvirtualLongMethodA,
CallNonvirtualFloatMethod,
CallNonvirtualFloatMethodV,
CallNonvirtualFloatMethodA,
CallNonvirtualDoubleMethod,
CallNonvirtualDoubleMethodV,
CallNonvirtualDoubleMethodA,
CallNonvirtualVoidMethod,
CallNonvirtualVoidMethodV,
CallNonvirtualVoidMethodA,

GetFieldID,

GetObjectField,
GetBooleanField,
GetByteField,
GetCharField,
GetShortField,

GetIntField,
GetLongField,
GetFloatField,
GetDoubleField,
SetObjectField,
SetBooleanField,
SetByteField,
SetCharField,
SetShortField,
SetIntField,
SetLongField,
SetFloatField,
SetDoubleField,

GetStaticMethodID,

CallStaticObjectMethod,
CallStaticObjectMethodV,
CallStaticObjectMethodA,
CallStaticBooleanMethod,
CallStaticBooleanMethodV,
CallStaticBooleanMethodA,
CallStaticByteMethod,
CallStaticByteMethodV,
CallStaticByteMethodA,
CallStaticCharMethod,
CallStaticCharMethodV,
CallStaticCharMethodA,
CallStaticShortMethod,
CallStaticShortMethodV,
CallStaticShortMethodA,
CallStaticIntMethod,
CallStaticIntMethodV,
CallStaticIntMethodA,
CallStaticLongMethod,
CallStaticLongMethodV,
CallStaticLongMethodA,

```
    CallStaticFloatMethod,
    CallStaticFloatMethodV,
    CallStaticFloatMethodA,
    CallStaticDoubleMethod,
    CallStaticDoubleMethodV,
    CallStaticDoubleMethodA,
    CallStaticVoidMethod,
    CallStaticVoidMethodV,
    CallStaticVoidMethodA,

    GetStaticFieldID,

    GetStaticObjectField,
    GetStaticBooleanField,
    GetStaticByteField,
    GetStaticCharField,
    GetStaticShortField,
    GetStaticIntField,
    GetStaticLongField,
    GetStaticFloatField,
    GetStaticDoubleField,

    SetStaticObjectField,
    SetStaticBooleanField,
    SetStaticByteField,
    SetStaticCharField,
    SetStaticShortField,
    SetStaticIntField,
    SetStaticLongField,
    SetStaticFloatField,
    SetStaticDoubleField,

    NewString,
    GetStringLength,
    GetStringChars,
    ReleaseStringChars,
```

```
    NewStringUTF,
    GetStringUTFLength,
    GetStringUTFChars,
    ReleaseStringUTFChars,

    GetArrayLength,

    NewObjectArray,
    GetObjectArrayElement,
    SetObjectArrayElement,

    NewBooleanArray,
    NewByteArray,
    NewCharArray,
    NewShortArray,
    NewIntArray,
    NewLongArray,
    NewFloatArray,
    NewDoubleArray,

    GetBooleanArrayElements,
    GetByteArrayElements,
    GetCharArrayElements,
    GetShortArrayElements,
    GetIntArrayElements,
    GetLongArrayElements,
    GetFloatArrayElements,
    GetDoubleArrayElements,

    ReleaseBooleanArrayElements,
    ReleaseByteArrayElements,
    ReleaseCharArrayElements,
    ReleaseShortArrayElements,
    ReleaseIntArrayElements,
    ReleaseLongArrayElements,
    ReleaseFloatArrayElements,
    ReleaseDoubleArrayElements,
```

```
    GetBooleanArrayRegion,
    GetByteArrayRegion,
    GetCharArrayRegion,
    GetShortArrayRegion,
    GetIntArrayRegion,
    GetLongArrayRegion,
    GetFloatArrayRegion,
    GetDoubleArrayRegion,
    SetBooleanArrayRegion,
    SetByteArrayRegion,
    SetCharArrayRegion,
    SetShortArrayRegion,
    SetIntArrayRegion,
    SetLongArrayRegion,
    SetFloatArrayRegion,
    SetDoubleArrayRegion,

    RegisterNatives,
    UnregisterNatives,

    MonitorEnter,
    MonitorExit,

    GetJavaVM,
};
```

版本信息

GetVersion

```
jint GetVersion(JNIEnv *env);
```

返回本地方法接口的版本。

参数

env：JNI 接口指针。

返回值：

高 16 位返回主版本号，低 16 位返回次版本号。

在 JDK1.1 中，GetVersion() 返回 0x00010001。

类操作

DefineClass

```
jclass DefineClass(JNIEnv *env, jobject loader,
const jbyte *buf, jsize bufLen);
```

从原始类数据的缓冲区中加载类。

参数：

env：JNI 接口指针。

loader：分派给所定义的类的类加载器。

buf：包含 .class 文件数据的缓冲区。

bufLen：缓冲区长度。

返回值：

返回 Java 类对象。如果出错则返回 NULL。

抛出：

ClassFormatError：如果类数据指定的类无效。

ClassCircularityError：如果类或接口是自身的超类或超接口。

OutOfMemoryError：如果系统内存不足。

FindClass

jclass FindClass(JNIEnv *env, const char *name);

该函数用于加载本地定义的类。它将搜索由 CLASSPATH 环境变量为具有指定名称的类所指定的目录和 zip 文件。

参数：

env：JNI 接口指针。

name：类全名（即包名后跟类名，之间由“/”分隔）。如果该名称以“[”（数组签名字符）打头，则返回一个数组类。

返回值：

返回类对象全名。如果找不到该类，则返回 NULL。

抛出：

ClassFormatError：如果类数据指定的类无效。

ClassCircularityError：如果类或接口是自身的超类或超接口。

NoClassDefFoundError：如果找不到所请求的类或接口的定义。

OutOfMemoryError：如果系统内存不足。

GetSuperclass

jclass GetSuperclass(JNIEnv *env, jclass clazz);

如果 clazz 代表类而非类 object，则该函数返回由 clazz 所指定的类的超类。

如果 clazz 指定类 object 或代表某个接口，则该函数返回 NULL。

参数：

env：JNI 接口指针。

clazz：Java 类对象。

返回值：

由 clazz 所代表的类的超类或 NULL。

IsAssignableFrom

```
jboolean IsAssignableFrom(JNIEnv *env, jclass clazz1,
jclass clazz2);
```

确定 clazz1 的对象是否可安全地强制转换为 clazz2。

参数：

env：JNI 接口指针。

clazz1：第一个类参数。

clazz2：第二个类参数。

返回值：

下列某个情况为真时返回 JNI_TRUE：

- 第一及第二个类参数引用同一个 Java 类。
- 第一个类是第二个类的子类。
- 第二个类是第一个类的某个接口。

异常

Throw

```
jint Throw(JNIEnv *env, jthrowable obj);
```

抛出 java.lang.Throwable 对象。

参数：

env：JNI 接口指针。

obj：java.lang.Throwable 对象。

返回值：

成功时返回 0，失败时返回负数。

抛出：

java.lang.Throwable 对象 obj。

ThrowNew

```
jint ThrowNew(JNIEnv *env, jclass clazz,
const char *message);
```

利用指定类的消息（由 message 指定）构造异常对象并抛出该异常。

参数：

env：JNI 接口指针。

clazz：java.lang.Throwable 的子类。

message：用于构造 java.lang.Throwable 对象的消息。

返回值：

成功时返回 0，失败时返回负数。

抛出：

新构造的 java.lang.Throwable 对象。

ExceptionOccurred

```
jthrowable ExceptionOccurred(JNIEnv *env);
```

确定是否某个异常正被抛出。在平台相关代码调用 ExceptionClear() 或 Java 代码处理该异常前，异常将始终保持抛出状态。

参数：

env：JNI 接口指针。

返回值：

返回正被抛出的异常对象，如果当前无异常被抛出，则返回 NULL。

ExceptionDescribe

```
void ExceptionDescribe(JNIEnv *env);
```

将异常及堆栈的回溯输出到系统错误报告信道（例如 stderr）。该例程可便利调试操作。

参数：

env：JNI 接口指针。

ExceptionClear

```
void ExceptionClear(JNIEnv *env);
```

清除当前抛出的任何异常。如果当前无异常，则此例程不产生任何效果。

参数：

env：JNI 接口指针。

FatalError

```
void FatalError(JNIEnv *env, const char *msg);
```

抛出致命错误并且不希望虚拟机进行修复。该函数无返回值。

参数：

env：JNI 接口指针。

msg：错误消息。

全局及局部引用

NewGlobalRef

jobject NewGlobalRef(JNIEnv *env, jobject obj);

创建 obj 参数所引用对象的新全局引用。obj 参数既可以是全局引用，也可以是局部引用。全局引用通过调用 DeleteGlobalRef() 来显式撤消。

参数：

env：JNI 接口指针。

obj：全局或局部引用。

返回值：

返回全局引用。如果系统内存不足则返回 NULL。

DeleteGlobalRef

void DeleteGlobalRef(JNIEnv *env, jobject globalRef);

删除 globalRef 所指向的全局引用。

参数：

env：JNI 接口指针。

globalRef：全局引用。

DeleteLocalRef

void DeleteLocalRef(JNIEnv *env, jobject localRef);

删除 localRef 所指向的局部引用。

参数：

env：JNI 接口指针。

localRef：局部引用。

对象操作

AllocObject

jobject AllocObject(JNIEnv *env, jclass clazz);

分配新 Java 对象而不调用该对象的任何构造函数。返回该对象的引用。

clazz 参数务必不要引用数组类。

参数：

env：JNI 接口指针。

clazz：Java 类对象。

返回值：

返回 Java 对象。如果无法构造该对象，则返回 NULL。

抛出：

InstantiationException：如果该类为一个接口或抽象类。

OutOfMemoryError：如果系统内存不足。

NewObject

NewObjectA

NewObjectV

```
jobject NewObject(JNIEnv *env, jclass clazz,
jmethodID methodID, ...);

jobject NewObjectA(JNIEnv *env, jclass clazz,
jmethodID methodID, jvalue *args);

jobject NewObjectV(JNIEnv *env, jclass clazz,
jmethodID methodID, va_list args);
```

构造新 Java 对象。方法 ID 指示应调用的构造函数方法。该 ID 必须通过调用 GetMethodID() 获得，且调用时的方法名必须为 <init>，而返回类型必须为 void (V)。

clazz 参数务必不要引用数组类。

NewObject

编程人员应将传递给构造函数的所有参数紧跟着放在 methodID 参数的后面。NewObject() 收到这些参数后，将把它们传给编程人员所要调用的 Java 方法。

NewObjectA

编程人员应将传递给构造函数的所有参数放在 jvalues 类型的数组 args 中，该数组紧跟着放在 methodID 参数的后面。NewObject() 收到数组中的这些参数后，将把它们传给编程人员所要调用的 Java 方法。

NewObjectV

编程人员应将传递给构造函数的所有参数放在 va_list 类型的参数 args 中，该参数紧跟着放在 methodID 参数的后面。NewObject() 收到这些参数后，将把它们传给编程人员所要调用的 Java 方法。

参数：

env：JNI 接口指针。

clazz：Java 类对象。

methodID：构造函数的方法 ID。

NewObject 的其它参数：

传给构造函数的参数。

NewObjectA 的其它参数：

args：传给构造函数的参数数组。

NewObjectV 的其它参数：

args：传给构造函数的参数 va_list。

返回值：

返回 Java 对象，如果无法构造该对象，则返回 NULL。

抛出：

InstantiationException：如果该类为接口或抽象类。

OutOfMemoryError：如果系统内存不足。

构造函数抛出的任何异常。

GetObjectClass

jclass GetObjectClass(JNIEnv *env, jobject obj);

返回对象的类。

参数：

env：JNI 接口指针。

obj：Java 对象（不能为 NULL）。

返回值：

返回 Java 类对象。

IsInstanceOf

```
jboolean IsInstanceOf(JNIEnv *env, jobject obj,
jclass clazz);
```

测试对象是否为某个类的实例。

参数：

env：JNI 接口指针。

obj：Java 对象。

clazz：Java 类对象。

返回值：

如果可将 obj 强制转换为 clazz，则返回 JNI_TRUE。否则返回 JNI_FALSE。NULL 对象可强制转换为任何类。

IsSameObject

```
jboolean IsSameObject(JNIEnv *env, jobject ref1,
jobject ref2);
```

测试两个引用是否引用同一 Java 对象。

参数：

env：JNI 接口指针。

ref1：Java 对象。

ref2：Java 对象。

返回值：

如果 ref1 和 ref2 引用同一 Java 对象或均为 NULL，则返回 JNI_TRUE。否则返回 JNI_FALSE。

访问对象的域

GetFieldID

```
jfieldID GetFieldID(JNIEnv *env, jclass clazz,
const char *name, const char *sig);
```

返回类的实例（非静态）域的域 ID。该域由其名称及签名指定。访问器函数的 *Get<type>Field* 及 *Set<type>Field* 系列使用域 ID 检索对象域。

GetFieldID() 将未初始化的类初始化。

GetFieldID() 不能用于获取数组的长度域。应使用 GetArrayLength()。

参数：

env：JNI 接口指针。

clazz：Java 类对象。

name：0 终结的 UTF-8 字符串中的域名。

sig：0 终结的 UTF-8 字符串中的域签名。

返回值：

域 ID。如果操作失败，则返回 NULL。

抛出：

NoSuchFieldError：如果找不到指定的域。

ExceptionInInitializerError：如果由于异常而导致类初始化程序失败。

OutOfMemoryError：如果系统内存不足。

Get<type>Field 例程

NativeType *Get<type>Field*(JNIEnv *env, jobject obj,
jfieldID fieldID);

该访问器例程系列返回对象的实例（非静态）域的值。要访问的域由通过调用 GetFieldID() 而得到的域 ID 指定。

下表说明了 *Get<type>Field* 例程名及结果类型。应将 *Get<type>Field* 中的 *type* 替换为域的 Java 类型（或使用表中的某个实际例程名），然后将 *NativeType* 替换为该例程对应的本地类型。

表 4-1 Get<type>Field 访问器例程系列

Get<type>Field 例程名	本地类型
GetObjectField()	jobject
GetBooleanField()	jboolean
GetByteField()	jbyte
GetCharField()	jchar
GetShortField()	jshort
GetIntField()	jint
GetLongField()	jlong
GetFloatField()	jfloat
GetDoubleField()	jdouble

参数：

env：JNI 接口指针。

obj：Java 对象（不能为 NULL）。

fieldID：有效的域 ID。

返回值：

域的内容。

Set<type>Field 例程

void *Set<type>Field*(JNIEnv *env, jobject obj, jfieldID fieldID,
NativeType value);

该访问器例程系列设置对象的实例（非静态）域的值。要访问的域由通过调用 SetFieldID() 而得到的域 ID 指定。

下表说明了 *Set<type>Field* 例程名及结果类型。应将 *Set<type>Field* 中的 *type* 替换为域的 Java 类型（或使用表中的某个实际例程名），然后将 *NativeType* 替换为该例程对应的本地类型。

表 4-2 Set<type>Field 访问器例程系列

Set<type>Field 例程名	本地类型
SetObjectField()	jobject
SetBooleanField()	jboolean
SetByteField()	jbyte
SetCharField()	jchar
SetShortField()	jshort
SetIntField()	jint
SetLongField()	jlong
SetFloatField()	jfloat
SetDoubleField()	jdouble

参数：

env：JNI 接口指针。

obj：Java 对象（不能为 NULL）。

fieldID：有效的域 ID。

value：域的新值。

调用实例方法

GetMethodID

```
jmethodID GetMethodID(JNIEnv *env, jclass clazz,
const char *name, const char *sig);
```

返回类或接口实例（非静态）方法的方法 ID。方法可在某个 clazz 的超类中定义，也可从 clazz 继承。该方法由其名称和签名决定。

GetMethodID() 可使未初始化的类初始化。

要获得构造函数的方法 ID，应将 <init> 作为方法名，同时将 void (V) 作为返回类型。

参数：

env：JNI 接口指针。

clazz：Java 类对象。

name：0 终结的 UTF-8 字符串中的方法名。

sig：0 终结的 UTF-8 字符串中的方法签名。

返回值：

方法 ID，如果找不到指定的方法，则为 NULL。

抛出：

NoSuchMethodError：如果找不到指定方法。

ExceptionInInitializerError：如果由于异常而导致类初始化程序失败。

OutOfMemoryError：如果系统内存不足。

Call<type>Method 例程

Call<type>MethodA 例程

Call<type>MethodV 例程

NativeType *Call<type>Method*(JNIEnv *env, jobject obj,
jmethodID methodID, ...);

NativeType Call<type>MethodA(JNIEnv *env, jobject obj,
jmethodID methodID, jvalue *args);

NativeType Call<type>MethodV(JNIEnv *env, jobject obj, jmethodID methodID, va_list args);

这三个操作的方法用于从本地方法调用 Java 实例方法。它们的差别仅在于向其所调用的方法传递参数时所用的机制。

这三个操作将根据所指定的方法 ID 调用 Java 对象的实例（非静态）方法。参数 methodID 必须通过调用 GetMethodID() 来获得。

当这些函数用于调用私有方法和构造函数时，方法 ID 必须从 obj 的真实类派生而来，而不应从其某个超类派生。

Call<type>Method 例程

编程人员应将要传给方法的所有参数紧跟着放在 methodID 参数之后。*Call<type>Method* 例程接受这些参数并将其传给编程人员所要调用的 Java 方法。

Call<type>MethodA 例程

编程人员应将要传给方法的所有参数放在紧跟在 methodID 参数之后的 jvalues 类型数组 args 中。*Call<type>MethodA* routine 接受这些数组中的参数并将其传给编程人员所要调用的 Java 方法。

Call<type>MethodV 例程

编程人员将方法的所有参数放在紧跟着在 methodID 参数之后的 va_list 类型参数变量中。*Call<type>MethodV* routine 接受这些参数并将其传给编程人员所要调用的 Java 方法。

下表根据结果类型说明了各个方法调用例程。用户应将 *Call<type>Method* 中的 *type* 替换为所调用方法的 Java 类型（或使用表中的实际方法调用例程名），同时将 *NativeType* 替换为该例程相应的本地类型。

表 4-3 实例方法调用例程

Call<type>Method 例程名	本地类型
CallVoidMethod() CallVoidMethodA() CallVoidMethodV()	void
CallObjectMethod() CallObjectMethodA() CallObjectMethodV()	jobject
CallBooleanMethod() CallBooleanMethodA() CallBooleanMethodV()	jboolean
CallByteMethod() CallByteMethodA() CallByteMethodV()	jbyte
CallCharMethod() CallCharMethodA() CallCharMethodV()	jchar

CallShortMethod() CallShortMethodA() CallShortMethodV()	jshort
CallIntMethod() CallIntMethodA() CallIntMethodV()	jint
CallLongMethod() CallLongMethodA() CallLongMethodV()	jlong
CallFloatMethod() CallFloatMethodA() CallFloatMethodV()	jfloat
CallDoubleMethod() CallDoubleMethodA() CallDoubleMethodV()	jdouble

参数：

env：JNI 接口指针。

obj：Java 对象。

methodID：方法 ID。

Call<type>Method 例程的其它参数：

要传给 Java 方法的参数。

Call<type>MethodA 例程的其它参数：

args：参数数组。

Call<type>MethodV 例程的其它参数：

args：参数的 va_list。

返回值：

返回调用 Java 方法的结果。

抛出：

执行 Java 方法时抛出的异常。

CallNonvirtual<type>Method 例程

CallNonvirtual<type>MethodA 例程

CallNonvirtual<type>MethodV 例程

NativeType *CallNonvirtual<type>Method*(JNIEnv *env, jobject obj, jclass clazz, jmethodID methodID, ...);

NativeType CallNonvirtual<type>MethodA(JNIEnv *env, jobject obj, jclass clazz, jmethodID methodID, jvalue *args);

NativeType CallNonvirtual<type>MethodV(JNIEnv *env, jobject obj, jclass clazz, jmethodID methodID, va_list args);

这些操作根据指定的类和方法 ID 调用某 Java 对象的实例（非静态）方法。参数 methodID 必须通过调用 clazz 类的 GetMethodID() 获得。

CallNonvirtual<type>Method 和 *Call<type>Method* 例程系列并不相同。*Call<type>Method* 例程根据对象的类调用方法，而 *CallNonvirtual<type>Method* 例程则根据获得方法 ID 的（由 clazz 参数指定）类调用方法。方法 ID 必须从对象的真实类或其某个超类获得。

CallNonvirtual<type>Method 例程

编程人员应将要传给方法的所有参数紧跟着放在 methodID 参数之后。*CallNonvirtual<type>Method* routine 接受这些参数并将其传给编程人员所要调用的 Java 方法。

CallNonvirtual<type>MethodA 例程

编程人员应将要传给方法的所有参数放在紧跟在 methodID 参数之后的 jvalues 类型数组 args 中。*CallNonvirtual<type>MethodA* routine 接受这些数组中的参数并将其传给编程人员所要调用的 Java 方法。

CallNonvirtual<type>MethodV 例程

编程人员应将要传给方法的所有参数放在紧跟在 methodID 参数之后的 va_list 类型参数 args 中。*CallNonvirtualMethodV* routine 接受这些参数并将其传给编程人员所要调用的 Java 方法。

下表根据结果类型说明了各个方法调用例程。用户应将 *CallNonvirtual<type>Method* 中的 *type* 替换为所调用方法的 Java 类型（或使用表中的实际方法调用例程名），同时将 *NativeType* 替换为该例程相应的本地类型。

表 4-4 CallNonvirtual<type>Method 例程

CallNonvirtual<type>Method 例程名	本地类型

CallNonvirtualVoidMethod() CallNonvirtualVoidMethodA() CallNonvirtualVoidMethodV()	void
CallNonvirtualObjectMethod() CallNonvirtualObjectMethodA() CallNonvirtualObjectMethodV()	jobject
CallNonvirtualBooleanMethod() CallNonvirtualBooleanMethodA() CallNonvirtualBooleanMethodV()	jboolean
CallNonvirtualByteMethod() CallNonvirtualByteMethodA() CallNonvirtualByteMethodV()	jbyte
CallNonvirtualCharMethod() CallNonvirtualCharMethodA() CallNonvirtualCharMethodV()	jchar
CallNonvirtualShortMethod() CallNonvirtualShortMethodA() CallNonvirtualShortMethodV()	jshort
CallNonvirtualIntMethod() CallNonvirtualIntMethodA() CallNonvirtualIntMethodV()	jint
CallNonvirtualLongMethod() CallNonvirtualLongMethodA() CallNonvirtualLongMethodV()	jlong
CallNonvirtualFloatMethod() CallNonvirtualFloatMethodA() CallNonvirtualFloatMethodV()	jfloat
CallNonvirtualDoubleMethod() CallNonvirtualDoubleMethodA() CallNonvirtualDoubleMethodV()	jdouble

参数：

env：JNI 接口指针。

clazz：Java 类。

obj: Java 对象。

methodID：方法 ID。

CallNonvirtual<type>Method 例程的其它参数：

要传给 Java 方法的参数。

CallNonvirtual<type>MethodA 例程的其它参数：

args：参数数组。

CallNonvirtual<type>MethodV 例程的其它参数：

args：参数的 va_list。

返回值：

调用 Java 方法的结果。

抛出：

执行 Java 方法时所抛出的异常。

访问静态域

GetStaticFieldID

```
jfieldID GetStaticFieldID(JNIEnv *env, jclass clazz,
const char *name, const char *sig);
```

返回类的静态域的域 ID。域由其名称和签名指定。*GetStatic<type>Field* 和 *SetStatic<type>Field* 访问器函数系列使用域 ID 检索静态域。

GetStaticFieldID() 将未初始化的类初始化。

参数：

env：JNI 接口指针。

clazz：Java 类对象。

name： 0 终结的 UTF-8 字符串中的静态域名。

sig：0 终结的 UTF-8 字符串中的域签名。

返回值：

域 ID。如果找不到指定的静态域，则为 NULL。

抛出：

NoSuchFieldError：如果找不到指定的静态域。

ExceptionInInitializerError：如果由于异常而导致类初始化程序失败。

OutOfMemoryError：如果系统内存不足。

GetStatic<type>Field 例程

NativeType *GetStatic<type>Field*(JNIEnv *env, jclass clazz,
jfieldID fieldID);

该访问器例程系列返回对象的静态域的值。要访问的域由通过调用 GetStaticFieldID() 而得到的域 ID 指定。

下表说明了 *GetStatic<type>Field* 例程名及结果类型。应将 *GetStatic<type>Field* 中的 *type* 替换为域的 Java 类型（或使用表中的某个实际例程名），然后将 *NativeType* 替换为该例程对应的本地类型。

表 4-5 GetStatic<type>Field 访问器例程系列

GetStatic<type>Field 例程名	本地类型
GetStaticObjectField()	jobject
GetStaticBooleanField()	jboolean
GetStaticByteField()	jbyte
GetStaticCharField()	jchar
GetStaticShortField()	jshort
GetStaticIntField()	jint
GetStaticLongField()	jlong
GetStaticFloatField()	jfloat
GetStaticDoubleField()	jdouble

参数：

env：JNI 接口指针。

clazz：Java 类对象。

fieldID：静态域 ID。

返回值：

静态域的内容。

SetStatic<type>Field 例程

void *SetStatic<type>Field*(JNIEnv *env, jclass clazz,
jfieldID fieldID, *NativeType* value);

该访问器例程系列设置对象的静态域的值。要访问的域由通过调用 GetStaticFieldID() 而得到的域 ID 指定。

下表说明了 *SetStatic<type>Field* 例程名及结果类型。应将 *SetStatic<type>Field* 中的 *type* 替换为域的 Java 类型（或使用表中的某个实际例程名），然后将 *NativeType* 替换为该例程对应的本地类型。

表 4-6 SetStatic<type>Field 访问器例程系列

SetStatic<type>Field 例程名	本地类型
SetStaticObjectField()	jobject
SetStaticBooleanField()	jboolean
SetStaticByteField()	jbyte
SetStaticCharField()	jchar
SetStaticShortField()	jshort
SetStaticIntField()	jint
SetStaticLongField()	jlong
SetStaticFloatField()	jfloat
SetStaticDoubleField()	jdouble

参数：

env：JNI 接口指针。

clazz：Java 类对象。

fieldID：静态域 ID。

value：域的新值。

调用静态方法

GetStaticMethodID

```
jmethodID GetStaticMethodID(JNIEnv *env, jclass clazz,
const char *name, const char *sig);
```

返回类的静态方法的方法 ID。方法由其名称和签名指定。

GetStaticMethodID() 将未初始化的类初始化。

参数：

env：JNI 接口指针。

clazz：Java 类对象。

name：0 终结 UTF-8 字符串中的静态方法名。

sig：0 终结 UTF-8 字符串中的方法签名。

返回值：

方法 ID，如果操作失败，则为 NULL。

抛出：

NoSuchMethodError：如果找不到指定的静态方法。

ExceptionInInitializerError：如果由于异常而导致类初始化程序失败。

OutOfMemoryError：如果系统内存不足。

CallStatic<type>Method 例程

CallStatic<type>MethodA 例程

CallStatic<type>MethodV 例程

NativeType *CallStatic<type>Method*(JNIEnv *env, jclass clazz, jmethodID methodID, ...);

NativeType CallStatic<type>MethodA(JNIEnv *env, jclass clazz, jmethodID methodID, jvalue *args);

NativeType CallStatic<type>MethodV(JNIEnv *env, jclass clazz, jmethodID methodID, va_list args);

这些操作将根据指定的方法 ID 调用 Java 对象的静态方法。methodID 参数必须通过调用 GetStaticMethodID() 得到。

方法 ID 必须从 clazz 派生，而不能从其超类派生。

CallStatic<type>Method 例程

编程人员应将要传给方法的所有参数紧跟着放在 methodID 参数之后。*CallStatic<type>Method* routine 接受这些参数并将其传给编程人员所要调用的 Java 方法。

CallStatic<type>MethodA 例程

编程人员应将要传给方法的所有参数放在紧跟在 methodID 参数之后的 jvalues 类型数组 args 中。*CallStaticMethodA* routine 接受这些数组中的参数并将其传给编程人员所要调用的 Java 方法。

CallStatic<type>MethodV 例程

编程人员应将要传给方法的所有参数放在紧跟在 methodID 参数之后的 va_list 类型参数 args 中。*CallStaticMethodV* routine 接受这些参数并将其传给编程人员所要调用的 Java 方法。

下表根据结果类型说明了各个方法调用例程。用户应将 *CallStatic<type>Method* 中的 *type* 替换为所调用方法的 Java 类型（或使用表中的实际方法调用例程名），同时将 *NativeType* 替换为该例程相应的本地类型。

表 4-7 CallStatic<type>Method 调用例程

CallStatic<type>Method 例程名	本地类型
CallStaticVoidMethod() CallStaticVoidMethodA() CallStaticVoidMethodV()	void
CallStaticObjectMethod() CallStaticObjectMethodA()	jobject

CallStaticObjectMethodV()	
CallStaticBooleanMethod() CallStaticBooleanMethodA() CallStaticBooleanMethodV()	jboolean
CallStaticByteMethod() CallStaticByteMethodA() CallStaticByteMethodV()	jbyte
CallStaticCharMethod() CallStaticCharMethodA() CallStaticCharMethodV()	jchar
CallStaticShortMethod() CallStaticShortMethodA() CallStaticShortMethodV()	jshort
CallStaticIntMethod() CallStaticIntMethodA() CallStaticIntMethodV()	jint
CallStaticLongMethod() CallStaticLongMethodA() CallStaticLongMethodV()	jlong
CallStaticFloatMethod() CallStaticFloatMethodA() CallStaticFloatMethodV()	jfloat
CallStaticDoubleMethod() CallStaticDoubleMethodA() CallStaticDoubleMethodV()	jdouble

参数：

env：JNI 接口指针。

clazz：Java 类对象。

methodID：静态方法 ID。

CallStatic<type>Method 例程的其它参数：

要传给静态方法的参数。

CallStatic<type>MethodA 例程的其它参数：

args：参数数组。

CallStatic<type>MethodV 例程的其它参数：

args：参数的 va_list。

返回值：

返回调用静态 Java 方法的结果。

抛出：

执行 Java 方法时抛出的异常。

字符串操作

NewString

```
jstring NewString(JNIEnv *env, const jchar *unicodeChars,
jsize len);
```

利用 Unicode 字符数组构造新的 java.lang.String 对象。

参数：

env：JNI 接口指针。

unicodeChars：指向 Unicode 字符串的指针。

len：Unicode 字符串的长度。

返回值：

Java 字符串对象。如果无法构造该字符串，则为 NULL。

抛出：

OutOfMemoryError：如果系统内存不足。

GetStringLength

```
jsize GetStringLength(JNIEnv *env, jstring string);
```

返回 Java 字符串的长度（Unicode 字符数）。

参数：

env：JNI 接口指针。

string：Java 字符串对象。

返回值：

Java 字符串的长度。

GetStringChars

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

返回指向字符串的 Unicode 字符数组的指针。该指针在调用 ReleaseStringchars() 前一直有效。

如果 isCopy 非空,则在复制完成后将 *isCopy 设为 JNI_TRUE。如果没有复制,则设为 JNI_FALSE。

参数：

env：JNI 接口指针。

string：Java 字符串对象。

isCopy：指向布尔值的指针。

返回值：

指向 Unicode 字符串的指针，如果操作失败，则返回 NULL。

ReleaseStringChars

```
void ReleaseStringChars(JNIEnv *env, jstring string,
const jchar *chars);
```

通知虚拟机平台相关代码无需再访问 chars。参数 chars 是一个指针，可通过 GetStringChars() 从 string 获得。

参数：

env：JNI 接口指针。

string：Java 字符串对象。

chars：指向 Unicode 字符串的指针。

NewStringUTF

jstring NewStringUTF(JNIEnv *env, const char *bytes);

利用 UTF-8 字符数组构造新 java.lang.String 对象。

参数：

env：JNI 接口指针。如果无法构造该字符串，则为 NULL。

bytes：指向 UTF-8 字符串的指针。

返回值：

Java 字符串对象。如果无法构造该字符串，则为 NULL。

抛出：

OutOfMemoryError：如果系统内存不足。

GetStringUTFLength

jsize GetStringUTFLength(JNIEnv *env, jstring string);

以字节为单位返回字符串的 UTF-8 长度。

参数：

env：JNI 接口指针。

string：Java 字符串对象。

返回值：

返回字符串的 UTF-8 长度。

GetStringUTFChars

```
const char* GetStringUTFChars(JNIEnv *env, jstring string,
jboolean *isCopy);
```

返回指向字符串的 UTF-8 字符数组的指针。该数组在被 ReleaseStringUTFChars() 释放前将一直有效。

如果 isCopy 不是 NULL，*isCopy 在复制完成后即被设为 JNI_TRUE。如果未复制，则设为 JNI_FALSE。

参数：

env：JNI 接口指针。

string：Java 字符串对象。

isCopy：指向布尔值的指针。

返回值：

指向 UTF-8 字符串的指针。如果操作失败，则为 NULL。

ReleaseStringUTFChars

```
void ReleaseStringUTFChars(JNIEnv *env, jstring string,
const char *utf);
```

通知虚拟机平台相关代码无需再访问 utf。utf 参数是一个指针，可利用 GetStringUTFChars() 从 string 获得。

参数：

env：JNI 接口指针。

string：Java 字符串对象。

utf：指向 UTF-8 字符串的指针。

数组操作

GetArrayLength

```
jsize GetArrayLength(JNIEnv *env, jarray array);
```

返回数组中的元素数。

参数：

env：JNI 接口指针。

array：Java 数组对象。

返回值：

数组的长度。

NewObjectArray

```
jarray NewObjectArray(JNIEnv *env, jsize length,
jclass elementClass, jobject initialElement);
```

构造新的数组，它将保存类 elementClass 中的对象。所有元素初始值均设为 initialElement。

参数：

env：JNI 接口指针。

length：数组大小。

elementClass：数组元素类。

initialElement：初始值。

返回值：

Java 数组对象。如果无法构造数组，则为 NULL。

抛出：

OutOfMemoryError：如果系统内存不足。

GetObjectArrayElement

```
jobject GetObjectArrayElement(JNIEnv *env,
jobjectArray array, jsize index);
```

返回 Object 数组的元素。

参数：

env：JNI 接口指针。

array：Java 数组。

index：数组下标。

返回值：

Java 对象。

抛出：

ArrayIndexOutOfBoundsException：如果 index 不是数组中的有效下标。

SetObjectArrayElement

```
void SetObjectArrayElement(JNIEnv *env, jobjectArray array,
jsize index, jobject value);
```

设置 Object 数组的元素。

参数：

env：JNI 接口指针。

array：Java 数组。

index：数组下标。

value：新值。

抛出：

ArrayIndexOutOfBoundsException：如果 index 不是数组中的有效下标。

ArrayStoreException：如果 value 的类不是数组元素类的子类。

New<PrimitiveType>Array 例程

ArrayType *New<PrimitiveType>Array*(JNIEnv *env, jsize length);

用于构造新基本类型数组对象的一系列操作。表 4-8 说明了特定的基本类型数组构造函数。用户应把 *New<PrimitiveType>Array* 替换为某个实际的基本类型数组构造函数例程名（见下表），然后将 *ArrayType* 替换为该例程相应的数组类型。

表 4-8 New<PrimitiveType>Array 数组构造函数系列

New<PrimitiveType>Array 例程	数组类型
NewBooleanArray()	jbooleanArray
NewByteArray()	jbyteArray
NewCharArray()	jcharArray
NewShortArray()	jshortArray
NewIntArray()	jintArray
NewLongArray()	jlongArray
NewFloatArray()	jfloatArray
NewDoubleArray()	jdoubleArray

参数：

env：JNI 接口指针。

length：数组长度。

返回值：

Java 数组。如果无法构造该数组，则为 NULL。

Get<PrimitiveType>ArrayElements 例程

NativeType *Get<PrimitiveType>ArrayElements(JNIEnv *env,
ArrayType array, jboolean *isCopy);

一组返回基本类型数组体的函数。结果在调用相应的 *Release<PrimitiveType>ArrayElements()* 函数前将一直有效。*由于返回的数组可能是 Java 数组的副本，因此对返回数组的更改不必在基本类型数组中反映出来，直到调用了* Release<PrimitiveType>ArrayElements()。

如果 isCopy 不是 NULL，*isCopy 在复制完成后即被设为 JNI_TRUE。如果未复制，则设为 JNI_FALSE。

下表说明了特定的基本类型数组元素访问器。应进行下列替换；

- 将 *Get<PrimitiveType>ArrayElements* 替换为表中某个实际的基本类型元素访问器例程名。
- 将 *ArrayType* 替换为对应的数组类型。
- 将 *NativeType* 替换为该例程对应的本地类型。

不管布尔数组在 Java 虚拟机中如何表示，GetBooleanArrayElements() 将始终返回一个 jbooleans 类型的指针，其中每一字节代表一个元素（开包表示）。内存中将确保所有其它类型的数组为连续的。

表 4-9 Get<PrimitiveType>ArrayElements 访问器例程系列

Get<PrimitiveType>ArrayElements 例程	数组类型	本地类型
GetBooleanArrayElements()	jbooleanArray	jboolean
GetByteArrayElements()	jbyteArray	jbyte
GetCharArrayElements()	jcharArray	jchar
GetShortArrayElements()	jshortArray	jshort
GetIntArrayElements()	jintArray	jint
GetLongArrayElements()	jlongArray	jlong
GetFloatArrayElements()	jfloatArray	jfloat
GetDoubleArrayElements()	jdoubleArray	jdouble

参数：

env：JNI 接口指针。

array：Java 字符串对象。

isCopy：指向布尔值的指针。

返回值：

返回指向数组元素的指针，如果操作失败，则为 NULL。

Release<PrimitiveType>ArrayElements 例程

void *Release<PrimitiveType>ArrayElements*(JNIEnv *env,
ArrayType array, *NativeType* *elems, jint mode);

通知虚拟机平台相关代码无需再访问 elems 的一组函数。elems 参数是一个通过使用对应的 *Get<PrimitiveType>ArrayElements()* 函数由 array 导出的指针。必要时，该函数将把对 elems 的修改复制回基本类型数组。

mode 参数将提供有关如何释放数组缓冲区的信息。如果 elems 不是 array 中数组元素的副本，mode 将无效。否则，mode 将具有下表所述的功能：

表 4-10 基本类型数组释放模式

模式	动作
0	复制回内容并释放 elems 缓冲区
JNI_COMMIT	复制回内容但不释放 elems 缓冲区
JNI_ABORT	释放缓冲区但不复制回变化

多数情况下，编程人员将把“0”传给 mode 参数以确保固定的数组和复制的数组保持一致。其它选项可以使编程人员进一步控制内存管理，但使用时务必慎重。

下表说明了构成基本类型数组撤消程序系列的特定例程。应进行如下替换；

- 将 *Release<PrimitiveType>ArrayElements* 替换为表 4-11 中的某个实际基本类型数组撤消程序例程名。
- 将 *ArrayType* 替换为对应的数组类型。
- 将 *NativeType* 替换为该例程对应的本地类型。

表 4-11 Release<PrimitiveType>ArrayElements 数组例程系列

Release<PrimitiveType>ArrayElements 例程	数组类型	本地类型
ReleaseBooleanArrayElements()	jbooleanArray	jboolean

ReleaseByteArrayElements()	jbyteArray	jbyte
ReleaseCharArrayElements()	jcharArray	jchar
ReleaseShortArrayElements()	jshortArray	jshort
ReleaseIntArrayElements()	jintArray	jint
ReleaseLongArrayElements()	jlongArray	jlong
ReleaseFloatArrayElements()	jfloatArray	jfloat
ReleaseDoubleArrayElements()	jdoubleArray	jdouble

参数：

env：JNI 接口指针。

array：Java 数组对象。

elems：指向数组元素的指针。

mode：释放模式。

Get<PrimitiveType>ArrayRegion 例程

void *Get<PrimitiveType>ArrayRegion*(JNIEnv *env, *ArrayType* array, jsize start, jsize len, *NativeType* *buf);

将基本类型数组某一区域复制到缓冲区中的一组函数。

下表说明了特定的基本类型数组元素访问器。应进行如下替换：

- 将 *Get<PrimitiveType>ArrayRegion* 替换为表 4-12 中的某个实际基本类型元素访问器例程名。
- 将 *ArrayType* 替换为对应的数组类型。
- 将 *NativeType* 替换为该例程对应的本地类型。

表 4-12 Get<PrimitiveType>ArrayRegion 数组访问器例程系列

Get<PrimitiveType>ArrayRegion 例程	数组类型	本地类型
GetBooleanArrayRegion()	jbooleanArray	jboolean
GetByteArrayRegion()	jbyteArray	jbyte
GetCharArrayRegion()	jcharArray	jchar
GetShortArrayRegion()	jshortArray	jhort

GetIntArrayRegion()	jintArray	jint
GetLongArrayRegion()	jlongArray	jlong
GetFloatArrayRegion()	jfloatArray	jloat
GetDoubleArrayRegion()	jdoubleArray	jdouble

参数：

env：JNI 接口指针。

array：Java 指针。

start：起始下标。

len：要复制的元素数。

buf：目的缓冲区。

抛出：

ArrayIndexOutOfBoundsException：如果区域中的某个下标无效。

Set<PrimitiveType>ArrayRegion 例程

void *Set<PrimitiveType>ArrayRegion*(JNIEnv *env, *ArrayType* array, jsize start, jsize len, *NativeType* *buf);

将基本类型数组的某一区域从缓冲区中复制回来的一组函数。

下表说明了特定的基本类型数组元素访问器。应进行如下替换：

- 将 *Set<PrimitiveType>ArrayRegion* 替换为表中的实际基本类型元素访问器例程名。
- 将 *ArrayType* 替换为对应的数组类型。
- 将 *NativeType* 替换为该例程对应的本地类型。

表 4-13 Set<PrimitiveType>ArrayRegion 数组访问器例程系列

Set<PrimitiveType>ArrayRegion 例程	数组类型	本地类型
SetBooleanArrayRegion()	jbooleanArray	jboolean
SetByteArrayRegion()	jbyteArray	jbyte

SetCharArrayRegion()	jcharArray	jchar
SetShortArrayRegion()	jshortArray	jshort
SetIntArrayRegion()	jintArray	jint
SetLongArrayRegion()	jlongArray	jlong
SetFloatArrayRegion()	jfloatArray	jfloat
SetDoubleArrayRegion()	jdoubleArray	jdouble

参数：

env：JNI 接口指针。

array: Java 数组。

start：起始下标。

len：要复制的元素数。

buf：源缓冲区。

抛出：

ArrayIndexOutOfBoundsException：如果区域中的某个下标无效。

注册本地方法

RegisterNatives

```
jint RegisterNatives(JNIEnv *env, jclass clazz,
const JNINativeMethod *methods, jint nMethods);
```

向 clazz 参数指定的类注册本地方法。methods 参数将指定 JNINativeMethod 结构的数组，其中包含本地方法的名称、签名和函数指针。nMethods 参数将指定数组中的本地方法数。JNINativeMethod 结构定义如下所示：

```
typedef struct {
    char *name;
    char *signature;
    void *fnPtr;
} JNINativeMethod;
```

函数指针通常必须有下列签名：

```
ReturnType (*fnPtr)(JNIEnv *env, jobject objectOrClass, ...);
```

参数：

env：JNI 接口指针。

clazz：Java 类对象。

methods：类中的本地方法。

nMethods：类中的本地方法数。

返回值：

成功时返回 "0"；失败时返回负数。

抛出：

NoSuchMethodError：如果找不到指定的方法或方法不是本地方法。

UnregisterNatives

```
jint UnregisterNatives(JNIEnv *env, jclass clazz);
```

取消注册类的本地方法。类将返回到链接或注册了本地方法函数前的状态。

该函数不应在常规平台相关代码中使用。相反，它可以为某些程序提供一种重新加载和重新链接本地库的途径。

参数：

env：JNI 接口指针。

clazz：Java 类对象。

返回值：

成功时返回“0”；失败时返回负数。

监视程序操作

MonitorEnter

jint MonitorEnter(JNIEnv *env, jobject obj);

进入与 obj 所引用的基本 Java 对象相关联的监视程序。

每个 Java 对象都有一个相关联的监视程序。如果当前线程已经拥有与 obj 相关联的监视程序，它将使指示该线程进入监视程序次数的监视程序计数器增 1。如果与 obj 相关联的监视程序并非由某个线程所拥有，则当前线程将变为该监视程序的所有者，同时将该监视程序的计数器设置为 1。如果另一个线程已拥有与 obj 关联的监视程序，则在监视程序被释放前当前线程将处于等待状态。监视程序被释放后，当前线程将尝试重新获得所有权。

参数：

env：JNI 接口指针。

obj：常规 Java 对象或类对象。

返回值：

成功时返回“0”；失败时返回负数。

MonitorExit

jint MonitorExit(JNIEnv *env, jobject obj);

当前线程必须是与 obj 所引用的基本 Java 对象相关联的监视程序的所有者。线程将使指示进入监视程序次数的计数器减 1。如果计数器的值变为 0，当前线程释放监视程序。

参数：

env：JNI 接口指针。

obj：常规 Java 对象或类对象。

返回值：

成功时返回“0”；失败时返回负数。

Java 虚拟机接口

GetJavaVM

jint GetJavaVM(JNIEnv *env, JavaVM **vm);

返回与当前线程相关联的 Java 虚拟机接口（用于调用 API 中）。结果将放在第二个参数 vm 所指向的位置。

参数：

env：JNI 接口指针。

vm：指向放置结果的位置的指针。

返回值：

成功时返回“0”；失败时返回负数。

---

5 - 调用 API

调用 API 允许软件厂商将 Java 虚拟机加载到任意的本地程序中。厂商可以交付支持 Java 的应用程序，而不必链接 Java 虚拟机源代码。

本章首先概述了调用 API。然后是所有调用 API 函数的引用页。

若要增强 Java 虚拟机的嵌入性，可以用几种方式来扩展 JDK 1.1.2 中的调用 API。

概述

以下代码示例说明了如何使用调用 API 中的函数。在本例中，C++ 代码创建 Java 虚拟机并且调用名为 Main.test 的静态方法。为清楚起见，我们略去了错误检查。

```
#include <jni.h>        /* 其中定义了所有的事项 */

...

JavaVM *jvm;        /* 表示 Java 虚拟机*/
JNIEnv *env;        /* 指向本地方法接口的指针 */

JDK1_1InitArgs vm_args; /* JDK 1.1 虚拟机初始化参数 */

vm_args.version = 0x00010001; /* 1.1.2 中新增的：虚拟机版本 */
/* 获得缺省的初始化参数并且设置类
 * 路径 */
JNI_GetDefaultJavaVMInitArgs(&vm_args);
vm_args.classpath = ...;

/* 加载并初始化 Java 虚拟机，返回 env 中的
 * JNI 接口指针 */
JNI_CreateJavaVM(&jvm, &env, &vm_args);

/* 用 JNI 调用 Main.test 方法 */
jclass cls = env->FindClass("Main");
jmethodID mid = env->GetStaticMethodID(cls, "test", "(I)V");
```

```
    env->CallStaticVoidMethod(cls, mid, 100);

    /* 结束。*/
    jvm->DestroyJavaVM();
```

本例使用了 API 中的三个函数。调用 API 允许本地应用程序用 JNI 接口指针来访问虚拟机特性。其设计类似于 Netscape 的 JRI 嵌入式接口。

创建虚拟机

JNI_CreateJavaVM() 函数加载并初始化 Java 虚拟机，然后将指针返回到 JNI 接口指针。调用 JNI_CreateJavaVM() 的线程被看作*主线程*。

连接虚拟机

JNI 接口指针 (JNIEnv) 仅在当前线程中有效。如果另一个线程需要访问 Java 虚拟机，则该线程首先必须调用 AttachCurrentThread() 以将自身连接到虚拟机并且获得 JNI 接口指针。连接到虚拟机之后，本地线程的工作方式就与在本地方法内运行的普通 Java 线程一样了。本地线程保持与虚拟机的连接，直到调用 DetachCurrentThread() 时才断开连接。

卸载虚拟机

主线程不能自己断开与虚拟机的连接。而是必须调用 DestroyJavaVM() 来卸载整个虚拟机。

虚拟机等到主线程成为唯一的用户线程时才真正地卸载。用户线程包括 Java 线程和附加的本地线程。之所以存在这种限制是因为 Java 线程或附加的本地线程可能正占用着系统资源，例如锁，窗口等。虚拟机不能自动释放这些资源。卸载虚拟机时，通过将主线程限制为唯一的运行线程，使释放任意线程所占用系统资源的负担落到程序员身上。

初始化结构

不同的 Java 虚拟机实现可能会需要不同的初始化参数。很难提出适合于所有现有和将来的 Java 虚拟机的标准初始化结构。作为一种折衷方式，我们保留了第一个域 (version) 来识别初始化结构的内容。嵌入到 JDK 1.1.2 中的本地应用程序必须将版本域设置为 0x00010001。尽管其它实现可能会忽略某些由 JDK 所支持的初始化参数，我们仍然鼓励虚拟机实现使用与 JDK 一样的初始化结构。

0x80000000 到 0xFFFFFFFF 之间的版本号需保留，并且不为任何虚拟机实现所识别。

以下代码显示了初始化 JDK 1.1.2 中的 Java 虚拟机所用的结构。

```
typedef struct JavaVMInitArgs {
   /* 前两个域在 JDK 1.1 中保留，并
      在 JDK 1.1.2 中正式引入。*/
   /* Java 虚拟机版本 */
    jint version;

   /* 系统属性。*/
    char **properties;

   /* 是否检查 Java 源文件与已编译的类文件
    *之间的新旧关系。*/
    jint checkSource;

   /* Java 创建的线程的最大本地堆栈大小。*/
    jint nativeStackSize;

   /* 最大 Java 堆栈大小。*/
    jint javaStackSize;

   /* 初始堆大小。*/
    jint minHeapSize;

   /* 最大堆大小。*/
    jint maxHeapSize;

   /* 控制是否校验 Java 字节码：
    * 0 无，1 远程加载的代码，2 所有代码。*/
    jint verifyMode;

   /* 类加载的本地目录路径。*/
    const char *classpath;

   /* 重定向所有虚拟机消息的函数的钩子。*/
    jint (*vfprintf)(FILE *fp, const char *format,
```

```
                        va_list args);

    /* 虚拟机退出钩子。*/
     void (*exit)(jint code);

    /* 虚拟机放弃钩子。*/
     void (*abort)();

    /* 是否启用类 GC。*/
     jint enableClassGC;

    /* GC 消息是否出现。*/
     jint enableVerboseGC;

    /* 是否允许异步 GC。*/
     jint disableAsyncGC;

    /* 三个保留的域。*/
     jint reserved0;
     jint reserved1;
     jint reserved2;
} JDK1_1InitArgs;
```

在 JDK 1.1.2 中，初始化结构提供了钩子，这样在虚拟机终止时，本地应用程序可以重定向虚拟机消息并获得控制权。

当本地线程与 JDK 1.1.2 中的 Java 虚拟机连接时，以下结构将作为参数进行传递。实际上，本地线程与 JDK 1.1.2 连接时不需要任何参数。
JDK1_1AttachArgs 结构仅由 C 编译器的填充槽组成，而 C 编译器不允许空结构。

```
typedef struct JDK1_1AttachArgs {
    /*
     * JDK 1.1 不需要任何参数来附加
     * 本地线程。此处填充的作用是为了满足不允许空结构的 C
     * 编译器的要求。
     */
     void *__padding;
} JDK1_1AttachArgs;
```

调用 API 函数

JavaVM 类型是指向调用 API 函数表的指针。以下代码示例显示了这种函数表。

```
typedef const struct JNIInvokeInterface *JavaVM;

const struct JNIInvokeInterface ... = {
    NULL,
    NULL,
    NULL,

    DestroyJavaVM,
    AttachCurrentThread,
    DetachCurrentThread,
};
```

注意，JNI_GetDefaultJavaVMInitArgs()、JNI_GetCreatedJavaVMs() 和 JNI_CreateJavaVM() 这三个调用 API 函数不是 JavaVM 函数表的一部分。不必先有 JavaVM 结构，就可以使用这些函数。

JNI_GetDefaultJavaVMInitArgs

```
jint JNI_GetDefaultJavaVMInitArgs(void *vm_args);
```

返回 Java 虚拟机的缺省配置。在调用该函数之前，平台相关代码必须将 vm_args->version 域设置为它所期望虚拟机支持的 JNI 版本。在 JDK 1.1.2 中，必须将 vm_args->version 设置为 0x00010001。（JDK 1.1 不要求平台相关代码设置版本域。为了向后兼容性，如果没有设置版本域，则 JDK 1.1.2 假定所请求的版本为 0x00010001。JDK 的未来版本将要求把版本域设置为适当的值。）该函数返回后,将把 vm_args->version 设置为虚拟机支持的实际 JNI 版本。

参数：

vm_args：指向 VM-specific initialization（特定于虚拟机的初始化）结构的指针，缺省参数填入该结构。

返回值：

如果所请求的版本得到支持，则返回“0”；如果所请求的版本未得到支持，则返回负数。

JNI_GetCreatedJavaVMs

```
jint JNI_GetCreatedJavaVMs(JavaVM **vmBuf, jsize bufLen,
jsize *nVMs);
```

返回所有已创建的 Java 虚拟机。将指向虚拟机的指针依据其创建顺序写入 vmBuf 缓冲区。最多写入 bufLen 项。在 *nVMs 中返回所创建虚拟机的总数。

JDK 1.1 不支持在单个进程中创建多个虚拟机。

参数：

vmBuf：指向将放置虚拟机结构的缓冲区的指针。

bufLen：缓冲区的长度。

nVMs：指向整数的指针。

返回值：

成功时返回“0”；失败则返回负数。

JNI_CreateJavaVM

```
jint JNI_CreateJavaVM(JavaVM **p_vm, JNIEnv **p_env,
void *vm_args);
```

加载并初始化 Java 虚拟机。当前线程成为主线程。将 env 参数设置为主线程的 JNI 接口指针。

JDK 1.1.2 不支持在单个进程中创建多个虚拟机。必须将 vm_args 中的版本域设置为 0x00010001。

参数：

p_vm：指向位置（其中放置所得到的虚拟机结构）的指针。

p_env：指向位置（其中放置主线程的 JNI 接口指针）的指针。

vm_args: Java 虚拟机初始化参数。

返回值：

成功时返回“0”；失败则返回负数。

DestroyJavaVM

```
jint DestroyJavaVM(JavaVM *vm);
```

卸载 Java 虚拟机并回收资源。只有主线程能够卸载虚拟机。调用 DestroyJavaVM() 时，主线程必须是唯一的剩余用户线程。

参数：

vm：将销毁的 Java 虚拟机。

返回值：

成功时返回“0”；失败则返回负数。

JDK 1.1.2 不支持卸载虚拟机。

AttachCurrentThread

```
jint AttachCurrentThread(JavaVM *vm, JNIEnv **p_env,
void *thr_args);
```

将当前线程连接到 Java 虚拟机。在 JNIEnv 参数中返回 JNI 接口指针。

试图连接已经连接的线程将不执行任何操作。

本地线程不能同时连接到两个 Java 虚拟机上。

参数：

vm：当前线程所要连接到的虚拟机。

p_env：指向位置（其中放置当前线程的 JNI 接口指针）的指针。

thr_args：特定于虚拟机的线程连接参数。

返回值：

成功时返回“0”；失败则返回负数。

DetachCurrentThread

jint DetachCurrentThread(JavaVM *vm);

断开当前线程与 Java 虚拟机之间的连接。释放该线程占用的所有 Java 监视程序。通知所有等待该线程终止的 Java 线程。

主线程（即创建 Java 虚拟机的线程）不能断开与虚拟机之间的连接。作为替代，主线程必须调用 JNI_DestroyJavaVM() 来卸载整个虚拟机。

参数：

vm：当前线程将断开连接的虚拟机。

返回值：

成功时返回“0”；失败则返回负数。

CreateFile

(1)函数原型
HANDLE CreateFile(
LPCTSTR lpfileName,
DWORD deDesiredAccess,
DWORD dwShareMode,
LPSECURITY_ATTRIBUTES lpSecurityAttributes
DWORD dwCreationDesposition,
DWORD dwFlagsAndAtrributes,
HANDLE hTemplateFile
);
(2)函数说明
该函数创建、打开或截断一个文件，并返回一个能够被用来存取该文件的句柄。此句柄允许读书据、写数据以及移动文件的指针。CreateFile 函数既可以做为一个宽自负函数使用，也可以作为一个 ANSI 函数来用。
(3)参数说明
lpFileName:指向文件字符串的指针。
dwDesireAccess:制定文件的存取模式，可以取下列值：
　0:制定可以查询对象。

GENERIC_READ:指定可以从文件中度去数据。
GENERIC_WRITE:指定可以向文件中写数据。
dwShareMode:指定文件的共享模式，可以取下列值：
0:不共享。
FILE_SHARE_DELETE:在 Windows NT 系统中，只有为了删除文件而进行的打开操作才会成功。
FILE_SHARE_READ:只有为了从文件中度去数据而进行的打开操作才会成功。
FILE_SHARE_WRITE:只有为了向文件中写数据而进行的打开操作才会成功。
lpSecurityAttributes:指定文件的安全属性。
dwCreationDisopsition:指定创建文件的方式，可以取以下值：
CREATE_NEW:创建新文件，如果文件已存在，则函数失败。
CREATE_ALWAYS:创建爱内心文件，如果文件已存在，则函数将覆盖并清除旧文件。
OPEN_EXISTING:打开文件，如果文件不存在，函数将失败。
OPEN_ALWAYS:打开文件，如果文件不存在，则函数将创建一个新文件。
TRUNCATE_EXISTING:打开外呢间，如果文件存在，函数将文件的大小设为零,如果文件不存在，函数将失败返回。
dwFlagsAndAtrributes:指定新建文件的属性和标志，它可以取以下值：
FILE_ATTRIBUTE_ARCHIVE:归档属性。
FILE_ATTRIBUTE_HIDDEN:隐藏属性。
FILE_ATTRIBUTE_NORMAL:正常属性。
FILE_ATTRIBUTE_READONLY:只读属性。
FILE_ATTRIBUTE_SYSTEM:系统文件。
FILE_ATTRIBUTE_TEMPORARY:临时存储文件，系统总是将临时文件的所有数据读入内存中，以加速对该文件的访问速度。用户应该尽快删除不再使用的临时文件。
FILE_FLAG_OVERLAPPED:用户可以通过一个 OVERLAPPED 结构变量来保存和设置文件读写指针，从而可以完成重叠式的读和写。一般用于数量比较大的读些操作。
hTemplateFile:指向一个具有 GENERIC_READ 属性的文件的句柄，该文件为要创建的文件提供文件属性和文件扩展属性。
(4)注意事项
函数成功将返回指定文件的句柄，否则返回 NULL。
(5)典型示例：

```
...
char szFile[64];
HANDLE handle;
unsigned long lWritten,lRead;

handle =
CreateFile("c:\\windows\\desktop\\example.txt",GENERIC_READ|GENERIC_WRITE,FILE_SHARE_READ,NULL,OPEN_ALWAYS,FILE_ATTRIBUTE_NORMAL,0);
```

```
if(handle==INVALID_HANDLE_VALUE){
   MessageBox(NULL,"Error Create File!","Error",MB_OK);
   break;
}else
   MessageBox(NULL,"Open file Success!","Open file",MB_OK);
```