Java Native Interface (JNI)标准是java平台的一部分，它允许Java代码和其他语言写的代码进行交互。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交互操作。

1.从如何载入.so 档案谈起

由于Android的应用层的类都是以Java写的，这些Java类编译为Dex型式的Bytecode之后，必须靠Dalvik虚拟机(VM: Virtual Machine)来执行。VM在Android平台里，扮演很重要的角色。

此外，在执行Java类的过程中，如果Java类需要与C组件沟通时，VM就会去载入C组件，然后让Java的函数顺利地调用到C组件的函数。此时，VM扮演着桥梁的角色，让Java与C组件能通过标准的JNI介面而相互沟通。

应用层的Java类是在虚拟机(VM: Vitual Machine)上执行的，而C件不是在VM上执行，那么Java程式又如何要求VM去载入(Load)所指定的C组件呢？可使用下述指令：

System.loadLibrary(*.so的档案名)；

例如，Android框架里所提供的MediaPlayer.java类，含指令：

java代码：

```
1.  public class MediaPlayer{
2.  static {
3.  System.loadLibrary(“media_jni”);
4.  }
5.  }
```

复制代码

这要求VM去载入Android的/system/lib/libmedia_jni.so档案。载入*.so之后，Java类与*.so档案就汇合起来，一起执行了。

2.如何撰写*.so的入口函数

—— JNI_OnLoad()与JNI_OnUnload()函数的用途

当Android的VM(Virtual Machine)执行到System.loadLibrary()函数时，首先会去执行C组件里的JNI_OnLoad()函数。它的用途有二：

(1)告诉VM此C组件使用那一个JNI版本。如果你的*.so档没有提供JNI_OnLoad()函数，VM会默认该*.so档是使用最老的JNI 1.1版本。由于新版的JNI做了许多扩充，如果需要使用JNI的新版功能，例如JNI 1.4的java.nio.ByteBuffer,就必须藉由JNI_OnLoad()函数来告知VM。

(2)由于VM执行到System.loadLibrary()函数时，就会立即先呼叫JNI_OnLoad()，所以C组件的开发者可以藉由JNI_OnLoad()来进行C组件内的初期值之设定(Initialization) 。

例如，在Android的/system/lib/libmedia_jni.so档案里，就提供了JNI_OnLoad()函数，其程式码片段为：

java 代码：

```
1.  //#define LOG_NDEBUG 0
2.  #define LOG_TAG “MediaPlayer-JNI”
3.  jint JNI_OnLoad(JavaVM* vm, void* reserved)
4.  {
5.  JNIEnv* env = NULL;
6.  jint result = -1;
7.  if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {
8.  LOGE(“ERROR: GetEnv failed\n”);
9.  goto bail;
10. }
11. assert(env != NULL);
12. if (register_android_media_MediaPlayer(env) < 0) {
13. LOGE("ERROR: MediaPlayer native registration failed\n");
14. goto bail;
15. }
16. if (register_android_media_MediaRecorder(env) < 0) {
17. LOGE("ERROR: MediaRecorder native registration failed\n");
18. goto bail;
19. }
20. if (register_android_media_MediaScanner(env) < 0) {
21. LOGE("ERROR: MediaScanner native registration failed\n");
22. goto bail;
23. }
24. if (register_android_media_MediaMetadataRetriever(env) < 0) {
25. LOGE("ERROR: MediaMetadataRetriever native registration failed\n");
26. goto bail;
27. }
28. /* success -- return valid version number */
29. result = JNI_VERSION_1_4;
30. bail:
31. return result;
32. }
```

此函数回传JNI_VERSION_1_4值给VM，于是VM知道了其所使用的JNI版本了。此外，它也做了一些初期的动作(可呼叫任何本地函数)，例如指令：

java 代码：

```
1.  if (register_android_media_MediaPlayer(env) < 0) {
```

```
2.  LOGE("ERROR: MediaPlayer native registration failed\n");
3.  goto bail;
4.  }
```

复制代码

就将此组件提供的各个本地函数(Native Function)登记到 VM 里，以便能加快后续呼叫本地函数的效率。
JNI_OnUnload()函数与 JNI_OnLoad()相对应的。在载入 C 组件时会立即呼叫 JNI_OnLoad()来进行组件内的初期动作；而当 VM 释放该 C 组件时，则会呼叫 JNI_OnUnload()函数来进行善后清除动作。当 VM 呼叫 JNI_OnLoad()或 JNI_Unload()函数时，都会将 VM 的指针(Pointer)传递给它们，其参数如下：

java 代码：

```
1.  jint JNI_OnLoad(JavaVM* vm, void* reserved) { }
2.  jint JNI_OnUnload(JavaVM* vm, void* reserved){ }
```

复制代码

在 JNI_OnLoad()函数里，就透过 VM 之指标而取得 JNIEnv 之指标值，并存入 env 指标变数里，如下述指令：

java 代码：

```
1.  jint JNI_OnLoad(JavaVM* vm, void* reserved){
2.  JNIEnv* env = NULL;
3.  jint result = -1;
4.  if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {
5.  LOGE(“ERROR: GetEnv failed\n”);
6.  goto bail;
7.  }
8.  }
```

复制代码

由于 VM 通常是多执行绪(Multi-threading)的执行环境。每一个执行绪在呼叫 JNI_OnLoad()时，所传递进来的 JNIEnv 指标值都是不同的。为了配合这种多执行绪的环境，C 组件开发者在撰写本地函数时，可藉由 JNIEnv 指标值之不同而避免执行绪的资料冲突问题，才能确保所写的本地函数能安全地在 Android 的多执行绪 VM 里安全地执行。基于这个理由，当在呼叫 C 组件的函数时，都会将 JNIEnv 指标值传递给它，如下：

java 代码：

```
1.  jint JNI_OnLoad(JavaVM* vm, void* reserved)
```

```
2.  {
3.  JNIEnv* env = NULL;
4.  if (register_android_media_MediaPlayer(env) < 0) {
5.  }
6.  }
```

复制代码

这 JNI_OnLoad()呼叫 register_android_media_MediaPlayer(env)函数时，就将 env 指标值传递过去。如此，在 register_android_media_MediaPlayer()函数就能藉由该指标值而区别不同的执行绪，以便化解资料冲突的问题。

例如，在 register_android_media_MediaPlayer()函数里，可撰写下述指令：

java 代码：

```
1.  if ((*env)->MonitorEnter(env, obj) != JNI_OK) {
2.  }
```

复制代码

查看是否已经有其他执行绪进入此物件，如果没有，此执行绪就进入该物件里执行了。还有，也可撰写下述指令：

java 代码：

```
1.  if ((*env)->MonitorExit(env, obj) != JNI_OK) {
2.  }
```

复制代码

查看是否此执行绪正在此物件内执行，如果是，此执行绪就会立即离开。

3.registerNativeMethods()函数的用途

应用层级的 Java 类别透过 VM 而呼叫到本地函数。一般是依赖 VM 去寻找*.so 里的本地函数。如果需要连续呼叫
很多次，每次都需要寻找一遍，会多花许多时间。此时，组件开发者可以自行将本地函数向 VM 进行登记。例
如，在 Android 的/system/lib/libmedia_jni.so 档案里的代码段如下：

java 代码：

```
1.  //#define LOG_NDEBUG 0
2.  #define LOG_TAG “MediaPlayer-JNI”
```

```
3.  static JNINativeMethod gMethods[] = {
4.  {“setDataSource”,  “(Ljava/lang/String;)V”,
5.  (void *)android_media_MediaPlayer_setDataSource},
6.  {“setDataSource”,  “(Ljava/io/FileDescriptor;JJ)V”,
7.  (void *)android_media_MediaPlayer_setDataSourceFD},
8.  {“prepare”,  “()V”,  (void *)android_media_MediaPlayer_prepare},
9.  {“prepareAsync”,  “()V”,  (void
    *)android_media_MediaPlayer_prepareAsync},
10. {“_start”,  “()V”,  (void *)android_media_MediaPlayer_start},
11. {“_stop”,  “()V”,  (void *)android_media_MediaPlayer_stop},
12. {“getVideoWidth”,  “()I”,  (void
    *)android_media_MediaPlayer_getVideoWidth},
13. {“getVideoHeight”,  “()I”,  (void
    *)android_media_MediaPlayer_getVideoHeight},
14. {“seekTo”,  “(I)V”,  (void *)android_media_MediaPlayer_seekTo},
15. {“_pause”,  “()V”,  (void *)android_media_MediaPlayer_pause},
16. {“isPlaying”,  “()Z”,  (void *)android_media_MediaPlayer_isPlaying},
17. {“getCurrentPosition”,  “()I”,  (void
    *)android_media_MediaPlayer_getCurrentPosition},
18. {“getDuration”,  “()I”,  (void
    *)android_media_MediaPlayer_getDuration},
19. {“_release”,  “()V”,  (void *)android_media_MediaPlayer_release},
20. {“_reset”,  “()V”,  (void *)android_media_MediaPlayer_reset},
21. {“setAudioStreamType”,”(I)V”,  (void
    *)android_media_MediaPlayer_setAudioStreamType},
22. {“setLooping”,  “(Z)V”,  (void
    *)android_media_MediaPlayer_setLooping},
23. {“setVolume”,  “(FF)V”,  (void *)android_media_MediaPlayer_setVolume},
24. {“getFrameAt”,  “(I)Landroid/graphics/Bitmap;”,
25. (void *)android_media_MediaPlayer_getFrameAt},
26. {“native_setup”,  “(Ljava/lang/Object;)V”,
27. (void *)android_media_MediaPlayer_native_setup},
28. {“native_finalize”,  “()V”,  (void
    *)android_media_MediaPlayer_native_finalize},
29. };
30. static int register_android_media_MediaPlayer(JNIEnv *env){
31. return AndroidRuntime::registerNativeMethods(env,
32.  “android/media/MediaPlayer”, gMethods, NELEM(gMethods));
33. }
34. jint JNI_OnLoad(JavaVM* vm, void* reserved){
35. if (register_android_media_MediaPlayer(env) < 0) {
36. LOGE("ERROR: MediaPlayer native registration failed\n");
37. goto bail;
38. }
```

```
39. }
```

复制代码

当 VM 载入 libmedia_jni.so 档案时，就呼叫 JNI_OnLoad()函数。接着，JNI_OnLoad()呼叫 register_android_media_MediaPlayer()函数。此时，就呼叫到 AndroidRuntime::registerNativeMethods()函数，向 VM(即 AndroidRuntime)登记 gMethods[]表格所含的本地函数了。简而言之，registerNativeMethods()函数的用途有二：(1)更有效率去找到函数。(2)可在执行期间进行抽换。由于 gMethods[]是一个<名称，函数指针>对照表，在程序执行时，可多次呼叫 registerNativeMethods()函数来更换本地函数之指针，而达到弹性抽换本地函数之目的。

4.Andoird 中使用了一种不同传统 Java JNI 的方式来定义其 native 的函数。

其中很重要的区别是 Andorid 使用了一种 Java 和 C 函数的映射表数组，并在其中描述了函数的参数和返回值。这个数组的类型是 JNINativeMethod，定义如下：

java 代码：

```
1. typedef struct {
2. const char* name; /*Java 中函数的名字*/
3. const char* signature; /*描述了函数的参数和返回值*/
4. void* fnPtr; /*函数指针，指向 C 函数*/
5. } JNINativeMethod;
```

复制代码

其中比较难以理解的是第二个参数，例如

java 代码：

```
1. “()V”
2. “(II)V”
3. “(Ljava/lang/String;Ljava/lang/String;)V”
```

复制代码

实际上这些字符是与函数的参数类型一一对应的。
“()” 中的字符表示参数，后面的则代表返回值。例如”()V” 就表示 void Func();

“(II)V” 表示 void Func(int, int);

具体的每一个字符的对应关系如下

字符 Java类型 C类型

java 代码：

```
1. V void void
2. Z jboolean boolean
3. I jint int
4. J jlong long
5. D jdouble double
6. F jfloat float
7. B jbyte byte
8. C jchar char
9. S jshort short
```

复制代码

数组则以”[″开始，用两个字符表示

java 代码：

```
1. I jintArray int[]
2. [F jfloatArray float[]
3. [B jbyteArray byte[]
4. [C jcharArray char[]
5. [S jshortArray short[]
6. [D jdoubleArray double[]
7. [J jlongArray long[]
8. [Z jbooleanArray boolean[]
```

复制代码

上面的都是基本类型。如果Java函数的参数是class，则以”L”开头，以”;”结尾，中间是用”/” 隔开的包及类名。而其对应的C函数名的参数则为 jobject. 一个例外是String类，其对应的类为 jstring

Ljava/lang/String; String jstring

Ljava/net/Socket; Socket jobject

如果JAVA函数位于一个嵌入类，则用$作为类名间的分隔符。

例如 “(Ljava/lang/String;Landroid/os/FileUtils$FileStatus;)Z”

一、直接使用 java 本身 jni 接口（windows/ubuntu）

1.在Eclipsh中新建一个android应用程序。两个类：一个继承于Activity，UI显示用。另一个包含native方法。编译生成所有类。

java 代码：

```
1.  package com.hello.jnitest;
2.
3.
4.  import android.app.Activity;
5.  import android.os.Bundle;
6.  public class jnitest extends Activity {
7.  /** Called when the activity is first created. */
8.
9.
10. @Override
11. public void onCreate(Bundle savedInstanceState) {
12. super.onCreate(savedInstanceState);
13. setContentView(R.layout.main);
14. Nadd cal = new Nadd();
15. setTitle("The Native Add Result is " +String.valueOf(cal.nadd(10, 19)));
16. }
17. }
```

复制代码

java 代码：

```
1.  package com.hello.jnitest;
2.
3.
4.  public class Nadd {
5.  static {
6.  System.loadLibrary("Nadd");
7.  }
8.  public native int nadd(int a, int b);
9.  }
```

复制代码

2. 使用 javah 命令生成 C/C++的.h 文件。注意类要包含包名，路径文件夹下要包含所有包中的类，否则会报找不到类的错误。classpath 参数指定到包名前一级文件夹，文件夹层次结构要符合 java 类的组织层次结构。

java 代码：

```
1.  javah -classpath ../jnitest/bin com.hello.jnitest.Nadd
2.  com_hello_jnitest_Nadd .h 文件：
3.  /* DO NOT EDIT THIS FILE - it is machine generated */
4.  #include
5.  /* Header for class com_hello_jnitest_Nadd */
```

```
#ifndef _Included_com_hello_jnitest_Nadd
#define _Included_com_hello_jnitest_Nadd
#ifdef __cplusplus
extern “C” {
#endif
/*
* Class: com_hello_jnitest_Nadd
* Method: nadd
* Signature: (II)I
*/
JNIEXPORT jint JNICALL Java_com_hello_jnitest_Nadd_nadd
(JNIEnv *, jobject, jint, jint);
#ifdef __cplusplus
}
#endif
#endif
```

复制代码

3.编辑.c 文件实现 native 方法。

com_hello_jnitest_Nadd.c 文件：

java 代码：

```
#include
#include “com_hello_jnitest_Nadd.h”
JNIEXPORT jint JNICALL Java_com_hello_jnitest_Nadd_nadd(JNIEnv * env,
jobject c, jint a, jint b)
{
return (a+b);
}
```

复制代码

4.编译.c 文件生存动态库。

java 代码：

```
arm-none-linux-gnueabi-gcc -I/home/a/work/android/jdk1.6.0_17/include
-I/home/a/work/android/jdk1.6.0_17/include/linux -fpic -c
com_hello_jnitest_Nadd.c
arm-none-linux-gnueabi-ld
-T/home/a/CodeSourcery/Sourcery_G++_Lite/arm-none-linux-gnueabi/lib/lds
cripts/armelf_linux_eabi.xsc -share -o libNadd.so
com_hello_jnitest_Nadd.o
```

复制代码

得到 libNadd.so 文件。以上在 ubuntu 中完成。

5.将相应的动态库文件 push 到 avd 的 system/lib 中:adb push libNadd.so /system/lib。若提示 Read-only file system 错误，运行 adb remount 命令，即可。
Adb push libNadd.so /system/lib

6.在 eclipsh 中运行原应用程序即可。
以上在 windows 中完成。
对于一中生成的 so 文件也可采用二中的方法编译进 apk 包中。只需在工程文件夹中建 libs\armeabi 文件夹（其他文件夹名无效，只建立 libs 文件夹也无效），然后将 so 文件拷入，编译工程即可。

二.使用 NDK 生成本地方法（ubuntu and windows）
1.安装 NDK：解压，然后进入 NDK 解压后的目录，运行 build/host-setup.sh（需要 Make 3.81 和 awk）。若有错，修改 host-setup.sh 文件：将#!/bin/sh 修改为#!/bin/bash，再次运行即可。

2.在 apps 文件夹下建立自己的工程文件夹,然后在该文件夹下建一文件 Application.mk 和项 project 文件夹。
Application.mk 文件：
APP_PROJECT_PATH := $(call my-dir)/project
APP_MODULES := myjni

3.在 project 文件夹下建一 jni 文件夹，然后新建 Android.mk 和 myjni.c。这里不需要用 javah 生成相应的.h 文件，但函数名要包含相应的完整的包、类名。

4.编辑相应文件内容。
Android.mk 文件：

java 代码：

```
1.  # Copyright (C) 2009 The Android Open Source Project
2.  #
3.  # Licensed under the Apache License, Version 2.0 (the “License”);
4.  # you may not use this file except in compliance with the License.
5.  # You may obtain a copy of the License at
6.  #
7.  # http://www.apache.org/licenses/LICENSE-2.0
8.  #
9.  # Unless required by applicable law or agreed to in writing, software
10. # distributed under the License is distributed on an “AS IS” BASIS,
11. # WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
```

```
# See the License for the specific language governing permissions and
# limitations under the License.
#
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE := myjni
LOCAL_SRC_FILES := myjni.c
include $(BUILD_SHARED_LIBRARY)
myjni.c 文件：
#include
#include
jstring
Java_com_hello_NdkTest_NdkTest_stringFromJNI( JNIEnv* env,
jobject thiz )
{
return (*env)->NewStringUTF(env, “Hello from My-JNI !”);
}
```

复制代码

myjni 文件组织：

java 代码：

```
a@ubuntu:~/work/android/ndk-1.6_r1/apps$ tree myjni
myjni
|- Application.mk
`- project
|- jni
| |- Android.mk
| `- myjni.c
`- libs
`- armeabi
`- libmyjni.so
```

复制代码

4 directories, 4 files

5.编译：make APP=myjni.

以上内容在 ubuntu 完成。以下内容在 windows 中完成。当然也可以在 ubuntu 中完成。

6.在 eclipsh 中创建 android application。将 myjni 中自动生成的 libs 文件夹拷贝到当前工程文件夹中，编译运行即可。

NdkTest.java 文件：

java 代码：

```
1.  package com.NdkTest;
2.
3.
4.  import android.app.Activity;
5.  import android.os.Bundle;
6.  import android.widget.TextView;
7.  public class NdkTest extends Activity {
8.  /** Called when the activity is first created. */
9.  @Override
10. public void onCreate(Bundle savedInstanceState) {
11. super.onCreate(savedInstanceState);
12. TextView tv = new TextView(this);
13. tv.setText( stringFromJNI() );
14. setContentView(tv);
15. }
16. public native String stringFromJNI();
17. static {
18. System.loadLibrary(“myjni”);
19. }
20. }
```

复制代码

对于二中生成的 so 文件也可采用一中的方法 push 到 avd 中运行。

本篇将介绍在 JNI 编程中如何传递参数和返回值。

首先要强调的是，native 方法不但可以传递 Java 的基本类型做参数，还可以传递更复杂的类型，比如 String，数组，甚至自定义的类。这一切都可以在 jni.h 中找到答案。

1. Java 基本类型的传递

用过 Java 的人都知道，Java 中的基本类型包括 boolean，byte，char，short，int，long，float，double 这样几种，如果你用这几种类型做 native 方法的参数，当你通过 javah -jni 生成.h 文件的时候，只要看一下生成的.h 文件，就会一清二楚，这些类型分别对应的类型是 jboolean，jbyte，jchar，jshort，jint，jlong，jfloat，jdouble 。这几种类型几乎都可以当成对应的 C++类型来用，所以没什么好说的。

2. String 参数的传递

Java 的 String 和 C++的 string 是不能对等起来的，所以处理起来比较麻烦。先看一个例子，

Java 代码：

```
1.  class Prompt {
2.
3.  // native method that prints a prompt and reads a line
4.  private native String getLine(String prompt);
5.
6.  public static void main(String args[]) {
7.  Prompt p = new Prompt();
8.  String input = p.getLine("Type a line: ");
9.  System.out.println("User typed: " + input);
10. }
11.
12. static {
13. System.loadLibrary("Prompt");
14. }
15. }
```

复制代码

在这个例子中，我们要实现一个 native 方法 String getLine(String prompt);

读入一个 String 参数，返回一个 String 值。
通过执行 javah -jni 得到的头文件是这样的

Java 代码：

```
1.  #include
2.
3.  #ifndef _Included_Prompt
4.  #define _Included_Prompt
5.  #ifdef __cplusplus
6.
7.  extern "C" {
8.  #endif
9.
10. JNIEXPORT jstring JNICALL Java_Prompt_getLine(JNIEnv *env, jobject this,
    jstring prompt);
11.
12. #ifdef __cplusplus
13. }
14.
15. #endif
16. #endif
```

复制代码

jstring 是 JNI 中对应于 String 的类型，但是和基本类型不同的是，jstring 不能直接当作 C++的 string 用。如果你用 cout << prompt << endl;

编译器肯定会扔给你一个错误信息的。

其实要处理 jstring 有很多种方式，这里只讲一种我认为最简单的方式，看下面这个例子,

Java 代码：

```
1.  #include "Prompt.h"
2.  #include
3.
4.  JNIEXPORT jstring JNICALL Java_Prompt_getLine(JNIEnv *env, jobject obj,
    jstring prompt){
5.
6.  const char* str;
7.  str = env->GetStringUTFChars(prompt, false);
8.
9.  if(str == NULL) {
10. return NULL;
11. }
12.
13. std::cout << str << std::endl;
14. env->ReleaseStringUTFChars(prompt, str);
15. char* tmpstr = "return string succeeded";
16. jstring rtstr = env->NewStringUTF(tmpstr);
17. return rtstr;
18. }
```

复制代码

在上面的例子中，作为参数的 prompt 不能直接被 C++程序使用，先做了如下转换

str = env->GetStringUTFChars(prompt, false);

将 jstring 类型变成一个 char*类型。

返回的时候，要生成一个 jstring 类型的对象，也必须通过如下命令，

jstring rtstr = env->NewStringUTF(tmpstr);

这里用到的 GetStringUTFChars 和 NewStringUTF 都是 JNI 提供的处理 String 类型的函数，还有其他的函数这里就不一一列举了。

3. 数组类型的传递

和 String 一样，JNI 为 Java 基本类型的数组提供了 j*Array 类型，比如 int[]对应的就是

jintArray。来看一个传递 int 数组的例子，

Java 代码：

```
1.  JNIEXPORT jint JNICALL Java_IntArray_sumArray(JNIEnv *env, jobject obj,
    jintArray arr){
2.
3.  jint *carr;
4.  carr = env->GetIntArrayElements(arr, false);
5.
6.  if(carr == NULL) {
7.  return 0;
8.  }
9.
10. jint sum = 0;
11. for(int i=0; i<10; i++) {
12. sum += carr[i];
13. }
14.
15. env->ReleaseIntArrayElements(arr, carr, 0);
16. return sum;
17. }
```

复制代码

这个例子中的GetIntArrayElements和ReleaseIntArrayElements函数就是JNI提供用于处理 int 数组的函数。如果试图用 arr 的方式去访问 jintArray 类型，毫无疑问会出错。JNI 还提供了另一对函数 GetIntArrayRegion 和 ReleaseIntArrayRegion 访问 int 数组，就不介绍了，对于其他基本类型的数组，方法类似。

4. 二维数组和 String 数组

在JNI中，二维数组和String数组都被视为object数组，因为数组和String被视为object。仍然用一个例子来说明，这次是一个二维 int 数组，作为返回值。

Java 代码：

```
1.  JNIEXPORT jobjectArray JNICALL
    Java_ObjectArrayTest_initInt2DArray(JNIEnv *env, jclass cls, int size){
2.
3.  jobjectArray result;
4.
5.  jclass intArrCls = env->FindClass("[I");
6.  result = env->NewObjectArray(size, intArrCls, NULL);
7.
```

```
8.  for (int i = 0; i < size; i++) {
9.  jint tmp[256];
10. jintArray iarr = env->NewIntArray(size);
11.
12.
13. for(int j = 0; j < size; j++) {
14. tmp[j] = i + j;
15. }
16.
17. env->SetIntArrayRegion(iarr, 0, size, tmp);
18. env->SetObjectArrayElement(result, i, iarr);
19. env->DeleteLocalRef(iarr);
20.
21. }
22. return result;
23.
24. }
```

复制代码

上面代码中的第三行，

jobjectArray result;

因为要返回值，所以需要新建一个 jobjectArray 对象。

jclass intArrCls = env->FindClass("[I");

是创建一个 jclass 的引用，因为 result 的元素是一维 int 数组的引用，所以 intArrCls 必须是一维 int 数组的引用，这一点是如何保证的呢?注意 FindClass 的参数"[I"，JNI 就是通过它来确定引用的类型的，I 表示是 int 类型，[标识是数组。对于其他的类型，都有相应的表示方法，

Java 代码：

```
1.  Z boolean
2.  B byte
3.  C char
4.  S short
5.  I int
6.  J long
7.  F float
8.  D double
```

复制代码

String 是通过“Ljava/lang/String;”表示的，那相应的，String 数组就应该是

“[Ljava/lang/String;”。

还是回到代码，

```
result = env->NewObjectArray(size, intArrCls, NULL);
```

的作用是为 result 分配空间。

```
jintArray iarr = env->NewIntArray(size);
```

是为一维 int 数组 iarr 分配空间。

```
env->SetIntArrayRegion(iarr, 0, size, tmp);
```

是为 iarr 赋值。

```
env->SetObjectArrayElement(result, i, iarr);
```

是为 result 的第 i 个元素赋值。

通过上面这些步骤，我们就创建了一个二维 int 数组，并赋值完毕，这样就可以做为参数返回了。

如果了解了上面介绍的这些内容，基本上大部分的任务都可以对付了。虽然在操作数组类型，尤其是二维数组和 String 数组的时候，比起在单独的语言中编程要麻烦，但既然我们享受了跨语言编程的好处，必然要付出一定的代价。

有一点要补充的是，本文所用到的函数调用方式都是针对 C++的，如果要在 C 中使用，所有的 env->都要被替换成(*env)->，而且后面的函数中需要增加一个参数 env，具体请看一下 jni.h 的代码。另外还有些省略的内容，可以参考 JNI 的文档：Java Native Interface 6.0 Specification，在 JDK 的文档里就可以找到。如果要进行更深入的 JNI 编程，需要仔细阅读这个文档。接下来的高级篇，也会讨论更深入的话题。

转载来自:http://www.eoeandroid.com/thread-90138-1-1.html 整合在一起了，谢谢阅读！