###### Android JNI作用及其详解

2016年07月12日 10:58:38

阅读数：17132

**[Android](http://lib.csdn.net/base/15" \o "undefined" \t "https://blog.csdn.net/suofeng1234/article/details/_blank)**JNI作用及其详解****

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：  
　　public class MediaPlayer{  
　　static {  
　　System.loadLibrary("media\_jni");  
　　}  
　　}  
　　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：  
　　//#define LOG\_NDEBUG 0  
　　#define LOG\_TAG "MediaPlayer-JNI"  
jint JNI\_OnLoad(JavaVM\* vm, void\* reserved)  
　　{  
　　JNIEnv\* env = NULL;  
　　jint result = -1;  
　　if (vm->GetEnv((void\*\*) &env, JNI\_VERSION\_1\_4) != JNI\_OK) {  
　　LOGE("ERROR: GetEnv failed ");  
　　goto bail;  
　　}  
　　assert(env != NULL);  
　　if (register\_android\_media\_MediaPlayer(env) < 0) {  
　　LOGE("ERROR: MediaPlayer native registration failed ");  
　　goto bail;  
　　}  
　　if (register\_android\_media\_MediaRecorder(env) < 0) {  
　　LOGE("ERROR: MediaRecorder native registration failed ");  
　　goto bail;  
　　}  
　　if (register\_android\_media\_MediaScanner(env) < 0) {  
　　LOGE("ERROR: MediaScanner native registration failed ");  
　　goto bail;  
　　}  
　　if (register\_android\_media\_MediaMetadataRetriever(env) < 0) {  
　　LOGE("ERROR: MediaMetadataRetriever native registration failed ");  
　　goto bail;  
　　}  
　　result = JNI\_VERSION\_1\_4;  
　　bail:  
　　return result;  
　　}  
　　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：  
　　if (register\_android\_media\_MediaPlayer(env) < 0) {  
　　LOGE("ERROR: MediaPlayer native registration failed ");  
　　goto bail;  
　　}  
　　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：  
　　jint JNI\_OnLoad(JavaVM\* vm, void\* reserved) { }  
　　jint JNI\_OnUnload(JavaVM\* vm, void\* reserved){ }  
　　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：  
　jint JNI\_OnLoad(JavaVM\* vm, void\* reserved){  
　　JNIEnv\* env = NULL;  
　　jint result = -1;  
　　if (vm->GetEnv((void\*\*) &env, JNI\_VERSION\_1\_4) != JNI\_OK) {  
　　LOGE("ERROR: GetEnv failed ");  
　　goto bail;  
　　}  
　　}  
　　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：  
　　jint JNI\_OnLoad(JavaVM\* vm, void\* reserved)  
　　{  
　　JNIEnv\* env = NULL;  
　　if (register\_android\_media\_MediaPlayer(env) < 0) {  
　　}  
　　}  
　　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：  
　　if ((\*env)->MonitorEnter(env, obj) != JNI\_OK) {  
　　}  
　　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：  
　　if ((\*env)->MonitorExit(env, obj) != JNI\_OK) {  
　　}  
　　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。一般是仰赖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：  
　　//#define LOG\_NDEBUG 0  
　　#define LOG\_TAG "MediaPlayer-JNI"  
　　static JNINativeMethod gMethods[] = {  
　　{"setDataSource", "(Ljava/lang/String;)V",  
　　(void \*)android\_media\_MediaPlayer\_setDataSource},  
{"setDataSource", "(Ljava/io/FileDescriptor;JJ)V",  
　　(void \*)android\_media\_MediaPlayer\_setDataSourceFD},  
　　{"prepare", "()V", (void \*)android\_media\_MediaPlayer\_prepare},  
　　{"prepareAsync", "()V", (void \*)android\_media\_MediaPlayer\_prepareAsync},  
　　{"\_start", "()V", (void \*)android\_media\_MediaPlayer\_start},  
　　{"\_stop", "()V", (void \*)android\_media\_MediaPlayer\_stop},  
　　{"getVideoWidth", "()I", (void \*)android\_media\_MediaPlayer\_getVideoWidth},  
　　{"getVideoHeight", "()I", (void \*)android\_media\_MediaPlayer\_getVideoHeight},  
　　{"seekTo", "(I)V", (void \*)android\_media\_MediaPlayer\_seekTo},  
　　{"\_pause", "()V", (void \*)android\_media\_MediaPlayer\_pause},  
　　{"isPlaying", "()Z", (void \*)android\_media\_MediaPlayer\_isPlaying},  
　　{"getCurrentPosition", "()I", (void \*)android\_media\_MediaPlayer\_getCurrentPosition},  
　　{"getDuration", "()I", (void \*)android\_media\_MediaPlayer\_getDuration},  
　　{"\_release", "()V", (void \*)android\_media\_MediaPlayer\_release},  
　　{"\_reset", "()V", (void \*)android\_media\_MediaPlayer\_reset},  
　　{"setAudioStreamType","(I)V", (void \*)android\_media\_MediaPlayer\_setAudioStreamType},  
　　{"setLooping", "(Z)V", (void \*)android\_media\_MediaPlayer\_setLooping},  
　　{"setVolume", "(FF)V", (void \*)android\_media\_MediaPlayer\_setVolume},  
　　{"getFrameAt", "(I)Landroid/graphics/Bitmap;",  
　　(void \*)android\_media\_MediaPlayer\_getFrameAt},  
　　{"native\_setup", "(Ljava/lang/Object;)V",  
　　(void \*)android\_media\_MediaPlayer\_native\_setup},  
　　{"native\_finalize", "()V", (void \*)android\_media\_MediaPlayer\_native\_finalize},  
　　};  
　　static int register\_android\_media\_MediaPlayer(JNIEnv \*env){  
　　return AndroidRuntime::registerNativeMethods(env,  
　　"android/media/MediaPlayer", gMethods, NELEM(gMethods));  
　　}  
　　jint JNI\_OnLoad(JavaVM\* vm, void\* reserved){  
　if (register\_android\_media\_MediaPlayer(env) < 0) {  
　　LOGE("ERROR: MediaPlayer native registration failed ");  
　　goto bail;  
　　}  
　　}  
　　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：　　typedef struct {  
　　const char\* name;  
　　const char\* signature;  
　　void\* fnPtr;  
　　} JNINativeMethod;  
　　其中比较难以理解的是第二个参数，例如  
　　"()V"  
　　"(II)V"  
　　"(Ljava/lang/String;Ljava/lang/String;)V"  
　　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  
　　V void void  
　　Z jboolean boolean  
　　I jint int  
　　J jlong long  
　　D jdouble double  
　　F jfloat float  
　　B jbyte byte  
　　C jchar char  
　　S jshort short  
　　数组则以"["开始，用两个字符表示  
　　[I jintArray int[]  
　　[F jfloatArray float[]  
　　[B jbyteArray byte[]  
　　[C jcharArray char[]  
　　[S jshortArray short[]  
　　[D jdoubleArray double[]  
　　[J jlongArray long[]  
[Z jbooleanArray boolean[]  
　　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"  
　　Android JNI编程实践  
　　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。  
　　jnitest.java文件：  
　　package com.hello.jnitest;  
　　import android.app.Activity;  
　　import android.os.Bundle;  
　　public class jnitest extends Activity {  
　　@Override  
　　public void onCreate(Bundle savedInstanceState) {  
　　super.onCreate(savedInstanceState);  
　　setContentView(R.layout.main);  
　　Nadd cal = new Nadd();  
　　setTitle("The Native Add Result is " + String.valueOf(cal.nadd(10, 19)));  
　　}  
　　}  
　　Nadd.java文件：  
　　package com.hello.jnitest;  
　　public class Nadd {  
　　static {  
　　System.loadLibrary ("Nadd");  
　　}  
　　public native int nadd(int a, int b);  
　　}  
　　以上在windows中完成。  
　　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。  
　　javah -classpath ../jnitest/bin com.hello.jnitest.Nadd  
　　com\_hello\_jnitest\_Nadd .h文件：  
　　#include   
　　#ifndef \_Included\_com\_hello\_jnitest\_Nadd  
　　#define \_Included\_com\_hello\_jnitest\_Nadd  
　　#ifdef \_\_cplusplus  
　　extern "C" {  
　　#endif  
　　JNIEXPORT jint JNICALL Java\_com\_hello\_jnitest\_Nadd\_nadd  
(JNIEnv \*, jobject, jint, jint);  
　　#ifdef \_\_cplusplus  
　　}  
　　#endif  
　　#endif  
　　3.编辑.c文件实现native方法。  
　　com\_hello\_jnitest\_Nadd.c文件：  
　　#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);  
　　}  
　　4.编译.c文件生存动态库。  
　　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/ldscripts/armelf\_linux\_eabi.xsc -share -o libNadd.so com\_hello\_jnitest\_Nadd.o  
　　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libsarmeabi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：  
　　# Copyright (C) 2009 The Android Open Source Project  
　　#  
　　# Licensed under the Apache License, Version 2.0 (the "License");  
　　# you may not use this file except in compliance with the License.  
　　# You may obtain a copy of the License at  
　　#  
　　# [http://www.apache.org/licenses/LICENSE-2.0](http://www.apache.org/licenses/LICENSE-2.0" \t "https://blog.csdn.net/suofeng1234/article/details/_blank)  
　　#  
　　# Unless required by applicable law or agreed to in writing, software  
　　# distributed under the License is distributed on an "AS IS" BASIS,  
　　# 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 $(BUILD\_SHARED\_LIBRARY)  
　　myjni.c文件：  
　　#include   
　　#include   
　　jstring  
　　Java\_com\_hello\_NdkTest\_NdkTest\_stringFromJNI( JNIEnv\* env,  
　　jobject thiz )  
　　{  
　　return (\*env)->NewStringUTF(env, "Hello from My-JNI !");  
　　}  
　　myjni文件组织：  
　　a@ubuntu:~/work/android/ndk-1.6\_r1/apps$ tree myjni  
　　myjni  
　　|-- Application.mk  
　　`-- project  
　　|-- jni  
　　| |-- Android.mk  
　　| `-- myjni.c  
　　`-- libs  
　　`-- armeabi  
　　`-- libmyjni.so  
　　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：  
　　package com.hello.NdkTest;  
　　import android.app.Activity;  
　　import android.os.Bundle;  
　import android.widget.TextView;  
　　public class NdkTest extends Activity {  
　　@Override  
　　public void onCreate(Bundle savedInstanceState) {  
　　super.onCreate(savedInstanceState);  
　　TextView tv = new TextView(this);  
　　tv.setText( stringFromJNI() );  
　　setContentView(tv);  
　　}  
　　public native String stringFromJNI();  
　　static {  
　　System.loadLibrary("myjni");  
　　}  
　　}  
　　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，  
　　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");  
　　}  
　　}  
　　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  
　　#include   
　#ifndef \_Included\_Prompt  
　　#define \_Included\_Prompt  
　　#ifdef \_\_cplusplus  
　　extern "C" {  
　　#endif  
　　JNIEXPORT jstring JNICALL Java\_Prompt\_getLine(JNIEnv \*env, jobject this, jstring prompt);  
　　#ifdef \_\_cplusplus  
　　}  
　　#endif  
　　#endif  
　　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，  
　　#include "Prompt.h"  
　　#include   
　　JNIEXPORT jstring JNICALL Java\_Prompt\_getLine(JNIEnv \*env, jobject obj, jstring prompt)  
　　{  
　　const char\* str;  
　　str = env->GetStringUTFChars(prompt, false);  
　　if(str == NULL) {  
　　return NULL;  
　　}  
　　std::cout << str << std::endl;  
　　env->ReleaseStringUTFChars(prompt, str);  
　　char\* tmpstr = "return string succeeded";  
　　jstring rtstr = env->NewStringUTF(tmpstr);  
　　return rtstr;  
　　}  
　　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程序就不写了，  
　　JNIEXPORT jint JNICALL Java\_IntArray\_sumArray(JNIEnv \*env, jobject obj, jintArray arr)  
　　{  
　　jint \*carr;  
　　carr = env->GetIntArrayElements(arr, false);  
if(carr == NULL) {  
　　return 0;  
　　}  
　　jint sum = 0;  
　　for(int i=0; i<10; i++) {  
　　sum += carr[i];  
　　}  
　　env->ReleaseIntArrayElements(arr, carr, 0);  
　　return sum;  
　　}  
　　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[i]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。  
　　JNIEXPORT jobjectArray JNICALL Java\_ObjectArrayTest\_initInt2DArray(JNIEnv \*env, jclass cls, int size)  
　　{  
　　jobjectArray result;  
　　jclass intArrCls = env->FindClass("[I");  
　　result = env->NewObjectArray(size, intArrCls, NULL);  
　　for (int i = 0; i < size; i++) {  
　　jint tmp[256];  
　　jintArray iarr = env->NewIntArray(size);  
　　for(int j = 0; j < size; j++) {  
　　tmp[j] = i + j;  
　　}  
　　env->SetIntArrayRegion(iarr, 0, size, tmp);  
　　env->SetObjectArrayElement(result, i, iarr);  
　　env->DeleteLocalRef(iarr);  
　　}  
　　return result;  
　　}  
　　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，  
　　Z boolean  
　　B byte  
　　C char  
　　S short  
　　I int  
　　J long  
　　F float  
　　D double  
　　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

在本篇中，将会涉及关于JNI编程更深入的话题，包括：在native方法中访问Java类的域和方法，将Java中自定义的类作为参数和返回值传递等等。了解这些内容，将会对JNI编程有更深入的理解，写出的程序也更清晰，易用性更好。　　1. 在一般的Java类中定义native方法  
　　在前两篇的例子中，都是将native方法放在main方法的Java类中，实际上，完全可以在任何类中定义native方法。这样，对于外部来说，这个类和其他的Java类没有任何区别。  
　　2. 访问Java类的域和方法  
　　native方法虽然是native的，但毕竟是方法，那么就应该同其他方法一样，能够访问类的私有域和方法。实际上，JNI的确可以做到这一点，我们通过几个例子来说明，  
　　public class ClassA {  
　String str\_ = "abcde";  
　　int number\_;  
　　public native void nativeMethod();  
　　private void javaMethod() {  
　　System.out.println("call java method succeeded");  
　　}  
　　static {  
　　System.loadLibrary("ClassA");  
　　}  
　　}  
　　在这个例子中，我们在一个没有main方法的Java类中定义了native方法。我们将演示如何在nativeMethod()中访问域str\_，number\_和方法javaMethod()，nativeMethod()的C++实现如下，  
　　JNIEXPORT void JNICALL Java\_testclass\_ClassCallDLL\_nativeMethod(JNIEnv \*env, jobject obj) {  
　　// access field  
　　jclass cls = env->GetObjectClass(obj);  
　　jfieldID fid = env->GetFieldID(cls, "str\_", "Ljava/lang/String;");  
　　jstring jstr = (jstring)env->GetObjectField(obj, fid);  
　　const char \*str = env->GetStringUTFChars(jstr, false);  
　　if(std::string(str) == "abcde")  
　　std::cout << "access field succeeded" << std::endl;  
　　jint i = 2468;  
　　fid = env->GetFieldID(cls, "number\_", "I");  
　　env->SetIntField(obj, fid, i);  
　　// access method  
　　jmethodID mid = env->GetMethodID(cls, "javaMethod", "()V");  
　　env->CallVoidMethod(obj, mid);  
　　}  
　　上面的代码中，通过如下两行代码获得str\_的值，  
　　jfieldID fid = env->GetFieldID(cls, "str\_", "Ljava/lang/String;");  
　　jstring jstr = (jstring)env->GetObjectField(obj, fid);  
　　第一行代码获得str\_的id，在GetFieldID函数的调用中需要指定str\_的类型，第二行代码通过str\_的id获得它的值，当然我们读到的是一个jstring类型，不能直接显示，需要转化为char\*类型。  
　　接下来我们看如何给Java类的域赋值，看下面两行代码，  
　　fid = env->GetFieldID(cls, "number\_", "I");  
　　env->SetIntField(obj, fid, i);  
　　第一行代码同前面一样，获得number\_的id，第二行我们通过SetIntField函数将i的值赋给number\_，其他类似的函数可以参考JDK的文档。  
　访问javaMethod()的过程同访问域类似，  
　　jmethodID mid = env->GetMethodID(cls, "javaMethod", "()V");  
　　env->CallVoidMethod(obj, mid);  
　　需要强调的是，在GetMethodID中，我们需要指定javaMethod方法的类型，域的类型很容易理解，方法的类型如何定义呢，在上面的例子中，我们用的是()V，V表示返回值为空，()表示参数为空。如果是更复杂的函数类型如何表示?看一个例子，  
　　long f (int n, String s, int[] arr);  
　　这个函数的类型符号是(ILjava/lang/String;[I)J，I表示int类型，Ljava/lang/String;表示String类型，[I表示int数组，J表示long。这些都可以在文档中查到。  
　　3. 在native方法中使用用户定义的类  
　　JNI不仅能使用Java的基础类型，还能使用用户定义的类，这样灵活性就大多了。大体上使用自定义的类和使用Java的基础类(比如 String)没有太大的区别，关键的一点是，如果要使用自定义类，首先要能访问类的构造函数，看下面这一段代码，我们在native方法中使用了自定义的Java类ClassB，  
　　jclass cls = env->FindClass("Ltestclass/ClassB;");  
　　jmethodID id = env->GetMethodID(cls, "", "(D)V");  
　　jdouble dd = 0.033;  
　　jvalue args[1];  
　　args[0].d = dd;  
　　jobject obj = env->NewObjectA(cls, id, args);  
　　首先要创建一个自定义类的引用，通过FindClass函数来完成，参数同前面介绍的创建String对象的引用类似，只不过类名称变成自定义类的名称。然后通过GetMethodID函数获得这个类的构造函数，注意这里方法的名称是""，它表示这是一个构造函数。  
　　jobject obj = env->NewObjectA(cls, id, args);  
　　生成了一个ClassB的对象，args是ClassB的构造函数的参数，它是一个jvalue\*类型。  
　　通过以上介绍的三部分内容，native方法已经看起来完全像Java自己的方法了，至少主要功能上齐备了，只是实现上稍麻烦。而了解了这些，JNI编程的水平也更上一层楼。下面要讨论的话题也是一个重要内容，至少如果没有它，我们的程序只能停留在演示阶段，不具有实用价值。  
　　4. 异常处理  
　　在C++和Java的编程中，异常处理都是一个重要的内容。但是在JNI中，麻烦就来了，native方法是通过C++实现的，如果在native方法中发生了异常，如何传导到Java呢?  
　JNI提供了实现这种功能的机制。我们可以通过下面这段代码抛出一个Java可以接收的异常，  
　　jclass errCls;  
　　env->ExceptionDescribe();  
　　env->ExceptionClear();  
　　errCls = env->FindClass("java/lang/IllegalArgumentException");  
　　env->ThrowNew(errCls, "thrown from C++ code");  
　　如果要抛出其他类型的异常，替换掉FindClass的参数即可。这样，在Java中就可以接收到native方法中抛出的异常。  
　　至此，JNI编程系列的内容就完全结束了，这些内容都是本人的原创，通过查阅文档和网上的各种文章总结出来的，相信除了JDK的文档外，没有比这更全面的讲述JNI编程的文章了。当然，限于篇幅，有些地方不可能讲的很细。限于水平，也可能有一些错误。文中所用的代码，都亲自编译执行过