Android的应用程序(Dalvik VM)中使用JNI（Java Native Interface）调用C/C++开发的共享库。

开发环境:（eclipse + Android ADT）, WindowsXP SP3, JDK 1.6.0.12 , VMWARE+Ubuntu Linux 8.04

**Android Jni使用方法步骤(1) Eclipse中新建Android工程**

1. 工程名 JNItest
2. Package名com.ura.test
3. Activity名 JNItest
4. 应用程序名 JNItest

**Android Jni使用方法步骤(3)编辑java文件**

1. *package com.ura.test;*
2. *import Android.app.Activity;*
3. *import Android.os.Bundle;*
4. *import Android.widget.TextView;*
5. *public class JNITest extends Activity {*
6. */\*\* Called when the activity is first created. \*/*
7. *static { System.loadLibrary("JNITest"); }*
8. *public native String GetTest();*
9. *@Override*
10. *public void onCreate(Bundle savedInstanceState) {*
11. *super.onCreate(savedInstanceState);*
12. *setContentView(R.layout.main);*
13. *String str =GetTest();*
14. *TextView JNITest = (TextView)findViewById(R.id.JNITest);*
15. *JNITest.setText(str);*
16. *}*
17. *}*

**Android 生成head文件**

编译上面工程声称class文件，然后用javah工具生成c/c++头文件。

1. *javah -classpath ./bin/ -jni com.ura.test.JNItest*

**Android Jni编写c/c++文件如下**

**Android Jni使用方法步骤(6)编写Android.mk文件**

1. *LOCAL\_PATH:= $(call my-dir)*
2. *include $(CLEAR\_VARS)*
3. *LOCAL\_SRC\_FILES:= com\_ura\_test\_JNITest.c*
4. *LOCAL\_C\_INCLUDES := $(JNI\_H\_INCLUDE)*
5. *LOCAL\_SHARED\_LIBRARIES := libutils*
6. *LOCAL\_PRELINK\_MODULE := false*
7. *LOCAL\_MODULE := libJNITest*
8. *include $(BUILD\_SHARED\_LIBRARY)*

**Android Jni使用方法步骤(7)编译生成动态库**

新建文件夹

1. *~/mydroid/external/libJNITest*

把上面编写好的头文件，c/c++源文件，make文件拷贝进上面目录中

\* 需要注意的是把PRELINK\_MOUDULE设置成false否则需要重新做成img文件再烧入。在ubuntu中执行

1. *cd mydroid/*
2. *./build/envsetup.sh*
3. *cd ./external/libJNITest/*
4. *mm*

编译成功的后会在下面目录中生成libJNITest.so文件:*~mydroid/out/target/product/generic/system/lib/*

**Android Jni使用方法步骤(8)在模拟器中执行程序**

首先要把动态库拷进/system/lib中。

启动模拟器

1. *adb shell*
2. *adb remount*
3. *adb push libJNITest.so /system/lib*

然后不要关闭模拟器（关掉再开动态库就没了，因为模拟器rom是只读），执行java程序JNITest，会看到屏幕上打印出

JNITest Native String

看过JDK源代码的人肯定会注意到在源码里有很多标记成native的方法。这些个方法只有方法签名但是没有方法体。其实这些naive方法就是我们说的java native interface。他提供了一个调用（invoke）的接口，然后用C或者C++去实现。我们首先来编写这个“桥梁”。先在eclipse里建立一个HelloFore的Java工程，然后编写下面的代码。

Java代码

1. *package com.chnic.jni;*
2. *public class SayHellotoCPP {*
3. *public SayHellotoCPP(){}*
4. *public native void sayHello(String name);*
5. *}*

我的native本地方法有一个String的参数。会传递一个name到后台去。本地方法已经完成，接下来就要用javah方法来生成一个相对应的.h头文件。javah是一个专门为JNI生成头文件的一个命令。利用javah產生與SayHellotoCPP.class的.h頭文件。用的是 -jni这个参数，这个参数也是默认的参数，他会生成一个JNI式的.h头文件。在控制台进入到工程的根目录(./bin/com/chnic/jni/SayHellotoCPP.class)，也就是HelloFore这个目录，然后输入命令。

Java代码

1. *javah -jni com.chnic.jni.SayHellotoCPP*

命令执行完之后在工程的根目录就会发现**com\_chnic\_jni\_SayHellotoCPP.h** 这个头文件。在执行javah的时候，要输入完整的包名+类名。否则在以后的测试调用过程中会发生java.lang.UnsatisfiedLinkError这个异常。到这里java部分算是基本完成了，接下来我们来编写后端的C/C++代码。

Cpp代码

1. */\**
2. *\* Class:     com\_chnic\_jni\_SayHellotoCPP*
3. *\* Method:    sayHello*
4. *\* Signature: (Ljava/lang/String;)V*
5. *\*/*
6. *JNIEXPORT void JNICALL Java\_com\_chnic\_jni\_SayHellotoCPP\_sayHello(JNIEnv \*, jobject, jstring);*

在注释上标注类名、方法名、签名（Signature）。在这里最重要的是 Java\_com\_chnic\_jni\_SayHellotoCPP\_sayHello这个方法签名。在Java端我们执行 sayHello(String name)这个方法之后，JVM就会帮我们唤醒在DLL里的Java\_com\_chnic\_jni\_SayHellotoCPP\_sayHello这个方法。因此我们新建一个C++ source file来实现这个方法。

Cpp代码

1. *#include <iostream.h>*
2. *#include "com\_chnic\_jni\_SayHellotoCPP.h"*
3. *JNIEXPORT void JNICALL Java\_com\_chnic\_jni\_SayHellotoCPP\_sayHello(JNIEnv\* env, jobject obj, jstring name)*
4. *{*
5. *const char\* pname = (\*env)->GetStringUTFChars(env, name, NULL);*
6. *cout << "Hello, " << pname << endl;*
7. *}*

接下来让java代码“认识”这个动态链接库。加入*System.loadLibrary("HelloEnd")*;这句到静态初始化块里。

Java代码

1. *package com.chnic.jni;*
2. *public class SayHellotoCPP {*
3. *static{ System.loadLibrary("HelloEnd");  }*
4. *public SayHellotoCPP(){ }*
5. *public native void sayHello(String name);*
6. *}*

这样我们的代码就能认识并加载这个动态链接库文件了。接下来编写测试代码。

Java代码

1. *SayHellotoCPP shp = new SayHellotoCPP();*
2. *shp.sayHello("World");*

 Cpp代码

1. *const char\* pname = (\*env)->GetStringUTFChars(env, name, NULL);*

GetStringUTFChars就是JNI给developer提供的API。

1. 因为JNI有一个Native这个特点，一点有项目用了JNI，也就说明这个项目基本不能跨平台了。
2. JNI调用是相当慢的，在实际使用的之前一定要先想明白是否有这个必要。
3. 因为C++和C这样的语言非常灵活，一不小心就容易出错，比如我刚刚的代码就没有写析构字符串释放内存，对于java developer来说因为有了GC 垃圾回收机制，所以大多数人没有写析构函数这样的概念。所以JNI也会增加程序中的风险，增大程序的不稳定性。

其实在Java代码中，除了对本地方法标注native关键字和加上要加载动态链接库之外，JNI基本上是对上层coder透明的，上层coder调用那些本地方法的时候并不知道这个方法的方法体究竟是在哪里，这个道理就像我们用JDK所提供的API一样。所以在Java中使用JNI还是很简单的，相比之下在C++中调用java，就比前者要复杂的多了。

现在来介绍下JNI里的数据类型。在C++里，编译器会很据所处的平台来为一些基本的数据类型来分配长度，因此也就造成了平台不一致性，而这个问题在Java中则不存在，因为有JVM的缘故，所以Java中的基本数据类型在所有平台下得到的都是相同的长度，比如int的宽度永远都是32位。基于这方面的原因，java和c++的基本数据类型就需要实现一些mapping：

|  |  |  |
| --- | --- | --- |
| Java类型 | 本地类型 | JNI中定义的别名 |
| int | long | jint |
| long | \_int64 | jlong |
| byte | signed char | jbyte |
| boolean | unsigned char | jboolean |
| char | unsigned short | jchar |
| short | short | jshort |
| float | float | jfloat |
| double | double | jdouble |
| Object | \_jobject\* | jobject |

如果想了解一些更加细致的信息，可以去看一些jni.h这个头文件，各种数据类型的定义以及别名就被定义在这个文件中。了解了JNI中的数据类型，下面就来看这次的例子。

Java代码

1. *package com.chnic.service;*
2. *import com.chnic.bean.Order;*
3. *public class Business {*
4. *static{ System.loadLibrary("FruitFactory"); }*
5. *public Business(){}*
6. *public native double getPrice(String name);*
7. *public native Order getOrder(String name, int amount);*
8. *public native Order getRamdomOrder();*
9. *public native void analyzeOrder(Order order);*
10. *public void notification(){ System.out.println("Notification."); }*
11. *public static void notificationByStatic(){ System.out.println("Notification in a static method.); }*
12. *}*

这个类里面包含4个本地方法，一个静态初始化块加载将要生成的so文件。这个类需要一个名为Order的JavaBean。

Java代码

1. *package com.chnic.bean;*
2. *public class Order {*
3. *private String name = "Fruit";*
4. *private double price;*
5. *private int amount = 30;*
6. *public Order(){}*
7. *public int getAmount() { return amount; }*
8. *public void setAmount(int amount){ this.amount = amount; }*
9. *public String getName() { return name; }*
10. *public void setName(String name) { this.name = name; }*
11. *public double getPrice() { return price; }*
12. *public void setPrice(double price) { this.price = price; }*
13. *}*

在工程里我们新建一个名为Foctory的C++ source file 文件，去实现那些native方法。具体的代码如下。

Cpp代码

1. *#include <iostream.h>*
2. *#include <string.h>*
3. *#include "com\_chnic\_service\_Business.h"*
4. *jobject getInstance(JNIEnv\* env, jclass obj\_class);*
5. *JNIEXPORT jdouble JNICALL Java\_com\_chnic\_service\_Business\_getPrice(JNIEnv\* env, jobject obj, jstring name)*
6. *{*
7. *const char\* pname = (\*env)->GetStringUTFChars(env, name, NULL);*
8. *cout << "Before release: "  << pname << endl;*
9. *if (strcmp(pname, "Apple") == 0)*
10. *{*
11. *env->ReleaseStringUTFChars(name, pname);*
12. *cout << "After release: " << pname << endl;*
13. *return 1.2;*
14. *} else {*
15. *env->ReleaseStringUTFChars(name, pname);*
16. *cout << "After release: " << pname << endl;*
17. *return 2.1;*
18. *}*
19. *}*
20. *JNIEXPORT jobject JNICALL Java\_com\_chnic\_service\_Business\_getOrder(JNIEnv\* env, jobject obj, jstring name, jint amount)*
21. *{*
22. *jclass order\_class = env->FindClass("com/chnic/bean/Order");*
23. *jobject order = getInstance(env, order\_class);*
24. *jmethodID setName\_method = env->GetMethodID(order\_class, "setName", "(Ljava/lang/String;)V");*
25. *env->CallVoidMethod(order, setName\_method, name);*
26. *jmethodID setAmount\_method = env->GetMethodID(order\_class, "setAmount", "(I)V");*
27. *env->CallVoidMethod(order, setAmount\_method, amount);*
28. *return order;*
29. *}*

Java中的四个本地方法在这里全部被实现，接下来针对这四个方法来解释下：

**1.getPrice(String name)**

这个方法是从foreground传递一个类型为string的参数到backend，然后backend判断返回相应的价格。在cpp的代码中，我们用GetStringUTFChars这个方法来把传来的jstring变成一个UTF-8编码的char型字符串。因为jstring的实际类型是 jobject，所以无法直接比较。GetStringUTFChars方法包含两个参数，第一参数是你要处理的jstring对象，第二个参数是否需要在内存中生成一个副本对象。将 jstring转换成为了一个const char\*了之后。在这里还要多说一下ReleaseStringUTFChars这个函数，就是释放内存用的。有点像cpp里的析构函数，只不过Sun帮我们已经封装好了。由于在JVM中有GC这个东东，所以多数java coder并没有写析构的习惯，不过在JNI里是必须的了，否则容易造成内存泄露。我们在这里在release之前和之后分别打出这个字符串来看一下效果，我们编写测试代码。

Java代码

1. *Business b = new Business();*
2. *System.out.println(b.getPrice("Apple"));*

运行这段测试代码，控制台上打出

Before release: Apple  
After release: ��  
1.2

在release之前打印出来的是Apple，release之后就成了乱码了。由于传递的是Apple，所以得到1.2。

**2. getOrder(String name, int amount)**

在foreground中可以通过这个方法让backend返回一个你“指定”的Order。所谓“指定”，其实也就是指方法里的两个参数：name和amout，在cpp的代码在中，会根据传递的两个参数来构造一个Order。回到cpp的代码里。

Java代码

1. *jclass obj\_class = env->FindClass("com/chnic/bean/Order");*

是不是觉得这句代码似曾相识？没错，这句代码很像我们java里写的Class.forName(className)反射的代码。其实在这里 FindClass的作用和上面的forName是类似的。只不过在forName中要用完整的类名，但是在这里必须用"/"来代替“.”。这个方法会返回一个jclass的对象，其实也就是我们在Java中说的类对象。

Java代码

1. *jmethodID construction\_id = env->GetMethodID(obj\_class, "<init>", "()V");*
2. *jobject obj = env->NewObject(obj\_class, construction\_id);*

拿到"类对象"了之后，按照Java RTTI的逻辑我们接下来就要唤醒那个类对象的构造函数了。在JNI中，包括构造函数在内的所有方法都被看成Method。每个method都有一个特定的ID，我们通过GetMethodID这个方法就可以拿到我们想要的某一个java 方法的ID。GetMethodID需要传三个参数，第一个是很显然jclass，第二个参数是java方法名，也就是你想取的method ID的那个方法的方法名，第三个参数是方法签名。在这里有必要单独来讲一讲这个方法签名，在Java里方法是可以被重载的，比如我一个类里有public void a(int arg)和public void a(String arg)这两个方法，在这里用方法名来区分方法显然就是行不通的了。方法签名包括两部分：参数类型和返回值类型；具体的格式：(参数1类型签名 参数2类型签名)返回值类型签名。下面是java类型和年名类型的对照的一个表

|  |  |
| --- | --- |
| Java类型 | 对应的签名 |
| boolean | Z |
| byte | B |
| char | C |
| short | S |
| int | I |
| long | L |
| float | F |
| double | D |
| void | V |
| Object | L用/分割包的完整类名;  Ljava/lang/String; |
| Array | [签名       [I       [Ljava/lang/String; |

回到我们刚刚的getMethodID这个方法上。因为是调用构造函数，JNI规定调用构造函数的时候传递的方法名应该为<init> ,通过javap查看我们要的那个无参的构造函数的方法签是()V。得到方法签名，最后我们调用NewObject方法来生成一个新的对象。拿到了对象，之后我们开始为对象jobject填充数值，还是首先拿到setXXX方法的Method ID，之后调用Call<Type>Method来调用java方法。这里的<Type>所指的是方法的返回类型，我们刚刚调用的是set方法的返回值是void，因此这里的方法也就是CallVoidMethod，这个方法的参数除了前两个要传入jobject和 jmethodID之外还要传入要调用的那个方法的参数，而且要顺序必须一致。终于介绍完了第二个方法，下来就是测试代码测试。

Java代码

1. *Order o = b.getOrder("Watermelom", 100);*
2. *System.out.println("java: " + o.getName());*
3. *System.out.println("java: " + o.getAmount());*

控制台打出

java: Watermelom  
java: 100

就此，我们完成了第二个方法的测试。

### [JNI調用機制](http://blog.csdn.net/dnfchan/article/details/8568313)

<http://blog.csdn.net/dnfchan/article/details/8568313>

Java Native Interface（JNI）是java本地接口，所謂的本地（native）一般是指c/c++語言。當使用java進行程序設計的時候，一般主要有三種情況需要c的協助。

1）調用驅動。由於操作系統提供的驅動接口一般都是c接口，java語言本身不具備操作這些驅動的能力

2）對於某些大量數據處理的模塊，java效率可能遠低於c，因此，程序員希望使用c完成

3）對於某些功能模塊，可能java和c的效率差不多，但是這些模塊已經有現成的c代碼，程序員不想用java重寫，而是複用已有的c代碼。

這就是java提出jni概念的原因。無論出於什麼原因，從程序的角度來看，jni接口主要包含兩種情況。第一種是java中調用c，第二種是c中訪問java，只要解決了這兩個問題，那麼就可以任意進行java和c的應用組合。framework中大量使用JNI完成本地接口的實現，因此，理解JNI對於閱讀Framework代碼有很大幫助。

**1.java中訪問c**

java中可以定義某個函數為native類型，對於native函數，只要聲明即可，因為該函數的實現是native的，既有相應的c去實現。java編譯器遇到native函數時，不會關心函數的具體實現，因此編譯不會出錯誤。程序運行時，在調用native方法之前，程序員必須把c生成的動態庫裝載進來，否則程序會因為找不到相應的native方法而出錯。

調用native函數時，java會自動產生一個對應的c中的函數名稱，因為java中聲明的函數名稱和c中實現的函數名稱是不同的。其關係為，後者等於包名加前者的名稱，並且中間以下劃線分割，比如Framework，AssetManager類聲明以下方法。

private native final void init() ;

該方法對應的c函數是: *static void android\_content\_AssetManager\_init(JNIEnv\* env,jobject clazz)*

這種映射關係並不是java編譯器內涵的，程序員完全可以改變，但這是一種編程規範。事實上當java調用native時，編譯器會向native引擎傳遞調用者的包名，以及函數名稱，含有參數類型，僅此而已，native引擎根據這些信息決定應該具體調用哪個本地函數。native引擎中AndroidRuntime類提供了一個registerNativeMethods()方法，可以通過該函數來定義java native函數和c函數名稱的映射關係。

在產生c函數中，會包含至少兩個參數。前者是JNIEnv對象，該對像是Java VM所運行的環境，相當於JVM管家，通過它可以訪問JVM內部的各種對象；第二個參數是jobject，是調用該函數的對象，比如上面的指的是AssetManager對象。

如果native聲明的函數本身有參數，那麼這些參數會依次放在以上兩個參數後面，比如以下方法

*private native final Sting[] getArrayStringResource(int arrayRes);*

其對應的c函數為：

*static jobjectArray android\_content\_AssetManager\_getArrayStringResource(JNIEnv\* env,jobject clazz,jint arrayResId);*

在以上對應關係中發現，native所使用的類型跟java不一樣，這些具體的定義位於jni.h中該文件所在路徑為：

dalvik/libnativehelper/include/nativehelper/jni.h

external/webkit/webkit/android/javaVM/jni.h

ndk/build/platforms/android-x/arch-xxx/usr/include/jni.h

不同的平台會有不同的定義，因為java是跨平台的，而各自平台的cpu數據寬度不同，所以又各自的定義。

以上介紹了java和c函數的名稱轉換，如何具體操作呢？

依照轉換關係，手工編寫相應的c代碼，然後編譯成動態庫，並在java代碼執行的時候加載該庫。為了輔助，java提供了javah工具，可以從一個java文件生成相應的c頭文件剩下的就是根據這些頭文件在實現具體內部代碼。比如我們的Foo.java

*package com.haiii.android.client;*

*public class Foo{*

*native void foo1();*

*native int foo2(int a,String b);*

使用javah命令javah -d ~/Desktop -jni com.haiii.android.client.Foo

-d 是指定輸出路徑，-jni表示生成jni頭文件，後面的是class文件所在類。

生成默認文件名稱為com\_haiii\_andorid\_client\_Foo.h。

編譯c代碼，產生動態庫以後，在代碼調用native函數前，使用System.loadLibrary('lib\_name')函數加載該庫。

java代碼不能直接訪問c的變量，c的變量對java來講都是私有的。如果想訪問某個變量，就讓c提供get/set類型的方法，以達到簡介訪問的目的。

**2.c訪問java**

c為什麼訪問java，如果c中需要使用java的某個變量而進行相應的處理，或者c中想調用java中函數完成某個操作，那麼c就要訪問java。

由於java中的函數在native引擎中沒有直接的函數指針，java函數只能由java引擎去執行，而不是c。所以c訪問java不能通過函數指針，而只能通過通用的函數接口，正如java調用c一樣。java把類名、參數類型、傳遞給native引擎，然後由native引擎處理c函數，同理c調用java也需要把想要訪問的類名、參數名稱、參數傳遞給java引擎。其步驟如下：

1）獲取Java對象的類

*cls=env->GetObjectClass(jobject);*

其中env為java調用c函數時的第一個參數，這意味著c調用java函數只能在java調用c函數中進行，否則無法獲取env變量。換句話說，對於c來講，就是“你不惹我，我不惹你”。jobject為第二個參數。class的類型是jclass。

2）獲取java函數的id值

*jmethodId mid=env->GetMethodId(cls,"method\_name","([Ljava/lang/String;)V");*

該方法第二個參數為java中的函數名稱，第三個參數代表了java函數的參數和返回值，參數在括弧之中，返回值在括弧之外。參數[Ljava/lang/String代表了String類型的參數，由於String本身是一個類，而不是java的原子類型，所以前面加了包的名稱，並用斜線分割，最前面還要用一個中括弧進行標示，後面還要用分號隔離。返回值V代表void。java和native數據類型對應

java類型 native類型

boolean          Z

byte          B

char  C

double  D

float  F

int  I

long  L

Object  'L'+'packageName'+';'

short  S

java提供了一個javap工具，查看java函數的輸入，返回參數，用法如下

javap -s Foo

-s含義是簽名，因為java允許函數重載，所以不同的參數、返回值代表著不同的函數，或者說是不同函數簽名。

3）找到函數後，就可以調用該函數了

env->CallXXXMethod(jobject,mid,ret);

其中XXX代表了函數的返回值類型，具體包括Void、Object、Boolean、Byte、Char、Short、Int、Long、Float、Double。第二個參數mid即為第二步獲取的函數id。通過以上三步，實現了c中調用java函數。

c中如何訪問java中的變量呢？實現步驟如下

1）與前面相同: class=env->getObjectClass(jobject);

2)獲取變量ID值: jfieldId fid=env->getFiledId(class,"filed\_name","I")

參數filed\_name為java變量名稱，第三個參數是變量的類型，其格式與上面相同。

3）獲取變量值: value=env->getXXXField(env,jobject,fid)

該函數與上面的顯著不同，其中第一、第二參數為原裝java訪問c函數的前兩個參數，該方法以返回值的方式獲取變量值，而不是通過參數引用