沙翁
向昨天要经验； 向今天要结果； 向明天要动力

JNI介绍

JNI是在学习Android HAL时必须要面临一个知识点，如果你不了解它的机制，不了解它的使用方式，你会被本地代码绕的晕头转向，JNI作为一个中间语言的翻译官在运行Java代码的Android中有着重要的意义，这儿的内容比较多，也是最基本的，如果想彻底了解JNI的机制，请查看：

http://docs.oracle.com/javase/1.5.0/docs/guide/jni/spec/design.html

本文结合了网友ljeagle写的JNI学习笔记和自己通过JNI的手册及Android中常用的部分写得本文。

JNI学习笔记：

http://blog.csdn.net/ljeagle/article/details/6660901

让我们开始吧！！

JNI概念

JNI是本地语言编程接口。它允许运行在JVM中的Java代码和用C、C++或汇编写的本地代码相互操作。

在以下几种情况下需要使用到JNI：

l 应用程序依赖于特定平台，平**立的Java类库代码不能满足需要

l 你已经有一个其它语言写的一个库，并且这个库需要通过JNI来访问Java代码

l 需要执行速度要求的代码实现功能，比如低级的汇编代码

通过JNI编程，你可以使用本地方法来：

l 创建、访问、更新Java对象

l 调用Java方法

l 捕获及抛出异常

l 加载并获得类信息

l 执行运行时类型检查

JNI的原理

JVM将JNI接口指针传递给本地方法，本地方法只能在当前线程中访问该接口指针，不能将接口指针传递给其它线程使用。在VM中 JNI接口指针指向的区域用来分配和存储线程本地数据。

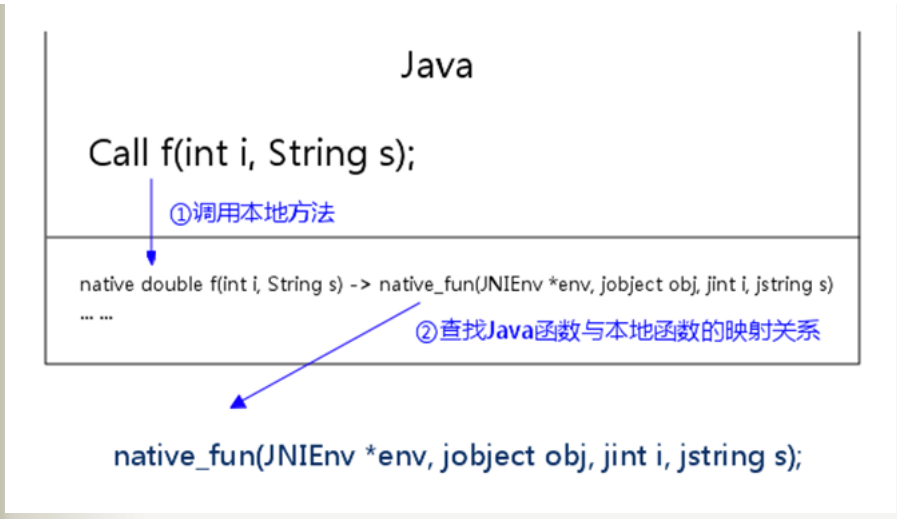
当Java代码调用本地方法时，VM将JNI接口指针作为参数传递给本地方法，当同一个Java线程调用本地方法时VM保证传递给本地方法的参数是相同的。不过，不同的Java线程调用本地方法时，本地方法接收到的JNI接口指针是不同的。

加载和链接本地方法

在Java里通过System.loadLibrary()来加载动态库，但是，动态库只能被加载一次，因此，通常动态库的加载放在静态初始化语句块中。

```
package pkg;
class Cls {
     native double f(int i, String s);          // 声明为本地方法
     static {
         System.loadLibrary(“pkg_Cls”);        // 通过静态初始化语句块来加载动态库
     }
}
```

通常在动态库中声明大量的函数，这些函数被Java调用，这些本地函数由VM维护在一张函数指针数组中，在本地方法里通过调用JNI方法RegisterNatives()来注册本地方法和Java方法的映射关系。



本地方法可以由C或C++来实现，C语言版本：

```
jdouble native_fun (
     JNIEnv *env,               /* interface pointer */
     jobject obj,               /* "this" pointer */
     jint i,                    /* argument #1 */
     jstring s)                 /* argument #2 */
{
     /* Obtain a C-copy of the Java string */
     const char *str = (*env)->GetStringUTFChars(env, s, 0);

     /* process the string */
     ...

     /* Now we are done with str */
     (*env)->ReleaseStringUTFChars(env, s, str);
     return ...
}
```

C++语言版本：

```
extern "C"                              /* specify the C calling convention */
jdouble native_fun (
     JNIEnv *env,               /* interface pointer */
     jobject obj,               /* "this" pointer */
     jint i,                    /* argument #1 */
     jstring s)                 /* argument #2 */
{
     const char *str = env->GetStringUTFChars(s, 0);
     ...
     env->ReleaseStringUTFChars(s, str);
     return ...
}
```

由上面两段代码对比可知，本地代码使用C++来实现更简洁。

两段本地代码第一个参数都是JNIEnv*env，它代表了VM里的环境，本地代码可以通过这个env指针对Java代码进行操作，例如：创建Java类对象，调用Java对象方法，获取Java对象属性等。jobject obj相当于Java中的Object类型，它代表调用这个本地方法的对象，例如：如果有new NativeTest.CallNative()，CallNative()是本地方法，本地方法第二个参数是jobject表示的是NativeTest类的对象的本地引用。

如果本地方法声明为static类型

```
static jint native_get_count(JNIEnv* env, jobject thiz);
```

数据传递

I 基本类型

用Java代码调用C\C++代码时候，肯定会有参数数据的传递。两者属于不同的编程语言，在数据类型上有很多差别，应该要知道他们彼此之间的对应类型。例如，尽管C拥有int和long的数据类型，但是他们的实现却是取决于具体的平台。在一些平台上，int类型是16位的，而在另外一些平台上市32位的整数。基于这个原因，Java本地接口定义了jint，jlong等等。

Java Language Type	JNI Type

boolean	jboolean
byte	jbyte
char	jchar
short	jshort
int	jint
long	jlong
float	jfloat
double	jdouble
All Reference type	jobject

由Java类型和C/C++数据类型的对应关系，可以看到，这些新定义的类型名称和Java类型名称具有一致性，只是在前面加了个j，如int对应jint，long对应jlong。

我们看看jni.h和jni_md.h头文件，可以更直观的了解：

```
typedef unsigned char           jboolean;
typedef unsigned short  jchar;
typedef short           jshort;
typedef float           jfloat;
typedef double          jdouble;
typedef long                    jint;
typedef __int64                         jlong;
typedef signed char             jbyte;
```

由jni头文件可以看出，jint对应的是C/C++中的long类型，即32位整数，而不是C/C++中的int类型（C/C++中的int类型长度依赖于平台），它和Java 中int类型一样。

所以如果要在本地方法中要定义一个jint类型的数据，规范的写法应该是 jint i=123L;

再比如jchar代表的是Java类型的char类型，实际上在C/C++中却是unsigned short类型，因为Java中的char类型为两个字节。而在C/C++中有这样的定义：typedef unsigned short wchar_t。所以jchar就是相当于C/C++中的宽字符。所以如果要在本地方法中要定义一个jchar类型的数据，规范的写法应该是jchar c=L'C';

实际上，所有带j的类型，都是代表Java中的类型，并且jni中的类型接口与本地代码在类型大小是完全匹配的，而在语言层次却不一定相同。在本地方法中与JNI接口调用时，要在内部都要转换，我们在使用的时候也需要小心。

l **Java对象类型**

Java对象在C\C++代码中的形式如下：

```
class _jclass : public _jobject {};
class _jthrowable : public _jobject {};
class _jstring : public _jobject {};
class _jarray : public _jobject {};
class _jbooleanArray : public _jarray {};
class _jbyteArray : public _jarray {};
class _jcharArray : public _jarray {};
class _jshortArray : public _jarray {};
class _jintArray : public _jarray {};
class _jlongArray : public _jarray {};
class _jfloatArray : public _jarray {};
class _jdoubleArray : public _jarray {};
class _jobjectArray : public _jarray {};
```

所有的_j开头的类，都是继承于_jobject，这也是Java语言的特别，所有的类都是Object的子类，这些类就是和Java中的类一一对应，只不过名字稍有不同而已。

1) jclass类和如何取得jclass对象

在Java中，Class类型代表一个Java类编译的字节码，即：这个Java类，里面包含了这个类的所有信息。在JNI中，同样定义了这样一个类：jclass。了解反射的人都知道Class类是如何重要，可以通过反射获得java类的信息和访问里面的方法和成员变量。

JNIEnv有几个方法可以取得jclass对象：

```
jclass FindClass(const char *name) {
        return functions->FindClass(this, name);
 }
```

FindClass会在系统classpath环境变量下寻找name类，注意包的间隔使用 “/“，而不是“.“，如：

```
jclass cls_string=env->FindClass("java/lang/String");
```

获得对象对应的jclass类型：

```
jclass GetObjectClass(jobject obj) {
        return functions->GetObjectClass(this,obj);
}
```

获得一个类的父类jclass类型：

```
jclass GetSuperclass(jclass sub) {
      return functions->GetSuperclass(this,sub);
}
```

JNI本地方法访问Java属性和方法

在JNI调用中，不仅仅Java可以调用本地方法，本地代码也可以调用Java中的方法和成员变量。在Java1.0中“原始的”Java到C的绑定中，程序员可以直接访问对象数据域。然而，直接方法要求虚拟机暴露他们的内部数据布局，基于这个原因，JNI要求程序员通过特殊的JNI函数来获取和设置数据以及调用java方法。

1)　　取得代表属性和方法的jfieldID和jmethodID

为了在C/C++中表示属性和方法，JNI在jni.h头文件中定义了jfieldID和jmethodID类型来分别代表Java对象的属性和方法。我们在访问或是设置Java属性的时候，首先就要先在本地代码取得代表该Java属性的jfieldID，然后才能在本地代码进行Java属性操作。同样的，我们需要调用Java对象方法时，也是需要取得代表该方法的jmethodID才能进行Java方法调用。

使用JNIEnv提供的JNI方法，我们就可以获得属性和方法相对应的jfieldID和jmethodID：

l GetFieldID　：取得成员变量的id

l GetStaticFieldID　：取得静态成员变量的id

l GetMethodID　：取得方法的id

l GetStaticMethodID　：取得静态方法的id

jfieldID GetFieldID(jclass clazz, const char *name,const char *sig)

jfieldID GetStaticFieldID(jclass clazz, const char*name, const char *sig)

jmethodID GetStaticMethodID(jclass clazz, const char*name, const char *sig)

jmethodID GetMethodID(jclass clazz, const char *name,constchar *sig)

可以看到这四个方法的参数列表都是一模一样的，下面来分析下每个参数的含义：

第一个参数jclassclazz　：

上一节讲到的jclass类型，相当于Java中的Class类，代表一个Java类，而这里面的代表的就是我们操作的Class类，我们要从这个类里面取的属性和方法的ID。

第二个参数constchar *name：

这是一个常量字符数组，代表我们要取得的方法名或者变量名。

第三个参数constchar *sig：
这也是一个常量字符数组，代表我们要取得的方法或变量的签名。

什么是方法或者变量的签名呢？

我们来看下面的例子，如何来获得属性和方法ID：

```
public class NativeTest {

        publicvoid show(int i){

                System.out.println(i);

        }

public void show(double d){

                System.out.println(d);

        }

}
```

本地代码部分：

```
//首先取得要调用的方法所在的类的Class对象，在C/C++中即jclass对象

jclass clazz_NativeTest=env->FindClass("cn/itcast/NativeTest");

//取得jmethodID

jmethodID id_show=env->GetMethodID(clazz_NativeTest,"show","???");
```

上述代码中的id_show取得的jmethodID到底是哪个show方法呢？由于Java语言有方法重载的面向对象特性，所以只通过函数名不能明确的让JNI找到Java里对应的方法。所以这就是第三个参数sig的作用，它用于指定要取得的属性或方法的类型签名。

2)　　JNI签名：

类型签名	Java 类型	类型签名	Java 类型

Z	boolean	[	[]
B	byte	[I	int[]
C	char	[F	float[]
S	short	[B	byte[]
I	int	[C	char[]
J	long	[S	short[]
F	float	[D	double[]
D	double	[J	long[]
L	fully-qualified-class (全限定的类)	[Z	boolean[]

l 基本类型

以特定的大写字母表示

l 引用类型

Java对象以L开头，然后以“/”分隔包的完整类型，例如String的签名为：Ljava/lang/String;

在Java里数组类型也是引用类型，数组以[开头，后面跟数组元素类型的签名，例如：int[] 签名就是[I ，对于二维数组，如int[][] 签名就是[[I，object数组签名就是[Ljava/lang/Object;

l 方法签名

(参数1类型签名参数2类型签名参数3类型签名.....)返回值类型签名

注意：

函数名，在签名中没有体现出来

参数列表相挨着，中间没有逗号，没有空格

返回值出现在（ ）后面

如果参数是引用类型，那么参数应该为：L类型;

如果函数没有返回值，也要加上V类型

例如：

Java方法	**对应签名**
boolean isLedOn(void) ;	()Z
void setLedOn(int ledNo);	(I)
String substr(String str, int idx, int count);	(Ljava/lang/String;II)Ljava/lang/String
char fun (int n, String s, int[] value);	(ILjava/lang/String;[I)C
boolean showMsg(View v, String msg);	(Landroid/View;Ljava/lang/String;)Z

3) 根据获取的ID，来取得和设置属性，以及调用方法。

l 获得、设置属性和静态属性

取得了代表属性和静态属性的jfieldID，就可以使用JNIEnv中提供的方法来获取和设置属性/静态属性。

获取属性/静态属性的形式：

Get<Type>Field GetStatic<Type>Field。

设置属性/静态属性的形式：

Set<Type>Field SetStatic<Type>Field。

取得成员属性：

jobject **GetObjectField**(jobjectobj, jfieldID fieldID);

jboolean **GetBooleanField**(jobjectobj, jfieldID fieldID);

jbyte **GetByteField**(jobjectobj, jfieldID fieldID);

取得静态属性：

jobject **GetStaticObjectField**(jclassclazz, jfieldID fieldID);

jboolean **GetStaticBooleanField**(jclassclazz, jfieldID fieldID);

jbyte **GetStaticByteField**(jclassclazz, jfieldID fieldID);

Get方法的第一个参数代表要获取的属性所属对象或jclass对象，第二个参数即属性ID。

设置成员属性：

void **SetObjectField**(jobjectobj, jfieldID fieldID, jobject val);

void **SetBooleanField**(jobjectobj, jfieldID fieldID,jboolean val);

void **SetByteField**(jobjectobj, jfieldID fieldID, jbyte val);

设置静态属性：

void **SetStaticObjectField**(jobjectobj, jfieldID fieldID, jobject val);

void **SetStaticBooleanField**(jobjectobj, jfieldID fieldID,jboolean val);

void **SetStaticByteField**(jobjectobj, jfieldID fieldID, jbyte val);

Set方法的第一个参数代表要设置的属性所属的对象或jclass对象，第二个参数即属性ID，第三个参数代表要设置的值。

l 调用方法

取得了代表方法和静态方法的jmethodID，就可以用在JNIEnv中提供的方法来调用方法和静态方法。

调用普通方法：

Call<Type>Method(jobject obj, jmethodID methodID,...);

Call<Type>MethodV(jobject obj, jmethodID methodID,va_listargs);

Call<Type>tMethodA(jobject obj, jmethodID methodID,constjvalue *args);

调用静态方法：

CallStatic<Type>Method(jclass clazz, jmethodID methodID,...);

CallStatic<Type>MethodV(jclass clazz, jmethodID methodID,va_listargs);

CallStatic<Type>tMethodA(jclass clazz, jmethodID methodID,constjvalue *args);

上面的Type这个方法的返回值类型，如Int，Char，Byte等等。

第一个参数代表调用的这个方法所属于的对象，或者这个静态方法所属的类。

第二个参数代表jmethodID。

后面的参数，就代表这个方法的参数列表了。

上述方法的调用有三种形式：

a) Call<Type>Method(jobject obj, jmethodIDmethodID,...);

```
// Java方法
public int show(int i,double d,char c){
…
}

// 本地调用Java方法
jint i=10L;
jdouble d=2.4;
jchar c=L'd';
env->CallIntMethod(obj, id_show, i, d, c);
```

b) Call<Type>MethodV(jobject obj, jmethodIDmethodID,va_list args)

这种方式使用较少。

c) Call<Type>MethodA(jobject obj, jmethodIDmethodID,jvalue* v)

这种调用方式其第三个参数是一个jvalue的指针。jvalue类型是在 jni.h头文件中定义的联合体union，看它的定义：

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

例如：

```
    jvalue * args=new jvalue[3];
    args[0].i=12L;
```

```
        args[1].d=1.2;
        args[2].c=L'c';
        jmethodIDid_goo=env->GetMethodID(env->GetObjectClass(obj),"goo","(IDC)V");
        env->CallVoidMethodA(obj,id_goo,args);
    delete []args;  //释放内存
```

静态方法的调用方式和成员方法调用一样。

3.5 本地创建Java对象

1)　本地代码创建Java对象

JNIEnv提供了下面几个方法来创建一个Java对象：

jobject NewObject(jclass clazz, jmethodID methodID,...);

jobject NewObjectV(jclass clazz, jmethodIDmethodID,va_list args);

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

本地创建Java对象的函数和前面本地调用Java方法很类似：

第一个参数jclass class 代表的你要创建哪个类的对象

第二个参数jmethodID methodID 代表你要使用哪个构造方法ID来创建这个对象。

只要有jclass和jmethodID ，我们就可以在本地方法创建这个Java类的对象。

指的一提的是：由于Java的构造方法的特点，方法名与类名一样，并且没有返回值，所以对于获得构造方法的ID的方法env->GetMethodID(clazz,method_name ,sig)中的第二个参数是固定为类名，第三个参数和要调用的构造方法有关，默认的Java构造方法没有返回值，没有参数。例如：

```
jclassclazz=env->FindClass("java/util/Date");                    //取得java.util.Date类的jclass对象
jmethodID id_date=env->GetMethodID(clazz,"Date","()V");   //取得某一个构造方法的jmethodID
jobject date=env->NewObject(clazz,id_date);                    //调用NewObject方法创建java.util.Date对象
```

2)　本地方法对Java字符串的操作

在Java中，字符串String对象是Unicoode（UTF-16）编码，每个字符不论是中文还是英文还是符号，一个字符总是占用两个字节。在C/C++中一个字符是一个字节， C/C++中的宽字符是两个字节的。所以Java通过JNI接口可以将Java的字符串转换到C/C++的宽字符串（wchar_t*），或是传回一个UTF-8的字符串（char*）到C/C++，反过来，C/C++可以通过一个宽字符串，或是一个UTF-8编码的字符串创建一个Java端的String对象。

可以看下面的一个例子：

在Java端有一个字符串 String str="abcde";，在本地方法中取得它并且输出：

```
void native_string_operation (JNIEnv * env,  jobject obj)
{
    //取得该字符串的jfieldID
jfieldIDid_string=env->GetFieldID(env->GetObjectClass(obj), "str", "Ljava/lang/String;");
    jstringstring=(jstring)(env->GetObjectField(obj, id_string));   //取得该字符串，强转为jstring类型。
  printf("%s",string);
}
```

由上面的代码可知，从java端取得的String属性或者是方法返回值的String对象，对应在JNI中都是jstring类型，它并不是C/C++中的字符串。所以，我们需要对取得的 jstring类型的字符串进行一系列的转换，才能使用。

JNIEnv提供了一系列的方法来操作字符串：

l const jchar *GetStringChars(jstring str, jboolean*isCopy)

将一个jstring对象，转换为（UTF-16）编码的宽字符串（jchar*）。

l const char *GetStringUTFChars(jstring str,jboolean *isCopy)

将一个jstring对象，转换为（UTF-8）编码的字符串（char*）。

这两个函数的参数中，第一个参数传入一个指向Java 中String对象的jstring引用。第二个参数传入的是一个jboolean的指针，其值可以为NULL、JNI_TRUE、JNI_FLASE。

如果为JNI_TRUE则表示开辟内存，然后把Java中的String拷贝到这个内存中，然后返回指向这个内存地址的指针。

如果为JNI_FALSE，则直接返回指向Java中String的内存指针。这时不要改变这个内存中的内容，这将破坏String在Java中始终是常量的规则。

如果是NULL，则表示不关心是否拷贝字符串。

使用这两个函数取得的字符，在不适用的时候，要分别对应的使用下面两个函数来释放内存。

RealeaseStringChars(jstring jstr, const jchar*str)

RealeaseStringUTFChars(jstring jstr, constchar* str)

第一个参数指定一个jstring变量，即要释放的本地字符串的资源

第二个参数就是要释放的本地字符串

3)　创建Java String对象

jstring NewString(const jchar *unicode, jsizelen) // 根据传入的宽字符串创建一个Java String对象

jstring NewStringUTF(const char *utf) // 根据传入的UTF-8字符串创建一个Java String对象

4) 返回Java String对象的字符串长度

jsize GetStringLength(jstring jstr) //返回一个java String对象的字符串长度

jsize GetStringUTFLength(jstring jstr) //返回一个java String对象经过UTF-8编码后的字符串长度

3.6 Java数组在本地代码中的处理

我们可以使用GetFieldID获取一个Java数组变量的ID，然后用GetObjectFiled取得该数组变量到本地方法，返回值为jobject，然后我们可以强制转换为j<Type>Array类型。

typedef jarray jbooleanArray;

typedef jarray jbyteArray;

typedef jarray jcharArray;

typedef jarray jshortArray;

typedef jarray jintArray;

typedef jarray jlongArray;

typedef jarray jfloatArray;

typedef jarray jdoubleArray;

typedef jarray jobjectArray;

j<Type>Array类型是JNI定义的一个对象类型，它并不是C/C++的数组，如int[]数组，double[]数组等等。所以我们要把j<Type>Array类型转换为C/C++中的数组来操作。

JNIEnv定义了一系列的方法来把一个j<Type>Array类型转换为C/C++数组或把C/C++数组转换为j<Type>Array。

jsize GetArrayLength(jarray array) // 获得数组的长度

jobjectArray NewObjectArray(jsize len, jclass clazz, jobjectinit) // 创建对象数组，指定其大小

jobject GetObjectArrayElement(jobjectArray array, jsizeindex) // 获得数组的指定元素

void SetObjectArrayElement(jobjectArray array, jsizeindex,jobject val) // 设置数组元素

jbooleanArrayNewBooleanArray(jsize len) // 创建Boolean数组，指定其大小

jbyteArrayNewByteArray(jsize len) //下面的都类似，创建对应类型的数组，并指定大小

jcharArrayNewCharArray(jsize len)

jshortArrayNewShortArray(jsize len)

jintArrayNewIntArray(jsize len)

jlongArrayNewLongArray(jsize len)

jfloatArrayNewFloatArray(jsize len)

jdoubleArrayNewDoubleArray(jsize len)

// 获得指定类型数组的元素

jboolean * GetBooleanArrayElements(jbooleanArray array,jboolean *isCopy)

jbyte * GetByteArrayElements(jbyteArray array, jboolean*isCopy)

jchar * GetCharArrayElements(jcharArray array, jboolean*isCopy)

jshort * GetShortArrayElements(jshortArray array, jboolean*isCopy)

jint * GetIntArrayElements(jintArray array, jboolean*isCopy)

jlong * GetLongArrayElements(jlongArray array, jboolean*isCopy)

jfloat * GetFloatArrayElements(jfloatArray array,jboolean *isCopy)

jdouble * GetDoubleArrayElements(jdoubleArray array,jboolean *isCopy)

// 释放指定数组

void ReleaseBooleanArrayElements(jbooleanArrayarray,jboolean *elems,jint mode)

void ReleaseByteArrayElements(jbyteArray array,jbyte*elems,jint mode)

void ReleaseCharArrayElements(jcharArray array,jchar*elems,jint mode)

void ReleaseShortArrayElements(jshortArray array,jshort*elems,jint mode)

void ReleaseIntArrayElements(jintArray array,jint*elems,jint mode)

void ReleaseLongArrayElements(jlongArray array,jlong*elems,jint mode)

void ReleaseFloatArrayElements(jfloatArray array,jfloat*elems,jint mode)

void ReleaseDoubleArrayElements(jdoubleArrayarray,jdouble *elems,jint mode)

void * GetPrimitiveArrayCritical(jarray array, jboolean*isCopy)

void ReleasePrimitiveArrayCritical(jarray array, void*carray, jint mode)

void GetBooleanArrayRegion(jbooleanArray array,jsizestart, jsize len, jboolean *buf)

void GetByteArrayRegion(jbyteArray array,jsize start,jsize len, jbyte *buf)

void GetCharArrayRegion(jcharArray array,jsize start,jsize len, jchar *buf)

void GetShortArrayRegion(jshortArray array,jsize start,jsize len, jshort *buf)

void GetIntArrayRegion(jintArray array,jsize start,jsize len, jint *buf)

void GetLongArrayRegion(jlongArray array,jsize start,jsize len, jlong *buf)

void GetFloatArrayRegion(jfloatArray array,jsize start,jsize len, jfloat *buf)

void GetDoubleArrayRegion(jdoubleArray array,jsizestart, jsize len, jdouble *buf)

void SetBooleanArrayRegion(jbooleanArray array, jsizestart, jsize len,const jboolean *buf)

void SetByteArrayRegion(jbyteArray array, jsize start,jsize len,const jbyte *buf)

void SetCharArrayRegion(jcharArray array, jsize start,jsize len,const jchar *buf)

void SetShortArrayRegion(jshortArray array, jsizestart, jsize len,const jshort *buf)

void SetIntArrayRegion(jintArray array, jsize start,jsize len,const jint *buf)

void SetLongArrayRegion(jlongArray array, jsize start,jsize len,const jlong *buf)

void SetFloatArrayRegion(jfloatArray array, jsizestart, jsize len,const jfloat *buf)

void SetDoubleArrayRegion(jdoubleArray array, jsizestart, jsize len,const jdouble *buf)

上面是JNIEnv提供给本地代码调用的数组操作函数，大致可以分为下面几类：

1) **获取数组的长度**

jsize GetArrayLength(jarray array);

2) 对象类型数组的操作

jobjectArray NewObjectArray(jsize len, jclassclazz,jobject init) // 创建

jobject GetObjectArrayElement(jobjectArray array, jsizeindex) // 获得元素

void SetObjectArrayElement(jobjectArray array, jsizeindex,jobject val) // 设置元素

JNI没有提供直接把Java的对象类型数组（Object[ ]）直接转到C++中的jobject[ ]数组的函数。而是直接通过Get/SetObjectArrayElement这样的函数来对Java的Object[ ]数组进行操作

3) 对基本数据类型数组的操作

基本数据类型数组的操作方法比较多，大致可以分为如下几类：

Get<Type>ArrayElements/Realease<Type>ArrayElements;

Get<Type>ArrayElements(<Type>Array arr, jboolean*isCopied);

这类函数可以把Java基本类型的数组转换到C/C++中的数组。有两种处理方式，一是拷贝一份传回本地代码，另一种是把指向Java数组的指针直接传回到本地代码，处理完本地化的数组后，通过Realease<Type>ArrayElements来释放数组。处理方式由Get方法的第二个参数isCopied来决定。

Realease<Type>ArrayElements(<Type>Arrayarr,<Type>* array, jint mode)用这个函数可以选择将如何处理Java和C/C++本地数组：

其第三个参数mode可以取下面的值：

l 0：对Java的数组进行更新并释放C/C++的数组

l JNI_COMMIT：对Java的数组进行更新但是不释放C/C++的数组

l JNI_ABORT：对Java的数组不进行更新，释放C/C++的数组

例如：

Test.java

```
public class Test {
	privateint [] arrays=new int[]{1,2,3,4,5};
	publicnative void show();
	static{
		System.loadLibrary("NativeTest");
	}
	publicstatic void main(String[] args) {
		newTest().show();
	}
}
```

本地方法：

```
void native_test_show(JNIEnv * env,  jobject obj)
{
	jfieldIDid_arrsys=env->GetFieldID(env->GetObjectClass(obj),"arrays","[I");
```

```
        jintArrayarr=(jintArray)(env->GetObjectField(obj, id_arrsys));
        jint*int_arr=env->GetIntArrayElements(arr,NULL);
        jsizelen=env->GetArrayLength(arr);
        for(inti=0; i<len; i++)
        {
                cout<<int_arr[i]<<endl;
        }
        env->ReleaseIntArrayElements(arr,int_arr,JNI_ABORT);
}
```

1.7局部引用与全局引用

1) JNI中的引用变量

Java代码与本地代码里在进行参数传递与返回值复制的时候，要注意数据类型的匹配。对于int, char等基本类型直接进行拷贝即可，对于Java中的对象类型，通过传递引用实现。VM保证所有的Java对象正确的传递给了本地代码，并且维持这些引用，因此这些对象不会被Java的gc（垃圾收集器）回收。因此，本地代码必须有一种方式来通知VM本地代码不再使用这些Java对象，让gc来回收这些对象。

JNI将传递给本地代码的对象分为两种：局部引用和全局引用。

l 局部引用：只在上层Java调用本地代码的函数内有效，当本地方法返回时，局部引用自动回收。

l 全局引用：只有显示通知VM时，全局引用才会被回收，否则一直有效，Java的gc不会释放该引用的对象。

默认的话，传递给本地代码的引用是局部引用。所有的JNI函数的返回值都是局部引用。

```
jstring
MyNewString(JNIEnv *env, jchar *chars, jint len)
{
   static jclassstringClass = NULL;              //static 不能保存一个局部引用
   jmethodID cid;
   jcharArrayelemArr;
   jstringresult;
   if(stringClass == NULL) {
      stringClass = (*env)->FindClass(env, "java/lang/String");    // 局部引用
       if(stringClass == NULL) {
          return NULL; /* exception thrown */
       }
   }
   /* It iswrong to use the cached stringClass here,
      because itmay be invalid. */
   cid =(*env)->GetMethodID(env, stringClass, "<init>","([C)V");
   ...
   elemArr =(*env)->NewCharArray(env, len);
   ...
   result =(*env)->NewObject(env, stringClass, cid, elemArr);
  (*env)->DeleteLocalRef(env, elemArr);
   returnresult;
}
```

2) 手动释放局部引用情况

虽然局部引用会在本地代码执行之后自动释放，但是有下列情况时，要手动释放：

l 本地代码访问一个很大的Java对象时，在使用完该对象后，本地代码要去执行比较复杂耗时的运算时，由于本地代码还没有返回，Java收集器无法释放该本地引用的对象，这时，应该手动释放掉该引用对象。

```
/* A native method implementation */
JNIEXPORT void JNICALL
func(JNIEnv *env, jobject this)
{
   lref =...             /* a large Java object*/
   ...                  /* last use of lref */
  (*env)->DeleteLocalRef(env, lref);
  lengthyComputation();   /* maytake some time */
```

```
    return;              /* all local refs are freed */
}
```

这个情形的实质，就是允许程序在native方法执行期间，java的垃圾回收机制有机会回收native代码不在访问的对象。

l 本地代码创建了大量局部引用，这可能会导致JNI局部引用表溢出，此时有必要及时地删除那些不再被使用的局部引用。比如：在本地代码里创建一个很大的对象数组。

```
for (i = 0; i < len; i++) {
    jstring jstr= (*env)->GetObjectArrayElement(env, arr, i);
    ... /*process jstr */
   (*env)->DeleteLocalRef(env, jstr);
}
```

在上述循环中，每次都有可能创建一个巨大的字符串数组。在每个迭代之后，native代码需要显示地释放指向字符串元素的局部引用。

l 创建的工具函数，它会被未知的代码调用，在工具函数里使用完的引用要及时释放。

l 不返回的本地函数。例如，一个可能进入无限事件分发的循环中的方法。此时在循环中释放局部引用，是至关重要的，这样才能不会无限期地累积，进而导致内存泄露。

局部引用只在创建它们的线程里有效，本地代码不能将局部引用在多线程间传递。一个线程想要调用另一个线程创建的局部引用是不被允许的。将一个局部引用保存到全局变量中，然后在其它线程中使用它，这是一种错误的编程。

3) 全局引用

在一个本地方法被多次调用时，可以使用一个全局引用跨越它们。一个全局引用可以跨越多个线程，并且在被程序员手动释放之前，一直有效。和局部引用一样，全局引用保证了所引用的对象不会被垃圾回收。

JNI允许程序员通过局部引用来创建全局引用, 全局引用只能由NewGlobalRef函数创建。下面是一个使用全局引用例子：

```
jstring
MyNewString(JNIEnv *env, jchar *chars, jint len)
{
    static jclassstringClass = NULL;
    ...
    if(stringClass == NULL) {
        jclasslocalRefCls =
            (*env)->FindClass(env, "java/lang/String");
        if(localRefCls == NULL) {
            return NULL;
        }
        /* Createa global reference */
       stringClass = (*env)->NewGlobalRef(env, localRefCls);
        /* Thelocal reference is no longer useful */
       (*env)->DeleteLocalRef(env, localRefCls);
        /* Is theglobal reference created successfully? */
        if(stringClass == NULL) {
            return NULL; /* out of memory exception thrown */
        }
    }
    ...
}
```

4) 释放全局引用

在native代码不再需要访问一个全局引用的时候，应该调用DeleteGlobalRef来释放它。如果调用这个函数失败，Java VM将不会回收对应的对象。

1.8 本地C代码中创建Java对象及本地JNI对象的保存

1) Android中Bitmap对象的创建

通常在JVM里创建Java的对象就是创建Java类的实例，再调用Java类的构造方法。而有时Java的对象需要在本地代码里创建。以Android中的Bitmap的构建为例，Bitmap中并没有Java对象创建的代码及外部能访问的构造方法，所以它的实例化是在JNI的c中实现的。

BitmapFactory.java中提供了得到Bitmap的方法，时序简化为：

BitmapFactory.java->BitmapFactory.cpp -> GraphicsJNI::createBitmap() [graphics.cpp]

GraphicsJNI::createBitmap()[graphics.cpp]的实现：

```
jobjectGraphicsJNI::createBitmap(JNIEnv* env, SkBitmap*bitmap, bool isMutable,
                                 jbyteArrayninepatch, intdensity)
```

```
{
   SkASSERT(bitmap != NULL);
    SkASSERT(NULL!= bitmap->pixelRef());


    jobject obj=env->AllocObject(gBitmap_class);
    if (obj) {
       env->CallVoidMethod(obj,gBitmap_constructorMethodID,
                          (jint)bitmap,isMutable, ninepatch, density);
        if(hasException(env)) {
            obj =NULL;
        }
    }
    return obj;
}
```

而gBitmap_class的得到是通过：

```
jclass c=env->FindClass("android/graphics/Bitmap");
gBitmap_class =(jclass)env->NewGlobalRef(c);
//gBitmap_constructorMethodID是Bitmap的构造方法（方法名用”<init>”）的jmethodID:
gBitmap_constructorMethodID=env->GetMethodID(gBitmap_class, "<init>",  "(IZ[BI)V");
```

总结一下，c中如何访问Java对象的属性：

1) 通过JNIEnv::FindClass()找到对应的jclass；

2) 通过JNIEnv::GetMethodID()找到类的构造方法的jfieldID；

3) 通过JNIEnv::AllocObject创建该类的对象；

4) 通过JNIEnv::CallVoidMethod()调用Java对象的构造方法。

2) 本地JNI对象保存在Java环境中

C代码中某次被调用时生成的对象，在其他函数调用时是不可见的，虽然可以设置全局变量但那不是好的解决方式，Android中通常是在Java域中定义一个int型的变量，在本地代码生成对象的地方，与这个Java域的变量关联，在别的使用到的地方，再从这个变量中取值。

以JNICameraContext为例来说明：

JNICameraContext是android_hardware_camera.cpp中定义的类型，并会在本地代码中生成对象并与Java中android.hardware.Camera的mNativeContext关联。

在注册native函数之前，C中就已经把Java域中的属性的jfieldID得到了。通过下列方法：

```
jclass clazz =env->FindClass("android/hardware/Camera ");
jfieldID field = env->GetFieldID(clazz, "mNativeContext","I");
```

如果执行成功，把field保存到fileds.context成员变量中。

生成cpp对象时，通过JNIEnv::SetIntField()设置为Java对象的属性

```
static void android_hardware_Camera_native_setup(JNIEnv*env, jobject thiz,
    jobjectweak_this, jintcameraId)
{
    // …
   sp<JNICameraContext>context = new JNICameraContext(env, weak_this,clazz, camera);
    // …
    // 该处通过context.get()得到context对象的地址，保存到了Java中的mNativeContext属性里
env->SetIntField(thiz,fields.context, (int)context.get());
}
```

而要使用时，又通过JNIEnv::GetIntField()获取Java对象的属性，并转化为JNICameraContext类型：

```
   JNICameraContext* context=reinterpret_cast<JNICameraContext*>(env->GetIntField(thiz,fields.context));
    if (context!= NULL) {
        // …
    }
```

总结一下，c++中生成的对象如何保存和使用：

1) 通过JNIEnv::FindClass()找到对应的jclass；

2) 通过JNIEnv::GetFieldID()找到类中属性的jfieldID；

3) 某个调用过程中，生成cpp对象时，通过JNIEnv::SetIntField()设置为Java对象的属性；

4) 另外的调用过程中，通过JNIEnv::GetIntField()获取Java对象的属性，再转化为真实的对象类型。

转 : http://www.xue5.com/Mobile/Mobile/649592.html

沙翁
关注 - 12
粉丝 - 68
+加关注

« 上一篇 : JNIEnv解析
» 下一篇 : 使用 ContentProviderOperation 来提升性能

posted on 2014-10-09 14:42 沙翁 阅读(15056) 评论(0) 编辑 收藏