###### Android JNI基础篇（一）

2017年05月30日 16:00:00

https://blog.csdn.net/kgdwbb/article/details/72810251

阅读数：4633

# **Android JNI基础篇**

## **前言**

JNI学习其实并不难，在这里，我将引导大家学习JNI的基础知识，认真学完本教程，你将更加坚信我说的话。来吧，我们一起学习！

## **JNI基础**

### **JNI是什么？**

JNI的全称就是Java Native Interface，顾名思义，就是Java和C/C++相互通信的接口，就好比买卖房子都需要找中介一样，这里的JNI就是Java和C/C++通信的中介，一个中间人。

### **JNI头文件**

JNI开发前提是要引入jni.h头文件，这个文件Android NDK目录下面

示例如下：

#include<jni.h>

### **怎么加载so库？**

Android提供了3个实用的函数用来加载JNI库，分别是System.loadLibrary(libname)，Runtime.getRuntime().loadLibrary(libname)，以及Runtime.getRuntime().load(libFilePath)。

#### **用loadLibrary函数加载**

用System.loadLibrary(libname)和Runtime.getRuntime().loadLibrary(libname)这两个函数加载so库，不需要指定so库的路径，Android会默认从系统的共享库目录里面去查找，Android的共享库目录就是vendor/lib和system/lib，如果在共享库路径里面找到指定名字的so库，就会立即加载这个so库，所以我们给so库起名的时候要尽量避免和Android共享库里面的so库同名。如果在共享库目录里面查找不到，就会在APP的安装目录里面查找APP的私有so库，如果查找到，会立即加载这个so库。

#### **用load函数加载**

Runtime.getRuntime().load(libFilePath)用这个函数加载so库，需要指定完整的so库路径，优点是加载速度快，并且不会加载错误的so库，缺点就是需要指定完整的so库路径，有时候并不方便，大家常用的加载so库的方式还是用loadLibrary函数来加载。

#### **加载so库示例**

****static**** {

    System.*loadLibrary*(****"native-lib"****);

*//用这种方式加载so库和System.loadLibrary函数加载so库的效果是一样的*

*//Runtime.getRuntime().loadLibrary("native-lib");*

*//String soLibFilePath;*

*//用这种方式加载so库需要指定完整的so库路径*

*//Runtime.getRuntime().load(soLibFilePath);*}

## **Android Studio so库配置**

Android Studio通过CMakeLists.txt文件配置需要生成的so库，下面详细给大家介绍一下这个CMakeLists.txt文件如何配置。

Android Studio通过cmake命令来生成so库。

#### **CMakeLists.txt文件配置详解**

##### **add\_library**

add\_library函数用来配置要生成的so库的基本信息，比如库的名字，要生成的so库是静态的还是共享的，so库的C/C++源文件列表

示例如下：

add\_library( native-lib

             SHARED

             src/main/cpp/native-lib.cpp

             src/main/cpp/native-lib2.cpp

             src/main/cpp/native-lib3.cpp)

第一个参数是so库的名字

第二个参数是要生成的so库的类型，静态so库是STATIC,共享so库是SHARED

第三个参数是C/C++源文件，可以包括多个源文件

##### **find\_library**

find\_library函数用来从NDK目录下面查找特定的so库

示例如下：

find\_library( log-lib

              log )

第一个参数是我们给要查找的so库起的名字，名字可以随便写

第二个参数是要查找的so库的名字，这个名字是从真实的so库的名字去掉前缀和后缀后的名字，比如liblog.so这个so库的名字就是log

##### **target\_link\_libraries**

target\_link\_libraries函数用来把要生成的so库和依赖的其它so库进行链接，生成我们需要的so库文件

示例如下：

target\_link\_libraries( native-lib

                       ${log-lib} )

第一个参数是我们要生成的so库的名字去掉前缀和后缀后的名字，在这个例子中，要生成的真实的so库的名字是libnative-lib.so

第二个参数是链接我们用find\_library函数定义的查找的依赖库的名字log-lib，语法就是${依赖的库的名字}

## **Java和JNI类型对照表**

这里详细介绍一下Java类型和C/C++类型的对照关系，方便我们下面的学习，这一部分知识很基础，也很重要。

### **Java和JNI基本类型对照表**

Java的基本类型可以直接与C/C++的基本类型映射，因此Java的基本类型对开发人员是透明的。

|  |  |  |  |
| --- | --- | --- | --- |
| ****Java类型**** | ****JNI类型**** | ****C/C++类型**** | ****大小**** |
| Boolean | jboolean | unsigned char | 无符号8位 |
| Byte | jbyte | char | 有符号8位 |
| Char | jchar | unsigned short | 无符号16位 |
| Short | jshort | short | 有符号16位 |
| Integer | jint | int | 有符号32位 |
| Long | jlong | long long | 有符号64位 |
| Float | jfloat | float | 32位浮点值 |
| Double | jdouble | double | 64位双精度浮点值 |

### **Java和JNI引用类型对照表**

与Java基本类型不同，引用类型对开发人员是不透明的。Java类的内部数据结构并不直接向原生代码公开。也就是说原生C/C++代码并不能直接访问Java代码的字段和方法。

|  |  |
| --- | --- |
| Java类型 | C/C++类型 |
| java.lang.Class | jclass |
| java.lang.Throwable | jthrowable |
| java.lang.String | jstring |
| java.lang.Object | jobject |
| java.util.Objects | jobjects |
| java.lang.Object[] | jobjectArray |
| Boolean[] | jbooleanArray |
| Byte[] | jbyteArray |
| Char[] | jcharArray |
| Short[] | jshortArray |
| int[] | jintArray |
| long[] | jlongArray |
| float[] | jfloatArray |
| double[] | jdoubleArray |
| 通用数组 | jarray |

说明任何Java数组在JNI里面都可以使用jarray来表示，比如Java的int[]数组，用JNI可以表示为jintArray，也可以表示为jarray

## **JNI函数详解**

### **JNI字符串相关的函数**

#### **C/C++字符串转JNI字符串**

NewString函数用来生成Unicode JNI字符串

NewStringUTF函数用来生成UTF-8 JNI字符串

示例如下：

****extern** **"C"  
JNIEXPORT** **void** **JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_testJString(JNIEnv\* env, jobject thiz,jstring jstr) {  
    ****char****\*str=****"helloboy"****;  
    jstring jstr2=env->NewStringUTF(str);  
  
    ****const****jchar \*jchar2=env->GetStringChars(jstr,****NULL****);  
    size\_t len=env->GetStringLength(jstr);  
    jstring jstr3=env->NewString(jchar2,len);  
}

#### **JNI字符串转C/C++字符串**

GetStringChars函数用来从jstring获取Unicode C/C++字符串

GetStringUTFChars函数用来从jstring获取UTF-8 C/C++字符串

示例如下：

****extern** **"C"  
JNIEXPORT** **void** **JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_testJString(JNIEnv\* env, jobject thiz,jstring jstr) {  
    ****const char****\*str=env->GetStringUTFChars(jstr,****NULL****);  
    ****const****jchar \*jchar2=env->GetStringChars(jstr,****NULL****);  
}

#### **释放JNI字符串**

ReleaseStringChars函数用来释放Unicode C/C++字符串

ReleaseStringUTFChars函数用来释放UTF-8 C/C++字符串

示例如下：

****extern** **"C"  
JNIEXPORT** **void** **JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_testJString(JNIEnv\* env, jobject thiz,jstring jstr) {  
    ****const char****\*str=env->GetStringUTFChars(jstr,****NULL****);  
   env->ReleaseStringUTFChars(jstr,str);  
      
    ****const****jchar \*jchar2=env->GetStringChars(jstr,****NULL****);  
   env->ReleaseStringChars(jstr,jchar2);  
}

#### **JNI字符串截取**

GetStringRegion函数用来截取Unicode JNI字符串

GetStringUTFRegion函数用来截取UTF-8 JNI字符串

示例如下：

****extern** **"C"  
JNIEXPORT** **void** **JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_testJString(JNIEnv\* env, jobject thiz,jstring jstr) {  
    ****const char****\*str=env->GetStringUTFChars(jstr,****NULL****);  
    ****char****\*subStr=****new char****;  
    env->GetStringUTFRegion(jstr,0,3,subStr);  
   env->ReleaseStringUTFChars(jstr,str);  
  
    ****const****jchar \*jchar2=env->GetStringChars(jstr,****NULL****);  
    jchar \*subJstr=****new****jchar;  
    env->GetStringRegion(jstr,0,3,subJstr);  
   env->ReleaseStringChars(jstr,jchar2);  
}

#### **获取JNI字符串的长度**

GetStringLength用来获取Unicode JNI字符串的长度

GetStringUTFLength函数用来获取UTF-8 JNI字符串的长度

示例如下：

****extern** **"C"  
JNIEXPORT** **void** **JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_testJString(JNIEnv\* env, jobject thiz,jstring jstr) {  
    jsize len=env->GetStringLength(jstr);  
    jsize len2=env->GetStringUTFLength(jstr);  
}

### **JNI数组相关的函数**

#### **JNI数组相关的类**

|  |  |
| --- | --- |
| JNI类 | 备注 |
| jbooleanArray | 对应Java的boolean[] |
| jbyteArray | 对应Java的byte[] |
| jcharArray | 对应Java的char[] |
| jshortArray | 对应Java的short[] |
| jintArray | 对应Java的int[] |
| jlongArray | 对应Java的long[] |
| jfloatArray | 对应Java的float[] |
| jdoubleArray | 对应Java的double[] |
| jobjectArray | 对应Java的对象数组object[] |

#### **JNI基本类型数组**

##### **获取JNI基本类型数组元素**

Get<Type>ArrayElements函数用来获取基本类型JNI数组的元素，这里面的<Type>需要被替换成实际的类型，比如GetIntArrayElements，GetLongArrayElements等

示例代码如下：  
****extern** **"C"  
JNIEXPORT** **void** **JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_testJIntArray(JNIEnv\* env, jobject thiz,jintArray array) {  
    jint \*intArray=env->GetIntArrayElements(array,****NULL****);  
    ****int****len=env->GetArrayLength(array);  
    ****for****(****int****i=0;i<len;i++){  
        jint item=intArray[i];  
    }  
}

##### **获取JNI基本类型数组的子数组**

Get<Type>ArrayRegion函数用来获取JNI数组的子数组，这里面的<Type>需要被替换成实际的类型，比如GetIntArrayRegion，GetLongArrayRegion等

示例如下：

****extern** **"C"  
JNIEXPORT** **void** **JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_testJIntArray(JNIEnv\* env, jobject thiz,jintArray array) {  
    jint \*subArray=****new****jint;  
    env->GetIntArrayRegion(array,0,3,subArray);  
}

##### **设置JNI基本类型数组的子数组**

Set<Type>ArrayRegion函数用来获取JNI基本类型数组的子数组，这里面的<Type>需要被替换成实际的类型，比如SetIntArrayRegion，SetLongArrayRegion等

示例如下：

****extern** **"C"  
JNIEXPORT** **void** **JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_testJIntArray(JNIEnv\* env, jobject thiz,jintArray array) {  
    jint \*subArray=****new****jint;  
    env->GetIntArrayRegion(array,0,3,subArray);  
    env->SetIntArrayRegion(array,0,3,subArray);  
}

#### **JNI对象数组**

GetObjectArrayElement函数用来获取JNI对象数组元素

SetObjectArrayElement函数用来设置JNI对象数组元素

示例如下：

****extern** **"C"  
JNIEXPORT** **void** **JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_testJObjectArray(JNIEnv\* env, jobject thiz,jobjectArray array) {  
    ****int****len=env->GetArrayLength(array);  
    ****for****(****int****i=0;i<len;i++)  
    {  
        jobject item=env->GetObjectArrayElement(array,i);  
    }  
}  
  
****extern** **"C"  
JNIEXPORT** **void** **JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_testJStringArray(JNIEnv\* env, jobject thiz,jobjectArray array) {  
    ****int****len=env->GetArrayLength(array);  
    ****for****(****int****i=0;i<len;i++)  
    {  
        jstring item=(jstring)env->GetObjectArrayElement(array,i);  
    }  
}

****extern** **"C"  
JNIEXPORT** **void** **JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_testJObjectArray(JNIEnv\* env, jobject thiz,jobjectArray array) {  
    jobject obj;  
    env->SetObjectArrayElement(array,1,obj);  
}

#### **获取JNI数组的长度**

GetArrayLength用来获取数组的长度

示例如下：

****extern "C"JNIEXPORT void JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_testJObjectArray(JNIEnv\* env, jobject thiz,jobjectArray array) {

****int**** len=env->GetArrayLength(array);

}

****extern** **"C"  
JNIEXPORT** **void** **JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_testJIntArray(JNIEnv\* env, jobject thiz,jintArray array) {  
    ****int****len=env->GetArrayLength(array);  
}

### **JNI NIO缓冲区相关的函数**

使用NIO缓冲区可以在Java和JNI代码中共享大数据，性能比传递数组要快很多，当Java和JNI需要传递大数据时，推荐使用NIO缓冲区的方式来传递。

NewDirectByteBuffer函数用来创建NIO缓冲区

GetDirectBufferAddress函数用来获取NIO缓冲区的内容

GetDirectBufferCapacity函数用来获取NIO缓冲区的大小

示例代码如下：

****extern "C"JNIEXPORT void JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_testDirectBuffer(JNIEnv\* env, jobject thiz) {

****const char**** \*data=****"hello world"****;

****int**** len=strlen(data);

    jobject obj=env->NewDirectByteBuffer((****void****\*)data,len);

****long**** capicity=env->GetDirectBufferCapacity(obj);

****char**** \*data2=(****char****\*)env->GetDirectBufferAddress(obj);

}

### **JNI访问Java类的方法和字段**

#### **Java类型签名映射表**

JNI获取Java类的方法ID和字段ID，都需要一个很重要的参数，就是Java类的方法和字段的签名，这个签名需要通过下面的表来获取，这个表很重要，建议大家一定要记住。

|  |  |
| --- | --- |
| Java类型 | 签名 |
| Boolean | Z |
| Byte | B |
| Char | C |
| Short | S |
| Integer | I |
| Long | J |
| Float | F |
| Double | D |
| Void | V |
| 任何Java类的全名 | L任何Java类的全名;  比如Java String类对应的签名是Ljava/lang/String; |
| type[] | type[  这个就是Java数组的签名，比如Java int[]的签名是[I，Java long[]的签名就是[J，Java String[]的签名是 [Ljava/lang/String; |
| 方法类型 | （参数类型）返回值 类型，  比如Java方法void hello(String msg,String msg2)对应的签名就是(Ljava/lang/String; Ljava/lang/String;)V  再比如Java方法String getNewName(String name)对应的签名是（Ljava/lang/String;) Ljava/lang/String;  再比如Java方法long add(int a,int b)对应的签名是(II)J |

#### **JNI访问Java类方法相关的函数**

##### **JNI访问Java类的实例方法**

GetObjectClass函数用来获取Java对象对应的类类型

GetMethodID函数用来获取Java类实例方法的方法ID

Call<Type>Method函数用来调用Java类实例特定返回值的方法，比如CallVoidMethod，调用java没有返回值的方法，CallLongMethod用来调用Java返回值为Long的方法，等等。

示例如下：

Java代码：

****public native void**** callJavaHelloWorld2();

****public void**** helloWorld2(String msg){

    Log.*i*(****"hello"****,****"hello world "****+msg);

}

JNI代码：

****extern "C"JNIEXPORT void JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_callJavaHelloWorld2(JNIEnv\* env, jobject thiz) {

    jclass clazz=env->GetObjectClass(thiz);

****if****(clazz==****NULL****) ****return****;

    jmethodID helloWorld2\_methodID=env->GetMethodID(clazz,****"helloWorld2"****,****"(java/lang/String;)V"****);

****if****(helloWorld2\_methodID==****NULL****) ****return****;

****const char**** \*msg=****"hello world"****;

    jstring jmsg=env->NewStringUTF(msg);

    env->CallVoidMethod(thiz,helloWorld2\_methodID,jmsg);

}

##### **JNI访问Java类的静态方法**

GetObjectClass函数用来获取Java对象对应的类类型

GetStaticMethodID函数用来获取Java类静态方法的方法ID

CallStatic<Type>Method函数用来调用Java类特定返回值的静态方法，比如CallStaticVoidMethod，调用java没有返回值的静态方法，CallStaticLongMethod用来调用Java返回值为Long的静态方法，等等。

示例如下：

Java代码：

****public native void**** callStaticJavaHelloWorld2();

****public static void**** helloWorldStatic2(String msg){

    Log.*i*(****"hello"****,****"hello world static "****+msg);

}

JNI代码：

****extern "C"JNIEXPORT void JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_callStaticJavaHelloWorld2(JNIEnv\* env, jobject thiz) {

    jclass clazz=env->GetObjectClass(thiz);

****if****(clazz==****NULL****) ****return****;

    jmethodID helloWorldStatic2\_methodID=env->GetStaticMethodID(clazz,****"helloWorldStatic2"****,****"(java/lang/String;)V"****);

****if****(helloWorldStatic2\_methodID==****NULL****) ****return****;

****const char**** \*msg=****"hello world"****;

    jstring jmsg=env->NewStringUTF(msg);

    env->CallStaticVoidMethod(clazz,helloWorldStatic2\_methodID,msg);

}

#### **JNI访问Java类字段相关的函数**

##### **JNI访问Java类实例字段**

GetFieldID函数用来获取Java字段的字段ID

Get<Type>Field用来获取Java类字段的值，比如用GetIntField函数获取Java int型字段的值，用GetLongField函数获取Java long字段的值，用GetObjectField函数获取Java引用类型字段的值

示例如下：

Java代码：

****public class**** Person{

****public**** String ****name****;

****public int age****;

}

****public native void**** getJavaObjectField(Person person);

****private void**** test(){

    Person person=****new**** Person();

    person.****name****=****"wubb"****;

    person.****age****=20;

    getJavaObjectField(person);

}

JNI代码：

****extern "C"JNIEXPORT void JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_getJavaObjectField(JNIEnv\* env, jobject thiz,jobject person) {

    jclass clazz=env->GetObjectClass(person);

    jfieldID name\_fieldID=env->GetFieldID(clazz,****"name"****,****"Ljava/lang/String;"****);

    jstring name=(jstring) env->GetObjectField(person,name\_fieldID);

    jfieldID age\_fieldID=env->GetFieldID(clazz,****"age"****,****"I"****);

    jint age=env->GetIntField(person,age\_fieldID);

}

##### **JNI访问Java类静态字段**

GetStaticFieldID函数用来获取Java静态字段的字段ID

GetStatic<Type>Field用来获取Java类静态字段的值，比如用GetStaticIntField函数获取Java 静态int型字段的值，用GetStaticLongField函数获取Java 静态long字段的值，用GetStaticObjectField函数获取Java静态引用类型字段的值

示例如下：

Java代码：

****public class**** Person {

****public**** String ****name****;

****public int age****;

****public static**** String *name\_static*;

****public static int**** *age\_static*;

}

****public native void**** getJavaObjectStaticField(Person person);

****private void**** test(){

    Person.*name\_static*=****"wubb"****;

    Person.*age\_static*=20;

    Person person=****new**** Person();

    getJavaObjectStaticField(person);

}

JNI代码：

****extern "C"JNIEXPORT void JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_getJavaObjectStaticField(JNIEnv\* env, jobject thiz,jobject person) {

    jclass clazz=env->GetObjectClass(person);

    jfieldID name\_fieldID=env->GetStaticFieldID(clazz,****"name\_static"****,****"Ljava/lang/String;"****);

    jstring name=(jstring) env->GetStaticObjectField(clazz,name\_fieldID);

    jfieldID age\_fieldID=env->GetStaticFieldID(clazz,****"age\_static"****,****"I"****);

    jint age=env->GetStaticIntField(clazz,age\_fieldID);

}

### **JNI线程同步相关的函数**

JNI可以使用Java对象进行线程同步

MonitorEnter函数用来锁定Java对象

MonitorExit函数用来释放Java对象锁

示例如下：

Java代码:

jniLock(****new**** Object());

JNI代码：

****extern "C"JNIEXPORT void JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_jniLock(JNIEnv\* env, jobject thiz,jobject obj) {

    env->MonitorEnter(obj);

*//do something*

env->MonitorExit(obj);

}

### **JNI异常相关的函数**

#### **JNI处理Java异常**

当JNI函数调用的Java方法出现异常的时候，并不会影响JNI方法的执行，但是我们并不推荐JNI函数忽略Java方法出现的异常继续执行，这样可能会带来更多的问题。我们推荐的方法是，当JNI函数调用的Java方法出现异常的时候，JNI函数应该合理的停止执行代码。

ExceptionOccurred函数用来判断JNI函数调用的Java方法是否出现异常

ExceptionClear函数用来清除JNI函数调用的Java方法出现的异常

请看如下示例：

Java代码

****public void**** helloWorld(){

****throw new**** NullPointerException(****"null pointer occurred"****);

*//Log.i("hello","hello world");*}

C++代码

****extern "C"JNIEXPORT void JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_callJavaHelloWorld(JNIEnv\* env, jobject thiz) {

    jclass clazz=env->GetObjectClass(thiz);

****if****(clazz==****NULL****) ****return****;

    jmethodID helloWorld\_methodID=env->GetMethodID(clazz,****"helloWorld"****,****"()V"****);

****if****(helloWorld\_methodID==****NULL****) ****return****;

    env->CallVoidMethod(thiz,helloWorld\_methodID);

****if****(env->ExceptionOccurred()!=****NULL****){

        env->ExceptionClear();

        \_\_android\_log\_print(***ANDROID\_LOG\_VERBOSE***,****"hello"****,****"%s"****,****"program end with java exception"****);

****return****;

    }

    \_\_android\_log\_print(***ANDROID\_LOG\_VERBOSE***,****"hello"****,****"%s"****,****"program end normallly"****);

}

#### **JNI抛出Java类型的异常**

JNI通过ThrowNew函数抛出Java类型的异常

示例如下：

Java代码

****try****{

    testNativeException();

}****catch**** (NullPointerException e){

    e.printStackTrace();

}

C++代码

****extern "C"JNIEXPORT void JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_testNativeException(JNIEnv\* env, jobject thiz) {

    jclass clazz=env->FindClass(****"java/lang/NullPointerException"****);

****if****(clazz==****NULL****) ****return****;

    env->ThrowNew(clazz,****"null pointer exception occurred"****);

}

### **JNI对象的全局引用和局部引用**

我们知道Java代码的内存是由垃圾回收器来管理，而JNI代码则不受Java的垃圾回收器来管理，所以JNI代码提供了一组函数，来管理通过JNI代码生成的JNI对象，比如jobject，jclass，jstring，jarray等，对于这些对象，我们不能简单的在JNI代码里面声明一个全局变量，然后把JNI对象赋值给全局变量，我们需要采用JNI代码提供的专有函数来管理这些全局的JNI对象。

#### **JNI对象的局部引用**

在JNI接口函数中引用JNI对象的局部变量，都是对JNI对象的局部引用，一旦JNI接口函数返回，所有这些JNI对象都会被自动释放。不过我们也可以采用JNI代码提供的DeleteLocalRef函数来删除一个局部JNI对象引用

请看下面的示例代码：

****extern** **"C"  
JNIEXPORT** **void** **JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_testDeleteLocalRef(JNIEnv\* env, jobject thiz) {  
    jclass clazz=env->GetObjectClass(thiz);  
    ****if****(clazz==****NULL****) ****return****;  
    jmethodID helloWorld\_methodID=env->GetMethodID(clazz,****"helloWorld"****,****"()V"****);  
    ****if****(helloWorld\_methodID==****NULL****) ****return****;  
   env->CallVoidMethod(thiz,helloWorld\_methodID);  
    env->DeleteLocalRef(clazz);  
}

#### **JNI对象的全局引用**

对于JNI对象，绝对不能简单的声明一个全局变量，在JNI接口函数里面给这个全局变量赋值这么简单，一定要使用JNI代码提供的管理JNI对象的函数，否则代码可能会出现预想不到的问题。JNI对象的全局引用分为两种，一种是强全局引用，这种引用会阻止Java的垃圾回收器回收JNI代码引用的Java对象，另一种是弱全局引用，这种全局引用则不会阻止垃圾回收器回收JNI代码引用的Java对象。

##### **强全局引用**

NewGlobalRef用来创建强全局引用的JNI对象

DeleteGlobalRef用来删除强全局引用的JNI对象

示例如下：

jobject gThiz;  
****extern** **"C"  
JNIEXPORT** **void** **JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_testStrongGlobalRef(JNIEnv\* env, jobject thiz) {  
    *//gThiz=thiz;//不能这样给全局JNI对象赋值，要采用下面这种方式*gThiz=env->NewGlobalRef(thiz);*//生成全局的JNI对象引用，这样生成的全局的JNI对象才可以在其它函数中使用*env->DeleteGlobalRef(gThiz);*//假如我们不需要gThiz这个全局的JNI对象引用，我们可以把它删除掉*}

##### **弱全局引用**

NewWeakGlobalRef用来创建弱全局引用的JNI对象

DeleteWeakGlobalRef用来删除弱全局引用的JNI对象

IsSameObject用来判断两个JNI对象是否相同

示例如下：  
jobject gThiz;  
****extern** **"C"  
JNIEXPORT** **void** **JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_testWeakGlobalRef(JNIEnv\*env, jobject thiz) {  
    *//gThiz=thiz;//不能这样给全局JNI对象赋值，要采用下面这种方式*gThiz=env->NewWeakGlobalRef(thiz);*//生成全局的JNI对象引用，这样生成的全局的JNI对象才可以在其它函数中使用*****if****(env->IsSameObject(gThiz,****NULL****)){  
        *//弱全局引用已经被Java的垃圾回收器回收*}  
  
    env->DeleteWeakGlobalRef(gThiz);*//假如我们不需要gThiz这个全局的JNI对象引用，我们可以把它删除掉*}

## **Java代码和JNI代码通信**

### **Java通过JNI接口调用C/C++方法**

首先我们需要在Java代码里面声明Native方法原型，比如：

****public native void**** helloJNI(String msg);

其次我们需要在C/C++代码里面声明JNI方法原型，比如：

****extern "C"JNIEXPORT void JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_helloJNI(JNIEnv\* env, jobject thiz,jstring msg) {

*//do something*}

现在这段JNI函数声明代码采用的是C++语言写的，所以需要添加****extern** **"C"****声明，如果源代码是C语言，则不需要添加这个声明。

****JNIEXPORT****这个关键字说明这个函数是一个可导出函数，学过C/C++的朋友都知道，C/C++ 库里面的函数有些可以直接被外部调用，有些不可以，原因就是每一个C/C++库都有一个导出函数列表，只有在这个列表里面的函数才可以被外部直接调用，类似Java的public函数和private函数的区别。

****JNICALL****说明这个函数是一个JNI函数，用来和普通的C/C++函数进行区别，实际发现不加这个关键字，Java也是可以调用这个JNI函数的。

****Void****说明这个函数的返回值是void，如果需要返回值，则把这个关键字替换成要返回的类型即可。

Java\_com\_kgdwbb\_jnistudy\_MainActivity\_helloJNI(JNIEnv\*env, jobject thiz,jstring msg)这是完整的JNI函数声明，JNI函数名的原型如下：

Java\_ + JNI方法所在的完整的类名，把类名里面的”.”替换成”\_” + 真实的JNI方法名，这个方法名要和Java代码里面声明的JNI方法名一样+ JNI函数必须的默认参数(JNIEnv\* env, jobjectthiz)

env参数是一个指向JNIEnv函数表的指针，

thiz参数代表的就是声明这个JNI方法的Java类的引用

msg参数就是和Java声明的JNI函数的msg参数对于的JNI函数参数

### **JNI函数的原型**

[extern “C”]JNIEXPORT 函数返回值 JNICALL 完整的函数声明(JNIENV \*env, jobject thiz, …)

其中extern “C”根据需要动态添加，如果是C++代码，则必须要添加extern “C”声明，如果是C代码，则不用添加

### **静态JNI方法和实例JNI方法区别**

先看一个示例：

Java代码：

****public native void**** showHello();****public native static void**** showHello2();

C++代码：

****extern "C"JNIEXPORT void JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_showHello(JNIEnv\* env, jobject thiz) {

*//do something*}

****extern "C"JNIEXPORT void JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_showHello2(JNIEnv\* env, jclass thiz) {

*//do something*}

相信明眼的同学很快就能发现这两个JNI函数的区别，对就是这个区别，普通的JNI方法对应的JNI函数的第二个参数是jobject类型，而静态的JNI方法对应的JNI函数的第二个参数是jclass类型

### **常见的Java JNI方法声明和JNI函数声明示例**

Java Native方法声明：

****public class**** Person{

****public**** String ****name****;

****public int age****;

}

****public native void**** helloJNI(String msg);****public native int**** func1(****int**** a,****int**** b);****public native**** String func2(String str);****public native void**** func3(****boolean**** b);****public native void**** func4(Person person);

****public native static void**** func5();

C++JNI函数声明：

****extern "C"JNIEXPORT void JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_helloJNI(JNIEnv\* env, jobject thiz,jstring msg) {

*//do something*}

****extern "C"JNIEXPORT**** jint ****JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_func1(JNIEnv\* env, jobject thiz,jint a,jint b) {

*//do something*}

****extern "C"JNIEXPORT**** jstring ****JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_func2(JNIEnv\* env, jobject thiz,jstring str) {

*//do something*}

****extern "C"JNIEXPORT void JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_func3(JNIEnv\* env, jobject thiz,jboolean b) {

*//do something*}

****extern "C"JNIEXPORT void JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_func4(JNIEnv\* env, jobject thiz,jobject person) {

*//do something*}

****extern "C"JNIEXPORT void JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_func5(JNIEnv\* env, jclass thiz) {

*//do something*}

所有的Java类对象在JNI函数里面都使用jobject来表示

## **JNI代码和Java代码通信**

### **C++调用Java实例方法示例**

Java代码

****public native void**** callJavaHelloWorld();****public native void**** callJavaHelloWorld2();****public native void**** callJavaHelloWorld3();

****public void****helloWorld(){  
    Log.*i*(****"hello"****,****"helloworld"****);  
}  
  
****public void****helloWorld2(String msg){  
    Log.*i*(****"hello"****,****"helloworld "****+msg);  
}  
  
****public void****helloWorld3(****int****a,****int****b){  
    ****int****c=a+b;  
    Log.*i*(****"hello"****,****"helloworld c="****+c);  
}

C++代码

****extern "C"JNIEXPORT void JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_callJavaHelloWorld(JNIEnv\* env, jobject thiz) {

    jclass clazz=env->GetObjectClass(thiz);

****if****(clazz==****NULL****) ****return****;

    jmethodID helloWorld\_methodID=env->GetMethodID(clazz,****"helloWorld"****,****"()V"****);

****if****(helloWorld\_methodID==****NULL****) ****return****;

    env->CallVoidMethod(thiz,helloWorld\_methodID);

}

****extern "C"JNIEXPORT void JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_callJavaHelloWorld2(JNIEnv\* env, jobject thiz) {

    jclass clazz=env->GetObjectClass(thiz);

****if****(clazz==****NULL****) ****return****;

    jmethodID helloWorld2\_methodID=env->GetMethodID(clazz,****"helloWorld2"****,****"(java/lang/String;)V"****);

****if****(helloWorld2\_methodID==****NULL****) ****return****;

****const char**** \*msg=****"hello world"****;

    jstring jmsg=env->NewStringUTF(msg);

    env->CallVoidMethod(thiz,helloWorld2\_methodID,jmsg);

}

****extern "C"JNIEXPORT void JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_callJavaHelloWorld3(JNIEnv\* env, jobject thiz) {

    jclass clazz=env->GetObjectClass(thiz);

****if****(clazz==****NULL****) ****return****;

    jmethodID helloWorld3\_methodID=env->GetMethodID(clazz,****"helloWorld3"****,****"(II)V"****);

****if****(helloWorld3\_methodID==****NULL****) ****return****;

    env->CallVoidMethod(clazz,helloWorld3\_methodID,2,3);

}

### **C++调用Java静态方法示例**

Java代码

****public native void**** callStaticJavaHelloWorld();****public native void**** callStaticJavaHelloWorld2();****public native void**** callStaticJavaHelloWorld3();

****public static void****helloWorldStatic(){  
    Log.*i*(****"hello"****,****"helloworld static"****);  
}  
  
****public static void****helloWorldStatic2(String msg){  
    Log.*i*(****"hello"****,****"helloworld static "****+msg);  
}  
  
****public static void****helloWorldStatic3(****int****a,****int****b){  
    ****int****c=a+b;  
    Log.*i*(****"hello"****,****"helloworld static c="****+c);  
}

C++代码

****extern "C"JNIEXPORT void JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_callStaticJavaHelloWorld(JNIEnv\* env, jobject thiz) {

    jclass clazz=env->GetObjectClass(thiz);

****if****(clazz==****NULL****) ****return****;

    jmethodID helloWorldStatic\_methodID=env->GetStaticMethodID(clazz,****"helloWorldStatic"****,****"()V"****);

****if****(helloWorldStatic\_methodID==****NULL****) ****return****;

    env->CallStaticVoidMethod(clazz,helloWorldStatic\_methodID);

}

****extern "C"JNIEXPORT void JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_callStaticJavaHelloWorld2(JNIEnv\* env, jobject thiz) {

    jclass clazz=env->GetObjectClass(thiz);

****if****(clazz==****NULL****) ****return****;

    jmethodID helloWorldStatic2\_methodID=env->GetStaticMethodID(clazz,****"helloWorldStatic2"****,****"(java/lang/String;)V"****);

****if****(helloWorldStatic2\_methodID==****NULL****) ****return****;

****const char**** \*msg=****"hello world"****;

    jstring jmsg=env->NewStringUTF(msg);

    env->CallStaticVoidMethod(clazz,helloWorldStatic2\_methodID,msg);

}

****extern "C"JNIEXPORT void JNICALL****Java\_com\_kgdwbb\_jnistudy\_MainActivity\_callStaticJavaHelloWorld3(JNIEnv\* env, jobject thiz) {

    jclass clazz=env->GetObjectClass(thiz);

****if****(clazz==****NULL****) ****return****;

    jmethodID helloWorldStatic3\_methodID=env->GetStaticMethodID(clazz,****"helloWorldStatic3"****,****"(II)V"****);

****if****(helloWorldStatic3\_methodID==****NULL****) ****return****;

    env->CallStaticVoidMethod(clazz,helloWorldStatic3\_methodID,2,3);

}

## **本篇结束语**

到这里，相信大家对JNI基础知识都会有一个清晰的认识，现在大家就可以运用本篇学到的知识进行JNI开发了，你们都是好样的。

JNI开发真的很简单，大家只要多加练习，肯定能快速掌握，灵活运用本篇学到的JNI基础知识。

加油，看好你们。