

# 001-1、JNI交流

## 一、如何实现一个JNI的调用

### (一) 常规JNI写法步骤：

- 1、编写声明了native方法的Java类,调用System.loadLibrary("库名");

```
class XXXX{
    private/public static native int add(int x,int y);
    static{
        System.loadLibrary("jniLib");
    }
}
```

- 2、将Java源代码编译成class字节码文件

A、直接用AndroidStudio，编译：Build->Make Project

忽略错误，class: app/build/intermediates/class下

B、在terminal中执行：javac src/com/study/jnilearn/HelloWorld.java -d ./bin

注：注意命令执行的路径；-d制定生成的Class的存放路径

- 3、生成h头文件

用javah是jdk自带的一个命令，-jni参数表示将class中用native声明的函数生成jni规则的函数

javah -jni -classpath ./bin -o HelloWorld.h com.study.jnilearn.HelloWorld

或者

javah -jni -classpath ./bin -d ./jni com.study.jnilearn.HelloWorld

注：-classpath，类搜索路径；-d指定头文件的存放路径；-o：指定生成的头文件名称，默认以类全路径名生成（包名+类名.h）；-

d和-o不能同时指定

- 4、编写C/C++代码实现h头文件中的函数

注意：此处C/C++代码的编写需符合JNI规范

- 5、将本地代码编译成动态库

编写Android.mk，参考frameworks/base/media/jni/Android.mk

(windows: \*.dll, Linux/unix: \*.so, mac os x: \*.jnilib)

- 6、拷贝动态库至 java.library.path 本地库搜索目录下，并运行程序

### (二) 动态注册方法实现JNI标准步骤：

- 1、编写声明了native方法的Java类，调用System.loadLibrary("库名");

- 2、将Java源代码编译成class字节码文件

- 3、用javah -jni命令生成h头文件（javah是jdk自带的一个命令，-jni参数表示将class中用native声明的函数生成jni规则的函数）

- 4、编写C/C++代码实现h头文件中的函数

--> 利用onload方法和RegisterNatives方法动态注册，编写C/C++

- 5、将本地代码编译成动态库

- 6、拷贝动态库至 java.library.path 本地库搜索目录下，并运行程序

### (三) 利用AndroidStudio和CMake和NDK实现JNI

- 1、编写声明了native方法的Java类，调用System.loadLibrary("库名");

- 2、将Java源代码编译成class字节码文件

- 3、用javah -jni命令生成h头文件（javah是jdk自带的一个命令，-jni参数表示将class中用native声明的函数生成jni规则的函数）

- 4、编写C/C++代码实现h头文件中的函数

--> 利用onload方法和RegisterNatives方法动态注册，编写C/C++

- 5、将本地代码编译成动态库

- 6、拷贝动态库至 java.library.path 本地库搜索目录下，并运行程序

--> 编写cmake文件

## 二、对标准写法代码做解释

- 1、java中编写native方法无须多解释

- 2、观察生成的头文件中的方法

2-1、方法名：Java+package name+class name+function name，以下划线连接

2-2、JNIEXPORT 和JNICALL（了解即可）

Windows环境：

JNIEXPORT = \_\_declspec(dllexport) // 见Windows JDK下的jni\_md.h文件

含义：dll动态库规定，如果动态库中的函数要被外部调用，需要在函数声明中添加\_\_declspec(dllexport)标识，表示将该函数导出

在外部可以调用

JNICALL = \_stdcall // 见Windows JDK下的jni\_md.h文件

含义：\_stdcall，用于约束函数入栈顺序和堆栈清理的规则。

Linux环境：

Linux下的jni\_md.h头文件可以看出来，JNIEXPORT 和 JNICALL是一个空定义

## 2-3、返回类型和参数 -->数据类型

1) 基本数据类型：可以直接访问

JAVA类型	本地类型	JNI中自定义类型	描述
int	long	jint/jsize	signed 32 bits
long	_int64	jlong	signed 64 bits
byte	signed char	jbyte	signed 8 bits
boolean	unsigned char	jboolean	unsigned 8 bits
char	unsigned short	jchar	unsigned 16 bits
short	short	jshort	signed 16 bits
float	float	jfloat	32 bits
double	double	jdouble	64 bits
void	void	void	N/A

2) 引用数据类型：相当于C/C++的指针，需要利用JNIEnv来获取

例如：Java中String，native程序要获取其值的方式 `const char *c_str = (*env)->GetStringUTFChars(env, j_str, &isCopy);`

jobject	(all objects)
jclass	(java.lang.Class instances)
jstring	(java.lang.String instances)
jarray	(arrays)
jobjectArray	(Object[])
jbooleanArray	(boolean[])
jbyteArray	(byte[])
jcharArray	(char[])
jshortArray	(short[])
jintArray	(int[])
jlongArray	(long[])
jfloatArray	(float[])
jdoubleArray	(double[])
jthrowable	(java.lang.Throwable objects)

## 2-4、Signature

方法签名格式：(形参参数列表)返回值类型。注意：形参参数列表之间不需要用空格或其它字符分隔

类描述符格式：L包名路径/类名;，包名之间用/分隔。如：Ljava/lang/String;

数组类型描述符：[类型]，其描述符为：[I]

域	Java 语言
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double

## 2-5、头文件：jni.h

JNIEnv/jfieldID/jmethodID/jclass jobject 理解

env指针是一个包含了JVM接口的结构，它包含了与JVM进行交互以及与Java对象协同工作所必需的函数，示例中的JNI函数可以在本地数组和Java数组类型之间、本地字符串和Java字符串类型之间进行转换，其功能还包括对象的实例化、抛出异常等。基本上您可以使用JNIEnv来实现所有Java能做到的事情，虽然要简单很多。

更加正式的解释是这样的，本地代码通过调用JNI的函数来访问JVM，这是通过一个界面指针实现的（界面指针实际上是指向指针的指针），该指针指向一个指针数组，数组中的每个指针都指向了一个界面函数，而每个界面函数都是在数组中预先定义过的。

2-6、思考：

A、Java如何调用到（找到）对应的native方法？

答案：符合JNI的规范，两种方式：

1)、native的方法名必须符合JNI规范（通常用javah预处理java class自动生成头文件），然后Java文件执行

```
System.loadLibrary();
```

当我们熟悉了JNI的native函数命名规则之后，就可以不用通过javah命令去生成相应Java native方法的函数原型了，只需要按照函数命名规则编写相应的函数原型和实现即可。比如com.study.jni.Utils类中还有一个计算加法的native实例方法add，有两个int参数和一个int返回值：public native int add(int num1, int num2)，对应JNI的函数原型就是：JNIEXPORT jint JNICALL

```
Java_com_study_jni_Utils_add(JNIEnv *, jobject, jint, jint);
```

2)、动态注册法

B、native方法，如何访问Java传递过来的参数？

C、native代码中，如何获取class实例；为什么可以获取？ --- 有点像反射

```
jclass clazz = env->FindClass("android/media/MediaRecorder");
```

D、native代码中，如何获取java中的方法；为什么可以获取？

#### 四、常用的参考代码和书写步骤

思考：

- 1、Java为什么可以调到C/C++中的方法
- 2、Java为什么可以取得C/C++中的返回值
- 3、Java如何传递自己的参数给C/C++
- 4、C/C++如何调用Java中的方法

##### (一) C/C++访问Java实例方法和静态方法

###### 1、访问Java实例方法：

- 1>、调用GetObjectClass函数获取实例对象的Class引用
- 2>、获取一个实例方法的ID，使用GetMethodID函数，传入方法名称和方法签名
- 3>、调用实例方法使用CallXXXMethod/V/A函数，XXX代表返回的数据类型，如：CallIntMethod

###### 2、访问Java静态方法：

- 1>、调用FindClass函数获取类的Class引用
- 2>、获取一个静态方法的ID，使用GetStaticMethodID函数，传入方法名称和方法签名调用静态方法使用
- 3>、CallStaticXXXMethod/V/A函数，XXX代表返回值的的数据类型。如：CallStaticIntMethod

###### 3、在Native代码中new一个Class对象的方法：

- 1>、调用FindClass函数获取类的Class引用
- 2>、获取构造方法ID，方法名称使用" <init>"
- 3>、创建一个类的实例，使用NewObject函数，传入Class引用和构造方法ID

注：调用GetMethodID获取方法ID和调用FindClass获取Class实例后，要做异常判断

获取一个类的Class实例，使用FindClass函数，传入类描述符。JVM会从classpath目录下开始搜索。

特别注意：

删除局部变量引用，使用DeleteLocalRef，传入引用变量

##### (二) C/C++访问Java实例变量和静态变量

1、由于JNI函数是直接操作JVM中的数据结构，不受Java访问修饰符的限制。即，在本地代码中可以调用JNI函数可以访问Java对象中的非public属性和方法

###### 2、访问和修改实例变量操作步骤：

- 1>、调用GetObjectClass函数获取实例对象的Class引用
- 2>、调用GetFieldID函数获取Class引用中某个实例变量的ID
- 3>、调用GetXXXField函数获取变量的值，需要传入实例变量所属对象和变量ID
- 4>、调用SetXXXField函数修改变量的值，需要传入实例变量所属对象、变量ID和变量的值

###### 3、访问和修改静态变量操作步骤：

- 1>、调用FindClass函数获取类的Class引用
- 2>、调用GetStaticFieldID函数获取Class引用中某个静态变量ID
- 3>、调用GetStaticXXXField函数获取静态变量的值，需要传入变量所属Class的引用和变量ID
- 4>、调用SetStaticXXXField函数设置静态变量的值，需要传入变量所属Class的引用、变量ID和变量的值

```
//定义变量
```

```
jclass cls; //clazz
```

```
jobject obj;
```

```
jfieldID fid;
```

```

jemthodID mid_construct
jemthodID mid
jstring j_str;
jint num;
//获取xxx指定的java类
cls = (*env)->FindClass(env, "xxx")
// 调用默认构造函数，实例化xxx类
obj = (*env)->AllocObject(env, cls);

// 调用指定构造方法，实例化xxx类
mid_construct = (*env)->GetMethodID(env, cls, "<init>", "()V"); //获取特定构造方法
obj = (*env)->NewObject(env, cls, mid_construct);

//获取类xxx的非静态方法
mid = (*env)->GetMethodID(env, cls, "sayHello", "(Ljava/lang/String;)Ljava/lang/String;");
//获取类xxx的静态方法sayHello,signature=(Ljava/lang/String;)Ljava/lang/String;
mid = (*env)->GetStaticMethodID(env, cls, "sayHello", "(Ljava/lang/String;)Ljava/lang/String;");

//调用Java的方法sayHello
jstring result = (jstring)(*env)->CallStaticObjectMethod(env, cls, mid, arg)
jstring result = (jstring)(*env)->CallObjectMethod(env, obj, mid);

//类实例变量str的属性ID
fid = (*env)->GetFieldID(env,cls,"str", "Ljava/lang/String;");
//获取ClassField类静态变量num的属性ID
fid = (*env)->GetStaticFieldID(env, cls, "num", "I");

//获取实例变量str的值，假设fid为String类型
j_str = (jstring)(*env)->GetObjectField(env,obj,fid);
//获取静态变量num的值，假设fid为int类型
j_int = (*env)->GetStaticIntField(env,cls,fid);

// 6.删除局部引用
(*env)->DeleteLocalRef(env, cls);
(*env)->DeleteLocalRef(env, obj);
(*env)->DeleteLocalRef(env, j_str);

```

## 五、JNI动态注册方法标准步骤：

- 1、编写声明了native方法的Java类，调用System.loadLibrary("库名");
- 2、将Java源代码编译成class字节码文件
- 3、用javah -jni命令生成h头文件（javah是jdk自带的一个命令，jni参数表示将class中用native声明的函数生成jni规则的函数）
- 4、用本地代码实现h头文件中的函数  
----> 见下面(利用JNI规范中的onload方法和(\*env)->RegisterNatives)
- 5、将本地代码编译成动态库
- 6、拷贝动态库至 java.library.path 本地库搜索目录下，并运行程序

### 具体实现和解释：

1> JNINativeMethod

```

static JNINativeMethod gMethods[] = {
    {"setDataSource", "(Ljava/lang/String;)V", (void *)com_media_ffmpeg_FFMpegPlayer_setDataSource},
    {"_setVideoSurface", "(Landroid/view/Surface;)(void *)com_media_ffmpeg_FFMpegPlayer_setVideoSurface",
    {"prepare", "()V", (void *)com_media_ffmpeg_FFMpegPlayer_prepare},
};

```

JNINativeMethod 结构体的官方定义:

```

typedef struct {
    const char* name; //name：是Java中函数的名字。

```

```

const char* signature; //signature : 用字符串描述了Java中函数的参数和返回值
void* fnPtr; //fnPtr : 是函数指针, 指向native函数。前面都要接 (void *)
} JNINativeMethod;

```

2>我们必须重写JNI\_OnLoad ( ) 方法这样就会当调用 System.loadLibrary( "XXXX" )方法的时候直接来调用JNI\_OnLoad ( )

```

JNIEXPORT jint JNI_OnLoad(JavaVM* vm, void* reserved);
JNIEXPORT void JNI_OnUnload(JavaVM* vm, void* reserved);

```

```

jint JNI_OnLoad(JavaVM* vm, void* /* reserved */)
{
    JNIEnv* env = NULL;
    jint result = -1;

    if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {
        ALOGE("ERROR: GetEnv failed\n");
        goto bail;
    }
    assert(env != NULL);

    if (register_android_media_ImageWriter(env) != JNI_OK) {
        ALOGE("ERROR: ImageWriter native registration failed");
        goto bail;
    }
    /* success -- return valid version number */
    result = JNI_VERSION_1_4;

bail:
    return result;
}

```

3> jint RegisterNatives(jclass clazz, const JNINativeMethod\* methods, jint nMethods)

标准写法:

```

#define NELEM (sizeof(x)/sizeof(*(x)))
int register_android_media_MediaRecorder(JNIEnv *env)
{
    return AndroidRuntime::registerNativeMethods(env,
        "android/media/MediaRecorder", gMethods, NELEM(gMethods));
}

```

——JNI资料整理——

Java官方文档: <https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/jniTOC.html>

Android Studio add C/C++ : <https://developer.android.com/studio/projects/add-native-code.html>

JNI/NDK开发指南:<http://blog.csdn.net/xyang81/article/category/2759987>

JNI 实战全面解析 : <http://blog.csdn.net/banketree/article/details/40535325>

专栏学习android JNI 的那些事儿 : <http://blog.csdn.net/column/details/jnijni.html>

Android中关于JNI 的学习 : <http://blog.csdn.net/foolsheep/article/category/2244007>

JNI 实战全面解析 (JNI手册) : <http://blog.csdn.net/banketree/article/details/40535325>

Android中JNI的使用方法 : <http://www.cnblogs.com/bastard/archive/2012/05/19/2508913.html>

安卓实战开发之JNI : <http://blog.csdn.net/u013278099/article/details/51927631>