layout: post title: "浅析JNI" date: 2013-06-26 22:31 comments: true

## categories:

最近的项目和一些图像处理有关，需要用C、C++实现，生成so文件，再通过JNI结合到Android的app中，有时候项目需要还会查看android的源码，做些调整，也会涉及到许多so文件，了解了一些JNI的技术。并且，正在读的一本书叫《深入理解Android》卷一，作者：邓平凡。该书是写的深入浅出，作者功力深厚，大力推荐购买。本文关于JNI的技术大部分参考该书第二章的内容，有兴趣的同学可以购买该书查看原文，这里作为我个人关于JNI的知识整理。

在进入正题之前，需要读者了解一些预备知识，比如关于JNI环境的配置，第一个jni程序hello-jni实现，具体参考：http://whbzju.github.io/blog/2013/06/01/android-jni-config/

# 内容概述

本文从以下4个部分进行：

1. Java层，声明、使用native方法
2. Java与Native如何关联，即注册的方式与实现
3. Java与Native方法通信，即如何互相调用
4. Java与Native的数据结构对应关系

我相信，如果你弄懂了以上的问题，可以使用jni技术进行基本的开发。本文通过实现一个简单的demo，对以上的问题的进行解答。

# Java层---声明、使用native方法

先看MediaScanner.java的代码

{% codeblock MediaScanner lang:java %}

public class jniActivity extends Activity {

```
private TextView tv;

/**
 * Called when the activity is first created.
 */
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    setContentView(R.layout.main);

    TextView  tv = new TextView(this);
    tv.setText( stringFromJNI() );
    setContentView(tv);
}

// 这个方法采用静态注册，参考ndk自带的例子hello-jni的实现
public native String  stringFromJNI();

// 修改成动态注册
public native String DynamicStringFromJNI();

// 提供方法，让native层调用
public void testMethodForNativeCallJava(){
    Toast.makeText(getApplicationContext(), "Call from Native",
Toast.LENGTH_SHORT).show();
}

// 加载jni库
static{
   System.loadLibrary("learnJni");
}
```

}

{% endcodeblock %}

可以大致猜到，static块load了jni的库文件，有好几个带native声明的方法，这些方法都没有具体的实现，其实现应该在native层也就是c层。该方法对java来讲使用起来没什么不同。Java层需要做的事情就结束了，秘密应该就在这两个我们不熟悉的部分。总结下Java层的工作：

- 通过static块来加载对应的jni库
- 声明由关键字native修饰的方法

看来Jni对Java程序员还是很友好的，使用起来很方便。

# Java与Native如何关联---注册JNI方法

首先，我们看下最简单的native方法实现，修改自ndk自带的sample：hello-jni的hellojni.c

{% codeblock learnjni lang:java %}

# include <string.h>

# include <jni.h>

/ *This is a trivial JNI example where we use a native method * to return a new VM String. See the corresponding Java source * file located at: * *
/ jstring Java_com_example_learnjni_LearnJni_stringFromJNI( JNIEnv *env, jobject thiz ) { return (*env)->NewStringUTF(env, "Hello from JNI !"); } {% endcodeblock %}

明显，这里的函数名好奇怪，一大串。其实如果你有jni的基础，你会发现这个函数名和你知道的静态注册和动态注册都不一样，这里并没有使用包含方法签名的头文件，而是使用虚拟机默认的函数调用方式。

注册这个概念不难理解，打个比方，java和c是两个世界的东西，若要将他们连接在一起，必须要有一个统一的沟通标准，注册就是将双方的方法函数用一个标准描述，通知对方。JNI里面有两种注册方式，分别是静态注册和动态注册。下面我们来详细介绍下这两种注册方式：

# 静态注册

静态注册比较简单，其步骤有二：

1. 编译声明了native函数的java类，对每个生成的class用javah生成一个头文件，包含了native方法的签名。操作如下：`javah -o output package.classname`，这样会生成一个output.h的jni头文件，package.classname是java编译好的class文件。
2. 在native层包含这个头文件，实现里面的函数声明。比如生成learnjni头文件的步骤如下：
    1. 先生成class文件，由于learnjni是一个android工程，无法直接用javac生成class文件。此时有两种方式处理：一是通过android源码编译环境生成class；二是借助ice生成class，比如eclipse或者intellij idea，找到它们的输出路径。比如我的intellij idea输出路径是`IdeaProject/*/out`
    2. `cd IdeaProject/*/out`
    3. `javah -o output -classpath ~/IdeaProjects/Learn/learn-jni/src/com.example.learn_jni.jniActivity`
    4. 生成的output头文件如下： {% codeblock output lang:java %} / *DO NOT EDIT THIS FILE - it is machine generated* /

        # include <jni.h>

        / *Header for class com_example_learn_jni_jniActivity* /

# ifndef _Included_com_example_learn_jni_jniActivity

# define _Included_com_example_learn_jni_jniActivity

# ifdef __cplusplus

extern "C" {

# endif

/ * *Class: com_example_learn_jni_jniActivity * Method: stringFromJNI * Signature: ()Ljava/lang/String;* /
JNIEXPORT jstring JNICALL Java_com_example_learn_1jni_jniActivity_stringFromJNI (JNIEnv *, jobject);

/ * *Class: com_example_learn_jni_jniActivity * Method: DynamicStringFromJNI * Signature: ()Ljava/lang/String;* / JNIEXPORT jstring JNICALL

Java_com_example_learn_1jni_jniActivity_DynamicStringFromJNI (JNIEnv *, jobject);

# ifdef __cplusplus

}

# endif

# endif

{% endcodeblock %}

需要解释下静态方法中native方法是如何和jni函数对应上。当Java层调用native方法，比如stringFromJNI时，它会从对应的JNI库中寻找 `Java_com_example_learn_1jni_jniActivity_stringFromJNI`，如果找不到就报错。如果找到，则为二者建立一个映射关系，其实是保存了jni层函数的函数指针。当然，这项工作由虚拟机完成。

## 小结

可以明显的看出，静态注册方法有不少弊端。

1. 需要用javah对java类生成头文件，而且生成的jni层函数名字特别长，不方便书写。
2. 若是将来函数名有改动，或者native函数数量有变化，都需要重新生成头文件，不易维护
3. 初次调用native函数要根据函数名字搜索对应的jni层函数来建立关联关系，影响效率。

## 未完待续