

Android JNI/NDK 学习笔记

likunarmstrong@gmail.com

JNI，全称Java Native Interface，是用于让运行在JVM中的Java代码和运行在JVM外的Native代码（主要是C或者C++）沟通的桥梁。代码编写者即可以使用JNI从Java的程序中调用Native代码，又可以从Native程序中调用Java代码。这样，编程人员可以将低阶的代码逻辑包装到高阶的程序框架中，获得高性能高效率的同时保证了代码框架的高抽象性。

在Android中，仅有以下类库是允许在JNI中使用的：

- libc (C library) headers
- libm (math library) headers
- JNI interface headers
- libz (Zlib compression) headers
- liblog (Android logging) header
- OpenGL ES 1.1 (3D graphics library) headers (since 1.6)
- A Minimal set of headers for C++ support

JNI本身仅仅是一个把两者融合的工具，作为编程者需要做的，就是在Java代码和Native代码中按照固定的格式告诉JNI如何调用对方。在Android中，有两种方式可以调用JNI，一种是Google release的专门针对Android Native开发的工具包，叫做NDK。去Android网站上下载该工具包后，就可以通过阅读里面的文档来setup一个新的包含Native代码的工程，创建自己的Android.mk文件，编译等等；另一种是完整的源码编译环境，也就是通过git从官方网站获取完全的Android源代码平台。这个平台中提供有基于make的编译系统。更多细节请参考[这里](#)。不管选择以上两种方法的哪一个，都必须编写自己的Android.mk文件，有关该文件的编写请参考相关文档。

下面通过一个简单的使用例子来讲解JNI。Android给C和C++提供的是两套不同的Native API，本文仅以C++举例说明。假设这么一个需求，Java代码需要打印一个字符串，而该字符串需要Native代码计算生成。对应的JNI流程是这样的：

1. 在准备打印字符串的Android类中，添加两段代码。

第一段是：

```
private native String getPrintStr();
```

这一行代码的目的是告诉JNI，这个Java文件中有这么一个函数，该函数是在Native代码中执行的，Native代码会返回一个字符串供Java代码来输出。

第二段是：

```
try {System.loadLibrary("LIBNAME" )
```

```
catch (UnsatisfiedLinkError ule) {Log.e(TAG, "Could not load native library");}
```

这两行代码是告诉JNI，你需要找的所有Native函数都在libLIBNAME.so这个动态库中。注意JNI会自动补全lib和so给LIBNAME，你只需要提供LIBNAME给loadLibrary就行了。在最后执行的时候，JNI会先找到这个动态库，然后找里面的OnLoad函数，具体注册流程由OnLoad函数接管。

关于如何确定这个LIBNAME，和如何定义OnLoad函数，下面就会讲。

2. 上面的第一步是告诉JNI，java代码需要和Native代码交互，同时把在哪里找，找什么都通知了。接下来的事情就由Native端接管。

如果把上面的getPrintString函数申明比作原型，那么本地代码中的具体函数定义就应该和该原型匹配，JNI才能知道具体在哪里执行代码。具体来说，应该有一个对应的Native函数，有和Java中定义的函数同样的参数列表以及返回值。另外，还需要有某种机制让JNI将两者相互映射，方便参数和返回值的传递。在老版的JNI中，这是通过丑陋的命名匹配实现的，比如说在Java中定义的函数名是getPrintStr，该函数属于package java.come.android.xxx，那么中对应Native代码中的函数名就应该是Java_com_android_xxx_getPrintStr。这样给开发人员带来了很大不便。可以用javah命令来生成对应Java code中定义函数的Native code版本header文件，从中得知传统的匹配方法是如何做的。具体过程如下：

- a. 通过SDK的方式编译Java代码。
- b. 找到Eclipse的工程目录，进入bin目录下。这里是编译出的java文件所对应的class文件所在。
- c. 假设包括Native函数调用的java文件属于com.android.xxx package，名字叫test.java，那么在bin下执行javah -jni com.android.xxx test

执行完后，可以看到一个新生成的header文件，名字为com_android_xxx_test.h。打开后会发现已经有一个函数申明，函数名为java_com_android_xxx_test_getPrintStr。这个名字就包括了该函数所对应Java版本所在的包，文件以及名称。这就是JNI传统的确定名字的方法。

值得注意的是，header文件中不仅包含了基于函数名的映射信息，还包含了另一个重要信息，就是signature。一个函数的signature是一个字符串，描述了这个函数的参数和返回值。其中"()"中的字符表示参数，后面的则代表返回值。例如"()V"就表示void Func(); "(II)V"表示 void Func(int, int); 数组则以"[]"开始，用两个字符表示。

具体的每一个字符的对应关系如下：

字符	Java类型	C类型
V	void	void
I	jint	int
Z	jboolean	boolean
J	jlong	long
D	jdouble	double
F	jfloat	float
B	jbyte	byte
C	jchar	char

S	jshort	short
---	--------	-------

上面的都是基本类型。如果Java函数的参数是class，则以"L"开头，以";"结尾，中间是用"/" 隔开的包及类名。而其对应的C函数名的参数则为jobject。一个例外是String类，其对应的类为jstring。举例：

```
Ljava/lang/String; String jstring
Ljava/net/Socket; Socket jobject
```

如果JAVA函数位于一个嵌入类，则用\$作为类名间的分隔符。例如 "(Ljava/lang/String;Landroid/os/FileUtils\$FileStatus;)Z"

这个signature非常重要，是下面要介绍的新版命名匹配方法的关键点之一。所以，即使传统的命名匹配已经不再使用，javah这一步操作还是必须的，因为可以从中得到Java代码中需要Native执行的函数的签名，以供后面使用。

3. 在新版（版本号大于1.4）的JNI中，Android提供了另一个机制来解决命名匹配问题，那就是JNI_OnLoad。正如前面所述，每一次JNI执行Native代码，都是通过调用JNI_OnLoad实现的。下面的代码是针对本例的OnLoad代码：

```
/* Returns the JNI version on success, -1 on failure.
jint JNI_OnLoad(JavaVM* vm, void* reserved) {
    JNIEnv* env = NULL;
    jint result = -1;
    if (vm->GetEnv((void**) &env, JNI_VERSION_1_4) != JNI_OK) {
        LOGE("ERROR: GetEnv failed");
        goto bail;
    }
    assert(env != NULL);
    if (!register_Test(env)) {
        LOGE("ERROR: Test native registration failed");
        goto bail;
    }
    /* success -- return valid version number */
    result = JNI_VERSION_1_4;
bail:
    return result;
}
```

仔细分析这个函数。首先，OnLoad通过GetEnv函数获取JNI的环境对象，然后通过register_Test来注册Native函数。register_Test的实现如下：

```
int register_Test(JNIEnv *env) {
    const char* const ClassPathName = "com/android/xxx/test";
    return registerNativeMethods(env, ClassPathName, TestMethods,
        sizeof(TestMethods) / sizeof(TestMethods[0]));
}
```

在这里，ClassPathName是Java类的全名，包括package的全名。只是用"/"代替"."。然后我们把

类名以及TestMethods这个参数一同送到registerNativeMethods这个函数中注册。这个函数是基于JNI_OnLoad的命名匹配方式的重点。

在JNI中，代码编写者通过函数signature名和映射表的配合，来告诉JNI_OnLoad，你要找的函数在Native代码中是如何定义的（signature），以及在哪定义的（映射表）。关于signature的生成和含义，在上面已经介绍。而映射表，是Android使用的一种用于映射Java和C/C++函数的数组，这个数组的类型是JNINativeMethod，定义为：

```
typedef struct {
    const char* name;
    const char* signature;
    void* fnPtr;
} JNINativeMethod;
```

其中，第一个变量是Java代码中的函数名称。第二个变量是该函数对应的Native signature。第三个变量是该函数对应的Native函数的函数指针。例如，在上面register_Test的函数实现中，传给registerNativeMethods的参数TestMethods就是映射表，定义如下：

```
static JNINativeMethod TestMethods[] = {
    {"getPrintStr", "()Ljava/lang/String", (void*)test_getPrintStr}
};
```

其中getPrintStr是在Java代码中定义的函数的名称，()Ljava/lang/String是签名，因为该函数无参数传入，并返回一个String。test_getPrintStr则是我们即将在Native code中定义的函数名称。该映射表和前面定义的类名ClassPathName一起传入registerNativeMethods：

```
static int registerNativeMethods(JNIEnv* env, const char* className, JNINativeMethod*
Methods, int numMethods) {
    jclass clazz;
    clazz = env->FindClass(className);
    if (clazz == NULL) {
        LOGE("Native registration unable to find class '%s'", className);
        return JNI_FALSE ;
    }
    if (env->RegisterNatives(clazz, gMethods, numMethods) < 0) {
        LOGE("RegisterNatives failed for '%s'", className);
        return JNI_FALSE;
    }
    return JNI_TRUE;
}
```

在这里，先load目标类，然后注册Native函数，然后返回状态。

可以看出，通过映射表方式，Java code中的函数名不须再和Native code中的函数名呆板对应。只需要将函数注册进映射表中，Native code的函数编写就有了很大的灵活性。虽说和前一种传统的匹配方法比，这种方式并没有效率上的改进，因为两者本质上都是从JNI load开始做函数映射。但是这一种register的方法极大降低了两边的耦合性，所以实际使用中会受欢迎得多。比如说，由于映射表是一个<名称，函数指针>对照表，在程序执行时，可多次调用registerNativeMethods()函数来更换本

地函数指针，而达到弹性抽换本地函数的目的。

4. 接下来本应介绍test_getPrintStr。但在此之前，简单介绍Android.mk，也就是编译NDK所需要的Makefile，从而完成JNI信息链的讲解。Android.mk可以基于模版修改，里面重要的变量包括：
 - a. LOCAL_C_INCLUDES：包含的头文件。这里需要包含JNI的头文件。
 - b. LOCAL_SRC_FILES: 包含的源文件。
 - c. LOCAL_MODULE：当前模块的名称，也就是第一步中我们提到的LIBNAME。注意这个需要加上lib前缀，但不需要加.so后缀，也就是说应该是libLIBNAME。
 - d. LOCAL_SHARED_LIBRARIES：当前模块需要依赖的共享库。
 - e. LOCAL_PRELINK_MODULE：该模块是否被启动就加载。该项设置依具体程序的特性而定。
5. 至此，JNI作为桥梁所需要的所有信息均已就绪。JNI知道在调用Java代码中的getPrintStr函数时，需要执行Native代码。于是通过System.loadLibrary所加载的libLIBNAME.so找到OnLoad入口。在OnLoad中，JNI发现了函数映射表，发现getPrintStr对应的Native函数是test_getPrintStr。于是JNI将参数（如果有的话）传递给test_getPrintStr并执行，再将返回值（如果有的话）传回Java中的getPrintStr。
6. 用于最后测试的test_getPrintStr函数实现如下：

```
const jstring testStr = env->NewStringUTF("hello, world");
return testStr;
```

然后在Java代码中打印出返回的字符串即可。[这个网页](#)详细介绍了env可以调用的所有方法。

7. 关于测试时使用Log。调用JNI进行Native Code的开发有两种环境，完整源码环境以及NDK。两种环境对应的Log输出方式也并不相同，差异则主要体现在需要包含的头文件中。如果是在完整源码编译环境下，只要include <utils/Log.h>头文件（位于Android-src/system/core/include/cutils），就可以使用对应的LOGI、LOGD等方法了，当然LOG_TAG，LOG_NDEBUG等宏值需要自定义。如果是在NDK环境下编译，则需要include <android/log.h>头文件（位于ndk/android-ndk-r4/platforms/android-8/arch-arm/usr/include/android/），另外自己定义宏映射，例如：

```
#include <android/log.h>
#ifndef LOG_TAG
#define LOG_TAG "MY_LOG_TAG"
#endif
#define LOGD(...) __android_log_print(ANDROID_LOG_DEBUG,LOG_TAG,__VA_ARGS__)
#define LOGI(...) __android_log_print(ANDROID_LOG_INFO,LOG_TAG,__VA_ARGS__)
#define LOGW(...) __android_log_print(ANDROID_LOG_WARN,LOG_TAG,__VA_ARGS__)
#define LOGE(...) __android_log_print(ANDROID_LOG_ERROR,LOG_TAG,__VA_ARGS__)
#define LOGF(...) __android_log_print(ANDROID_LOG_FATAL,LOG_TAG,__VA_ARGS__)
```

另外，在Android.mk文件中对类库的应用在两种环境下也不相同。如果是NDK环境下，需要包括

```
LOCAL_LDLIBS := -llog
```

而在完整源码环境下，则需要包括

```
LOCAL_SHARED_LIBRARIES := libutils libcutils
```

