Android NDK开发：JNI实战篇

紧接上篇：[Android NDK开发：JNI基础篇 | cfanr](http://cfanr.cn/2017/07/29/Android-NDK-dev-JNI-s-foundation/)，这篇主要介绍JNI Native层调用Java层的代码（涉及JNI数据类型映射和描述符的使用）和如何动态注册JNI。

## 1.Hello World NDK

在开始实战练习前，你需要先大致了解运行一个Hello World的项目大概需要做什么，有哪些配置以及配置的具体意思。**Android Studio(2.2以上版本)提供两种方式编译原生库：CMake(默认方式)和ndk-build**。对于初学者可以先了解CMake的方式，另外，对于本文可以暂时不用了解so库如何编译和使用。

一个Hello World的NDK项目很简单，按照流程新建一个native库工程就可以，由于太简单，而且网上也有很多教程，这里就没必要浪费时间再用图文介绍了。详细操作方法，可以参考这篇文章，[AS2.2使用CMake方式进行JNI/NDK开发-于连林- CSDN博客](http://blog.csdn.net/yulianlin/article/details/53168350)

列出项目中涉及NDK的内容或配置几点需要注意的地方：

* .externalNativeBuild文件是CMake编译好的文件，显示支持的各种硬件平台的信息，如ARM、x86等；
* cpp文件是放置native文件的地方，名字可以修改成其他的（只要里面函数名字对应Java native方法就好）；
* CMakeLists.txt，AS自动生成的CMake脚本配置文件

|  |
| --- |
| # value of 3.4.0 or lower.  # 1.指定cmake版本  cmake\_minimum\_required(VERSION 3.4.1)  add\_library( # Sets the name of the library. ——>2.生成函数库的名字，需要写到程序中的 so 库的名字  native-lib  # Sets the library as a shared library. 生成动态函数  SHARED  # Provides a relative path to your source file(s).  # Associated headers in the same location as their source  # file are automatically included. ——> 依赖的cpp文件，每添加一个 C/C++文件都要添加到这里，不然不会被编译  src/main/cpp/native-lib.cpp  )  find\_library( # Sets the name of the path variable. 设置path变量的名称  log-lib  # Specifies the name of the NDK library that  # you want CMake to locate. #指定要查询库的名字  log )  # Specifies libraries CMake should link to your target library. You  # can link multiple libraries, such as libraries you define in the  # build script, prebuilt third-party libraries, or system libraries.  target\_link\_libraries( # Specifies the target library. 目标库, 和上面生成的函数库名字一致  native-lib  # Links the target library to the log library  # included in the NDK. 连接的库，根据log-lib变量对应liblog.so函数库  ${log-lib} ) |

build.gradle文件，注意两个externalNativeBuild {}的配置

|  |
| --- |
| apply plugin: 'com.android.application'  android {  compileSdkVersion 25  buildToolsVersion "25.0.2"  defaultConfig {  applicationId "cn.cfanr.jnisample"  minSdkVersion 15  targetSdkVersion 25  versionCode 1  versionName "1.0"  testInstrumentationRunner "android.support.test.runner.AndroidJUnitRunner"  externalNativeBuild {  cmake {  cppFlags "" //如果使用 C++11 标准，则改为 "-std=c++11"  // 生成.so库的目标平台，使用的是genymotion模拟器，需要加上 x86  abiFilters "armeabi-v7a", "armeabi", "x86"  }  }  }  buildTypes {  release {  minifyEnabled false  proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'  }  }  externalNativeBuild {  cmake {  path "CMakeLists.txt" //配置 CMake 文件的路径  }  }  } |

local.properties文件会多了ndk路径的配置

|  |
| --- |
| ndk.dir=/Users/cfanr/Library/Android/sdk/ndk-bundle  sdk.dir=/Users/cfanr/Library/Android/sdk |

MainActivity 调用 so 库

|  |
| --- |
| public class MainActivity extends AppCompatActivity {  // Used to load the 'native-lib' library on application startup.  static {  System.loadLibrary("native-lib");  }  //……  } |

另外，还需要回顾上篇 JNI 基础篇-静态注册也提到 JNI 的函数命名规则：JNIEXPORT 返回值 JNICALL Java\_全路径类名\_方法名\_参数签名(JNIEnv\* , jclass, 其它参数);**其中第二个参数，当 java native 方法是 static 时，为 jclass，当为非静态方法时，为 jobject**，为了简单起见，下面的例子 JNI 函数都标记extern "C"，函数名就不需要写参数签名了

**2. JNI 函数访问 Java 对象的变量**

*注：以下练习中 Java native 方法都是非静态的*  
步骤：

* 1）通过env->GetObjectClass(jobject)获取Java 对象的 class 类，返回一个 jclass；
* 2）调用env->GetFieldID(jclazz, fieldName, signature)得到该实例域（变量）的 id，即 jfieldID；如果变量是静态 static 的，则调用的方法为 GetStaticFieldID
* 3）最后通过调用env->Get{type}Field(jobject, fieldId) 得到该变量的值。其中{type} 是变量的类型；如果变量是静态 static 的，则调用的方法是GetStatic{type}Field(jclass, fieldId)，注意 static 的话， 是使用 jclass 作为参数；

**2.1 访问某个变量，并通过某个方法对其处理后返回**

native方法定义和调用

|  |
| --- |
| public int num = 10;  public native int addNum();  FieldJni fieldJni = new FieldJni();  Log.e(TAG, "调用前：num = " + fieldJni.num);  Log.e(TAG, "调用后：" + fieldJni.addNum()); |

C++层：

|  |
| --- |
| extern "C"  JNIEXPORT jint JNICALL  Java\_cn\_cfanr\_jnisample\_FieldJni\_addNum(JNIEnv \*env, jobject jobj) {  //获取实例对应的 class  jclass jclazz = env->GetObjectClass(jobj);  //通过class获取相应的变量的 field id  jfieldID fid = env->GetFieldID(jclazz, "num", "I");  //通过 field id 获取对应的值  jint num = env->GetIntField(jobj, fid); //注意，不是用 jclazz, 使用 jobj  num++;  return num;  } |

输出结果：

|  |
| --- |
| MainActivity: 调用前：num = 10  MainActivity: 调用后：11 |

由于 jclass 也是继承 jobject，所以使用 GetIntField 时不要混淆两个参数

**2.2 访问一个 static 变量，并对其修改**

native方法定义和调用

|  |
| --- |
| public static String name = "cfanr";  public native void accessStaticField();  //调用  Log.e(TAG, "调用前：name = " + fieldJni.name);  fieldJni.accessStaticField();  Log.e(TAG, "调用后：" + fieldJni.name); |

C++代码：

|  |
| --- |
| extern "C"  JNIEXPORT void JNICALL  Java\_cn\_cfanr\_jnisample\_FieldJni\_accessStaticField(JNIEnv \*env, jobject jobj) {  jclass jclazz = env->GetObjectClass(jobj);  jfieldID fid = env->GetStaticFieldID(jclazz, "name", "Ljava/lang/String;"); //注意是用GetStaticFieldID，不是GetFieldID  jstring name = (jstring) env->GetStaticObjectField(jclazz, fid);  const char\* str = env->GetStringUTFChars(name, JNI\_FALSE);  /\*  \* 不要用 == 比较字符串  \* name == (jstring) "cfanr"  \* 或用 = 直接赋值  \* name = (jstring) "navy"  \* 警告：warning: result of comparison against a string literal is unspecified (use strncmp instead) [-Wstring-compare]  \*/  char ch[30] = "hello, ";  strcat(ch, str);  jstring new\_str = env->NewStringUTF(ch);  // 将jstring类型的变量，设置到java  env->SetStaticObjectField(jclazz, fid, new\_str);  } |

输出结果：

|  |
| --- |
| MainActivity: 调用前：name = cfanr  MainActivity: 调用后：hello, cfanr |

需要注意的是，获取 java 静态变量，都是调用 JNI 相应静态的函数，不能调用非静态的，同时留意传入的参数是 jclass，而不是 jobject

**2.3 访问一个 private 变量，并对其修改**

native方法定义和调用

|  |
| --- |
| private int age = 21;  public native void accessPrivateField();  public int getAge() {  return age;  }  //调用  Log.e(TAG, "调用前：age = " + fieldJni.getAge());  fieldJni.accessPrivateField();  Log.e(TAG, "调用后：age = " + fieldJni.getAge()); |

C++:

|  |
| --- |
| extern "C"  JNIEXPORT void JNICALL  Java\_cn\_cfanr\_jnisample\_FieldJni\_accessPrivateField(JNIEnv \*env, jobject jobj) {  jclass clazz = env->GetObjectClass(jobj);  jfieldID fid = env->GetFieldID(clazz, "age", "I");  jint age = env->GetIntField(jobj, fid);  if(age > 18) {  age = 18;  } else {  age--;  }  env->SetIntField(jobj, fid, age);  } |

输出结果：

|  |
| --- |
| MainActivity: 调用前：age = 21  MainActivity: 调用后：age = 18 |

**3. JNI 函数调用 Java 对象的方法**

步骤：（和访问 Java 对象的变量有点类型）

* 1）通过env->GetObjectClass(jobject)获取Java 对象的 class 类，返回一个 jclass；
* 2）通过env->GetMethodID(jclass, methodName, sign)获取到 Java 对象的方法 Id，即 jmethodID，当获取的方法是 static 的时，使用GetStaticMethodID;
* 3）通过 JNI 函数env->Call{type}Method(jobject, jmethod, param...)实现调用 Java的方法；若调用的是 static 方法，则使用CallStatic{type}Method(jclass, jmethod, param...)，使用的是 jclass

**3.1 调用 Java 公有方法**

native方法定义和调用

|  |
| --- |
| private String sex = "female";  public void setSex(String sex) {  this.sex = sex;  }  public String getSex(){  return sex;  }  public native void accessPublicMethod();  //调用  MethodJni methodJni = new MethodJni();  Log.e(TAG, "调用前：getSex() = " + methodJni.getSex());  methodJni.accessPublicMethod();  Log.e(TAG, "调用后：getSex() = " + methodJni.getSex()); |

C++

|  |
| --- |
| extern "C"  JNIEXPORT void JNICALL  Java\_cn\_cfanr\_jnisample\_MethodJni\_accessPublicMethod(JNIEnv \*env, jobject jobj) {  //1.获取对应 class 的实体类  jclass jclazz = env->GetObjectClass(jobj);  //2.获取方法的 id  jmethodID mid = env->GetMethodID(jclazz, "setSex", "(Ljava/lang/String;)V");  //3.字符数组转换为字符串  char c[10] = "male";  jstring jsex = env->NewStringUTF(c);  //4.通过该 class 调用对应的 public 方法  env->CallVoidMethod(jobj, mid, jsex);  } |

结果：

|  |
| --- |
| MainActivity: 调用前：getSex() = female  MainActivity: 调用后：getSex() = male |

调用 java private 方法也是类似， Java 的访问域修饰符对 C++无效

**3.2 调用 Java 静态方法**

native方法定义和调用

|  |
| --- |
| private static int height = 170;  public static int getHeight() {  return height;  }  public native int accessStaticMethod();  //调用  Log.e(TAG, "调用静态方法：getHeight() = " + methodJni.accessStaticMethod()); |

C++

|  |
| --- |
| extern "C"  JNIEXPORT jint JNICALL  Java\_cn\_cfanr\_jnisample\_MethodJni\_accessStaticMethod(JNIEnv \*env, jobject jobj) {  //1.获取对应 class 实体类  jclass jclazz = env->GetObjectClass(jobj);  //2.通过 class 类找到对应的方法 id  jmethodID mid = env->GetStaticMethodID(jclazz, "getHeight", "()I"); //注意静态方法是调用GetStaticMethodID, 不是GetMethodID  //3.通过 class 调用对应的静态方法  return env->CallStaticIntMethod(jclazz, mid);  } |

输出结果：  
MainActivity: 调用静态方法：getHeight() = 170

注意调用的静态方法要一致。

**3.3 调用 Java 父类方法**

native方法定义和调用

|  |
| --- |
| public class SuperJni {  public String hello(String name) {  return "welcome to JNI world, " + name;  }  }  public class MethodJni extends SuperJni{  public native String accessSuperMethod();  }  //调用  Log.e(TAG, "调用父类方法：hello(name) = " + methodJni.accessSuperMethod()); |

C++

|  |
| --- |
| extern "C"  JNIEXPORT jstring JNICALL  Java\_cn\_cfanr\_jnisample\_MethodJni\_accessSuperMethod(JNIEnv \*env, jobject jobj) {  //1.通过反射获取 class 实体类  jclass jclazz = env-> FindClass("cn/cfanr/jnisample/SuperJni"); //注意 FindClass 不要 L和;  if(jclazz == NULL) {  char c[10] = "error";  return env->NewStringUTF(c);  }  //通过 class 找到对应的方法 id  jmethodID mid = env->GetMethodID(jclazz, "hello", "(Ljava/lang/String;)Ljava/lang/String;");  char ch[10] = "cfanr";  jstring jstr = env->NewStringUTF(ch);  return (jstring) env->CallNonvirtualObjectMethod(jobj, jclazz, mid, jstr);  } |

注意两点不同的地方，

* 获取的是父类的方法，所以不能通过GetObjectClass获取，需要通过反射 FindClass 获取；
* 调用父类的方法是 CallNonvirtual{type}Method 函数。Nonvirtual是非虚拟函数

**4. Java 方法传递参数给 JNI 函数**

native 方法既可以传递基本类型参数给 JNI（可以不经过转换直接使用），也可以传递复杂的类型（需要转换为 C/C++ 的数据结构才能使用），如数组，String 或自定义的类等。  
基础类型，这里就不举例子了，详细可以看 GitHub 上的源码： [AndroidTrainingDemo/JNISample](https://github.com/navyifanr/AndroidTrainingDemo/tree/master/JNISample)  
要用到的 JNI 函数：

* 获取数组长度：GetArrayLength(j{type}Array)，type 为基础类型；
* 数组转换为对应类型的指针：Get{type}ArrayElements(jarr, 0)
* 获取构造函数的 jmethodID 时，仍然是用env->GetMethodID(jclass, methodName, sign)获取，方法 name 是；
* 通过构造函数 new 一个 jobject，env->NewObject(jclass, constructorMethodID, param...)，无参构造函数 param 则为空

**4.1 数组参数的传递**

计算整型数组参数的和  
native方法定义和调用

|  |
| --- |
| public native int intArrayMethod(int[] arr);  //调用  ParamsJni paramsJni = new ParamsJni();  Log.e(TAG, "intArrayMethod: " + paramsJni.intArrayMethod(new int[]{4, 9, 10, 16})+""); |

C++

|  |
| --- |
| extern "C"  JNIEXPORT jint JNICALL  Java\_cn\_cfanr\_jnisample\_ParamsJni\_intArrayMethod(JNIEnv \*env, jobject jobj, jintArray arr\_) {  jint len = 0, sum = 0;  jint \*arr = env->GetIntArrayElements(arr\_, 0);  len = env->GetArrayLength(arr\_);  //由于一些版本不兼容，i不定义在for循环中  jint i=0;  for(; i < len; i++) {  sum += arr[i];  }  env->ReleaseIntArrayElements(arr\_, arr, 0); //释放内存  return sum;  } |

输出结果：  
MainActivity: intArrayMethod: 39

**4.2 自定义对象参数的传递**

Person 定义，native方法定义和调用

|  |
| --- |
| public class Person {  private String name;  private int age;  public Person() {  }  public Person(int age, String name) {  this.age = age;  this.name = name;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public int getAge() {  return age;  }  public void setAge(int age) {  this.age = age;  }  @Override  public String toString() {  return "Person :{ name: "+name+", age: "+age+"}";  }  }  //传递复杂对象person，再jni函数中新构造一个person传回java层输出  public native Person objectMethod(Person person);  //调用  Log.e(TAG, "objectMethod: " + paramsJni.objectMethod(new Person()).toString() + ""); |

C++:

|  |
| --- |
| extern "C"  JNIEXPORT jobject JNICALL  Java\_cn\_cfanr\_jnisample\_ParamsJni\_objectMethod(JNIEnv \*env, jobject jobj, jobject person) {  jclass clazz = env->GetObjectClass(person); //注意是用 person，不是 jobj  // jclass jclazz = env->FindClass("cn/cfanr/jnisample/model/Person;"); //或者通过反射获取  if(clazz == NULL) {  return env->NewStringUTF("cannot find class");  }  //获取方法 id  jmethodID constructorMid = env->GetMethodID(clazz, "<init>", "(ILjava/lang/String;)V");  if(constructorMid == NULL) {  return env->NewStringUTF("not find constructor method");  }  jstring name = env->NewStringUTF("cfanr");  return env->NewObject(clazz, constructorMid, 21, name);  } |

输出结果  
MainActivity: objectMethod: Person :{ name: cfanr, age: 21}

注意：**传递对象时，获取的 jclass 是获取该参数对象的 jobject 获取，而不是第二个参数（定义该 native 方法的对象）取；**

**4.3 自定义对象的集合参数的传递**

native方法定义和调用

|  |
| --- |
| public native ArrayList<Person> personArrayListMethod(ArrayList<Person> persons);  //调用  ArrayList<Person> personList = new ArrayList<>();  Person person;  for (int i = 0; i < 3; i++) {  person = new Person();  person.setName("cfanr");  person.setAge(10 + i);  personList.add(person);  }  Log.e(TAG, "调用前：java list = " + personList.toString());  Log.e(TAG, "调用后：jni list = " + paramsJni.personArrayListMethod(personList).toString()); |

C++

|  |
| --- |
| extern "C"  JNIEXPORT jobject JNICALL  Java\_cn\_cfanr\_jnisample\_ParamsJni\_personArrayListMethod(JNIEnv \*env, jobject jobj, jobject persons) {  //通过参数获取 ArrayList 对象的 class  jclass clazz = env->GetObjectClass(persons);  if(clazz == NULL) {  return env->NewStringUTF("not find class");  }  //获取 ArrayList 无参数的构造函数  jmethodID constructorMid = env->GetMethodID(clazz, "<init>", "()V");  if(constructorMid == NULL) {  return env->NewStringUTF("not find constructor method");  }  //new一个 ArrayList 对象  jobject arrayList = env->NewObject(clazz, constructorMid);  //获取 ArrayList 的 add 方法的id  jmethodID addMid = env->GetMethodID(clazz, "add", "(Ljava/lang/Object;)Z");  //获取 Person 类的 class  jclass personCls = env->FindClass("cn/cfanr/jnisample/model/Person");  //获取 Person 的构造函数的 id  jmethodID personMid = env->GetMethodID(personCls, "<init>", "(ILjava/lang/String;)V");  jint i=0;  for(; i < 3; i++) {  jstring name = env->NewStringUTF("Native");  jobject person = env->NewObject(personCls, personMid, 18 +i, name);  //添加 person 到 ArrayList  env->CallBooleanMethod(arrayList, addMid, person);  }  return arrayList;  } |

输出结果：

|  |
| --- |
| MainActivity: 调用前：java list = [Person :{ name: cfanr, age: 10}, Person :{ name: cfanr, age: 11}, Person :{ name: cfanr, age: 12}]  MainActivity: 调用后：jni list = [Person :{ name: Native, age: 18}, Person :{ name: Native, age: 19}, Person :{ name: Native, age: 20}] |

复杂的集合参数也是需要通过获取集合的 class 和对应的方法来调用实现的

**5. JNI 函数的字符串处理**

* 访问字符串函数

其中 isCopy 是取值为JNI\_TRUE和JNI\_FALSE（或者1，0），值为JNI\_TRUE，表示返回JVM内部源字符串的一份拷贝，并为新产生的字符串分配内存空间。如果值为JNI\_FALSE，表示返回JVM内部源字符串的指针，意味着可以通过指针修改源字符串的内容，不推荐这么做，因为这样做就打破了Java字符串不能修改的规定；Java默认使用Unicode编码，而C/C++默认使用UTF编码，所以在本地代码中操作字符串的时候，必须使用合适的JNI函数把jstring转换成C风格的字符串

* + UTF-8字符：const char\* GetStringUTFChars(jstring string, jboolean\* isCopy)
  + Unicode字符：const jchar\* GetStringChars(jstring string, jboolean\* isCopy)
* 释放字符串内存
  + UTF-8字符： void ReleaseStringUTFChars(jstring string, const char\* utf)
  + Unicode 字符： void ReleaseStringChars(jstring string, const jchar\* chars)
* 创建 String 对象，UTF-8: NewStringUTF，Unicode: NewString
* 取char\*的长度，UTF-8: GetStringUTFLength，Unicode: GetStringLength
* GetStringRegion和GetStringUTFRegion：分别表示获取Unicode和UTF-8编码字符串指定范围内的内容。这对函数会把源字符串复制到一个预先分配的缓冲区内。GetStringUTFRegion与 GetStringUTFChars 比较相似，不同的是，GetStringUTFRegion 内部不分配内存，不会抛出内存溢出异常。

代码示例就不写了，其他详细可参考：  
[JNI开发之旅（9）JNI函数字符串处理 - 猫的阁楼 - CSDN博客](http://blog.csdn.net/honjane/article/details/53965966)  
[JNI/NDK开发指南（四）——字符串处理 - 技术改变生活- CSDN博客](http://blog.csdn.net/xyang81/article/details/42066665)

**6. 动态注册 JNI**

学了上面的练习，发现静态注册的方式还是挺麻烦的，生成的 JNI 函数名太长，文件、类名、变量或方法重构时，需要重新修改头文件或 C/C++ 内容代码（而且还是各个函数都要修改，没有一个统一的地方），动态注册 JNI 的方法就可以解决这个问题。

由上篇回顾下，[Android NDK开发：JNI基础篇 | cfanr](http://cfanr.cn/2017/07/29/Android-NDK-dev-JNI-s-foundation/)  
**动态注册 JNI 的原理：直接告诉 native 方法其在JNI 中对应函数的指针**。通过使用 JNINativeMethod 结构来保存 Java native 方法和 JNI 函数关联关系，步骤：

* 先编写 Java 的 native 方法；
* 编写 JNI 函数的实现（函数名可以随便命名）；
* 利用结构体 JNINativeMethod 保存Java native方法和 JNI函数的对应关系；
* 利用registerNatives(JNIEnv\* env)注册类的所有本地方法；
* 在 JNI\_OnLoad 方法中调用注册方法；
* 在Java中通过System.loadLibrary加载完JNI动态库之后，会自动调用JNI\_OnLoad函数，完成动态注册；

代码实例：  
native 方法和调用：

|  |
| --- |
| public class DynamicRegisterJni {  public native String getStringFromCpp();  }  //调用  String hello = new DynamicRegisterJni().getStringFromCpp();  Log.e(TAG, hello); |

C++动态注册 JNI 代码：

|  |
| --- |
| #include <jni.h>  #include "android/log.h"  #include <stdio.h>  #include <string>  #ifndef LOG\_TAG  #define LOG\_TAG "JNI\_LOG" //Log 的 tag 名字  //定义各种类型 Log 的函数别名  #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\_\_)  #endif  #ifdef \_\_cplusplus  extern "C" {  #endif  //定义类名  static const char \*className = "cn/cfanr/jnisample/DynamicRegisterJni";  //定义对应Java native方法的 C++ 函数，函数名可以随意命名  static jstring sayHello(JNIEnv \*env, jobject) {  LOGI("hello, this is native log.");  const char\* hello = "Hello from C++.";  return env->NewStringUTF(hello);  }  /\*  \* 定义函数映射表（是一个数组，可以同时定义多个函数的映射）  \* 参数1：Java 方法名  \* 参数2：方法描述符，也就是签名  \* 参数3：C++定义对应 Java native方法的函数名  \*/  static JNINativeMethod jni\_Methods\_table[] = {  {"getStringFromCpp", "()Ljava/lang/String;", (void \*) sayHello},  };  //根据函数映射表注册函数  static int registerNativeMethods(JNIEnv \*env, const char \*className,  const JNINativeMethod \*gMethods, int numMethods) {  jclass clazz;  LOGI("Registering %s natives\n", className);  clazz = (env)->FindClass(className);  if (clazz == NULL) {  LOGE("Native registration unable to find class '%s'\n", className);  return JNI\_ERR;  }  if ((env)->RegisterNatives(clazz, gMethods, numMethods) < 0) {  LOGE("Register natives failed for '%s'\n", className);  return JNI\_ERR;  }  //删除本地引用  (env)->DeleteLocalRef(clazz);  return JNI\_OK;  }  jint JNI\_OnLoad(JavaVM \*vm, void \*reserved) {  LOGI("call JNI\_OnLoad");  JNIEnv \*env = NULL;  if (vm->GetEnv((void \*\*) &env, JNI\_VERSION\_1\_4) != JNI\_OK) { //判断 JNI 版本是否为JNI\_VERSION\_1\_4  return JNI\_EVERSION;  }  registerNativeMethods(env, className, jni\_Methods\_table, sizeof(jni\_Methods\_table) / sizeof(JNINativeMethod));  return JNI\_VERSION\_1\_4;  }  #ifdef \_\_cplusplus  }  #endif |

输出结果：

|  |
| --- |
| JNI\_LOG: call JNI\_OnLoad  JNI\_LOG: Registering cn/cfanr/jnisample/DynamicRegisterJni natives  JNI\_LOG: hello, this is native log.  MainActivity: Hello from C++. |

上面代码涉及到 **JNI 调用 Android 的 Log**，只需要引入#include "android/log.h"头文件和对函数别名命名即可。其他具体说明见上面代码。

实际开发中可以**采取动态和静态注册结合的方式**，写一个Java 的 native 方法完成调用动态注册的代码，大概代码如下：

|  |
| --- |
| static {  System.loadLibrary("native-lib");  registerNatives();  }  private static native void registerNatives(); |

C++

|  |
| --- |
| JNIEXPORT void JNICALL Java\_com\_zhixin\_jni\_JniSample\_registerNatives  (JNIEnv \*env, jclass clazz){  (env)->RegisterNatives(clazz, gJni\_Methods\_table, sizeof(gJni\_Methods\_table) / sizeof(JNINativeMethod));  } |

**7. 小结**

虽然都是按照网上的例子做的练习记录，但还是遇到不少小问题的，不过只要仔细查找，也比较容易发现问题的所在，以前觉得 JNI 挺难懂的，但这次练习下来，觉得 JNI 也只不过是一套语法规则而已，按照规则去实现代码也不算特别难，**当然这只是 JNI 的一小部分内容，JNI 还有很多内容，如反射、异常处理、多线程、NIO 等**。虽然这次练习比较简单，但建议还是自己亲自敲一遍代码，在练习中发现问题，并解决，以后遇到同类型的问题也比较容易解决。

注意一些报错的问题：

* 如果没加 extern “C” 或者 没将 C/C++ 文件配置到 CMake 文件上，可能会报java.lang.UnsatisfiedLinkError: Native method not found: xxx错误
* 一般报java.lang.NoSuchMethodError: no method with xxx错误，可能是因为 class 和方法不对应，env->GetObjectClass( jobject jobj)这里用错了对象
* 报java.lang.NoClassDefFoundError，可能是类名写错找不到类；

本文完整代码可以到 GitHub 查看源码： [AndroidTrainingDemo/JNISample](https://github.com/navyifanr/AndroidTrainingDemo/tree/master/JNISample)

参考资料：  
[专栏：JNI开发之旅 -猫的阁楼- CSDN博客](http://blog.csdn.net/column/details/honjane.html)  
[Andoid NDK编程1- 动态注册native函数 // Coding Life](http://zhixinliu.com/2015/07/01/2015-07-01-jni-register/)  
[Android Stuido Ndk-Jni 开发：Jni中打印log信息 - 简书](http://www.jianshu.com/p/acbf724fdcc9)