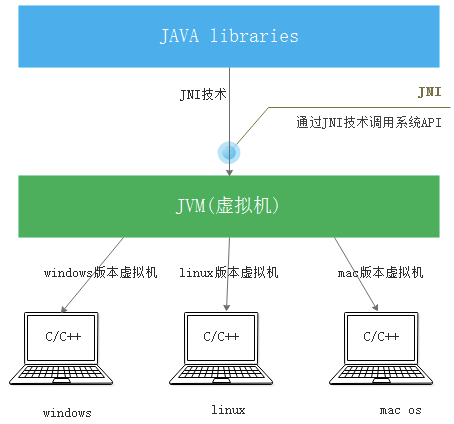
Android NDK开发：JNI基础篇

## 1. JNI概念

### 1.1 概念

JNI全称Java Native Interface，Java本地化接口，可以通过JNI调用系统提供的API。操作系统，无论是Linux，Windows还是Mac OS，或者一些汇编语言写的底层硬件驱动都是C/C++写的。**Java和C/C++不同，它不会直接编译成平台机器码，而是编译成虚拟机可以运行的Java字节码的.class文件**，通过JIT技术即时编译成本地机器码，所以有效率就比不上C/C++代码，JNI技术就解决了这一痛点，**JNI可以说是C语言和Java语言交流的适配器、中间件**，下面我们来看看JNI调用示意图：来自[JNI开发系列①JNI概念及开发流程 - 简书](https://www.jianshu.com/p/68bca86a84ce)



JNI调用示意图

JNI技术通过JVM调用到各个平台的API，虽然JNI可以调用C/C++，但是JNI调用还是比C/C++编写的原生应用还是要慢一点，不过对高性能计算来说，这点算不得什么，享受它的便利，也要承担它的弊端。

### 1.2 JNI与NDK区别

* JNI：JNI是一套编程接口，用来实现Java代码与本地的C/C++代码进行交互；
* NDK：NDK是Google开发的一套开发和编译工具集，可以生成动态链接库，主要用于Android的JNI开发；

## 2. JNI作用

* **扩展：**JNI扩展了JVM能力，驱动开发，例如开发一个wifi驱动，可以将手机设置为无限路由；
* **高效：**本地代码效率高，游戏渲染，音频视频处理等方面使用JNI调用本地代码，C语言可以灵活操作内存；
* **复用：**在文件压缩算法7zip开源代码库，机器视觉OpenCV开放算法库等方面可以复用C平台上的代码，不必在开发一套完整的Java体系，避免重复发明轮子；
* **特殊：**产品的核心技术一般也采用JNI开发，不易破解；

JNI在Android中作用：

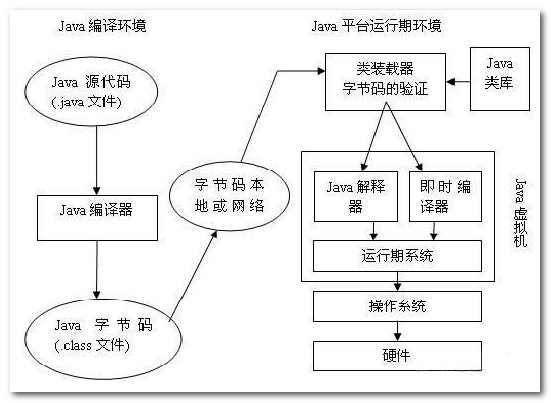
JNI可以调用本地代码库(即C/C++代码)，并通过Dalvik虚拟机与应用层和应用框架层进行交互，Android中JNI代码主要位于应用层和应用框架层；

* 应用层:该层是由JNI开发，主要使用标准JNI编程模型；
* 应用框架层:使用的是Android中自定义的一套JNI编程模型，该自定义的JNI编程模型弥补了标准JNI编程模型的不足；

补充知识点：

Java语言执行流程：

* 编译字节码：Java编译器编译.java源文件，获得.class字节码文件；
* 装载类库：使用类装载器装载平台上的Java类库，并进行字节码验证；
* Java虚拟机：将字节码加入到JVM中，Java解释器和即时编译器同时处理字节码文件，将处理后的结果放入运行时系统；
* 调用JVM所在平台类库：JVM处理字节码后，转换成相应平台的操作，调用本平台底层类库进行相关处理；



Java语言执行流程

**Java一次编译到处执行**：JVM在不同的操作系统都有实现，Java可以一次编译到处运行，字节码文件一旦编译好了，可以放在任何平台的虚拟机上运行；

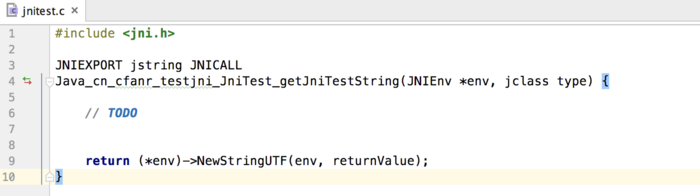
## 3. 查看jni.h文件源码方法

jni.h头文件就是为了让C/C++类型和Java原始类型相匹配的头文件定义。

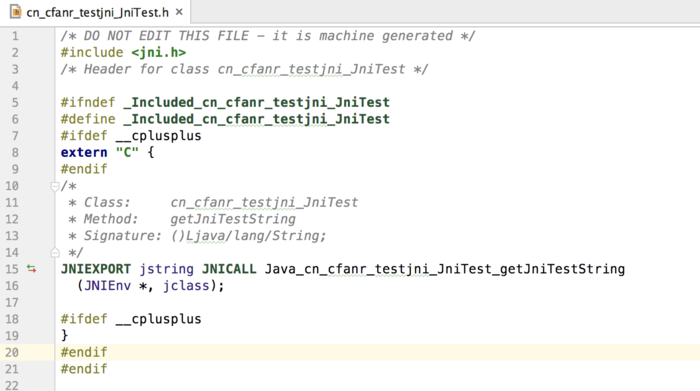
可以通过点击Android项目的含有#include<jni.h>的头文件或C/C++文件跳转到jni.h头文件查看；

如果没有这样的文件的话，可以在Android Studio上新建一个类，随便写一个native方法，然后点击红色的方法，AS会自动生成一个对应的C语言文件jnitest.c，就可以找到jni.h文件了





或者，通过javah命令javah cn.cfanr.testjni.JniTest，就可以生成对应头文件cn\_cfanr\_testjni\_JniTest.h：



javah生成的

## 4. JNI数据类型映射

由头文件代码可以看到，jni.h有很多类型预编译的定义，并且区分了C和C++的不同环境。

#ifdef HAVE\_INTTYPES\_H

# include <inttypes.h> /\* C99 \*/

typedef uint8\_t jboolean; /\* unsigned 8 bits \*/

typedef int8\_t jbyte; /\* signed 8 bits \*/

typedef uint16\_t jchar; /\* unsigned 16 bits \*/

typedef int16\_t jshort; /\* signed 16 bits \*/

typedef int32\_t jint; /\* signed 32 bits \*/

typedef int64\_t jlong; /\* signed 64 bits \*/

typedef float jfloat; /\* 32-bit IEEE 754 \*/

typedef double jdouble; /\* 64-bit IEEE 754 \*/

#else

typedef unsigned char jboolean; /\* unsigned 8 bits \*/

typedef signed char jbyte; /\* signed 8 bits \*/

typedef unsigned short jchar; /\* unsigned 16 bits \*/

typedef short jshort; /\* signed 16 bits \*/

typedef int jint; /\* signed 32 bits \*/

typedef long long jlong; /\* signed 64 bits \*/

typedef float jfloat; /\* 32-bit IEEE 754 \*/

typedef double jdouble; /\* 64-bit IEEE 754 \*/

#endif

/\* "cardinal indices and sizes" \*/

typedef jint jsize;

#ifdef \_\_cplusplus

/\*

\* Reference types, in C++

\*/

class \_jobject {};

class \_jclass : public \_jobject {};

class \_jstring : public \_jobject {};

class \_jarray : public \_jobject {};

class \_jobjectArray : public \_jarray {};

class \_jbooleanArray : public \_jarray {};

//……

typedef \_jobject\* jobject;

typedef \_jclass\* jclass;

typedef \_jstring\* jstring;

typedef \_jarray\* jarray;

typedef \_jobjectArray\* jobjectArray;

typedef \_jbooleanArray\* jbooleanArray;

//……

#else /\* not \_\_cplusplus \*/

/\*

\* Reference types, in C.

\*/

typedef void\* jobject;

typedef jobject jclass;

typedef jobject jstring;

typedef jobject jarray;

typedef jarray jobjectArray;

typedef jarray jbooleanArray;

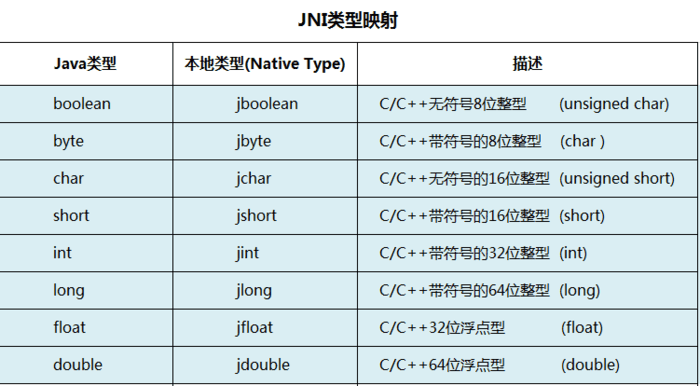
//……

#endif

当是C++环境时，jobject、jclass、jstring、jarray等都是继承自\_jobject类，而在C语言环境是，则它的本质都是空类型指针typedef void\* jobject;

### 4.1 基本数据类型

下图是Java基本数据类型和本地类型的映射关系，这些**基本数据类型都是可以直接在Native层直接使用的**：



基本数据类型映射

### 4.2 引用数据类型

另外，还有引用数据类型和本地类型的映射关系：



引用数据类型映射

需要注意的是：

* 1）引用类型不能直接在Native层使用，需要根据JNI函数进行类型的转化后，才能使用；
* 2）多维数组（含二维数组）都是引用类型，需要使用jobjectArray类型存取其值；

例如，二维整型数组就是指向一位数组的数组，其声明使用方式如下：

//获得一维数组的类引用，即jintArray类型

jclass intArrayClass = env->FindClass("[I");

//构造一个指向jintArray类一维数组的对象数组，该对象数组初始大小为length，类型为jsize

jobjectArray obejctIntArray = env->NewObjectArray(length, intArrayClass, NULL);

### 4.3 方法和变量ID

同样不能直接在Native层使用。当Native层需要调用Java的某个方法时，需要通过JNI函数获取它的ID，根据ID调用JNI函数获取该方法；变量的获取也是类似。ID的结构体如下：

struct \_jfieldID; /\* opaque structure \*/

typedef struct \_jfieldID\* jfieldID; /\* field IDs \*/

struct \_jmethodID; /\* opaque structure \*/

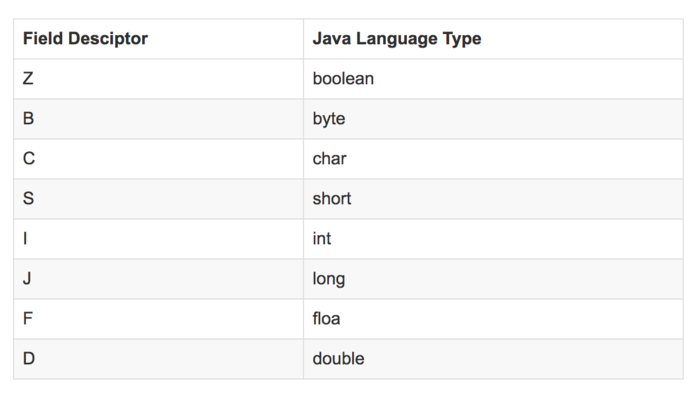
typedef struct \_jmethodID\* jmethodID; /\* method IDs \*/

## 5. JNI描述符

### 5.1域描述符

1）基本类型描述符

下面是基本的数据类型的描述符，除了boolean和long类型分别是Z和J外，其他的描述符对应的都是Java类型名的大写首字母。另外，void的描述符为V。



基本类型描述符

2）引用类型描述符

一般引用类型描述符的规则如下，**注意不要丢掉“;”**

L + 类描述符 + ;

如，String类型的域描述符为：

Ljava/lang/String;

数组的域描述符特殊一点，如下，其中有多少级数组就有多少个“[”，数组的类型为类时，则有分号，为基本类型时没有分号

[ + 其类型的域描述符

例如：

int[] 描述符为：[I

double[] 描述符为：[D

String[] 描述符为：[Ljava/lang/String;

Object[] 描述符为：[Ljava/lang/Object;

int[][] 描述符为：[[I

double[][] 描述符为：[[D

对应在jni.h获取Java的字段的native函数如下，name为Java的字段名字，sig为域描述符

//C

jfieldID (\*GetFieldID)(JNIEnv\*, jclass, const char\*, const char\*);

jobject (\*GetObjectField)(JNIEnv\*, jobject, jfieldID);

//C++

jfieldID GetFieldID(jclass clazz, const char\* name, const char\* sig)

{ return functions->GetFieldID(this, clazz, name, sig); }

jobject GetObjectField(jobject obj, jfieldID fieldID)

{ return functions->GetObjectField(this, obj, fieldID); }

具体使用，后面会讲到。

### 5.2 类描述符

类描述符是类的完整名称：包名+类名，java中包名用 . 分割，jni中改为用 / 分割。

如，Java中java.lang.String类的描述符为：java/lang/String

native层获取Java的类对象，需要通过FindClass()函数获取，jni.h的函数定义如下：

//C

jclass (\*FindClass)(JNIEnv\*, const char\*);

//C++

jclass FindClass(const char\* name)

{ return functions->FindClass(this, name); }

字符串参数就是类的引用类型描述符，如Java对象cn.cfanr.jni.JniTest，对应字符串为Lcn/cfanr/jni/JniTest;如下：

jclass jclazz = env->FindClass(“Lcn/cfanr/jni/JniTest;”);

详细用法的例子，后面会讲到。

### 5.3 方法描述符

方法描述符需要将所有参数类型的域描述符按照声明顺序放入括号，然后再加上返回值类型的域描述符，其中**没有参数时，不需要括号，**如下规则：

(参数……)返回类型

例如：

Java层方法——>JNI函数签名

String getString()——>Ljava/lang/String;

int sum(int a, int b)——>(II)I

void main(String[] args) ——> ([Ljava/lang/String;]V

另外，对应在jni.h获取Java方法的native函数如下，其中jclass是获取到的类对象，name是Java对应的方法名字，sig就是上面说的方法描述符

//C

jmethodID (\*GetMethodID)(JNIEnv\*, jclass, const char\*, const char\*);

//C++

jmethodID GetMethodID(jclass clazz, const char\* name, const char\* sig)

{ return functions->GetMethodID(this, clazz, name, sig); }

不过在实际编程中，如果使用javah工具来生成对应的native代码，就不需要手动编写对应的类型转换了。(???)

## 6. JNIEnv分析

JNIEnv是jni.h文件最重要的部分，它的**本质是指向函数表指针的指针（JavaVM也是）**，函数表里面定义了很多JNI函数，同时它也是区分C和C++环境的（由上面介绍描述符时也可以看到），在C语言环境中，JNIEnv是strut JNINativeInterface\*的指针别名。

struct \_JNIEnv;

struct \_JavaVM;

typedef const struct JNINativeInterface\* C\_JNIEnv;

#if defined(\_\_cplusplus)

typedef \_JNIEnv JNIEnv; //C++中的JNIEnv类型

typedef \_JavaVM JavaVM;

#else

typedef const struct JNINativeInterface\* JNIEnv; //C语言的JNIEnv类型

typedef const struct JNIInvokeInterface\* JavaVM;

#endif

### 6.1 JNIEnv特点

* JNIEnv是一个指针，指向一组JNI函数，通过这些函数可以实现Java层和JNI层的交互，就是说**通过JNIEnv调用JNI函数可以访问Java虚拟机，操作Java对象；**
* **所有本地函数都会接收JNIEnv作为第一个参数；**（不过C++的JNI函数已经对JNIEnv参数进行了封装，不用写在函数参数上）
* 用作线程局部存储，不能在线程间共享一个JNIEnv变量，也就是说JNIEnv只在创建它的线程有效，不能跨线程传递；相同的Java线程调用本地方法，所使用的JNIEnv是相同的，一个native方法不能被不同的Java线程调用；

### 6.2 JavaEnv和JavaVM的关系

[JNIEnv和Dalvik的JavaVM的关系-CSDN](https://link.jianshu.com?t=http%3A%2F%2Fblog.csdn.net%2Fdtryl%2Farticle%2Fdetails%2F50682599)

* 1）每个进程只有一个JavaVM（理论上一个进程可以拥有多个JavaVM对象，但Android只允许一个），每个线程都会有一个JNIEnv，大部分JNIAPI通过JNIEnv调用；也就是说，JNI全局只有一个JavaVM，而可能有多个JNIEnv；
* 2）一个JNIEnv内部包含一个Pointer，Pointer指向Dalvik的JavaVM对象的Function Table，JNIEnv内部的函数执行环境来源于Dalvik虚拟机；
* 3）Android中每当一个Java线程第一次要调用本地C/C++代码时，Dalvik虚拟机实例会为该Java线程产生一个JNIEnv指针；
* 4）Java每条线程在和C/C++互相调用时，JNIEnv是互相独立，互不干扰的，这样就提升了并发执行时的安全性；
* 5）当本地的C/C++代码想要获得当前线程所想要使用的JNIEnv时，可以使用Dalvik VM对象的JavaVM\* jvm->GetEnv()方法，该方法会返回当前线程所在的JNIEnv\*；
* 6）Java的dex字节码和C/C++的.so同时运行Dalvik VM之内，共同使用一个进程空间；

### 6.3 C语言的JNIEnv

由上面代码可知，C语言的JNIEnv就是const struct JNINativeInterface\*，而JNIEnv\* env就等价于JNINativeInterface\*\* env，env实际是一个二级指针，所以想要得到JNINativeInterface结构体中定义的函数指针，就需要先获取JNINativeInterface的一级指针对象\*env，然后才能通过一级指针对象调用JNI函数，例如：

(\*env)->NewStringUTF(env, "hello")

struct JNINativeInterface {

void\* reserved0;

void\* reserved1;

void\* reserved2;

void\* reserved3;

jint (\*GetVersion)(JNIEnv \*);

jclass (\*DefineClass)(JNIEnv\*, const char\*, jobject, const jbyte\*, jsize);

jclass (\*FindClass)(JNIEnv\*, const char\*);

jmethodID (\*FromReflectedMethod)(JNIEnv\*, jobject);

jfieldID (\*FromReflectedField)(JNIEnv\*, jobject);

/\* spec doesn't show jboolean parameter \*/

jobject (\*ToReflectedMethod)(JNIEnv\*, jclass, jmethodID, jboolean);

jclass (\*GetSuperclass)(JNIEnv\*, jclass);

jboolean (\*IsAssignableFrom)(JNIEnv\*, jclass, jclass);

/\* spec doesn't show jboolean parameter \*/

jobject (\*ToReflectedField)(JNIEnv\*, jclass, jfieldID, jboolean);

//……定义了一系列关于Java操作的函数

}

### 6.4 C++的JNIEnv

由typedef \_JNIEnv JNIEnv;可知，C++的JNIEnv是\_JNIEnv结构体，而\_JNIEnv结构体定义了JNINativeInterface的结构体指针，内部定义的函数实际上是调用JNINativeInterface的函数，所以C++的env是一级指针，调用时不需要加env作为函数的参数，例如：env->NewStringUTF(env, "hello")

struct \_JNIEnv {

/\* do not rename this; it does not seem to be entirely opaque \*/

const struct JNINativeInterface\* functions;

#if defined(\_\_cplusplus)

jint GetVersion()

{ return functions->GetVersion(this); }

jclass DefineClass(const char \*name, jobject loader, const jbyte\* buf, jsize bufLen)

{ return functions->DefineClass(this, name, loader, buf, bufLen); }

jclass FindClass(const char\* name)

{ return functions->FindClass(this, name); }

jmethodID FromReflectedMethod(jobject method)

{ return functions->FromReflectedMethod(this, method); }

jfieldID FromReflectedField(jobject field)

{ return functions->FromReflectedField(this, field); }

jobject ToReflectedMethod(jclass cls, jmethodID methodID, jboolean isStatic)

{ return functions->ToReflectedMethod(this, cls, methodID, isStatic); }

jclass GetSuperclass(jclass clazz)

{ return functions->GetSuperclass(this, clazz); }

//……

}

## 7. JNI的两种注册方式

Java的native方法是如何链接C/C++中的函数的呢？可以通过静态和动态的方式注册JNI。

### 7.1静态注册

原理：根据函数名建立Java方法和JNI函数的一一对应关系。流程如下：

* 先编写Java的native方法；
* 然后用javah工具生成对应的头文件，执行命令javah packagename.classname可以生成由包名加类名命名的jni层头文件，或执行命名javah -o custom.h packagename.classname，其中custom.h为自定义的文件名；
* 实现JNI里面的函数，再在Java中通过System.loadLibrary加载so库即可；

静态注册的方式有两个重要的关键词JNIEXPORT和JNICALL，这两个关键词是宏定义，主要是注明该函数式JNI函数，当虚拟机加载so库时，如果发现函数含有这两个宏定义时，就会链接到对应的Java层的native方法。

由前面《3. 查看jni.h文件源码方法》生成头文件的方法，重新创建一个cn.cfanr.test\_jni.Jni\_Test.java的类

public class Jni\_Test {

private static native int swap();

private static native void swap(int a, int b);

private static native void swap(String a, String b);

private native void swap(int[] arr, int a, int b);

private static native void swap\_0(int a, int b);

}

用javah工具生成以下头文件：

#include <jni.h>

/\* Header for class cn\_cfanr\_test\_jni\_Jni\_Test \*/

#ifndef \_Included\_cn\_cfanr\_test\_jni\_Jni\_Test

#define \_Included\_cn\_cfanr\_test\_jni\_Jni\_Test

#ifdef \_\_cplusplus

extern "C" {

#endif

/\*

\* Class: cn\_cfanr\_test\_jni\_Jni\_Test

\* Method: swap

\* Signature: ()I

\*/

JNIEXPORT jint JNICALL Java\_cn\_cfanr\_test\_1jni\_Jni\_1Test\_swap\_\_

(JNIEnv \*, jclass); // 凡是重载的方法，方法后面都会多一个下划线

/\*

\* Class: cn\_cfanr\_test\_jni\_Jni\_Test

\* Method: swap

\* Signature: (II)V

\*/

JNIEXPORT void JNICALL

Java\_cn\_cfanr\_test\_1jni\_Jni\_1Test\_swap\_\_II (JNIEnv \*, jclass, jint, jint);

/\*

\* Class: cn\_cfanr\_test\_jni\_Jni\_Test

\* Method: swap

\* Signature: (Ljava/lang/String;Ljava/lang/String;)V

\*/

JNIEXPORT void JNICALL

Java\_cn\_cfanr\_test\_1jni\_Jni\_1Test\_swap\_\_Ljava\_lang\_String\_2Ljava\_lang\_String\_2

(JNIEnv \*, jclass, jstring, jstring);

/\*

\* Class: cn\_cfanr\_test\_jni\_Jni\_Test

\* Method: swap

\* Signature: ([III)V

\*/

JNIEXPORT void JNICALL Java\_cn\_cfanr\_test\_1jni\_Jni\_1Test\_swap\_\_\_3III

(JNIEnv \*, jobject, jintArray, jint, jint); // 非 static 的为 jobject

/\*

\* Class: cn\_cfanr\_test\_jni\_Jni\_Test

\* Method: swap\_0

\* Signature: (II)V

\*/

JNIEXPORT void JNICALL

Java\_cn\_cfanr\_test\_1jni\_Jni\_1Test\_swap\_10

(JNIEnv \*, jclass, jint, jint); // 不知道为什么后面没有 II

#ifdef \_\_cplusplus

}

#endif

#endif

可以看出JNI的调用函数的定义是按照一定规则命名的：

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Java\_是为了标识该函数来源于Java。**经检验（不一定正确），如果是重载的方法，则有“参数签名”，否则没有；另外如果使用的是C++，在函数前面加上extern “C”（表示按照C的方式编译），函数命名后面就不需要加上“参数签名”。**

**另外还需要注意几点特殊规则：**（参考：[官方JNI规范翻译 | linlinjava的博客](https://link.jianshu.com?t=http%3A%2F%2Flinlinjava.org%2F2015%2F04%2F01%2Flearning-jni.html) 2.2.1本地方法名解析）

* **1.** 包名或类名或方法名中含下划线\_要用\_1连接；
* **2.** 重载的本地方法命名要用双下划线\_\_连接；
* **3. 参数签名的斜杠“/”改为下划线“\_”连接，分号“;”改为“\_2”连接，左方括号“[”改为“\_3”连接；**  
  另外，对于Java的native方法，static和非static方法的区别在于第二个参数，static的为jclass，非static的为jobject；JNI函数中是没有修饰符的。

优点：

实现比较简单，可以通过javah工具将Java代码的native方法直接转化为对应的native层代码的函数；

缺点：

* javah生成的native层函数名特别长，可读性很差；
* 后期修改文件名、类名或函数名时，头文件的函数将失效，需要重新生成或手动改，比较麻烦；
* 程序运行效率低，首次调用native函数时，需要根据函数名在JNI层搜索对应的本地函数，建立对应关系，有点耗时；

### 7.2 动态注册

**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，会调用JNI\_OnLoad函数，完成动态注册；

//JNINativeMethod结构体

typedef struct {

const char\* name; //Java中native方法的名字

const char\* signature; //Java中native方法的描述符

void\* fnPtr; //对应JNI函数的指针

} JNINativeMethod;

/\*\*

\* @param clazz java类名，通过FindClass获取

\* @param methods JNINativeMethod结构体指针

\* @param nMethods方法个数

\*/

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

//JNI\_OnLoad

JNIEXPORT jint JNICALL JNI\_OnLoad(JavaVM\* vm, void\* reserved);

由于篇幅原因，具体的静态注册、动态注册、数据类型映射和描述符的练习放到下一篇文章：[Android NDK开发：JNI实战篇](https://www.jianshu.com/p/464cd879eaba)。

注：文章通过阅读JNI的文档和参照网上的博客总结出来的，如有错误，还望指出！

参考：  
[JNI开发系列①JNI概念及开发流程 - 简书](https://www.jianshu.com/p/68bca86a84ce)  
[JNI 数据类型映射、域描述符说明 - qinjuning - CSDN博客](https://link.jianshu.com?t=http%3A%2F%2Fblog.csdn.net%2Fqinjuning%2Farticle%2Fdetails%2F7599796)  
[Android JNI 之 JNIEnv 解析 - 韩曙亮 - CSDN博客](https://link.jianshu.com?t=http%3A%2F%2Fblog.csdn.net%2Fshulianghan%2Farticle%2Fdetails%2F38012515)  
[Android 开发 之 JNI入门 - NDK从入门到精通 -韩曙亮 - CSDN博客](https://link.jianshu.com?t=http%3A%2F%2Fblog.csdn.net%2Fshulianghan%2Farticle%2Fdetails%2F18964835)  
[JNI 两种注册过程实战 - Android - 掘金](https://link.jianshu.com?t=https%3A%2F%2Fjuejin.im%2Fentry%2F5885b019128fe1006c3f0149)  
[Andoid NDK编程 注册native函数 // Coding Life](https://link.jianshu.com?t=http%3A%2F%2Fzhixinliu.com%2F2015%2F07%2F01%2F2015-07-01-jni-register%2F)

扩展阅读：  
[JNI 使用指南-胡凯](https://link.jianshu.com?t=http%3A%2F%2Fhukai.me%2Fandroid-training-course-in-chinese%2Fperformance%2Fperf-jni%2Findex.html)  
[JNI 常用函数大全 qinjuning- CSDN博客](https://link.jianshu.com?t=http%3A%2F%2Fblog.csdn.net%2Fqinjuning%2Farticle%2Fdetails%2F7595104)  
[Android JNI原理分析 - Gityuan博客 | 袁辉辉博客](https://link.jianshu.com?t=http%3A%2F%2Fgityuan.com%2F2016%2F05%2F28%2Fandroid-jni%2F)  
[JNI API 文档 (Java8): Java Native Interface Specification](https://link.jianshu.com?t=http%3A%2F%2Fdocs.oracle.com%2Fjavase%2F8%2Fdocs%2Ftechnotes%2Fguides%2Fjni%2Fspec%2FjniTOC.html)  
[官方JNI API 规范翻译 | linlinjava的博客](https://link.jianshu.com?t=http%3A%2F%2Flinlinjava.org%2F2015%2F04%2F01%2Flearning-jni.html)