Losileeya

编程中我们会遇到多少挫折？不放弃，沙漠尽头必是绿洲，不是天方夜谭

个人资料

u013278099

关注 发私信

访问： 422763次

积分： 5205

等级： BLOG 6

安卓实战开发之JNI从小白到伪老白深入了解JNI动态注册native方法及JNI数据使用

标签： jni

2016-07-23 19:06 3575人阅读 评论(1) 收藏 举报

分类：

android (82)

前言

或许你知道了jni的简单调用，其实不算什么百度谷歌一大把，虽然这些jni绝大多数情况下都不会让我们安卓工程师来弄，毕竟还是有点难，但是我们还是得打破砂锅知道为什么这样干吧，至少也让我们知道调用流程和数据类型以及处理方法，或许你会有不一样的发现。

排名： 第5972名

原创： 96篇 转载： 20篇

译文： 1篇 评论： 275条

博客专栏

安卓实战开发

文章：12篇

阅读：78174

Kotlin从语法到安卓开发

文章：5篇

阅读：11561

安卓使用第三方库开速开发

文章：7篇

阅读：51673

安卓特效开发

文章：15篇

阅读：71682

文章分类

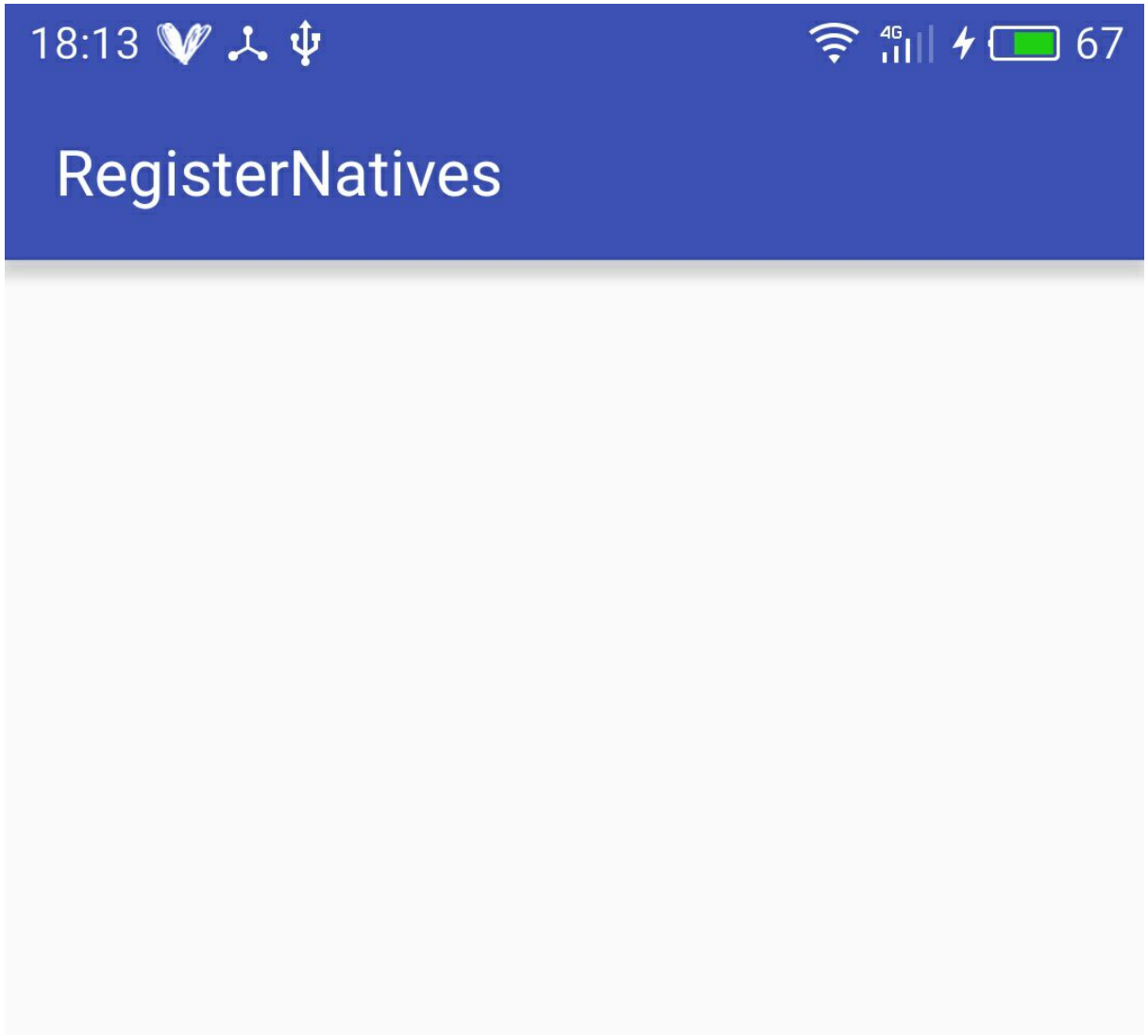
android (83)

ios (2)

html5 (2)

> 其实总的来说从java的角度来看.h文件就是java中的interface(插座)，然后.c/.cpp文件呢就是实现类罢了，然后数据类型和java还是有点出入我们还是得了解下（妈蛋，天气真热不适合生存了）。

今天也给出一个JNI动态注册native方法的例子，如图：



Kotlin (5)
AngularJs (0)
React-native (2)
java (11)
开发环境 (13)
Python(Ubuntu) (3)
微信小程序 (1)

JNI实现步骤

JNI 开发流程主要分为以下步骤：

- 编写声明了 native 方法的 Java 类

- 将 Java 源代码编译成 class 字节码文件
- 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）
- 用本地代码（c/c++）实现.h头文件中的函数
- 将(c/c++)文件编译成动态库（Windows：*.dll，linux/unix：*.so，mac os x：*.jnilib）
- 拷贝动态库至本地库目录下，并运行 Java 程序（System.loadLibrary(“xxx”)）

我们安卓开发工程师显然只需要编写native的java类，然后clean下编译器知道把我们的java编译成了class文件，但是我们必须知道是调用了javac命令，javah jni命令我们还是得执行，其他的工作就差不多了，不管是什么编译器，反正jni步骤就这样。

JVM 查找 native 方法

JVM 查找 native 方法有两种方式：

- 按照 JNI 规范的命名规则
- 调用 JNI 提供的 RegisterNatives 函数，将本地函数注册到 JVM 中。

是不是感到特别的意外，jni还能够利用RegisterNatives 函数查找native方法，其实我也才刚刚知道有这方法，因为要根据包名类名方法名的规范来写是很傻逼的，哈哈，有的人或许觉得这样很直观。

严格按照命名规则实现native方法的调用

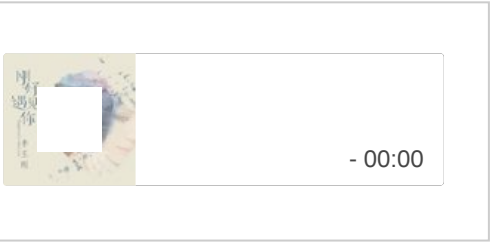
我们还是按步骤来说吧，先来解读JNI规范的命名规则：

*** 我们先来看下.h文件 ***

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class com_losileeya_jnimaster_JNIUtils */
#ifndef _Included_com_losileeya_jnimaster_JNIUtils
#define _Included_com_losileeya_jnimaster_JNIUtils
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     com_losileeya_jnimaster_JNIUtils
 * Method:    say
 * Signature: ()Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL  Java_com_losileeya_jnimaster_JNIUtils_say
      (JNIEnv *, jclass,jstring);
#ifdef __cplusplus
}
#endif
#endif
```

我们再来看下Linux 下jni_md.h头文件内容：

```
#ifndef _JAVASOFT_JNI_MD_H_
#define _JAVASOFT_JNI_MD_H_
#define JNIEXPORT
#define JNIIMPORT
#define JNICALL
```




```
6  typedef int jint;
7  #ifdef _LP64 /* 64-bit Solaris */
8  typedef long jlong;
9  #else
10 typedef long long jlong;
11 #endif
12 typedef signed char jbyte;
13 #endif
```

从上面我们可以看出文件以#ifndef开始然后#endif 结尾，不会C的话是不是看起来有点蛋疼，#号呢代表宏，这里来普及一下宏的使用和定义。

> #define 标识符 字符串
> 其中，#表示这是一条预处理命令；#define为宏定义命令；“标识符”为宏定义的宏名；“字符串”可以上常数、表达式、格式串等。

举例如下：

```
1 #define PI 3.14 // 对3.14进行宏定义，宏名为PI
2 void main()
3 {
4 printf("%f", PI); // 输出3.14
5 }
```

条件编译的命令

```
#ifndef def
语句1
# else
语句2
# endif
表示如果def在前面进行了宏定义那么就编译语句1（语句2不编译），否则编译语句2（语句1不编译）
```

再看我们.h文件并没有else，所以我们就编译宏定义的本地方法类（com_losileeya_jnimaster_JNIUtils），你突然就会发现我们的宏是我们的native类，然后把包名点类名的点改成了下划线，然后你会发现多了_Included不要多想，就是included关键字加个下划线，这样我们就给本地类进行了宏定义。然后

```
#ifdef __cplusplus
extern “C” {#endif
```

这是说明如果宏定义了c++，并且里面有c我们还是支持c的，并且c代码写extern “C” { } 里面。可以看出#endif对应上面的#ifdef-cplusplus，#ifdef-cplusplus对应最后的#endif, #ifdef与#endif总是一一对应的，表明条件编译开始和结束。

JNIEXPORT 和 JNICALL 的作用

因为安卓是跑在 Linux 下的，所以从 Linux 下的jni_md.h头文件可以看出来，JNIEXPORT 和 JNICALL 是一个空定义，所以在 Linux 下 JNI 函数声明可以省略这两个宏。

再来看我们的方法：

```
JNIEXPORT jstring JNICALL  Java_com_losileeya_jnimaster_JNIUtils_say
        (JNIEnv *, jclass,jstring);
```

函数命名规则为：Java_类全路径_方法名。
如：Java_com_losileeya_jnimaster_JNIUtils_say，其中Java_是函数的前缀，com_losileeya_jnimaster_JNIUtils是类名，say是方法名，它们之间用 _(下划线) 连接。

- 第一个参数：JNIEnv* 是定义任意 native 函数的第一个参数（包括调用 JNI 的 RegisterNatives 函数注册的函数），指向 JVM 函数表的指针，函数表中的每一个入口指向一个 JNI 函数，每个函数用于访问 JVM 中特定的数据结构。
- 第二个参数：调用 Java 中 native 方法的实例或 Class 对象，如果这个 native 方法是实例方法，则该参数是 jobject，如果是静态方法，则是 jclass。
- 第三个参数：Java 对应 JNI 中的数据类型，Java 中 String 类型对应 JNI 的 jstring 类型。（后面会详细介绍 JAVA 与 JNI 数据类型的映射关系）。

函数返回值类型：夹在 JNIEXPORT 和 JNICALL 宏中间的 jstring，表示函数的返回值类型，对应 Java 的String 类型。

> 如果你需要装逼的话你就可以自己去写.h文件，然后就可以抛弃javah -jni 命令，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（逼就是这么装出来的）

RegisterNatives动态获取本地方法

是不是感觉一个方法的名字太长非常的蛋疼，然后我们呢直接使用，RegisterNatives来自己命名调用native方法，这样是不是感觉好多了。

要实现呢，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（），这样就达到了动态注册实现native方法的作用。

```
/*
* System.loadLibrary("lib")时调用
* 如果成功返回JNI版本, 失败返回-1
*/
JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM* vm, void* reserved) {
    JNIEnv* env = NULL;
    jint result = -1;
    if ((*vm)->GetEnv(vm, (void**) &env, JNI_VERSION_1_4) != JNI_OK) {
        return -1;
    }
    assert(env != NULL);
    if (!registerNatives(env)) {//注册
        return -1;
    }
    //成功
    result = JNI_VERSION_1_4;
    return result;
}
```

并且我们需要为类注册本地方法，那样就能方便我们去调用，不多说看方法：

```
/*
* 为所有类注册本地方法
*/
static int registerNatives(JNIEnv* env) {
    return registerNativeMethods(env, JNIREG_CLASS, gMethods,sizeof(gMethods) / sizeof(gMethods[0]));
}
```

也可以为某一个类注册本地方法

```
/*
* 为某一个类注册本地方法
*/
static int registerNativeMethods(JNIEnv* env
        , const char* className
        , JNINativeMethod* gMethods, int numMethods) {
    jclass clazz;
    clazz = (*env)->FindClass(env, className);
    if (clazz == NULL) {
        return JNI_FALSE;
    }
    if ((*env)->RegisterNatives(env, clazz, gMethods, numMethods) < 0) {
        return JNI_FALSE;
    }
    return JNI_TRUE;
}
```

JNINativeMethod 结构体的官方定义

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

- 第一个变量name是Java中函数的名字。
- 第二个变量signature，用字符串是描述了Java中函数的参数和返回值
- 第三个变量fnPtr是函数指针，指向native函数。前面都要接 (void *)

第一个变量与第三个变量是对应的，一个是java层方法名，对应着第三个参数的native方法名字(***不明白请看后面代码就会清楚了***)。

哈哈最后我们就把native方法绑定到JNINativeMethod上我们来看下事例:

```
static JNINativeMethod gMethods[] = {
    {"setDataSource",       "(Ljava/lang/String;)V",          (void *)com_media_ffmpeg_FFMpegPlayer_setDataSourc
    {"_setVideoSurface",    "(Landroid/view/Surface;)V",      (void *)com_media_ffmpeg_FFMpegPlayer_setVideo
    {"prepare",             "()V",                            (void *)com_media_ffmpeg_FFMpegPlayer_prepare},
    {"_start",              "()V",                            (void *)com_media_ffmpeg_FFMpegPlayer_start},
    {"_stop",               "()V",                            (void *)com_media_ffmpeg_FFMpegPlayer_stop},
    {"getVideoWidth",       "()I",                            (void *)com_media_ffmpeg_FFMpegPlayer_getVideoWidth},
    {"getVideoHeight",      "()I",                            (void *)com_media_ffmpeg_FFMpegPlayer_getVideoHeight},
    {"seekTo",              "(I)V",                           (void *)com_media_ffmpeg_FFMpegPlayer_seekTo},
    {"_pause",              "()V",                            (void *)com_media_ffmpeg_FFMpegPlayer_pause},
    {"isPlaying",           "()Z",                            (void *)com_media_ffmpeg_FFMpegPlayer_isPlaying},
    {"getCurrentPosition",  "()I",                            (void *)com_media_ffmpeg_FFMpegPlayer_getCurrentPosition
    {"getDuration",         "()I",                            (void *)com_media_ffmpeg_FFMpegPlayer_getDuration},
    {"_release",            "()V",                            (void *)com_media_ffmpeg_FFMpegPlayer_release},
    {"_reset",              "()V",                            (void *)com_media_ffmpeg_FFMpegPlayer_reset},
    {"setAudioStreamType",  "(I)V",                           (void *)com_media_ffmpeg_FFMpegPlayer_setAudioStreamT
    {"native_init",         "()V",                            (void *)com_media_ffmpeg_FFMpegPlayer_native_init},
    {"native_setup",        "(Ljava/lang/Object;)V",          (void *)com_media_ffmpeg_FFMpegPlayer_native_setup},
    {"native_finalize",     "()V",                            (void *)com_media_ffmpeg_FFMpegPlayer_native_finalize},
    {"native_suspend_resume", "(Z)I",                         (void *)com_media_ffmpeg_FFMpegPlayer_native_suspend
};
```

第一个参数就是我们写的方法，第三个就是.h文件里面的方法，第二个参数显得有点难度，这里会主要去讲。

主要是第二个参数比较复杂：

> 括号里面表示参数的类型，括号后面表示返回值。
>
> - “()” 中的字符表示参数，后面的则代表返回值。例如”()V” 就表示void * Fun();
> - “(II)V” 表示 void Fun(int a, int b);
> - “(II)I” 表示 int sum(int a, int b);

这些字符与函数的参数类型的映射表如下：

字符 Java类型 C类型

V void void

Z jboolean boolean

I jint int

J jlong long

D jdouble double

F jfloat float

B jbyte byte

C jchar char

S jshort short

数组则以”[“开始，用两个字符表示

[I jintArray int[]

[F jfloatArray float[]

[B jbyteArray byte[]

[C jcharArray char[]

[S jshortArray short[]

[D jdoubleArray double[]

[J jlongArray long[]

[Z jbooleanArray boolean[]

如图：

Field Descriptor	Java Language Type
"Ljava/lang/String;"	String
"[I"	int[]
"[Ljava/lang/Object;"	Object[]

- 对象类型：以”L”开头，以”;”结尾，中间是用”/” 隔开。如上表第1个
- 数组类型：以”[“开始。如上表第2个（n维数组的话，则是前面多少个”[“而已，如”[[[D”表示“double[][][]” ）
- 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

  Ljava/lang/String; String jstring

  Ljava/net/Socket; Socket jobject

Method Descriptor	Java Language Type
"()Ljava/lang/String;"	String f();
"(ILjava/lang/Class;)J"	long f(int i, Class c);
"([B)V"	String(byte[] bytes);

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$”

好了，所有 的介绍也完了，那么我们就来实现我们的代码：（果断把h文件删除，看效果）
JNIUtil.c：

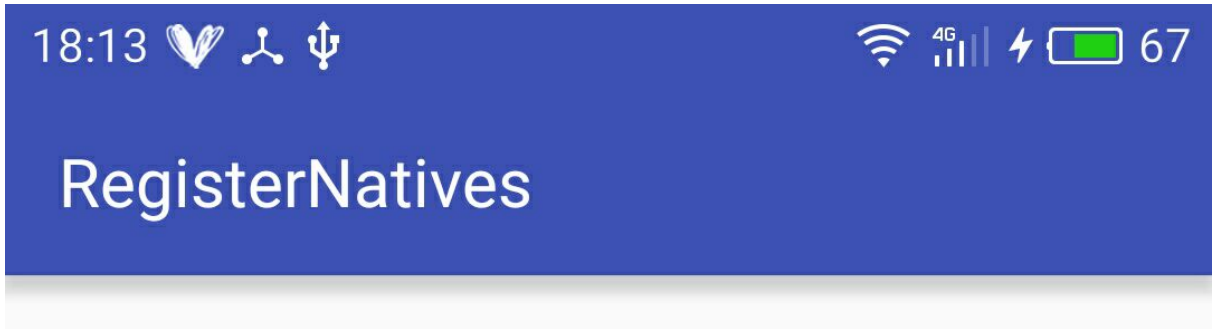
```
1  #include <stdlib.h>
2  #include <string.h>
3  #include <stdio.h>
4  #include <jni.h>
5  #include <assert.h>
6  #define JNIREG_CLASS "com/losileeya/registernatives/JNIUtil"//指定要注册的类
7  jstring call(JNIEnv* env, jobject thiz)
8  {
9     return (*env)->NewStringUTF(env, "动态注册JNI,居然可以把头文件删掉也不会影响结果，牛逼不咯");
10 }
11 jint sum(JNIEnv* env, jobject jobj,jint num1,jint num2){
12    return num1+num2;
13 }
14 /**
```

```
15  * 方法对应表
16  */
17  static JNINativeMethod gMethods[] = {
18          {"stringFromJNI", "()Ljava/lang/String;", (void*)call},
19          {"sum", "(II)I", (void*)sum},
20  };
21
22  /*
23  * 为某一个类注册本地方法
24  */
25  static int registerNativeMethods(JNIEnv* env
26          , const char* className
27          , JNINativeMethod* gMethods, int numMethods) {
28      jclass clazz;
29      clazz = (*env)->FindClass(env, className);
30      if (clazz == NULL) {
31          return JNI_FALSE;
32      }
33      if ((*env)->RegisterNatives(env, clazz, gMethods, numMethods) < 0) {
34          return JNI_FALSE;
35      }
36
37      return JNI_TRUE;
38  }
39
40
41  /*
42  * 为所有类注册本地方法
43  */
44  static int registerNatives(JNIEnv* env) {
45      return registerNativeMethods(env, JNIREG_CLASS, gMethods,
```

```
                    sizeof(gMethods) / sizeof(gMethods[0]));
}

/*
* System.loadLibrary("lib")时调用
* 如果成功返回JNI版本, 失败返回-1
*/
JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM* vm, void* reserved) {
    JNIEnv* env = NULL;
    jint result = -1;
    if ((*vm)->GetEnv(vm, (void**) &env, JNI_VERSION_1_4) != JNI_OK) {
        return -1;
    }
    assert(env != NULL);
    if (!registerNatives(env)) {//注册
        return -1;
    }
    //成功
    result = JNI_VERSION_1_4;
    return result;
}
```

代码写完了，主要也是利用findClass来获取方法，从而实现方法的调用。效果重现：



动态注册JNI,居然可以把头文件删掉也不会影响结果，牛逼不咯

38 + 58 = 96

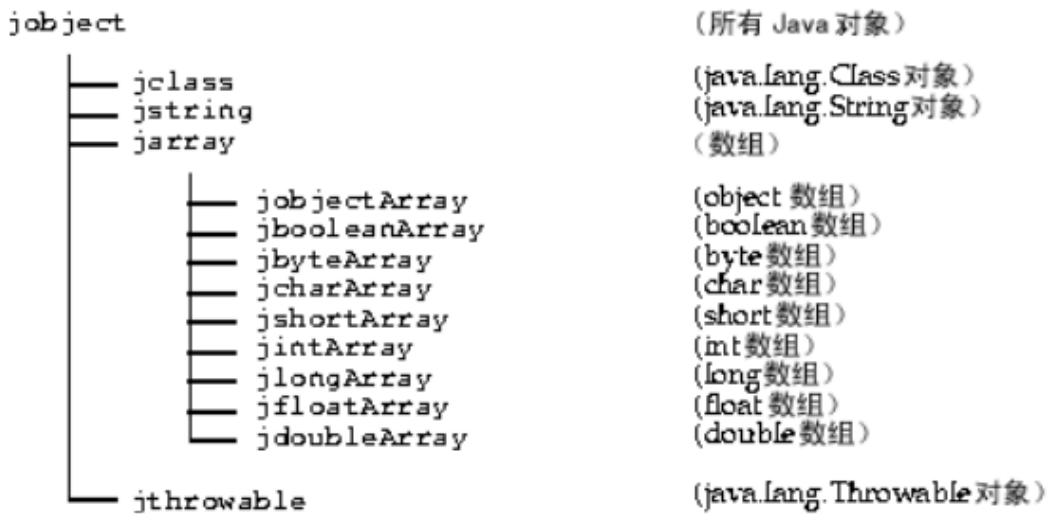
demo 传送梦：RegisterNatives.rar

JNI数据类型及常用方法（JNI安全手册）

基本类型和本地等效类型表：

Java类型	本地类型	说明
boolean	jboolean	无符号，8位
byte	jbyte	无符号，8位
char	jchar	无符号，16位
short	jshort	有符号，16位
int	jint	有符号，32位
long	jlong	有符号，64位
float	jfloat	32位
double	jdouble	
void	void	N/A

引用类型：



接口函数表:

```
1  const struct JNINativeInterface ... = {
2      NULL,
3      NULL,
4      NULL,
5      NULL,
6      GetVersion,
7
8      DefineClass,
9      FindClass,
10     NULL,
11     NULL,
12     NULL,
```

```
13      GetSuperclass,
14      IsAssignableFrom,
15      NULL,
16
17      Throw,
18      ThrowNew,
19      ExceptionOccurred,
20      ExceptionDescribe,
21      ExceptionClear,
22      FatalError,
23      NULL,
24      NULL,
25
26      NewGlobalRef,
27      DeleteGlobalRef,
28      DeleteLocalRef,
29      IsSameObject,
30      NULL,
31      NULL,
32      AllocObject,
33
34      NewObject,
35      NewObjectV,
36      NewObjectA,
37      GetObjectClass,
38
39      IsInstanceOf,
40
41      GetMethodID,
42
43      CallObjectMethod,
```

```
44      CallObjectMethodV,
45      CallObjectMethodA,
46      CallBooleanMethod,
47      CallBooleanMethodV,
48      CallBooleanMethodA,
49      CallByteMethod,
50      CallByteMethodV,
51      CallByteMethodA,
52      CallCharMethod,
53      CallCharMethodV,
54      CallCharMethodA,
55      CallShortMethod,
56      CallShortMethodV,
57      CallShortMethodA,
58      CallIntMethod,
59      CallIntMethodV,
60      CallIntMethodA,
61      CallLongMethod,
62      CallLongMethodV,
63      CallLongMethodA,
64      CallFloatMethod,
65      CallFloatMethodV,
66      CallFloatMethodA,
67      CallDoubleMethod,
68      CallDoubleMethodV,
69      CallDoubleMethodA,
70      CallVoidMethod,
71      CallVoidMethodV,
72      CallVoidMethodA,
73
74      CallNonvirtualObjectMethod,
```

```
75      CallNonvirtualObjectMethodV,
76      CallNonvirtualObjectMethodA,
77      CallNonvirtualBooleanMethod,
78      CallNonvirtualBooleanMethodV,
79      CallNonvirtualBooleanMethodA,
80      CallNonvirtualByteMethod,
81      CallNonvirtualByteMethodV,
82      CallNonvirtualByteMethodA,
83      CallNonvirtualCharMethod,
84      CallNonvirtualCharMethodV,
85      CallNonvirtualCharMethodA,
86      CallNonvirtualShortMethod,
87      CallNonvirtualShortMethodV,
88      CallNonvirtualShortMethodA,
89      CallNonvirtualIntMethod,
90      CallNonvirtualIntMethodV,
91      CallNonvirtualIntMethodA,
92      CallNonvirtualLongMethod,
93      CallNonvirtualLongMethodV,
94      CallNonvirtualLongMethodA,
95      CallNonvirtualFloatMethod,
96      CallNonvirtualFloatMethodV,
97      CallNonvirtualFloatMethodA,
98      CallNonvirtualDoubleMethod,
99      CallNonvirtualDoubleMethodV,
100     CallNonvirtualDoubleMethodA,
101     CallNonvirtualVoidMethod,
102     CallNonvirtualVoidMethodV,
103     CallNonvirtualVoidMethodA,
104
105     GetFieldID,
```

```
106
107     GetObjectField,
108     GetBooleanField,
109     GetByteField,
110     GetCharField,
111     GetShortField,
112     GetIntField,
113     GetLongField,
114     GetFloatField,
115     GetDoubleField,
116     SetObjectField,
117     SetBooleanField,
118     SetByteField,
119     SetCharField,
120     SetShortField,
121     SetIntField,
122     SetLongField,
123     SetFloatField,
124     SetDoubleField,
125     GetStaticMethodID,
126     CallStaticObjectMethod,
127     CallStaticObjectMethodV,
128     CallStaticObjectMethodA,
129     CallStaticBooleanMethod,
130     CallStaticBooleanMethodV,
131     CallStaticBooleanMethodA,
132     CallStaticByteMethod,
133     CallStaticByteMethodV,
134     CallStaticByteMethodA,
135     CallStaticCharMethod,
136     CallStaticCharMethodV,
```

```
137     CallStaticCharMethodA,
138     CallStaticShortMethod,
139     CallStaticShortMethodV,
140     CallStaticShortMethodA,
141     CallStaticIntMethod,
142     CallStaticIntMethodV,
143     CallStaticIntMethodA,
144     CallStaticLongMethod,
145     CallStaticLongMethodV,
146     CallStaticLongMethodA,
147     CallStaticFloatMethod,
148     CallStaticFloatMethodV,
149     CallStaticFloatMethodA,
150     CallStaticDoubleMethod,
151     CallStaticDoubleMethodV,
152     CallStaticDoubleMethodA,
153     CallStaticVoidMethod,
154     CallStaticVoidMethodV,
155     CallStaticVoidMethodA,
156     GetStaticFieldID,
157     GetStaticObjectField,
158     GetStaticBooleanField,
159     GetStaticByteField,
160     GetStaticCharField,
161     GetStaticShortField,
162     GetStaticIntField,
163     GetStaticLongField,
164     GetStaticFloatField,
165     GetStaticDoubleField,
166     SetStaticObjectField,
167     SetStaticBooleanField,
```

```
168     SetStaticByteField,
169     SetStaticCharField,
170     SetStaticShortField,
171     SetStaticIntField,
172     SetStaticLongField,
173     SetStaticFloatField,
174     SetStaticDoubleField,
175     NewString,
176     GetStringLength,
177     GetStringChars,
178     ReleaseStringChars,
179     NewStringUTF,
180     GetStringUTFLength,
181     GetStringUTFChars,
182     ReleaseStringUTFChars,
183     GetArrayLength,
184     NewObjectArray,
185     GetObjectArrayElement,
186     SetObjectArrayElement,
187     NewBooleanArray,
188     NewByteArray,
189     NewCharArray,
190     NewShortArray,
191     NewIntArray,
192     NewLongArray,
193     NewFloatArray,
194     NewDoubleArray,
195     GetBooleanArrayElements,
196     GetByteArrayElements,
197     GetCharArrayElements,
198     GetShortArrayElements,
```

```
199     GetIntArrayElements,
200     GetLongArrayElements,
201     GetFloatArrayElements,
202     GetDoubleArrayElements,
203     ReleaseBooleanArrayElements,
204     ReleaseByteArrayElements,
205     ReleaseCharArrayElements,
206     ReleaseShortArrayElements,
207     ReleaseIntArrayElements,
208     ReleaseLongArrayElements,
209     ReleaseFloatArrayElements,
210     ReleaseDoubleArrayElements,
211     GetBooleanArrayRegion,
212     GetByteArrayRegion,
213     GetCharArrayRegion,
214     GetShortArrayRegion,
215     GetIntArrayRegion,
216     GetLongArrayRegion,
217     GetFloatArrayRegion,
218     GetDoubleArrayRegion,
219     SetBooleanArrayRegion,
220     SetByteArrayRegion,
221     SetCharArrayRegion,
222     SetShortArrayRegion,
223     SetIntArrayRegion,
224     SetLongArrayRegion,
225     SetFloatArrayRegion,
226     SetDoubleArrayRegion,
227     RegisterNatives,
228     UnregisterNatives,
229     MonitorEnter,
```

```
    MonitorExit,
    GetJavaVM,
};
```

基本上jni的数据和方法都差不多放这里了，你就可以随便开发了。这个你也可以去看jni完全手册

JNI与C/C++数据类型的转换（效率开发）

字符数组与jbyteArray

- jbyteArray转字符数组

```
int byteSize = (int) env->GetArrayLength(jbyteArrayData);  //jbyteArrayData是jbyteArray类型的数据
unsigned char* data = new unsigned char[byteSize + 1];
env->GetByteArrayRegion(jbyteArrayData, 0, byteSize, reinterpret_cast<jbyte*>(data));
data[byteSize] = '\0';
```

- 字符数组转jbyteArray

```
jbyte *jb =  (jbyte*) data;   //data是字符数组类型
jbyteArray jarray = env->NewByteArray(byteSize);   //byteSize是字符数组大小
env->SetByteArrayRegion(jarray, 0, byteSize, jb);
```

字符数组与jstring

- jstring转字符数组

```
char* JstringToChar(JNIEnv* env, jstring jstr) {
    if(jstr == NULL) {
        return NULL;
    }
    char* rtn = NULL;
    jclass clsstring = env->FindClass("java/lang/String");
    jstring strencode = env->NewStringUTF("utf-8");
    jmethodID mid = env->GetMethodID(clsstring, "getBytes",
            "(Ljava/lang/String;)[B");
    jbyteArray barr = (jbyteArray) env->CallObjectMethod(jstr, mid, strencode);
    jsize alen = env->GetArrayLength(barr);
    jbyte* ba = env->GetByteArrayElements(barr, JNI_FALSE);
    if (alen > 0) {
        rtn = (char*) malloc(alen + 1);
        memcpy(rtn, ba, alen);
        rtn[alen] = 0;
    }
    env->ReleaseByteArrayElements(barr, ba, 0);
    return rtn;
}
```

- 字符数组转jstring

```
jstring StrtoJstring(JNIEnv* env, const char* pat)
{
    jclass strClass = env->FindClass("java/lang/String");
    jmethodID ctorID = env->GetMethodID(strClass, "<init>", "([BLjava/lang/String;)V");
    jbyteArray bytes = env->NewByteArray(strlen(pat));
    env->SetByteArrayRegion(bytes, 0, strlen(pat), (jbyte*)pat);
    jstring encoding = env->NewStringUTF("utf-8");
```

```
8     return (jstring)env->NewObject(strClass, ctorID, bytes, encoding);
9 }
```

特么最简单的可以直接使用

```
1 jstring jstr = env->NewStringUTF(str);
```

jint与int的互转都可以直接使用强转，如：

```
1 jint i = (jint) 1024;
```

上面的代码你看见了吗，都是env的一级指针来做的，所以是cpp的使用方法，如果你要转成c的那么就把env替换为（*env）好了，具体的方法可能有点小改动（请自行去参考jni手册），报错的地方请自行引入相关的.h文件，估计对你了解jni有更深的了解。

总结

本篇主要介绍了JNI动态注册native方法，并且顺便截了几个jni的图，以及使用的基本数据转换处理，至于实际应用中比如java 调用c,c调用java以及混合调用等我们都需要实践中去处理问题。

> 至于学习过程中可能用到反射或其他更多的东西，还值得我们去挖掘，难道你就不想知道美图秀秀哪些对图片处理是怎么利用jni来实现的？
>
> come on.enjoy it !

- 上一篇 安卓实战开发之JNI入门及高效的配置（android studio一键生成.h,so及方法签名）
- 下一篇 安卓实战开发之JNI再深入了解