一：什么是JNI

JNi就是java调用本地方法的技术，最简单的来说，java运行一个程序需要要和不同的系统平台打交道，在windows里就是和windows平台底层打交道，mac就是要和mac打交道，jvm就是通过大量的jni技术使得java能够在不同平台上运行。

使用了这技术的一个标志就是native,如果一个类里的一个方法被native修饰，那就说明这个方法是jni来实现的，他是通过本地系统api里的方法来实现的。当然这个本地方法可能是c或者C++，当然也可能是别的语言。jni是java跨平台的基础，jvm通过在不同系统上调用不同的本地方法使得jvm可以在不同平台间移植。

如下图所示

上图中有一处不太严谨的地方，那便是（C/C++），如果是mac os那就不能说是C/C++了，但是在**Android系统**中我们只能用C/C++。

1.1 Java和C/C++ 中的基本类型的映射关系：

JNI是接口语言，因而，会有一个中间的转型过程，在这个过程中，有一个非常重要的也是非常关键的类型对接方式，这个方式便是，数据类型的转变，下表给出了相关的对于的数据格式。

下表中的数据为JNI基本数据类型及对应的长度

java类型	jni类型	描述
boolean	jboolean	unsigned 8 bits
byte	jbyte	signed 8 bits
char	jchar	unsigned 16 bits
short	jshort	signed 16 bits
int	jint	signed 32 bits
long	jlong	signed 64 bits
float	jfloat	signed 32bit
double	jdouble	signed 64 bits
Class	jclass	class类对象
String	jstring	字符串对象
Object	jobject	任何java对象
byte[]	jbyteArray	byte数组

这个表都是JNI开发中 java 和 JNI之间数据的适配。

二：动态库和静态库

Android NDK种的动态库和静态库就是linux下的动态库和静态库，因为NDK的开发可以理解从基于Linux的开发。

在平时工作中我们经常把一些常用的函数或者功能封装为一个个库供给别人使用，java开发我们可以封装为jar包提供给别人用，安卓平台后来可以打包成aar包，同样的，C/C++中我们封装的功能或者函数可以通过静态库或者动态库的方式提供给别人使用。

Linux平台静态库以.a结尾，而动态库以.so结尾。

要分析链接库首先要分析**交叉编译**

2.1什么是交叉编译?

交叉编译就是在A平台编译出可以在B平台执行的文件，对于我们安卓开发者来说交叉编译就是在window或者mac或者linux系统上编译出可在安卓系统上运行的可执行文件，什么时候需要用到交叉编译呢？音视频开发基本都会用到ffmpeg，opengl es等三方库，这时我们就需要在window或者mac或者linux系统上编译出可在安卓系统执行的文件，这里可编译出静态库或者动态库使用，这时候就会用到交叉编译。

交叉编译的目的很清楚，就是编译出除了自己平台意外其他平台可以用的库的过程。那么在Android 平台的代码，由于Android平台是基于linux，因此很多Android 中可以运行的库就是在linux中编译的，或者是在mac上面编译，而在windows上面编译就比较难了。那么到底什么是动态库和静态库呢?

2.2 动态库和静态库（共享库）

2.2.1 静态库

这类库的名字一般是libxxx.a；利用静态函数库编译成的文件比较大，因为整个 函数库的所有数据都会被整合进目标代码中，他的优点就显而易见了，即编译后的执行程序不需要外部的函数库支持，因为所有使用的函数都已经被编译进去了。当然这也会成为他的缺点，因为如果静态函数库改变了，那么你的程序必须重新编译。

2.2.2 动态库

这类库的名字一般是libxxx.so;相对于静态函数库，动态函数库在编译的时候 并没有被编译进目标代码中，你的程序执行到相关函数时才调用该函数库里的相应函数，因此动态函数库所产生的可执行文件比较小。由于函数库没有被整合进你的程序，而是程序运行时动态的申请并调用，所以程序的运行环境中必须提供相应的库。动态函数库的改变并不影响你的程序，所以动态函数库的升级比较方便

静态库的代码在编译过程中已经被载入可执行程序,因此体积比较大；动态库(共享库)的代码在可执行程序运行时才载入内存，在编译过程中仅简单的引用，因此代码体积比较小。

2.3 Android如何配置cmakelist.txt 配置编译动态库和静态库呢?

```
add_library(jinInterface SHARED library.c library.h)// SHARED 表示是动态库
add_library(jinInterface STATIC library.c library.h)// STATIC 表示是静态库
ADD_LIBRARY(...)
语法：ADD_LIBRARY(libname [SHARED|STATIC] )
上面的表达式等同于：
set(LIB_SRC library.c library.h)
add_library(jinInterface SHARED ${LIB_SRC})
```

三 JNI 动态注册与静态注册

3.1. 静态注册

步骤：

1）编写java类，假如是JniTest.java

2）在命令行下输入 javac JniTest.java 生成JniTest.class文件

3) 在 JniTest.class 目录下 通过 javah xxx.JniTest(全类名)生成 xxx_JniTest.h 头文件

4）编写xxx_JniTest.c 源文件，并拷贝xxx_JniTest.h 下的函数，并实现这些函数，且在其中添加jni.h头文件；

5）编写 cmakelist.txt 文件，编译生成动态/静态链接库

3.2. 动态注册：

在此之前我们一直在jni中使用的 Java_PACKAGENAME_CLASSNAME_METHODNAME 来进行与java方法的匹配，这种方式我们称之为静态注册。

而动态注册则意味着方法名可以不用这么长了，在android aosp源码中就大量的使用了动态注册的形式

```

```

```
//Java:
native void dynamicNative();
native String dynamicNative(int i);

//C++:
void  dynamicNative1(JNIEnv *env, jobject jobj){
    LOGE("dynamicNative1 动态注册");
}
jstring  dynamicNative2(JNIEnv *env, jobject jobj,jint i){
    return env->NewStringUTF("我是动态注册的dynamicNative2方法");
}

//需要动态注册的方法数组
static const JNINativeMethod mMethods[] = {
        {"dynamicNative","()V", (void *)dynamicNative1},
        {"dynamicNative", "(I)Ljava/lang/String;", (jstring *)dynamicNative2}

};
//需要动态注册native方法的类名
static const char* mClassName = "com/dongnao/jnitest/MainActivity";
jint JNI_OnLoad(JavaVM* vm, void* reserved){
    JNIEnv* env = NULL;
    //获得 JniEnv
    int r = vm->GetEnv((void**) &env, JNI_VERSION_1_4);
    if( r != JNI_OK){
        return -1;
    }
    jclass mainActivityCls = env->FindClass( mClassName);
    // 注册 如果小于0则注册失败
    r = env->RegisterNatives(mainActivityCls,mMethods,2);
    if(r  != JNI_OK )
    {
        return -1;
    }
    return JNI_VERSION_1_4;
}
```

3.3 system.load()/system.loadLibrary() 区别

https://zhidao.baidu.com/question/1241601021778319299.html

System.load

System.load 参数必须为库文件的绝对路径，可以是任意路径，例如： System.load("C:\Documents and Settings\TestJNI.dll"); //Windows

System.load("/usr/lib/TestJNI.so"); //Linux

System.loadLibrary

System.loadLibrary 参数为库文件名，不包含库文件的扩展名。

System.loadLibrary ("TestJNI"); //加载Windows下的TestJNI.dll本地库

System.loadLibrary ("TestJNI"); //加载Linux下的libTestJNI.so本地库

注意：TestJNI.dll 或 libTestJNI.so 必须是在JVM属性java.library.path所指向的路径中。**

loadLibary需要配置当前项目的java.library.path 路径，具体办法如：https://zhidao.baidu.com/question/1241601021778319299.html

四：JNIEnv类型和jobject类型的解释

```
JNIEXPORT **void JNICALL Java_com_jni_demo_JNIDemo_sayHello (JNIEnv \* env, jobject obj)**

{
    cout<<"Hello World"<<endl;
}
```

这里JNIEXPORT 与 JNICALL 都是JNI的关键字，表示此函数是要被JNI调用的，无需过多解释

4.1 JNIEnv* env参数的解释

JNIEnv类型实际上代表了Java环境，通过这个JNIEnv* 指针，就可以对Java端的代码进行操作。例如，创建Java类中的对象，调用Java对象的方法，获取Java对象中的属性等等。JNIEnv的指针会被JNI传入到本地方法的实现函数中来对Java端的代码进行操作。如下代码所示

```
#ifdef __cplusplus
typedef JNIEnv_ JNIEnv;
#else
typedef const struct JNINativeInterface_ *JNIEnv;
#endif
......

struct JNIInvokeInterface_;
......

struct JNINativeInterface_ {
    void *reserved0;
    void *reserved1;
    void *reserved2;

    void *reserved3;
    jint (JNICALL *GetVersion)(JNIEnv *env);

//全是函数指针
    jclass (JNICALL *DefineClass)
      (JNIEnv *env, const char *name, jobject loader, const jbyte *buf,
       jsize len);
    jclass (JNICALL *FindClass)
      (JNIEnv *env, const char *name);

    jmethodID (JNICALL *FromReflectedMethod)
      (JNIEnv *env, jobject method);
    jfieldID (JNICALL *FromReflectedField)
```

```
      (JNIEnv *env, jobject field);

    jobject (JNICALL *ToReflectedMethod)
      (JNIEnv *env, jclass cls, jmethodID methodID, jboolean isStatic);
  ...
  }
```

4.2 参数:jobject obj的解释

如果native方法不是static的话，**这个obj就代表这个native方法的类实例。**

如果native方法是static的话，**这个obj就代表这个native方法的类的class对象实例**(static方法不需要类实例的，所以就代表这个类的class对象)。

代码如下：

java代码

```
public native void test();
public static native void testStatic();
```

Jni代码 .h

```
JNIEXPORT void JNICALL Java_Hello_test
  (JNIEnv *, jobject);
JNIEXPORT void JNICALL Java_Hello_testStatic
  (JNIEnv *, jclass);
```

五：C/C++代码调用Java代码

上面讲解的大多数内容用于阐述java调用C/C++端代码，然而在JNI中还有 一个非常重要的内容，那便是在C/C++本地代码中访问Java端的代码，一个常见的应用就是获取类的属性和调用类的方法，为了在C/C++中表示属性和方法，JNI在jni.h头文件中定义了jfieldId,jmethodID类型来分别代表Java端的属性和方法。

我们在访问，或者设置Java属性的时候，首先就要先在本地代码取得代表该Java属性的jfieldID,然后才能在本地代码中进行Java属性操作，同样的，我们需要呼叫Java端的方法时，也是需要取得代表该方法的jmethodID才能进行Java方法调用。

使用JNIEnv的：GetFieldID/GetMethodID

GetStaticFieldID/GetStaticMethodID

来取得相应的jfieldID和jmethodID。

下面来具体看一下这几个方法：

```
GetFieldID(jclass clazz,const char* name,const char* sign)
方法的参数说明:
    clazz:这个简单就是这个方法依赖的类对象的class对象
    name:这个是这个字段的名称
    sign:这个是这个字段的签名(我们知道每个变量，每个方法都是有签名的)
```

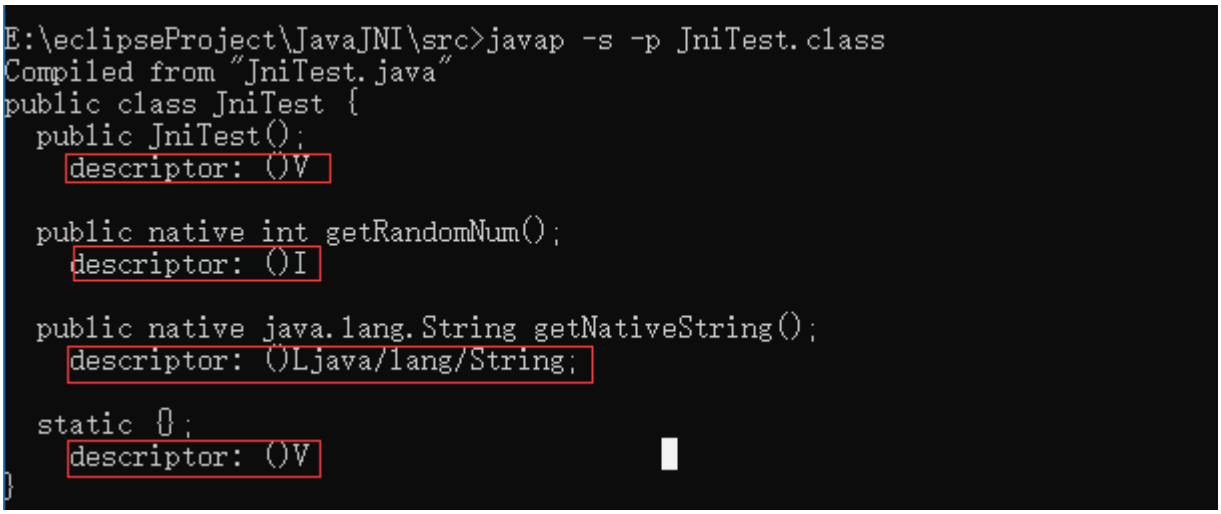
在上面代码中有一个新的问题，那便是sign，签名怎么来的，签名的格式是怎样的?

5.1 签名问题

5.1.1 怎么查看类中的字段和方法的签名:**

使用javap命令:

javap -s -p JniTes.class



5.1.2 java 字段的签名

很多时候，我们的签名除了根据命令行来定，其实还可以依据规律自己写出来，这个规则就是以下面的表格为基本准则来制定的。

数据类型	签名
boolean	Z
byte	B
char	C
short	S
int	I
long	L
float	F
double	D
void	V
object	L开头，然后以/ 分隔包的完整类型，后面再加；比如说 String 签名就是Ljava/lang/String
Array	以[开头，在加上数组元素类型的签名比如 int[] 签名就是[I ，在比如int[][] 签名就是[[I ，object 数组签名就是[Ljava/lang/Object;

使用签名取得属性/方法ID的例子

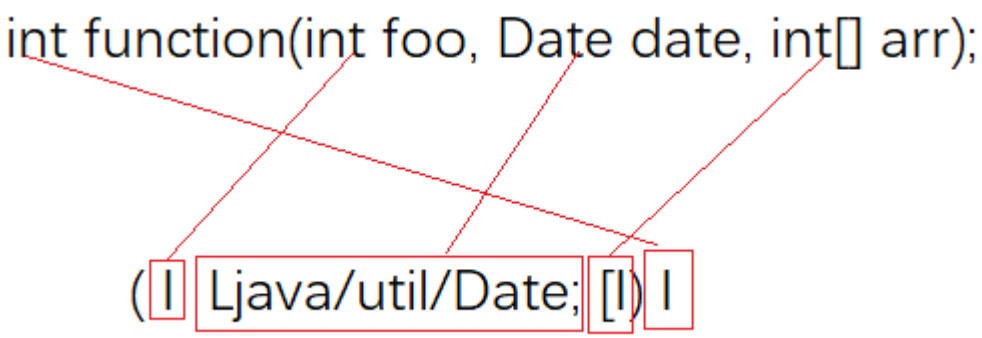
Java代码：

```java
import java.util.Date;

public class Hello {
    public int property;
    public int function(int fu, Date date, int[] arr) {
        System.out.println("funtion");
        return 0;
    }

    public native void test();
}
```

JNI代码：

```
JNIEXPORT void JNICALL Java_Hello_test
        (JNIEnv * env, jobject jobj){
    jclass hello_clz = (*env)->GetObjectClass(env, jobj);
    jfieldID  fieldID_prop = (*env)->GetFieldID(env, hello_clz, "property", "I");
    jmethodID methodID_func = (*env)->GetMethodID(env, hello_clz, "function", "(ILjava/util/Date;[I)I");
    (*env)->CallIntMethod(env,jobj,methodID_func,0L,NULL,NULL);
}
```

解说：



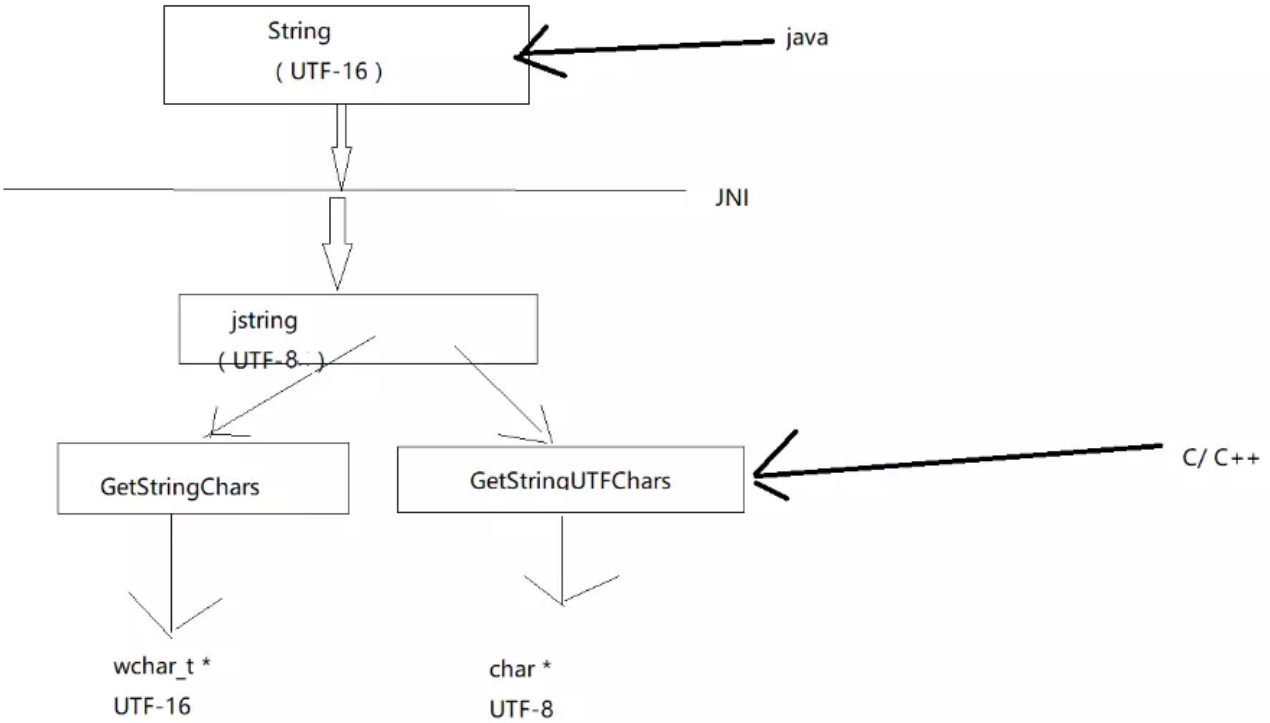
所以在最后得到（ILjava/util/Date;[I)I 这个签名.

了解Java反射的童鞋应该知道，在Java中任何一个类的.class字节码文件被加载到内存中之后，该class子节码文件统一使用Class类来表示该类的一个引用（相当于Java中所有类的基类是Object一样）。然后就可以从该类的Class引用中动态的获取类中的任意方法和属性。注意：Class类在Java SDK继承体系中是一个独立的类，没有继承自Object。请看下面的例子，通过Java反射机制，动态的获取一个类的私有实例变量的值：

```
public static void main(String[] args) throws Exception {
    ClassField obj = new ClassField();
    obj.setStr("YangXin");
    // 获取ClassField字节码对象的Class引用
    Class<?> clazz = obj.getClass();
    // 获取str属性
    Field field = clazz.getDeclaredField("str");
    // 取消权限检查，因为Java语法规定，非public属性是无法在外部访问的
    field.setAccessible(true);
    // 获取obj对象中的str属性的值
    String str = (String)field.get(obj);
    System.out.println("str = " + str);
}
```

运行程序后，输出结果当然是打印出str属性的值“YangXin”。所以我们在本地代码中调用JNI函数访问Java对象中某一个属性的时候，首先第一步就是要获取该对象的Class引用，然后在Class中查找需要访问的字段ID，最后调用JNI函数的GetXXXField系列函数，获取字段(属性)的值。

5.2. JNI 访问字符串




```
//java内部使用的是utf-16 16bit 的编码方式
//jni 里面使用的utf-8  unicode编码方式  英文是1个字节，中文 3个字节
//C/C++ 使用 ascii编码 ，中文的编码方式 GB2312编码 中文 2个字节
```

代码如下：

Java代码：

```
public native static String sayHello(String text);
```

C/C++代码：

```
JNIEXPORT jstring JNICALL Java_JString_sayHello
        (JNIEnv * env, jclass jclaz, jstring jstr) {
    const char * c_str = NULL;
    char buf[128] = {0};
    jboolean  iscopy;
    c_str = (*env)->GetStringUTFChars(env, jstr, &iscopy);
    printf("isCopy:%d\n", iscopy);
    if(c_str == NULL) {
        return NULL;
    }
    printf("C_str: %s \n", c_str);
    sprintf(buf,"Hello, 你好 %s", c_str);
    printf("C_str: %s \n", buf);
    (*env)->ReleaseStringUTFChars(env, jstr, c_str);
    return (*env)->NewStringUTF(env,buf);
}
```

5.3 访问Java 成员变量

Java成员变量一般有两类：静态和非静态。所以在JNI中对这两种不同的类型就有了两种不太相同的调用方法。

5.3.1. 非静态变量

java代码

```
public int property;
```

Jni代码

```
JNIEXPORT void JNICALL Java_Hello_testField(JNIEnv *env, jobject jobj) {
    jclass  claz = (*env)->GetObjectClass(env,jobj);
    jfieldID  jfid = (*env)->GetFieldID(env, claz, "property","I");
    jint va = (*env)->GetIntField(env,jobj, jfid);
    printf("va: %d", va);
    (*env)->SetIntField(env, jobj, jfid, va + 10086);
}
```

上例中，首先调用GetObjectClass函数获取ClassField的Class引用：

```
clazz = (*env)->GetObjectClass(env,obj);
```

然后调用GetFieldID函数从Class引用中获取字段的ID（property是字段名，I是字段的签名）

```
jfieldID  jfid = (*env)->GetFieldID(env, claz, "property","I");
```

最后调用GetIntField函数，传入实例对象和字段ID，获取属性的值

```
jint va = (*env)->GetIntField(env,jobj, jfid);
```

额外的，如果要修改这个值就可以使用SetIntField函数：

```
 (*env)->SetIntField(env, jobj, jfid, va + 10086);
```

5.3.2. 访问静态变量

访问静态变量和实例变量不同的是，获取字段ID使用GetStaticFieldID，获取和修改字段的值使用Get/SetStaticXXXField系列函数，比如上例中获取和修改静态变量num：

```
num = (*env)->GetStaticIntField(env,clazz,fid);
// 4.修改静态变量num的值
(*env)->SetStaticIntField(env, clazz, fid, 80);
```

5.3.3总结：

1、由于JNI函数是直接操作JVM中的数据结构，不受Java访问修饰符的限制。即，在本地代码中可以调用JNI函数可以访问Java对象中的非public属性和方法

2、访问和修改实例变量操作步聚：

1>、调用GetObjectClass函数获取实例对象的Class引用

2>、调用GetFieldID函数获取Class引用中某个实例变量的ID

3>、调用GetXXXField函数获取变量的值，需要传入实例变量所属对象和变量ID

4>、调用SetXXXField函数修改变量的值，需要传入实例变量所属对象、变量ID和变量的值

3、访问和修改静态变量操作步聚： 1>、调用FindClass函数获取类的Class引用

2>、调用GetStaticFieldID函数获取Class引用中某个静态变量ID

3>、调用GetStaticXXXField函数获取静态变量的值，需要传入变量所属Class的引用和变量ID

4>、调用SetStaticXXXField函数设置静态变量的值，需要传入变量所属Class的引用、变量ID和变量的值

5.4 访问Java中的函数

Java成员函数一般有两类：静态和非静态。所以在JNI中对这两种不同的类型就有了两种不太相同的调用方法，这两种不同类型虽然他们的调用方式有些许不同，但是，他们的实质上是一样的。只是调用的接口的名字有区别，而对于流程是没有区别的。

Java代码如下：

```
private static void callStaticMethod(String str, int i) {
        System.out.format("ClassMethod::callStaticMethod called!-->str=%s," +
                " i=%d\n", str, i);
}
```

JNI代码如下：

```
JNIEXPORT void JNICALL Java_JniTest_callJavaStaticMethod(JNIEnv *env, jobject jobje){
    jclass  clz = (*env)->FindClass(env,"ClassMethod");
    if(clz == NULL) {
        printf("clz is null");
        return;
    }
    jmethodID  jmeid = (*env)->GetStaticMethodID(env, clz, "callStaticMethod", "
(Ljava/lang/String;I)V");
    if(jmeid == NULL) {
        printf("jmeid is null");
        return;
    }
    jstring arg = (*env)->NewStringUTF(env, "我是静态类");
    (*env)->CallStaticVoidMethod(env,clz,jmeid,arg,100);
    (*env)->DeleteLocalRef(env,clz);
    (*env)->DeleteLocalRef(env,arg);
}
```

流程如下：

1：从classpath路径下搜索ClassMethod这个类，并返回该类的Class对象

```
jclass  clz = (*env)->FindClass(env,"ClassMethod");
```

2：从clazz类中查找callStaticMethod方法

```
 jmethodID  jmeid = (*env)->GetStaticMethodID(env, clz, "callStaticMethod", "
(Ljava/lang/String;I)V");
```

3：调用clazz类的callStaticMethod静态方法

```
(*env)->CallStaticVoidMethod(env,clz,jmeid,arg,100);
```

该函数接收4个参数：

env：JNI函数表指针

clz：调用该静态方法的Class对象

methodID：方法ID（因为一个类中会存在多个方法，需要一个唯一标识来确定调用类中的哪个方法） 参数4：方法实参列表

六：JNI引用

在 JNI 规范中定义了三种引用：局部引用（Local Reference）、全局引用（Global Reference）、弱全局引用（Weak Global Reference）。

6.1 局部引用

通过NewLocalRef和各种JNI接口创建（FindClass、NewObject、GetObjectClass和NewCharArray等）。会阻止GC回收所引用的对象，不能本地函数中跨函数使用，不能跨线前使用。函数返回后局部引用所引用的对象会被JVM自动释放，或调用DeleteLocalRef释放。`(*env)->DeleteLocalRef(env,local_ref)`

```
jclass cls_string = (*env)->FindClass(env, "java/lang/String");
jcharArray charArr = (*env)->NewCharArray(env, len);
jstring str_obj = (*env)->NewObject(env, cls_string, cid_string, elemArray);
jstring str_obj_local_ref = (*env)->NewLocalRef(env,str_obj);   // 通过NewLocalRef函数创建
...
```

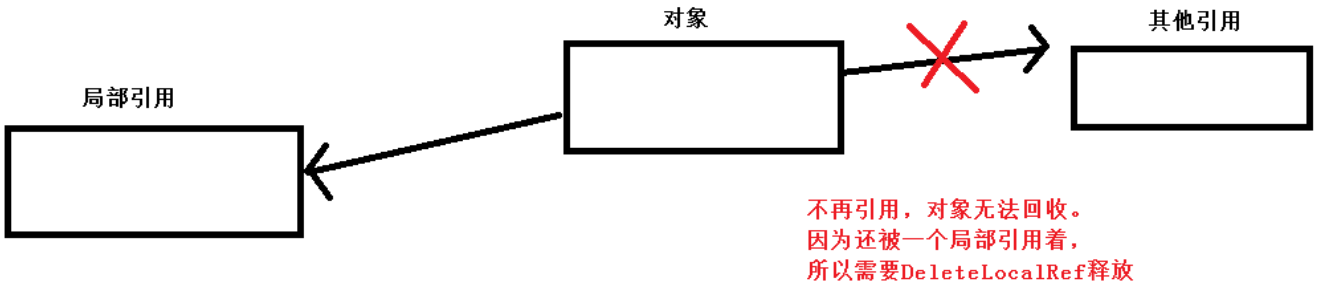
释放一个局部引用有两种方式:

1、本地方法执行完毕后VM自动释放;

2、通过DeleteLocalRef手动释放;

VM会自动释放局部引用，为什么还需要手动释放呢?

因为局部引用会阻止它所引用的对象被GC回收。



6.2 全局引用

调用NewGlobalRef基于局部引用创建，会阻GC回收所引用的对象。可以跨方法、跨线程使用。JVM不会自动释放，必须调用DeleteGlobalRef手动释放`(*env)->DeleteGlobalRef(env,g_cls_string);`

```
static jclass g_cls_string;
void TestFunc(JNIEnv* env, jobject obj) {
    jclass cls_string = (*env)->FindClass(env, "java/lang/String");
    g_cls_string = (*env)->NewGlobalRef(env,cls_string);
}
```

6.3 弱全局引用

调用NewWeakGlobalRef基于局部引用或全局引用创建，不会阻止GC回收所引用的对象，可以跨方法、跨线程使用。引用不会自动释放，在JVM认为应该回收它的时候（比如内存紧张的时候）进行回收而被释放。或调用DeleteWeakGlobalRef手动释放。(*env)->DeleteWeakGlobalRef(env,g_cls_string)

```
static jclass g_cls_string;
void TestFunc(JNIEnv* env, jobject obj) {
    jclass cls_string = (*env)->FindClass(env, "java/lang/String");
    g_cls_string = (*env)->NewWeakGlobalRef(env,cls_string);
}
```

6.4 一个关于引用错误导致的野指针问题

在C/C++中最容易出现的问题也是最容易非忽视的问题就是**野指针**问题，它就像Java 中的 IllegalAccessException类似。

6.4.1 野指针是什么问题呢?

野指针指向一个已删除的对象或未申请访问受限内存区域的指针。通俗的讲，就是该指针就像野孩子一样，不受程序控制，不知道该指针指向了什么地址。 与空指针不同，野指针无法通过简单地判断是否为NULL避免，而只能通过养成良好的编程习惯来尽力减少。对野指针进行操作很容易造成程序错误。

6.4.2 野指针的危害:

野指针的问题在于，指针指向的内存已经无效了，而指针没有被置空，此时指针随机指向某个地址。引用一个非空的无效指针是一个未被定义的行为，也就是说不一定导致段错误，野指针很难定位到是哪里出现的问题，在哪里这个指针就失效了，不好查找出错的原因。所以调试起来会很麻烦，有时候会需要很长的时间。 因此，要想彻底地避免野指针，最好的办法就是养成一个良好的编程习惯。 1）初始化指针时将其置为NULL，之后再对其进行操作。 2）释放指针时将其置为NULL，最好在编写代码时将free()函数封装一下，在调用free()后就将指针置为NULL。

如下引用就带来了野指针：

jni代码：

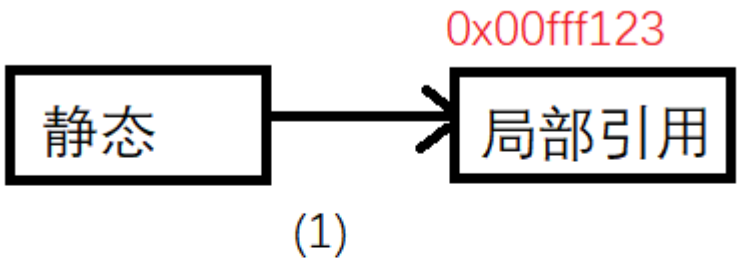
```
JNIEXPORT jstring JNICALL Java_Refrence_newString(JNIEnv * env, jobject jobj,jint len){
    jcharArray elemArray;
    jchar *chars = NULL;
    jstring j_str = NULL;
    // 定义静态的局部变量
    static jclass cls_string = NULL;
    static jmethodID  cid_string = NULL;

    if (cls_string == NULL) {
        printf("cls_string is null \n");
        cls_string = (*env)->FindClass(env, "java/lang/String");
        if (cls_string == NULL) {
            return NULL;
        }
    }

    if(cid_string == NULL) {
        printf("cid_string is null \n");
        cid_string = (*env)->GetMethodID(env, cls_string, "<init>", "([C)V");
        if (cid_string == NULL) {
            return NULL;
        }
    }
    printf("this is a line ------------\n");

    elemArray = (*env)->NewCharArray(env, len);
    j_str = (*env)->NewObject(env, cls_string, cid_string, elemArray);

    (*env)->DeleteLocalRef(env, elemArray);
    printf("end of function \n");
    return j_str;
}
```

```
static jclass cls_string = NULL;
static jmethodID  cid_string = NULL;
```

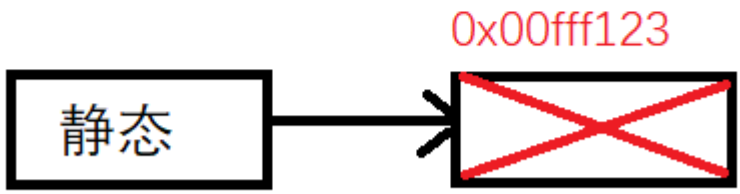
以上两个变量都是静态的局部变量。

静态局部变量的作用域：作用域仍为局部作用域，当定义它的函数或者语句块结束的时候，作用域结束。但是当局部静态变量离开作用域后，并没有销毁，而是仍然驻留在内存当中，只不过我们不能再对它进行访问，直到该函数再次被调用，并且值不变。换句话说，静态局部变量在函数内有效，由于是静态变量，所以当再次调用这个函数的时候，静态变量的值继续存在着。

cls_string/cid_string 这两个静态局部变量的值是 **局部引用**，局部引用的特点是：函数返回后局部引用所引用的对象会被JVM自动释放。这样一来，给我们的结果就是：静态局部变量的值 所指向的内容被释放，出现野指针异常。 如下图所示：



(1)



(2)

上图（2）就是说明0x00fff123 这个地址里面的东西为NULL了，然后，静态变量的值还是：0x00fff123，因而就无法用 静态变量 == NULL来判断变量，因而，在再次使用的时候就出现了野指针异常。

解决办法如下：

```
(*env)->DeleteLocalRef(env, cls_string);
cls_string = NULL;
// 此处的 delete不能存在，因为 cid_string不是jobject，应用只需要对object类型的引用而言的，
// (*env)->DeleteLocalRef(env, cid_string);
cid_string = NULL;
```