banketree

专注移动平台

JNI 实战全面解析

分类： android 2014-11-01 09:05 6426人阅读 评论(1) 收藏 举报

目录(?) [+]

简介

项目决定移植一款C++开源项目到Android平台，开始对JNI深入研究。

JNI是什么？

JNI(Java Native Interface)意为JAVA本地调用，它允许Java代码和其他语言写的代码进行交互，简单的说，一种在Java虚拟机控制下执行代码的标准机制。

NDK是什么？

Android NDK（Native Development Kit ）是一套工具集合，允许你用像C/C++语言那样实现应用程序的一部分。

为什么要用NDK？

1、安全性，java是半解释型语言，很容易被反汇编后拿到源代码文件，我们可以在重要的交互功能使用C语言代替。
2、效率，C语言比起java来说效率要高出很多。

JNI和NDK的区别？

从工具上说，NDK其实多了一个把.so和.apk打包的工具，而JNI开发并没有打包，只是把.so文件放到文件系统的特定位置。
从编译库说，NDK开发C/C++只能能使用NDK自带的有限的头文件，而使用JNI则可以使用文件系统中带的头文件。
从编写方式说，它们一样。

详解

1、JNI 元素

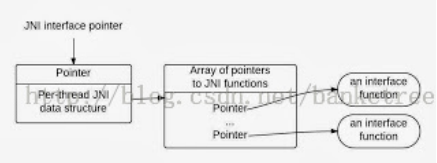
1、JNI组织结构

VC++ (26)
外挂 (1)
linux (4)
vmware (1)
java (3)
梦想 (1)
IOS (2)
通讯、游戏引擎 (7)

文章存档
2015年01月 (4)
2014年12月 (5)
2014年11月 (4)
2014年10月 (5)
2014年09月 (6)
展开

评论排行
Android 双卡双待识别 (60)
开源项目之Android http请 (49)
c/c++成长之捷径 (48)
Android 上百实例源码分 (39)
CSipsimple的封装 (39)
JAVA上百实例源码以及开 (38)
Sipdroid的封装 (37)
Android 面试有感 (28)
编程之路 (24)
Android 设计模式 (23)

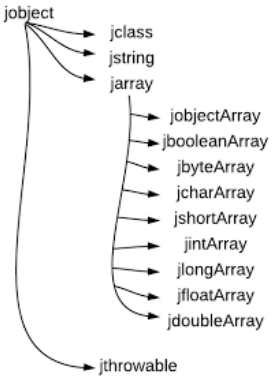
推荐文章



JNI函数表的组成就像C++的虚函数表，虚拟机可以运行多张函数表。
JNI接口指针仅在当前线程中起作用，指针不能从一个线程进入另一个线程，但可以在不同的线程中调用本地方法。

2、原始数据

Jobject 对象 引用类型



Java 类型	本地类型（JNI）	描述
boolean（布尔型）	jboolean	无符号8个比特
byte(字节型)	jbyte	有符号8个比特
char(字符型)	jchar	无符号16个比特
short(短整型)	jshort	有符号16个比特
int(整型)	jint	有符号32个比特
long(长整型)	jlong	有符号64个比特
float(浮点型)	jfloat	32个比特
double(双精度浮点型)	jdouble	64个比特
void(空型)	void	N/A

函数操作

函数	Java 数组类型	本地类型	说明
GetBooleanArrayElements	jbooleanArray	jboolean	ReleaseBooleanArrayElements 释放
GetByteArrayElements	jbyteArray	jbyte	ReleaseByteArrayElements 释放

GetCharArrayElements	jcharArray	jchar	ReleaseShortArrayElements 释放
GetShortArrayElements	jshortArray	jshort	ReleaseBooleanArrayElements 释放
GetIntArrayElements	jintArray	jint	ReleaseIntArrayElements 释放
GetLongArrayElements	jlongArray	jlong	ReleaseLongArrayElements 释放
GetFloatArrayElements	jfloatArray	jfloat	ReleaseFloatArrayElements 释放
GetDoubleArrayElements	jdoubleArray	jdouble	ReleaseDoubleArrayElements 释放
GetObjectArrayElement	自定义对象	object	
SetObjectArrayElement	自定义对象	object	
GetArrayLength			获取数组大小
New<Type>Array			创建一个指定长度的原始型的数组
GetPrimitiveArrayCritical			得到指向原始数据类型内针，该方法可能使垃圾回执行，该方法可能返回数贝，因此必须释放此资源
ReleasePrimitiveArrayCritical			释放指向原始数据类型内容的指针，该方法可能使垃圾回收不能执行，该方法可能返回数组的拷贝，因此必须释放此资源。
NewStringUTF			jstring类型的方法转换
GetStringUTFChars			jstring类型的方法转换
DefineClass			从原始类数据的缓冲区中加载类
FindClass			该函数用于加载本地定义的类。它将搜索由CLASSPATH 环境变量为具有指定名称的类所指定的目录和 zip文件
GetObjectClass			通过对象获取这个类。该函数比较简单，唯一注意的是对象不能为NULL，否则获取的class肯定返回也为NULL
GetSuperclass			获取父类或者说超类 。 如果 clazz 代表类class而非类 object，则该函数返回由 clazz 所指定的类的超类。 如果 clazz 指定类 object 或代表某个接口，则该函数返回NULL
IsAssignableFrom			确定 clazz1 的对象是否可安全地强制转换为clazz2
Throw			抛出 java.lang.Throwable 对象

ThrowNew			利用指定类的消息（由 message 指定）构造异常对象并抛出该异常
ExceptionOccurred			确定是否某个异常正被抛出。在平台相关代码调用 ExceptionClear() 或 Java 代码处理该异常前，异常将始终保持抛出状态
ExceptionDescribe			将异常及堆栈的回溯输出到系统错误报告信道（例如 stderr）。该例程可便利调试操作
ExceptionClear			清除当前抛出的任何异常。如果当前无异常，则此例程不产生任何效果
FatalError			抛出致命错误并且不希望虚拟机进行修复。该函数无返回
NewGlobalRef			创建 obj 参数所引用对象的全局引用。obj 参数既可以是全局引用，也可以是局部引用。全局引用通过调用DeleteGlobalRef来显式撤消。
DeleteGlobalRef			删除 globalRef 所指向的全局引用
DeleteLocalRef			删除 localRef所指向的局部引用
AllocObject			分配新 Java 对象而不调用该对象的任何构造函数。返回该对象的引用。clazz 参数务必不要引用数组类。
getObjectClass			返回对象的类
IsSameObject			测试两个引用是否引用同一 Java 对象
NewString			利用 Unicode 字符数组构造新的 java.lang.String 对象
GetStringLength			返回 Java 字符串的长度（Unicode 字符数）
GetStringChars			返回指向字符串的 Unicode 字符数组的指针。该指针在调用 ReleaseStringchars() 前一直有效
ReleaseStringChars			通知虚拟机平台相关代码无需再访问 chars。参数chars 是一个指针，可通过 GetStringChars() 从 string 获得
NewStringUTF			利用 UTF-8 字符数组构造新 java.lang.String 对象

GetStringUTFLength			以字节为单位返回字符串的 UTF-8 长度
GetStringUTFChars			返回指向字符串的 UTF-8 字符数组的指针。该数组在被 ReleaseStringUTFChars() 释放前将一直有效
ReleaseStringUTFChars			通知虚拟机平台相关代码无需再访问 utf。utf 参数是一个指针，可利用 GetStringUTFChars() 获得
NewObjectArray			构造新的数组，它将保存类 elementClass 中的对象。所有元素初始值均设为 initialElement
Set<PrimitiveType>ArrayRegion			将基本类型数组的某一区域从缓冲区中复制回来的一组函数
GetFieldID			返回类的实例（非静态）性 ID。该域由其名称及签定。访问器函数的 Get<type>Field 及 Set<type>Field系列使用搜索对象域。GetFieldID() 不能用于获取数组的长度域。应GetArrayLength()。
Get<type>Field			该访问器例程系列返回对象的实例（非静态）域的值。要访问的域由通过调用GetFieldID() 而得到的域 ID 指定。
Set<type>Field			该访问器例程系列设置对象的实例（非静态）属性的值。要访问的属性由通过调用 SetFieldID() 而得到的属性 ID指定。
GetStaticFieldID GetStatic<type>Field SetStatic<type>Field			同上，只不过是静态属性。
GetMethodID			返回类或接口实例（非静态）方法的方法 ID。方法可在某个 clazz 的超类中定义，也可从 clazz 继承。该方法由其名称和签名决定。 GetMethodID() 可使未初始化的类初始化。要获得构造函数的方法 ID，应将<init> 作为方法名，同时将void (V) 作为返回类型。

CallVoidMethod			
CallObjectMethod			
CallBooleanMethod			
CallByteMethod			
CallCharMethod			
CallShortMethod			
CallIntMethod			
CallLongMethod			
CallFloatMethod			
CallDoubleMethod			
GetStaticMethodID			调用静态方法
Call<type>Method			
RegisterNatives			向 clazz 参数指定的类注册本地方法。methods 参数将指 JNINativeMethod 结构的 其中包含本地方法的名称 和函数指针。nMethods 参 指定数组中的本地方法数
UnregisterNatives			取消注册类的本地方法。类将返回到链接或注册了本地方前的状态。该函数不应在常规平台相关代码中使用。相反，它可以为某些程序提供一种重新加载和重新链接本地库的途径。

域描述符

域	Java 语言
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double

引用类型则为 L + 该类型类描述符 + 。

数组，其为：[+ 其类型的域描述符 + 。

多维数组则是 n个[+该类型的域描述符 , N代表的是几维数组。

```
[html]

01. String类型的域描述符为 Ljava/lang/String;
02.
03. [ + 其类型的域描述符 + ;
04. int[ ]     其描述符为[I
05. float[ ]   其描述符为[F
06. String[ ]  其描述符为[Ljava/lang/String;
07. Object[ ]类型的域描述符为[Ljava/lang/Object;
08. int  [ ][ ] 其描述符为[[I
09. float[ ][ ] 其描述符为[[F
```

将参数类型的域描述符按照申明顺序放入一对括号中后跟返回值类型的域描述符，规则如下：(参数的域描述符的叠加)返回类型描述符。对于，没有返回值的，用V(表示void型)表示。
举例如下：

```
[html]

01. Java层方法                                                    JNI函数签名
02.                 String test ( )
    /String;
03.                 int f (int i, Object object)                          (ILj
    /Object;)I
04.                 void set (byte[ ] bytes)                              ([B)V
```

JNIEnv与JavaVM

JNIEnv 概念 : 是一个线程相关的结构体, 该结构体代表了 Java 在本线程的运行环境 ;

JNIEnv 与 JavaVM : 注意区分这两个概念;
-- JavaVM : JavaVM 是 Java虚拟机在 JNI 层的代表, JNI 全局只有一个;
-- JNIEnv : JavaVM 在线程中的代表, 每个线程都有一个, JNI 中可能有很多个 JNIEnv;

JNIEnv 作用 :
-- 调用 Java 函数 : JNIEnv 代表 Java 运行环境, 可以使用 JNIEnv 调用 Java 中的代码;
-- 操作 Java 对象 : Java 对象传入 JNI 层就是 Jobject 对象, 需要使用 JNIEnv 来操作这个 Java 对象;

JNIEnv 体系结构

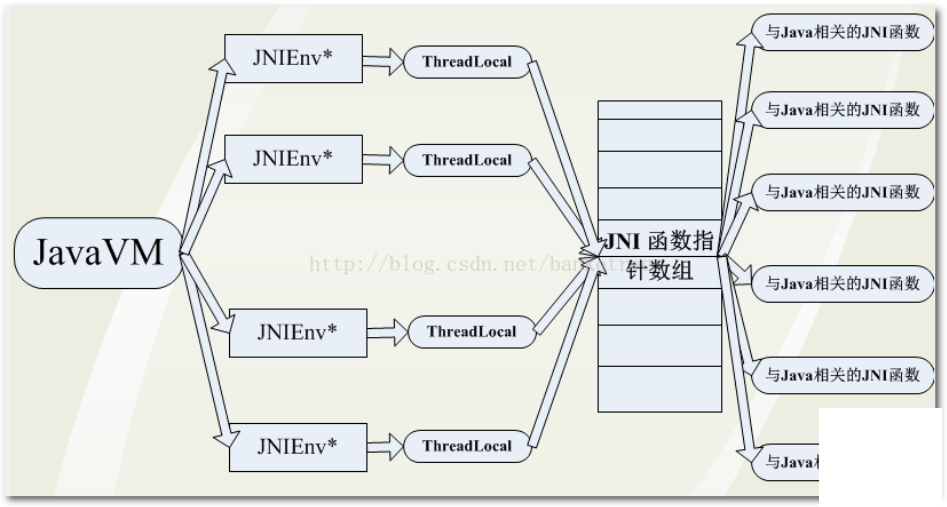
线程相关 : JNIEnv 是线程相关的, 即 在 每个线程中 都有一个 JNIEnv 指针, 每个JNIEnv 都是线程专有的, 其它线程不能使用本线程中的 JNIEnv, 线程 A 不能调用 线程 B 的 JNIEnv;

JNIEnv 不能跨线程 :
-- 当前线程有效 : JNIEnv 只在当前线程有效, JNIEnv 不能在 线程之间进行传递, 在同一个线程中, 多次调用 JNI层方法, 传入的 JNIEnv 是相同的;
-- 本地方法匹配多JNIEnv : 在 Java 层定义的本地方法, 可以在不同的线程调用, 因此 可以接受不同的 JNIEnv;

JNIEnv 结构 : 由上面的代码可以得出, JNIEnv 是一个指针, 指向一个线程相关的结构, 线程相关结构指向 JNI 函数指针 数组, 这个数组中存放了大量的 JNI 函数指针, 这些指针指向了具体的 JNI 函数;

注意：JNIEnv只在当前线程中有效。本地方法不能将JNIEnv从一个线程传递到另一个线程中。相同的 Java 线程

中对本地方法多次调用时，传递给该本地方法的JNIEnv是相同的。但是，一个本地方法可被不同的 Java 线程所调用，因此可以接受不同的 JNIEnv。



UTF-8编码

JNI使用改进的UTF-8字符串来表示不同的字符类型。Java使用UTF-16编码。UTF-8编码主要使用于它的编码用\u000表示为0xc0，而不是通常的0×00。非空ASCII字符改进后的字符串编码中可以用一

错误

JNI不会检查NullPointerException、IllegalArgumentException这样的错误，原因是：导致性能下降。

在绝大多数C的库函数中，很难避免错误发生。
JNI允许用户使用Java异常处理。大部分JNI方法会返回错误代码但本身并不会报出异常。因此，很有必要在代码本身进行处理，将异常抛给Java。在JNI内部，首先会检查调用函数返回的错误代码，之后会调用ExpectOccurred()返回一个错误对象。

```
[html]
01. jthrowable ExceptionOccurred(JNIEnv *env);
02. 例如：一些操作数组的JNI函数不会报错，因此可以调用ArrayIndexOutofBoundsException或
    ArrayStoreExpection方法报告异常。
```

3、JNI函数实战

1、*.so的入口函数

JNI_OnLoad()与JNI_OnUnload()
当Android的VM(Virtual Machine)执行到System.loadLibrary()函数时，首先会去执行C组件里的JNI_OnLoad()函数。它的用途有二：
(1)告诉VM此C组件使用那一个JNI版本。如果你的*.so档没有提供JNI_OnLoad()函数，VM会默认该*.so档是使用最老的JNI 1.1版本。由于新版的JNI做了许多扩充，如果需要使用JNI的新版功能，例如JNI 1.4的java.nio.ByteBuffer,就必须藉由JNI_OnLoad()函数来告知VM。
(2)由于VM执行到System.loadLibrary()函数时，就会立即先呼叫JNI_OnLoad()，所以C组件的开发者可以藉由JNI_OnLoad()来进行C组件内的初期值之设定(Initialization) 。

2、返回值

```
[html]
01. jstring str = env->newStringUTF("HelloJNI");  //直接使用该JNI构造一个jstring对象返回
02.     return str ;
```

```
[html]
01. jobjectArray ret = 0;
02. jsize len = 5;
03. jstring str;
04. string value("hello");
05.
06. ret = (jobjectArray)(env->NewObjectArray(len, env->FindClass("java/lang
    /String"), 0));
07. for(int i = 0; i < len; i++)
08. {
09.     str = env->NewStringUTF(value..c_str());
10.     env->SetObjectArrayElement(ret, i, str);
11. }
12. return ret; 返回数组
```

```
[html]
01. jclass    m_cls   = env->FindClass("com/ldq/ScanResult");
02.
03.    jmethodID m_mid   = env->GetMethodID(m_cls,"<init>","()V");
04.
05.    jfieldID  m_fid_1 = env->GetFieldID(m_cls,"ssid","Ljava/lang/String;");
06.    jfieldID  m_fid_2 = env->GetFieldID(m_cls,"mac","Ljava/lang/String;");
07.    jfieldID  m_fid_3 = env->GetFieldID(m_cls,"level","I");
08.
09.    jobject   m_obj   = env->NewObject(m_cls,m_mid);
10.
11.                        env->SetObjectField(m_obj,m_fid_1,env->NewStringUTF("AP1"));
12.                        env->SetObjectField(m_obj,m_fid_2,env->NewStringUTF("00-11-22-33-
13.                        env->SetIntField(m_obj,m_fid_3,-50);
14.    return m_obj;  返回自定义对象
```

```
[html]
01. jclass list_cls = env->FindClass("Ljava/util/ArrayList;");//获得ArrayList类引用
02.
03.     if(listcls == NULL)
04.     {
05.         cout << "listcls is null \n" ;
06.     }
07.     jmethodID list_costruct = env->GetMethodID(list_cls , "<init>","()V"); //获得得构造函
    数Id
08.
09.     jobject list_obj = env->NewObject(list_cls , list_costruct); //创建一个Arraylist集合对
    象
10.     //或得Arraylist类中的 add()方法ID，其方法原型为： boolean add(Object object) ;
11.     jmethodID list_add  = env->GetMethodID(list_cls,"add","(Ljava/lang
    /Object;)Z");
12.
13.     jclass stu_cls = env->FindClass("Lcom/feixun/jni/Student;");//获得Student类引用
14.     //获得该类型的构造函数  函数名为 <init> 返回类型必须为 void 即 V
15.     jmethodID stu_costruct = env->GetMethodID(stu_cls , "<init>", "(ILjava/lang
    /String;)V");
16.
17.     for(int i = 0 ; i < 3 ; i++)
18.     {
19.         jstring str = env->NewStringUTF("Native");
20.         //通过调用该对象的构造函数来new 一个 Student实例
21.         jobject stu_obj = env->NewObject(stucls , stu_costruct , 10,str);  //构造一个对
    象
22.
23.         env->CallBooleanMethod(list_obj , list_add , stu_obj); //执行Arraylist类实例的add
    方法，添加一个stu对象
```

```
    }

    return list_obj ;   返回对象集合
```

3、操作Java层的类

```
[html]

//获得jfieldID 以及 该字段的初始值
   jfieldID  nameFieldId ;

   jclass cls = env->GetObjectClass(obj);  //获得Java层该对象实例的类引用，即HelloJNI类引用

   nameFieldId = env->GetFieldID(cls , "name" , "Ljava/lang/String;"); //获得属性句柄

   if(nameFieldId == NULL)
   {
       cout << " 没有得到name 的句柄Id \n;" ;
   }
   jstring javaNameStr = (jstring)env->GetObjectField(obj ,nameFieldId);  // 获得该属性的值
   const char * c_javaName = env->GetStringUTFChars(javaNameStr , NULL);  //转换为 char *类型
   string str_name = c_javaName ;
   cout << "the name from java is " << str_name << endl ; //输出显示
   env->ReleaseStringUTFChars(javaNameStr , c_javaName);  //释放局部引用

   //构造一个jString对象
   char * c_ptr_name = "I come from Native" ;

   jstring cName = env->NewStringUTF(c_ptr_name); //构造一个jstring对象

   env->SetObjectField(obj , nameFieldId , cName); // 设置该字段的值
```

4、回调Java层方法

```
[html]

jstring str = NULL;

  jclass clz = env->FindClass("cc/androidos/jni/JniTest");
  //获取clz的构造函数并生成一个对象
  jmethodID ctor = env->GetMethodID(clz, "<init>", "()V");
  jobject obj = env->NewObject(clz, ctor);

  // 如果是数组类型，则在类型前加[,如整形数组int[] intArray,则对应类型为[I,整形数组String[] strArray对应为[Ljava/lang/String;
  jmethodID mid = env->GetMethodID(clz, "sayHelloFromJava", "(Ljava/lang/String;II[I)I");
  if (mid)
  {
      LOGI("mid is get");
      jstring str1 = env->NewStringUTF("I am Native");
      jint index1 = 10;
      jint index2 = 12;
      //env->CallVoidMethod(obj, mid, str1, index1, index2);

      // 数组类型转换 testIntArray能不能不申请内存空间
      jintArray testIntArray = env->NewIntArray(10);
      jint *test = new jint[10];
      for(int i = 0; i < 10; ++i)
      {
          *(test+i) = i + 100;
      }
      env->SetIntArrayRegion(testIntArray, 0, 10, test);


      jint javaIndex = env->CallIntMethod(obj, mid, str1, index1, index2, testIntArray);
      LOGI("javaIndex = %d", javaIndex);
```

```
30.       delete[] test;
31.       test = NULL;
32.   }
```

```
[html]
01. static void event_callback(int eventId,const char* description) {  //主进程回调可以，线程中
回调失败。
02.     if (gEventHandle == NULL)
03.         return;
04.
05.     JNIEnv *env;
06.     bool isAttached = false;
07.
08.     if (myVm->GetEnv((void**) &env, JNI_VERSION_1_2) < 0) { //获取当前的JNIEnv
09.         if (myVm->AttachCurrentThread(&env, NULL) < 0)
10.             return;
11.         isAttached = true;
12.     }
13.
14.     jclass cls = env->GetObjectClass(gEventHandle); //获取类对象
15.     if (!cls) {
16.         LOGE("EventHandler: failed to get class reference");
17.         return;
18.     }
19.
20.     jmethodID methodID = env->GetStaticMethodID(cls, "callbackStatic",
21.         "(ILjava/lang/String;)V");  //静态方法或成员方法
22.     if (methodID) {
23.         jstring content = env->NewStringUTF(description);
24.         env->CallVoidMethod(gEventHandle, methodID,eventId,
25.             content);
26.         env->ReleaseStringUTFChars(content,description);
27.     } else {
28.         LOGE("EventHandler: failed to get the callback method");
29.     }
30.
31.     if (isAttached)
32.         myVm->DetachCurrentThread();
33. }
```

线程中回调

把c/c++中所有线程的创建，由pthread_create函数替换为由Java层的创建线程的函数

AndroidRuntime::createJavaThread。

```
[html]
01. static pthread_t create_thread_callback(const char* name, void (*start)(void *), void* a
02. {
03.     return (pthread_t)AndroidRuntime::createJavaThread(name, start, arg);
04. }
05.
06.
07. static void checkAndClearExceptionFromCallback(JNIEnv* env, const char* methodName) {  /
异常检测和排除
08.     if (env->ExceptionCheck()) {
09.         LOGE("An exception was thrown by callback '%s'.", methodName);
10.         LOGE_EX(env);
11.         env->ExceptionClear();
12.     }
13. }
14.
15. static void receive_callback(unsigned char *buf, int len)  //回调
16. {
17.     int i;
18.     JNIEnv* env = AndroidRuntime::getJNIEnv();
19.     jcharArray array = env->NewCharArray(len);
20.     jchar *pArray ;
```

```

    if(array == NULL){
        LOGE("receive_callback: NewCharArray error.");
        return;
    }

    pArray = (jchar*)calloc(len, sizeof(jchar));
    if(pArray == NULL){
        LOGE("receive_callback: calloc error.");
        return;
    }

    //copy buffer to jchar array
    for(i = 0; i < len; i++)
    {
        *(pArray + i) = *(buf + i);
    }
    //copy buffer to jcharArray
    env->SetCharArrayRegion(array,0,len,pArray);
    //invoke java callback method
    env->CallVoidMethod(mCallbacksObj, method_receive,array,len);
    //release resource
    env->DeleteLocalRef(array);
    free(pArray);
    pArray = NULL;

    checkAndClearExceptionFromCallback(env, __FUNCTION__);
}


public void Receive(char buffer[],int length){  //java层函数
        String msg = new String(buffer);
        msg = "received from jni callback" + msg;
        Log.d("Test", msg);
    }
```

```
[html]
jclass cls = env->GetObjectClass(obj);//获得Java类实例
jmethodID callbackID = env->GetMethodID(cls , "callback" , "(Ljava/lang/String;)V") ;//或得该回调方法句柄

if(callbackID == NULL)
{
     cout << "getMethodId is failed \n" << endl ;
}

jstring native_desc = env->NewStringUTF(" I am Native");

env->CallVoidMethod(obj , callbackID , native_desc); //回调该方法，并且
```

5、传对象到JNI调用

```
[html]
jclass stu_cls = env->GetObjectClass(obj_stu); //或得Student类引用

  if(stu_cls == NULL)
  {
      cout << "GetObjectClass failed \n" ;
  }
  //下面这些函数操作，我们都见过的。O(∩_∩)O~
  jfieldID ageFieldID = env->GetFieldID(stucls,"age","I"); //获得得Student类的属性id
  jfieldID nameFieldID = env->GetFieldID(stucls,"name","Ljava/lang/String;"); // 获得属性ID

  jint age = env->GetIntField(objstu , ageFieldID);  //获得属性值
  jstring name = (jstring)env->GetObjectField(objstu , nameFieldID);//获得属性值
```

```
13.
14.   const char * c_name = env->GetStringUTFChars(name ,NULL);//转换成 char *
15.
16.   string str_name = c_name ;
17.   env->ReleaseStringUTFChars(name,c_name); //释放引用
18.
19.   cout << " at Native age is :" << age << " # name is " << str_name << endl ;
```

6、与C++互转

jbytearray转c++byte数组

```
[html]
01. jbyte * arrayBody = env->GetByteArrayElements(data,0);
02. jsize theArrayLengthJ = env->GetArrayLength(data);
03. BYTE * starter = (BYTE *)arrayBody;
```

jbyteArray 转 c++中的BYTE[]

```
[html]
01. jbyte * olddata = (jbyte*)env->GetByteArrayElements(strIn, 0);
02. jsize  oldsize = env->GetArrayLength(strIn);
03. BYTE* bytearr = (BYTE*)olddata;
04. int len = (int)oldsize;
```

C++中的BYTE[]转jbyteArray

```
[html]
01. jbyte *by = (jbyte*)pData;
02. jbyteArray jarray = env->NewByteArray(nOutSize);
03. env->SetByteArrayRegin(jarray, 0, nOutSize, by);
```

jbyteArray 转 char *

```
[html]
01. char* data = (char*)env->GetByteArrayElements(strIn, 0);
```

char* 转jstring

```
[html]
01. jstring WindowsTojstring(JNIEnv* env, char* str_tmp)
02. {
03.  jstring rtn=0;
04.  int slen = (int)strlen(str_tmp);
05.  unsigned short* buffer=0;
06.  if(slen == 0)
07.  {
08.   rtn = env->NewStringUTF(str_tmp);
09.  }
10.  else
11.  {
12.   int length = MultiByteToWideChar(CP_ACP, 0, (LPCSTR)str_tmp, slen, NULL, 0);
13.   buffer = (unsigned short*)malloc(length*2+1);
14.   if(MultiByteToWideChar(CP_ACP, 0, (LPCSTR)str_tmp, slen, (LPWSTR)buffer, length) > 0)
15.   {
```

```
    rtn = env->NewString((jchar*)buffer, length);
   }
  }
  if(buffer)
  {
   free(buffer);
  }
  return rtn;
}
```

char* jstring互转

```
[html]
JNIEXPORT jstring JNICALL Java_com_explorer_jni_SambaTreeNative_getDetailsBy
  (JNIEnv *env, jobject jobj, jstring pc_server, jstring server_user, jstring server_pas
{
    const char *pc = env->GetStringUTFChars(pc_server, NULL);
    const char *user = env->GetStringUTFChars(server_user, NULL);
    const char *passwd = env->GetStringUTFChars(server_passwd, NULL);
    const char *details = smbtree::getPara(pc, user, passwd);
    jstring jDetails = env->NewStringUTF(details);
    return jDetails;
}
```

4、Android.mk、Application.mk

1、Android.mk

Android.mk文件是GNU Makefile的一小部分，它用来对Android程序进行编译,Android.mk中的变量都是全局的，解析过程会被定义。

一个Android.mk文件可以编译多个模块，模块包括：APK程序、JAVA库、C\C++应用程序、C\C++静态库、C\C++共享库。

简单实例如下：

```
[html]
LOCAL_PATH := $(call my-dir)  #表示是当前文件的路径
include $(CLEAR_VARS)        #指定让GNU MAKEFILE该脚本为你清除许多 LOCAL_XXX 变量
LOCAL_MODULE:= helloworld    #标识你在 Android.mk 文件中描述的每个模块
MY_SOURCES := foo.c          #自定义变量
ifneq ($(MY_CONFIG_BAR),)
 MY_SOURCES += bar.c
endif
LOCAL_SRC_FILES += $(MY_SOURCES)    #包含将要编译打包进模块中的 C 或 C++源代码文件
include $(BUILD_SHARED_LIBRARY) #根据LOCAL_XXX系列变量中的值，来编译生成共享库（动态链接库）
```

GNU Make系统变量

变量	描述
CLEAR_VARS	指向一个编译脚本，几乎所有未定义的 LOCAL_XXX 变量都在"Module-description"节中列出。必须在开始一个新模块之前包含这个脚本：include$(CLEAR_VARS)，用于重置除LOCAL_PATH变量外的，所有LOCAL_XXX系列变量。

BUILD_SHARED_LIBRARY	指向编译脚本，根据所有的在 LOCAL_XXX 变量把列出的源代码文件编译成一个共享库。
BUILD_STATIC_LIBRARY	一个 BUILD_SHARED_LIBRARY 变量用于编译一个静态库。静态库不会复制到的APK包中，但是能够用于编译共享库。
TARGET_ARCH	目标 CPU平台的名字，和 android 开放源码中指定的那样。如果是arm，表示要生成 ARM 兼容的指令，与 CPU架构的修订版无关。
TARGET_PLATFORM	Android.mk 解析的时候，目标 Android 平台的名字.详情可参考/development/ndk/docs/stable- apis.txt.
TARGET_ARCH_ABI	支持目标平台
TARGET_ABI	目标平台和 ABI 的组合，它事实上被定义成$(TARGET_PLATFORM)-$(TARGET_ARCH_ABI) ，在想要在真实的设备中针对一个特别的目标系统进行测试时，会有用。在默认的情况下，它会是'android-3-arm'。

模块描述变量

变量	述
LOCAL_PATH	这个变量用于给出当前文件的路径。必须在 Android.n 定义，可以这样使用：LOCAL_PATH := $(call my-dir) 这个变量不会被$(CLEAR_VARS)清除，因此每个 Android.mk 只需要定义一次(即使在一个文件中定义了几个模块的情况下)。
LOCAL_MODULE	这是模块的名字，它必须是唯一的，而且不能包含空格。必须在包含任一的$(BUILD_XXXX)脚本之前定义它。模块的名字决定了生成文件的名字。例如，如果一个一个共享库模块的名字是，那么生成文件的名字就是 lib.so。但是，在的 NDK 生成文件中(或者 Android.mk 或者 Application.mk)，应该只涉及(引用)有正常名字的其他模块。
LOCAL_SRC_FILES	这是要编译的源代码文件列表。只要列出要传递给编译器的文件，因为编译系统自动计算依赖。注意源代码文件名称都是相对于 LOCAL_PATH的，你可以使用路径部分。
LOCAL_CPP_EXTENSION	这是一个可选变量， 用来指定C++代码文件的扩展名，默认是'.cpp',但是可以改变它。
LOCAL_C_INCLUDES	可选变量，表示头文件的搜索路径。
LOCAL_CFLAGS	可选的编译器选项，在编译 C 代码文件的时候使用。
LOCAL_CXXFLAGS	与 LOCAL_CFLAGS同理，针对 C++源文件。
LOCAL_CPPFLAGS	与 LOCAL_CFLAGS同理，但是对 C 和 C++ source files都适用。
LOCAL_STATIC_LIBRARIES	表示该模块需要使用哪些静态库，以便在编译时进行链接。
LOCAL_SHARED_LIBRARIES	表示模块在运行时要依赖的共享库（动态库），在链接时就需要，以便在生成文件时嵌入其相应的信息。注意：它不会附加列出的模块到编译图，也就是仍然需要在Application.mk 中把

	它们添加到程序要求的模块中。
LOCAL_LDLIBS	编译模块时要使用的附加的链接器选项。这对于使用'-l'前缀传递指定库的名字是有用的。
LOCAL_ALLOW_UNDEFINED_SYMBOLS	默认情况下， 在试图编译一个共享库时，任何未定义的引用将导致一个"未定义的符号"错误。
LOCAL_ARM_MODE	默认情况下， arm目标二进制会以 thumb 的形式生成(16 位)，你可以通过设置这个变量为 arm如果你希望你的 module 是以 32 位指令的形式。
LOCAL_MODULE_PATH 和 LOCAL_UNSTRIPPED_PATH	在 Android.mk 文件中， 还可以用LOCAL_MODULE_PATH 和 LOCAL_UNSTRIPPED_PATH指定最后的目标安装路径. 不同的文件系统路径用以下的宏进行选择： TARGET_ROOT_OUT：表示根文件系统。 TARGET_OUT：表示 system文件系统。 TARGET_OUT_DATA：表示 data文件系统。 用法如：LOCAL_MODULE_PATH :=$(TARGET_ROOT_OUT) 至于LOCAL_MODULE_PATH 和 LOCAL_UNSTRIPPED_PATH的区别，暂时还不清楚

GNU Make 功能宏

变量	描述
my-dir	返回当前 Android.mk 所在的目录的路径，相对于 NDK 编译系统的顶层。
all-subdir-makefiles	返回一个位于当前'my-dir'路径的子目录中的所有Android.mk的列表。
this-makefile	返回当前Makefile 的路径(即这个函数调用的地方)
parent-makefile	返回调用树中父 Makefile 路径。即包含当前Makefile的 Makefile 路径。
grand-parent-makefile	返回调用树中父Makefile的父Makefile的路径

范例：

2、

编译一个简单的APK

```
[html]
01. LOCAL_PATH := $(call my-dir)
02. include $(CLEAR_VARS)
03. # Build all java files in the java subdirectory
04. LOCAL_SRC_FILES := $(call all-subdir-java-files)
05. # Name of the APK to build
06. LOCAL_PACKAGE_NAME := LocalPackage
07. # Tell it to build an APK
```

```
08. include $(BUILD_PACKAGE)
```

编译一个依赖静态.jar文件的APK

```
[html]
01. LOCAL_PATH := $(call my-dir)
02.   include $(CLEAR_VARS)
03.   # List of static libraries to include in the package
04.   LOCAL_STATIC_JAVA_LIBRARIES := static-library
05.   # Build all java files in the java subdirectory
06.   LOCAL_SRC_FILES := $(call all-subdir-java-files)
07.   # Name of the APK to build
08.   LOCAL_PACKAGE_NAME := LocalPackage
09.   # Tell it to build an APK
10.   include $(BUILD_PACKAGE)
11.  注: LOCAL_STATIC_JAVA_LIBRARIES 后面应是你的APK程序所需要的JAVA库的JAR文件名。
```

编译一个需要platform key签名的APK

```
[html]
01. LOCAL_PATH := $(call my-dir)
02.   include $(CLEAR_VARS)
03.   # Build all java files in the java subdirectory
04.   LOCAL_SRC_FILES := $(call all-subdir-java-files)
05.   # Name of the APK to build
06.   LOCAL_PACKAGE_NAME := LocalPackage
07.   LOCAL_CERTIFICATE := platform
08.   # Tell it to build an APK
09.   include $(BUILD_PACKAGE)
10.  注: LOCAL_CERTIFICATE 后面应该是签名文件的文件名
```

编译一个需要特殊vendor key签名的APK

```
[html]
01. LOCAL_PATH := $(call my-dir)
02.  include $(CLEAR_VARS)
03.  # Build all java files in the java subdirectory
04.  LOCAL_SRC_FILES := $(call all-subdir-java-files)
05.  # Name of the APK to build
06.  LOCAL_PACKAGE_NAME := LocalPackage
07.  LOCAL_CERTIFICATE := vendor/example/certs/app
08.  # Tell it to build an APK
09.  include $(BUILD_PACKAGE)
```

装载一个普通的第三方APK

```
[html]
01. LOCAL_PATH := $(call my-dir)
02.  include $(CLEAR_VARS)
03.  # Module name should match apk name to be installed.
04.  LOCAL_MODULE := LocalModuleName
05.  LOCAL_SRC_FILES := $(LOCAL_MODULE).apk
06.  LOCAL_MODULE_CLASS := APPS
07.  LOCAL_MODULE_SUFFIX := $(COMMON_ANDROID_PACKAGE_SUFFIX)
08.  LOCAL_CERTIFICATE := platform
09.  include $(BUILD_PREBUILT)
```

装载需要.so（动态库）的第三方apk

```
[html]
01. LOCAL_PATH := $(my-dir)
02. include $(CLEAR_VARS)
03. LOCAL_MODULE := baiduinput_android_v1.1_1000e
04. LOCAL_SRC_FILES := $(LOCAL_MODULE).apk
05. LOCAL_MODULE_CLASS := APPS
06. LOCAL_MODULE_SUFFIX := $(COMMON_ANDROID_PACKAGE_SUFFIX)
07. LOCAL_CERTIFICATE := platform
08. include $(BUILD_PREBUILT)
09.
10. ##################################################################
11. ####### copy the library to /system/lib ########################
12. ##################################################################
13. include $(CLEAR_VARS)
14. LOCAL_MODULE := libinputcore.so
15. LOCAL_MODULE_CLASS := SHARED_LIBRARIES
16. LOCAL_MODULE_PATH := $(TARGET_OUT_SHARED_LIBRARIES)
17. LOCAL_SRC_FILES := lib/$(LOCAL_MODULE)
18. OVERRIDE_BUILD_MODULE_PATH := $(TARGET_OUT_INTERMEDIATE_LIBRARIES)
19. include $(BUILD_PREBUILT)
```

编译一个静态java库

```
[html]
01.   LOCAL_PATH := $(call my-dir)
02.   include $(CLEAR_VARS)
03.   # Build all java files in the java subdirectory
04.   LOCAL_SRC_FILES := $(call all-subdir-java-files)
05.   # Any libraries that this library depends on
06.   LOCAL_JAVA_LIBRARIES := android.test.runner
07.   # The name of the jar file to create
08.   LOCAL_MODULE := sample
09.   # Build a static jar file.
10.   include $(BUILD_STATIC_JAVA_LIBRARY)
11. 注：LOCAL_JAVA_LIBRARIES表示生成的java库的jar文件名。
```

编译C/C++应用程序模板

```
[html]
01. LOCAL_PATH := $(call my-dir)
02. #include $(CLEAR_VARS)
03. LOCAL_SRC_FILES := main.c
04. LOCAL_MODULE := test_exe
05. #LOCAL_C_INCLUDES :=
06. #LOCAL_STATIC_LIBRARIES :=
07. #LOCAL_SHARED_LIBRARIES :=
08. include $(BUILD_EXECUTABLE)
09. 注：‘:=’是赋值的意思，'+='是追加的意思，‘$’表示引用某变量的值
10. LOCAL_SRC_FILES中加入源文件路径，LOCAL_C_INCLUDES中加入需要的头文件搜索路径
11. LOCAL_STATIC_LIBRARIES 加入所需要链接的静态库(*.a)的名称，
12. LOCAL_SHARED_LIBRARIES 中加入所需要链接的动态库(*.so)的名称，
13. LOCAL_MODULE表示模块最终的名称，BUILD_EXECUTABLE 表示以一个可执行程序的方式进行编译。
14. （4）编译C\C++静态库
15. LOCAL_PATH := $(call my-dir)
16. include $(CLEAR_VARS)
17. LOCAL_SRC_FILES := \
18.  helloworld.c
19. LOCAL_MODULE:= libtest_static
20.  #LOCAL_C_INCLUDES :=
21. #LOCAL_STATIC_LIBRARIES :=
22. #LOCAL_SHARED_LIBRARIES :=
23. include $(BUILD_STATIC_LIBRARY)
```

```
24. 和上面相似，BUILD_STATIC_LIBRARY 表示编译一个静态库。
```

编译C\C++动态库的模板

```
[html]
01. LOCAL_PATH := $(call my-dir)
02. include $(CLEAR_VARS)
03. LOCAL_SRC_FILES := helloworld.c
04. LOCAL_MODULE := libtest_shared
05. TARGET_PRELINK_MODULES := false
06. #LOCAL_C_INCLUDES :=
07. #LOCAL_STATIC_LIBRARIES :=
08. #LOCAL_SHARED_LIBRARIES :=
09. include $(BUILD_SHARED_LIBRARY)
10. 和上面相似，BUILD_SHARED_LIBRARY 表示编译一个共享库。
11. 以上三者的生成结果分别在如下目录中，generic 依具体 target 会变:
12. out/target/product/generic/obj/APPS
13. out/target/product/generic/obj/JAVA_LIBRARIES
14. out/target/product/generic/obj/EXECUTABLE
15. out/target/product/generic/obj/STATIC_LIBRARY
16. out/target/product/generic/obj/SHARED_LIBRARY
17. 每个模块的目标文件夹分别为:
18. 1)APK程序: XXX_intermediates
19. 2)JAVA库程序: XXX_intermediates
20. 这里的XXX
21.  3)C\C++可执行程序: XXX_intermediates
22.  4)C\C++静态库:  XXX_static_intermediates
23.  5)C\C++动态库:  XXX_shared_intermediates
```

实例:

```
[html]
01. LOCAL_PATH := $(call my-dir)  #项目地址
02. include $(CLEAR_VARS)       #清除变量
03.
04. LOCAL_MODULE    := libvlcjni    #库
05.
06. #源文件
07. LOCAL_SRC_FILES := libvlcjni.c libvlcjni-util.c libvlcjni-track.c libvlcjni-
    medialist.c aout.c vout.c libvlcjni-equalizer.c native_crash_handler.c
08. LOCAL_SRC_FILES += thumbnailer.c pthread-condattr.c pthread-rwlocks.c pthread-
    once.c eventfd.c sem.c
09. LOCAL_SRC_FILES += pipe2.c
10. LOCAL_SRC_FILES += wchar/wcpcpy.c
11. LOCAL_SRC_FILES += wchar/wcpncpy.c
12. LOCAL_SRC_FILES += wchar/wcscasecmp.c
13. LOCAL_SRC_FILES += wchar/wcscat.c
14. LOCAL_SRC_FILES += wchar/wcschr.c
15. LOCAL_SRC_FILES += wchar/wcscmp.c
16. LOCAL_SRC_FILES += wchar/wcscoll.c
17. LOCAL_SRC_FILES += wchar/wcscpy.c
18. LOCAL_SRC_FILES += wchar/wcscspn.c
19. LOCAL_SRC_FILES += wchar/wcsdup.c
20. LOCAL_SRC_FILES += wchar/wcslcat.c
21. LOCAL_SRC_FILES += wchar/wcslcpy.c
22. LOCAL_SRC_FILES += wchar/wcslen.c
23. LOCAL_SRC_FILES += wchar/wcsncasecmp.c
24. LOCAL_SRC_FILES += wchar/wcsncat.c
25. LOCAL_SRC_FILES += wchar/wcsncmp.c
26. LOCAL_SRC_FILES += wchar/wcsncpy.c
27. LOCAL_SRC_FILES += wchar/wcsnlen.c
28. LOCAL_SRC_FILES += wchar/wcspbrk.c
29. LOCAL_SRC_FILES += wchar/wcsrchr.c
30. LOCAL_SRC_FILES += wchar/wcsspn.c
```

```
LOCAL_SRC_FILES += wchar/wcsstr.c
LOCAL_SRC_FILES += wchar/wcstok.c
LOCAL_SRC_FILES += wchar/wcswidth.c
LOCAL_SRC_FILES += wchar/wcsxfrm.c
LOCAL_SRC_FILES += wchar/wmemchr.c
LOCAL_SRC_FILES += wchar/wmemcmp.c
LOCAL_SRC_FILES += wchar/wmemcpy.c
LOCAL_SRC_FILES += wchar/wmemmove.c
LOCAL_SRC_FILES += wchar/wmemset.c


LOCAL_C_INCLUDES := $(VLC_SRC_DIR)/include  #包含头

ARCH=$(ANDROID_ABI) #变量 平台

CPP_STATIC=$(ANDROID_NDK)/sources/cxx-stl/gnu-libstdc++$(CXXSTL)/libs/$(ARCH)
/libgnustl_static.a #应用静态库

LOCAL_CFLAGS := -std=gnu99  #编译器标识
ifeq ($(ARCH), armeabi)
    LOCAL_CFLAGS += -DHAVE_ARMEABI
    # Needed by ARMv6 Thumb1 (the System Control coprocessor/CP15 is mandatory on ARMv6)
    # On newer ARM architectures we can use Thumb2
    LOCAL_ARM_MODE := arm
endif
ifeq ($(ARCH), armeabi-v7a)
    LOCAL_CFLAGS += -DHAVE_ARMEABI_V7A
endif
LOCAL_LDLIBS := -L$(VLC_CONTRIB)/lib \  #使用本地库
    $(VLC_MODULES) \
    $(VLC_BUILD_DIR)/lib/.libs/libvlc.a \
    $(VLC_BUILD_DIR)/src/.libs/libvlccore.a \
    $(VLC_BUILD_DIR)/compat/.libs/libcompat.a \
    -ldl -lz -lm -llog \
    -ldvbpsi -lebml -lmatroska -ltag \
    -logg -lFLAC -ltheora -lvorbis \
    -lmpeg2 -la52 \
    -lavformat -lavcodec -lswscale -lavutil -lpostproc -lgsm -lopenjpeg \
    -lliveMedia -lUsageEnvironment -lBasicUsageEnvironment -lgroupsock \
    -lspeex -lspeexdsp \
    -lxml2 -lpng -lgnutls -lgcrypt -lgpg-error \
    -lnettle -lhogweed -lgmp \
    -lfreetype -liconv -lass -lfribidi -lopus \
    -lEGL -lGLESv2 -ljpeg \
    -ldvdnav -ldvdread -ldvdcss \
    $(CPP_STATIC)

include $(BUILD_SHARED_LIBRARY) #编译成动态库


include $(CLEAR_VARS)   #清除变量

LOCAL_MODULE     := libiomx-gingerbread
LOCAL_SRC_FILES  := ../$(VLC_SRC_DIR)/modules/codec/omxil/iomx.cpp
LOCAL_C_INCLUDES := $(VLC_SRC_DIR)/modules/codec
/omxil $(ANDROID_SYS_HEADERS_GINGERBREAD)/frameworks
/base/include $(ANDROID_SYS_HEADERS_GINGERBREAD)/system/core/include
LOCAL_CFLAGS     := -Wno-psabi
LOCAL_LDLIBS     := -L$(ANDROID_LIBS) -lgcc -lstagefright -lmedia -lutils -lbinder

include $(BUILD_SHARED_LIBRARY)

include $(CLEAR_VARS)

LOCAL_MODULE     := libiomx-hc
LOCAL_SRC_FILES  := ../$(VLC_SRC_DIR)/modules/codec/omxil/iomx.cpp
LOCAL_C_INCLUDES := $(VLC_SRC_DIR)/modules/codec/omxil $(ANDROID_SYS_HEADERS_HC)
/frameworks/base/include $(ANDROID_SYS_HEADERS_HC)/frameworks/base/native
/include $(ANDROID_SYS_HEADERS_HC)/system/core/include $(ANDROID_SYS_HEADERS_HC)
/hardware/libhardware/include
LOCAL_CFLAGS     := -Wno-psabi
```

```
96.  LOCAL_LDLIBS     := -L$(ANDROID_LIBS) -lgcc -lstagefright -lmedia -lutils -lbinder
97.
98.  include $(BUILD_SHARED_LIBRARY)
99.
100. include $(CLEAR_VARS)
101.
102. LOCAL_MODULE     := libiomx-ics
103. LOCAL_SRC_FILES  := ../$(VLC_SRC_DIR)/modules/codec/omxil/iomx.cpp
104. LOCAL_C_INCLUDES := $(VLC_SRC_DIR)/modules/codec/omxil $(ANDROID_SYS_HEADERS_ICS)
     /frameworks/base/include $(ANDROID_SYS_HEADERS_ICS)/frameworks/base/native
     /include $(ANDROID_SYS_HEADERS_ICS)/system/core/include $(ANDROID_SYS_HEADERS_ICS)
     /hardware/libhardware/include
105. LOCAL_CFLAGS     := -Wno-psabi
106. LOCAL_LDLIBS     := -L$(ANDROID_LIBS) -lgcc -lstagefright -lmedia -lutils -lbinder
107.
108. include $(BUILD_SHARED_LIBRARY)
```

2、Application.mk

Application.mk目的是描述在你的应用程序中所需要的模块(即静态库或动态库)。

变量	描述
APP_PROJECT_PATH	这个变量是强制性的，并且会给出应用程序工程的根目录的一个绝对路径。
APP_MODULES	这个变量是可选的，如果没有定义，NDK将由在Android.mk中声明的默认的模块编译，并且包含所有的子文件（makefile文件）如果APP_MODULES定义了，它不许是一个空格分隔的模块列表，这个模块名字被定义在Android.mk文件中的LOCAL_MODULE中。
APP_OPTIM	这个变量是可选的，用来义“release”或"debug"。在编译您的应用程序模块的时候，可以用来改变优先级。
APP_CFLAGS	当编译模块中有任何C文件或者C++文件的时候，C编译器的信号就会被发出。
APP_CXXFLAGS	APP_CPPFLAGS的别名，已经考虑在将在未来的版本中废除了
APP_CPPFLAGS	当编译的只有C++源文件的时候，可以通过这个C++编译器来设置
APP_BUILD_SCRIPT	默认情况下，NDK编译系统会 在$(APP_PROJECT_PATH)/jni目录下寻找名为 Android.mk文件： $(APP_PROJECT_PATH)/jni/Android.mk
APP_ABI	默认情况下，NDK的编译系统回味"armeabi"ABI生成机器代码。
APP_STL	默认情况下，NDK的编译系统为最小的C++运行时库（/system/lib/libstdc++.so）提供C++头文件。然而，NDK的C++的实现，可以让你使用或着链接在自己的应用程序中。 例如： APP_STL := stlport_static --> static STLport library APP_STL := stlport_shared --> shared STLport library APP_STL := system --> default C++ runtime

	library

实例：

```
[html]
01. APP_OPTIM := release   //调试版还是发行版
02. APP_PLATFORM := android-8  //平台
03. APP_STL := gnustl_static  //C++运行时库
04. APP_CPPFLAGS += -frtti      //编译标识
05. APP_CPPFLAGS += -fexceptions  //编译标识 异常
06. APP_CPPFLAGS += -DANDROID   //编译标识
07. APP_MODULES := test     //静态模块
```

```
[html]
01.
```

```
[html]
01.
```

JNI内存泄漏

JAVA 编程中的内存泄漏，从泄漏的内存位置角度可以分为两种：JVM 中 Java Heap 的内存泄漏；[illegible] native memory 的内存泄漏。

Java Heap 的内存泄漏：

Java 对象存储在 JVM 进程空间中的 Java Heap 中，Java Heap 可以在 JVM 运行过程中动态变化。如果 Java 对象越来越多，占据 Java Heap 的空间也越来越大，JVM 会在运行时扩充 Java Heap 的容量。如果 Java Heap 容量扩充到上限，并且在 GC 后仍然没有足够空间分配新的 Java 对象，便会抛出 out of memory 异常，导致 JVM 进程崩溃。
Java Heap 中 out of memory 异常的出现有两种原因①程序过于庞大，致使过多 Java 对象的同时存在；②程序编写的错误导致 Java Heap 内存泄漏。

JVM 中 **native memory** 的内存泄漏

从操作系统角度看，JVM 在运行时和其它进程没有本质区别。在系统级别上，它们具有同样的调度机制，同样的内存分配方式，同样的内存格局。
JVM 进程空间中，Java Heap 以外的内存空间称为 JVM 的 native memory。进程的很多资源都是存储在 JVM 的 native memory 中，例如载入的代码映像，线程的堆栈，线程的管理控制块，JVM 的静态数据、全局数据等等。也包括 JNI 程序中 native code 分配到的资源。
在 JVM 运行中，多数进程资源从 native memory 中动态分配。当越来越多的资源在 native memory 中分配，占据越来越多 native memory 空间并且达到 native memory 上限时，JVM 会抛出异常，使 JVM 进程异常退出。而此时 Java Heap 往往还没有达到上限。
多种原因可能导致 JVM 的 native memory 内存泄漏。
例如:
JVM 在运行中过多的线程被创建，并且在同时运行。
JVM 为线程分配的资源就可能耗尽 native memory 的容量。
JNI 编程错误也可能导致 native memory 的内存泄漏。

Native Code 本身的内存泄漏

JNI 编程首先是一门具体的编程语言，或者 C 语言，或者 C++，或者汇编，或者其它 native 的编程语言。每门编程语言环境都实现了自身的内存管理机制。因此，JNI 程序开发者要遵循 native 语言本身的内存管理机制，避免造成内存泄漏。以 C 语言为例，当用 malloc() 在进程堆中动态分配内存时，JNI 程序在使用完后，应当调用 free() 将内存释放。总之，所有在 native 语言编程中应当注意的内存泄漏规则，在 JNI 编程中依然适应。
Native 语言本身引入的内存泄漏会造成 native memory 的内存，严重情况下会造成 native memory 的 out of memory。

Global Reference 引入的内存泄漏

JNI 编程还要同时遵循 JNI 的规范标准，JVM 附加了 JNI 编程特有的内存管理机制。
JNI 中的 Local Reference 只在 native method 执行时存在，当 native method 执行完后自动失效。这种自动失效，使得对 Local Reference 的使用相对简单，native method 执行完后，它们所引用的 Java 对象的 reference count 会相应减 1。不会造成 Java Heap 中 Java 对象的内存泄漏。
而 Global Reference 对 Java 对象的引用一直有效，因此它们引用的 Java 对象会一直存在 Java Heap 中。程序员在使用 Global Reference 时，需要仔细维护对 Global Reference 的使用。如果一定要使用 Global Reference，务必确保在不用的时候删除。就像在 C 语言中，调用 malloc() 动态分配一块内存之后，调用 free() 释放一样。否则，Global Reference 引用的 Java 对象将永远停留在 Java Heap 中，造成 Java Heap 的内存泄漏

LocalReference 的深入理解

Local Reference 在 native method 执行完成后，会自动被释放，似乎不会造成任何的内存泄漏。但

泄漏实例1：创建大量的 JNI Local Reference

```
[html]
Java 代码部分
 class TestLocalReference {
 private native void nativeMethod(int i);
 public static void main(String args[]) {
        TestLocalReference c = new TestLocalReference();
        //call the jni native method
        c.nativeMethod(1000000);
 }
 static {
 //load the jni library
 System.loadLibrary("StaticMethodCall");
 }
 }


 JNI 代码，nativeMethod(int i) 的 C 语言实现
 #include<stdio.h>
 #include<jni.h>
 #include"TestLocalReference.h"
 JNIEXPORT void JNICALL Java_TestLocalReference_nativeMethod
 (JNIEnv * env, jobject obj, jint count)
 {
 jint i = 0;
 jstring str;


 for(; i<count; i++)
        str = (*env)->NewStringUTF(env, "0");
 }
运行结果
 JVMCI161: FATAL ERROR in native method: Out of memory when expanding
 local ref table beyond capacity
 at TestLocalReference.nativeMethod(Native Method)
 at TestLocalReference.main(TestLocalReference.java:9)
```

泄漏实例2：建立一个 String 对象，返回给调用函数。

```
[html]
JNI 代码，nativeMethod(int i) 的 C 语言实现
 #include<stdio.h>
 #include<jni.h>
 #include"TestLocalReference.h"
 jstring CreateStringUTF(JNIEnv * env)
 {
 return (*env)->NewStringUTF(env, "0");
 }
 JNIEXPORT void JNICALL Java_TestLocalReference_nativeMethod
 (JNIEnv * env, jobject obj, jint count)
 {
 jint i = 0;
 for(; i<count; i++)
 {
         str = CreateStringUTF(env);
 }
 }
运行结果
 JVMCI161: FATAL ERROR in native method: Out of memory when expanding local
 table beyond  capacity
 at TestLocalReference.nativeMethod(Native Method)
 at TestLocalReference.main(TestLocalReference.java:9)
```

编译问题：SLES/OpenSLES.h: No such file or directory

解决方法：ndk-build TARGET_PLATFORM=android-9

编译断点问题：有没有好用的断点工具

解决方法：visualGDB 神器

上一篇 编程之路

下一篇 Android RakNet 系列之一 项目介绍

C语言及程序设计提高

查看评论

1楼 cheyiliu 2015-01-19 18:52发表

mark

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker C
VPN Spark ERP IE10 Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery
BI HTML5 Spring Apache .NET API HTML SDK IIS Fedora XML LBS
Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack
FTC coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo
Compuware 大数据 aptech Perl Tornado Ruby Hibernate ThinkPHP HBase Pure Solr
Angular Cloud Foundry Redis Scala Django Bootstrap