**JNI编程**

**文档修改记录**

|  |  |  |  |
| --- | --- | --- | --- |
| **版号** | **修改日期** | **修改内容** | **修改人** |
| 01 | 2017-04-18 | JNI编程文档的编写 | 刘祚亮 |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**[一、 简介](#_Toc2821)** [3](#_Toc2821)

[1.1](#_Toc2239)[JNI角色 3](#_Toc2239)

[1.2 JNI开发环境 3](#_Toc30846)

[二、](#_Toc5160) **[基本概念](#_Toc5160)** [5](#_Toc5160)

[2.1 jni组织结构 5](#_Toc20327)

**[三、 基本实现](#_Toc17961)** [9](#_Toc17961)

[3.1 Java端 9](#_Toc11990)

[3.2 C/C++端 10](#_Toc31395)

[3.3 Android.mk 11](#_Toc32683)

[3.4 Application.mk 11](#_Toc2422)

[3.5 编译生成libdemo.so 动态库 11](#_Toc27355)

**[四、 常见函数](#_Toc11789)** [11](#_Toc11789)

**[五、 字符串操作](#_Toc17223)** [17](#_Toc17223)

**[六、 参数传递](#_Toc26716)** [21](#_Toc26716)

[6.1 基本类型传递 21](#_Toc1855)

[6.2 String参数的传递 21](#_Toc2782)

[6.3 数组类型传递 22](#_Toc32538)

[6.4 二维数组和字符串数组 22](#_Toc1352)

**[七、 调用Java](#_Toc27019)** [23](#_Toc27019)

[7.1 JNIEnv对象 23](#_Toc14861)

[7.2 获取jclass 23](#_Toc27543)

[7.3 本地代码访问Java类中的属性与方法 24](#_Toc23795)

**[八、 异常处理](#_Toc26979)** [25](#_Toc26979)

[8.1 现状 25](#_Toc2876)

[8.2 异常检查 25](#_Toc22054)

[8.3 异常处理 26](#_Toc4662)

**[九、 内存泄漏](#_Toc15627)** [27](#_Toc15627)

[9.1 常见内存泄漏 27](#_Toc16176)

[9.2 深层分析Local Reference 29](#_Toc4551)

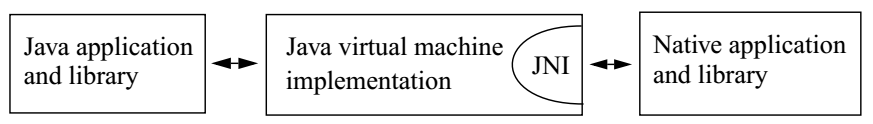
[9.3 控制Local Reference生命周期 30](#_Toc11295)

[9.4 总结 30](#_Toc3882)

1. **简介**

JNI是本地编程接口。使得在Java虚拟机（VM）内部运行的Java代码能够与用其它编程语言（如C/C++）编写的应用程序和库进行互操作。

1.1JNI角色

****

1.2 JNI开发环境

Jdk + Eclipse +ndk(r10d)

环境配置如下图：

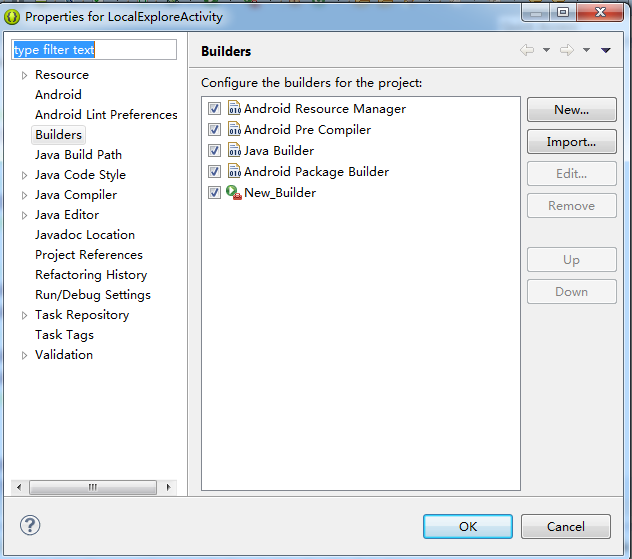


图 1

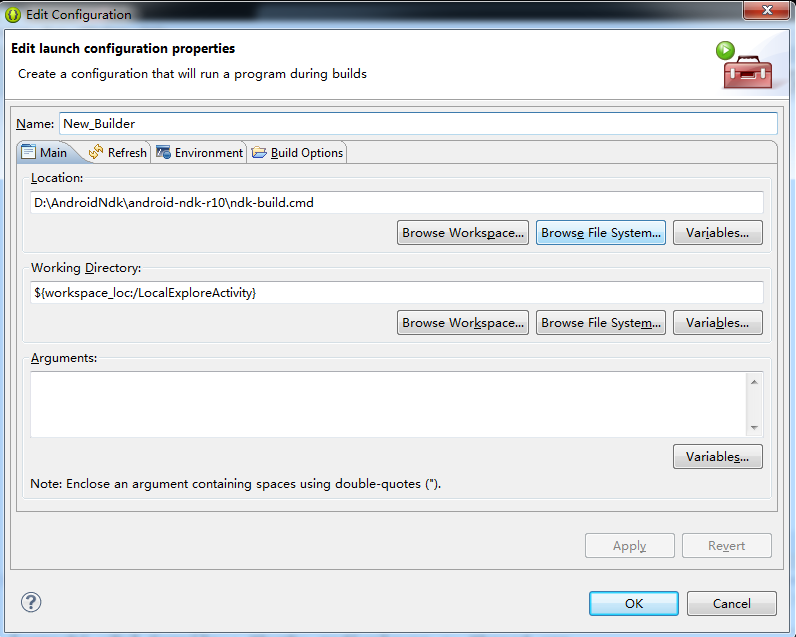
****

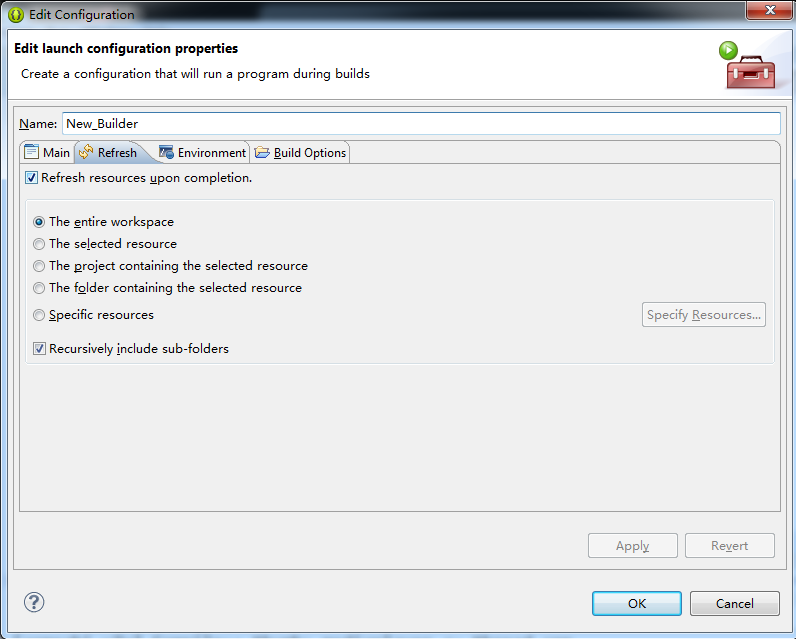
图 2 ****

图 3

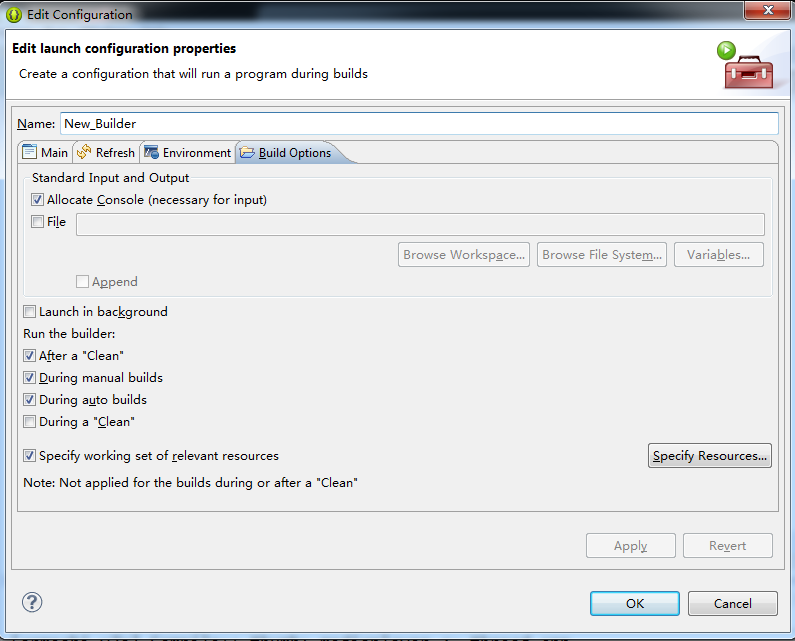
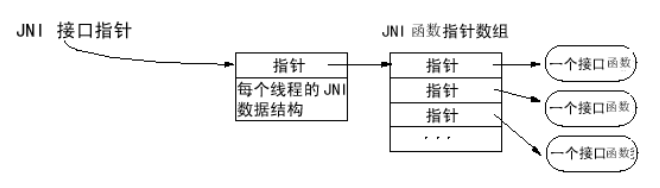
****

图 4

1. **基本概念**

2.1 **jni组织结构**



2.2 **数据类型**

2.2.1 void

java的void与JNI的void是一致的。

2.2.2 基本数据类型

|  |  |  |  |
| --- | --- | --- | --- |
| Java类型 | JNI类型 | C/C++类型 | 描述 |
| boolean | jboolean | unsigned char 或 uint8\_t | C/C++8位整型 |
| byte | jbyte | signec char 或 int8\_t | C/C++有符号的8位整数 |
| char | jchar | unsigned short 或 uint16\_t | C/C++无符号的16位整型 |
| short | jshort | short 或 int16\_t | C/C++有符号的16位整型 |
| int | jint | int 或 int32\_t | C/C++有符号的32位整型 |
| long | jlong | long long 或 int62\_t | C/C++有符号的64位整型 |
| float | jfloat | float | C/C++32位浮点数 |
| double | jdouble | double | C/C++64位浮点数 |

表 1

2.2.3 引用类型

|  |  |  |
| --- | --- | --- |
| Java类型 | JNI类型 | 描述 |
| Object | jobject | 任意Java对象，或者没有对应的java类型的对象 |
| Class | jclass | Class对象 |
| String | jstring | 字符串对象 |

表 2

2.2.4 数组类型

|  |  |  |
| --- | --- | --- |
| Java类型 | Jni类型 | 描述 |
| boolean[] | jbooleanArray | 布尔型数组 |
| byte[] | jbyteArray | 比特型数组 |
| char[] | jcharArray | 字符型数组 |
| short[] | jshortArray | 短整型数组 |
| int[] | jintArray | 整型数组 |
| long[] | jlongArray | 长整型数组 |
| float[] | jfloatArray | 浮点型数组 |
| double[] | jdoubleArray | 双浮点型数组 |

表 3

2.3 **域描述符**

2.3.1 基本类型

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

2.3.2 引用类型

L+该类型描述符+; 如：

String： Ljava/lang/String;

2.3.3 数组类型

[+其类型的域描述符，如：

Int[] : [I

2.3.4 函数

Java 方法：

Long f(int n,String s,int[] arr);

函数描述符：

(ILjava/lang/String;[I)J

2.4 **JNIEnv与JavaVm**

1. JavaVM概念：

JavaVM是Java虚拟机在JNI层的代表，JNI全局只有一个。

2. 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，每个线程都有一个，JNI中可能有很多个JNIEnv。

3. JNIEnv作用：

1. 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。
2. 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。

4. 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，其他线程不能使用本线程中的JNIEnv。

5. JNIEnv结构：

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。

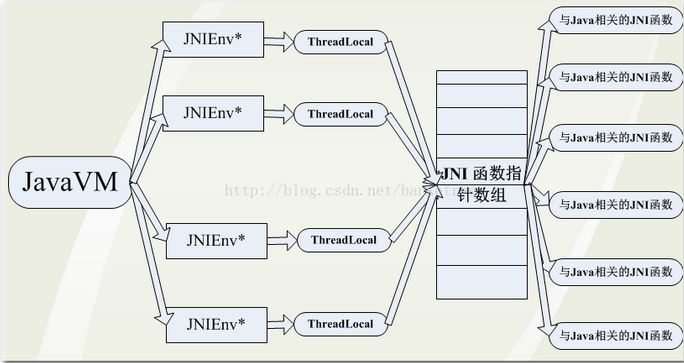


图 5

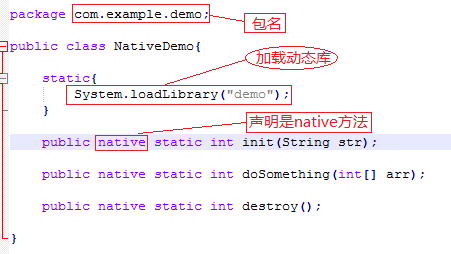
2.5 **UTF-8编码**

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C语言，因为它的编码用\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个字节表示。

1. **基本实现**

3.1 **Java端**

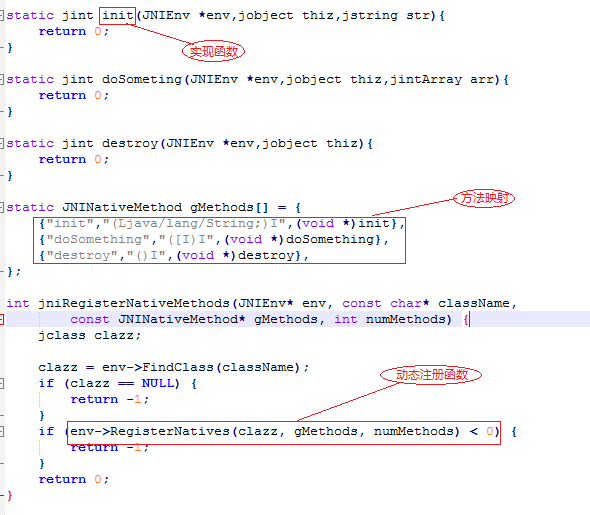
1. 定义native方法和加载动态库

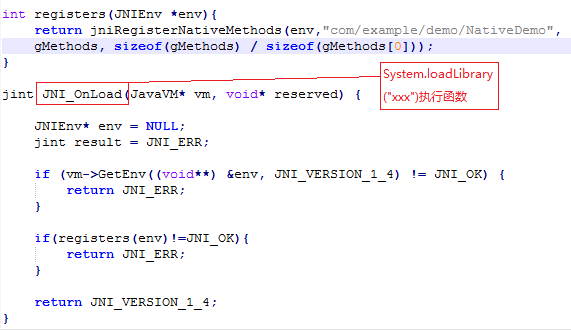


3.2 C/C++端

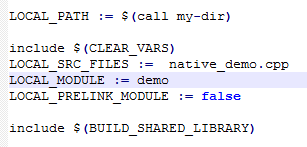
2.1 本地函数注册

* 1. 采用默认的本地函数注册流程
  2. 自己重写JNI\_OnLoad()函数

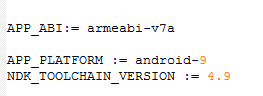




3.3 Android.mk



3.4 Application.mk



3.5 编译生成libdemo.so 动态库

1. **常见函数**

|  |  |  |  |
| --- | --- | --- | --- |
| 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数据类型的数组 |
| GetPrimitiveArrayCritical |  |  | 得到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。 |
| 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。该函数无返回值 |
| NewGlobalRef |  |  | 创建 obj 参数所引用对象的新全局引用。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）域的属性 ID。该域由其名称及签名指定。访问器函数的 |
| Get<type>Field 及 Set<type>Field系列使用域 ID 检索对象域。GetFieldID() 不能用于获取数组的长度域。应使用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 |  |  | 同上，只不过是静态属性。 |
|  |
| GetStatic<type>Field |
|  |
| SetStatic<type>Field |
| 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 参数将指定 JNINativeMethod 结构的数组，其中包含本地方法的名称、签名和函数指针。nMethods 参数将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法函数前的状态。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。 |

1. **字符串操作**

**GetStringUTFChars(JNIEnv \*env,jstring str,&is\_Copy)**

isCopy: 取值JNI\_TRUE 和JNI\_FALSE，如果值为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。但我们在开发当中，并不关心这个值是多少，通常情况下这个参数填NULL即可。

GetStringUTFChars可以把一个jstring指针（指向JVM内部的Unicode字符序列）转换成一个UTF-8格式的C字符串。

**ReleaseStringUTFChars**：

通知JVM这块内存已经不使用了，你可以清除了。与GetStringUTFChars配对使用的。

**NewStringUTF**

会构建一个新的java.lang.String字符串对象。这个新创建的字符串会自动转换成Java支持的Unicode编码。

**GetStringChars和ReleaseStringChars：**

这对函数和Get/ReleaseStringUTFChars函数功能差不多，用于获取和释放以Unicode格式编码的字符串。后者是用于获取和释放UTF-8编码的字符串。

**GetStringLength：**

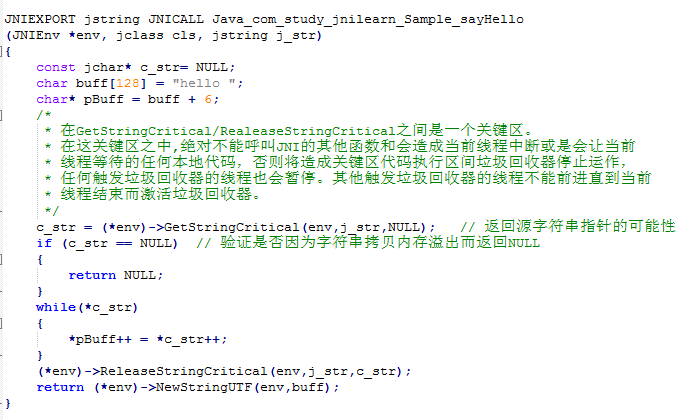
由于UTF-8编码的字符串以'\0'结尾，而Unicode字符串不是。如果想获取一个指向Unicode编码的jstring字符串长度，在JNI中可通过这个函数获取。

**GetStringUTFLength：**

获取UTF-8编码字符串的长度。

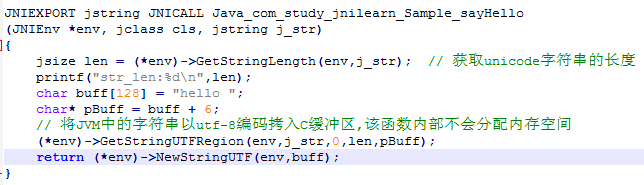
**GetStringCritical和ReleaseStringCritical：**

Get/ReleaseStringChars和Get/ReleaseStringUTFChars这对函数返回的源字符串会后分配内存，如果有一个字符串内容相当大，有1M左右，而且只需要读取里面的内容打印出来，用这两对函数就有些不太合适了。此时用Get/ReleaseStringCritical可直接返回源字符串的指针应该是一个比较合适的方式。不过这对函数有一个很大的限制，在这两个函数之间的本地代码不能调用任何会让线程阻塞或等待JVM中其它线程的本地函数或JNI函数。因为通过GetStringCritical得到的是一个指向JVM内部字符串的直接指针，获取这个直接指针后会导致暂停GC线程，当GC被暂停后，如果其它线程触发GC继续运行的话，都会导致阻塞调用者。所以在Get/ReleaseStringCritical这对函数中间的任何本地代码都不可以执行导致阻塞的调用或为新对象在JVM中分配内存，否则，JVM有可能死锁。另外一定要记住检查是否因为内存溢出而导致它的返回值为NULL，因为JVM在执行GetStringCritical这个函数时，仍有发生数据复制的可能性，尤其是当JVM内部存储的数组不连续时，为了返回一个指向连续内存空间的指针，JVM必须复制所有数据。下面代码演示这对函数的正确用法：



**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。下面代码用GetStringUTFRegion重新实现sayHello函数：



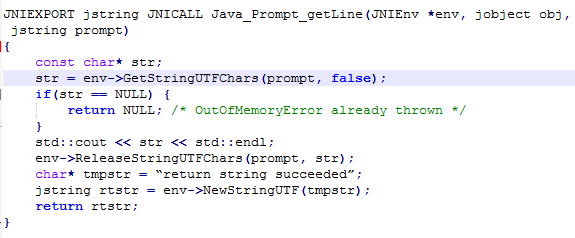
**字符串操作总结：**

1. 对于小字符串来说，GetStringRegion和GetStringUTFRegion这两对函数是最佳选择，因为缓冲区可以被编辑器提前分配，而且永远不会产生内存溢出的异常。当你需要处理一个字符串的一部分时，使用这对函数也是不错。因为它们提供了一个开始索引和子字符串的长度值。另外，复制少量字符串的消耗 也是非常小的。
2. 使用GetStringCritical和ReleaseStringCritical这对函数时，必须非常小心。一定要确保在持有一个由 GetStringCritical 获取到的指针时，本地代码不会在 JVM 内部分配新对象，或者做任何其它可能导致系统死锁的阻塞性调用。
3. 获取Unicode字符串和长度，使用GetStringChars和GetStringLength函数。
4. 获取UTF-8字符串的长度，使用GetStringUTFLength函数。
5. 创建Unicode字符串，使用NewStringUTF函数。
6. 从Java字符串转换成C/C++字符串，使用GetStringUTFChars函数。
7. 通过GetStringUTFChars、GetStringChars、GetStringCritical获取字符串，这些函数内部会分配内存，必须调用相对应的ReleaseXXXX函数释放内存。
8. **参数传递**

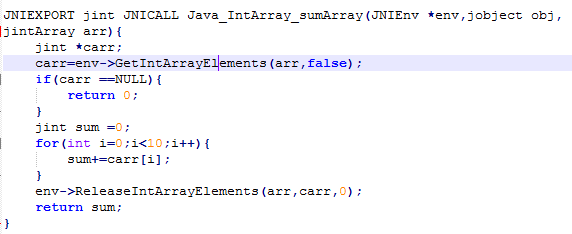
6.1 基本类型传递

Java中的基本类型包括 boolean,byte,char,short,int,long,float,double，对应于JNI中有jboolean,jbyte,jchar,jshort,jint,jlong,jfloat,jdouble。这几种类型几乎都可以当成对应的C++类型来用。

6.2 String参数的传递

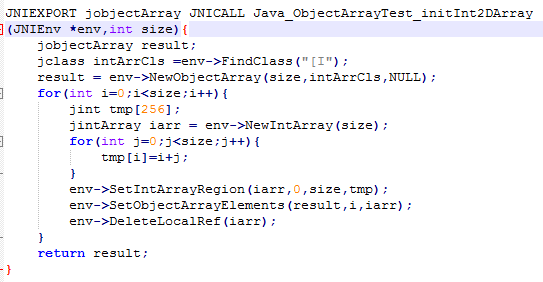
****

6.3 数组类型传递



6.4 二维数组和字符串数组

在JNI中，二位数组和String数组都被视为object数组，因为数组和String被视为object。将”[I”替换成其他的Object对象，可返回相应对象数组。



1. **调用Java**

7.1 JNIEnv对象

JNIEnv类型代表Java环境。通过这个JNIEnv\*指针，就可以对Java端的代码进行操作。如：创建Java类得对象，调用Java对象的方法，获取Java对象的属性等。JNIEnv的指针会被JNI传送到本地方法的实现函数中来对Java端的代码进行操作。

JNIEnv类中的函数：

NewObject/NewString/New<TYPE>Array： new新对象

Get/Set<TYPE>Field:获取属性

Get/SetStatic<TYPE>Field:获取静态属性

Call<TYPE>Method/CallStatic<TYPE>Method:调用方法

7.2 获取jclass

为了能够在C/C++使用Java类，jni.h头文件中专门定义了jclass类型来表示Java中的Class类。Jclass的获取方式：

jclass FindClass(const char\* clsName) 根据类名来查找一个类，完整类名。FindClass会在classpath系统环境下寻找类，需要传入完整的类名。

jclass GetObjectClass(jobject obj)   根据一个对象，获取该对象的类。

jclass GetSuperClass(jclass obj)     获取一个类的父类。

7.3 本地代码访问Java类中的属性与方法

JNI在jni.h头文件中定义了jfieldID，jmethodID类表示Java端的属性和方法。

如何获取属性： 在访问或设置Java属性的时候，首先就要现在本地代码中取得代表Java属性的jfieldID，然后才能在本地代码中进行Java属性操作。

如何调用java的方法：调用Java端的方法时，需要取得代表方法的jmethodID才能进行Java方法调用。

JNIEnv获取相应的jfieldID和jmethodID的方法：

GetFieldID/GetMethodID

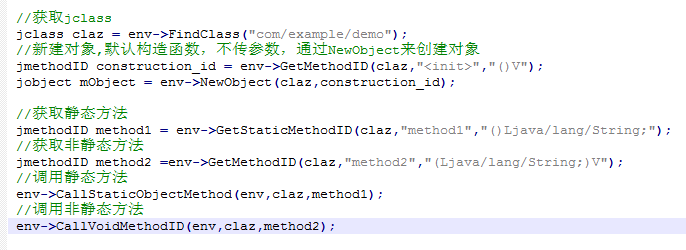
GetStaticFieldID/GetStaticMethodID

GetMethodID也可以取得构造函数的jmethodID。创建java对象时调用指定的构造函数。如：

(\*jniEnv)->GetMethodID(jniEnv, Clazz,"<init>", "()V");

这个比较特殊，这个是默认构造函数的方法，一般用这个来初始化对象， 但是再实际过程中，为了快速生成一个实例，一般通过工厂方法类创建 jobject。

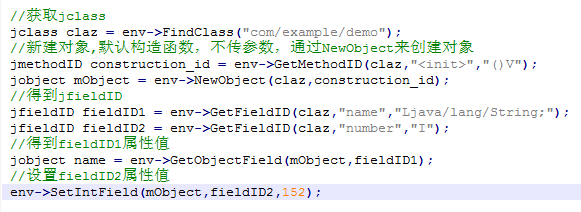
获取jmethodID，然后调用Java中方法代码如下：



注意：

GetXXXMethod 和 CallXXXMethod。第一个XXX表示的是映射方法的类型，如：静态和非静态；第二个XXX表示调用方法的返回值，如：Void、Object等等。

获取jfieldID，然后获取或修改Java对象的成员变量，代码如下：



1. **异常处理**

8.1 现状

JNI没有像Java一样有try…catch…final这样的异常处理机制，面且在本地代码中调用某个JNI接口时如果发生了异常，后续的本地代码不会立即停止执行，而会继续往下执行后面的代码。

8.2 异常检查

1. 第一种方式是：大部分JNI函数会通过特定的返回值（比如NULL）来表示已经发生了一个错误，并且当前线程中有一个异常需要处理。
2. 当一个JNI函数返回一个明确的错误码时，你仍然可以用ExceptionCheck来检查是否有异常发生。但是，用返回的错误码来判断比较高效。一旦JNI函数的返回值是一个错误码，那么接下来调用ExceptionCheck肯定会返回JNI\_TURE。

jint num = env->CallIntMethod(fraction,MID);

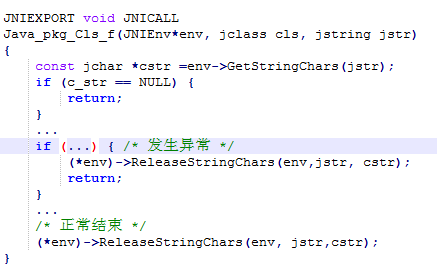
if(env->ExceptionCheck()){

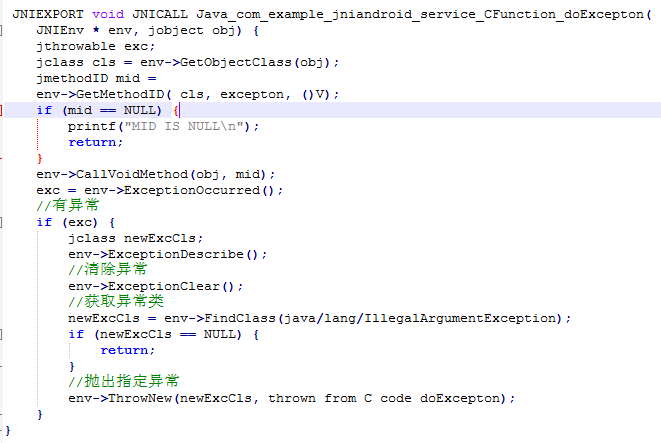
return;

}

8.3 异常处理

1. 一旦发生异常，立即返回，让调用者处理这个异常。
2. 通过ExceptionClear清除异常，然后执行自己的异常处理代码。通常来说，当有一个未处理的异常时，你只可以调用两种JNI函数：异常处理函数和清除VM资源的函数。如：



****

1. 制作一个抛出异常的工具函数

void JNU\_ThrowByName(JNIEnv \*env,const char \*name,const char \*msg){

jclass cls=env->FindClass(name);

If(cls !=NULL){

env->ThrowNew(cls,msg);

}

Env->DeleteLocalRef(cls);

}

1. **内存泄漏**

9.1 常见内存泄漏

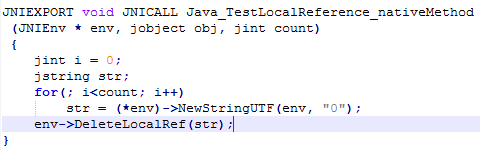
JNI 中的 Local Reference 只在 native method 执行时存在，当 native method 执行完后自动失效。这种自动失效，使得对 Local Reference 的使用相对简单，native method 执行完后，它们所引用的 Java 对象的 reference count 会相应减 1。不会造成 Java Heap 中 Java 对象的内存泄漏。

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。

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。但这是错误的。对 Local Reference 的理解不够，会造成潜在的内存泄漏。

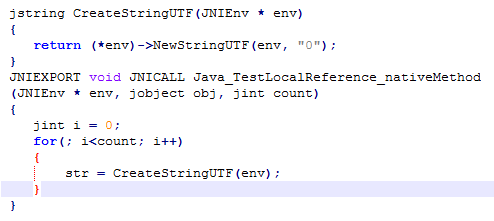
错误示例1：

我们循环执行 count 次，JNI function NewStringUTF() 在每次循环中从 Java Heap 中创建一个 String 对象，str 是 Java Heap 传给 JNI native method 的 Local Reference，每次循环中新创建的 String 对象覆盖上次循环中 str 的内容。str 似乎一直在引用到一个 String 对象。整个运行过程中，我们看似只创建一个 Local Reference。实则不然。



错误示例2：

nativeMethod 在循环中调用 count 次，utility 函数在创建一个 String 对象后即返回，并且会有一个退栈过程，似乎所创建的 Local Reference 会在退栈时被删除掉，所以应该不会有很多 Local Reference 被创建。实际运行结果并非如此。



9.2 深层分析Local Reference

JNI Local Reference 的生命期是在 native method 的执行期（从 Java 程序切换到 native code 环境时开始创建，或者在 native method 执行时调用 JNI function 创建），在 native method 执行完毕切换回 Java 程序时，所有 JNI Local Reference 被删除，生命期结束（调用 JNI function 可以提前结束其生命期）。

LocalRef 与局部变量的区别：

1. 局部变量存储在线程堆栈中，而 Local Reference 存储在 Local Ref 表中。
2. 局部变量在函数退栈后被删除，而 Local Reference 在调用 DeleteLocalRef() 后才会从 Local Ref 表中删除，并且失效，或者在整个 Native Method 执行结束后被删除。
3. 可以在代码中直接访问局部变量，而 Local Reference 的内容无法在代码中直接访问，必须通过 JNI function 间接访问。

9.3 控制Local Reference生命周期

在 JNI 编程时，正确控制 JNI Local Reference 的生命期。如果需要创建过多的 Local Reference，那么在对被引用的 Java 对象操作结束后，需要调用 JNI function（如 DeleteLocalRef()），及时将 JNI Local Reference 从 Local Ref 表中删除，以避免潜在的内存泄漏。

9.4 总结

1. native code本身的内存管理机制依然要遵循
2. 使用Global reference 时，当 native code 不再需要访问 Global reference 时，应当调用 JNI 函数 DeleteGlobalRef() 删除 Global reference 和它引用的 Java 对象。Global reference 管理不当会导致 Java Heap 的内存泄漏。
3. 透彻理解 Local reference，区分 Local reference 和 native code 的局部变量，避免混淆两者所引起的 native memory 的内存泄漏。
4. 使用 Local reference 时，如果 Local reference 引用了大的 Java 对象，当不再需要访问 Local reference 时，应当调用 JNI 函数 DeleteLocalRef() 删除 Local reference，从而也断开对 Java 对象的引用。这样可以避免 Java Heap 的 out of memory。
5. 使用 Local reference 时，如果在 native method 执行期间会创建大量的 Local reference，当不再需要访问 Local reference 时，应当调用 JNI 函数 DeleteLocalRef() 删除 Local reference。Local reference 表空间有限，这样可以避免 Local reference 表的内存溢出，避免 native memory 的 out of memory。