文档修订说明

|  |  |  |  |
| --- | --- | --- | --- |
| **版本号** | **修订日期** | **修订说明** | **修订人** |
| V1.0.0 | 2018/12/01 | 创建文档 | 王海 |
| V1.1.0 | 2018/12/07 | 添加了Jni异常处理章节 | 王海 |
| V1.2.0 | 2018/12/11 | 更新了<JNI中的内存泄露章节> | 王海 |
|  |  |  |  |

目录

[1. 什么是JNI 3](#_Toc532291061)

[1.1. JNI 编程中的优点 3](#_Toc532291062)

[1.2. JNI 编程中的弊端 3](#_Toc532291063)

[1.3. Jni加载方法 3](#_Toc532291064)

[2. Java、Jni、C之间的数据类型对应关系 4](#_Toc532291065)

[2.1. 基本类型 4](#_Toc532291066)

[2.2. 数组类型 4](#_Toc532291067)

[2.3. 对象类型 5](#_Toc532291068)

[2.4. 方法的签名 5](#_Toc532291069)

[3. JNI中JNIEnv、jobject和jclass是什么? 6](#_Toc532291070)

[3.1. JNIEnv指针 6](#_Toc532291071)

[3.1.1. 常用的Jni函数列表 8](#_Toc532291072)

[3.2. Jni接口参数jobject和jclass的区别 11](#_Toc532291073)

[4. 常用的Jni函数说明 12](#_Toc532291074)

[4.1. 获得类： 12](#_Toc532291075)

[4.2. 获得类成员变量： 12](#_Toc532291076)

[4.3. 操作成员变量： 12](#_Toc532291077)

[4.4. 获得类的方法： 12](#_Toc532291078)

[4.5. 创建java类对象： 13](#_Toc532291079)

[5. JNI的使用场景 14](#_Toc532291080)

[6. JNI编程中的内存泄露 14](#_Toc532291081)

[6.1. JAVA 中的内存泄漏 14](#_Toc532291082)

[6.1.1. Java Heap 的内存泄漏 14](#_Toc532291083)

[6.2. JVM 中 native memory 的内存泄漏 15](#_Toc532291084)

[6.2.1. Native Code 本身的内存泄漏 15](#_Toc532291085)

[6.2.2. Global Reference 引入的内存泄漏 15](#_Toc532291086)

[6.2.3. Local Reference引入的内存泄漏 15](#_Toc532291087)

[6.3. 局部引用和全局引用 16](#_Toc532291088)

[6.4. Jni内存泄露的处理方法汇总 17](#_Toc532291089)

[6.4.1. New开头的 17](#_Toc532291090)

[6.4.2. Get开头的 17](#_Toc532291091)

[6.4.3. FindClass类型的 18](#_Toc532291092)

[6.5. JNI内存泄露总结 18](#_Toc532291093)

[7. 异常 19](#_Toc532291094)

[7.1. 异常来自于C回调Java的方法 19](#_Toc532291095)

[7.1.1. Java的异常处理 19](#_Toc532291096)

[7.1.2. JNI的异常处理 19](#_Toc532291097)

[7.1.3. 总结 20](#_Toc532291098)

[8. C和C++在Jni接口上的差异 21](#_Toc532291099)

[8.1. C语言实现 21](#_Toc532291100)

[8.2. C++语言实现 21](#_Toc532291101)

[9. 参考文档 22](#_Toc532291102)

# 什么是JNI

JNI是（Java Native Interface）本地语言编程接口。它允许运行在JVM中的Java代码和用C、C++或汇编写的本地代码相互操作。

也就是在JNI调用中，不仅仅Java可以调用本地方法，本地代码也可以调用Java中的方法和成员变量。

## JNI 编程中的优点

1. 防止反编译：由于java代码容易被反编译。
2. 对于运行速度有严格要求的代码。

3、native code 底层操作，更加高效。

4、对 native code 的代码重用。

## JNI 编程中的弊端

1、从 Java 环境到 native code 的上下文切换耗时、低效。

2、JNI 编程，如果操作不当，可能引起 Java 虚拟机的崩溃。

3、JNI 编程，如果操作不当，可能引起内存泄漏。

## Jni加载方法

在Java里通过System.loadLibrary()来加载动态库，但是，动态库只能被加载一次，因此，通常动态库的加载放在静态初始化语句块中。

package pkg;

class Cls {

public static native void init(); // 声明为本地方法

static {

System.loadLibrary("testNative"); // 通过静态初始化语句块，来加载动态库

init(); //做一些初始化动作

}

}

# Java、Jni、C之间的数据类型对应关系

用Java代码调用C\C++代码时候，肯定会有参数数据的传递。两者属于不同的编程语言，在数据类型上有很多差别。

例如，尽管C拥有int和long的数据类型，但是他们的实现却是取决于具体的平台。在一些平台上，int类型是16位的，而在另外一些平台上市32位的整数。基于这个原因，Java本地接口定义了jint，jlong等等。

JNI使用签名作为Java虚拟机内容的表示。

## 基本类型

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **基本类型** | | | | |
| **Java类型** | **C/C++类型** | **Jni类型** | **签名** | **备注** |
| int | int | jint | **I** |  |
| boolean | int/bool | jboolean | **Z** |  |
| double | double | jdouble | **D** |  |
| **byte** | signed char | jbyte | **B** |  |
| short |  |  | **S** |  |
| long |  |  | **J** |  |
| float | float | jfloat | **F** |  |
| char |  |  | **C** | Java里char是两个字节 |
| void |  | void | **V** |  |
|  |  |  |  |  |

注意：

1. C里的signed char数组 对应的是 Java里的byte数组
2. Java没有无符号类型，全部是有符号类型的数据类型

## 数组类型

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **数组类型** | | | | |
| **Java类型** | **C/C++类型** | **Jni类型** | **签名** | **备注** |
| Int[] | Int[] | jintArray | **[I** |  |
| boolean[] | Int/bool[] | jbooleanArray | **[Z** |  |
| double[] | double[] |  | **[D** |  |
| byte[] | signed char[] | jbyteArray | **[B** |  |
| short[] |  |  |  |  |
| long[] |  |  |  |  |
| float[] | float[] | jfloatArray | **[F** |  |
| char[] |  |  |  |  |
|  |  |  |  |  |

**注意**：

1、一个数组int[]，就需要表示为这样"[I"。如果多个数组double[][][]就需要表示为这样 "[[[D"。也就是说每一个方括号开始，就表示一个数组维数。多个方框后面，就是数组 的类型。

## 对象类型

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **对象类型** | | | | |
| **Java类型** | **C/C++类型** | **Jni类型** | **签名** | **备注** |
| String | char[] | jstring | **Ljava/lang/String;** | C里没有String对应的类型，用object对应，如GetObjectField |
| Student |  | jobject | **Lcom/wanghai/util/Student;** |  |
|  |  |  | **Ljava/lang/Object;** |  |
|  |  |  |  |  |

注意：

1、对象类型，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。

2、如果是一个嵌入类(内部类)，则用$作为类名间的分隔符。例如 "Landroid/os/FileUtils$FileStatus;"

## 方法的签名

(参数1类型签名**参数2类型签名**参数3类型签名....... )**返回值**类型签名

**注意**：

1、函数名，在签名中没有体现出来

2、参数列表相挨着，中间没有逗号，没有空格

3、返回值出现在()后面

4、如果参数是引用类型，那么参数应该为：L类型;

5、如果函数没有返回值，也要加上V类型

# JNI中JNIEnv、jobject和jclass是什么?

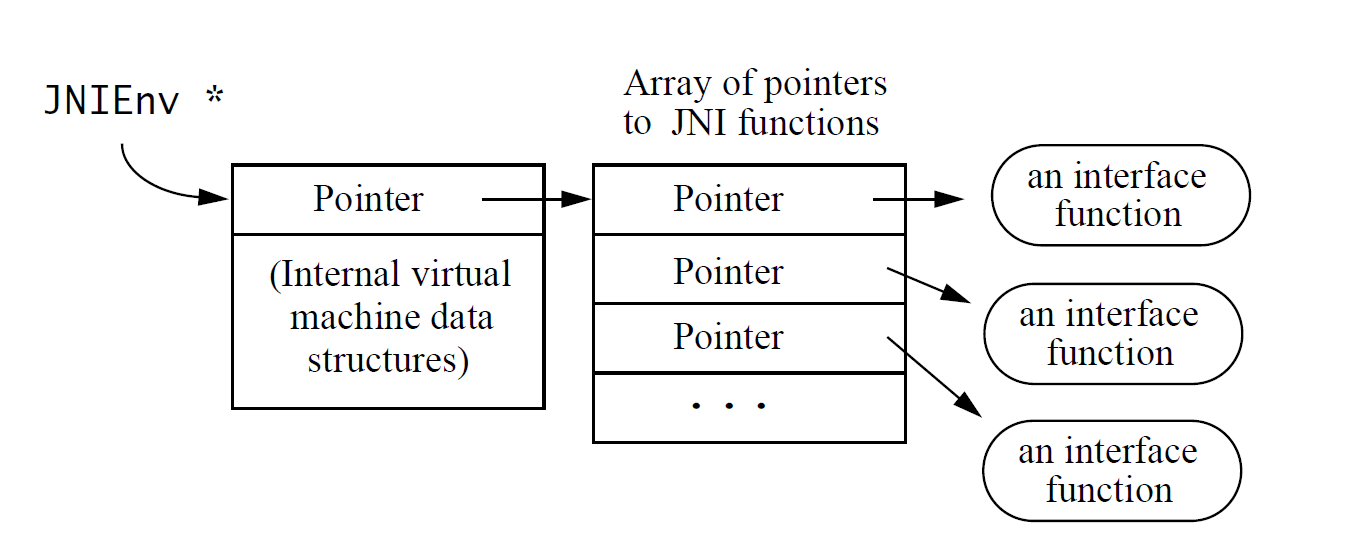
在进行JNI编程开发的时候，使用javah生成Native方法对应的Native函数声明，会发现所有的Native函数的第一个参数永远是JNIEnv指针，而第二个参数永远是jobject或jclass中的一个。

JNIEnv是一个与线程相关的变量，不同线程的JNIEnv彼此独立。当Java代码调用本地方法时，JVM将JNI接口指针作为参数传递给本地方法，当**同一个**Java线程调用本地方法时JVM保证传递给本地方法的参数是相同的。不同的Java线程调用本地方法时，本地方法接收到的JNI接口指针是不同的。

JNIEnv指针指代何物？具有何种功能？jobject和jclass又有何区别？

## JNIEnv指针

JNIEnv，顾名思义，指代了Java本地接口环境(Java Native Interface Environment），是一个JNI**接口指针**，指向了本地方法的一个函数表，该函数表中的每一个成员指向了一个JNI函数，本地方法通过JNI函数来访问JVM中的数据结构，详情如下图：



而JNIEnv指针在<jni.h>文件中的具体实现是一个包含诸多JNI函数的结构体，定义在如下文件：

/usr/lib/jvm/java-7-openjdk-amd64/include/**jni**.h

**定义 JNIEnv 结构体**：

#if defined(\_\_cplusplus) **//C++走这里；**

typedef **\_JNIEnv** JNIEnv;

typedef \_JavaVM JavaVM;

#else **//C走这里；**

typedef const struct **JNINativeInterface\*** JNIEnv;

typedef const struct JNIInvokeInterface\* JavaVM;

#endif

在 C里定义的 **JNINativeInterface**\*：注意：这里是一个 指针

/\*

\* Table of interface function pointers.

\*/

struct **JNINativeInterface** {

void\* reserved0;

void\* reserved1;

void\* reserved2;

void\* reserved3;

/\*

\* 它们都是一些函数指针，这个结构体的每个native方法的第一个参数JNIEnv env 就是它自己

\*/

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

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

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

.......................

};

**在 C++里定义的 \_JNIEnv：**

struct **\_JNIEnv** {

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

const struct **JNINativeInterface**\* functions; //**C++里，最终也是调用C里的接口；**

#if defined(\_\_cplusplus)

jint GetVersion()

{ return functions->GetVersion(this); }

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

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

jclass FindClass(const char\* name)

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

............................

jlong GetDirectBufferCapacity(jobject buf)

{ return functions->GetDirectBufferCapacity(this, buf); }

/\* added in JNI 1.6 \*/

jobjectRefType GetObjectRefType(jobject obj)

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

#endif /\*\_\_cplusplus\*/

};

以下这两个方法，是由java虚拟机调用的，它传入一个vm，通过vm我们可以得到JNIEnv, 通过JNIEnv我们可以得到java与c, c++之间的交互接口。

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

JNIEXPORT void JNI\_OnUnload(JavaVM\* vm, void\* reserved);

深入去看**JNIEnv**结构体的话，不难发现，这个结构体当中包含了几乎有所的JNI函数，大致如下;

### 常用的Jni函数列表

|  |  |  |
| --- | --- | --- |
| **函数名** | **功能** | **备注** |
|  |  |  |
| **对于 数组操作** | | |
| Get<**Type**>**Array**Elements | 获取java **Type**类型数组的首地址 | 后续直接当作C/C++类型的数组来使用 |
| Release<**Type**>**Array**Elements | 释放数组，其第三个参数mode可以取下面的值：  **0**：对Java的数组进行更新，并释放C/C++的数组  JNI\_COMMIT：对Java的数组进行更新但是不释放C/C++的数组  JNI\_ABORT：对Java的数组不进行更新，释放C/C++的数组 | 对于各个类型的数组，直接替换**Type**即可，例如：Int, Float等 |
| Get**Array**Length | 获取数组的长度 |  |
| New<**Type**>Array | 创建类型为**Type**的数组对象 | 例如：New**ByteArray**  创建一个java Byte数组 |
| Set<**Type**>**Arra**yRegion | 把一个C的char数组复制到**Type**数组里 |  |
| DeleteLocalRef | 释放局部引用 |  |
|  |  |  |
| **对于 类、成员、方法的操作** | | |
| **FindClass** | 获取Java中的类定义  分隔符用'/', 不能使用'.' | 返回jclass，此函数类似java的反射调用 |
| **GetObjectClass** | 获取当前object的类定义 | 获取jclass |
| Get**Field**ID | 获得 **类**的成员变量(**非静态**) ID |  |
| Get<**Type**>Field | 获取java对象，类型为Type的(**非静态**)成员变量 **值**  要访问的域由通过调用GetFieldID() 而得到的域 ID 指定。 | 例如：**GetBoolean**Field  但是 对于**String**类型， C里没有对应的类型，用Object对应，使用Get**Object**Field |
| Set<**Type**>Field | 更新java对象，类型为Type的(**非静态**) 成员变量 **值** | 例如：**SetBoolean**Field  更新java对象的**Boolean**成员变量 |
| **GetMethod**ID | 获取**类**的(**非静态**)方法 | 要获得**构造函数**的方法 ID，应将 <init> 作为方法名，同时将 ()V 作为签名类型。 |
| **Call**<**Type**>Method: | 从本地方法调用Java 实例方法。  根据所指定的方法 ID 调用 Java 对象的实例（非静态）方法。  参数 methodID 必须通过调用 GetMethodID() 来获得。  调用 类对象，**返回 类型为Type**的方法 | JNI中，根据不同的参数类型 和 返回值类型, 需要调用不同的方法。  传给方法的所有参数紧跟着放在 methodID 参数之后 |
| **New**Object | 创建新 Java类的 对象  jobject NewObject(JNIEnv \*env, jclass clazz, jmethodID methodID, ...);  注意：  methodID指示应调用的构造函数。methodID 必须通过调用 GetMethodID() 获得，且调用时的方法名必须为 <init>，而签名类型必须为 void ()V。 | 传递给构造函数的所有参数紧跟着放在 methodID 参数的后面。 |
| **Alloc**Object | 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。 |  |
| IsInstanceOf | 测试对象是否为某个类的实例 | NULL 对象可强制转换为任何类 |
| IsSameObject | 测试两个引用，是否引用同一个 Java 对象 |  |
|  |  |  |
|  |  |  |
| **对 静态变量、静态方法 的操作** | | |
| **GetStaticFieldID** | 获取 **类**的 **静态成员**变量 ID |  |
| **GetStatic**<**Type**>Field | 获取 类里类型为Type的static的变量的值  也就是 获取 类里的 静态变量 **值** | 如：获取类的静态变量boolean的值  GetStaticBooleanField |
| **SetStatic**<**Type**>Field | **更新** 类里类型为Type的static的变量的值 | 如：更新类的静态变量boolean的值  SetStaticBooleanField |
| **GetStaticMethodID** | 获取类里的静态方法 | 返回类的静态方法的方法 ID。方法由其名称和签名指定。 |
| **CallStatic**<**Type**>Method | 根据指定的方法 ID 调用 Java 对象的静态方法。methodID 参数必须通过调用GetStaticMethodID() 得到。 | 调用返回值类型为Type的static方法 |
|  |  |  |
|  |  |  |
| **对于Java的String操作** | | |
| **GetObject**Field | 获取java对象的String变量 | 此函数用来获取java对象里的**类**变量、**数组**变量 |
| GetString**Chars** | 把java的String转换为const char \*  **当不使用的时候，需要释放资源** | 使用UTF-16编码 |
| GetString**UTFChars** | 使用UTF-8编码 |
| Get**String**Length | 返回一个java String对象的字符串长度 | 例如”王a海”为3个字节 |
| GetStringUTFLength | 返回一个java String对象经过UTF-8编码后的字符串长度 | 例如”王a海”为7个字节，因为UTF-8里，一个汉字占3个字节，因为占1个字节 |
| GetStringRegion | 复制字符串到一个C的buffer里 |  |
| NewString | 根据传入的**宽**字符串创建一个Java String对象 |  |
| New**String**UTF | 根据传入的UTF-8字符串创建一个Java String对象 |  |
| **Set**ObjectField | 更新java对象的String成员变量 | 此函数用来更新java对象里的**类**变量 |
| **Realease**StringChars | 在不使用的时候，要分别对应的使用这两个函数来释放内存 |  |
| **Realease**StringUTFChars |  |
|  |  |  |
| **对于Java的异常操作** | | |
| ExceptionCheck | 检查最近一次Jni调用是否发生了异常(C调用java的函数) | 如果有异常，此函数返回JNITRUE，否则返回JNI\_FALSE |
| ExceptionOccurred | 检查是否发生了异常，若用异常返回该异常的引用，否则返回NULL |  |
| ExceptionDescribe | 打印异常的堆栈信息 | 就像Java中的printStackTrace |
| ExceptionClear | 清空当前产生的全部异常信息 | 情况后，异常就不会到java端了，也就是java端不会在打印异常的堆栈信息 |
| ThrowNew | 在当前线程触发一个异常，并自定义输出异常信息 |  |
| Throw | 在当前线程触发一个新的异常 | 参数是ExceptionOccurred的返回值 |
| FatalError | 致命异常，用于输出一个异常信息，并终止当前VM实例（即退出程序） |  |
|  |  |  |

**注意**：

1、**C++ 和 C**调用**JNIEnv**的函数区别，如下：

C++：const char\* c\_string = env->**GetStringUTFChars**( j\_string, 0);

C ：const char\* c\_string = (\*env)->**GetStringUTFChars**(env, j\_string, 0);

2、获取**java对象的String成员变量**，方法如下：

jfieldID j\_String\_d = (\*env)->GetFieldID(env, j\_MyInfo, "str", "Ljava/lang/String;");

jstring jstr = (\*env)->GetObjectField(env, jobj, j\_String\_d);

const char\* cstr = (\*env)->GetStringUTFChars(env, jstr, 0);

## Jni接口参数jobject和jclass的区别

jobject与**jclass**通常作为JNI函数的**第二个参数**，当所声明Native方法是**静态方法**时，对应参数**jclass**，因为静态方法不依赖对象实例，而依赖于类，所以参数中传递的是一个jclass类型。相反，如果声明的Native方法是**非**静态方法时，那么对应参数是**jobject** 。

他们的定义如下：

typedef \_jobject \*jobject;

typedef \_jclass \*jclass;

...

class \_jobject {};

class \_jclass : public \_jobject {}; //注意jclass继承了 jobject

...

由于我们一般使用javah指令直接生成Native函数的函数原型，故而不必纠结该使用哪种类型；

注意：

1、对于一个 类 操作，如 GetFieldID, 需要参数 jclass，因为它从一个类中获得field的描述。

2、对于一个 实例 操作，如 GetIntField 需要参数jobject，因为它从这个实例中获得这个field的值。

在所有的JNI方法中：jobject和**实例**操作结合，jclass和**类**操作结合。

# 常用的Jni函数说明

## 获得类：

jclass cls = env->FindClass("com/ldq/Student");

cls 可认为是类的句柄

"com/ldq/Student" 就是类文件，注意不能用 "com.ldq.Student"

class GetObjectClass(jobject obj):

通过对象实例obj来获取jclass，相当于java中的getClass方法

jclass GetSuperClass(jclass obj):

通过jclass可以获取其父类的jclass对象

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和jmethodID类型来分别代表Java对象的属性和方法。我们在访问或是设置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进行Java属性操作。

同样的，我们需要调用Java对象方法时，也是需要取得代表该方法的jmethodID才能进行Java方法调用。

## 获得类成员变量：

jfieldID fid = env->GetFieldID(cls,"age","I");

获取int age变量；

## 操作成员变量：

jint a=env->GetIntField(obj, fid);

获取类对象obj的变量

age=age+10;

env->SetIntField(obj, fid, a);

更新类对象的变量

## 获得类的方法：

jmethodID mid = env->GetMethodID(cls,"<init>","()V");

获取构造函数，"<init>" 代表构造函数，参数和返回值是 "()V"

jmethodID mid = env->GetMethodID(cls,"getAge","()I");

获取getAge函数，参数和返回值是 "()I"

## 创建java类对象：

jobject obj = env->NewObject(cls, mid);

mid：为cls的的构造函数。mid 必须通过调用 GetMethodID() 获得，且调用时的方法名必须为 <init>，而返回类型必须为 void ()V。

# JNI的使用场景

|  |  |  |
| --- | --- | --- |
| **编号** | **java 给 C 传入 常用的 基本 类型;** | **备注** |
| 1 | java 给 C 传入 常用的 基本类型 |  |
| 2 | java 给 C 传入 String 类型; |  |
| 3 | C 给 java 返回一个 String 类型 |  |
| 4 | java 给 C 传入 数组类型 | 传入指针，数组大小固定 |
| 5 | C 给 java 返回一个 byte[] 类型 | 返回一个字符串，数组大小不固定 |
| 6 | Java类 与 C结构体 转换 | java 给 C 传入一个类, 此类对应 C的一个结构体 |
| 7 | C结构体 与 Java类 转换 | C 给java 返回一个 类, 此类对应 C的一个结构体 |
| 8 | 非静态的Native方法里，更新Java类的静态变量 |  |
| 9 | 静态的Native方法里，更新Java类的静态变量 |  |
| 10 | Jni里的异常处理 |  |
|  |  |  |
|  |  |  |

# JNI编程中的内存泄露

## JAVA 中的内存泄漏

JAVA 编程中，从泄漏的内存位置角度可以分为两种：

1. JVM 中 Java Heap 的内存泄漏。
2. JVM 内存中 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 异常的出现有三种原因：

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。
3. JNI 编程错误也可能导致 Java Heap 的内存泄漏。

## 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。

JVM 的 native memory 内存泄漏原因：

1. JVM 在运行中过多的线程被创建，并且在同时运行。就可能耗尽 native memory 的容量。
2. JNI 编程错误也可能导致 native memory 的内存泄漏。
3. 申请的内存没有释放等。

### 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。

### 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。

### Local Reference引入的内存泄漏

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。

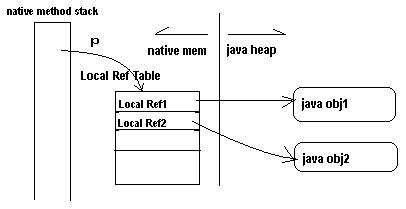
#### Local Reference和Local Reference表

理解 Local Reference 表的存在是理解 JNI Local Reference 的关键。

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 或者 调用 JNI function 可以提前结束其生命期。

实际上，每当线程从 Java 环境切换到 native code 上下文时（J2N），JVM 会分配一块内存，创建一个 Local Reference 表（大小固定），这个表用来存放本次 native method 执行中创建的所有的 Local Reference。每当在 native code 中引用到一个 Java 对象时，JVM 就会在这个表中创建一个 Local Reference。比如，我们调用 NewStringUTF() 在 Java Heap 中创建一个 String 对象后，在 Local Reference 表中就会相应新增一个 Local Reference。

图 1. Local Reference 表、Local Reference 和 Java 对象的关系



#### Local Ref不是native code的局部变量

很多人会误将 JNI 中的 Local Reference 理解为 Native Code 的局部变量。这是错误的。

Native Code的局部变量 和Local Reference 是完全不同的，区别可以总结为：

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（**在 native method 执行完毕切换回 Java 程序时**）被删除。

#### 控制Local Reference的生命期

在 JNI 编程时，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。

## 局部引用和全局引用

局部引用只在创建它们的线程里有效，本地代码不能将局部引用在多线程间传递。一个线程想要调用另一个线程创建的局部引用是不被允许的。将一个局部引用保存到全局变量中，然后在其它线程中使用它，这是一种错误的编程。

一个全局引用可以跨越多个线程，并且在被程序员手动释放之前，一直有效。和局部引用一样，全局引用保证了所引用的对象不会被垃圾回收。

## Jni内存泄露的处理方法汇总

在C++中new的对象，如果不返回java，必须用release掉，否则内存泄露。如果返回java不必release，java会自己回收。

总体原则：释放所有对object的引用，需要release的接口如下：

### New开头的

NewString/NewStringUTF/NewObject/New<Type>Array

jstring jstr = env->NewStringUTF(“hello java”);

...

env->**DeleteLocalRef**( jstr);

jobject ref= env->NewGlobalRef(customObj);

…………

env->DeleteGlobalRef(customObj);

### Get开头的

**1、String类型：**

const jchar\* (\*GetStringChars)(JNIEnv\*, jstring, jboolean\*);

void (\*ReleaseStringChars)(JNIEnv\*, jstring, const jchar\*);

const char\* (\*GetStringUTFChars)(JNIEnv\*, jstring, jboolean\*);

void (\*ReleaseStringUTFChars)(JNIEnv\*, jstring, const char\*);

**2、数组类型：**

jbyte\* array= (\*env)->GetByteArrayElements(env,jarray,&isCopy);

……….

(\*env)->ReleaseByteArrayElements(env,jarray,array,0);

### FindClass类型的

jclass ref= (env)->FindClass("java/lang/String");

jclass ref = env->GetObjectClass(robj);

…

env-> **DeleteLocalRef** (ref);

## JNI内存泄露总结

本章节阐述了 JNI 编程可能引发的内存泄漏，JNI 编程既可能引发 Java Heap 的内存泄漏，也可能引发 native memory 的内存泄漏，严重的情况可能使 JVM 运行异常终止。

JNI 软件开发人员在编程中，应当考虑以下几点，避免内存泄漏：

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，如果在 native method 执行期间，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。

# 异常

异常，就是程序在运行期间没有按照正常的程序逻辑执行，在执行过程当中出现了某种错误，导致程序崩溃。在Java里，对于程序有可能运行期间发生异常的逻辑，我们会用try…catch…来处理，如果没有处理的话，在运行期间发生异常就会导致程序奔溃。

C++中可以使用其内部的异常机制，但是这套机制抛出的异常并不会传递给我们的JVM，所以为了你补这个缺点，JNI实现了一套可以和JVM进行交流的异常机制。

## 异常来自于C回调Java的方法

### Java的异常处理

1、在Java中如果觉得某段逻辑可能会引发异常，用try…catch机制来捕获并处理异常即可；

2、如果在Java中发生运行时异常，没有使用try…catch来捕获，会导致程序直接奔溃退出，后续的代码都不会被执行；

3、java里在方法定义时，显式的用throws声明了某一个异常，编译器要求在调用此方法的的时候，必须**显式**的捕获处理 ，否则编译报错；

//定义

public static void testException() throws Exception {}

//调用

try {

testException ();

} catch (Exception e) {

e.printStackTrace();

}

### JNI的异常处理

1. 我们在写JNI程序的时候，JNI**没有**像Java一样有try…catch…final这样的异常处理机制；
2. 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。

3、检查到异常后必须予以处理。处理异常的方法也有两种：

1>Native方法可选择立即返回，这样异常就会在调用该Native方法的Java 代码中抛出。所以在Java代码中必须有捕获相应异常的代码，否则程序直接退出，记得释放必要的资源。

2>Native方法可以调用ExceptionClear()来清除异常，然后执行自己的异常处理代码。

在本地方法中，回调java函数时若引发了异常，例如：引发了一个java.lang.ArithmeticException异常，但本地接口函数并不会马上退出，而是会继续执行后面的代码，所以我们在调用完任何一个JNI接口之后，必须要做的一件事情就是**检查**这次JNI调用是否发生了异常，如果发生了异常不处理，而继续让程序执行后面的逻辑，将会产生不可预知的后果。

所以，在调用完任何一个JNI接口之后，我们需要调用JNI的ExceptionCheck函数检查最近一次JNi调用是否发生了异常，如果有异常这个函数返回JNI\_TRUE，否则返回JNI\_FALSE。当检测到异常时，我们需要先处理异常，否则Jni的函数会继续执行下去。

异常检查JNI还提供了另外一个接口，ExceptionOccurred，如果检测有异常发生时，该函数会返回一个指向当前异常的引用。作用和ExceptionCheck一样，两者的区别在于返回值不一样。

### 总结

1、当调用一个JNI函数后，必须先**检查、处理、清除**异常后再做其它 JNI 函数调用，否则会产生不可预知的结果。

2、一旦发生异常，立即返回，让调用者处理这个异常。或 调用 ExceptionClear 清除异常，然后执行自己的异常处理代码。

3、异常处理的相关JNI函数总结：

1> ExceptionCheck：检查是否发生了异常，若有异常返回JNI\_TRUE，否则返回JNI\_FALSE

2> ExceptionOccurred：检查是否发生了异常，若用异常返回该异常的引用，否则返回NULL

3> ExceptionDescribe：打印异常的堆栈信息

4> ExceptionClear：清除异常堆栈信息

5> ThrowNew：在当前线程触发一个异常，并自定义输出异常信息

jint (JNICALL \*ThrowNew) (JNIEnv \*env, jclass clazz, const char \*msg);

6> Throw：丢弃一个现有的异常对象，在当前线程触发一个新的异常

jint (JNICALL \*Throw) (JNIEnv \*env, jthrowable obj);

7> FatalError：致命异常，用于输出一个异常信息，并终止当前VM实例（即退出程序）

void (JNICALL \*FatalError) (JNIEnv \*env, const char \*msg);

# C和C++在Jni接口上的差异

本地方法可以由C或C++来实现，如下：

## C语言实现

jdouble native\_fun (

JNIEnv \*env, /\* interface pointer \*/

jobject obj, /\* "this" pointer \*/

jint i, /\* argument #1 \*/

jstring s) /\* argument #2 \*/

{

/\* Obtain a C-copy of the Java string \*/

const char \*str = (\*env)->GetStringUTFChars(env, s, 0); // **C比C++的接口多一个参数**env**；**

/\* process the string \*/

...

/\* Now we are done with str \*/

(\*env)->ReleaseStringUTFChars(env, s, str);

return ...

}

## C++语言实现

extern "C" /\* specify the C calling convention \*/

jdouble native\_fun (

JNIEnv \*env, /\* interface pointer \*/

jobject obj, /\* "this" pointer \*/

jint i, /\* argument #1 \*/

jstring s) /\* argument #2 \*/

{

const char \*str = env->GetStringUTFChars(s, 0); //**C++比C的接口少一个参数**env**；**

...

env->ReleaseStringUTFChars(s, str);

return ...

}

由上面两段代码对比可知，本地代码使用C++来实现更简洁。

两段本地代码第一个参数都是JNIEnv\*env，它代表了VM里的环境，本地代码可以通过这个env指针对Java代码进行操作，例如：创建Java类对象，调用Java对象方法，获取Java对象属性等。

## C和C++的差异

**在C中**：

使用JNIEnv\* env要这样 (\*env)->方法名(env,参数列表)

使用JavaVM\* vm要这样 (\*vm)->方法名(vm,参数列表)

**在C++中**：

使用JNIEnv\* env要这样 env->方法名(参数列表)

使用JavaVM\* vm要这样 vm->方法名(参数列表)

# 参考文档

<https://www.cnblogs.com/shaweng/p/4013320.html>

<https://blog.csdn.net/woshinia/article/details/25132353>

<https://www.jianshu.com/p/caad51c9cc34>

<https://blog.csdn.net/mennoa/article/details/42425901>

<https://blog.csdn.net/jacky198554/article/details/77899377>

<https://blog.csdn.net/CV_Jason/article/details/80026265?utm_source=blogxgwz6>

内存泄露：

[**https://www.ibm.com/developerworks/cn/java/j-lo-jnileak/**](https://www.ibm.com/developerworks/cn/java/j-lo-jnileak/)

<https://www.cnblogs.com/shaweng/p/4013320.html>

Jni常用接口说明：

<https://blog.csdn.net/autumn20080101/article/details/8646431>

Jni系列：

<https://www.zybuluo.com/cxm-2016/note/563686>

Jni里的异常处理：

<https://blog.csdn.net/xyang81/article/details/45770551>