解析Java的JNI编程中的对象引用与内存泄漏问题

JNI，Java Native Interface，是 native code 的编程接口。JNI 使 Java 代码程序可以与 native code 交互——在 Java 程序中调用 native code；在 native code 中嵌入 Java 虚拟机调用 Java 的代码。

1.1 JNI 编程在软件开发中运用广泛，其优势可以归结为以下几点：

> 1. 利用 native code 的平台相关性，在平台相关的编程中彰显优势。
> 2. 对 native code 的代码重用。
> 3. native code 底层操作，更加高效

然而任何事物都具有**两面性**，JNI 编程也同样如此。

程序员在使用 JNI 时应当认识到 JNI 编程中如下的几点弊端，扬长避短，才可以写出更加完善、高性能的代码：

> 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。
> 4. JAVA 中的内存泄漏
> 5. 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；JVM 内存中 native memory 的内存泄漏。

1.2 局部 和全局引用

JNI将实例、数组类型暴露为不透明的引用。**native代码从不会直接检查一个不透明的引用指针的上下文**，

而是通过使用JNI函数来访问由不透明的引用所指向的数据结构。因为只处理不透明的引用，

这样就不需要担心不同的**java VM实现而导致的不同的内部对象的布局。**

然而，还是有必要了解一下JNI中不同种类的引用：

JNI 支持3中不透明的引用：

> 局部引用
>
> 全局引用
>
> 弱全局引用。

局部和全局引用，有着各自不同的生命周期。**局部引用能够被自动释放**，而全局引用和弱全局引用在被程序员释放之前，是一直有效的。
一个局部或者全局引用，使所提及的对象不能被垃圾回收。而弱全局引用，则允许提及的对象进行垃圾回收。

> 不是所有的引用都可以在所有上下文中使用的。例如：在一个创建返回引用native方法之后，使用一个局部引用，这是非法的。

那么到底什么是局部引用，什么事全局引用，它们有什么不同?

1.3 局部引用

多数JNI函数都创建局部引用。例如JNI函数NewObject创建一个实例，并且返回一个指向该实例的局部引用。

局部引用只在创建它的native方法的动态上下文中有效，并且只在native方法的一次调用中有效。所有局部引用只在一个native方法的执行期间有效，在该方法返回时，它就被回收。

在native方法中使用一个静态变量来保存一个局部引用，以便在随后的调用中使用该局部引用，这种方式是行不通的。例如以下例子，误用了局部引用：
/* This code is illegal */
jstring

```
jclass stringClass;

extern "C"
JNIEXPORT jlong JNICALL
Java_com_maniu_giframework_GifHandler_loadGif1(JNIEnv *env, jclass clazz,
jstring path_) {
    stringClass = env->FindClass( "java/lang/String");
    return -1;
}
```

这种保存局部引用的方式是不正确的，因为FindClass()返回的是对java.lang.String的局部引用。这是因为，在native代码从MyNewString返回退出时，

VM 会释放所有局部引用，包括存储在stringClass变量中的指向类对象的引用。

这样当再次后继调用MyNewString时，可能会访问非法地址，导致内存被破坏，或者系统崩溃。

局部引用失效，有两种方式：

> 1系统会自动释放局部变量。
> 2程序员可以显示地管理局部引用的生命周期，例如调用DeleteLocalRef。

一个局部引用可能在被摧毁之前，被传给多个native方法。例如，MyNewString中，返回一个由NewObject创建的字符串引用，它将由NewObject的调用者来决定是否释放该引用。而在以下代码中：

```
extern "C"
JNIEXPORT jlong JNICALL
Java_com_maniu_giframework_GifHandler_loadGif2(JNIEnv *env, jclass clazz,
jstring path_) {
    const char *path = env->GetStringUTFChars(path_, 0);
    env->ReleaseStringUTFChars(path_, path);
    return -1;
}
```

在VM接收到来自Java_C_f的局部引用以后，将基础字符串对象传递给ava_C_f的调用者，然后摧毁原本由MyNewString中调用的JNI函数NewObject所创建的局部引用。

局部对象只属于创建它们的线程，只在该线程中有效。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。

全局引用

在一个native方法被多次调用之间，可以使用一个全局引用跨越它们。

一个全局引用可以跨越多个线程，并且在被程序员释放之前，一致有效。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。

和局部引用不一样（局部变量可以由多数JNI函数创建），全局引用只能由一个JNI函数创建（NewGlobalRef）。下面是一个使用全局引用版本的MyNewString：

jstring

```
extern "C"
JNIEXPORT jlong JNICALL
Java_com_maniu_giframework_GifHandler_loadGif3(JNIEnv *env, jclass clazz,
jstring path_) {
    jclass tempClass = env->FindClass( "java/lang/String");
    stringClass = static_cast<jclass>(env->NewGlobalRef(tempClass));
//    合适的地方销毁
     env ->DeleteLocalRef( stringClass);
    /* Is the global reference created successfully? */
    if (stringClass == NULL) {
        return NULL; /* out of memory exception thrown */
    }
    return -1;
}
```

弱全局引用

弱全局引用是在java 2 SDK1.2才出现的。它由NewGolableWeakRef函数创建，并且被DeleteGloablWeakRef函数摧毁。和全局引用一样，它可以跨native方法调用，也可以跨越不同线程。但是和全局引用不同的是，它不阻止对基础对象的垃圾回收。下面是弱全局引用版的MyNewString：

JNIEXPORT void JNICALL

```
//弱全局引用
extern "C"
JNIEXPORT jlong JNICALL
Java_com_maniu_giframework_GifHandler_loadGif4(JNIEnv *env, jclass clazz,
jstring path_) {
    jclass tempClass = env->FindClass( "java/lang/String");
//    创建弱全局引用
    stringClass = static_cast<jclass>(env->NewWeakGlobalRef(tempClass));
//    合适的地方销毁
    env ->DeleteLocalRef( stringClass);
    /* Is the global reference created successfully? */
    if (stringClass == NULL) {
        return NULL; /* out of memory exception thrown */
    }
    return -1;
}
```

弱全局引用在一个被native代码缓存着的引用不想阻止基础对象被垃圾回收时，非常有用。如以上例子，mypkg.MyCls.f需要缓存mypkg.MyCls2的引用。而通过将mypkg.MyCls2缓存到弱引用中，能够实现MyCls2类依旧可以被卸载。

上面代码中，我们假设了MyCls类和MyCls2类的生命周期是相同的（例如，在同一个类中被加载、卸载）。所以没有考虑MyCls2被卸载了，然后在类MyCls和native方法的实现Java_mypkg_MyCls_f还要被继续使用时，再被重新加载起来的情况。针对于这个MyCls2类可能被卸载再加载的情况，在使用时，需要检查该弱全局引用是否还有效。如何检查，这将在下面提到。

比较引用

可以用JNI函数IsSameObject来检查给定的两个局部引用、全局引用或者弱全局引用，是否指向同一个对象。
(*env)->IsSameObject(env, obj1, obj2)
返回值为：
JNI_TRUE，表示两个对象一致，是同一个对象。
JNI_FALSE，表示两个对象不一致，不是同一个对象。

在java VM中NULL是null的引用。
如果一个对象obj是局部引用或者全局引用，则可以这样来检查它是否指向null对象：

```
(*env)->IsSameObject(env, obj, NULL)
```

或者：

```
NULL == obj
```

而对于弱全局引用，以上规则需要改变一下：
我们可以用这个函数来判断一个非0弱全局引用wobj所指向的对象是否仍旧存活着（依旧有效）。

```
(*env)->IsSameObject(env, wobj, NULL)
```

返回值：
JNI_TRUE，表示对象已经被回收了。
JNI_FALSE，表示wobj指向的对象，依旧有效。

释放引用
除了引用的对象要占用内存，每个JNI引用本身也会消耗一定内存。作为一个JNI程序员，应该对在一段给定的时间里，程序会用到的引用的个数，做到心中有数。特别是，尽管程序所创建的局部引用最终会被VM会被自动地释放，仍旧需要知道在程序在执行期间的任何时刻，创建的局部引用的上限个数。创建过多的引用，即便他们是瞬间、短暂的，也会导致内存耗尽。

释放局部引用
多数情况下，在执行一个native方法时，你不需要担心局部引用的释放，java VM会在native方法返回调用者的时候释放。然而有时候需要JNI程序员显示的释放局部引用，来避免过高的内存使用。那么什么时候需要显示的释放呢，且看一下情景：
1）在单个native方法调用中，创建了大量的局部引用。这可能会导致JNI局部引用表溢出。此时有必要及时地删除那些不再被使用的局部引用。例如以下代码，在该循环中，每次都有可能创建一个巨大的字符串数组。在每个迭代之后，native代码需要显示地释放指向字符串元素的局部引用：

```
for (i = 0; i < len; i++) {
  jstring jstr = (*env)->GetObjectArrayElement(env, arr, i);

  (*env)->DeleteLocalRef(env, jstr);
}
```

2）你可能要创建一个工具函数，它会被未知的上下文调用。例如之前提到到MyNewString这个例子，它在每次返回调用者欠，都及时地将局部引用释放。

3）native方法，可能不会返回（例如，一个可能进入无限事件分发的循环中的方法）。此时在循环中释放局部引用，是至关重要的，这样才能不会无限期地累积，进而导致内存泄露。

4）native方法可能访问一个巨大的对象，因此，创建了一个指向该对象的局部引用。native方法在返回调用者之前，除访问对象之外，还执行了额外的计算。指向这个大对象的局部引用，将会包含该对象，以防被垃圾回收。这个现象会持续到native方法返回到调用者时，即便这个对象不会再被使用，也依旧会受保护。在以下例子中，由于在lengthyComputation()前，显示地调用了DeleteLocalRef，所以垃圾回收器有机会可以释放lref所指向的对象。

```
NIEXPORT void JNICALL
Java_pkg_Cls_func(JNIEnv *env, jobject this)
{
  lref = ...       /* a large Java object */
  ...           /* last use of lref */
  (*env)->DeleteLocalRef(env, lref);
  lengthyComputation();  /* may take some time */
  return;         /* all local refs are freed */
}
```

这个情形的实质，就是允许程序在native方法执行期间，java的垃圾回收机制有机会回收native代码不在访问的对象。

管理局部引用
不知道java 7怎么样了，应该更强大吧，有时间，去看看，这里且按照java2的特性来吧。
SDK1.2中提供了一组额外的函数来管理局部引用的生命周期。他们是EnsureLocalCapacity、NewLocalRef、PushLocalFram以及PopLocalFram。
JNI的规范要求VM可以自动确保每个native方法可以创建至少16个局部引用。经验显示，如果native方法中未包含和java VM的对象进行复杂的互相操作，这个容量对大多数native方法而言，已经足够了。如果，出现这还不够的情况，需要创建更多的局部引用，那么native方法可以调用EnsureLocalCapacity来保证这些局部引用有足够的空间。

```
if ((*env)->EnsureLocalCapacity(env, len)) < 0) {

}
for (i = 0; i < len; i++) {

    jstring jstr = (*env)->GetObjectArrayElement(env, arr, i);

}
```

这样做，所消耗的内存，自然就有可能比之前的版本来的多。

另外，PushLocalFram\PopLocalFram函数允许程序员创建嵌套作用域的局部引用。如下代码：

```
#define N_REFS
for (i = 0; i < len; i++) {
  if ((*env)->PushLocalFrame(env, N_REFS) < 0) {
  }
  jstr = (*env)->GetObjectArrayElement(env, arr, i);
  (*env)->PopLocalFrame(env, NULL);
}
```

PushLocalFram为指定数目的局部引用，创建一个新的作用域，PopLocalFram摧毁最上层的作用域，并且释放该域中的所有局部引用。

使用这两个函数的好处是它们可以管理局部引用的生命周期，而不需关系在执行过程中可能被创建的每个单独局部引用。例子中，如果处理jstr的过程，创建了额外的局部引用，它们也会在PopLocalFram之后被立即释放。

NewLocalRef函数，在你写一个工具函数时，非常有用。这个会在下面章节——管理引用的规则，具体分析。

native代码可能会创建超出16个局部引用的范围，也可能将他们保存在PushLocalFram或者EnsureLocalCapacity调用，VM会为局部引用分配所需要的内存。然而，这些内存是否足够，是没有保证的。如果内存分配失败，虚拟机将会退出。

释放全局引用

在native代码不再需要访问一个全局引用的时候，应该调用DeleteGlobalRef来释放它。如果调用这个函数失败，Java VM将不会回收对应的对象。

在native代码不在需要访问一个弱全局引用的时候，应该调用DeleteWeakGlobalRef来释放它。如果调用这个函数失败了，java VM 仍旧将会回收对应的底层对象，但是，不会回收这个弱引用本身所消耗掉的内存。

管理引用的规则

管理引用的目的是为了清除不需要的内存占用和对象保留。

总体来说，只有两种类型的native代码：直接实现native方法的函数，在二进制上下文中被使用的工具函数。

在写native方法的实现的时候，需要当心在循环中过度创建局部引用，以及在native方法中被创建的，却不返回给调用者的局部引用。在native方法方法返回后还留有16个局部引用在使用中，将它们交给java VM来释放，这是可以接受的。但是native方法的调用，不应该引起全局引用和弱全局引用的累积。应为这些引用不会在native方法返后被自动地释放。

在写工具函数的时候，必须要注意不能泄露任何局部引用或者超出该函数之外的执行。因为一个工具函数，可能在意料之外的上下文中，被不停的重复调用。任何不需要的引用创建都有可能导致内存泄露。
1）当一个返回一个基础类型的工具函数被调用，它必须应该没有局部引用、若全局引用的累积。
2）当一个返回一个引用类型的工具函数被调用，它必须应该没有局部、全局或若全局引用的累积，除了要被作为返回值的引用。

一个工具函数以捕获为目的创建一些全局或者弱全局引用，这是可接受的，因为只有在最开始的时候，才会创建这些引用。

如果一个工具函数返回一个引用，你应该使返回的引用的类型(例如局部引用、全局引用）作为函数规范的一部分。它应该始终如一，而不是有时候返回一个局部引用，有时候却返回一个全局引用。调用者需要知道工具函数返回的引用的类型，以便正确地管理自己的JNI引用。以下代码重复地调用一个工具工具函数（GetInfoString）。我们需要知道GetInfoString返回的引用的类型，以便释放该引用：

```
while (JNI_TRUE) {
  jstring infoString = GetInfoString(info);
}
```

在java2 SDK1.2中，NewLocalRef函数可以用来保证一个工具函数一直返回一个局部引用。为了说明这个问题，我们对MyNewString做一些改动,它缓存了一个被频繁请求的字符串（“CommonString”）到全局引用：

jstring

```
MyNewString(JNIEnv *env, jchar *chars, jint len)
{
  static jstring result;

  if (wstrncmp("CommonString", chars, len) == 0) {

    static jstring cachedString = NULL;
    if (cachedString == NULL) {
      cachedString =
        (*env)->NewGlobalRef(env, cachedStringLocal);
    }
    return (*env)->NewLocalRef(env, cachedString);
  }
  return result;
}
```

正常的流程返回的时候局部引用。就像之前解释的那样，我们必须将缓存字符保存到一个全局引用中，这样就可以在多个线程中调用native方法时，都能访问它。

```
return` `(*env)->NewLocalRef(env, cachedString);
```

这条语句，创建了一个局部引用，它指向了缓存在全局引用的指向的统一对象。作为和调用者的约定的一部分，MyNewString总是返回一个局部引用。

PushLocalFram、PopLocalFram函数用来管理局部引用的生命周期特别得方便。只需要在native函数的入口调用PushLocalFram，在函数退出时调用PopLocalFram，局部变量就会被释放。

```
jobject f(JNIEnv *env, ...)
{
  jobject result;
  if ((*env)->PushLocalFrame(env, 10) < 0) {

    return NULL;
  }

  result = ...;
  if (...) {

    result = (*env)->PopLocalFrame(env, result);
    return result;
  }

  result = (*env)->PopLocalFrame(env, result);
  /* normal return */
```

```
    return result;
}
```

PopLocalFram函数调用失败时，可能会导致未定义的行为，例如VM崩溃。

内存泄漏问题

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。

多种原因可能导致 Java Heap 内存泄漏。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。对这个话题的讨论是本文的重点。

JNI 编程实现了 native code 和 Java 程序的交互，因此 JNI 代码编程既遵循 native code 编程语言的编程规则，同时也遵守 JNI 编程的文档规范。在内存管理方面，native code 编程语言本身的内存管理机制依然要遵循，同时也要考虑 JNI 编程的内存管理。

本章简单概括 JNI 编程中显而易见的内存泄漏。从 native code 编程语言自身的内存管理，和 JNI 规范附加的内存管理两方面进行阐述。

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。

总结

至此我们把JNI规范中的三种引用都进行了一个简单的介绍，在此我对以上内容做一个简单总结：

1、局部引用是Native代码中最常用的引用。大部分局部引用都是通过JNI API返回来创建，也可以通过调用NewLocalRef来创建。另外强烈建议Native函数返回值为局部引用。局部引用只在当前调用上下文中有效，所以局部引用不能用Native代码中的静态变量和全局变量来保存。另外时刻要记着Java虚拟机局部引用的个数是有限的，编程的时候强烈建议调用EnsureLocalCapacity，PushLocalFrame和PopLocalFrame来确保Native代码能够获得足够的局部引用数量。

2、全局变量必须要通过NewGlobalRef创建，通过DeleteGlobalRef删除。主要用来缓存Field ID和Method ID。全局引用可以在多线程之间共享其指向的对象。在C语言中以静态变量和全局变量来保存。

3、全局引用和局部引用可以阻止Java虚拟机回收其指向的对象。

4、弱全局引用必须要通过NewWeakGlobalRef创建，通过DeleteWeakGlobalRef销毁。可以在多线程之间共享其指向的对象。在C语言中通过静态变量和全局变量来保持弱全局引用。弱全局引用指向的对象随时都可能会被Java虚拟机回收，所以使用的时候需要时刻注意检查其有效性。弱全局引用经常用来缓存jclass对象。

5、全局引用和弱全局引用可以在多线程中共享其指向对象，但是在多线程编程中需要注意多线程同步。强烈建议在JNI_OnLoad初始化全局引用和弱全局引用，然后在多线程中进行读全局引用和弱全局引用，这样不需要对全局引用和弱全局引用同步（只有读操作不会出现不一致情况）。