





跟上面的JNI空方法是一个数量级。而如果每次都根据名称查找class和field的话，性能要下降高达**40倍**。读取一个字段值的性能在百万级上，在交互频繁的JNI应用中是不能忍受的。消耗时间最多的就是查找class，因此在native里保存class和member id是很有必要的。class和member id在一定范围内是稳定的，但在动态加载的class loader下，保存全局的class要么可能失效，要么可能造成无法卸载classloader,在诸如OSGI框架下的JNI应用还要特别注意这方面的问题。在读取字段值和查找FieldID上，JDK1.4和1.5、 1.6的差距是非常明显的。但在最耗时的查找class上，三个版本没有明显差距。

通过上面的测试可以明显的看出，在调用JNI接口获取方法ID、字段ID和Class引用时，如果没用使用缓存的话，性能低至4倍。所以在JNI开发中，合理的使用缓存技术能给程序提高极大的性能。缓存有两种，分别为使用时缓存和类静态初始化时缓存，区别主要在于缓存发生的时刻。

## 使用时缓存

字段ID、方法ID和Class引用在函数当中使用的同时就缓存起来。下面看一个示例：

```
1 package com.study.jnilearn;
2
3 public class AccessCache {
4
5     private String str = "Hello";
6
7     public native void accessField(); // 访问str成员变量
8     public native String newString(char[] chars, int len); // 根据字符数组和指定长度创建String对象
9
10    public static void main(String[] args) {
11        AccessCache accessCache = new AccessCache();
12        accessCache.nativeMethod();
13        char chars[] = new char[7];
14        chars[0] = "中";
15        chars[1] = "华";
16        chars[2] = "人";
17        chars[3] = "民";
18        chars[4] = "共";
19        chars[5] = "和";
20        chars[6] = "国";
21        String str = accessCache.newString(chars, 6);
22        System.out.println(str);
23    }
24
25    static {
26        System.loadLibrary("AccessCache");
27    }
28 }
```

javah生成的头文件：com\_study\_jnilearn\_AccessCache.h

25922

Android NDK开发Crash错误定位 (http://blog.csdn.net/xyang81/article/details/42319789)  
24940



返回顶部

- 11
- 
- 
- 
- 

```
1  /* DO NOT EDIT THIS FILE - it is machine generated */
2  #include <jni.h>
3  /* Header for class com_study_jnilearn_AccessCache */
4  #ifndef _Included_com_study_jnilearn_AccessCache
5  #define _Included_com_study_jnilearn_AccessCache
6  #ifdef __cplusplus
7  extern "C" {
8  #endif
9  /*
10   * Class:   com_study_jnilearn_AccessCache
11   * Method:  accessField
12   * Signature: (JV
13   */
14   JNIEXPORT void JNICALL Java_com_study_jnilearn_AccessCache_accessField(JNIEnv *, jobject);
15
16   /*
17   * Class:   com_study_jnilearn_AccessCache
18   * Method:  newString
19   * Signature: ([C)Ljava/lang/String;
20   */
21   JNIEXPORT jstring JNICALL Java_com_study_jnilearn_AccessCache_newString(JNIEnv *, jobject,
22   jcharArray, jint);
23
24   #ifdef __cplusplus
25   }
26   #endif
27   #endif
```

实现头文件中的函数：AccessCache.c



内容举报

返回顶部

```
1  // AccessCache.c
2  #include "com_study_jnilearn_AccessCache.h"
3
4  JNIEXPORT void JNICALL Java_com_study_jnilearn_AccessCache_accessField
5  (JNIEnv *env, jobject obj)
```





11



```
6 {
7 // 第一次访问时将字段存到内存数据区，直到程序结束才会释放，可以起到缓存的作用
8 static jfieldID fid_str = NULL;
9 jclass cls_AccessCache;
10 jstring j_str;
11 const char *c_str;
12 cls_AccessCache = (*env)->GetObjectClass(env, obj); // 获取该对象的Class引用
13 if (cls_AccessCache == NULL) {
14     return;
15 }
16
17 // 先判断字段ID之前是否已经缓存过，如果已经缓存过则不进行查找
18 if (fid_str == NULL) {
19     fid_str = (*env)->GetFieldID(env, cls_AccessCache, "str", "Ljava/lang/String;");
20
21     // 再次判断是否找到该类的str字段
22     if (fid_str == NULL) {
23         return;
24     }
25 }
26
27 j_str = (*env)->GetObjectField(env, obj, fid_str); // 获取字段的值
28 c_str = (*env)->GetStringUTFChars(env, j_str, NULL);
29 if (c_str == NULL) {
30     return; // 内存不够
31 }
32 printf("In C:\n str = \"%s\\n", c_str);
33 (*env)->ReleaseStringUTFChars(env, j_str, c_str); // 释放从JVM新分配字符串的内存空间
34
35 // 修改字段的值
36 j_str = (*env)->NewStringUTF(env, "12345");
37 if (j_str == NULL) {
38     return;
39 }
40 (*env)->SetObjectField(env, obj, fid_str, j_str);
41
42 // 释放本地引用
43 (*env)->DeleteLocalRef(env, cls_AccessCache);
44 (*env)->DeleteLocalRef(env, j_str);
45 }
46
47 JNIEXPORT jstring JNICALL Java_com_study_jnilearn_AccessCache_newString
48 (JNIEnv *env, jobject obj, jcharArray j_char_arr, jint len)
49 {
50     jcharArray elemArray;
51     jchar *chars = NULL;
52     jstring j_str = NULL;
53     static jclass cls_string = NULL;
54     static jmethodID cid_string = NULL;
55     // 注意：这里缓存局引用的做法是错误的，这里做为一个反面教材提醒大家，下面会说到。
56     if (cls_string == NULL) {
57         cls_string = (*env)->FindClass(env, "java/lang/String");
58         if (cls_string == NULL) {
59             return NULL;
60         }
61     }
62     jmethodID cid_getchar = (*env)->GetMethodID(cls_string, "charAt", "I");
63     jmethodID cid_getlength = (*env)->GetMethodID(cls_string, "length", "I");
64     jmethodID cid_newstringUTF = (*env)->GetMethodID(cls_string, "newStringUTF", "Ljava/lang/String;");
65     jmethodID cid_setstringUTF = (*env)->GetMethodID(cls_string, "setStringUTF", "Ljava/lang/String;I");
66     elemArray = (*env)->GetObjectArrayElement(obj, 0);
67     if (elemArray == NULL) {
68         return NULL;
69     }
70     jchar *chars = (*env)->GetStringChars(elemArray, NULL);
71     if (chars == NULL) {
72         return NULL;
73     }
74     jstring j_str = (*env)->NewStringUTF(env, "12345");
75     if (j_str == NULL) {
76         return NULL;
77     }
78     (*env)->SetObjectField(obj, cid_string, j_str);
79     (*env)->DeleteLocalRef(env, j_str);
80     return j_str;
81 }
```



内容举报

返回顶部



11



```
59     return NULL;
60 }
61 }
62
63 // 缓存String的构造方法ID
64 if (cid_string == NULL) {
65     cid_string = (*env)->GetMethodID(env, cls_string, "<init>", "([C)V");
66     if (cid_string == NULL) {
67         return NULL;
68     }
69 }
70
71 printf("In C Array Len: %d\n", len);
72 // 创建一个字符数组
73 elemArray = (*env)->NewCharArray(env, len);
74 if (elemArray == NULL) {
75     return NULL;
76 }
77
78 // 获取数组的指针引用，注意：不能直接将jcharArray作为SetCharArrayRegion函数最后一个参数
79 chars = (*env)->GetCharArrayElements(env, j_char_arr, NULL);
80 if (chars == NULL) {
81     return NULL;
82 }
83
84 // 将Java字符串组中的内容复制指定长度到新的字符串组中
85 (*env)->SetCharArrayRegion(env, elemArray, 0, len, chars);
86
87 // 调用String对象的构造方法，创建一个指定字符串组为内容的String对象
88 j_str = (*env)->NewObject(env, cls_string, cid_string, elemArray);
89
90 // 释放本地引用
91 (*env)->DeleteLocalRef(env, elemArray);
92
93 return j_str;
94 }
```

例1、在Java\_com\_study\_jnilearn\_AccessCache\_accessField函数中的第8行定义了一个静态变量fid\_str用于存储字段的ID，每次调用函数的时候，在第18行先判断字段ID是否已经缓存，如果没有先取出来存到fid\_str中，下次再调用的时候该变量已经有值了，不用再去找JVM中获取，起到了缓存的作用。

例2、在Java\_com\_study\_jnilearn\_AccessCache\_newString函数中的53和54行定义了两个变量cls\_string和cid\_string，分别用于存储java.lang.String类的Class引用和String的构造方法ID。在56行和64行处，使用前会先判断是否已经缓存过，如果没有则调用JNI的接口从JVM中获取String的Class引用和构造方法ID存储到静态变量当中。下次再调用该函数时就可以直接使用，不需要再去找一次了，也达到了缓存的效果，大家第一反映都会这么认为。但是请注意：cls\_string是一个局部引用，与方法 and 字段ID不一样，局部引用在函数结束后会被VM自动释放掉，这时cls\_string成为了一个野针对（指向的内存空间已被释放，但变量的值仍然是被释放后的内存地址，不为NULL），下次再调用Java\_com\_xxxx\_newString这个函数的时候，会试图访问一个无效的局部引用，从而导致非法的内存访问造成程序崩溃。所以在函数内用static缓存局部引用这种方式是错误的。下篇文章会介绍局部引用和全局引用，利用全局引用来防止这种问题，请关注。

本文链接：<a href="http://blog.csdn.net/xyang81/article/details/44279725">http://blog.csdn.net/xyang81/article/details/44279725</a>



内容举报



返回顶部



11



# 类静态初始化缓存

在调用一个类的方法或属性之前，Java虚拟机会先检查该类是否已经加载到内存当中，如果没有则会先加载，然后紧接着会调用该类的静态初始化代码块，所以在静态初始化该类的过程当中计算并缓存该类当中的字段ID和方法ID也是个不错的选择。下面看一个示例：

```
1 package com.study.jnilearn;
2
3 public class AccessCache {
4
5     public static native void initIDs();
6
7     public native void nativeMethod();
8     public void callback() {
9         System.out.println("AccessCache.callback invoked!");
10    }
11
12    public static void main(String[] args) {
13        AccessCache accessCache = new AccessCache();
14        accessCache.nativeMethod();
15    }
16
17    static {
18        System.loadLibrary("AccessCache");
19        initIDs();
20    }
21 }
```



内容举报

返回顶部

内容举报

返回顶部



11



```
1  /* DO NOT EDIT THIS FILE - it is machine generated */
2  #include <jni.h>
3  /* Header for class com_study_jnilearn_AccessCache */
4  #ifndef _Included_com_study_jnilearn_AccessCache
5  #define _Included_com_study_jnilearn_AccessCache
6  #ifdef __cplusplus
7  extern "C" {
8  #endif
9  /*
10   * Class:   com_study_jnilearn_AccessCache
11   * Method:  initIDs
12   * Signature: ()V
13   */
14   JNIEXPORT void JNICALL Java_com_study_jnilearn_AccessCache_initIDs
15   (JNIEnv *, jclass);
16
17   /*
18   * Class:   com_study_jnilearn_AccessCache
19   * Method:  nativeMethod
20   * Signature: ()V
21   */
22   JNIEXPORT void JNICALL Java_com_study_jnilearn_AccessCache_nativeMethod
23   (JNIEnv *, jobject);
24
25   #ifdef __cplusplus
26   }
27   #endif
28   #endif
```



内容举报

返回顶部



11



```
1  // AccessCache.c
2
3  #include "com_study_jnilearn_AccessCache.h"
4
5  jmethodID MID_AccessCache_callback;
6
7  JNIEXPORT void JNICALL Java_com_study_jnilearn_AccessCache_initIDs
8  (JNIEnv *env, jclass cls)
9  {
10   printf("initIDs called!!!\n");
11   MID_AccessCache_callback = (*env)->GetMethodID(env,cls,"callback","()V");
```







```
12 }
13
14 JNIEXPORT void JNICALL Java_com_study_jnilearn_AccessCache_nativeMethod
15 (JNIEnv *env, jobject obj)
16 {
17     printf("In C Java_com_study_jnilearn_AccessCache_nativeMethod called!!!\n");
18     (*env)->CallVoidMethod(env, obj, MID_AccessCache_callback);
19 }
```

JVM加载AccessCache.class到内存当中之后，会调用该类的静态初始化代码块，即static代码块，先调用System.loadLibrary加载动态库到JVM中，紧接着调用native方法initIDs，会调用用到本地函数Java\_com\_study\_jnilearn\_AccessCache\_initIDs，在该函数中获取需要缓存的ID，然后存入全局变量当中。下次需要用到这些ID的时候，直接使用全局变量当中的即可，如18行当中调用Java的callback函数。

```
1 (*env)->CallVoidMethod(env, obj, MID_AccessCache_callback);
```

## 两种缓存方式比较


如果在写JNI接口时，不能控制方法和字段所在类的源码的话，用使用时缓存比较合理。但比起类静态初始化时缓存来说，用使用时缓存有一些缺点：

- 1. 使用前，每次都需要检查是否已经缓存该ID或Class引用
- 2. 如果在用使用时缓存的ID，要注意只要本地代码依赖于这个ID的值，那么这个类就不会被unload。另外一方面，如果缓存发生在静态初始化时，当类被unload或reload时，ID会被重新计算。因为，尽量在类静态初始化时就缓存字段ID、方法ID和类的Class引用。

版权声明：本文为博主原创文章，未经博主允许不得转载。  
本文已收录于以下专栏：JNI/NDK开发指南 (<http://blog.csdn.net/column/details/blogjndk.html>)




([http://my.csdn.net/weixin\\_35068028](http://my.csdn.net/weixin_35068028))

 xq325 (/xq325) 2015-09-07 14:47 2楼

(/xq325)用耗时结果的单位应该是微妙吧，你这个毫秒差的太多了吧

回复 1条回复

 xdcj2012 (/xdcj2012) 2015-03-27 17:04 1楼

(/xdcj2012)持续跟进中

回复

### 相关文章推荐

JNI与JNA性能比较 (<http://blog.csdn.net/Drifter.1/article/details/7841810>)



 内容举报

 返回顶部



11



在介绍JNA时，提到了JNA是基于JNI的，是在JNI上封装了一层，JNI性能不如JNA。最近在网看到篇简单的比较这两者性能

能的文档，感觉不错，现转载一下： 分别用JNI和JNA的方式建立dll，d...

DrifterJ (http://blog.csdn.net/DrifterJ) 2012年08月08日 10:57 14141

JNI DETECTED ERROR IN APPLICATION解决记录 (http://blog.csdn.net/HuntCode/article/...

最近遇到一个JNI的问题，同一套代码在Android4.4版本前的设备上运行是OK的，但是在Android5.0之后的设备上就会崩溃，查看logcat发现报JNI DETECTED ERROR IN ...

HuntCode (http://blog.csdn.net/HuntCode) 2015年08月13日 14:23 23592



一个普通程序员的内心独白....躺枪！躺枪！

我，一个普通通程序员，没有过人的天赋，没有超乎寻常的好运，该如何逆袭走上人生巅峰？

广告

(http://www.baidu.com/cb.php?c=lgF\_pyfqHmknjDLnjT0IZ0qnfK9ujYzP1nsrjD10Aw-5Hc3rHnYnHb0TAq15HfLPWRznjb0T1YznWnzmvf4PAwBmHnduH6v0AwY5HDdnHc4njD4njf0lgF\_5y9YIZ0IQzq-uZR8mLPbUB48ugfEXyN9T-KzUvdEIA-EUBqbugw9pysEn1qdIAdxTvqdThP-5yF\_UvTkn0KzujYk0AFV5H00TZcqn0KdpyfqHRLPjnvnfKEpyfqHc4rj6kP0KWpyfqP1cvrHnz0AqLUWYs0ZK45HcsP...

android jni基本使用方式 (http://blog.csdn.net/H291850336/article/details/50942468)

JNI是Java Native Interface的缩写，它提供了若干的API实现了Java和其他语言的通信（主要是C&C++）一旦使用JNI，JAVA程序就丧失了JAVA平台的两个优点：1、程序...

H291850336 (http://blog.csdn.net/H291850336) 2016年03月21日 09:37 1075



Delphi7高级应用开发随书源码 (http://download.csdn.net/download/chenx...

2003年04月30日 00:00 676KB 下载

Android开发艺术探索》综合技术，JNI和性能优化小结 (http://blog.csdn.net/doom20082004...

1. 当crash发生时，系统就会调用UncaughtExceptionHandler的uncaughtException方法，可以读取到异常信息；实现UncaughtExceptionHandle...

doom20082004 (http://blog.csdn.net/doom20082004) 2016年12月08日 14:04 263



人人都能看懂的 AI 入门课

本课程将讲述人工智能的现状、应用场景和入门方法，并通过运用 TensorFlow，使得受众能清晰了解人工智能的运作方式。

(http://www.baidu.com/cb.php?c=lgF\_pyfqHmknjfrjc0IZ0qnfK9ujYzP1f4Pjn10Aw-5Hc4nj6vPjm0TAq15Hf4rjn1n1b0T1dBnWbdPvRznH6zrAFYmvmnL0AwY5HDdnHc4njD4njf0lgF\_5y9YIZ0IQzqMpgwBUvqoQhP8QvGIAPCmgfEmvq\_lyd8Q1R4uWi-n16kPWKWwRHnnvHRvnnvNBuYD4PHqdIAdxTvqdThP-5HDknWFWmhkEusKzujYk0AFV5H00TZcqn0KdpyfqHRLPjnvnfKEpyfqHnsnj0YnsKWpyfqP1cvrHnz0AqLUWYs0ZK45HcsP6KWThnqrHnLPs)

JNI线程 (http://blog.csdn.net/shanhuazun/article/details/43140103)



内容举报



返回顶部



内容举报

[illegible]

JNI多个本地线程进入Java层,Java层线程进入C本地函数,线程安全

shaohuazuo (<http://blog.csdn.net/shaohuazuo>) 2015年01月29日 19:15 3200



如何在多线程中使用JNI？(<http://blog.csdn.net/booirror/article/details/37778283>)

如果你想了解JNI在如何在多线程下使用如果你在子线程使用JNI时遇到findClass不能找到目标Class，而在主线程下却能找到该Class的问题。或是GetEnv返回NULL的问题如果你想多学点编...

booirror (<http://blog.csdn.net/booirror>) 2014年07月15日 00:08 8333

**JNI 各种优化方案** (<http://blog.csdn.net/a15874647/article/details/9883419>)

转自：<http://www.ibm.com/developerworks/cn/java/j-jni/> 最好看原版吧，排版非常好，这里粘贴是为了备份一下！...

 a15874647 (<http://blog.csdn.net/a15874647>) 2013年08月10日 19:12  1423

Java、Android超精确测量代码执行时间差 (<http://blog.csdn.net/brandon2015/article/detail...>)

平时产生随机数时我们经常拿时间做种子，比如用System.currentTimeMillis的结果，但是在执行一些循环中使用了System.currentTimeMillis，那么每次的结果将会差别很...

brandon2015 (<http://blog.csdn.net/brandon2015>) 2016年01月18日 14:01 2605

windows下Java JNI测试Demo ([http://blog.csdn.net/shen\\_jz2012/article/details/50849788](http://blog.csdn.net/shen_jz2012/article/details/50849788))

关于Linux下使用gcc编译动态库.so文件在上一篇已经介绍过，现在来讲讲如何在windows平台下。前面很多步骤都跟在linux环境下一样，javah生成头文件，建立java程序。区别就是，现在在...

shen\_jz2012 ([http://blog.csdn.net/shen\\_jz2012](http://blog.csdn.net/shen_jz2012)) 2016年03月10日 20:49 778

Android JNI用于驱动测试 (<http://blog.csdn.net/wu20093346/article/details/37603099>)

硬件平台: S3C6410 操作系统: Ubuntu、windows 板子系统: Android 开发工具: jdk, ndk, eclipse 本次测试从linux内核模块编译开始, 以S3C6410的pwm驱动...

wu20093346 (<http://blog.csdn.net/wu20093346>) 2014年07月11日 13:39 1842

关于android的JNI几点注意问题。 (<http://blog.csdn.net/u013282523/article/details/17576823>)

1.注册函数映射表 JNI API为了避免丑陋的函数名,提供了方法向Java虚拟机注册函数映射表。 这样当Java调用Native接口的时候,Java虚拟机就可以不用根...

u013282523 (<http://blog.csdn.net/u013282523>) 2013年12月26日 09:49 581

## 利用JNI调用C++函数的测试 ([http://blog.csdn.net/Running\\_J/article/details/52103532](http://blog.csdn.net/Running_J/article/details/52103532))

开发平台介绍: VS2013, Eclipse 步骤如下: 1、eclipse下新建一个javaproject, 编写.class文件如下: package jni.exercise; public cl...



[TOP](#)

 内容举报

 [返回顶部](#)

<http://blog.csdn.net/xyang81/article/details/44279725>

11/12



11



11



Running\_J ([http://blog.csdn.net/Running\\_J](http://blog.csdn.net/Running_J)) 2016年08月03日 14:48 464

**LIBPNG读写PNG图像 (<http://blog.csdn.net/Augusdi/article/details/10427879>)**

//file.pngtest.c //changed from the libpng,对照libpng中源码阅读 //myers #include "png.h" #include #include...

Augusdi (<http://blog.csdn.net/Augusdi>) 2013年08月28日 09:34 7366

**Drawable资源——LayerDrawable 图层列表 (<http://blog.csdn.net/reflse/article/details/5130...>)**

Drawable资源——LayerDrawable 图层列表 1, 认识 它表示一种层次化的Drawable集合, 通过将不同的Drawable放置在不同的层上面从而达到一种叠加后的效果。系统将会...

refse (<http://blog.csdn.net/reflse>) 2016年05月03日 13:14 392

**JNI/NDK开发指南（八）---JNI调用性能测试及优化 (<http://blog.csdn.net/yishifu/article/detai...>)**

在前面几章我们学习到了, 在Java中声明一个native方法, 然后生成本地接口的函数原型声明, 再用C/C++实现这些函数, 并生成对应平台的动态共享库放到Java程序的类路径下, 最后在Java程序中调用...

yishifu (<http://blog.csdn.net/yishifu>) 2016年08月10日 14:13 748

**JNI/NDK开发指南（八）——调用构造方法和父类实例方法 (<http://blog.csdn.net/xyang81/art...>)**

转载请注明出处: <http://blog.csdn.net/xyang81/article/details/44002089>在第6章我们学习到了在Native层如何调用Java静态方法和实例方法, 其中调...

xyang81 (<http://blog.csdn.net/xyang81>) 2015年03月01日 21:18 5851

**JNI/NDK开发指南（七）——C/C++访问Java实例变量和静态变量 (<http://blog.csdn.net/xyan...>)**

在上一章中我们学习到了如何在本地代码中访问任意Java类中的静态方法和实例方法, 本章我们也通过一个示例来学习Java中的实例变量和静态变量, 在本地代码中如何来访问。静态变量也称为类变量（属性），在所有...

xyang81 (<http://blog.csdn.net/xyang81>) 2015年01月18日 21:37 7526

**JNI/NDK开发指南（四）——字符串处理 (<http://blog.csdn.net/JavaerDev/article/details/441...>)**

从(三)中可以看出JNI中的基本类型和Java中的基本类型都是一一对应的, 接下来先看一下JNI的基本类型定义: typedef unsigned char jboolean; typedef u...

JavaerDev (<http://blog.csdn.net/JavaerDev>) 2015年03月09日 13:05 400

**JNI/NDK开发指南（二）——JVM查找java native方法的规则 (<http://blog.csdn.net/JavaerDe...>)**

通过第一篇文章, 大家明白了调用native方法之前, 首先要调用System.loadLibrary接口加载一个实现了native方法的动态库才能正常访问, 否则就会抛出java.lang.Unsatis...

JavaerDev (<http://blog.csdn.net/JavaerDev>) 2015年03月06日 14:00 539



内容举报

返回顶部