<https://blog.csdn.net/hty1053240123/article/details/79226569>

**终于建了一个自己个人小站：**[**https://huangtianyu.gitee.io**](https://huangtianyu.gitee.io/)**，以后优先更新小站博客，欢迎进站，O(∩\_∩)O~~**

### 1. 简介

从Java虚拟机创建的对象当传入到native层时会产生一个引用，在进行垃圾回收时如果有native的引用，改对象同样也不会被回收。在native引用中分局部引用和全局引用。

#### 1.1 局部引用

局部引用又称本地引用，大多数见到的引用都是局部引用，例如通过NewLocalRef和各种JNI接口创建（FindClass、NewObject、GetObjectClass和NewCharArray等），局部引用只会在本次native调用中有效，当本次调用结束后该引用即被自动释放。局部引用会阻止GC进行回收。同时也可以调用DeleteLocalRef函数来手动释放（比如在循环里面用到了局部引用而退出循环没有使用该局部引用，那么就需要在循环中释放该局部引用）。通常使用NewObject创建的实例返回的也是局部引用。千万不要把局部引用保存为c++的全局变量或者把它定义为静态变量，局部引用的有效期是一次Java本地调用。   
JNI提供了一系列函数来管理局部引用的生命周期。这些函数包括：EnsureLocalCapacity、NewLocalRef、PushLocalFrame、PopLocalFrame、DeleteLocalRef。

//申请扩充局部引用的最大个数限制，返回值等于0的时候表示成功,>0时表示内存溢出。默认至少16个局部引用可以使用，引用数超出时报FatalError

jint (\*EnsureLocalCapacity)(JNIEnv\*, jint);

/\* PushLocalFrame是一个创建本地引用新作用域的有用函数，这使得PushLocalFrame函数可以释放其使用的框架中所有已分配的本地引用。当该函数被调用时，本地引用的最低数量将在本框架中被创建。该函数如果执行成功则返回0，如果由于错误抛出一个OutOfMemoryException，则返回一个负值。\*/

jint PushLocalFrame(jint capacity);

/\*PopLocalFrame函数释放当前框架中的所有本地引用（弹出一个框架）。因为存储该函数的结果（返回值）可能会导致在即将被弹出的框架中创建一个本地引用，该函数接收一个可以导致引用在当前框架被弹出之后的最高框架中创建的参数。这就确保可以维护一个存储PopLocalFrame函数结果的引用。\*/

jobject PopLocalFrame(jobject result);

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8

PushLocalFrame为当前函数中局部引用创建了一个引用堆栈，在每遍历一次调用(\*env)->GetObjectArrayElement(env, arr, i);返回一个局部引用时，JVM会自动将该引用压入当前局部引用栈中。而PopLocalFrame负责将栈中所有引用释放。这样一来，Push/PopLocalFrame函数对提供了对局部引用生命周期更方便的管理，不用再去一个个Delete了。

#### 1.2 全局引用

全局引用可以在当前线程使用，也可以在其他线程使用，可以保存在本地的static静态变量或全局变量中，全局引用需要调用NewGlobalRel函数创建，释放时采用ReleaseGlobalRef函数释放。有效作用域在创建后，一直到调用ReleaseGlobalRef释放时。

#### 1.3 弱全局引用

在Java1.2中，新增了弱全局引用，与全局变量一样其创建、删除均需要编程写出，也可以在本地多个代码中使用，也可以跨进程使用。不一样的是，它的存在不影响垃圾回收机制对该引用所指向对象实例的回收。其创建采用NewWeakGlobalRef，释放采用ReleaseWeakGlobalRel。

以上涉及的函数主要有以下几个：

//创建局部引用

jobject NewLocalRef(jobject obj);

//释放局部引用

void DeleteLocalRef(jobject obj);

//创建全局引用

jobject NewGlobalRef(jobject obj);

//释放全局引用

void DeleteGobalRef(jobject obj);

//创建弱全局引用

jobject NewWeakGlobalRef(jobject obj);

//释放弱全局引用

void DeleteWeakGlobalRef(jobject obj);

//该方法判断两个引用是否相等，对于弱全局引用如果对比的是NULL那么还可以判断该引用指向的对象是否被回收

jboolean IsSameObject(jobject obj1 , jobject obj2);

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

上述三中引用会影响内存的回收，在C/C++中没有向Java一样的垃圾回收机制，自己申请的内存要记得自己去释放了，否则会导致内存泄漏。虽然现在C/C++里面也有智能指针，但相对而言这个智能指针用起来不如Java。所以在C的世界里要遵循谁申请，谁释放的基本原则。

### 2. 举个栗子

上面介绍了基本知识，下面给出相应的例子来进行说明下。

#### 2.1局部引用

//1. 局部引用不要存储在static变量中，即使存了下次也不能用

//static jclass cls;

//以下创建的局部引用都放入到栈中

env->PushLocalFrame(16);

jclass cls;

if (!cls) {//这里就错误了，前一次方法完成后jvm会释放局部引用，这里static存的值仅第一次有效

cls = env->GetObjectClass(instance);//这里的cls是局部引用

}

//删除栈里面的局部引用

env->PopLocalFrame(NULL);

env->EnsureLocalCapacity(20);//将本地引用的最大限制改为20

//下面可以进行其他操作。。。

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

在局部引用中要注意以下几方面：   
1. 循环体内创建的局部引用，要在循环体内就直接释放了。   
2. 编写的工具函数，里面创建的局部引用，要在该工具函数里面释放了。   
3. 局部引用引用了一个大的Java对象，这时候一定一定要早点释放了。   
4. 局部引用不要缓存在native层

#### 2.2 全局引用

extern "C"

JNIEXPORT void JNICALL

Java\_zqc\_com\_example\_NativeTest\_jniGlobalRef(JNIEnv \*env, jobject instance) {

static jobject obj;

static jclass pCls;

if (obj) {//第二次点击时，这里就不会空

//由于obj和personCls被保存为全局引用了，所有这里使用仍然有效

jmethodID getId = env->GetMethodID(pCls, "getName", "()Ljava/lang/String;");

jstring name = (jstring) env->CallObjectMethod(obj, getId);

LOGE("obj is not null, name:%s", jstringToChar(env, name));

return;

}

if (!pCls) {//为空就去新建

jclass tmpCls = env->FindClass("zqc/com/example/Person");

pCls = (jclass) env->NewGlobalRef(tmpCls);

env->DeleteLocalRef(tmpCls);

}

jmethodID conMid = env->GetMethodID(pCls, "<init>", "()V");

jobject tmpObj = env->NewObject(pCls, conMid);

jmethodID setId = env->GetMethodID(pCls, "setName", "(Ljava/lang/String;)V");

env->CallVoidMethod(tmpObj, setId, env->NewStringUTF("看看姓名"));

obj = env->NewGlobalRef(tmpObj);

env->DeleteLocalRef(tmpObj);

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24

#### 2.3 弱全局引用

弱全局引用和全局引用基本差不多，最大的区别就是弱全局引用不影响GC的回收。在使用弱全局引用的时候一定要注意，使用前要检查下是不是被GC回收了。

extern "C"

JNIEXPORT void JNICALL

Java\_zqc\_com\_example\_NativeTest\_jniWeakGlobalRef(JNIEnv \*env, jobject instance) {

static jclass pCls;

if (!pCls) {

jclass tmpCls = env->FindClass("zqc/com/example/Person");

pCls = (jclass) env->NewWeakGlobalRef(tmpCls);

env->DeleteLocalRef(tmpCls);

}

//除了第一次需要FindClass外，在没有回收pCls之前都可以使用

//这里使用...

//可以手动释放

//env->DeleteWeakGlobalRef(pCls);

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16

### 3. 引用的比较

jni提供了相应的函数

jboolean IsSameObject(jobject ref1, jobject ref2)

{ return functions->IsSameObject(this, ref1, ref2); }

* 1
* 2

如果两个引用指向同一个实例则返回JNI\_TRUE，否则返回JNI\_FALSE。

本文为作者(石岩)原创,转载请指明出处:   
<https://blog.csdn.net/a1002450926/article/details/79812815>

#### ****JAVA层代码****

public class JniTest {

public String key = "key:JniTest";

// 局部引用

public native void localRef();

// 创建全局引用

public native void createGlobalRef();

// 获取全局引用

public native String getGlobalRef();

// 释放全局引用

public native void deleteGlobalRef();

// 异常

public native void exception();

public static void main(String[] args) {

JniTest test = new JniTest();

test.createGlobalRef();

System.out.println(test.getGlobalRef());

test.deleteGlobalRef();

try {

test.exception();

} catch (Exception e) {

System.out.println("Java层捕捉到了异常:" + e.getMessage());

}

}

static {

System.loadLibrary("JniTest");

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37

#### ****C层代码****

#pragma once

#pragma execution\_character\_set("utf-8")

#define \_CRT\_SECURE\_NO\_WARNINGS

#include "com\_shiyan\_jni\_JniTest.h"

#include<string.h>

/\*

局部引用

模拟:循环创建数组

\*/

JNIEXPORT void JNICALL Java\_com\_shiyan\_jni\_JniTest\_localRef

(JNIEnv \*env, jobject jobj){

int i = 0;

for (; i < 10; i++){

jclass cls = (\*env)->FindClass(env, "java/util/Date");

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

jobject jobj = (\*env)->NewObject(env, cls, mid);

//通知垃圾回收期回收对象

(\*env)->DeleteLocalRef(env, jobj);

}

}

//全局引用

jstring global\_str;

/\*

创建全局引用

\*/

JNIEXPORT void JNICALL Java\_com\_shiyan\_jni\_JniTest\_createGlobalRef

(JNIEnv \*env, jobject jobj){

jstring obj = (\*env)->NewStringUTF(env, "hello jni!");

global\_str = (\*env)->NewGlobalRef(env, obj);

}

/\*

获取全局引用

\*/

JNIEXPORT jstring JNICALL Java\_com\_shiyan\_jni\_JniTest\_getGlobalRef

(JNIEnv \*env, jobject jobj){

return global\_str;

}

/\*

释放全局引用

\*/

JNIEXPORT void JNICALL Java\_com\_shiyan\_jni\_JniTest\_deleteGlobalRef

(JNIEnv \*env, jobject jobj){

(\*env)->DeleteGlobalRef(env, global\_str);

}

//弱全局引用

//节省内存,在内存不足时可以释放所引用的对象

//可以引用一个不常用的对象，如果为NULL，临时创建

//创建:NewWeakGlobalRef,销毁:DeleteGlobalWeakRef

/\*

异常处理

\*/

JNIEXPORT void JNICALL Java\_com\_shiyan\_jni\_JniTest\_exception

(JNIEnv \*env, jobject jobj){

jclass cls = (\*env)->GetObjectClass(env, jobj);

jfieldID mid = (\*env)->GetMethodID(env, cls, "key2", "Ljava/lang/String;");

//检测是否发生java异常

jthrowable exception = (\*env)->ExceptionOccurred(env);

if (exception != NULL){

//让JAVA代码可以继续运行

//清空异常信息

(\*env)->ExceptionClear(env);

//补救措施

mid = (\*env)->GetFieldID(env, cls, "key", "Ljava/lang/String;");

}

//获取属性的值

jstring jstr = (\*env)->GetObjectField(env, jobj, mid);

//jstring->c

char \*str = (\*env)->GetStringUTFChars(env, jstr, NULL);

//判断是否相等

if (\_stricmp(str, "key:JniTest2") != 0){

//认为抛出异常，给JAVA层处理

jclass newExcCls = (\*env)->FindClass(env, "java/lang/IllegalArgumentException");

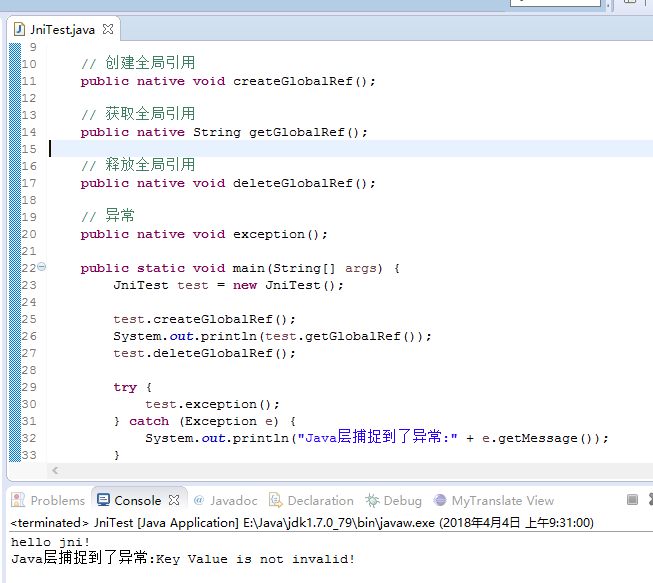
(\*env)->ThrowNew(env, newExcCls, "Key Value is not invalid!");

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* 72
* 73
* 74
* 75
* 76
* 77
* 78
* 79
* 80
* 81
* 82
* 83
* 84
* 85
* 86
* 87
* 88

#### ****结果演示****



* 上一篇 [NDK开发之数组参数和数组返回值](https://blog.csdn.net/a1002450926/article/details/79800440)