JNI/NDK开发指南

极客学院出版

前言

JNI 是 Java 语言提供的 Java 和 C/C++ 相互沟通的机制，Java 可以通过 JNI 调用本地的 C/C++ 代码，本地的 C/C++ 的代码也可以调用 Java 代码。JNI 是本地编程接口，Java 和 C/C++ 互相通过的接口。Java 通过 C/C++ 使用本地的代码的一个关键性原因在于 C/C++ 代码的高效性。 代码和其他语言写的代码进行交互。

NDK 是一系列工具的集合。它提供了一系列的工具，帮助开发者快速开发 C（或C++）的动态库，并能自动将 so 和 Java 应用一起打包成 apk。这些工具对开发者的帮助是巨大的。它集成了交叉编译器，并提供了相应的 mk 文件隔离 CPU、平台、ABI 等差异，开发人员只需要简单修改 mk 文件（指出“哪些文件需要编译”、“编译特性要求”等），就可以创建出 so。它可以自动地将 so 和 Java 应用一起打包，极大地减轻了开发人员的打包工作。

本指南首先会讲 JNI 开发的一些基础知识，每个知识点都会结合一个案例来贯通，最后讲 NDK 开发，NDK 这块主要讲编译环境的配置、Android.mk 的编写，通过示例代码，让读者了解 JNI 技术的原理，帮助开发者搭建 NDK 开发环境。

出处地址 (http://blog.csdn.net/xyang81/article/details/41759643)

适合人群

本教程旨在帮助 Android 开发者或 Java 开发者实现 Java 与 C/C++ 语言的交互。

学习基础

了解 Java 语言，对常用编程工具能够使用(eclipse)。

目录

1

JNI 概述

相信很多做过 Java 或 Android 开发的朋友经常会接触到 JNI 方面的技术，由其做过 Android 的朋友，为了应用的安全性，会将一些复杂的逻辑和算法通过本地代码（C或C++）来实现，然后打包成 .so 动态库文件，并提供 Java 接口供应用层调用，这么做的目的主要就是为了提供应用的安全性，防止被反编译后被不法分子分析应用的逻辑。当然打包成 .so 也不能说完全安全了，只是相对反编译 Java 的 class 字节码文件来说，反汇编 .so 动态库来分析程序的逻辑要复杂得多，没那么容易被破解。比如百度开放平台提供的定位服务、搜索服务、LBS服务、推送服务的Android SDK，除了Java接口的jar包之外，还有一个 .so 文件，这个 .so 就是实现了Java层定义的native接口的动态库。

以前公司有一个 JavaWeb 的项目，其中有一个用户注册的模块，需要验证用户的手机号（流程大家都懂的），由于这个项目的用户量不大，没用采用运营商的短信网关接口，直接采购了一台 16 口的短信猫设备和 SIM 卡来解决这个事情。由于短信猫设备只提供了 C 的接口，而 Java 是不能直接与 C 语言进行交互的，所以 JNI 就派上用场了，先在 Java 层定义好发送短信、接收短信、短信发送队列等相关 native 方法，然后用 javah 命令将定义 Java native 接口的 class 字节码文件生成 .h 头文件（这个后面会讲到），最后用设备场商提供的 C 接口来实现 java 的 native 方法，完了之后编译成 .dll 或 .so 动态库，提供给 Java 程序使用即可。

JNI 在 Cocos2d-x 游戏引擎中也经常用到，该引擎是用纯 C++ 开发的，而且是跨平台的，依托 C++ 的跨平台特性，只需用 C++ 编写一次逻辑，就可以将游戏打包发布到不同的平台（IOS、Android、WinPhone、黑莓、Linux、Windows），打包发布的细节就不在这里讨论了。如果游戏要发布到 Android 平台，开发过程当中，少不了 C++ 层和 Java 层进行交互，比如游戏当中要打开一个网页、播放一段视频或打开一个新的窗口等，这些在 C++ 层实现是非常麻烦的，如果用 Android 应用层提供的 API 就变得相当容易。所以这时就不得不写 JNI 来完成这些功能的需求。当然这些常用的 JNI 操作，Cocos2d-x 引擎进行了封装，相关的接口定义在 JniHelper.cpp这个类中，可以拿来直接使用。（后面会有例子详细介绍）

虽然现在的物联网和智能家居行业还处于萌芽状态，但随着这个时代在技术的创新与不断改进的发展下，想象 5 年后，物联网和智能家居行业真正成熟起来，由于 Android 系统的开源，自然会被各大硬件场商所采用，相当于这几年 Android 智能手机的市场一样，仍然可能会处于移动智能终端的霸主地位。你可能会问，但这和 JNI 和有什么关系呢？当各种设备接入互联网的同时，自然少不了人机交互的应用程序，当应用程序需要调用硬件特定的功能时，此时只能通过 C 或 C++ 封装对应功能的 JNI 接口来供上层应用使用。比如要用手机中的 app 控制家里的电灯、窗帘、冰箱、空调等一切智能的电子设备时，自然少不了应用要和底层硬件进行通讯，至于各种智能设备的运行控制，自然是由厂商来实现，他们只需提供操作设备相关功能的接口即可。虽然厂商会封装好 JNI 接口，但我们也要了解下 JNI 与 Java 通讯的原理，以便我们在开发过程当中遇到问题时，能够快速定位到问题。

2

JNI 开发流程

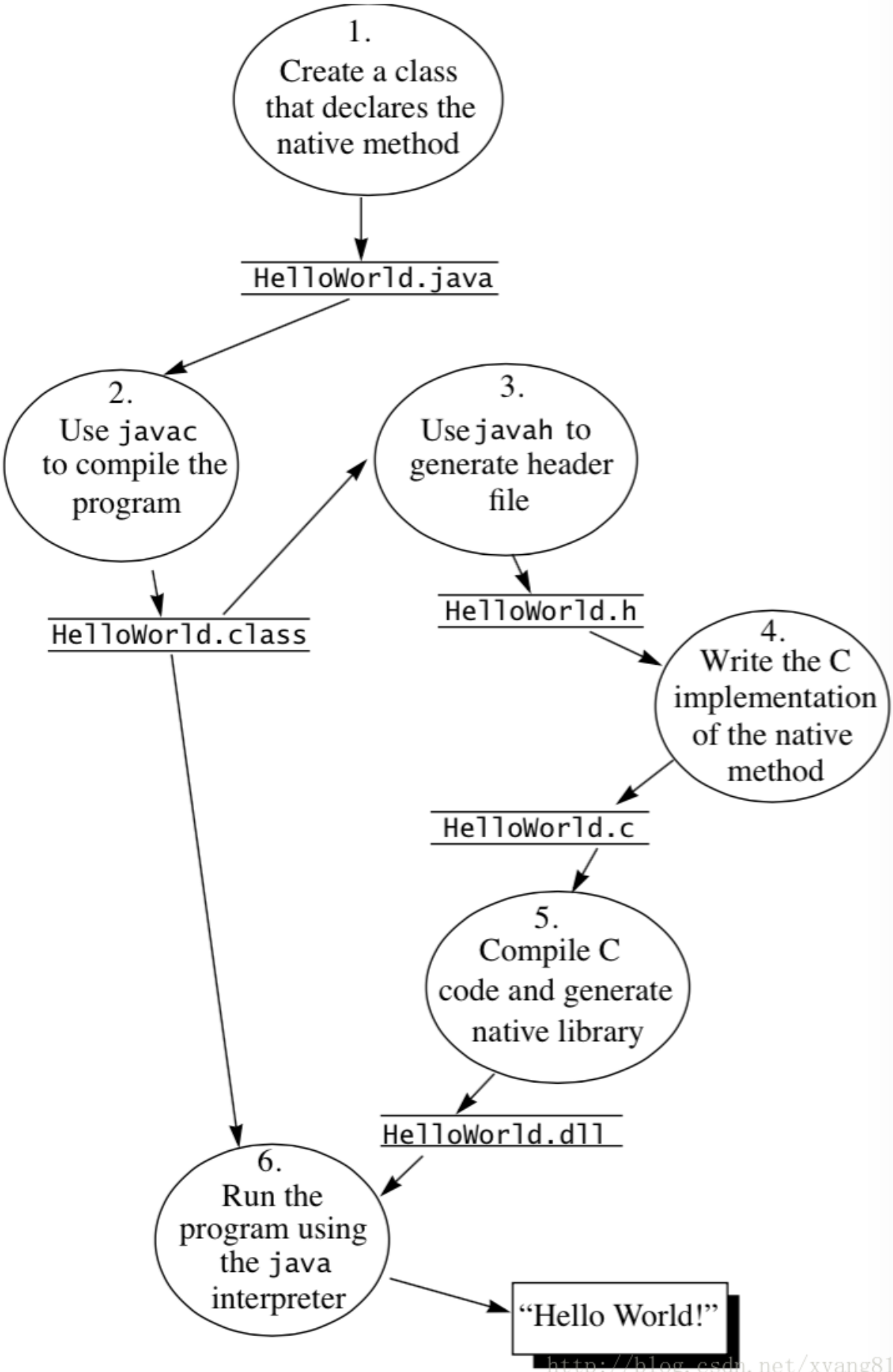
开发流程

JNI 全称是 Java Native Interface（Java 本地接口）单词首字母的缩写，本地接口就是指用 C 和 C++ 开发的接口。由于 JNI 是 JVM 规范中的一部份，因此可以将我们写的 JNI 程序在任何实现了 JNI 规范的 Java 虚拟机中运行。同时，这个特性使我们可以复用以前用 C/C++ 写的大量代码。

开发 JNI 程序会受到系统环境的限制，因为用 C/C++ 语言写出来的代码或模块，编译过程当中要依赖当前操作系统环境所提供的一些库函数，并和本地库链接在一起。而且编译后生成的二进制代码只能在本地操作系统环境下运行，因为不同的操作系统环境，有自己的本地库和 CPU 指令集，而且各个平台对标准 C/C++ 的规范和标准库函数实现方式也有所区别。这就造成使用了 JNI 接口的 JAVA 程序，不再像以前那样自由的跨平台。如果要实现跨平台，就必须将本地代码在不同的操作系统平台下编译出相应的动态库。

JNI 开发流程主要分为以下 6 步：

- 编写声明了 native 方法的 Java 类
- 将 Java 源代码编译成 class 字节码文件
- 用 javah -jni 命令生成 `.h` 头文件（javah 是 jdk 自带的一个命令，-jni 参数表示将 class 中用native 声明的函数生成 JNI 规则的函数）
- 用本地代码实现 `.h头` 文件中的函数
- 将本地代码编译成动态库（Windows：*.dll，linux/unix：*.so，mac os x：*.jnilib）
- 拷贝动态库至 java.library.path 本地库搜索目录下，并运行 Java 程序



通过上面的介绍，相信大家对 JNI 及开发流程有了一个整体的认识，下面通过一个 HelloWorld 的示例，再深入了解 JNI 开发的各个环节及注意事项。

HelloWorld

> 注意：这个案例用命令行的方式介绍开发流程，这样大家对 JNI 开发流程的印象会更加深刻，后面的案例都采用eclipse+cdt 来开发。

第一步，新建一个 HelloWorld.java 源文件

```
public class HelloWorld {

 public class HelloWorld {


  public static native String sayHello(String name); // 1.声明这是一个native函数，由本地代码实现

  public static void main(String[] args) {
    String text = sayHello("yangxin");   // 3.调用本地函数
    System.out.println(text);
  }

  static {
    System.loadLibrary("HelloWorld");   // 2.加载实现了native函数的动态库，只需要写动态库的名字
  }

}
```

第二步，用 javac 命令将 `.java` 源文件编译成 `.class` 字节码文件

> 注意：HelloWorld 放在 com.study.jnilearn 包下面

```
javac src/com/study/jnilearn/HelloWorld.java -d ./bin
```

-d 表示将编译后的 class 文件放到指定的目录下，这里我把它放到和 src 同级的 bin 目录下。

第三步，用 javah -jni 命令，根据 `class` 字节码文件生成 `.h` 头文件（-jni 参数是可选的）

```
javah -jni -classpath ./bin -d ./jni com.study.jnilearn.HelloWorld
```

默认生成的 `.h` 头文件名为：com_study_jnilearn_HelloWorld.h（包名+类名.h），也可以通过 `-o` 参数指定生成头文件名称：

```
javah -jni -classpath ./bin -o HelloWorld.h com.study.jnilearn.HelloWorld
```

参数说明：

- classpath：类搜索路径，这里表示从当前的 bin 目录下查找
- d：将生成的头文件放到当前的 jni 目录下
- o： 指定生成的头文件名称，默认以类全路径名生成（包名+类名.h）

> 注意：`-d`和`-o`只能使用其中一个参数。

第四步，用本地代码实现.h头文件中的函数

- com_study_jnilearn_HelloWorld.h

```
/* DO NOT EDIT THIS FILE - it is machine generated */
 #include <jni.h>
/* Header for class com_study_jnilearn_HelloWorld */

 #ifndef _Included_com_study_jnilearn_HelloWorld
 #define _Included_com_study_jnilearn_HelloWorld
 #ifdef __cplusplus
extern "C" {
 #endif
/*
 * Class:     com_study_jnilearn_HelloWorld
 * Method:    sayHello
 * Signature: (Ljava/lang/String;)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_com_study_jnilearn_HelloWorld_sayHello
  (JNIEnv *, jclass, jstring);

 #ifdef __cplusplus
}
 #endif
 #endif
```

- HelloWorld.c

```
// HelloWorld.c

 #include "com_study_jnilearn_HelloWorld.h"

 #ifdef __cplusplus
extern "C"
{
```

```
 #endif

/*
 * Class:     com_study_jnilearn_HelloWorld
 * Method:    sayHello
 * Signature: (Ljava/lang/String;)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_com_study_jnilearn_HelloWorld_sayHello(
        JNIEnv *env, jclass cls, jstring j_str)
{
    const char *c_str = NULL;
    char buff[128] = { 0 };
    c_str = (*env)->GetStringUTFChars(env, j_str, NULL);
    if (c_str == NULL)
    {
        printf("out of memory.\n");
        return NULL;
    }
    (*env)->ReleaseStringUTFChars(env, j_str, c_str);
    printf("Java Str:%s\n", c_str);
    sprintf(buff, "hello %s", c_str);
    return (*env)->NewStringUTF(env, buff);
}

 #ifdef __cplusplus
}
 #endif
```

第五步，将 C/C++ 代码编译成本地动态库文件动态库文件名命名规则：lib+动态库文件名+后缀（操作系统不一样，后缀名也不一样）如：

- Mac OS X : libHelloWorld.jnilib
- Windows： HelloWorld.dll（不需要 lib 前缀）
- Linux/Unix：libHelloWorld.so

1.Mac OS X

```
gcc -dynamiclib -o /Users/yangxin/Library/Java/Extensions/libHelloWorld.jnilib jni/HelloWorld.c -framework JavaVM -I/$J
```

$JAVA_HOME目录在：/Library/Java/JavaVirtualMachines/jdk1.7.0_21.jdk/Contents/Home (可根据具体情况自己设置)

参数选项说明：

- -dynamiclib：表示编译成动态链接库
- -o：指定动态链接库编译后生成的路径及文件名
- -framework JavaVM -I：编译 JNI 需要用到 JVM 的头文件(jni.h)，第一个目录是平台无关的，第二个目录是与操作系统平台相关的头文件

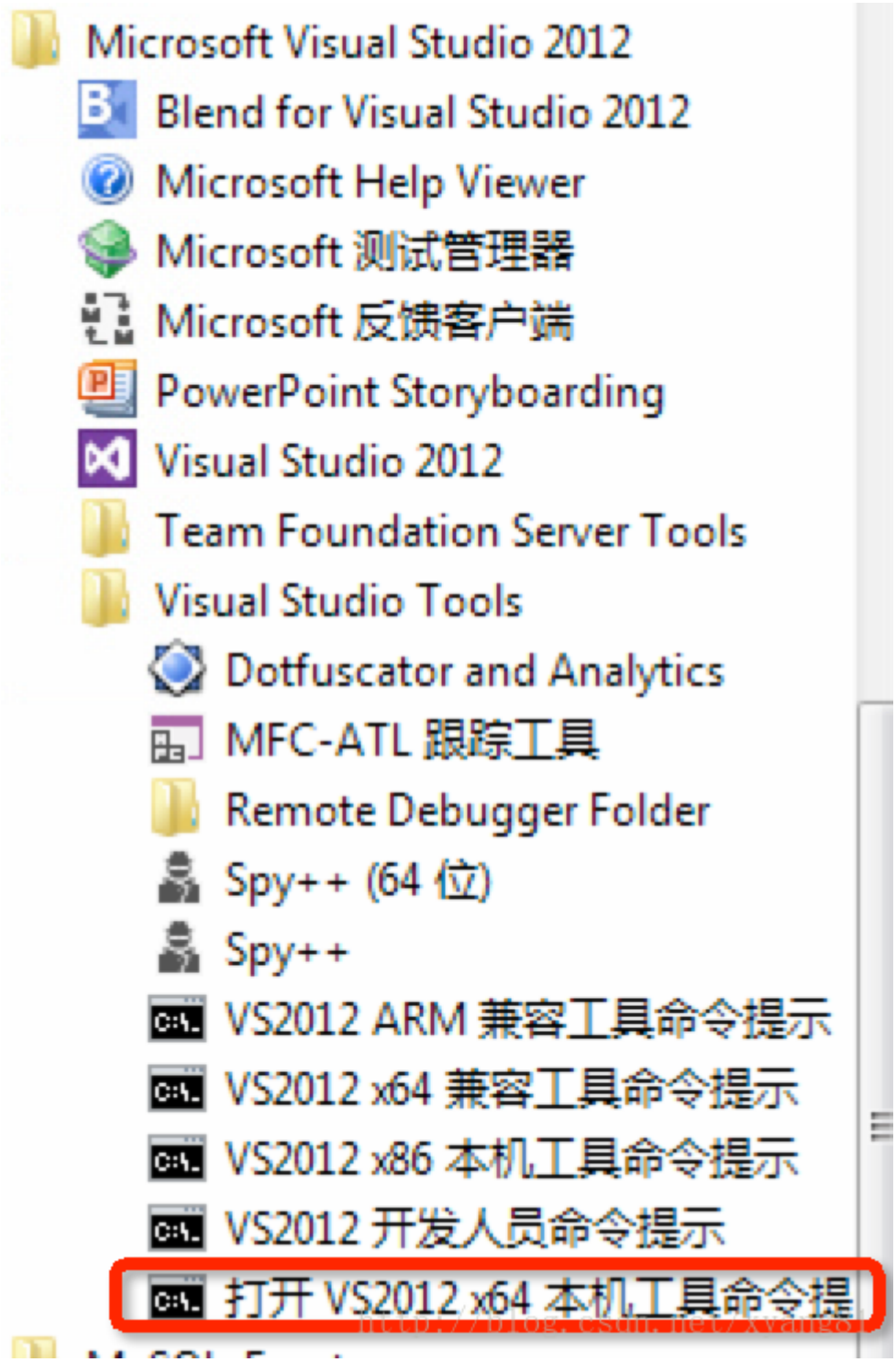
2.Windows (以 Windows7 下 VS2012 为例)

开始菜单-->所有程序-->Microsoft Visual Studio 2012-->打开 VS2012 X64 本机工具命令提示，用 cl 命令编译成 dll 动态库：

```
cl -I"%JAVA_HOME%\include" -I"%JAVA_HOME%\include\win32" -LD HelloWorld.c -FeHelloWorld.dll
```

参数选项说明：

- -I：和 mac os x 一样，包含编译 JNI 必要的头文件
- -LD：标识将指定的文件编译成动态链接库
- -Fe：指定编译后生成的动态链接库的路径及文件名



3.Linux/Unix

```
gcc -I$JAVA_HOME/include -I$JAVA_HOME/include/linux -fPIC -shared HelloWorld.c -o libHelloWorld.so
```

参数说明：

- -I： 包含编译JNI必要的头文件

- -fPIC： 编译成与位置无关的独立代码
- -shared：编译成动态库
- -o： 指定编译后动态库生成的路径和文件名

第六步，运行 Java 程序

Java 在调用 native (本地)方法之前，需要先加载动态库。如果在未加载动态之前就调用 native 方法，会抛出找不到动态链接库文件的异常。如下所示：

```
Exception in thread "main" java.lang.UnsatisfiedLinkError: com.study.jnilearn.HelloWorld.sayHello(Ljava/lang/String;)Ljava
    at com.study.jnilearn.HelloWorld.sayHello(Native Method)
    at com.study.jnilearn.HelloWorld.main(HelloWorld.java:9)
```

一般在类的静态（static）代码块中加载动态库最合适，因为在创建类的实例时，类会被 ClassLoader 先加载到虚拟机，随后立马调用类的 static 静态代码块。这时再去调用 native 方法就万无一失了。加载动态库的两种方式：

```
System.loadLibrary("HelloWorld");
System.load("/Users/yangxin/Desktop/libHelloWorld.jnilib");
```

方式1：只需要指定动态库的名字即可，不需要加 `lib` 前缀，也不要加 `.so` 、 `.dll` 和 `.jnilib` 后缀

方式2：指定动态库的绝对路径名，需要加上前缀和后缀

如果使用方式1，java 会去 java.library.path 系统属性指定的目录下查找动态库文件，如果没有找到会抛出java.lang.UnsatisfiedLinkError 异常。

```
Exception in thread "main" java.lang.UnsatisfiedLinkError: no HelloWorld2 in java.library.path
    at java.lang.ClassLoader.loadLibrary(ClassLoader.java:1860)
    at java.lang.Runtime.loadLibrary0(Runtime.java:845)
    at java.lang.System.loadLibrary(System.java:1084)
    at com.study.jnilearn.HelloWorld.<clinit>(HelloWorld.java:13)
```

大家从异常中可以看出来，他是在 java.library.path 中查找该名称对应的动态库，如果在 Mac 下找libHelloWorld.jnilib 文件，linux 下找 libHelloWorld.so 文件，Windows 下找 libHelloWorld.dll 文件，可以通过调用 System.getProperties("java.library.path")方法获取查找的目录列表，下面是我本机mac os x 系统下的查找目录：

```
String libraryDirs = System.getProperty("java.library.path");
System.out.println(libraryDirs);
// 输出结果如下：
/Users/yangxin/Library/Java/Extensions:/Library/Java/Extensions:/Network/Library/Java/Extensions:/System/Library/Java
```

有两种方式可以让 Java 从 java.library.path 找到动态链接库文件，聪明的你应该已经想到了。

方式1：将动态链接库拷贝到java.library.path目录下

方式2：给 jvm 添加“-Djava.library.path=动态链接库搜索目录”参数，指定系统属性 java.library.path 的值 java -Djava.library.path=/Users/yangxin/Desktop Linux/Unix 环境下可以通过设置 LD_LIBRARY_PATH 环境变量，指定库的搜索目录。

运行写好的 Java 程序了，结果如下：

```
yangxin-MacBook-Pro:JNILearn yangxin$ java -classpath ./bin com.study.jnilearn.HelloWorld
Java Str:yangxin
hello yangxin
```

如果没有将动态库拷贝到本地库搜索目录下，执行 `java` 命令，可通过添加系统属性 java.library.path 来指定动态库的目录，如下所示：

```
yangxin-MacBook-Pro:JNILearn yangxin$ java -Djava.library.path=/Users/yangxin/Desktop -classpath ./bin com.study.
Java Str:yangxin
hello yangxin
```

3

JVM 查找 native 方法的规则

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.UnsatisfiedLinkError 异常，找不到 XX 方法的提示。现在我们想想，在 Java 中调用某个 native 方法时，JVM 是通过什么方式，能正确的找到动态库中 C/C++ 实现的那个 native 函数呢?

JVM 查找 native 方法

JVM 查找 native 方法有两种方式：

- 按照 JNI 规范的命名规则
- 调用 JNI 提供的 RegisterNatives 函数，将本地函数注册到 JVM 中。（后面会详细介绍）

本文通过第一篇文章 HelloWorld 示例中的 Java_com_study_jnilearn_HelloWorld_sayHello 函数来详细介绍第一种方式：

```
/* DO NOT EDIT THIS FILE - it is machine generated */
 #include <jni.h>
/* Header for class com_study_jnilearn_HelloWorld */

 #ifndef _Included_com_study_jnilearn_HelloWorld
 #define _Included_com_study_jnilearn_HelloWorld
 #ifdef __cplusplus
extern "C" {
 #endif
/*
 * Class:     com_study_jnilearn_HelloWorld
 * Method:    sayHello
 * Signature: (Ljava/lang/String;)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_com_study_jnilearn_HelloWorld_sayHello
  (JNIEnv *, jclass, jstring);

 #ifdef __cplusplus
}
 #endif
 #endif
```

JNIEXPORT 和 JNICALL 的作用

在上篇文章中，我们在将 HelloWorld.c 编译成动态库的时候，用 -I 参数包含了 JDK 安装目录下的两个头文件目录：

```
gcc -dynamiclib -o /Users/yangxin/Library/Java/Extensions/libHelloWorld.jnilib jni/HelloWorld.c -framework JavaVM -I/$
```

其中第一个目录为 jni.h 头文件所在目录，第二个是跨平台头文件目录（Mac os x系统下的目录名为 darwin，在 Windows 下目录名为 win32，linux 下目录名为 linux），用于定义与平台相关的宏，其中用于标识函数用途的两个宏 JNIEXPORT 和 JNICALL，就定义在 darwin 目录下的 jni_md.h 头文件中。在 Windows 中编译 dll 动态库规定，如果动态库中的函数要被外部调用，需要在函数声明中添加 __declspec (dllexport)标识，表示将该函数导出在外部可以调用。在 Linux/Unix 系统中，这两个宏可以省略不加。这两个平台的区别是由于各自的编译器所产生的可执行文件格式不一样。这里有篇文章详细介绍了两个平台编译的动态库区别：http://www.cnblogs.com/chio/archive/2008/11/13/1333119.html。JNICALL 在 Windows 中的值为 __stdcall，用于约束函数入栈顺序和堆栈清理的规则。

Windows 下 jni_md.h 头文件内容：

```
#ifndef _JAVASOFT_JNI_MD_H_
#define _JAVASOFT_JNI_MD_H_

#define JNIEXPORT __declspec(dllexport)
#define JNIIMPORT __declspec(dllimport)
#define JNICALL __stdcall

typedef long jint;
typedef __int64 jlong;
typedef signed char jbyte;

#endif
```

Linux 下 jni_md.h 头文件内容：

```
#ifndef _JAVASOFT_JNI_MD_H_
#define _JAVASOFT_JNI_MD_H_

#define JNIEXPORT
#define JNIIMPORT
#define JNICALL

typedef int jint;
#ifdef _LP64 /* 64-bit Solaris */
```

```
typedef long jlong;
 #else
typedef long long jlong;
 #endif

typedef signed char jbyte;

 #endif
```

从 Linux 下的 jni_md.h 头文件可以看出来，JNIEXPORT 和 JNICALL 是一个空定义，所以在 Linux 下 JNI 函数声明可以省略这两个宏。

函数的命名规则：

用 javah 工具生成函数原型的头文件，函数命名规则为： Java_ 类全路径_方法名。如 Java_com_study_jnilearn_HelloWorld_sayHello ，其中 Java_ 是函数的前缀， com_study_jnilearn_HelloWorld 是类名， sayHello 是方法名，它们之间用 _ (下划线) 连接。

函数参数：

```
JNIEXPORT jstring JNICALL Java_com_study_jnilearn_HelloWorld_sayHello(JNIEnv *, jclass, jstring);
```

- 第一个参数： JNIEnv* 是定义任意 native 函数的第一个参数（包括调用 JNI 的 RegisterNatives 函数注册的函数），指向 JVM 函数表的指针，函数表中的每一个入口指向一个 JNI 函数，每个函数用于访问 JVM 中特定的数据结构。
- 第二个参数：调用 Java 中 native 方法的实例或 Class 对象，如果这个 native 方法是实例方法，则该参数是 jobject，如果是静态方法，则是 jclass。
- 第三个参数：Java 对应 JNI 中的数据类型，Java 中 String 类型对应 JNI 的 jstring 类型。（后面会详细介绍 JAVA 与 JNI 数据类型的映射关系）。

函数返回值类型：夹在 JNIEXPORT 和 JNICALL 宏中间的 jstring，表示函数的返回值类型，对应 Java 的String 类型。

总结：当我们熟悉了 JNI 的 native 函数命名规则之后，就可以不用通过 javah 命令去生成相应 java native方法的函数原型了，只需要按照函数命名规则编写相应的函数原型和实现即可。

比如 com.study.jni.Utils 类中还有一个计算加法的 native 实例方法 add ，有两个 int参数和一个 int 返回值: public native int add(int num1, int num2) ，对应 JNI 的函数原型就是： JNIEXPORT jint JNICALL Java_com_study_jni_Utils_add(JNIEnv *, jobject, jint,jint) 。

4

JNI 数据类型与 Java 数据类型的映射关系

当我们在调用一个 Java native 方法的时候，方法中的参数是如何传递给 C/C++ 本地函数中的呢？Java 方法中的参数与 C/C++ 函数中的参数，它们之间是怎么转换的呢？我猜你应该也有相关的疑虑吧，咱们先来看一个例子，还是以 HelloWorld 为例：

HelloWorld.java：

```
package com.study.jnilearn;

class MyClass {}

public class HelloWorld {

   public static native void test(short s, int i, long l, float f, double d, char c,
         boolean z, byte b, String str, Object obj, MyClass p, int[] arr);

   public static void main(String[] args) {
      String obj = "obj";
      short s = 1;
      long l = 20;
      byte b = 127;
      test(s, 1, l, 1.0f, 10.5, 'A', true, b, "中国", obj, new MyClass(), new int[] {});
   }

   static {
      System.loadLibrary("HelloWorld");
   }
}
```

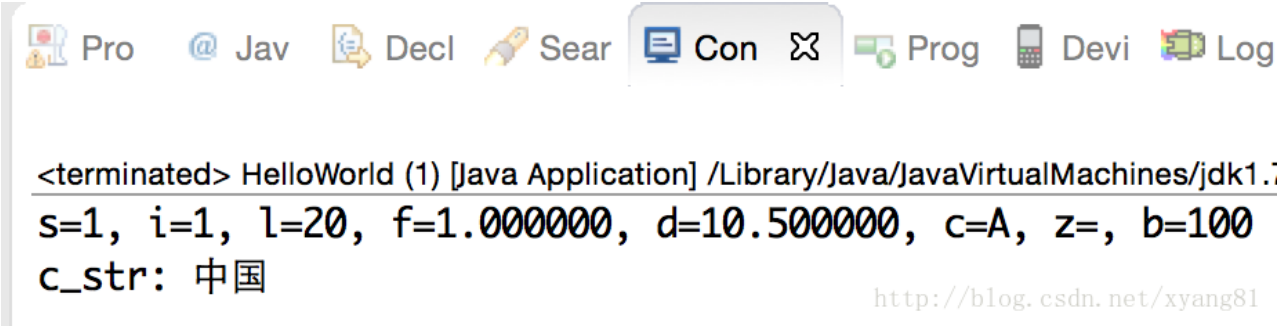
在 HelloWorld.java 中定义了一个 test 的 native 方法，该方法中一个共有 12 个参数，其中前面 8 个为基本数据类型，后面 4 个全部为引用类型。

由 HelloWorld.class 生成的 native 函数原型及实现：

```
/*
 * Class:     com_study_jnilearn_HelloWorld
 * Method:    test
 * Signature: (SIJFDCZBLjava/lang/String;Ljava/lang/Object;Lcom/study/jnilearn/MyClass;[I)V
 */
JNIEXPORT void JNICALL Java_com_study_jnilearn_HelloWorld_test
   (JNIEnv *env, jclass cls, jshort s, jint i, jlong l, jfloat f,
    jdouble d, jchar c, jboolean z, jbyte b, jstring j_str, jobject jobj1, jobject job2, jintArray j_int_arr)
{
   printf("s=%hd, i=%d, l=%ld, f=%f, d=%lf, c=%c, z=%c, b=%d", s, i, l, f, d, c, z, b);
   const char *c_str = NULL;
   c_str = (*env)->GetStringUTFChars(env, j_str, NULL);
```

```
    if (c_str == NULL)
    {
        return; // memory out
    }
    (*env)->ReleaseStringUTFChars(env, j_str, c_str);
    printf("c_str: %s\n", (char*)c_str);
}
```

调用 test 方法的输出结果：



从头文件函数的原型可以得知，test 方法中形参的数据类型自动转换成了 JNI 中相应的数据类型，不难理解，在调用 Java native 方法将实参传递给 C/C++ 函数的时候，会自动将 java 形参的数据类型自动转换成 C/C++ 相应的数据类型，所以我们在写 JNI 程序的时候，必须要明白它们之间数据类型的对应关系。

在 Java 语言中数据类型分为基本数据类型和引用类型，其中基本数据类型有 8 种：byte、char、short、int、long、float、double、boolean，除了基本数据类型外其它都是引用类型：Object、String、数组等。8 种基本数据类型分别对应JNI数据类型中的 jbyte、jchar、jshort、jint、jlong、jfloat、jdouble、jboolean。所有的 JNI 引用类型全部是 jobject 类型，为了使用方便和类型安全，JNI 定义了一个引用类型集合，集合当中的所有类型都是 jobject 的子类，这些子类和 Java 中常用的引用类型相对应。例如：jstring 表示字符串、jclass 表示 class 字节码对象、jthrowable 表示异常、jarray 表示数组，另外 jarray 派生了 8 个子类，分别对应Java 中的 8 种基本数据类型（jintArray、jshortArray、jlongArray等）。下面再回顾头来看看 test 方法与 `Java_com_study_jnilearn_HelloWorld_test` 函数中参数类型的对应关系：

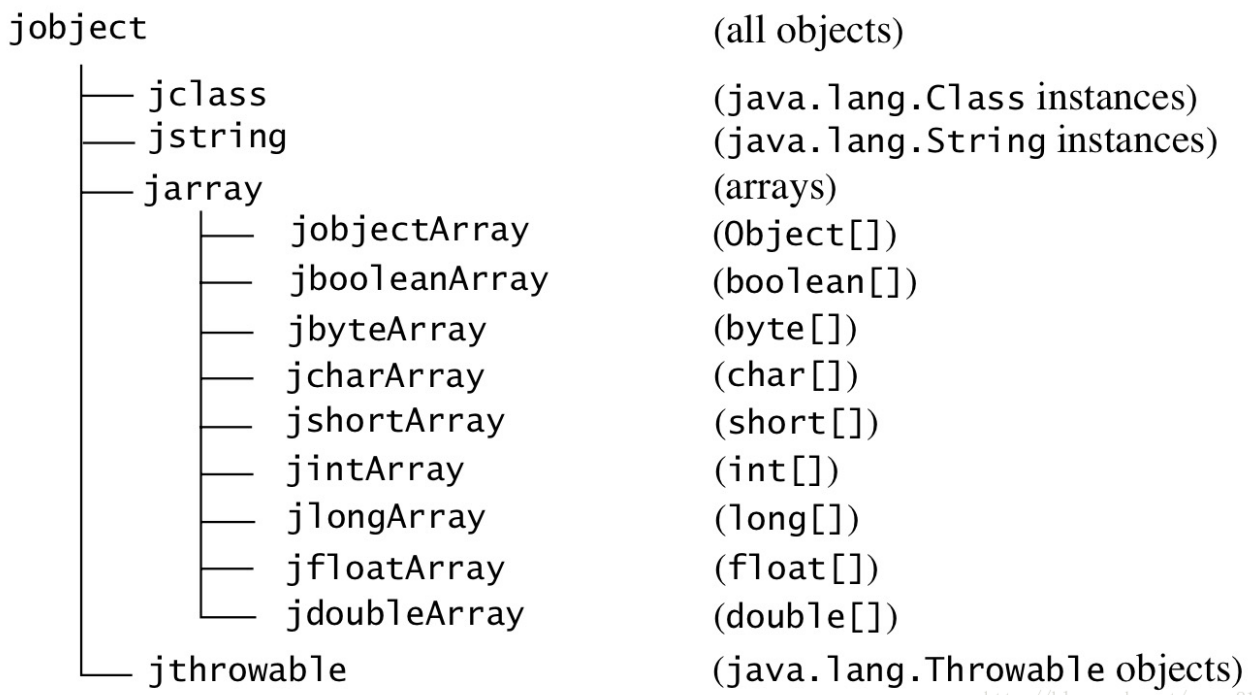
```
// HelloWorld.java
public static native void test(short s, int i, long l, float f, double d, char c,
        boolean z, byte b, String str, Object obj, MyClass p);

// HelloWorld.h
JNIEXPORT void JNICALL Java_com_study_jnilearn_HelloWorld_test
  (JNIEnv *, jclass, jshort, jint, jlong, jfloat, jdouble, jchar, jboolean, jbyte, jstring, jobject, jobject, jintArray);
```

从上面两个函数的参数中可以看出来，除了 JNIEnv 和 jclass 这两个参数外，其它参数都是一一对应的。下面是 JNI 规范文档中描述 Java 与 JNI数据类型的对应关系：

基本数据类型：

Java Language Type	Native Type	Description
boolean	jboolean	unsigned 8 bits
byte	jbyte	signed 8 bits
char	jchar	unsigned 16 bits
short	jshort	signed 16 bits
int	jint	signed 32 bits
long	jlong	signed 64 bits
float	jfloat	32 bits
double	jdouble	64 bits

引用类型：



JNI 如果使用 C++ 语言编写的话，所有引用类型派生自 jobject，使用 C++ 的继承结构特性，使用相应的类型。如下所示：

```
class _jobject {};
  class _jclass : public _jobject {};
  class _jstring : public _jobject {};
  class _jarray : public _jobject {};
  class _jbooleanArray : public _jarray {};
  class _jbyteArray : public _jarray {};
```

JNI 如果使用 C 语言编写的话，所有引用类型使用 jobject，其它引用类型使用 typedef 重新定义，如：typedef jobject jstring 。 jvalue 是一个 unio（联合）类型，在 C 语中为了节约内存，会用联合类型变量来存储声明在联合体中的任意类型数据 。在 JNI 中将基本数据类型与引用类型定义在一个联合类型中，表示用 jvalue 定义的变量，可以存储任意 JNI 类型的数据，后面会介绍 jvalue 在 JNI 编程当中的应用。原型如下：

```
typedef union jvalue {
   jboolean z;
   jbyte    b;
   jchar    c;
   jshort   s;
   jint     i;
   jlong    j;
   jfloat   f;
   jdouble  d;
   jobject  l;
} jvalue;
```

5

JNI 字符串处理

处理字符串

从第三章中可以看出 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 unsigned short  jchar;
typedef short           jshort;
typedef float           jfloat;
typedef double          jdouble;
typedef int jint;
 #ifdef _LP64 /* 64-bit Solaris */
typedef long jlong;
 #else
typedef long long jlong;
 #endif

typedef signed char jbyte;
```

基本类型很容易理解，就是对 C/C++ 中的基本类型用 typedef 重新定义了一个新的名字，在 JNI 中可以直接访问。JNI 把 Java 中的所有对象当作一个C指针传递到本地方法中，这个指针指向 JVM 中的内部数据结构，而内部的数据结构在内存中的存储方式是不可见的。只能从 JNIEnv 指针指向的函数表中选择合适的 JNI 函数来操作 JVM 中的数据结构。第三章的示例中，访问 java.lang.String 对应的 JNI 类型 jstring 时，没有像访问基本数据类型一样直接使用，因为它在 Java 是一个引用类型，所以在本地代码中只能通过 GetStringUTFChars 这样的 JNI 函数来访问字符串的内容。

下面先看一个例子：

Sample.java：

```
package com.study.jnilearn;

public class Sample {

    public native static String sayHello(String text);

    public static void main(String[] args) {
        String text = sayHello("yangxin");
        System.out.println("Java str: " + text);
    }

    static {
```

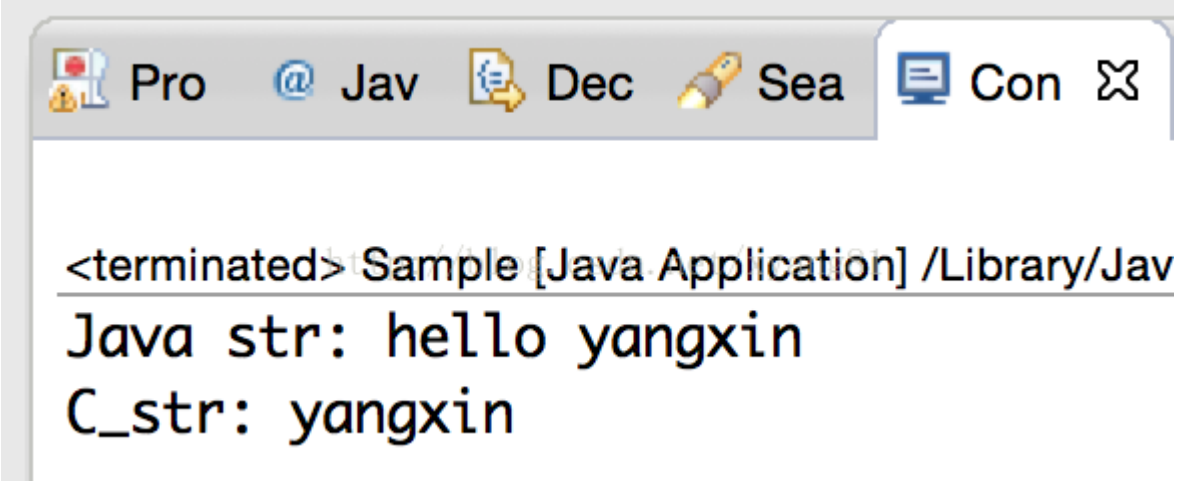
```
        System.loadLibrary("Sample");
    }
}
```

com_study_jnilearn_Sample.h和Sample.c：

```
/* DO NOT EDIT THIS FILE - it is machine generated */
 #include <jni.h>
/* Header for class com_study_jnilearn_Sample */

 #ifndef _Included_com_study_jnilearn_Sample
 #define _Included_com_study_jnilearn_Sample
 #ifdef __cplusplus
extern "C" {
 #endif
/*
 * Class:     com_study_jnilearn_Sample
 * Method:    sayHello
 * Signature: (Ljava/lang/String;)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_com_study_jnilearn_Sample_sayHello
  (JNIEnv *, jclass, jstring);

 #ifdef __cplusplus
}
 #endif
 #endif

// Sample.c
 #include "com_study_jnilearn_Sample.h"
/*
 * Class:     com_study_jnilearn_Sample
 * Method:    sayHello
 * Signature: (Ljava/lang/String;)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_com_study_jnilearn_Sample_sayHello
  (JNIEnv *env, jclass cls, jstring j_str)
{
    const char *c_str = NULL;
    char buff[128] = {0};
    jboolean isCopy;    // 返回JNI_TRUE表示原字符串的拷贝，返回JNI_FALSE表示返回原字符串的指针
    c_str = (*env)->GetStringUTFChars(env, j_str, &isCopy);
    printf("isCopy:%d\n",isCopy);
    if(c_str == NULL)
    {
        return NULL;
```

```
    }
    printf("C_str: %s \n", c_str);
    sprintf(buff, "hello %s", c_str);
    (*env)->ReleaseStringUTFChars(env, j_str, c_str);
    return (*env)->NewStringUTF(env,buff);
}
```

运行结果如下：



示例解析

访问字符串

sayHello 函数接收一个 jstring 类型的参数 text，但 jstring 类型是指向 JVM 内部的一个字符串，和 C 风格的字符串类型 char* 不同，所以在 JNI 中不能通把 jstring 当作普通 C 字符串一样来使用，必须使用合适的 JNI 函数来访问 JVM 内部的字符串数据结构。

GetStringUTFChars(env, j_str, &isCopy) 参数说明：

- env：JNIEnv 函数表指针
- j_str：jstring 类型（Java 传递给本地代码的字符串指针）
- 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。

因为 Java 默认使用 Unicode 编码，而 C/C++ 默认使用 UTF 编码，所以在本地代码中操作字符串的时候，必须使用合适的 JNI 函数把 jstring 转换成 C 风格的字符串。JNI 支持字符串在 Unicode 和 UTF-8 两种编码之间

转换，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。在上例中 sayHello 函数中我们通过 GetStringUTFChars 正确取得了 JVM 内部的字符串内容。

异常检查

调用完 GetStringUTFChars 之后不要忘记安全检查，因为 JVM 需要为新诞生的字符串分配内存空间，当内存空间不够分配的时候，会导致调用失败，失败后 GetStringUTFChars 会返回 NULL，并抛出一个OutOfMemoryError 异常。JNI 的异常和 Java 中的异常处理流程是不一样的，Java 遇到异常如果没有捕获，程序会立即停止运行。而 JNI 遇到未决的异常不会改变程序的运行流程，也就是程序会继续往下走，这样后面针对这个字符串的所有操作都是非常危险的，因此，我们需要用 return 语句跳过后面的代码，并立即结束当前方法。

释放字符串

在调用 GetStringUTFChars 函数从 JVM 内部获取一个字符串之后，JVM 内部会分配一块新的内存，用于存储源字符串的拷贝，以便本地代码访问和修改。即然有内存分配，用完之后马上释放是一个编程的好习惯。通过调用ReleaseStringUTFChars 函数通知 JVM 这块内存已经不使用了，你可以清除了。注意：这两个函数是配对使用的，用了 GetXXX 就必须调用 ReleaseXXX，而且这两个函数的命名也有规律，除了前面的 Get 和 Release 之外，后面的都一样。

创建字符串

通过调用 NewStringUTF 函数，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。如果 JVM 不能为构造 java.lang.String 分配足够的内存，NewStringUTF 会抛出一个 OutOfMemoryError 异常，并返回 NULL。在这个例子中我们不必检查它的返回值，如果NewStringUTF 创建 java.lang.String 失败，OutOfMemoryError 这个异常会被在 Sample.main 方法中抛出。如果 NewStringUTF 创建 java.lang.String 成功，则返回一个 JNI 引用，这个引用指向新创建的java.lang.String 对象。

其它字符串处理函数

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，也可以通过标准 C 函数 strlen 获取。

GetStringCritical和ReleaseStringCritical

提高 JVM 返回源字符串直接指针的可能性。

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：

```
JNIEXPORT jstring JNICALL Java_com_study_jnilearn_Sample_sayHello
  (JNIEnv *env, jclass cls, jstring j_str)
{
    const jchar* c_str= NULL;
    char buff[128] = "hello ";
    char* pBuff = buff + 6;
    /*
     * 在GetStringCritical/RealeaseStringCritical之间是一个关键区。
     * 在这关键区之中,绝对不能呼叫JNI的其他函数和会造成当前线程中断或是会让当前线程等待的任何本地代码，
     * 否则将造成关键区代码执行区间垃圾回收器停止运作，任何触发垃圾回收器的线程也会暂停。
     * 其他触发垃圾回收器的线程不能前进直到当前线程结束而激活垃圾回收器。
     */
    c_str = (*env)->GetStringCritical(env,j_str,NULL);  // 返回源字符串指针的可能性
    if (c_str == NULL)  // 验证是否因为字符串拷贝内存溢出而返回NULL
    {
        return NULL;
    }
    while(*c_str)
    {
        *pBuff++ = *c_str++;
    }
```

```
    (*env)->ReleaseStringCritical(env,j_str,c_str);
    return (*env)->NewStringUTF(env,buff);
}
```

JNI 中没有 Get/ReleaseStringUTFCritical 这样的函数，因为在进行编码转换时很可能会促使 JVM 对数据进行复制，因为 JVM 内部表示的字符串是使用 Unicode 编码的。

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：

```
JNIEXPORT jstring JNICALL Java_com_study_jnilearn_Sample_sayHello
  (JNIEnv *env, jclass cls, jstring j_str)
{
    jsize len = (*env)->GetStringLength(env,j_str);  // 获取unicode字符串的长度
    printf("str_len:%d\n",len);
    char buff[128] = "hello ";
    char* pBuff = buff + 6;
    // 将JVM中的字符串以utf-8编码拷入C缓冲区,该函数内部不会分配内存空间
    (*env)->GetStringUTFRegion(env,j_str,0,len,pBuff);
    return (*env)->NewStringUTF(env,buff);
}
```

GetStringUTFRegion 这个函数会做越界检查，如果检查发现越界了，会抛出StringIndexOutOfBoundsException 异常，这个方法与 GetStringUTFChars 比较相似，不同的是，GetStringUTFRegion 内部不分配内存，不会抛出内存溢出异常。

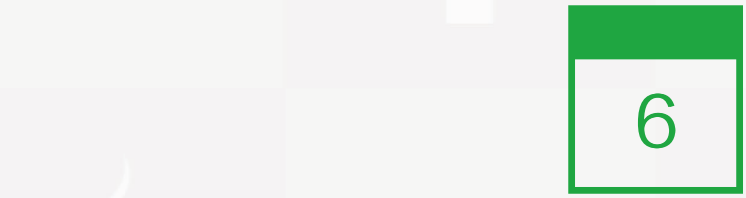
> 注意：GetStringUTFRegion 和 GetStringRegion 这两个函数由于内部没有分配内存，所以 JNI 没有提供ReleaseStringUTFRegion 和 ReleaseStringRegion 这样的函数。

字符串操作总结

总结：

- 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译器提前分配，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。
- 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。

- 获取 Unicode 字符串和长度，使用 GetStringChars 和 GetStringLength 函数。
- 获取 UTF-8 字符串的长度，使用 GetStringUTFLength 函数。
- 创建 Unicode 字符串，使用 NewStringUTF 函数。
- 从 Java 字符串转换成 C/C++ 字符串，使用 GetStringUTFChars 函数。
- 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。



6

JNI 访问数组

JNI 中的数组分为基本类型数组和对象数组，它们的处理方式是不一样的，基本类型数组中的所有元素都是 JNI 的基本数据类型，可以直接访问。而对象数组中的所有元素是一个类的实例或其它数组的引用，和字符串操作一样，不能直接访问 Java 传递给 JNI 层的数组，必须选择合适的 JNI 函数来访问和设置 Java 层的数组对象。阅读此文假设你已经了解了 JNI 与 Java 数据类型的映射关系，如果还不了解的，请移步《 JNI——JNI 数据类型与 Java 数据类型的映射关系 》阅读。下面以 int 类型为例说明基本数据类型数组的访问方式，对象数组类型用一个创建二维数组的例子来演示如何访问。

访问基本类型数组

```
package com.study.jnilearn;

// 访问基本类型数组
public class IntArray {

    // 在本地代码中求数组中所有元素的和
    private native int sumArray(int[] arr);

    public static void main(String[] args) {
        IntArray p = new IntArray();
        int[] arr = new int[10];
        for (int i = 0; i < arr.length; i++) {
            arr[i] = i;
        }
        int sum = p.sumArray(arr);
        System.out.println("sum = " + sum);
    }

    static {
        System.loadLibrary("IntArray");
    }
}
```

本地代码：

```
/* DO NOT EDIT THIS FILE - it is machine generated */
 #include <jni.h>
/* Header for class com_study_jnilearn_IntArray */

 #ifndef _Included_com_study_jnilearn_IntArray
 #define _Included_com_study_jnilearn_IntArray
 #ifdef __cplusplus
extern "C" {
 #endif
```

```
/*
 * Class:     com_study_jnilearn_IntArray
 * Method:    sumArray
 * Signature: ([I)I
 */
JNIEXPORT jint JNICALL Java_com_study_jnilearn_IntArray_sumArray
  (JNIEnv *, jobject, jintArray);

 #ifdef __cplusplus
}
 #endif
 #endif

// IntArray.c
 #include "com_study_jnilearn_IntArray.h"
 #include <string.h>
 #include <stdlib.h>

/*
 * Class:     com_study_jnilearn_IntArray
 * Method:    sumArray
 * Signature: ([I)I
 */
JNIEXPORT jint JNICALL Java_com_study_jnilearn_IntArray_sumArray
(JNIEnv *env, jobject obj, jintArray j_array)
{
    jint i, sum = 0;
    jint *c_array;
    jint arr_len;
    //1. 获取数组长度
    arr_len = (*env)->GetArrayLength(env,j_array);
    //2. 根据数组长度和数组元素的数据类型申请存放java数组元素的缓冲区
    c_array = (jint*)malloc(sizeof(jint) * arr_len);
    //3. 初始化缓冲区
    memset(c_array,0,sizeof(jint)*arr_len);
    printf("arr_len = %d ", arr_len);
    //4. 拷贝Java数组中的所有元素到缓冲区中
    (*env)->GetIntArrayRegion(env,j_array,0,arr_len,c_array);
    for (i = 0; i < arr_len; i++) {
        sum += c_array[i];  //5. 累加数组元素的和
    }
    free(c_array);  //6. 释放存储数组元素的缓冲区
    return sum;
}
```

上例中，在 Java 中定义了一个 sumArray 的 native 方法，参数类型是 int[]，对应 JNI 中 jintArray 类型。在本地代码中，首先通过 JNI的GetArrayLength 函数获取数组的长度，已知数组是 jintArray 类型，可以得出数组的元素类型是 jint，然后根据数组的长度和数组元素类型，申请相应大小的缓冲区。如果缓冲区不大的话，当然也可以直接在栈上申请内存，那样效率更高，但是没那么灵活，因为 Java 数组的大小变了，本地代码也跟着修改。接着调用 GetIntArrayRegion 函数将 Java 数组中的所有元素拷贝到 C 缓冲区中，并累加数组中所有元素的和，最后释放存储 java 数组元素的 C 缓冲区，并返回计算结果。GetIntArrayRegion 函数第 1 个参数是 JNI Env 函数指针，第 2 个参数是 Java 数组对象，第 3 个参数是拷贝数组的开始索引，第 4 个参数是拷贝数组的长度，第 5 个参数是拷贝目的地。下图是计算结果：

```
sun=45

arr_len=10
```

在前面的例子当中，我们通过调用 GetIntArrayRegion 函数，将 int 数组中的所有元素拷贝到 C 临时缓冲区中，然后在本地代码中访问缓冲区中的元素来实现求和的计算，JNI 还提供了一个和 GetIntArrayRegion 相对应的函 SetIntArrayRegion，本地代码可以通过这个函数来修改所有基本数据类型数组的元素。另外 JNI 还提供一系列直接获取数组元素指针的函数 Get/ReleaseArrayElements，比如：GetIntArrayElements、ReleaseArrayElements、GetFloatArrayElements、ReleaseFloatArrayElements 等。下面我们用这种方式重新实现计算数组元素的和：

```
JNIEXPORT jint JNICALL Java_com_study_jnilearn_IntArray_sumArray2
(JNIEnv *env, jobject obj, jintArray j_array)
{
    jint i, sum = 0;
    jint *c_array;
    jint arr_len;
    // 可能数组中的元素在内存中是不连续的，JVM可能会复制所有原始数据到缓冲区，然后返回这个缓冲区的指针
    c_array = (*env)->GetIntArrayElements(env,j_array,NULL);
    if (c_array == NULL) {
        return 0;  // JVM复制原始数据到缓冲区失败
    }
    arr_len = (*env)->GetArrayLength(env,j_array);
    printf("arr_len = %d\n", arr_len);
    for (i = 0; i < arr_len; i++) {
        sum += c_array[i];
    }
    (*env)->ReleaseIntArrayElements(env,j_array, c_array, 0); // 释放可能复制的缓冲区
    return sum;
}
```

GetIntArrayElements 第三个参数表示返回的数组指针是原始数组，还是拷贝原始数据到临时缓冲区的指针，如果是 JNI_TRUE：表示临时缓冲区数组指针，JNI_FALSE：表示临时原始数组指针。开发当中，我们并不关心

它从哪里返回的数组指针，这个参数填 NULL 即可，但在获取到的指针必须做校验，因为当原始数据在内存当中不是连续存放的情况下，JVM 会复制所有原始数据到一个临时缓冲区，并返回这个临时缓冲区的指针。有可能在申请开辟临时缓冲区内存空间时，会内存不足导致申请失败，这时会返回 NULL。

写过 Java 的程序员都知道，在 Java 中创建的对象全都由 GC（垃圾回收器）自动回收，不需要像 C/C++ 一样需要程序员自己管理内存。GC 会实时扫描所有创建的对象是否还有引用，如果没有引用则会立即清理掉。当我们创建一个像 int 数组对象的时候，当我们在本地代码想去访问时，发现这个对象正被 GC 线程占用了，这时本地代码会一直处于阻塞状态，直到等待 GC 释放这个对象的锁之后才能继续访问。为了避免这种现象的发生，JNI 提供了 Get/ReleasePrimitiveArrayCritical 这对函数，本地代码在访问数组对象时会暂停 GC 线程。不过使用这对函数也有个限制，在 Get/ReleasePrimitiveArrayCritical 这两个函数期间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，和处理字符串的 Get/ReleaseStringCritical 函数限制一样。这对函数和 GetIntArrayElements 函数一样，返回的是数组元素的指针。下面用这种方式重新实现上例中的功能：

```
JNIEXPORT jint JNICALL Java_com_study_jnilearn_IntArray_sumArray
(JNIEnv *env, jobject obj, jintArray j_array)
{
    jint i, sum = 0;
    jint *c_array;
    jint arr_len;
    jboolean isCopy;
    c_array = (*env)->GetPrimitiveArrayCritical(env,j_array,&isCopy);
    printf("isCopy: %d \n", isCopy);
    if (c_array == NULL) {
        return 0;
    }
    arr_len = (*env)->GetArrayLength(env,j_array);
    printf("arr_len = %d\n", arr_len);
    for (i = 0; i < arr_len; i++) {
        sum += c_array[i];
    }
    (*env)->ReleasePrimitiveArrayCritical(env, j_array, c_array, 0);
    return sum;
}
```

小结

- 对于小量的、固定大小的数组，应该选择 Get/SetArrayRegion 函数来操作数组元素是效率最高的。因为这对函数要求提前分配一个 C 临时缓冲区来存储数组元素，你可以直接在 Stack（栈）上或用 malloc 在堆上来动态申请，当然在栈上申请是最快的。有童鞋可能会认为，访问数组元素还需要将原始数据全部拷贝一份到临时缓冲区才能访问而觉得效率低？我想告诉你的是，像这种复制少量数组元素的代价是很小的，几乎可

以忽略。这对函数的另外一个优点就是，允许你传入一个开始索引和长度来实现对子数组元素的访问和操作（SetArrayRegion函数可以修改数组），不过传入的索引和长度不要越界，函数会进行检查，如果越界了会抛出 ArrayIndexOutOfBoundsException 异常。

- 如果不想预先分配 C 缓冲区，并且原始数组长度也不确定，而本地代码又不想在获取数组元素指针时被阻塞的话，使用 Get/ReleasePrimitiveArrayCritical 函数对，就像 Get/ReleaseStringCritical 函数对一样，使用这对函数要非常小心，以免死锁。
- Get/ReleaseArrayElements 系列函数永远是安全的，JVM 会选择性的返回一个指针，这个指针可能指向原始数据，也可能指向原始数据的复制。

访问对象数组

JNI 提供了两个函数来访问对象数组，GetObjectArrayElement 返回数组中指定位置的元素，SetObjectArrayElement 修改数组中指定位置的元素。与基本类型不同的是，我们不能一次得到数据中的所有对象元素或者一次复制多个对象元素到缓冲区。因为字符串和数组都是引用类型，只能通过 Get/SetObjectArrayElement 这样的 JNI 函数来访问字符串数组或者数组中的数组元素。下面的例子通过调用一个本地方法来创建一个二维的 int 数组，然后打印这个二维数组的内容：

```
package com.study.jnilearn;

public class ObjectArray {

    private native int[][] initInt2DArray(int size);

    public static void main(String[] args) {
        ObjectArray obj = new ObjectArray();
        int[][] arr = obj.initInt2DArray(3);
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                System.out.format("arr[%d][%d] = %d\n", i, j, arr[i][j]);
            }
        }
    }

    static {
        System.loadLibrary("ObjectArray");
    }
}
```

本地代码：

```
/* DO NOT EDIT THIS FILE - it is machine generated */
 #include <jni.h>
/* Header for class com_study_jnilearn_ObjectArray */

 #ifndef _Included_com_study_jnilearn_ObjectArray
 #define _Included_com_study_jnilearn_ObjectArray
 #ifdef __cplusplus
extern "C" {
 #endif
/*
 * Class:     com_study_jnilearn_ObjectArray
 * Method:    initInt2DArray
 * Signature: (I)[[I
 */
JNIEXPORT jobjectArray JNICALL Java_com_study_jnilearn_ObjectArray_initInt2DArray
  (JNIEnv *, jobject, jint);

 #ifdef __cplusplus
}
 #endif
 #endif


// ObjectArray.c

 #include "com_study_jnilearn_ObjectArray.h"
/*
 * Class:     com_study_jnilearn_ObjectArray
 * Method:    initInt2DArray
 * Signature: (I)[[I
 */
JNIEXPORT jobjectArray JNICALL Java_com_study_jnilearn_ObjectArray_initInt2DArray
  (JNIEnv *env, jobject obj, jint size)
{
    jobjectArray result;
    jclass clsIntArray;
    jint i,j;
    // 1.获得一个int型二维数组类的引用
    clsIntArray = (*env)->FindClass(env,"[I");
    if (clsIntArray == NULL)
    {
        return NULL;
    }
    // 2.创建一个数组对象（里面每个元素用clsIntArray表示）
    result = (*env)->NewObjectArray(env,size,clsIntArray,NULL);
```

```
    if (result == NULL)
    {
        return NULL;
    }

    // 3.为数组元素赋值
    for (i = 0; i < size; ++i)
    {
        jint buff[256];
        jintArray intArr = (*env)->NewIntArray(env,size);
        if (intArr == NULL)
        {
            return NULL;
        }
        for (j = 0; j < size; j++)
        {
            buff[j] = i + j;
        }
        (*env)->SetIntArrayRegion(env,intArr, 0,size,buff);
        (*env)->SetObjectArrayElement(env,result, i, intArr);
        (*env)->DeleteLocalRef(env,intArr);
    }

    return result;
}
```

结果：

```
arr[0][0]=0
arr[0][1]=1
arr[0][2]=2
arr[0][0]=1
arr[0][1]=2
arr[0][2]=3
arr[0][0]=2
arr[0][1]=3
arr[0][2]=4
```

本地函数 initInt2DArray 首先调用 JNI 函数 FindClass 获得一个 int 型的二维数组类的引用，传递给FindClass 的参数"`[I`"是 JNI class descript（JNI 类型描述符，后面为详细介绍），它对应着 JVM 中的int[]类型。如果 int[]类加载失败的话，FindClass 会返回 NULL，然后抛出一个java.lang.NoClassDefFoundError: `[I` 异常。

接下来，NewObjectArray 创建一个新的数组，这个数组里面的元素类型用 intArrCls(int[])类型来表示。函数NewObjectArray 只能分配第一维，JVM 没有与多维数组相对应的数据结构，JNI 也没有提供类似的函数来创建二维数组。由于 JNI 中的二维数组直接操作的是 JVM 中的数据结构，相比 JAVA 和 C/C++创建二维数组要复

杂很多。给二维数组设置数据的方式也非常直接，首先用 NewIntArray 创建一个 JNI 的 int 数组，并为每个数组元素分配空间，然后用 SetIntArrayRegion 把 buff[]缓冲中的内容复制到新分配的一维数组中去，最后在外层循环中依次将 int[]数组赋值到 jobjectArray 数组中，一维数组中套一维数组，就形成了一个所谓的二维数组。

另外，为了避免在循环内创建大量的 JNI 局部引用，造成 JNI 引用表溢出，所以在外层循环中每次都要调用DeleteLocalRef 将新创建的 jintArray 引用从引用表中移除。在 JNI 中，只有 jobject 以及子类属于引用变量，会占用引用表的空间，jint，jfloat，jboolean 等都是基本类型变量，不会占用引用表空间，即不需要释放。引用表最大空间为 512 个，如果超出这个范围，JVM 就会挂掉。

7

C/C++ 访问 Java 实例方法和静态方法

通过前面 5 章的学习，我们知道了如何通过 JNI 函数来访问 JVM 中的基本数据类型、字符串和数组这些数据类型。下一步我们来学习本地代码如何与 JVM 中任意对象的属性和方法进行交互。比如本地代码调用 Java 层某个对象的方法或属性，也就是通常我们所说的来自 C/C++层本地函数的 callback（回调）。这个知识点分 2 篇文章分别介绍，本篇先介绍方法回调，在第七章中介绍本地代码访问 Java 的属性。

在这之前，先回顾一下在 Java 中调用一个方法时在 JVM 中的实现原理，有助于下面讲解本地代码调用 Java 方法实现的机制。写过 Java 的童鞋都知道，调用一个类的静态方法，直接通过 `类名.方法` 就可以调用。这也太简单了，有什么好讲的呢。但在这个调用过程中，JVM 是帮我们做了很多工作的。当我们在运行一个 Java 程序时，JVM 会先将程序运行时所要用到所有相关的 class 文件加载到 JVM 中，并采用按需加载的方式加载，也就是说某个类只有在被用到的时候才会被加载，这样设计的目的也是为了提高程序的性能和节约内存。所以我们在用类名调用一个静态方法之前，JVM 首先会判断该类是否已经加载，如果没有被 ClassLoader 加载到 JVM 中，JVM 会从classpath 路径下查找该类，如果找到了，会将其加载到 JVM 中，然后才是调用该类的静态方法。如果没有找到，JVM 会抛出 java.lang.ClassNotFoundException 异常，提示找不到这个类。ClassLoader 是 JVM 加载class 字节码文件的一种机制，不太了解的童鞋，请移步阅读《深入分析Java ClassLoader原理》(http://blog.csdn.net/xyang81/article/details/7292380) 一文。其实在 JNI 开发当中，本地代码也是按照上面的流程来访问类的静态方法或实例方法的，下面通过一个例子，详细介绍本地代码调用 Java 方法流程当中的每个步聚：

```
package com.study.jnilearn;

/**
 * AccessMethod.java
 * 本地代码访问类的实例方法和静态方法
 * @author yangxin
 */
public class AccessMethod {

    public static native void callJavaStaticMethod();
    public static native void callJavaInstaceMethod();

    public static void main(String[] args) {
        callJavaStaticMethod();
        callJavaInstaceMethod();
    }

    static {
        System.loadLibrary("AccessMethod");
    }
}
```

```
package com.study.jnilearn;
```

```
/**
 * ClassMethod.java
 * 用于本地代码调用
 * @author yangxin
 */
public class ClassMethod {

    private static void callStaticMethod(String str, int i) {
        System.out.format("ClassMethod::callStaticMethod called!-->str=%s," +
                " i=%d\n", str, i);
    }

    private void callInstanceMethod(String str, int i) {
        System.out.format("ClassMethod::callInstanceMethod called!-->str=%s, " +
                "i=%d\n", str, i);
    }
}
```

由AccessMethod.class生成的头文件

```
/* DO NOT EDIT THIS FILE - it is machine generated */
 #include <jni.h>
/* Header for class com_study_jnilearn_AccessMethod */

 #ifndef _Included_com_study_jnilearn_AccessMethod
 #define _Included_com_study_jnilearn_AccessMethod
 #ifdef __cplusplus
extern "C" {
 #endif
/*
 * Class:     com_study_jnilearn_AccessMethod
 * Method:    callJavaStaticMethod
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_com_study_jnilearn_AccessMethod_callJavaStaticMethod
  (JNIEnv *, jclass);

/*
 * Class:     com_study_jnilearn_AccessMethod
 * Method:    callJavaInstaceMethod
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_com_study_jnilearn_AccessMethod_callJavaInstaceMethod
  (JNIEnv *, jclass);

 #ifdef __cplusplus
```

```
}
 #endif
 #endif
```

本地代码对头文件中函数原型的实现

```
// AccessMethod.c

 #include "com_study_jnilearn_AccessMethod.h"

/*
 * Class:     com_study_jnilearn_AccessMethod
 * Method:    callJavaStaticMethod
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_com_study_jnilearn_AccessMethod_callJavaStaticMethod
(JNIEnv *env, jclass cls)
{
    jclass clazz = NULL;
    jstring str_arg = NULL;
    jmethodID mid_static_method;
    // 1、从classpath路径下搜索ClassMethod这个类，并返回该类的Class对象
    clazz =(*env)->FindClass(env,"com/study/jnilearn/ClassMethod");
    if (clazz == NULL) {
        return;
    }

    // 2、从clazz类中查找callStaticMethod方法
    mid_static_method = (*env)->GetStaticMethodID(env,clazz,"callStaticMethod","(Ljava/lang/String;I)V");
    if (mid_static_method == NULL) {
        printf("找不到callStaticMethod这个静态方法。");
        return;
    }

    // 3、调用clazz类的callStaticMethod静态方法
    str_arg = (*env)->NewStringUTF(env,"我是静态方法");
    (*env)->CallStaticVoidMethod(env,clazz,mid_static_method, str_arg, 100);

    // 删除局部引用
    (*env)->DeleteLocalRef(env,clazz);
    (*env)->DeleteLocalRef(env,str_arg);
}

/*
 * Class:     com_study_jnilearn_AccessMethod
 * Method:    callJavaInstaceMethod
```

```
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_com_study_jnilearn_AccessMethod_callJavaInstaceMethod
(JNIEnv *env, jclass cls)
{
    jclass clazz = NULL;
    jobject jobj = NULL;
    jmethodID mid_construct = NULL;
    jmethodID mid_instance = NULL;
    jstring str_arg = NULL;
    // 1、从classpath路径下搜索ClassMethod这个类，并返回该类的Class对象
    clazz = (*env)->FindClass(env, "com/study/jnilearn/ClassMethod");
    if (clazz == NULL) {
        printf("找不到'com.study.jnilearn.ClassMethod'这个类");
        return;
    }

    // 2、获取类的默认构造方法ID
    mid_construct = (*env)->GetMethodID(env,clazz, "<init>","()V");
    if (mid_construct == NULL) {
        printf("找不到默认的构造方法");
        return;
    }

    // 3、查找实例方法的ID
    mid_instance = (*env)->GetMethodID(env, clazz, "callInstanceMethod", "(Ljava/lang/String;I)V");
    if (mid_instance == NULL) {

        return;
    }

    // 4、创建该类的实例
    jobj = (*env)->NewObject(env,clazz,mid_construct);
    if (jobj == NULL) {
        printf("在com.study.jnilearn.ClassMethod类中找不到callInstanceMethod方法");
        return;
    }

    // 5、调用对象的实例方法
    str_arg = (*env)->NewStringUTF(env,"我是实例方法");
    (*env)->CallVoidMethod(env,jobj,mid_instance,str_arg,200);

    // 删除局部引用
    (*env)->DeleteLocalRef(env,clazz);
    (*env)->DeleteLocalRef(env,jobj);
```

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

运行结果：

```
ClassMethod::callStaticMethod called!-->str==我是静态方法,i=100
ClassMethod::callInstanceMethod called!-->str==我是实例方法,i=200
```

代码解析：

AccessMethod.java 是程序的入口，在 main 方法中，分别调用了 callJavaStaticMethod 和callJavaInstaceMethod 这两个 native 方法，用于测试 native 层调用 MethodClass.java 中的callStaticMethod 静态方法和callInstanceMethod 实例方法，这两个方法的返回值都为 Void，参数都有两个，分别为 String 和 int。

callJavaStaticMethod 静态方法实现说明

```
JNIEXPORT void JNICALL Java_com_study_jnilearn_AccessMethod_callJavaStaticMethod
(JNIEnv *env, jclass cls)
```

定位到AccessMethod.c的代码

```
(*env)->CallStaticVoidMethod(env,clazz,mid_static_method, str_arg, 100);
```

CallStaticVoidMethod函数的原型如下

```
void (JNICALL *CallStaticVoidMethod)(JNIEnv *env, jclass cls, jmethodID methodID, ...);
```

该函数接收 4 个参数：

- env：JNI 函数表指针
- cls：调用该静态方法的 Class 对象
- methodID：方法 ID（因为一个类中会存在多个方法，需要一个唯一标识来确定调用类中的哪个方法）
- 参数4：方法实参列表

根据函数参数的提示，分以下四步完成 Java 静态方法的回调：

第一步：调用 FindClass 函数，传入一个 Class 描述符，JVM 会从 classpath 路径下搜索该类，并返回jclass 类型（用于存储 Class 对象的引用）。注意 ClassMethod 的 Class 描述符为 com/study/jnilearn/ClassMethod，要将 .（点）全部换成 /（反斜杠）。

```
(*env)->FindClass(env,"com/study/jnilearn/ClassMethod");
```

第二步：调用 GetStaticMethodID 函数，从 ClassMethod 类中获取 callStaticMethod 方法 ID，返回jmethodID 类型（用于存储方法的引用）。实参 clazz 是第一步找到的 jclass 对象，实参"callStaticMethod"为方法名称，实参“(Ljava/lang/String;I)V”为方法的签名。

```
(*env)->GetStaticMethodID(env,clazz,"callStaticMethod","(Ljava/lang/String;I)V");
```

第三步：调用 CallStaticVoidMethod 函数，执行 ClassMethod.callStaticMethod 方法调用。str_arg 和100是 callStaticMethod 方法的实参。

```
str_arg = (*env)->NewStringUTF(env,"我是静态方法");
(*env)->CallStaticVoidMethod(env,clazz,mid_static_method, str_arg, 100);
```

注意：JVM 针对所有数据类型的返回值都定义了相关的函数。上面 callStaticMethod 方法的返回类型为 void，所以调用 CallStaticVoidMethod。根据返回值类型不同，JNI 提供了一系列不同返回值的函数，如：CallStaticIntMethod、CallStaticFloatMethod、CallStaticShortMethod、CallStaticObjectMethod等，分别表示调用返回值为 int、float、short、Object 类型的函数，引用类型统一调用CallStaticObjectMethod 函数。另外，每种返回值类型的函数都提供了接收3种实参类型的实现：CallStaticXXXMethod(env, clazz, methodID, ...)，CallStaticXXXMethodV(env, clazz, methodID, va_list args)，CallStaticXXXMethodA(env, clazz, methodID, const jvalue *args)，分别表示：接收可变参数列表、接收 va_list 作为实参和接收 `const jvalue*` 为实参。下面是 `jni.h` 头文件中CallStaticVoidMethod 的三种实参的函数原型：

```
void (JNICALL *CallStaticVoidMethod)
    (JNIEnv *env, jclass cls, jmethodID methodID, ...);
  void (JNICALL *CallStaticVoidMethodV)
    (JNIEnv *env, jclass cls, jmethodID methodID, va_list args);
  void (JNICALL *CallStaticVoidMethodA)
    (JNIEnv *env, jclass cls, jmethodID methodID, const jvalue * args);
```

第四步：释放局部变量

```
// 删除局部引用
(*env)->DeleteLocalRef(env,clazz);
(*env)->DeleteLocalRef(env,str_arg);
```

虽然函数结束后，JVM 会自动释放所有局部引用变量所占的内存空间。但还是手动释放一下比较安全，因为在 JVM 中维护着一个引用表，用于存储局部和全局引用变量，经测试在 Android NDK 环境下，这个表的最大存储空间是512 个引用，如果超过这个数就会造成引用表溢出，JVM 崩溃。在 PC 环境下测试，不管申请多少局部引用也不释放都不会崩，我猜可能与 JVM 和 Android Dalvik 虚拟机实现方式不一样的原因。所以有申请就及时释放是一个好的习惯！（局部引用和全局引用在后面的文章中会详细介绍）

callInstanceMethod实例方法实现说明

```
JNIEXPORT void JNICALL Java_com_study_jnilearn_AccessMethod_callJavaInstaceMethod
(JNIEnv *env, jclass cls)
```

定位到AccessMethod.c的代码：

```
(*env)->CallVoidMethod(env,jobj,mid_instance,str_arg,200);
```

CallVoidMethod函数的原型如下

```
void (JNICALL *CallVoidMethod) (JNIEnv *env, jobject obj, jmethodID methodID, ...);
```

该函数接收 4 个参数： - env：JNI 函数表指针 - obj：调用该方法的实例 - methodID：方法 ID - 参数4：方法的实参列表

根据函数参数的提示，分以下六步完成 Java 静态方法的回调：

第一步：同调用静态方法一样，首先通过 FindClass 函数获取类的 Class 对象。

第二步：获取类的构造方法 ID，因为创建类的对象首先会调用类的构造方法。这里以默认构造方法为例。

```
(*env)->GetMethodID(env,clazz, "<init>","()V");
```

`<init>` 代表类的构造方法名称， `()V` 代表无参无返回值的构造方法（即默认构造方法）

第三步：调用 GetMethodID 获取 callInstanceMethod 的方法 ID。

```
(*env)->GetMethodID(env, clazz, "callInstanceMethod", "(Ljava/lang/String;I)V");
```

第四步：调用 NewObject 函数，创建类的实例对象。

```
<span style="font-size:18px;">(*env)->NewObject(env,clazz,mid_construct);</span>
```

第五步：调用 CallVoidMethod 函数，执行 ClassMethod.callInstanceMethod 方法调用，str_arg 和 200是方法实参。

```
str_arg = (*env)->NewStringUTF(env,"我是实例方法");
(*env)->CallVoidMethod(env,jobj,mid_instance,str_arg,200);
```

同 JNI 调用 Java 静态方法一样，JVM 针对所有数据类型的返回值都定义了相关的函数（CallXXXMethod），如：CallIntMethod、CallFloatMethod、CallObjectMethod 等，也同样提供了支持三种类型实参的函数实现，以 CallVoidMethod 为例，如下是 `jni.h` 头文件中该函数的原型：

```
void (JNICALL *CallVoidMethod)(JNIEnv *env, jobject obj, jmethodID methodID, ...);
void (JNICALL *CallVoidMethodV)(JNIEnv *env, jobject obj, jmethodID methodID, va_list args);
void (JNICALL *CallVoidMethodA)(JNIEnv *env, jobject obj, jmethodID methodID, const jvalue * args);
```

第六步：删除局部引用（从引用表中移除）。

```
// 删除局部引用
(*env)->DeleteLocalRef(env,clazz);
(*env)->DeleteLocalRef(env,jobj);
(*env)->DeleteLocalRef(env,str_arg);
```

方法签名

在上面的的例子中，无论是调用静态方法还是实例方法，都必须传入一个 jmethodID 的参数。因为在 Java 中存在方法重载（方法名相同，参数列表不同），所以要明确告诉 JVM 调用的是类或实例中的哪一个方法。调用 JNI 的 GetMethodID 函数获取一个 jmethodID 时，需要传入一个方法名称和方法签名，方法名称就是在 Java 中定义的方法名，方法签名的格式为：(形参参数类型列表)返回值。形参参数列表中，引用类型以 L 开头，后面紧跟类的全路径名（需将 `.` 全部替换成 `/`），以分号结尾。下面是一些示例：

Method Descriptor	Java Language Type
"()Ljava/lang/String;"	String f();
"(ILjava/lang/Class;)J"	long f(int i, Class c);
"([B)V"	String(byte[] bytes);

Java 基本类型与方法签名中参数类型和返回值类型的映射关系如下：

Field Descriptor	Java Language Type
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double

比如，`String fun(int a, float b, boolean c, String d)` 对应的 JNI 方法签名为：`"(IFZLjava/lang/String;)Ljava/lang/String;"` 。

总结：

- 调用静态方法使用 CallStaticXXXMethod/V/A 函数，XXX 代表返回值的数据类型。如：CallStaticIntMethod
- 调用实例方法使用 CallXXXMethod/V/A 函数，XXX 代表返回的数据类型，如：CallIntMethod
- 获取一个实例方法的 ID，使用 GetMethodID 函数，传入方法名称和方法签名
- 获以一个静态方法的 ID，使用 GetStaticMethodID 函数，传入方法名称和方法签名
- 获取构造方法 ID，方法名称使用""
- 获取一个类的 Class 实例，使用 FindClass 函数，传入类描述符。JVM 会从 classpath 目录下开始搜索。
- 创建一个类的实例，使用 NewObject 函数，传入 Class 引用和构造方法 ID
- 删除局部变量引用，使用 DeleteLocalRef，传入引用变量
- 方法签名格式：(形参参数列表)返回值类型。注意：形参参数列表之间不需要用空格或其它字符分隔
- 类描述符格式：L 包名路径/类名;，包名之间用/分隔。如：Ljava/lang/String;
- 调用 GetMethodID 获取方法 ID 和调用 FindClass 获取 Class 实例后，要做异常判断

示例代码下载地址：https://code.csdn.net/xyang81/jnilearn

8

C/C++ 访问 Java 实例变量和静态变量

实例变量和静态变量

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和修改。静态变量也称为类变量（属性），在所有实例对象中共享同一份数据，可以直接通过【类名.变量名】来访问。实例变量也称为成员变量（属性），每个实例都拥有一份实例变量数据的拷贝，它们之间修改后的数据互不影响。下面看一个例子：

```
package com.study.jnilearn;

/**
 * C/C++访问类的实例变量和静态变量
 * @author yangxin
 */
public class AccessField {

    private native static void accessInstanceField(ClassField obj);

    private native static void accessStaticField();

    public static void main(String[] args) {
        ClassField obj = new ClassField();
        obj.setNum(10);
        obj.setStr("Hello");

        // 本地代码访问和修改ClassField为中的静态属性num
        accessStaticField();
        accessInstanceField(obj);

        // 输出本地代码修改过后的值
        System.out.println("In Java--->ClassField.num = " + obj.getNum());
        System.out.println("In Java--->ClassField.str = " + obj.getStr());
    }

    static {
        System.loadLibrary("AccessField");
    }

}
```

AccessField 是程序的入口类，定义了两个 native 方法：accessInstanceField 和 accessStaticField，分别用于演示在本地代码中访问 Java 类中的实例变量和静态变量。其中 accessInstaceField 方法访问的是类的实例变量，所以该方法需要一个 ClassField 实例作为形参，用于访问该对象中的实例变量。

```
package com.study.jnilearn;

/**
 * ClassField.java
 * 用于本地代码访问和修改该类的属性
 * @author yangxin
 *
 */
public class ClassField {

    private static int num;

    private String str;

    public int getNum() {
        return num;
    }

    public void setNum(int num) {
        ClassField.num = num;
    }

    public String getStr() {
        return str;
    }

    public void setStr(String str) {
        this.str = str;
    }
}
```

在本例中没有将实例变量和静态变量定义在程序入口类中，新建了一个 ClassField 的类来定义类的属性，目的是为了加深在 C/C++ 代码中可以访问任意 Java 类中的属性。在这个类中定义了一个 int 类型的实例变量 num，和一个 java.lang.String 类型的静态变量 str。这两个变量会被本地代码访问和修改。

```
/* DO NOT EDIT THIS FILE - it is machine generated */
 #include <jni.h>
/* Header for class com_study_jnilearn_AccessField */

 #ifndef _Included_com_study_jnilearn_AccessField
 #define _Included_com_study_jnilearn_AccessField
 #ifdef __cplusplus
extern "C" {
 #endif
/*
```

```
 * Class:     com_study_jnilearn_AccessField
 * Method:    accessInstanceField
 * Signature: (Lcom/study/jnilearn/ClassField;)V
 */
JNIEXPORT void JNICALL Java_com_study_jnilearn_AccessField_accessInstanceField
  (JNIEnv *, jclass, jobject);

/*
 * Class:     com_study_jnilearn_AccessField
 * Method:    accessStaticField
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_com_study_jnilearn_AccessField_accessStaticField
  (JNIEnv *, jclass);

 #ifdef __cplusplus
}
 #endif
 #endif
```

以上代码是程序入口类AccessField.class为native方法生成的本地代码函数原型头文件。

```
// AccessField.c

 #include "com_study_jnilearn_AccessField.h"

/*
 * Class:     com_study_jnilearn_AccessField
 * Method:    accessInstanceField
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_com_study_jnilearn_AccessField_accessInstanceField
(JNIEnv *env, jclass cls, jobject obj)
{
    jclass clazz;
    jfieldID fid;
    jstring j_str;
    jstring j_newStr;
    const char *c_str = NULL;

    // 1.获取AccessField类的Class引用
    clazz = (*env)->GetObjectClass(env,obj);
    if (clazz == NULL) {
        return;
    }
```

```
    // 2. 获取AccessField类实例变量str的属性ID
    fid = (*env)->GetFieldID(env,clazz,"str", "Ljava/lang/String;");
    if (clazz == NULL) {
        return;
    }

    // 3. 获取实例变量str的值
    j_str = (jstring)(*env)->GetObjectField(env,obj,fid);

    // 4. 将unicode编码的java字符串转换成C风格字符串
    c_str = (*env)->GetStringUTFChars(env,j_str,NULL);
    if (c_str == NULL) {
        return;
    }
    printf("In C--->ClassField.str = %s\n", c_str);
    (*env)->ReleaseStringUTFChars(env, j_str, c_str);

    // 5. 修改实例变量str的值
    j_newStr = (*env)->NewStringUTF(env, "This is C String");
    if (j_newStr == NULL) {
        return;
    }

    (*env)->SetObjectField(env, obj, fid, j_newStr);

    // 6.删除局部引用
    (*env)->DeleteLocalRef(env, clazz);
    (*env)->DeleteLocalRef(env, j_str);
    (*env)->DeleteLocalRef(env, j_newStr);
}

/*
 * Class:     com_study_jnilearn_AccessField
 * Method:    accessStaticField
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_com_study_jnilearn_AccessField_accessStaticField
(JNIEnv *env, jclass cls)
{
    jclass clazz;
    jfieldID fid;
    jint num;

    //1.获取ClassField类的Class引用
    clazz = (*env)->FindClass(env,"com/study/jnilearn/ClassField");
```

```
    if (clazz == NULL) {    // 错误处理
        return;
    }

    //2.获取ClassField类静态变量num的属性ID
    fid = (*env)->GetStaticFieldID(env, clazz, "num", "I");
    if (fid == NULL) {
        return;
    }

    // 3.获取静态变量num的值
    num = (*env)->GetStaticIntField(env,clazz,fid);
    printf("In C--->ClassField.num = %d\n", num);

    // 4.修改静态变量num的值
    (*env)->SetStaticIntField(env, clazz, fid, 80);

    // 删除属部引用
    (*env)->DeleteLocalRef(env,clazz);
}
```

以上代码是对头文件中函数原型的实现。

运行程序，输出结果如下：

```
In Java--->ClassField.num=80
In Java--->ClassField.str=This is C String
In c--->ClassField.num=10
In c--->ClassField.str=Hello
```

代码解析

访问实例变量

在 main 方法中，通过调用 accessInstanceField()方法来调用本地函数 Java_com_study_jnilearn_AccessField_accessInstanceField。

```
j_str = (jstring)(*env)->GetObjectField(env,obj,fid);
```

该函数就是用于获取 ClassField 对象中 num 的值。下面是函数的原型

```
jobject (JNICALL *GetObjectField) (JNIEnv *env, jobject obj, jfieldID fieldID);
```

因为实例变量 str 是 String 类型，属于引用类型。在 JNI 中获取引用类型字段的值，调用 GetObjectField 函数获取。同样的，获取其它类型字段值的函数还有 GetIntField，GetFloatField，GetDoubleField，GetBooleanField 等。这些函数有一个共同点，函数参数都是一样的，只是函数名不同，我们只需学习其中一个函数如何调用即可，依次类推，就自然知道其它函数的使用方法。

GetObjectField 函数接受 3 个参数，env 是 JNI 函数表指针，obj 是实例变量所属的对象，fieldID 是变量的ID（也称为属性描述符或签名），和上一章中方法描述符是同一个意思。env 和 obj 参数从Java_com_study_jnilearn_AccessField_accessInstanceField 函数形参列表中可以得到，那 fieldID 怎么获取呢？了解 Java 反射的童鞋应该知道，在 Java 中任何一个类的 .class 字节码文件被加载到内存中之后，该 class 子节码文件统一使用 Class 类来表示该类的一个引用（相当于 Java 中所有类的基类是 Object一样）。然后就可以从该类的 Class 引用中动态的获取类中的任意方法和属性。注意：Class 类在 Java SDK 继承体系中是一个独立的类，没有继承自 Object。请看下面的例子，通过 Java 反射机制，动态的获取一个类的私有实例变量的值。

```
public static void main(String[] args) throws Exception {
    ClassField obj = new ClassField();
    obj.setStr("YangXin");
    // 获取ClassField字节码对象的Class引用
    Class<?> clazz = obj.getClass();
    // 获取str属性
    Field field = clazz.getDeclaredField("str");
    // 取消权限检查，因为Java语法规定，非public属性是无法在外部访问的
    field.setAccessible(true);
    // 获取obj对象中的str属性的值
    String str = (String)field.get(obj);
    System.out.println("str = " + str);
}
```

运行程序后，输出结果当然是打印出 str 属性的值“YangXin”。所以我们在本地代码中调用 JNI 函数访问 Java 对象中某一个属性的时候，首先第一步就是要获取该对象的 Class 引用，然后在 Class 中查找需要访问的字段ID，最后调用 JNI 函数的 GetXXXField 系列函数，获取字段(属性)的值。上例中，首先调用 GetObjectClass 函数获取 ClassField 的 Class 引用。

```
clazz = (*env)->GetObjectClass(env,obj);
```

然后调用 GetFieldID 函数从 Class 引用中获取字段的 ID（str 是字段名，Ljava/lang/String；是字段的类型）。

```
fid = (*env)->GetFieldID(env,clazz,"str", "Ljava/lang/String;");
```

最后调用 GetObjectField 函数，传入实例对象和字段 ID，获取属性的值。

```
j_str = (jstring)(*env)->GetObjectField(env,obj,fid);
```

调用 SetXXXField 系列函数，可以修改实例属性的值，最后一个参数为属性的值。引用类型全部调用SetObjectField 函数，基本类型调用 SetIntField、SetDoubleField、SetBooleanField 等。

```
(*env)->SetObjectField(env, obj, fid, j_newStr);
```

访问静态变量

访问静态变量和实例变量不同的是，获取字段 ID 使用 GetStaticFieldID，获取和修改字段的值使用 Get/SetStaticXXXField 系列函数，比如上例中获取和修改静态变量 num。

```
// 3.获取静态变量num的值
num = (*env)->GetStaticIntField(env,clazz,fid);
// 4.修改静态变量num的值
(*env)->SetStaticIntField(env, clazz, fid, 80);
```

总结

- 由于 JNI 函数是直接操作J VM 中的数据结构，不受 Java 访问修饰符的限制。即，在本地代码中可以调用 JNI 函数可以访问 Java 对象中的非 public 属性和方法
- 访问和修改实例变量操作步骤：
    - 调用 GetObjectClass 函数获取实例对象的 Class 引用
    - 调用 GetFieldID 函数获取 Class 引用中某个实例变量的 ID
    - 调用 GetXXXField 函数获取变量的值，需要传入实例变量所属对象和变量 ID
    - 调用 SetXXXField 函数修改变量的值，需要传入实例变量所属对象、变量 ID 和变量的值
- 访问和修改静态变量操作步骤：
    - 调用 FindClass 函数获取类的 Class 引用
    - 调用 GetStaticFieldID 函数获取 Class 引用中某个静态变量 ID
    - 调用 GetStaticXXXField 函数获取静态变量的值，需要传入变量所属 Class 的引用和变量 ID
    - 调用 SetStaticXXXField 函数设置静态变量的值，需要传入变量所属 Class 的引用、变量 ID和变量的值

9

JNI 调用构造方法和父类实例方法

在前面我们学习到了在 Native 层如何调用 Java 静态方法和实例方法，其中调用实例方法的示例代码中也提到了调用构造函数来实始化一个对象，但没有详细介绍，一带而过了。还没有阅读过的同学请移步《 JNI——C/C++ 访问 Java 实例方法和静态方法 》阅读。这章详细来介绍下初始一个对象的两种方式，以及如何调用子类对象重写的父类实例方法。

构造方法和父类实例方法

我们先回过一下，在 Java 中实例化一个对象和调用父类实例方法的流程。先看一段代码：

```
package com.study.jnilearn;
public class Animal {
    public void run() {
        System.out.println("Animal.run...");
    }
}

package com.study.jnilearn;
public class Cat extends Animal {
    @Override
    public void run() {
        System.out.println(name + " Cat.run...");
    }
}

public static void main(String[] args) {
    Animal cat = new Cat("汤姆");
    cat.run();
}
```

正如你所看到的那样，上面这段代码非常简单，有两个类 Animal 和 Cat，Animal 类中定义了 run 和 getName 两个方法，Cat 继承自 Animal，并重写了父类的 run 方法。在 main 方法中，首先定义了一个 Animal 类型的变量 cat，并指向了 Cat 类的实例对象，然后调用了它的 run 方法。**在执行 new Cat(“汤姆”)这段代码时，会先为 Cat 类分配内存空间（所分配的内存空间大小由 Cat 类的成员变量数量决定），然后调用 Cat 的带参构造方法初始化对象。** cat 是 Animal 类型，但它指向的是 Cat 实例对象的引用，而且 Cat 重写了父类的 run 方法，因为调用 run 方法时有多态存在，所以访问的是 Cat 的 run 而非 Animal 的 run，运行后打印的结果为：`汤姆 Cat.run…`

如果要调用父类的 run 方法，只需在 Cat 的 run 方法中调用 super.run() 即可，相当的简单。

写过 C 或 C++ 的同学应该都有一个很深刻的内存管理概念，栈空间和堆空间，栈空间的内存大小受操作系统限制，由操作系统自动来管理，速度较快，所以在函数中定义的局部变量、函数形参变量都存储在栈空间。操作系

统没有限制堆空间的内存大小，只受物理内存的限制，内存需要程序员自己管理。在 C 语言中用 malloc 关键字动态分配的内存和在 C++ 中用 new 创建的对象所分配内存都存储在堆空间，内存使用完之后分别用 free 或 delete/delete[] 释放。这里不过多的讨论 C/C++ 内存管理方面的知识，有兴趣的同学请自行百度。做 Java 的童鞋众所周知，写 Java 程序是不需要手动来管理内存的，内存管理那些烦锁的事情全都交由一个叫 GC 的线程来管理（当一个对象没有被其它对象所引用时，该对象就会被 GC 释放）。但我觉得 Java 内部的内存管理原理和 C/C++ 是非常相似的，上例中，Animal cat = new Cat(“汤姆”); 局部变量 cat 存放在栈空间上，new Cat (“汤姆”);创建的实例对象存放在堆空间，返回一个内存地址的引用，存储在 cat 变量中。这样就可以通过 cat 变量所指向的引用访问 Cat 实例当中所有可见的成员了。

所以创建一个对象分为 2 步：

- 为对象分配内存空间
- 初始化对象（调用对象的构造方法）

下面通过一个示例来了解在 JNI 中是如何调用对象构造方法和父类实例方法的。为了让示例能清晰的体现构造方法和父类实例方法的调用流程，定义了 Animal 和 Cat 两个类，Animal 定义了一个 String 形参的构造方法，一个成员变量 name、两个成员函数 run 和 getName，Cat 继承自 Animal，并重写了 run 方法。在 JNI 中实现创建 Cat 对象的实例，调用 Animal 类的 run 和 getName 方法。代码如下所示。

```
// Animal.java
package com.study.jnilearn;
public class Animal {

    protected String name;

    public Animal(String name) {
        this.name = name;
        System.out.println("Animal Construct call...");
    }

    public String getName() {
        System.out.println("Animal.getName Call...");
        return this.name;
    }

    public void run() {
        System.out.println("Animal.run...");
    }
}

// Cat.java
package com.study.jnilearn;
```

```
public class Cat extends Animal {

    public Cat(String name) {
        super(name);
        System.out.println("Cat Construct call....");
    }

    @Override
    public String getName() {
        return "My name is " + this.name;
    }

    @Override
    public void run() {
        System.out.println(name + " Cat.run...");
    }
}

// AccessSuperMethod.java
package com.study.jnilearn;
public class AccessSuperMethod {

    public native static void callSuperInstanceMethod();

    public static void main(String[] args) {
        callSuperInstanceMethod();
    }

    static {
        System.loadLibrary("AccessSuperMethod");
    }
}
```

AccessSuperMethod 类是程序的入口，其中定义了一个 native 方法 callSuperInstanceMethod。用 javah 生成的 jni 函数原型如下。

```
/* Header for class com_study_jnilearn_AccessSuperMethod */

 #ifndef _Included_com_study_jnilearn_AccessSuperMethod
 #define _Included_com_study_jnilearn_AccessSuperMethod
 #ifdef __cplusplus
extern "C" {
 #endif
/*
 * Class:     com_study_jnilearn_AccessSuperMethod
```

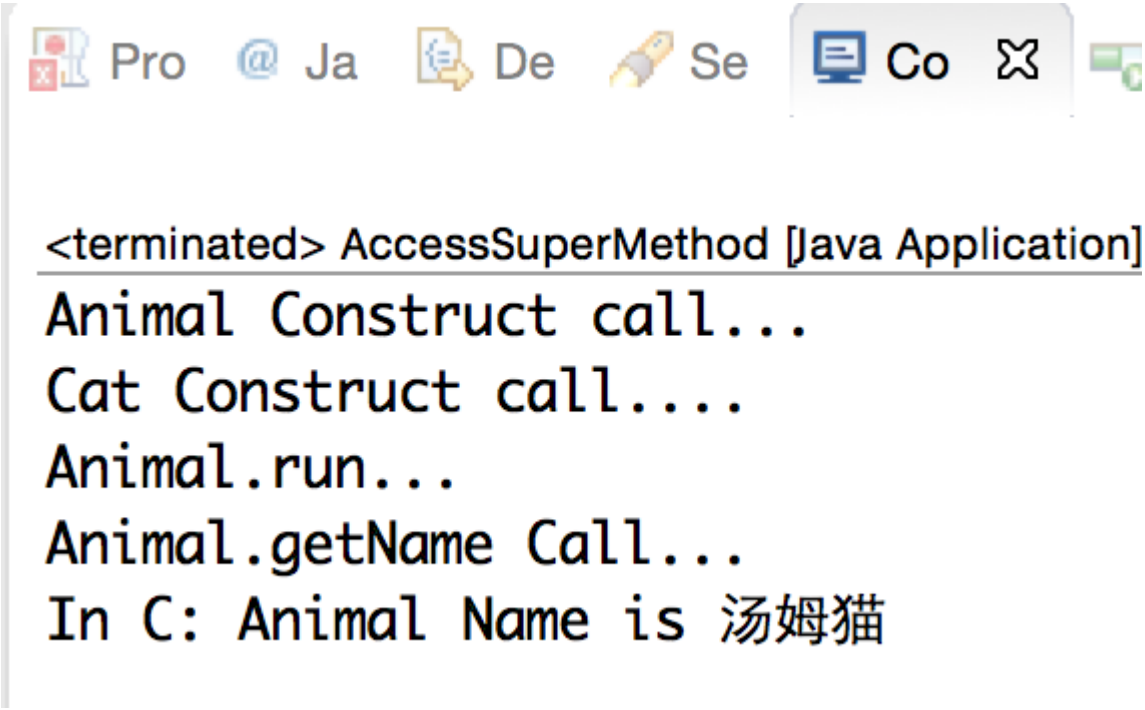
```
 * Method:    callSuperInstanceMethod
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_com_study_jnilearn_AccessSuperMethod_callSuperInstanceMethod
  (JNIEnv *, jclass);

 #ifdef __cplusplus
}
 #endif
 #endif
```

实现 Java_com_study_jnilearn_AccessSuperMethod_callSuperInstanceMethod 函数，如下所示。

```
/ AccessSuperMethod.c

 #include "com_study_jnilearn_AccessSuperMethod.h"

JNIEXPORT void JNICALL Java_com_study_jnilearn_AccessSuperMethod_callSuperInstanceMethod
  (JNIEnv *env, jclass cls)
{
    jclass cls_cat;
    jclass cls_animal;
    jmethodID mid_cat_init;
    jmethodID mid_run;
    jmethodID mid_getName;
    jstring c_str_name;
    jobject obj_cat;
    const char *name = NULL;

    // 1、获取Cat类的class引用
    cls_cat = (*env)->FindClass(env, "com/study/jnilearn/Cat");
    if (cls_cat == NULL) {
        return;
    }

    // 2、获取Cat的构造方法ID(构造方法的名统一为：<init>)
    mid_cat_init = (*env)->GetMethodID(env, cls_cat, "<init>", "(Ljava/lang/String;)V");
    if (mid_cat_init == NULL) {
        return; // 没有找到只有一个参数为String的构造方法
    }

    // 3、创建一个String对象，作为构造方法的参数
    c_str_name = (*env)->NewStringUTF(env, "汤姆猫");
    if (c_str_name == NULL) {
        return; // 创建字符串失败（内存不够）
    }
```

```
    // 4、创建Cat对象的实例(调用对象的构造方法并初始化对象)
    obj_cat = (*env)->NewObject(env,cls_cat, mid_cat_init,c_str_name);
    if (obj_cat == NULL) {
        return;
    }

    //--------------- 5、调用Cat父类Animal的run和getName方法 ---------------
    cls_animal = (*env)->FindClass(env, "com/study/jnilearn/Animal");
    if (cls_animal == NULL) {
        return;
    }

    // 例1: 调用父类的run方法
    mid_run = (*env)->GetMethodID(env, cls_animal, "run", "()V");    // 获取父类Animal中run方法的id
    if (mid_run == NULL) {
        return;
    }

    // 注意：obj_cat是Cat的实例，cls_animal是Animal的Class引用，mid_run是Animal类中的方法ID
    (*env)->CallNonvirtualVoidMethod(env, obj_cat, cls_animal, mid_run);

    // 例2：调用父类的getName方法
    // 获取父类Animal中getName方法的id
    mid_getName = (*env)->GetMethodID(env, cls_animal, "getName", "()Ljava/lang/String;");
    if (mid_getName == NULL) {
        return;
    }

    c_str_name = (*env)->CallNonvirtualObjectMethod(env, obj_cat, cls_animal, mid_getName);
    name = (*env)->GetStringUTFChars(env, c_str_name, NULL);
    printf("In C: Animal Name is %s\n", name);

    // 释放从java层获取到的字符串所分配的内存
    (*env)->ReleaseStringUTFChars(env, c_str_name, name);

quit:
    // 删除局部引用（jobject或jobject的子类才属于引用变量），允许VM释放被局部变量所引用的资源
    (*env)->DeleteLocalRef(env, cls_cat);
    (*env)->DeleteLocalRef(env, cls_animal);
    (*env)->DeleteLocalRef(env, c_str_name);
    (*env)->DeleteLocalRef(env, obj_cat);
}
```

运行结果



代码讲解 – 调用构造方法

调用构造方法和调用对象的实例方法方式是相似的，传入”< init >”作为方法名查找类的构造方法ID，然后调用JNI函数NewObject调用对象的构造函数初始化对象。如下代码所示。

```
obj_cat = (*env)->NewObject(env,cls_cat,mid_cat_init,c_str_name);
```

上述这段代码调用了 JNI 函数 NewObject 创建了 Class 引用的一个实例对象。这个函数做了 2 件事情

- 创建一个未初始化的对象并分配内存空间
- 调用对象的构造函数初始化对象。这两步也可以分开进行，为对象分配内存，然后再初始化对象，如下代码所示：

```
// 1、创建一个未初始化的对象，并分配内存
obj_cat = (*env)->AllocObject(env, cls_cat);
if (obj_cat) {
  // 2、调用对象的构造函数初始化对象
  (*env)->CallNonvirtualVoidMethod(env,obj_cat, cls_cat, mid_cat_init, c_str_name);
  if ((*env)->ExceptionCheck(env)) { // 检查异常
     goto quit;
  }
}
```

AllocObject 函数创建的是一个未初始化的对象，后面在用这个对象之前，必须调用CallNonvirtualVoidMethod 调用对象的构造函数初始化该对象。而且在使用时一定要非常小心，确保在一个对象上面，构造函数最多被调用一次。有时，先创建一个初始化的对象，然后在合适的时间再调用构造函数的方式是很有用的。尽管如此，大部分情况下，应该使用 NewObject，尽量避免使用容易出错的 AllocObject/CallNonvirtualVoidMethod 函数。

代码讲解 – 调用父类实例方法

如果一个方法被定义在父类中，在子类中被覆盖，也可以调用父类中的这个实例方法。JNI 提供了一系列函数Call NonvirtualXXXMethod 来支持调用各种返回值类型的实例方法。调用一个定义在父类中的实例方法，须遵循下面的步骤。

使用 GetMethodID 函数从一个指向父类的 Class 引用当中获取方法 ID。

```
cls_animal = (*env)->FindClass(env, "com/study/jnilearn/Animal");
if (cls_animal == NULL) {
    return;
}

//例1： 调用父类的run方法
mid_run = (*env)->GetMethodID(env, cls_animal, "run", "()V");    // 获取父类Animal中run方法的id
if (mid_run == NULL) {
    return;
}
```

传入子类对象、父类 Class 引用、父类方法 ID 和参数，并调用 CallNonvirtualVoidMethod、 CallNonvirtual BooleanMethod、CallNonvirtualIntMethod 等一系列函数中的一个。其中CallNonvirtualVoidMethod 也可以被用来调用父类的构造函数。

```
// 注意：obj_cat是Cat的实例，cls_animal是Animal的Class引用，mid_run是Animal类中的方法ID
(*env)->CallNonvirtualVoidMethod(env, obj_cat, cls_animal, mid_run);
```

其实在开发当中，这种调用父类实例方法的情况是很少遇到的，通常在 JAVA 中可以很简单地做到: `super.func()` ;但有些特殊需求也可能会用到，所以知道有这么回事还是很有必要的。

10

JNI 调用性能测试及优化

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 程序中调用声明的 native 方法就间接的调用到了 C/C++ 编写的函数了，在 C/C++ 中写的程序可以避开 JVM 的内存开销过大的限制、处理高性能的计算、调用系统服务等功能。同时也学习到了在本地代码中通过 JNI 提供的接口，调用 Java 程序中的任意方法和对象的属性。这是 JNI 提供的一些优势。但做过 Java 的童鞋应该都明白，Java 程序是运行在 JVM 上的，所以在 Java 中调用 C/C++ 或其它语言这种跨语言的接口时，或者说在 C/C++ 代码中通过 JNI 接口访问 Java 中对象的方法或属性时，相比 Java 调用自己的方法，性能是非常低的！网上有朋友针对 Java 调用本地接口，Java 调 Java 方法做了一次详细的测试，来充分说明在享受 JNI 给程序带来优势的同时，也要接受其所带来的性能开销，请看下面一组测试数据。

Java 调用 JNI 空函数与 Java 调用 Java 空方法性能测试。

测试环境：JDK1.4.2_19、JDK1.5.0_04 和 JDK1.6.0_14，测试的重复次数都是一亿次。测试结果的绝对数值意义不大，仅供参考。因为根据 JVM 和机器性能的不同，测试所产生的数值也会不同，但不管什么机器和 JVM 应该都能反应同一个问题，Java 调用 native 接口，要比 Java 调用 Java 方法性能要低很多。

Java 调用 Java 空方法的性能：

JDK 版本	Java 调 Java 耗时	平均每秒调用次数
1.6	329ms	303951367次
1.5	312ms	320512820次
1.4	312ms	27233115次

Java 调用 JNI 空函数的性能：

JDK版本	Java调Java耗时	平均每秒调用次数
1.6	1531ms	65316786次
1.5	1891ms	52882072次
1.4	3672ms	27233115次

从上述测试数据可以看出 JDK 版本越高，JNI 调用的性能也越好。在 JDK1.5 中，仅仅是空方法调用，JNI 的性能就要比 Java 内部调用慢将近 5 倍，而在 JDK1.4 下更是慢了十多倍。

JNI查找方法ID、字段ID、Class引用性能测试

当我们在本地代码中要访问 Java 对象的字段或调用它们的方法时，本机代码必须调用 FindClass()、GetFieldID()、GetStaticFieldID、GetMethodID()和 GetStaticMethodID()。对于 GetFieldID()、GetStaticFieldID、GetMethodID() 和 GetStaticMethodID()，为特定类返回的 ID 不会在 JVM 进程的生存期内发生变化。但

是，获取字段或方法的调用有时会需要在 JVM 中完成大量工作，因为字段和方法可能是从超类中继承而来的，这会让 JVM 向上遍历类层次结构来找到它们。由于 ID 对于特定类是相同的，因此只需要查找一次，然后便可重复使用。同样，查找类对象的开销也很大，因此也应该缓存它们。下面对调用 JNI 接口 FindClass 查找 Class、GetFieldID 获取类的字段 ID 和 GetFieldValue 获取字段的值的性能做的一个测试。缓存表示只调用一次，不缓存就是每次都调用相应的 JNI 接口：

java.version = 1.6.0_14

- JNI 字段读取 (缓存Class=false ,缓存字段ID=false) 耗时 : 79172 ms 平均每秒 : 1263072
- JNI 字段读取 (缓存Class=true ,缓存字段ID=false) 耗时 : 25015 ms 平均每秒 : 3997601
- JNI 字段读取 (缓存Class=false ,缓存字段ID=true) 耗时 : 50765 ms 平均每秒 : 1969861
- JNI 字段读取 (缓存Class=true ,缓存字段ID=true) 耗时 : 2125 ms 平均每秒 : 47058823

java.version = 1.5.0_04

- JNI 字段读取 (缓存Class=false ,缓存字段ID=false) 耗时 : 87109 ms 平均每秒 : 1147987
- JNI 字段读取 (缓存Class=true ,缓存字段ID=false) 耗时 : 32031 ms 平均每秒 : 3121975
- JNI 字段读取 (缓存Class=false ,缓存字段ID=true) 耗时 : 51657 ms 平均每秒 : 1935846
- JNI 字段读取 (缓存Class=true ,缓存字段ID=true) 耗时 : 2187 ms 平均每秒 : 45724737

java.version = 1.4.2_19

- JNI 字段读取 (缓存Class=false ,缓存字段ID=false) 耗时 : 97500 ms 平均每秒 : 1025641
- JNI 字段读取 (缓存Class=true ,缓存字段ID=false) 耗时 : 38110 ms 平均每秒 : 2623983
- JNI 字段读取 (缓存Class=false ,缓存字段ID=true) 耗时 : 55204 ms 平均每秒 : 1811462
- JNI 字段读取 (缓存Class=true ,缓存字段ID=true) 耗时 : 4187 ms 平均每秒 : 23883448

根据上面的测试数据得知，查找 class 和 ID (属性和方法 ID)消耗的时间比较大。只是读取字段值的时间基本上跟上面的 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：

```
package com.study.jnilearn;

public class AccessCache {

    private String str = "Hello";

    public native void accessField(); // 访问str成员变量
    public native String newString(char[] chars, int len); // 根据字符数组和指定长度创建String对象

    public static void main(String[] args) {
        AccessCache accessCache = new AccessCache();
        accessCache.nativeMethod();
        char chars[] = new char[7];
        chars[0] = '中';
        chars[1] = '华';
        chars[2] = '人';
        chars[3] = '民';
        chars[4] = '共';
        chars[5] = '和';
        chars[6] = '国';
        String str = accessCache.newString(chars, 6);
        System.out.println(str);
    }

    static {
        System.loadLibrary("AccessCache");
    }
}
```

javah 生成的头文件：com_study_jnilearn_AccessCache.h

```
/* DO NOT EDIT THIS FILE - it is machine generated */
 #include <jni.h>
/* Header for class com_study_jnilearn_AccessCache */
 #ifndef _Included_com_study_jnilearn_AccessCache
```

```
#define _Included_com_study_jnilearn_AccessCache
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     com_study_jnilearn_AccessCache
 * Method:    accessField
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_com_study_jnilearn_AccessCache_accessField(JNIEnv *, jobject);

/*
 * Class:     com_study_jnilearn_AccessCache
 * Method:    newString
 * Signature: ([CI)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_com_study_jnilearn_AccessCache_newString(JNIEnv *, jobject,
jcharArray, jint);

#ifdef __cplusplus
}
#endif
#endif
```

实现头文件中的函数：AccessCache.c

```
// AccessCache.c
#include "com_study_jnilearn_AccessCache.h"

JNIEXPORT void JNICALL Java_com_study_jnilearn_AccessCache_accessField
  (JNIEnv *env, jobject obj)
{
    // 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
    static jfieldID fid_str = NULL;
    jclass cls_AccessCache;
    jstring j_str;
    const char *c_str;
    cls_AccessCache = (*env)->GetObjectClass(env, obj); // 获取该对象的Class引用
    if (cls_AccessCache == NULL) {
        return;
    }

    // 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
    if (fid_str == NULL) {
        fid_str = (*env)->GetFieldID(env,cls_AccessCache,"str","Ljava/lang/String;");
```

```
        // 再次判断是否找到该类的str字段
        if (fid_str == NULL) {
            return;
        }
    }

    j_str = (*env)->GetObjectField(env, obj, fid_str);  // 获取字段的值
    c_str = (*env)->GetStringUTFChars(env, j_str, NULL);
    if (c_str == NULL) {
        return; // 内存不够
    }
    printf("In C:\n str = \"%s\"\n", c_str);
    (*env)->ReleaseStringUTFChars(env, j_str, c_str);   // 释放从从JVM新分配字符串的内存空间

    // 修改字段的值
    j_str = (*env)->NewStringUTF(env, "12345");
    if (j_str == NULL) {
        return;
    }
    (*env)->SetObjectField(env, obj, fid_str, j_str);

    // 释放本地引用
    (*env)->DeleteLocalRef(env,cls_AccessCache);
    (*env)->DeleteLocalRef(env,j_str);
}

JNIEXPORT jstring JNICALL Java_com_study_jnilearn_AccessCache_newString
(JNIEnv *env, jobject obj, jcharArray j_char_arr, jint len)
{
    jcharArray elemArray;
    jchar *chars = NULL;
    jstring j_str = NULL;
    static jclass cls_string = NULL;
    static jmethodID cid_string = NULL;
    // 注意：这里缓存局引用的做法是错误，这里做为一个反面教材提醒大家，下面会说到。
    if (cls_string == NULL) {
        cls_string = (*env)->FindClass(env, "java/lang/String");
        if (cls_string == NULL) {
            return NULL;
        }
    }

    // 缓存String的构造方法ID
    if (cid_string == NULL) {
        cid_string = (*env)->GetMethodID(env, cls_string, "<init>", "([C)V");
```

```
        if (cid_string == NULL) {
            return NULL;
        }
    }

    printf("In C array Len: %d\n", len);
    // 创建一个字符数组
    elemArray = (*env)->NewCharArray(env, len);
    if (elemArray == NULL) {
        return NULL;
    }

    // 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
    chars = (*env)->GetCharArrayElements(env, j_char_arr,NULL);
    if (chars == NULL) {
        return NULL;
    }
    // 将Java字符数组中的内容复制指定长度到新的字符数组中
    (*env)->SetCharArrayRegion(env, elemArray, 0, len, chars);

    // 调用String对象的构造方法，创建一个指定字符数组为内容的String对象
    j_str = (*env)->NewObject(env, cls_string, cid_string, elemArray);

    // 释放本地引用
    (*env)->DeleteLocalRef(env, elemArray);

    return j_str;
}
```

例1、在 Java_com_study_jnilearn_AccessCache_accessField 函数中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

```
static jfieldID fid_str = NULL;
```

在代码段

```
// 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
    if (fid_str == NULL) {
        fid_str = (*env)->GetFieldID(env,cls_AccessCache,"str","Ljava/lang/String;");

        // 再次判断是否找到该类的str字段
        if (fid_str == NULL) {
            return;
        }
    }
```

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 JVM 中获取，起到了缓存的作用。

在 Java_com_study_jnilearn_AccessCache_newString 函数中定义了两个变量 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，与方法和字段 ID 不一样，局部引用在函数结束后会被 JVM 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），当下次再调用 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。

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：

```
package com.study.jnilearn;

public class AccessCache {

    public static native void initIDs();

    public native void nativeMethod();
    public void callback() {
        System.out.println("AccessCache.callback invoked!");
    }

    public static void main(String[] args) {
        AccessCache accessCache = new AccessCache();
        accessCache.nativeMethod();
    }

    static {
        System.loadLibrary("AccessCache");
        initIDs();
    }
}
```

```
/* DO NOT EDIT THIS FILE - it is machine generated */
 #include <jni.h>
/* Header for class com_study_jnilearn_AccessCache */
 #ifndef _Included_com_study_jnilearn_AccessCache
 #define _Included_com_study_jnilearn_AccessCache
 #ifdef __cplusplus
extern "C" {
 #endif
/*
 * Class:     com_study_jnilearn_AccessCache
 * Method:    initIDs
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_com_study_jnilearn_AccessCache_initIDs
  (JNIEnv *, jclass);

/*
 * Class:     com_study_jnilearn_AccessCache
 * Method:    nativeMethod
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_com_study_jnilearn_AccessCache_nativeMethod
  (JNIEnv *, jobject);

 #ifdef __cplusplus
}
 #endif
 #endif
```

```
// AccessCache.c

 #include "com_study_jnilearn_AccessCache.h"

jmethodID MID_AccessCache_callback;

JNIEXPORT void JNICALL Java_com_study_jnilearn_AccessCache_initIDs
(JNIEnv *env, jclass cls)
{
    printf("initIDs called!!!\n");
    MID_AccessCache_callback = (*env)->GetMethodID(env,cls,"callback","()V");
}

JNIEXPORT void JNICALL Java_com_study_jnilearn_AccessCache_nativeMethod
(JNIEnv *env, jobject obj)
{
    printf("In C Java_com_study_jnilearn_AccessCache_nativeMethod called!!!\n");
```

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

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，调用 Java 的 callback 函数。

```
(*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：

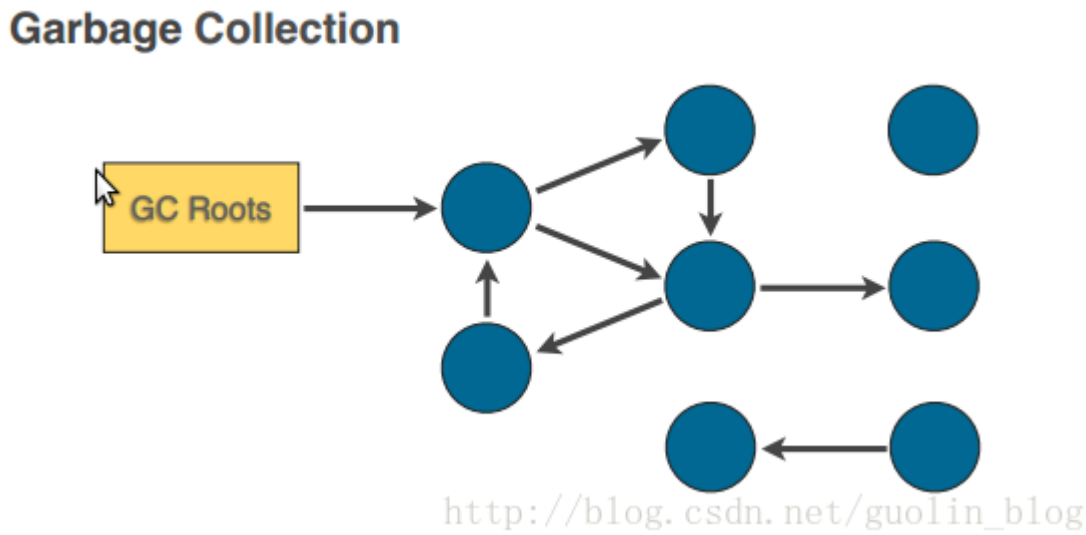
- 使用前，每次都需要检查是否已经缓存该 ID 或 Class 引用
- 如果在用使用时缓存的 ID，要注意只要本地代码依赖于这个 ID 的值，那么这个类就不会被 unload。另外一方面，如果缓存发生在静态初始化时，当类被 unload 或 reload 时，ID 会被重新计算。因为，尽量在类静态初始化时就缓存字段 ID、方法 ID 和类的 Class 引用。

11

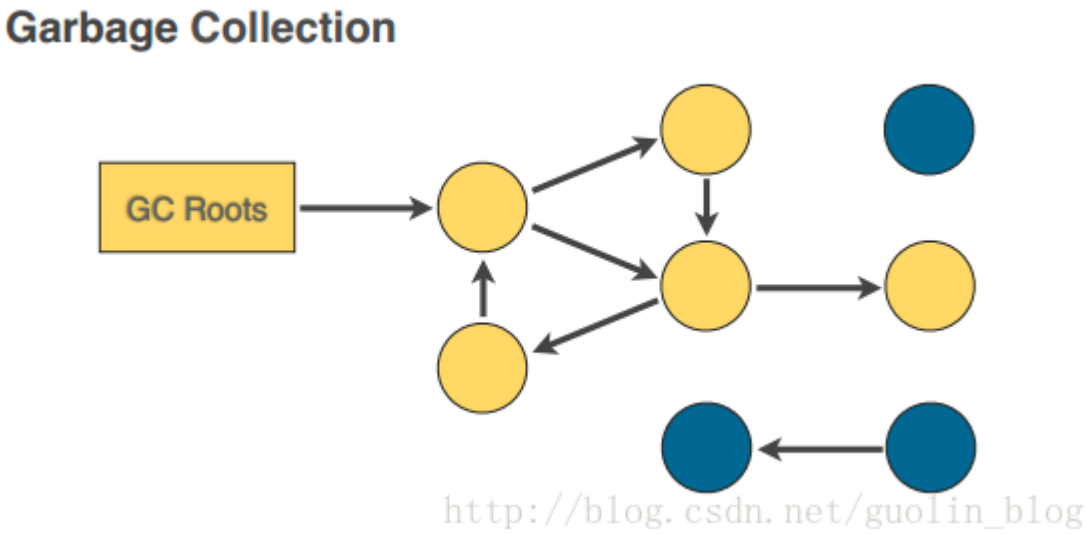
JNI 局部引用、全局引用和弱全局引用

这篇文章比较偏理论，详细介绍了在编写本地代码时三种引用的使用场景和注意事项。可能看起来有点枯燥，但引用是在 JNI 中最容易出错的一个点，如果使用不当，容易使程序造成内存溢出，程序崩溃等现象。《 Android JNI局部引用表溢出 》这篇文章是一个 JNI 引用使用不当造成引用表溢出，最终导致程序崩溃的例子。建议看完这篇文章之后，再去看。

做 Java 的朋友都知道，在编码的过程当中，内存管理这一块完全是透明的。new 一个类的实例时，只知道创建完这个类的实例之后，会返回这个实例的一个引用，然后就可以拿着这个引用访问它的所有数据成员了（ 属性、方法 ）。完全不用管 JVM 内部是怎么实现的，如何为新创建的对象来申请内存，也不用管对象使用完之后内存是怎么释放的，只需知道有一个垃圾回器在帮忙管理这些事情就 OK 的了。有经验的朋友也许知道启动一个 Java 程序，如果没有手动创建其它线程，默认会有两个线程在跑，一个是 main 线程，另一个就是 GC 线程（ 负责将一些不再使用的对象回收 ）。如果你曾经是做 Java 的然后转去做 C++，会感觉很不习惯，在 C++ 中 new 一个对象，使用完了还要做一次 delete 操作，malloc 一次同样也要调用 free 来释放相应的内存，否则你的程序就会有内存泄露了。而且在 C/C++ 中内存还分栈空间和堆空间，其中局部变量、函数形参变量、for 中定义的临时变量所分配的内存空间都是存放在栈空间（ 而且还要注意大小的限制 ），用 new 和 malloc 申请的内存都存放在堆空间。但 C/C++ 里的内存管理还远远不止这些，这些只是最基础的内存管理常识。做 Java 的人听到这些肯定会偷乐了，咱写 Java 的时候这些都不用管，全都交给 GC 就万事无优了。手动管理内存虽然麻烦，而且需要特别细心，一不小心就有可能造成内存泄露和野指针访问等程序致命的问题，但凡事都有利弊，手动申请和释放内存对程序的掌握比较灵活，不会受到平台的限制。比如我们写Android程序的时候，内存使用就受Dalivk虚拟机的限制，从最初版本的16~24M，到后来的 32M 到 64M，可能随着以后移动设备物理内存的不大扩大，后面的 Android 版本内存限制可能也会随着提高。但在 C/C++ 这层，就完全不受虚拟机的限制了。比如要在 Android 中要存储一张超高清的图片，刚好这张图片的大小超过了 Dalivk 虚拟机对每个应用的内存大小限制，Java 此时就显得无能为力了，但在C/C++ 看来就是小菜一碟了，malloc(1024*1024*50)。C/C++ 程序员得意的说道，Java 不是说是一门纯面象对象的语言吗，所以除了基本数据类型外，其它任何类型所创建的对象，JVM 所申请的内存都存在堆空间。上面提高到了 GC，是负责回收不再使用的对象，它的全称是 Garbage Collection，也就是所谓的垃圾回收。JVM 会在适当的时机触发 GC 操作，一旦进行 GC 操作，就会将一些不再使用的对象进行回收。那么哪些对象会被认为是不再使用，并且可以被回收的呢？我们来看下面二张图。（ 注：图摘自博主郭霖的《 Android 最佳性能实践(二)——分析内存的使用情况 》）



上图当中，每个蓝色的圆圈就代表一个内存当中的对象，而圆圈之间的箭头就是它们的引用关系。这些对象有些是处于活动状态的，而有些就已经不再被使用了。那么 GC 操作会从一个叫作 Roots 的对象开始检查，所有它可以访问到的对象就说明还在使用当中，应该进行保留，而其它的对象就表示已经不再被使用了，如下图所示：



可以看到，目前所有黄色的对象都处于活动状态，仍然会被系统继续保留，而蓝色的对象就会在 GC 操作当中被系统回收掉了，这就是 JVM 执行一次 GC 的简单流程。

上面说的废话好像有点多哈，下面进入正题。通过上面的讨论，大家都知道，如果一个 Java 对象没有被其它成员变量或静态变量所引用的话，就随时有可能会被 GC 回收掉。所以我们在编写本地代码时，要注意从 JVM 中获取到的引用在使用时被 GC 回收的可能性。由于本地代码不能直接通过引用操作 JVM 内部的数据结构，要进行这些操作必须调用相应的 JNI 接口来间接操作所引用的数据结构。JNI 提供了和 Java 相对应的引用类型，供本地代码配合 JNI 接口间接操作 JVM 内部的数据内容使用。如：jobject、jstring、jclass、jarray、jintArray

等。因为我们只通过 JNI 接口操作 JNI 提供的引用类型数据结构，而且每个 JVM 都实现了 JNI 规范相应的接口，所以我们不必担心特定 JVM 中对象的存储方式和内部数据结构等信息，我们只需要学习 JNI 中三种不同的引用即可。

由于 Java 程序运行在虚拟机中的这个特点，在 Java 中创建的对象、定义的变量和方法，内部对象的数据结构是怎么定义的，只有 JVM 自己知道。如果我们在 C/C++ 中想要访问 Java 中对象的属性和方法时，是不能够直接操作 JVM 内部 Java 对象的数据结构的。想要在 C/C++ 中正确的访问 Java 的数据结构，JVM 就必须有一套规则来约束 C/C++ 与 Java 互相访问的机制，所以才有了 JNI 规范，JNI 规范定义了一系列接口，任何实现了这套 JNI 接口的 Java 虚拟机，C/C++ 就可以通过调用这一系列接口来间接的访问 Java 中的数据结构。比如前面文章中学习到的常用 JNI 接口有：GetStringUTFChars（从 Java 虚拟机中获取一个字符串）、ReleaseStringUTFChars（释放从 JVM 中获取字符串所分配的内存空间）、NewStringUTF、GetArrayLength、GetFieldID、GetMethodID、FindClass 等。

三种引用简介及区别

在 JNI 规范中定义了三种引用：局部引用（Local Reference）、全局引用（Global Reference）、弱全局引用（Weak Global Reference）。区别如下：

局部引用

通过 NewLocalRef 和各种 JNI 接口创建（FindClass、NewObject、GetObjectClass和NewCharArray等）。会阻止 GC 回收所引用的对象，不在本地函数中跨函数使用，不能跨线前使用。函数返回后局部引用所引用的对象会被JVM 自动释放，或调用 DeleteLocalRef 释放。`(*env)->DeleteLocalRef(env,local_ref)`

```
jclass cls_string = (*env)->FindClass(env, "java/lang/String");
jcharArray charArr = (*env)->NewCharArray(env, len);
jstring str_obj = (*env)->NewObject(env, cls_string, cid_string, elemArray);
jstring str_obj_local_ref = (*env)->NewLocalRef(env,str_obj);  // 通过NewLocalRef函数创建
...
```

全局引用

调用 NewGlobalRef 基于局部引用创建，会阻 GC 回收所引用的对象。可以跨方法、跨线程使用。JVM 不会自动释放，必须调用 DeleteGlobalRef 手动释放。`(*env)->DeleteGlobalRef(env,g_cls_string)`

```
static jclass g_cls_string;
void TestFunc(JNIEnv* env, jobject obj) {
    jclass cls_string = (*env)->FindClass(env, "java/lang/String");
```

```
    g_cls_string = (*env)->NewGlobalRef(env,cls_string);
}
```

弱全局引用

调用 NewWeakGlobalRef 基于局部引用或全局引用创建，不会阻止 GC 回收所引用的对象，可以跨方法、跨线程使用。引用不会自动释放，在 JVM 认为应该回收它的时候（比如内存紧张的时候）进行回收而被释放。或调用DeleteWeakGlobalRef 手动释放。 `(*env)->DeleteWeakGlobalRef(env,g_cls_string)`

```
static jclass g_cls_string;
void TestFunc(JNIEnv* env, jobject obj) {
    jclass cls_string = (*env)->FindClass(env, "java/lang/String");
    g_cls_string = (*env)->NewWeakGlobalRef(env,cls_string);
}
```

局部引用

局部引用也称本地引用，通常是在函数中创建并使用。会阻止 GC 回收所引用的对象。比如，调用 NewObject 接口创建一个新的对象实例并返回一个对这个对象的局部引用。局部引用只有在创建它的本地方法返回前有效，本地方法返回到 Java 层之后，如果 Java 层没有对返回的局部引用使用的话，局部引用就会被 JVM 自动释放。你可能会为了提高程序的性能，在函数中将局部引用存储在静态变量中缓存起来，供下次调用时使用。这种方式是错误的，因为函数返回后局部引很可能马上就会被释放掉，静态变量中存储的就是一个被释放后的内存地址，成了一个野针对，下次再使用的时候就会造成非法地址的访问，使程序崩溃。请看下面一个例子，错误的缓存了 String 的 Class 引用。

```
/*错误的局部引用*/
JNIEXPORT jstring JNICALL Java_com_study_jnilearn_AccessCache_newString
(JNIEnv *env, jobject obj, jcharArray j_char_arr, jint len)
{
    jcharArray elemArray;
    jchar *chars = NULL;
    jstring j_str = NULL;
    static jclass cls_string = NULL;
    static jmethodID cid_string = NULL;
    // 注意：错误的引用缓存
    if (cls_string == NULL) {
        cls_string = (*env)->FindClass(env, "java/lang/String");
        if (cls_string == NULL) {
            return NULL;
        }
    }
```

```
    // 缓存String的构造方法ID
    if (cid_string == NULL) {
        cid_string = (*env)->GetMethodID(env, cls_string, "<init>", "([C)V");
        if (cid_string == NULL) {
            return NULL;
        }
    }

   //省略额外的代码.....
    elemArray = (*env)->NewCharArray(env, len);
    // ....
    j_str = (*env)->NewObject(env, cls_string, cid_string, elemArray);
    // 释放局部引用
    (*env)->DeleteLocalRef(env, elemArray);
    return j_str;
}
```

上面代码中，我们省略了和我们讨论无关的代码。因为 FindClass 返回一个对 java.lang.String 对象的局部引用，上面代码中缓存 cls_string 做法是错误的。假设一个本地方法 C.f 调用了 newString。

```
JNIEXPORT jstring JNICALL
 Java_C_f(JNIEnv *env, jobject this)
 {
    char *c_str = ...;
    ...
    return newString(c_str);
}
```

Java_com_study_jnilearn_AccessCache_newString 下面简称 newString。

C.f 方法返回后，JVM 会释放在这个方法执行期间创建的所有局部引用，也包含对 String 的 Class 引用cls_string。当再次调用 newString 时，newString 所指向引用的内存空间已经被释放，成为了一个野指针，再访问这个指针的引用时，会导致因非法的内存访问造成程序崩溃。

```
...
... = C.f(); // 第一次调是OK的
... = C.f(); // 第二次调用时，访问的是一个无效的引用.
...
```

释放局部引用

释放一个局部引用有两种方式，一个是本地方法执行完毕后 JVM 自动释放，另外一个是自己调用 DeleteLocalRef 手动释放。既然 JVM 会在函数返回后会自动释放所有局部引用，为什么还需要手动释放呢？大部分情况

下，我们在实现一个本地方法时不必担心局部引用的释放问题，函数被调用完成后，JVM 会自动释放函数中创建的所有局部引用。尽管如此，以下几种情况下，为了避免内存溢出，我们应该手动释放局部引用。

JNI 会将创建的局部引用都存储在一个局部引用表中，如果这个表超过了最大容量限制，就会造成局部引用表溢出，使程序崩溃。经测试，Android 上的 JNI 局部引用表最大数量是 512 个。当我们在实现一个本地方法时，可能需要创建大量的局部引用，如果没有及时释放，就有可能导致 JNI 局部引用表的溢出，所以，在不需要局部引用时就立即调用 DeleteLocalRef 手动删除。比如，在下面的代码中，本地代码遍历一个特别大的字符串数组，每遍历一个元素，都会创建一个局部引用，当对使用完这个元素的局部引用时，就应该马上手动释放它。

```
for (i = 0; i < len; i++) {
    jstring jstr = (*env)->GetObjectArrayElement(env, arr, i);
    ... /* 使用jstr */
    (*env)->DeleteLocalRef(env, jstr); // 使用完成之后马上释放
}
```

在编写 JNI 工具函数时，工具函数在程序当中是公用的，被谁调用你是不知道的。上面 newString 这个函数演示了怎么样在工具函数中使用完局部引用后，调用 DeleteLocalRef 删除。不这样做的话，每次调用 newString 之后，都会遗留两个引用占用空间（elemArray和cls_string，cls_string 不用 static 缓存的情况下）。

如果你的本地函数不会返回。比如一个接收消息的函数，里面有一个死循环，用于等待别人发送消息过来 `while(true) { if (有新的消息) { 处理之。。。。 } else { 等待新的消息。。。}}` 。如果在消息循环当中创建的引用你不显示删除，很快将会造成 JVM 局部引用表溢出。

局部引用会阻止所引用的对象被 GC 回收。比如你写的一个本地函数中刚开始需要访问一个大对象，因此一开始就创建了一个对这个对象的引用，但在函数返回前会有一个大量的非常复杂的计算过程，而在这个计算过程当中是不需要前面创建的那个大对象的引用的。但是，在计算的过程当中，如果这个大对象的引用还没有被释放的话，会阻止 GC 回收这个对象，内存一直占用者，造成资源的浪费。所以这种情况下，在进行复杂计算之前就应该把引用给释放了，以免不必要的资源浪费。

```
/* 假如这是一个本地方法实现 */
JNIEXPORT void JNICALL Java_pkg_Cls_func(JNIEnv *env, jobject this)
{
  lref = ...            /* lref引用的是一个大的Java对象 */
  ...                   /* 在这里已经处理完业务逻辑后，这个对象已经使用完了 */
  (*env)->DeleteLocalRef(env, lref); /* 及时删除这个对这个大对象的引用，GC就可以对它回收，并释放相应的资源*/
  lengthyComputation();  /* 在里有个比较耗时的计算过程 */
  return;               /* 计算完成之后，函数返回之前所有引用都已经释放 */
}
```

管理局部引用

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。JNI 规范指出，任何实现 JNI 规范的 JVM，必须确保每个本地函数至少可以创建 16 个局部引用（可以理解为虚拟机默认支持创建 16 个局部引用）。实际经验表明，这个数量已经满足大多数不需要和 JVM 中内部对象有太多交互的本地方函数。如果需要创建更多的引用，可以通过调用 EnsureLocalCapacity 函数，确保在当前线程中创建指定数量的局部引用，如果创建成功则返回 0，否则创建失败，并抛出 OutOfMemoryError 异常。EnsureLocalCapacity 这个函数是 1.2 以上版本才提供的，为了向下兼容，在编译的时候，如果申请创建的局部引用超过了本地引用的最大容量，在运行时 JVM 会调用 FatalError 函数使程序强制退出。在开发过程当中，可以为 JVM 添加 `-verbose:jni` 参数，在编译的时如果发现本地代码在试图申请过多的引用时，会打印警告信息提示我们要注意。在下面的代码中，遍历数组时会获取每个元素的引用，使用完了之后不手动删除，不考虑内存因素的情况下，它可以为这种创建大量的局部引用提供足够的空间。由于没有及时删除局部引用，因此在函数执行期间，会消耗更多的内存。

```
/*处理函数逻辑时，确保函数能创建len个局部引用*/
if((*env)->EnsureLocalCapacity(env,len) != 0) {
    ... /*申请len个局部引用的内存空间失败 OutOfMemoryError*/
    return;
}
for(i=0; i < len; i++) {
    jstring jstr = (*env)->GetObjectArrayElement(env, arr, i);
    // ... 使用jstr字符串
    /*这里没有删除在for中临时创建的局部引用*/
}
```

另外，除了 EnsureLocalCapacity 函数可以扩充指定容量的局部引用数量外，我们也可以利用 Push/PopLocalFrame 函数对创建作用范围层层嵌套的局部引用。例如，我们把上面那段处理字符串数组的代码用 Push/PopLocalFrame 函数对重写。

```
 #define N_REFS ... /*最大局部引用数量*/
for (i = 0; i < len; i++) {
    if ((*env)->PushLocalFrame(env, N_REFS) != 0) {
        ... /*内存溢出*/
    }
     jstring jstr = (*env)->GetObjectArrayElement(env, arr, i);
     ... /* 使用jstr */
     (*env)->PopLocalFrame(env, NULL);
}
```

PushLocalFrame 为当前函数中需要用到的局部引用创建了一个引用堆栈，（如果之前调用 PushLocalFrame 已经创建了 Frame，在当前的本地引用栈中仍然是有效的)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 负责销毁栈中所有的引用。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，而不需要时刻关注获取一个引用后，再调用 DeleteLocalRef 来释放引用。在上面的例子中，如果在处理 jstr 的过程当中又创建了局部引用，则 PopLocalFrame 执行时，这些局部引用将全都会被销毁。在调用 PopLocalFrame 销毁当前 frame 中的所有引用前，如果第二个参数 result 不为空，会由 result 生成一个新的局部引用，再把这个新生成的局部引用存储在上一个 frame 中。请看下面的示例。

```
// 函数原型
jobject (JNICALL *PopLocalFrame)(JNIEnv *env, jobject result);

jstring other_jstr;
for (i = 0; i < len; i++) {
    if ((*env)->PushLocalFrame(env, N_REFS) != 0) {
        ... /*内存溢出*/
    }
     jstring jstr = (*env)->GetObjectArrayElement(env, arr, i);
     ... /* 使用jstr */
     if (i == 2) {
        other_jstr = jstr;
     }
    other_jstr = (*env)->PopLocalFrame(env, other_jstr);  // 销毁局部引用栈前返回指定的引用
}
```

还要注意的一个问题是，局部引用不能跨线程使用，只在创建它的线程有效。不要试图在一个线程中创建局部引用并存储到全局引用中，然后在另外一个线程中使用。

全局引用

全局引用可以跨方法、跨线程使用，直到它被手动释放才会失效。同局部引用一样，也会阻止它所引用的对象被 GC 回收。与局部引用创建方式不同的是，只能通过 NewGlobalRef 函数创建。下面这个版本的 newString 演示怎么样使用一个全局引用。

```
JNIEXPORT jstring JNICALL Java_com_study_jnilearn_AccessCache_newString
(JNIEnv *env, jobject obj, jcharArray j_char_arr, jint len)
{
    // ...
    jstring jstr = NULL;
    static jclass cls_string = NULL;
    if (cls_string == NULL) {
```

```
    jclass local_cls_string = (*env)->FindClass(env, "java/lang/String");
    if (cls_string == NULL) {
        return NULL;
    }

    // 将java.lang.String类的Class引用缓存到全局引用当中
    cls_string = (*env)->NewGlobalRef(env, local_cls_string);

    // 删除局部引用
    (*env)->DeleteLocalRef(env, local_cls_string);

    // 再次验证全局引用是否创建成功
    if (cls_string == NULL) {
        return NULL;
    }
  }

  // ....
  return jstr;
}
```

弱全局引用

弱全局引用使用 NewGlobalWeakRef 创建，使用 DeleteGlobalWeakRef 释放。下面简称弱引用。与全局引用类似，弱引用可以跨方法、线程使用。但与全局引用很重要不同的一点是，弱引用不会阻止 GC 回收它引用的对象。在newString 这个函数中，我们也可以使用弱引用来存储 String 的 Class 引用，因为 java.lang.String 这个类是系统类，永远不会被 GC 回收。当本地代码中缓存的引用不一定要阻止 GC 回收它所指向的对象时，弱引用就是一个最好的选择。假设，一个本地方法 `mypkg.MyCls.f` 需要缓存一个指向类 `mypkg.MyCls2` 的引用，如果在弱引用中缓存的话，仍然允许 `mypkg.MyCls2` 这个类被 unload，因为弱引用不会阻止 GC 回收所引用的对象。请看下面的代码段。

```
JNIEXPORT void JNICALL
Java_mypkg_MyCls_f(JNIEnv *env, jobject self)
{
  static jclass myCls2 = NULL;
  if (myCls2 == NULL)
  {
    jclass myCls2Local = (*env)->FindClass(env, "mypkg/MyCls2");
    if (myCls2Local == NULL)
    {
      return; /* 没有找到mypkg/MyCls2这个类 */
    }
```

```
        myCls2 = NewWeakGlobalRef(env, myCls2Local);
        if (myCls2 == NULL)
        {
            return; /* 内存溢出 */
        }
    }
    ... /* 使用myCls2的引用 */
}
```

我们假设 MyCls 和 MyCls2 有相同的生命周期（例如，他们可能被相同的类加载器加载），因为弱引用的存在，我们不必担心 MyCls 和它所在的本地代码在被使用时，MyCls2 这个类出现先被 unload，后来又会 preload 的情况。当然，如果真的发生这种情况时（MyCls 和 MyCls2 此时的生命周期不同），我们在使用弱引用时，必须先检查缓存过的弱引用是指向活动的类对象，还是指向一个已经被 GC 给 unload 的类对象。下面马上告诉你怎样检查弱引用是否活动，即引用的比较。

引用比较

给定两个引用（不管是全局、局部还是弱全局引用），我们只需要调用 IsSameObject 来判断它们两个是否指向相同的对象。例如：`（*env)->IsSameObject(env, obj1, obj2)`，如果 obj1 和 obj2 指向相同的对象，则返回 JNI_TRUE（或者 1），否则返回 JNI_FALSE（或者 0）。有一个特殊的引用需要注意：NULL，JNI 中的 NULL 引用指向 JVM 中的 null 对象。如果 obj 是一个局部或全局引用，使用 `(*env)->IsSameObject(env, obj, NULL)` 或者 `obj == NULL` 来判断 obj 是否指向一个 null 对象即可。但需要注意的是，IsSameObject 用于弱全局引用与 NULL 比较时，返回值的意义是不同于局部引用和全局引用的。

```
jobject local_obj_ref = (*env)->NewObject(env, xxx_cls,xxx_mid);
jobject g_obj_ref = (*env)->NewWeakGlobalRef(env, local_ref);
// ... 业务逻辑处理
jboolean isEqual = (*env)->IsSameObject(env, g_obj_ref, NULL);
```

在上面的 IsSameObject 调用中，如果 g_obj_ref 指向的引用已经被回收，会返回 JNI_TRUE，如果 wobj 仍然指向一个活动对象，会返回 JNI_FALSE。

释放全局引用

每一个 JNI 引用被建立时，除了它所指向的 JVM 中对象的引用需要占用一定的内存空间外，引用本身也会消耗掉一个数量的内存空间。作为一个优秀的程序员，我们应该对程序在一个给定的时间段内使用的引用数量要十分小心。短时间内创建大量而没有被立即回收的引用很可能就会导致内存溢出。　　当我们的本地代码不再需要一个全局引用时，应该马上调用 DeleteGlobalRef 来释放它。如果不手动调用这个函数，即使这个对象已经没用了，JVM 也不会回收这个全局引用所指向的对象。　　同样，当我们的本地代码不再需要一个弱全局引用

时，也应该调用 DeleteWeakGlobalRef 来释放它，如果不手动调用这个函数来释放所指向的对象，JVM 仍会回收弱引用所指向的对象，但弱引用本身在引用表中所占的内存永远也不会被回收。

管理引用的规则

前面对三种引用已做了一个全面的介绍，下面来总结一下引用的管理规则和使用时的一些注意事项，使用好引用的目的就是为了减少内存使用和对象被引用保持而不能释放，造成内存浪费。所以在开发当中要特别小心！

通常情况下，有两种本地代码使用引用时要注意：

- 直接实现Java层声明的native函数的本地代码 当编写这类本地代码时，要当心不要造成全局引用和弱引用的累加，因为本地方法执行完毕后，这两种引用不会被自动释放。
- 被用在任何环境下的工具函数。例如：方法调用、属性访问和异常处理的工具函数等。

编写工具函数的本地代码时，要当心不要在函数的调用轨迹上遗漏任何的局部引用，因为工具函数被调用的场合和次数是不确定的，一量被大量调用，就很有可能造成内存溢出。所以在编写工具函数时，请遵守下面的规则：

- 一个返回值为基本类型的工具函数被调用时，它决不能造成局部、全局、弱全局引用被回收的累加。
- 当一个返回值为引用类型的工具函数被调用时，它除了返回的引用以外，它决不能造成其它局部、全局、弱引用的累加。

对于工具函数来说，为了使用缓存技术而创建一些全局引用或者弱全局引用是正常的。如果一个工具函数返回的是一个引用，我们应该写好注释详细说明返回引用的类型，以便于使用者更好的管理它们。下面的代码中，频繁地调用工具函数 GetInfoString，我们需要知道 GetInfoString 返回引用的类型是什么，以便于每次使用完成后调用相应的 JNI 函数来释放掉它。

```
while (JNI_TRUE) {
    jstring infoString = GetInfoString(info);
    ... /* 处理infoString */
    ??? /* 使用完成之后，调用DeleteLocalRef、DeleteGlobalRef、DeleteWeakGlobalRef哪一个函数来释放这个引用呢？ */
}
```

函数 NewLocalRef 有时被用来确保一个工具函数返回一个局部引用。我们改造一下 newString 这个函数，演示一下这个函数的用法。下面的 newString 是把一个被频繁调用的字符串 “CommonString” 缓存在了全局引用里。

```
JNIEXPORT jstring JNICALL Java_com_study_jnilearn_AccessCache_newString
{
    static jstring result;
    /* 使用wstrncmp函数比较两个Unicode字符串 */
```

```
    if (wstrncmp("CommonString", chars, len) == 0)
    {
        /* 将"CommonString"这个字符串缓存到全局引用中 */
        static jstring cachedString = NULL;
        if (cachedString == NULL)
        {
            /* 先创建"CommonString"这个字符串 */
            jstring cachedStringLocal = ...;
            /* 然后将这个字符串缓存到全局引用中 */
            cachedString = (*env)->NewGlobalRef(env, cachedStringLocal);
        }
        // 基于全局引用创建一个局引用返回，也同样会阻止GC回收所引用的这个对象，因为它们指向的是同一个对象
        return (*env)->NewLocalRef(env, cachedString);
    }
    ...
    return result;
}
```

在管理局部引用的生命周期中，Push/PopLocalFrame 是非常方便且安全的。我们可以在本地函数的入口处调用PushLocalFrame，然后在出口处调用 PopLocalFrame，这样的话，在函数内任何位置创建的局部引用都会被释放。而且，这两个函数是非常高效的，强烈建议使用它们。需要注意的是，如果在函数的入口处调用了PushLocalFrame，记住要在函数所有出口（有 return 语句出现的地方）都要调用 PopLocalFrame。在下面的代码中，对 PushLocalFrame 的调用只有一次，但调用 PopLocalFrame 确有多次，当然你也可以使用 goto 语句来统一处理。

```
jobject f(JNIEnv *env, ...)
{
    jobject result;
    if ((*env)->PushLocalFrame(env, 10) < 0)
    {
        /* 调用PushLocalFrame获取10个局部引用失败，不需要调用PopLocalFrame */
        return NULL;
    }
    ...
    result = ...; // 创建局部引用result
    if (...)
    {
        /* 返回前先弹出栈顶的frame */
        result = (*env)->PopLocalFrame(env, result);
        return result;
    }
    ...
    result = (*env)->PopLocalFrame(env, result);
    /* 正常返回 */
```

```
    return result;
}
```

上面的代码同样演示了函数 PopLocalFrame 的第二个参数的用法，局部引用 result 一开始在 PushLocalFrame 创建在当前 frame 里面，而把 result 传入 PopLocalFrame 中时，PopLocalFrame 在弹出当前的 frame 前，会由 result 生成一个新的局部引用，再将这个新生成的局部引用存储在上一个 frame 当中。

12

Android NDK 简介

NDK 产生的背景

Android 平台从诞生起，就已经支持 C、C++ 开发。众所周知，Android 的 SDK 基于 Java 实现，这意味着基于Android SDK 进行开发的第三方应用都必须使用 Java 语言。但这并不等同于“第三方应用只能使用 Java”。在Android SDK 首次发布时，Google 就宣称其虚拟机 Dalvik 支持 JNI 编程方式，也就是第三方应用完全可以通过 JNI 调用自己的 C 动态库，即在 Android 平台上，“Java+C”的编程方式是一直都可以实现的。

不过，Google 也表示，使用原生 SDK 编程相比 Dalvik 虚拟机也有一些劣势，Android SDK 文档里，找不到任何 JNI 方面的帮助。即使第三方应用开发者使用 JNI 完成了自己的 C 动态链接库（so）开发，但是 so 如何和应用程序一起打包成 apk 并发布？这里面也存在技术障碍。比如程序更加复杂，兼容性难以保障，无法访问Framework API，Debug 难度更大等。开发者需要自行斟酌使用。

于是 NDK 就应运而生了。NDK 全称是 Native Development Kit。

NDK 的发布，使“Java+C”的开发方式终于转正，成为官方支持的开发方式。NDK 将是 Android 平台支持 C 开发的开端。

为什么使用 NDK

- 代码的保护。由于 apk 的 java 层代码很容易被反编译，而 C/C++ 库反汇难度较大。
- 可以方便地使用现存的开源库。大部分现存的开源库都是用 C/C++ 代码编写的。
- 提高程序的执行效率。将要求高性能的应用逻辑使用 C 开发，从而提高应用程序的执行效率。
- 便于移植。用 C/C++ 写得库可以方便在其他的嵌入式平台上再次使用。

NDK 简介

NDK 是一系列工具的集合

NDK 提供了一系列的工具，帮助开发者快速开发 C（或C++）的动态库，并能自动将 so 和 java 应用一起打包成 apk。这些工具对开发者的帮助是巨大的。

NDK 集成了交叉编译器，并提供了相应的 mk 文件隔离 CPU、平台、ABI 等差异，开发人员只需要简单修改 mk 文件（指出“哪些文件需要编译”、“编译特性要求”等），就可以创建出 so。

NDK 可以自动地将 so 和 Java 应用一起打包，极大地减轻了开发人员的打包工作。

NDK 提供了一份稳定、功能有限的 API 头文件声明

Google 明确声明该 API 是稳定的，在后续所有版本中都稳定支持当前发布的 API。从该版本的 NDK 中看出，这些 API 支持的功能非常有限，包含有：C 标准库（libc）、标准数学库（libm）、压缩库（libz）、Log 库（liblog）。

13

Android NDK 开发环境

NDK 开发环境的搭建

下载安装 Android NDK

地址：http://blog.sina.com.cn/s/blog_718f290a01012ch0.html

下载安装 cygwin

由于 NDK 编译代码时必须要用到 make 和 gcc，所以你必须先搭建一个 linux 环境， cygwin 是一个在windows 平台上运行的 unix 模拟环境,它对于学习 unix/linux 操作环境，或者从 unix 到 Windows 的应用程序移植，非常有用。通过它，你就可以在不安装 linux 的情况下使用 NDK 来编译 C、C++ 代码了。下载地址：http://www.cygwin.com

- 然后双击运行吧，运行后你将看到安装向导界面。
- 点击下一步,此时让你选择安装方式：Install from Internet：直接从 Internet 上下载并立即安装（安装完成后，下载好的安装文件并不会被删除，而是仍然被保留，以便下次再安装）。Download Without Installing：只是将安装文件下载到本地，但暂时不安装。Install from Local Directory：不下载安装文件，直接从本地某个含有安装文件的目录进行安装。
- 选择第一项，然后点击下一步。
- 选择要安装的目录，注意，最好不要放到有中文和空格的目录里，似乎会造成安装出问题，其它选项不用变，之后点下一步。
- 上一步是选择安装cygwin的目录，这个是选择你下载的安装包所在的目录，默认是你运行setup.exe的目录，直接点下一步就可以。
- 此时你共有三种连接方式选择：Direct Connection：直接连接。Use IE5 Settings：使用IE的连接参数设置进行连接。Use HTTP/FTP Proxy：使用HTTP或FTP代理服务器进行连接（需要输入服务器地址、端口号）。 用户可根据自己的网络连接的实情情况进行选择，一般正常情况下，均选择第一种，也就是直接连接方式。然后再点击“下一步”。
- 这是选择要下载的站点，选择后点下一步。
- 此时会下载加载安装包列表
- Search 是可以输入你要下载的包的名称，能够快速筛选出你要下载的包。那四个单选按钮是选择下边树的样式，默认就行，不用动。View 默认是 Category，建议改成 full 显示全部包再查，省的一些包被隐藏掉。左下角那个复选框是是否隐藏过期包，默认打钩，不用管它就行，下边开始下载我们要安装的包吧，为

了避免全部下载，这里列出了后面开发NDK用得着的包：autoconf2.1、automake1.10、binutils、gcc-core、gcc- g++、gcc4-core、gcc4-g++、gdb、pcre、pcre-devel、gawk、make共12个包

- 然后开始选择安装这些包吧，点 skip，把它变成数字版本格式，要确保 Bin 项变成叉号，而 Src 项是源码，这个就没必要选了。 -下面测试一下 cygwin 是不是已经安装好了。

运行 cygwin，在弹出的命令行窗口输入：cygcheck -c cygwin 命令，会打印出当前 cygwin 的版本和运行状态，如果 status 是 ok 的话，则 cygwin 运行正常。

然后依次输入 gcc –version，g++ --version，make –version，gdb –version 进行测试，如果都打印出版本信息和一些描述信息，则 cygwin 安装成功!

配置 NDK 环境变量

- 首先找到 cygwin 的安装目录，找到一个 home\< 你的用户名 >.bash_profile 文件，我的是：E:\cygwin\home\Administrator.bash_profile ，（注意：我安装的时候我的 home 文件夹下面什么都没有，解决 的办法：首先打开环境变量，把里面的用户变量中的 HOME 变量删掉，在 E:\cygwin\home 文件夹下建立名为 Administrator 的文件夹（是用户名），然后把 E:\cygwin\etc\skel.bash_profile 拷贝到该文件夹下）。
- 打开 bash_profile 文件，添加 `NDK=/cygdrive/< 你的盘符 >/<android ndk 目录 >` 例如：

```
NDK=/cygdrive/e/android-ndk-r5
```

```
export NDK
```

NDK 这个名字是随便取的，为了方面以后使用方便，选个简短的名字，然后保存。

- 打开 cygwin ，输入 cd $NDK ，如果输出上面配置的 `/cygdrive/e/android-ndk-r5` 信息，则表明环境变量设置成功了。

用 NDK 来编译程序

- 现在我们用安装好的 NDK 来编译一个简单的程序吧，我们选择 ndk 自带的例子 hello-jni ，我的位于E:\android-ndk-r5\samples\hello-jni(根据你具体的安装位置而定)，
- 运行 cygwin ，输入命令 cd /cygdrive/e/android-ndk-r5/samples/hello-jni ，进入到 E:\android-ndk-r5\samples\hello-jni 目录。
- 输入 $NDK/ndk-build ，执行成功后，它会自动生成一个 libs 目录，把编译生成的 .so 文件放在里面。($NDK是调用我们之前配置好的环境变量， ndk-build 是调用 ndk 的编译程序)

- 此时去 hello-jni 的 libs 目录下看有没有生成的 .so 文件，如果有，你的 ndk 就运行正常啦!

在 eclipse 中集成 c/c++ 开发环境

- 装 Eclipse 的 C/C++ 环境插件： CDT ，这里选择在线安装。首先登录 http://www.eclipse.org/cdt/downloads.php ，找到对应你 Eclipse 版本的 CDT 插件 的在线安装地址。
- 然后点 Help 菜单，找到 Install New Software 菜单
- 点击 Add 按钮，把取的地址填进去，出来插件列表后，选 Select All ，然后选择下一步即可完成安装。
- 安装完成后，在 eclispe 中右击新建一个项目，如果出现了 c/c++ 项目，则表明你的 CDT 插件安装成功啦!

配置 C/C++ 的编译器

- 打开 eclipse ，导入ndk 自带的hello-jni 例子，右键单击项目名称，点击 Properties ，弹出配置界面，之后再点击 Builders ，弹出项目的编译工具列表，之后点击 New，新添加一个编译器，点击后出现添加界面，选择 Program ，点击 OK。
- 出现了添加界面，首先给编译配置起个名字，如： C_Builder，设置 Location 为 `< 你 cygwin 安装路径 >\bin\bash.exe` 程序，例： `E:\cygwin\bin\bash.exe` ，设置 Working Directory 为 `<你 cygwin 安装路径 >\bin` 目录 ，例如： `E:\cygwin\bin` ，设置 Arguments 为 `--login -c "cd /cygdrive/e/android-ndk-r5/samples/hello-jni && $NDK /ndk-build"` ，上面的配置中 `/cygdrive/e/android-ndk-r5/samples/hello-jni` 是你当前要编译的程序的目录， $NDK 是之前配置的 ndk 的环境变量，这两个根据你具体的安装目录进行配置，其他的不用变， Arguments 这串参数实际是给 bash.exe 命令行程序传参数，进入要编译的程序目录，然后运行 ndk-build 编译程序。
- 接着切换到 Refresh 选项卡，给 Refresh resources upon completion 打上钩。
- 然后切换到 Build Options 选项卡，勾选上最后三项。
- 之后点击 Specify Resources 按钮，选择资源目录，勾选你的项目目录即可。
- 最后点击 Finish，点击 OK 一路把刚才的配置都保存下来，注意：如果你配置的编译器在其它编译器下边，记得一定要点 Up 按钮，把它排到第一位，否则 C 代码的编译晚于Java代码的编译，会造成你的 C 代码要编译两次才能看到最新的修改。

- 编译配置也配置完成啦，现在来测试一下是否可以自动编译呢，打开项目 jni 目录里的 hello-jni.c 文件把提示 Hello from JNI! 改成其他的文字：如： Hello ， My name is alex. ，然后再模 拟器中运行你的程序，如果模拟器中显示了你最新修改的文字，那么 Congratulations ！ 你已经全部配置成功啦!

14

开发自己的 NDK 程序

入门的最好办法就是学习 Android 自带的例子， 这里就通过学习 Android 的 NDK 自带的 demo 程序：hello-jni来达到这个目的。

开发环境的搭建

- android 的 NDK 开发需要在 linux 下进行： 因为需要把 C/C++ 编写的代码生成能在 arm 上运行的 .so 文件，这就需要用到交叉编译环境，而交叉编译需要在 linux 系统下才能完成。
- 安装 android-ndk 开发包，这个开发包可以在 google android 官网下载： 通过这个开发包的工具才能将android jni 的 C/C++ 的代码编译成库
- android 应用程序开发环境： 包括 eclipse、java、android sdk、adt 等。

如何下载和安装 android-ndk 我这里就不啰嗦了，安装完之后，需要将 android-ndk 的路劲加到环境变量 PATH 中：

```
sudo gedit /etc/environment
```

在 environment 的 PATH 环境变量中添加你的 android-ndk 的安装路劲，然后再让这个更改的环境变量立即生效：

```
source  /etc/environment
```

经过了上述步骤，在命令行下敲：

```
ndk-bulid
```

弹出如下的错误，而不是说 ndk-build not found，就说明 ndk 环境已经安装成功了。

```
Android NDK: Could not find application project directory !
Android NDK: Please define the NDK_PROJECT_PATH variable to point to it.
/home/braincol/workspace/android/android-ndk-r5/build/core/build-local.mk:85: *** Android NDK: Aborting    . Stop.
```

代码的编写

编写 Java 代码

建立一个 Android 应用工程 HelloJni，创建 HelloJni.java 文件：

HelloJni.java :

```
import android.app.Activity;
import android.widget.TextView;
import android.os.Bundle;

public class HelloJni extends Activity
{
   /** Called when the activity is first created. */
   @Override
   public void onCreate(Bundle savedInstanceState)
   {
      super.onCreate(savedInstanceState);

      TextView  tv = new TextView(this);
      tv.setText( stringFromJNI() );
      setContentView(tv);
   }


   /* A native method that is implemented by the 'hello-jni' native library, which is packaged with this application. */
   public native String  stringFromJNI();

   public native String  unimplementedStringFromJNI();



   /* this is used to load the 'hello-jni' library on application startup. The library has already been unpacked into

     /data/data/com.example.HelloJni/lib/libhello-jni.so at installation time by the package manager. */
   static {
      System.loadLibrary("hello-jni");
   }
}
```

代码解释：

```
static{
System.loadLibrary("hello-jni");
}
```

表明程序开始运行的时候会加载 hello-jni, static 区声明的代码会先于 onCreate 方法执行。如果你的程序中有多个类，而且如果 HelloJni 这个类不是你应用程序的入口，那么 hello-jni（完整的名字是 libhello-jni.so）这个库会在第一次使用 HelloJni 这个类的时候加载。

```
public native String stringFromJNI();
public native String unimplementedStringFromJNI();
```

可以看到这两个方法的声明中有 native 关键字， 这个关键字表示这两个方法是本地方法，也就是说这两个方法是通过本地代码（ C/C++ ）实现的，在java代码中仅仅是声明。

用 eclipse 编译该工程，生成相应的 .class 文件，这步必须在下一步之前完成，因为生成 .h 文件需要用到相应的 .class 文件。

编写相应的 C/C++ 代码

刚开始学的时候，有个问题会让人很困惑，相应的 C/C++ 代码如何编写，函数名如何定义？ 这里讲一个方法，利用 javah 这个工具生成相应的 .h 文件，然后根据这个 .h 文件编写相应的 C/C++ 代码。

1.生成相应 .h 文件：

就拿我这的环境来说，首先在终端下进入刚刚建立的 HelloJni 工程的目录：

```
braincol@ubuntu:~$ cd workspace/android/NDK/hello-jni/
```

ls 查看工程文件

```
braincol@ubuntu:~/workspace/android/NDK/hello-jni$ ls

AndroidManifest.xml  assets  bin  default.properties  gen  res  src
```

可以看到目前仅仅有几个标准的 android 应用程序的文件（夹）。

首先我们在工程目录下建立一个 jni 文件夹：

```
braincol@ubuntu:~/workspace/android/NDK/hello-jni$ mkdir jni

braincol@ubuntu:~/workspace/android/NDK/hello-jni$ ls

AndroidManifest.xml  assets  bin  default.properties  gen  jni  res  src
```

下面就可以生成相应的 .h 文件了：

```
braincol@ubuntu:~/workspace/android/NDK/hello-jni$ javah -classpath bin -d jni com.example.hellojni.HelloJni
```

-classpath bin：表示类的路劲

-d jni: 表示生成的头文件存放的目录

com.example.hellojni.HelloJni 则是完整类名

这一步的成功要建立在已经在 bin/com/example/hellojni/ 目录下生成了 HelloJni.class 的基础之上。现在可以看到 jni 目录下多了个 .h 文件：

```
braincol@ubuntu:~/workspace/android/NDK/hello-jni$ cd jni/

braincol@ubuntu:~/workspace/android/NDK/hello-jni/jni$ ls

com_example_hellojni_HelloJni.h
```

我们来看看 com_example_hellojni_HelloJni.h 的内容：

com_example_hellojni_HelloJni.h :

```
/* DO NOT EDIT THIS FILE - it is machine generated */

 #include <jni.h>

/* Header for class com_example_hellojni_HelloJni */

 #ifndef _Included_com_example_hellojni_HelloJni

 #define _Included_com_example_hellojni_HelloJni

 #ifdef __cplusplus

extern "C" {

 #endif

/*

 * Class:     com_example_hellojni_HelloJni

 * Method:    stringFromJNI

 * Signature: ()Ljava/lang/String;

 */

JNIEXPORT jstring JNICALL Java_com_example_hellojni_HelloJni_stringFromJNI

  (JNIEnv *, jobject);


/*
```

```
 * Class:     com_example_hellojni_HelloJni
 * Method:    unimplementedStringFromJNI
 * Signature: ()Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_com_example_hellojni_HelloJni_unimplementedStringFromJNI
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

上面代码中的 JNIEXPORT 和 JNICALL 是 jni 的宏，在 android 的 jni 中不需要，当然写上去也不会有错。从上面的源码中可以看出这个函数名那是相当的长啊。不过还是很有规律的，完全按照：java_pacakege_class_mathod 形式来命名。

也就是说：

Hello.java 中 stringFromJNI() 方法对应于 C/C++中的 Java_com_example_hellojni_HelloJni_stringFromJNI() 方法

HelloJni.java 中的 unimplementedStringFromJNI() 方法对应于 C/C++中的 Java_com_example_hellojni_HelloJni_unimplementedStringFromJNI() 方法

注意下其中的注释：

Signature: ()Ljava/lang/String;

()Ljava/lang/String;()表示函数的参数为空（这里为空是指除了JNIEnv *, jobject 这两个参数之外没有其他参数，JNIEnv*, jobject 是所有 jni 函数必有的两个参数，分别表示 jni 环境和对应的 java 类（或对象）本身），Ljava/lang/String; 表示函数的返回值是 java 的 String 对象。

编写相应的 .c 文件

hello-jni.c :

```
#include <string.h>
#include <jni.h>


/* This is a trivial JNI example where we use a native method

 * to return a new VM String. See the corresponding Java source

 * file located at:

 *   apps/samples/hello-jni/project/src/com/example/HelloJni/HelloJni.java

 */

jstring Java_com_example_hellojni_HelloJni_stringFromJNI( JNIEnv* env, jobject thiz )
{
   return (*env)->NewStringUTF(env, "Hello from JNI !");
}
```

这里只是实现了 Java_com_example_hellojni_HelloJni_stringFromJNI 方法，而 Java_com_example_hellojni_HelloJni_unimplementedStringFromJNI 方法并没有实现，因为在HelloJni.java 中只调用了 stringFromJNI()方法，所以 unimplementedStringFromJNI()方法没有实现也没关系，不过建议最好还是把所有 java 中定义的本地方法都实现了，写个空函数也行。有总比没有好。

Java_com_example_hellojni_HelloJni_stringFromJNI() 函数只是简单的返回了一个内容为 "Hello from JNI !" 的 jstring 对象（对应于 Java 中的 String 对象）。hello-jni.c 文件已经编写好了，现在可以把 com_example_hellojni_HelloJni.h 文件给删了，当然留着也行，只是习惯是把不需要的文件给清理干净了。

编译 hello-jni.c 生成相应的库

编写 Android.mk 文件

在 jni 目录下（即 hello-jni.c 同级目录下）新建一个 Android.mk 文件， Android.mk 文件是 Android 的 make file 文件，内容如下：

```
# Copyright (C) 2009 The Android Open Source Project
```

```
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#      http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.
#
LOCAL_PATH := $(call my-dir)


include $(CLEAR_VARS)


LOCAL_MODULE    := hello-jni
LOCAL_SRC_FILES := hello-jni.c


include $(BUILD_SHARED_LIBRARY)
```

这个 Androd.mk 文件很短，下面我们来逐行解释下：

```
LOCAL_PATH := $(call my-dir)
```

一个 Android.mk 文件首先必须定义好 LOCAL_PATH 变量。它用于在开发树中查找源文件。在这个例子中，宏函数 ’my-dir’ , 由编译系统提供，用于返回当前路径（即包含 Android.mk file 文件的目录）。

```
include $( CLEAR_VARS)
```

CLEAR_VARS 由编译系统提供，指定让 GNU MAKEFILE 为你清除许多 LOCAL_XXX 变量（例如 LOCAL_MODULE, LOCAL_SRC_FILES, LOCAL_STATIC_LIBRARIES, 等等...), 除 LOCAL_PATH 。这是必要的，因为所有的编译控制文件都在同一个 GNU MAKE 执行环境中，所有的变量都是全局的。

```
LOCAL_MODULE := hello-jni
```

编译的目标对象，LOCAL_MODULE 变量必须定义，以标识你在 Android.mk 文件中描述的每个模块。名称必须是唯一的，而且不包含任何空格。

注意：编译系统会自动产生合适的前缀和后缀，换句话说，一个被命名为 hello-jni 的共享库模块，将会生成 libhello-jni.so 文件。

重要注意事项：如果你把库命名为 libhello-jni ，编译系统将不会添加任何的 lib 前缀，也会生成 libhello-jni.so ，这是为了支持来源于 Android 平台的源代码的 Android.mk 文件，如果你确实需要这么做的话。

```
LOCAL_SRC_FILES := hello-jni.c
```

LOCAL_SRC_FILES 变量必须包含将要编译打包进模块中的 C 或 C++ 源代码文件。注意，你不用在这里列出头文件和包含文件，因为编译系统将会自动为你找出依赖型的文件；仅仅列出直接传递给编译器的源代码文件就好。

注意，默认的 C++ 源码文件的扩展名是 .cpp 指定一个不同的扩展名也是可能的，只要定义LOCAL_DEFAULT_CPP_EXTENSION 变量，不要忘记开始的小圆点（也就是 .cxx ,而不是 cxx ）

```
include $(BUILD_SHARED_LIBRARY)
```

BUILD_SHARED_LIBRARY 表示编译生成共享库，是编译系统提供的变量，指向一个 GNU Makefile 脚本，负责收集自从上次调用 include $(CLEAR_VARS) 以来，定义在 LOCAL_XXX 变量中的所有信息，并且决定编译什么，如何正确地去做。还有 BUILD_STATIC_LIBRARY 变量表示生成静态库： lib$(LOCAL_MODULE).a ， BUILD_EXECUTABLE 表示生成可执行文件。

生成 .so 共享库文件

Andro 文件已经编写好了，现在可以用 android NDK 开发包中的 ndk-build 脚本生成对应的 .so 共享库了，方法如下：

```
braincol@ubuntu:~/workspace/android/NDK/hello-jni/jni$ cd ..
braincol@ubuntu:~/workspace/android/NDK/hello-jni$ ls
AndroidManifest.xml  assets  bin  default.properties  gen  jni  libs  obj  res  src
braincol@ubuntu:~/workspace/android/NDK/hello-jni$ ndk-build
Gdbserver      : [arm-linux-androideabi-4.4.3] libs/armeabi/gdbserver
Gdbsetup       : libs/armeabi/gdb.setup
Install        : libhello-jni.so => libs/armeabi/libhello-jni.so
```

可以看到已经正确的生成了 `libhello-jni.so` 共享库了， 我们去 `libs/armeabi/` 目录下看看：

```
braincol@ubuntu:~/workspace/android/NDK/hello-jni$ cd libs/
braincol@ubuntu:~/workspace/android/NDK/hello-jni/libs$ ls
armeabi
braincol@ubuntu:~/workspace/android/NDK/hello-jni/libs$ cd armeabi/
braincol@ubuntu:~/workspace/android/NDK/hello-jni/libs/armeabi$ ls
gdbserver  gdb.setup  libhello-jni.so
```

在 eclipse 重新编译 HelloJni 工程，生成 apk

eclipse 中刷新下 HelloJni 工程，重新编译生成 apk，libhello-jni.so 共享库会一起打包在 apk 文件内。在模拟器中看看运行结果。