<https://blog.csdn.net/promaster/article/details/70318695>

在Java中，要使用动态库，就要使用到 JNI。首先来看看百度百科对JNI的描述：从Java1.1开始，Java Native Interface(JNI)标准成为Java平台的一部分，它允许Java代码和其他语言写的代码进行交互。JNI一开始是为了本地已编译语言，尤其是C和C++而设计的，但是它并不妨碍你使用其他语言，只要调用约定受支持就可以了。   
使用java与本地已编译的代码交互，通常会丧失平台可移植性。但是，有些情况下这样做是可以接受的，甚至是必须的。例如，使用一些旧的库，与硬件、操作系统进行交互，或者为了提高程序的性能。JNI标准至少保证本地代码能工作在任何Java虚拟机环境下。闲话不说，直接上示例。

本例基于Windows平台，首先生成被调用的目的dll动态库：Lib4JNI.dll   
Lib4JNI动态库中导出了一个add方法。在本例中用JNI去调用。

// dllmain.cpp : Defines the entry point for the DLL application.

#include "stdafx.h"

#include "stdlib.h"

#include "stdio.h"

BOOL APIENTRY DllMain( HMODULE hModule,

DWORD ul\_reason\_for\_call,

LPVOID lpReserved

)

{

switch (ul\_reason\_for\_call)

{

case DLL\_PROCESS\_ATTACH:

case DLL\_THREAD\_ATTACH:

case DLL\_THREAD\_DETACH:

case DLL\_PROCESS\_DETACH:

break;

}

return TRUE;

}

extern "C"

int \_declspec(dllexport) add(int a, int b)

{

int iResult = a + b;

printf("%d + %d = ", a, b);

return iResult;

}

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

接下来要做的是，使用中介（或者叫代理）dll调用目的动态库（目的动态库就是本例中的Lib4JNI.dll）。在Java中，是不可以直接调用目的动态库的。因此，需要有一个中介（或代理），由本地的Java代码先调用这个中介（或代理）动态库，再由这个中介（或代理）动态库调用目的动态库。   
我们给中介dll起个名字，叫做Lib2Invoke。生成这个中介dll的过程比较复杂，下面，我们用分解动作，详细说明。

第一步：需要创建本地Java的调用代码。

public class TestJNI {

public native int getNumber(int a, int b); //声明Native方法

public static void main(String[] args) {

}

}

* 1
* 2
* 3
* 4
* 5
* 6

将以上代码，以文本形式，保存为TestJNI.java。

第二步：用javac编译。javac TestJNI.java

第三步：用javah生成中介（或代理）动态库的头文件。javah TestJNI，运行后，生成TestJNI.h文件，本例中生成的文件示例如下：

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

#include <jni.h>

/\* Header for class TestJNI \*/

#ifndef \_Included\_TestJNI

#define \_Included\_TestJNI

#ifdef \_\_cplusplus

extern "C" {

#endif

/\*

\* Class: TestJNI

\* Method: getNumber

\* Signature: (II)I

\*/

JNIEXPORT jint JNICALL Java\_TestJNI\_getNumber

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

#ifdef \_\_cplusplus

}

#endif

#endif

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

第四步：生成中介（或代理）动态库   
为什么到这里才开始生成中介dll呢？因为，我们需要上一步生成的TestJNI.h，作为中介dll的头文件。详细请参考代码。

// dllmain.cpp : Defines the entry point for the DLL application.

#include "stdafx.h"

#include "jni.h" //在这里，我们要注意的是，需要引用

#include "TestJNI.h"

#ifdef WIN32

#ifdef \_X86\_

#define \_T(x) x

#else

#ifdef \_AMD64\_

#define \_T(x) L ## x

#endif

#endif

#endif

BOOL APIENTRY DllMain( HMODULE hModule,

DWORD ul\_reason\_for\_call,

LPVOID lpReserved

)

{

switch (ul\_reason\_for\_call)

{

case DLL\_PROCESS\_ATTACH:

case DLL\_THREAD\_ATTACH:

case DLL\_THREAD\_DETACH:

case DLL\_PROCESS\_DETACH:

break;

}

return TRUE;

}

JNIEXPORT jint JNICALL Java\_TestJNI\_getNumber

(JNIEnv \* env, jobject o, jint x, jint y)

{

typedef int(\*ADD)(int, int);//函数指针类型

HINSTANCE Hint = ::LoadLibrary(\_T("Lib4JNI.dll"));//加载我们刚才生成的dll

ADD add = (ADD)GetProcAddress(Hint, "add");//取得dll导出的add方法

return add(x, y);

FreeLibrary(Hint);

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43

编译完成后，生成了我们需要的中介（或代理）动态库Lib2Invoke.dll。

第五步： 完善本地Java调用代码。

public class TestJNI {

public native int getNumber(int a, int b);

public static void main(String[] args) {

System.loadLibrary("Lib2Invoke");

TestJNI p = new TestJNI();

System.out.println(p.getNumber(100, 100));

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

编译完成后，把生成的class文件，和Lib4JNI.dll、Lib2Invoke.dll放在一个目录下。

第六步：执行 java TestJNI   
执行成功后即会输出结果。

另一种可以选择的方法JNative。以程序员特有的本质，少说话，多代码：

import org.xvolks.jnative.JNative;

import org.xvolks.jnative.Type;

import org.xvolks.jnative.exceptions.NativeException;

public class TestJNI {

static JNative myjnative = null;

public int getnumber(int a, int b) throws NativeException,

IllegalAccessException {

try {

if (myjnative == null) {

myjnative = new JNative("Lib4JNI.dll", "add");

myjnative.setRetVal(Type.INT);

}

myjnative.setParameter(0, a);

myjnative.setParameter(1, b);

myjnative.invoke();

return myjnative.getRetValAsInt();

} finally {

if (myjnative != null) {

myjnative.dispose();

}

}

}

public static void main(String[] args) throws NativeException, IllegalAccessException {

test uc = new test();

int result = uc.getnumber(1,100);

System.err.println("result:" + result);

}

}

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

优点是：不用中介dll，不用生成.h中介头文件   
缺点是：（缺点不准确，只是为了对齐优点）需要外部包支持。

再来说说Linux平台下JNI的使用方法。基本步骤同上。下面给出主要示例代码。   
1、Java调用部分

NativeAgent.java

public class NativeAgent {

static

{

try {

System.loadLibrary("NativeAgent");

}

catch(UnsatisfiedLinkError e) {

System.err.println(">>> Can not load library: " + e.toString());

}

}

public native int toConsole(String s);

public static void main(String[] args) {

NativeAgent na = new NativeAgent();

na.toConsole("This is a JNI Project test.\n");

}

}

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

2、生成动态库部分

NativeAgent.h

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

#include <jni.h>

/\* Header for class NativeAgent \*/

#ifndef \_Included\_NativeAgent

#define \_Included\_NativeAgent

#ifdef \_\_cplusplus

extern "C" {

#endif

/\*

\* Class: NativeAgent

\* Method: toConsole

\* Signature: (Ljava/lang/String;)I

\*/

JNIEXPORT jint JNICALL Java\_NativeAgent\_toConsole

(JNIEnv \*, jobject, jstring);

#ifdef \_\_cplusplus

}

#endif

#endif

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

NativeAgent.c

#include <stdio.h>

#include <stdlib.h>

#include <jni.h>

#include "NativeAgent.h"

JNIEXPORT jint JNICALL Java\_NativeAgent\_toConsole(JNIEnv \*pEnv, jobject obj, jstring str) {

const char\* msg = (\*pEnv)->GetStringUTFChars(pEnv, str, 0);

printf("%s", msg);

return 0;

}

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

3、编译生成动态库

javac NativeAgent.java

javah NativeAgent

gcc -o libNativeAgent.so -I/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.102-1.b14.el7\_2.x86\_64/include -I/usr/lib/jvm/java-1.8.0-openjdk-1.8.0.102-1.b14.el7\_2.x86\_64/include/linux -I. -fPIC -shared NativeAgent.c

java -Djava.library.path=. NativeAgent

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

这里有二个非常重要的地方，要十分注意：   
1、对于System.loadLibrary("NativeAgent");   
在Linux下，动态库输出的文件名要是libNativeAgent.so。   
也就是说，如果System.loadLibrary("XXX")；那么，在导出动态库时，动态库的名字就要是libXXX。否则，会报错：

java.lang.UnsatisfiedLinkError: no NativeAgent in java.library.path

Exception in thread "main" java.lang.UnsatisfiedLinkError: NativeAgent.toConsole(Ljava/lang/String;)I

at NativeAgent.toConsole(Native Method)

at NativeAgent.main(NativeAgent.java:20)

* 1
* 2
* 3
* 4

2、Linux一般默认的java.library.path在/usr/lib下。也可以自己通过VM参数-Djava.library.path=/usr/lib来显式的指定；或者通过增加环境变量export LD\_LIBRARY\_PATH=~/JavaNativeTest:$LD\_LIBRARY\_PATH

版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/promaster/article/details/70318695