JNI 设计实践之路

作者：杨小华

一、 前言

本文为在 32 位 Windows 平台上实现 Java 本地方法提供了实用的示例、步骤和准则。本文中的示例使用 Sun 公司的 Java Development Kit (JDK) 版本 1.4.2。 用 C ++语言编写的本地代码是用 Microsoft Visual C++ 6.0 编译器编译生成。规定在 Java 程序中 function/method 称为方法，在 C++程序中称为函数。

本文将围绕求圆面积逐步展开，探讨 java 程序如何调用现有的 DLL？如何在 C++程序中创建，检查及更新 Java 对象？如何在 C++和 Java 程序中互抛异常，并进行异常处理？最后将探讨 Eclipse 及 JBuilder 工具可执行文件为什么不到 100K 大小以及所采用的技术方案？

二、 JNI 基础知识简介

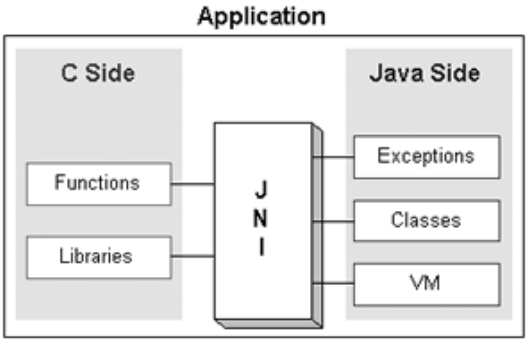
Java 语言及其标准 API 应付应用程序的编写已绰绰有余。但在某些情况下，还是必须使用非 Java 代码，例如：打印、图像转换、访问硬件、访问现有的非 Java 代码等。与非 Java 代码的沟通要求获得编译器和 JVM 的专门支持，并需附加的工具将 Java 代码映射成非 Java 代码。目前，不同的开发商为我们提供了不同的方案，主要有以下方法：

1. JNI（Java Native Interface）
2. JRI（Java Runtime Interface）
3. J/Direct
4. RNI（Raw Native Interface）
5. Java/COM 集成方案
6. CORBA（Common Object Request Broker Architecture）

其中方案 1 是 JDK 自带的一部分，方案 2 由网景公司所提供，方案 3 、 4 、 5 是微软所提供的方案，方案 6 是一家非盈利组织开发的一种集成技术，利用它可以在由不同语言实现的对象之间进行“相互操作”的能力。

在开发过程中，我们一般采用第 1 种方案——JNI 技术。因为只用当程序用 Microsoft Java 编译器编译，而且只有在 Microsoft Java 虚拟机(JVM)上运行的时候，才采用方案 3 、 4 、 5。而方案 6 一般应用在大型的分布式应用中。

JNI 是一种包容极广的编程接口，允许我们从 Java 应用程序里调用本地化方法。也就是说，JNI 允许运行在虚拟机上的 Java 程序能够与其它语言（例如 C/ C++/汇编语言）编写的程序或者类库进行相互间的调用。同时 JNI 也提供了一整套的 API，允许将 Java 虚拟机直接嵌入到本地的应用程序中。其中 JNI 所扮演的角色可用图一描述：



图一 JNI 基本结构描述图

目前 JNI 只能与用 C 和 C＋＋编写的本地化方法打交道。利用 JNI，我们本地化方法可以：

1. 创建、检查及更新 Java 对象
2. 调用 Java 和非 Java 程序所编写的方法(函数)，以及 win32 API 等.
3. 捕获和抛出“异常”
4. 装载类并获取类信息
5. 进行运行期类型检查

所以，原来在 Java 程序中能对类及对象所做的几乎所有事情都可以在本地化方法中实现。

下图表示了通过 JNI，Java 程序和非 Java 程序相互调用原理。

图二 Java 程序和非 Java 程序通过 JNI 相互调用原理

通过 JNI，编写 Java 程序调用非 Java 程序一般步骤：

1.) 编写对本地化方法及自变量进行声明的 Java 代码
2.) 利用头文件生成器 javah 生成本地化方法对应的头文件
3.) 利用 C 和 C++实现本地化方法（可调用非 Java 程序），并编译、链接生成 DLL 文件
4.) Java 程序通过生成的 DLL 调用非 Java 程序

同时我们也可以通过 JNI，将 Java 虚拟机直接嵌入到本地的应用程序中，步骤很简单，只需要在 C/C++程序中以 JNI API 函数为媒介调用 Java 程序。

以上步骤虽简单，但有很多地方值得注意。如果一招不慎，可能造成满盘皆输。

三、 Java 程序调用非 Java 程序

3.1 本地化方法声明及头文件生成

任务：现有一求圆面积的 Circle.dll（用 MFC 编写，参数：圆半径返回值：圆面积）文件，在 Java 程序中调用该 Dll。

在本地化声明中，可分无包和有包两种情况。我们主要对有包的情况展开讨论。

实例 1：

```
package com.testJni;
public class Circle
```

```
{
   public native void cAreas(int radius) ;
   static
   {
       //System.out.println(System.getProperty("java.library.path"));
       System.loadLibrary("CCircle");
    }
 }
```

在 Java 程序中，需要在类中声明所调用的库名称 System.loadLibrary(String libname);

该函数是将一个 Dll/so 库载入内存，并建立同它的链接。定位库的操作依赖于具体的操作系统。在 windows 下，首先从当前目录查找，然后再搜寻”PATH”环境变量列出的目录。如果找不到该库，则会抛出异常 UnsatisfiedLinkError。库的扩展名可以不用写出来，究竟是 Dll 还是 so，由系统自己判断。这里加载的是 3.2 中生成的 DLL，而不是其他应用程序生成的 Dll。还需要对将要调用的方法做本地声明，关键字为 **native**。表明此方法在本地方法中实现，而不是在 Java 程序中，有点类似于关键字 abstract。

我们写一个 Circle.bat 批处理文件编译 Circle.java 文件，内容如下(可以用其他工具编译)：

```
javac -d . Circle.java
javah com.testJni.Circle
pause
```

对于有包的情况一定要注意这一点，就是在用 javah 时有所不同。开始时我的程序始终运行都不成功，问题就出在这里。本类名称的前面均是包名。这样生成的头文件就是：com_testJni_Circle.h。开始时，在包含包的情况下我用 javah Circle 生成的头文件始终是 Circle.h。在网上查资料时，看见别人的头文件名砸那长，我的那短。但不知道为什么，现在大家和我一样知道为什么了吧。：）。

如果是无包的情况，则将批处理文件换成如下内容：

```
javac Circle.java
javah Circle
pause
```

3.2 本地化方法实现

刚才生成的 com_testJni_Circle.h 头文件内容如下：

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class com_testJni_Circle */
#ifndef _Included_com_testJni_Circle
#define _Included_com_testJni_Circle
#ifdef __cplusplus
extern "C" {
```

```
#endif
/*
 * Class:     com_testJni_Circle
 * Method:    cAreas
 * Signature: (I)V
 */
JNIEXPORT void JNICALL Java_com_testJni_Circle_cAreas
 (JNIEnv *, jobject, jint);
#ifdef __cplusplus
}
#endif
#endif
```

如果在本地化方法声明中，方法 cAreas ()声明为 static 类型，则与之相对应的 Java_com_testJni_Circle_cAreas()函数中的第二个参数类型为 jclass。也就是 JNIEXPORT void JNICALL Java_com_testJni_Circle_cAreas(JNIEnv *env, **jclass newCircle**,jint radius)。

这里 **JNIEXPORT** 和 **JNICALL** 都是 JNI 的关键字，其实是一些宏(具体参看 jni_md.h 文件)。

从以上头文件中，可以看出函数名生成规则为：**Java[_包名]_类名_方法名[_函数签名]**(其中[ ]是可选项)，均以字符下划线(_)分割。如果是无包的情况，则不包含[_包名]选项。如果本地化方法中有方法重载，则在该函数名最后面追加函数签名，也就是 Signature 对应的值，函数签名参见表一。

函数签名	Java 类型
V	void
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double
L fully-qualified-class ;	fully-qualified-class
[type	type[]
(arg-types) ret-type	method type

表一函数签名与 Java 类型的映射

在具体实现的时候，我们只关心函数原型：

JNIEXPORT void JNICALL Java_com_testJni_Circle_cAreas(JNIEnv *, jobject, jint);

现在就让我们开始激动人心的一步吧 :) 。启动 VC 集成开发环境，新建一工程，在 project 里选择 win32 Dynamic-link Library，输入工程名，然后点击 ok，接下去步骤均取默认(图三)。如果不取默认，生成的工程将会有 DllMain ()函数，反之将无这个函数。我在这里取的是空。

图三　新建 DLL 工程

然后选择菜单 File->new->Files->C++ Source File，生成一个空*.cpp 文件，取名为 CCircle。与 3.1 中 System.loadLibrary("CCircle");参数保持一致。将 JNIEXPORT void JNICALL Java_com_testJni_Circle_cAreas(JNIEnv *, jobject, jint);拷贝到 CPP 文件中，并包含其头文件。

对应的 CCircle.cpp 内容如下：

```
#include<iostream.h>
#include"com_testJni_Circle.h"
#include"windows.h"

JNIEXPORT void JNICALL Java_com_testJni_Circle_cAreas(JNIEnv *env, jobject newCircle,jint radius)
{
    //调用求圆面积的 Circle.dll
    typedef void (*PCircle)(int radius);
    HINSTANCE hDLL;
    PCircle Circle;
    hDLL=LoadLibrary("Circle.dll");//加载动态链接库 Circle.dll 文件
    Circle=(PCircle)GetProcAddress(hDLL,"Circle");
    Circle(8);
    FreeLibrary(hDLL);//卸载 Circle.dll 文件；
}
```

在编译前一定要注意下列情况。

注意：一定要把 SDK 目录下 include 文件夹及其下面的 win32 文件夹中的头文件拷贝到 VC 目录的 include 文件夹下。或者在 VC 的 tools\options\directories 中设置，如图四所示。

图四　头文件设置

我们知道 dll 文件有两种指明导出函数的方法，一种是在.def 文件中定义，另一种是在定义函数时使用关键字__declspec(dllexport)。而关键字 JNIEXPORT 实际在 jni_md.h 中如下定义，#define JNIEXPORT __declspec(dllexport)，可见 JNI 默认的导出函数使用第二种。使用第二种方式产生的导出函数名会根据编译器发生变化，在有的情况下会发生找不到导出函数的问题（我们在 java 控制

台程序中调用很正常，但把它移植到 JSP 页面时，就发生了该问题，JVM 开始崩溃，百思不得其解，后来加入一个.def 文件才解决问题）。其实在《windows 核心编程》一书中，第 19.3.2 节就明确指出创建用于非 Visual C++工具的 DLL 时，建议加入一个 def 文件，告诉 Microsoft 编译器输出没有经过改变的函数名。因此最好采用第一种方法，定义一个.def 文件来指明导出函数。本例中可以新建一个 CCircle.def 文件，内容如下：

```
; CCircle.def : Declares the module parameters for the DLL.

LIBRARY      "CCircle"
DESCRIPTION 'CCircle Windows Dynamic Link Library'

EXPORTS
    ; Explicit exports can go here
    Java_com_testJni_Circle_cAreas
```

现在开始对所写的程序进行编译。选择 build->rebuild all 对所写的程序进行编译。点击 build->build CCirclee.DLL 生成 DLL 文件。

也可以用命令行 cl 来编译。语法格式参见 JDK 文档 JNI 部分。

再次强调（曾经为这个东西大伤脑筋）： DLL 放置地方

1) 当前目录。
2) Windows 的系统目录及 Windows 目录
3) 放在 path 所指的路径中
4) 自己在 path 环境变量中设置一个路径，要注意所指引的路径应该到.dll 文件的上一级，如果指到.dll，则会报错。

下面就开始测试我们的所写的 DLL 吧（假设 DLL 已放置正确）。

```
import com.testJni.Circle;
public class test
{
   public static void main(String argvs[])
   {
      Circle myCircle;
      myCircle = new Circle();
      myCircle.cAreas(2);
   }
}
```

编译，运行程序，将会弹出如下界面：

图五 运行结果

以上是我们通过 JNI 方法调用的一个简单程序。实际情况要比这复杂的多。

现在开始来讨论 JNI 中参数的情况，我们来看一个程序片断。

实例二：

```
JNIEXPORT jstring JNICALL Java_MyNative_cToJava
(JNIEnv *env, jclass obj)
{
    jstring jstr;
    char str[]="Hello,word!\n";
    jstr=env->NewStringUTF(str);
    return jstr;
 }
```

在 C 和 Java 编程语言之间传送值时，需要理解这些值类型在这两种语言间的对应关系。这些都在头文件jni.h中定义，用typedef语句声明了这些类在目标平台上的代价类。头文件也定义了常量如：JNI_FALSE=0 和 JNI_TRUE=1;表二和表三说明了 Java 类型和 C 类型之间的映射关系。

Java 语言	C/C++语言	bit 位数
boolean	jboolean	8 unsigned
byte	jbyte	8
char	jchar	16 unsigned
short	jshort	16
int	jint	32
long	jlong	64
float	jfloat	32
double	jdouble	64
void	void	0

表二　Java 基本类型到本地类型的映射

表三 Java 中的类到本地类的映射

JNI 函数 NewStringUTF()是从一个包含 UTF 格式编码字符的 char 类型数组中创建一个新的 jstring 对象。jstring 是以 JNI 为中介使 Java 的 String 类型与本地的 string 沟通的一种类型，我们可以视而不见 (具体对应见表二和表三)。如果你使用的函数是 GetStringUTFChars()(将 jstring 转换为 UTF-8 字符串)，必须同时使用 ReleaseStringUTFChars()函数，通过它来通知虚拟机去回收 UTF-8 串占用的内存，否则将会造成内存泄漏，最终导致系统崩溃。因为 JVM 在调用本地方法时，是在虚拟机中开辟了一块本地方法栈供本地方法使用，当本地方法使用完 UTF-8 串后，得释放所占用的内存。其中程序片断 jstr=env->NewStringUTF(str);是 C++中的写法，不必使用 env 指针。因为 JNIEnv 函数的 C++版本包含有直接插入成员函数，他们负责查找函数指针。而对于 C 的写法，应改为：

jstr=**(*env)->**NewStringUTF(**env**,str);因为所有 JNI 函数的调用都使用 env 指针，它是任意一个本地方法的第一个参数。env 指针是指向一个函数指针表的指针。因此在每个 JNI 函数访问前加前缀(*env)->，以确保间接引用函数指针。

C/C++和 Java 互传参数需要自己在编程过程中仔细摸索与体味。

四、 C/C++访问 Java 成员变量和成员方法

我们修改 3.1 中的 Java 程序声明，加入如下代码：

```
private int circleRadius;
public Circle()
{
   circleRadius=0;
}
public void setCircleRadius(int radius)
{
   circleRadius=radius;
}
public void javaAreas()
{
   float PI = 3.14f;
   if(circleRadius<=0)
   {
      System.out.println (“error!”);
   }
   else
   {
      System.out.println (PI*circleRadius*circleRadius);
   }
 }
```

在 C++程序中访问 Circle 类中的 private 私有成员变量 circleRadius，并设置它的值，同时调用 Java 方法 javaAreas()。在函数 Java_com_testJni_Circle_cAreas()中加入如下代码：

```
  jclass circle;
       jmethodID AreasID;
       jfieldID radiusID;
  jint newRadius=5;
circle = env->GetObjectClass(newCircle);//get current class
radiusID=env->GetFieldID(circle,"circleRadius","I");//get field ID
env->SetIntField(newCircle,radiusID,newRadius);//set field value
```

AreasID=env->GetMethodID(circle,"javaAreas","()V");//get method ID

env->CallVoidMethod(newCircle,AreasID,NULL);//invoking method

在 C++代码中，创建、检查及更新 Java 对象，首先要得到该类，然后再根据类得到其成员的 ID，最后根据该类的对象，ID 号调用成员变量或者成员方法。

得到类，有两个 API 函数，分别为 FindClass()和 GetObjectClass();后者顾名思义用于已经明确知道其对象，然后根据对象找类。前者用于得到没有实例对象的类。这里也可以改成 circle = env-> FidnClass("com/testJni/Circle");其中包的分隔符用字符" /"代替。如果已知一个类，也可以在 C++代码中创建该类对象，其 JNI 函数为 NewObject();示例代码如下：

jclass circle =env->FindClass("com/testJni/ Circle ");

jmethodID circleID=env->GetMethodID(circle,"<init>","()V");//得到构造函数的 ID

jobject newException=env->NewObject(circle, circleID,NULL);

得到成员变量的 ID，根据其在 Java 代码中声明的类型不同而不同。具体分为两大类：非 static 型和 static 型，分别对应 GetFieldID()和 GetStaticFieldID()。同时也可以获得和设置成员变量的值，根据其声明的 type 而变化，获得其值的 API 函数为：**Get**type**Field**()和 **GetStatic**type**Field**()；与之相对应的设置其值的函数为 **Set**type**Field**()和 **SetStatic**type**Field**()；在本例中，成员变量 circleRadius 声明成 int 型，则对应的函数分别为 GetIntField()和 SetIntField();

其实 JNI API 函数名是很有规律的，从上面已窥全貌。获得成员方法的 ID 也是同样的分类方法。具体为 GetMethodID()和 GetStaticMethodID()。调用成员方法跟获得成员变量的值相类似，也根据其方法返回值的 type 不同而不同，分别为 **Call**type**Method**()和 **CallStatic**type**Method**()。对于返回值为 void 的类型，其相应 JNI 函数为 CallVoidMethod()；

以上获得成员 ID 函数的形参均一致。第一个参数为 jclass，第二个参数为成员变量或方法，第三个参数为该成员的签名(签名可参见表一)。但调用或设置成员变量或方法时，第一个参数为实例对象(即 jobject)，其余形参与上面相同。

特别要注意的是得到构造方法的 ID 时，第二个参数不遵循上面的原则，为 jmethodID constructorID = env->GetMethodID(jclass, **"<init>"**," 函数签名");

从上面代码中可以看出，在 C++中可以访问 java 程序 private 类型的变量，**严重破坏了类的封装原则。从而可以看出其不安全性。**

五、 异常处理

本地化方法稳定性非常差，调用任何一个 JNI 函数都会出错，为了**程序的健壮性**，非常有必要在本地化方法中加入异常处理。我们继续修改上面的类。

我们声明一个异常类，其代码如下：

```
package com.testJni;
import com.testJni.*;
public class RadiusIllegal extends Exception
{
    protected String MSG="error!";
```

```
        public RadiusIllegal(String message)
        {
           MSG=message;
        }
        public void print()
        {
          System.out.println(MSG);
        }
    }
```

同时也修改 Circle.java 中的方法,加入异常处理。

```
 public void javaAreas() throws RadiusIllegal     //修改 javaAreas()，加入异常处理
{
    float PI = 3.14f;
    if(circleRadius<=0)
    {
       throw new RadiusIllegal("warning:radius is illegal!");
    }
    else
    {
        System.out.println (PI*circleRadius*circleRadius);
    }
}
public native void cAreas(int radius) throws RadiusIllegal;     //修改 cAreas ()，加入异常处理
```

修改 C++代码中的函数，加入异常处理，实现 Java 和 C++互抛异常，并进行异常处理。

```
JNIEXPORT void JNICALL Java_com_testJni_Circle_cAreas(JNIEnv *env, jobject newCircle,jint
radius)
  {
    //此处省略部分代码
    radiusIllegal=env->FindClass("com/testJni/RadiusIllegal");//get the exception class
    if((exception=env->ExceptionOccurred())!=NULL)
    {
        cout<<"errors in com_testJni_RadiusIllegal"<<endl;
        env->ExceptionClear();
    }
    //此处省略部分代码
    env->CallVoidMethod(newCircle,AreasID,NULL);//invoking
    if((exception=env->ExceptionOccurred())!=NULL)
```

```
{
    if(env->IsInstanceOf(exception,radiusIllegal)==JNI_TRUE)
    {
        cout<<"errors in java method"<<endl;
        env->ExceptionClear();
    }
    else
    {
        cout<<"errors in invoking javaAreas() method of Circle"<<endl;
        env->ExceptionClear();
    }
 }
 if(radius<=0)
 {
    env->ThrowNew(radiusIllegal,"errors in C function!");//throw exception
  return ;
 }
 else
 {
    //此处为调用计算圆面积的 DLL
 }
}
```

在本地化方法(C++)中，可以自己处理异常，也可以重新抛出异常，让 Java 程序来捕获该异常，进行相关处理。

如果调用 JNI 函数发生异常，不及时进行处理，再次调用其他 JNI 函数时，可能会使 JVM 崩溃(crash),
大多数 JNI 函数都具有此特性。可以调用函数 ExceptionOccurred()来判断是否发生了异常。该函数返回 jthrowable 的实例对象，如本例 if((exception=env->ExceptionOccurred())!=NULL)就用来判断是否发生了异常。当要判断具体是哪个异常发生时，可以用 IsInstanceOf()来进行测试，此函数非彼 IsInstanceOf(Java 语言中的 IsInstanceOf)。在上面的代码中，我们在本地化方法中给 circleRadius 设置了一非法值，然后调用方法 javaAreas(),此时 java 代码会抛出异常，在本地化方法中进行捕获，然后用 IsInstanceOf()来进行测试是否发生了 RadiusIllegal 类型的异常，以便进行相关处理。在调用其他 JNI 函数之前，应当首先清除异常，其函数为 ExceptionClear()。

如果是 C++的程序发生异常，则可以用 JNI API 函数 ThrowNew()抛出该异常。但此时本地化方法并不返回退出，直到该程序执行完毕。所以当在本地化方法中发生异常时，应该人为的退出，及时进行处理，避免程序崩溃。函数 ThrowNew()中第一个参数为 jclass 的类，第二个参数为附加信息，用来描述异常信息。

如果要知道异常发生的详细信息，或者对程序进行调试时，可以用函数 ExceptionDescribe()来显示异常栈里面的内容。

六、 MFC 程序中嵌入 Java 虚拟机

可能大家天天都在用 Eclipse 和 Jbulider 这两款优秀的 IDE 进行程序开发，可能还不知道他们的可执行文件不到 100KB 大小，甚则连一副图片都可能比他们大。其实隐藏在他们背后的技术是 JNI，可执行文件只是去启动 Java 程序，所以也只有那么小。

我们只需要在 MFC 程序中创建一个 JVM，然后基于这个 JVM 调用 Java 的方法，启动 Java 程序，就可以模拟出 Eclipse 和 Jbulider 的那种效果，使 java 程序更专业。其实要实现这种效果，用上面的方法足有够有。创建 JVM,只需包含相应的类库，设置相关的属性。

首先进行环境设置,在 VC 环境的 tools-->options-->Directories 下的 Library files 选项中包含其创建 JVM 的库文件 jvm.lib，该库文件位于 JDK \ lib 目录下，如图 6 所示：

图六库文件路径设置

然后，在环境变量 path 中设置 jvm.dll 的路径。该 Dll 位于 jdk\jre\bin\server 目录或 jdk\jre\bin\client 目录下。**注意：一定不要将 jvm.dll 和 jvm.lib 拷贝到你应用程序路径下**，这样会引起 JVM 初始化失败。因为 Java 虚拟机是以相对路径来寻找和调用用到的库文件和其他相关文件。

接下来，我们在 MFC 程序(该程序请到《程序员》杂志频道下载)中进行创建 JVM 初始化工作。示例代码如下：

```
JNIEnv *env;
JavaVM *jvm;
jint res;
jclass cls;
jmethodID mid;
JavaVMInitArgs vm_args;
JavaVMOption options[3];
memset(&vm_args, 0, sizeof(vm_args));
//进行初始化工作
options[0].optionString = "-Djava.compiler=NONE";
options[1].optionString = "-Djava.class.path=.";
options[2].optionString = "-verbose:jni";
vm_args.version=JNI_VERSION_1_4;        //版本号设置
vm_args.nOptions = 3;
vm_args.options = options;
vm_args.ignoreUnrecognized = JNI_TRUE;
```

```
res = JNI_CreateJavaVM(&jvm,(void**)&env,&vm_args); //创建 JVM
if (res < 0)
{
    MessageBox( "Can't create Java VM","Error",MB_OK|MB_ICONERROR);
    exit(1);
}
cls = env->FindClass("prog");
if(env->ExceptionOccurred()!=NULL)
{
    MessageBox( "Can't find Prog class!","Error",MB_OK|MB_ICONERROR);
    exit(1);
}
mid = env->GetStaticMethodID(cls, "main", "([Ljava/lang/String;)V");
if(env->ExceptionOccurred()!=NULL)
{
    MessageBox("Can't find Prog.main!","Error",MB_OK|MB_ICONERROR);
    exit(1);
}
env->CallStaticVoidMethod( cls, mid, NULL); //调用 Java 程序 main()方法，启动 Java 程序
if(env->ExceptionOccurred()!=NULL)
{
    MessageBox( "Fatal Error!","Error",MB_OK|MB_ICONERROR);
    exit(1);
}
jvm->DestroyJavaVM();//释放 JVM 资源
```

程序首先进行 JVM 初始化设置。我们观察 jni.h 文件关于 JavaVMOption 和 JavaVMInitArgs 的定义

```
typedef struct JavaVMOption {
    char *optionString;
    void *extraInfo;
} JavaVMOption;

typedef struct JavaVMInitArgs {
    jint version;
    jint nOptions;
    JavaVMOption *options;
    jboolean ignoreUnrecognized;
```

} JavaVMInitArgs;

结构体 JavaVMInitArgs 中有四个参数，我们在程序中都得必须设置。其中版本号一定要设置正确，不同的版本有不同的设置方法，关于版本 1.1 和 1.2 的设置方法参看 sun 公司的文档，这里只给出版本 1.4 的设置方法。第二个参数表示 JavaVMOption 结构体变量的维数，这里设置为三维，其中 options[0].optionString = "-Djava.compiler=NONE";表示 disable JIT;options[1].optionString = "-Djava.class.path=.";表示你所调用 Java 程序的 Class 文件的路径，这里设置为该 exe 应用程序的根路径(最后一个字符"."表示根路径)；options[2].optionString = "-verbose:jni";用于跟踪运行时的信息。第三个参数是一个 JavaVMOption 的指针变量。第四个参数意思我们可以参看帮助文档的解释 If ignoreUnrecognized is JNI_FALSE, JNI_CreateJavaVM returns JNI_ERR as soon as it encounters any unrecognized option strings。

初始化完毕后，就可以调用创建 JVM 的函数 jint JNICALL JNI_CreateJavaVM(JavaVM **pvm, void **penv, void *args);如果返回值小于 0 表示创建 JVM 失败。最可能的原因就是 **jvm.dll 和 jvm.lib** 设置错误。

如果在运行的过程中找不到 java 程序的类，那么就是-Djava.class.path 设置错误。只要 JVM 创建成功，就可以根据上面的方法调用 java 程序。最后当程序结束后，调用函数 DestroyJavaVM()摧毁 JVM，释放资源。

七、 附录

利用 JNI 函数，我们可以从本地化方法的内部与 JVM 打交道。正如在前面的例子中所看到的那样，每个本地化方法中都会接收一个特殊的自变量作为自己的第一个参数：JNIEnv——它是指向类型为 JNIEnv_的一个特殊 JNI 数据结构的指针。JNI 数据结构的一个元素是指向由 JVM 生成的一个指针的数组；该数组的每个元素都是指向一个 JNI 函数的指针。可以从本地化方法的内部对 JNI 函数的调用。第二个参数会根据 Java 类中本地方法的定义不同而不同，如果是定义为 static 方法，类型会是 jclass，表示对特定 Class 对象的引用，如果是非 static 方法，类型是 jobject，表示当前对象的引用，相当于” this”。可以说这两个变量是本地化方法返回 JAVA 的大门。

注意：在本地化方法中生成的 Dll 不具备到处运行的特性，而具有”牵一发而动全身”的特点。只要包名一改变，那么你所有的工作就得重新做一遍。原因就是当用 javah 生成头文件时，函数名的生成规则为 **Java[_包名]_类名_方法名[_函数签名]**; 当你的包名改变时，生成的函数名也跟着改变了，那么你再次调用以前编写的 Dll 时，会抛出异常。

八、 参考文献

1. 《Java 编程思想》Bruce Eckel 机械工业出版社
2. 《Java2 核心技术卷 2》（第 6 版）Cay S.Horstmann,Gary Cornell 机械工业出版社
3. 《高级 Java2 大学教程》(英文版) Harvey M.Deitel ,Paul J.Deitel,Sean E.Santry 电子工业出版社
4. 《windows 核心编程》Jeffrey Richter 机械工业出版社
5. http://www.jguru.com

6. sun 公司文档

如对本文有任何疑问和异议，欢迎与作者探讨：normalnotebook@126.com

注：本文最初发表在 2004 年《开发高手》第 12 期上。