**JNI入门教程**

1. Java调用C++代码
2. 什么是JNI

Java Native Interface（JNI）是 Java 语言的本地编程接口；是 Java 与操作系统本地代码互相调用的功能的接口。

1. Java调用C++的步骤
2. 在Java类中声明native方法

->新建一个Java工程TestNativeCode；

->包com.jouav.jni，类TestNative；

package com.jouav.jni;

public class TestNative {

public native void sayHello();

    static {

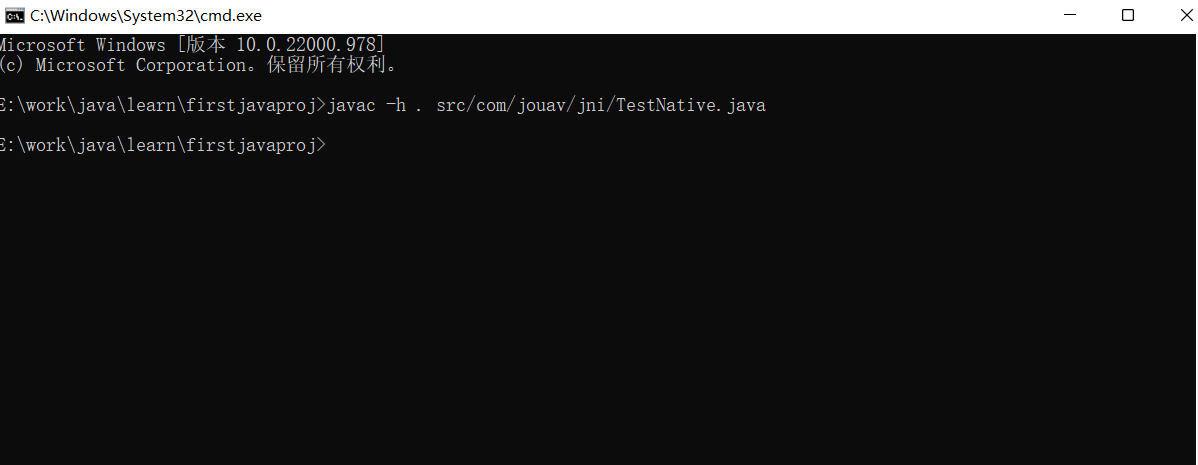
        //System.loadLibrary("nativecode1");

        System.load("E:/work/cpp/jni/jni-learn/x64/Debug/nativecode1.dll");

    }

}

1. 使用 javac -h 命令生成包含 native 方法定义的C/C++头文件





1. 按照生成的 C/C++头文件来写 C/C++ 源文件

（1）在 VS 中新建DLL项目；

（2）拷贝com\_jouav\_jni\_TestNative.h文件到项目目录下；

（3）添加现有项 com\_jouav\_jni\_TestNative.h引入头文件；

（4）新建源文件 com\_jouav\_jni\_TestNative.cpp 添加如下内容；

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

#include <jni.h>

/\* Header for class com\_jouav\_jni\_TestNative \*/

#ifndef \_Included\_com\_jouav\_jni\_TestNative

#define \_Included\_com\_jouav\_jni\_TestNative

#ifdef \_\_cplusplus

extern "C" {

#endif

/\*

 \* Class:     com\_jouav\_jni\_TestNative

 \* Method:    sayHello

 \* Signature: (Ljava/lang/String;)V

 \*/

JNIEXPORT void JNICALL Java\_com\_jouav\_jni\_TestNative\_sayHello

  (JNIEnv \*, jobject);

#ifdef \_\_cplusplus

}

#endif

#endif

void JNICALL Java\_com\_jouav\_jni\_TestNative\_sayHello(JNIEnv\* env, jobject obj)

{

  std::cout << "JNI: Hello world!" << std::endl;

}

1. 将C/C++源文件编译成动态链接库（DLL）可执行二进制代码执行生成解决方案，在项目目录下的Debug生成DLL文件。
2. 把DLL文件所在路径添加到PATH环境变量下。
3. java类中加载DLL，然后调用声明的native方法：

import com.jouav.jni.\*;

public class App {

    public static void main(String[] args) throws Exception {

        TestNative test\_native = new TestNative();

        test\_native.sayHello();

    }

}

1. 运行Java项目，执行结果：JNI: Hello world!

二. C++访问Java代码-知识点

这里主要是介绍JNI中提供的调用和修改以及其他一些操作java代码的方法，具体如下：

1. Java 工具生成的 C/C++函数声明中，可以看到两个参数：

//对 com\_jouav\_jni\_TestNative.h 中声明的方法定义

void JNICALL Java\_com\_jouav\_jni\_TestNative\_sayHello(JNIEnv\* env, jobject obj)

{

  std::cout << "JNI: Hello world!" << std::endl;

}

**JNIEnv \*env**

JNIEnv 类型可以看做是Java环境，通过这个JNIEnv\*指针，可以对Java端的代码进行操作，JNIEnv的指针会被JNI传入到本地方法的实现函数中来对Java端的代码操作。例如：创建Java类的对象，调用Java对象的方法，获取Java对象的属性等。

函数：

NewObject/NewString/New<TYPE>Array //创建数组

Get/Set<TYPE>Field //获取/设定变量属性

Get/SetStatic<TYPE>Method/CallStatic<TYPE>Method … //对方法进行操作

我们可以在 jni.h 头文件中找到所有定义的方法：

struct JNIEnv\_; //定义所有方法的结构体

typedef JNIEnv\_ JNIEnv;

………………

struct JNIEnv\_ {

const struct JNINativeInterface\_ \*functions;

………………

jclass FindClass(const char \*name) {

return functions->FindClass(this, name);

}

………………

jint GetIntField( jobject obj, jfieldID fieldID) { return functions->GetIntField(this,obj,fieldID);

}

}

**jobject obj**

同样也可以在jni.h中看到对jobject的定义，表示传进来的参数为调用该方法的对象，如果是静态native函数，则表示native方法所在的类。

class \_jobject {};

typedef \_jobject \*jobject; //定义\_jobject 类型别名 为 jobject

2. Java 类型在C/C++中的映射关系

这个没有什么要说的，直接看图，就是 JNI 为了我方便使用定义的别名；



3. jclass的取得

为了可以在 C/C++中使用 Java 类jni.h 头文件中专门定义了jclass类型来表示Java中的Class 类，JNIEnv类中有如下几个函数取得jclass：

jclass FindClass(const char\* clsName);

jclass GetObjectClass(jobject obj);

jclass GetSuperClass(jclass obj);

FindClass会在classpath系统环境变量下找类，传入完整类名，注意包与包之间用”/” 而不是”.”来连接，如：jclass cls\_string=env->FindClass(“java/lang/String”);

4. 访问 Java 类中的属性和方法

在C/C++本地代码中访问Java端的代码，一个常见的应用就是获取类的属性和调用类的方法，为了在C/C++中表示属性和方法，JNI在jni.h头文件中定义了jfieldID、jmethodID类型来分别代表Java端的属性和方法，在访问设置Java属性和方法时，首先在本地代码获取代表Java属性的jfieldID和jmethodID然后进行Java属性和方法的操作。

使用JNIEnv中GetFieldID/GetMethodID，GetStaticFieldID/GetStaticMethodID 获取。如：

env->GetMethodID(data\_Clazz,”<init>”,”()V”); //()V 指无返回类型void，下面讲到

5. Sign是什么？

例如 TestNative.java中有两个重载方法：

package com.jouav.jni;

public class TestNative {

    public void function(int i){

        System.out.println("Integer: "+i);

    }

    public void function(double d){

        System.out.println("Double: "+d);

    }

}

当在 C++中调用其中一个函数的时候，就需要按照以下步骤：

获取函数所在类：

jclass clazz\_TestNative=env->FindClass(“com/jouav/jni/TestNative;”);

获取方法的 jmethodID 来调用方法：

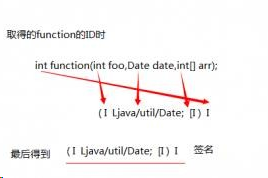
jmethodID id\_func = env->GetMethodID (clazz\_TestNative,”function”,” (I)V”);

jmethodID id\_func = env->GetMethodID (clazz\_TestNative,”function”,” (D)V”);

其中，通过指定函数的参数类型和返回值类型来返回jmethodID，从而对应的函数，这就是sign。

以下是对应类型：





6. 使用 javap 命令生成签名

JDK中提供了javap 工具来查看类的声明，可以输出每个方法与属性的签名package com.jouav.jni;

package com.jouav.jni;

import java.util.Date;

public class TestNative {

public native void sayHello(); //C++本地代码实现

    public static void main(String[] args) {

    System.loadLibrary("NativeCode");//加载动态链接库

        TestNative test=new TestNative();

        test.sayHello();

    }

public int property;

    public int function(int foo,Date date, int[] arr){

        return 0;

    }

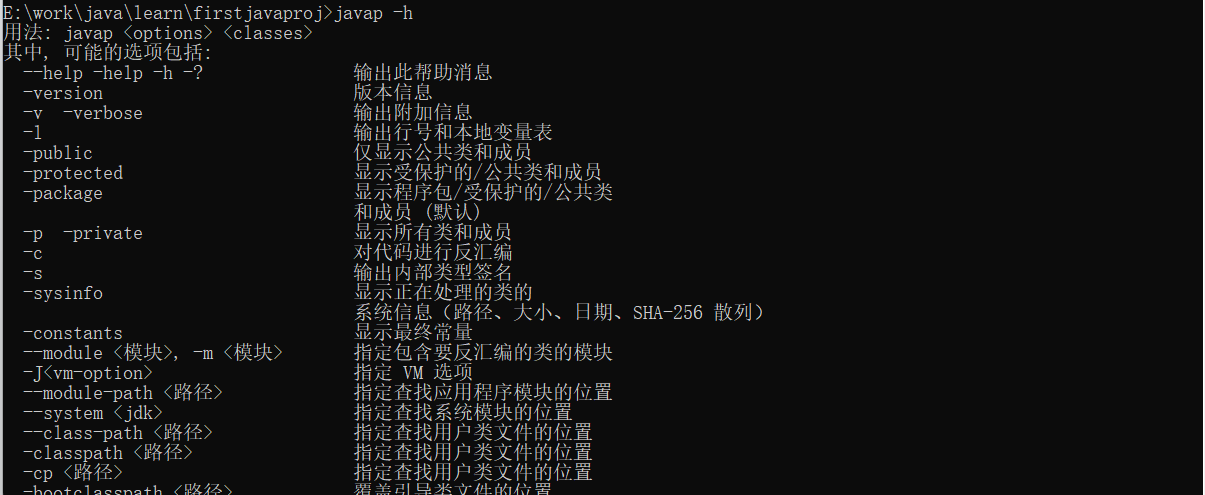
}

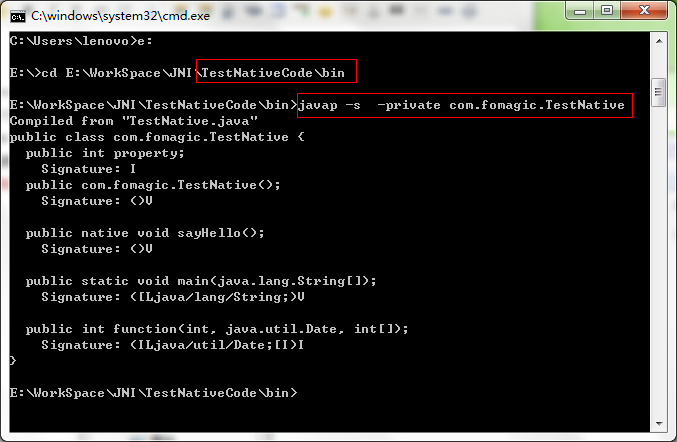
我们对上面代码通过 javap 工具直接生成方法和属性的签名：

->命令行定位到该类的绝对路径 cd TestNativeCode\bin 目录

->执行 javap -s -private com.jouav.jni.Testnative javap -s -p… [完整类名]

注：private 获取全部权限的成员、public 获取 public 修饰的成员 (即 获取大于或等于自己权限的成员方法或属性)





三. C++操作 java 属性和方法

在第二讲中讲到了C++中操作 java 程序的方法以及操作步骤,下面就通过实例的方式来体会到底怎么通过 JNI 提供的方法来获取、操作 java 属性、方法以及父类方法。

1. 取得/设定 Java 属性值

在原来 java 类 TestNative 的基础上编写如下代码：

package com.jouav.jni;

public class TestNative {

    public native void sayHello(); //C++本地代码实现

    public int number=10;   //定义变量初始值为 10，在 C++中进行修改

    public static void main(String[] args) {

        System.loadLibrary("NativeCode");//加载动态链接库

        TestNative test=new TestNative();   //创建本类实例

test.sayHello();

        System.out.println("C++中操作之后的结果："+test.number);

    }

}

C++中原有代码添加如下：

JNIEXPORT void JNICALL Java\_com\_jouav\_jni\_TestNative\_sayHello (JNIEnv \*env, jobject obj)

{

  cout<<"Hello World !"<<endl;

  jclass clazz\_TestNative=env->GetObjectClass(obj); //通过传入的 obj 类的对象获取当前类

  jfieldID id\_number=env->GetFieldID(clazz\_TestNative,"number","I");  //类实例, 属性名,(sign)数据类型

  jint number =env->GetIntField(obj,id\_number); //  通过 ID 获取当前变量

  cout<<"原数据： "<<number <<endl;

  env->SetIntField(obj,id\_number,100L); //类实例,变量 ID, 修改的 java 属性值

}

注：修改的属性值 100L，可以参看第二讲的 “Java 类型在 C/C++中的映射关系” 图表( java中int对应C++中long) 。

1. 调用 Java 类的方法

调用实例方法的三种形式:

Call<TYPE>Method( jobject obj ,jmethodID id ,….);

Call<TYPE>MethodV( jobject obj ,jmethodID id ,va\_list lst);

Call<TYPE>Method( jobject obj ,jmethodID id ,jvalue\* v) ;

第一个比较常用，第二个是当调用这个函数有一个指向参数表的 va\_list 变量时使用(只知道是可变参数列表)，第三个当调用函数时有一个指向 jvalue 或 jvalue 数组的指针时使用(即传入的是所有参数的变量数组指针)，下面就以第一种方式写下返回较大值的实例：

package com.jouav.jni;

public class TestNative {

    public native void sayHello(); //C++本地代码实现

    public static void main(String[] args) {

        System.loadLibrary("NativeCode");//加载动态链接库

        TestNative test=new TestNative();

        test.sayHello();

    }

    double max(double num1,double num2) {

        return num1>num2 ? num1 : num2;

    }

}

JNIEXPORT void JNICALL Java\_com\_jouav\_jni\_TestNative\_sayHello (JNIEnv \*env, jobject obj)

{

  jclass clazz\_TestNative=env->GetObjectClass(obj); //获取当前类

  jmethodID id\_max=env->GetMethodID(clazz\_TestNative,"max","(DD)D"); // 获取方法 ID(签名)

  jdouble maxValue = env->CallDoubleMethod(obj,id\_max,100.5,203.1); // 调用max 方法进行传值

  cout<<"较大值： "<<maxValue <<endl;

}

3. 调用 Java 类父类方法

我们都知道在 java 中父类创建子类对象只能调用子类的覆盖方法，而 C++中子类对象默认调用的父类的方法，当父类方法用虚拟函数关键字 virtual 修饰时，调用子类覆盖方法

java

Father p= new Child();

p.function();//调用子类覆盖方法

C++

Father\* p=new Child();

p->function();// 默认调用父类方法，用 virtual 修饰符修饰则调用子类覆盖方法

通过 JNI 的 CallNonvirtual<TYPE>Method 可以实现子类对象调用父类被覆盖的方法的功能,步骤：

->取得父类和需要调用的父类中方法的 jmethodID

->将 jmethodID 传入CallNonvirtual<TYPE>Method 代码：

新建 Father 类 、Child 类：

package com.jouav.jni;

public class TestNative {

    public native void sayHello(); // C++本地代码实现

    public Father p = new Child(); //创建 Child 的实例对象

    public static void main(String[] args) {

        System.loadLibrary("NativeCode");// 加载动态链接库

        TestNative test = new TestNative();

        test.sayHello();

    }

}

class Father {

    public void function() {

        System.out.println("Father function");

    }

}

class Child extends Father {

    public void function() {

        System.out.println("Child function");

    }

}

JNIEXPORT void JNICALL Java\_com\_jouav\_jni\_TestNative\_sayHello (JNIEnv \*env, jobject obj)

  {

  jclass clazz\_TestNative=env->GetObjectClass(obj);

  jfieldID id\_p=env->GetFieldID(clazz\_TestNative,"p","Lcom/jouav/jni/Father;"); jobject p= env->GetObjectField(obj,id\_p);

  jclass clazz\_Father = env->FindClass("com/jouav/jni/Father");

  jmethodID id\_Father\_function =

  env->GetMethodID(clazz\_Father,"function","()V");

  //env->CallVoidMethod(p,id\_Father\_function);;

  env->CallNonvirtualVoidMethod(p,clazz\_Father,id\_Father\_function);

}

四.一 在 C++中创建 Java 对象

本节讲的是在 C/C++本地代码中创建 Java 对象，JNI 中提供两个方法进行 Java 对象的操作，分别是 NewObject 和 Allocobject，先看下 jni.h 头文件中对两个方法的定义：

jobject NewObject( jclass clazz, jmethodID methodID, ...) {

  va\_list args;

  jobject result; va\_start(args, methodID);

  result = functions->NewObjectV(this,clazz,methodID,args);

  va\_end(args);

  return result;

}

jobject AllocObject( jclass clazz) {

  return functions->AllocObject(this,clazz);

}

1. NewObject 创建 Java 对象

GetMethodID 能够获取构造方法的 jmethod ，如果传入的要取得的方法名称设定为”<init>”就能取得构造方法构造方法没有返回值，签名始终为 Void (()V)。

下面是一个通过 C++创建 Java 的 Date 对象并在 Java 控制台输出 1970 到现在毫秒数

package com.jouav.jni;

public class TestNative {

    public native void sayHello(); // C++本地代码实现

    public static void main(String[] args) {

        System.loadLibrary("NativeCode");// 加载动态链接库，不能加 .dll

        TestNative test = new TestNative(); test.sayHello();

    }

}

C++

JNIEXPORT void JNICALL Java\_com\_jouav\_jni\_TestNative\_sayHello (JNIEnv \*env, jobject obj)

{

  jclass clazz\_date = env->FindClass("java/util/Date"); //找到 Date 类

  jmethodID id\_date= env->GetMethodID(clazz\_date,"<init>","()V");// 通过类获取构造方法 ID，默认无返回值

  jobject now=env->NewObject(clazz\_date,id\_date); //创建 Date 的实例对象

  jmethodID id\_date\_getTime=env->GetMethodID(clazz\_date,"getTime","()J"); // 获取 Date 类的 getTime()方法的 ID，返回值是 long 即 JNI 中对应的 J

  jlong time = env->CallLongMethod(now,id\_date\_getTime); //通过 now 对象调用 getTime()方法，得到时间

  cout<<time<<endl;

}

代码很简单通过前面的知识都是可以明白的，注释应该没什么问题（吧）

2、Allocobject 创建 Java 对象

使用函数 AllocObject 可以根据传入的 jclass 创建一个 Java 对象，但他的状态是非初始化的，在使用这个对象之前绝对要用 CallNovirtualVoidMethod 来调用该 jclass 的建构函数，这样可以延迟构造函数的调用，这一部分用的很少。

jclass clazz\_str =env->FindClass("java/lang/String");

jmethodID methodID\_str=env->GetMethodID(clazz\_str,"<init>","([C)V");//获取构造方法 ID

jObject string =env->AllocObject(clazz\_str);// 预先创建没有初始化的字符串jcharArray arg=env->NewCharArray(4); //创建长度为 4 的字符数组

env->SetCharArrayRegion(arg,0,4,L"奇幻未来"); //复制"奇幻未来"到数组 arg 中

env->CallNonvirtualVoidMethod(string,clazz\_str,methodID\_str,arg); // 将arg 字 符数组复制给 string

jclass clazz\_this=env->GetObjectClass(obj);

// 这是假设这个对象的类中有定义 static String STATIC\_STR;

jfieldID

fieldID\_str=env->GetStaticFieldID(clazz\_this,"STATIC\_STR","Ljava/lang/String;");

env->SetStaticObjectField(clazz\_str,fieldID\_str,string);

四.二、C++访问 Java 的 String 字符串对象

第四讲的第二部分：

在 C/C++本地代码中访问Java的String字符串对象Java与C++中字符串的区别。在 Java 中，使用的字符串 String 对象时 Unicode(UTF-16)码，即每个字符不论是中文还是英文还是符号，一个字符总是占两个字节。

Java 通过 JNI 接口可以将 Java 的字符串转换到 C/C++的宽字符串(wchar\_t\*),或是传回一个 UTF-8 的字符串(char\*)到 C/C++。

C/C++可以通过一个wchar\_t字符串或是一个 UTF-8 编码的字符串来创建一个 Java 端的 String对象。

JNI 中操作字符串相关函数

函数作用是取得某个 jstring 对象相关的 Java 字符串，介绍了三种类型的字符串操作函数。

1、GetStringChars 与 GetStringUTFChars

GetStringChars 取得 UTF-16 编码的宽字符串( jchar\*)

GetStringUTFChars 取得 UTF-8 编码的字符串(char\*)

const jchar \*GetStringChars( jstring str, jboolean \*isCopy) {

return functions->GetStringChars(this,str,isCopy);

}

const char\* GetStringUTFChars( jstring str, jboolean \*isCopy) {

return functions->GetStringUTFChars(this,str,isCopy);

}

void ReleaseStringChars( jstring str, const jchar \*chars) {

functions->ReleaseStringChars(this,str,chars);

}

void ReleaseStringUTFChars( jstring str, const char\* chars) {

functions->ReleaseStringUTFChars(this,str,chars);

}

1)、参数

第一个参数传入一个指向 Java 中的 String 对象的 jstring 变量。

第二个参数传入 jboolean 指针,标示是否对 Java 的 String 对象进行了 Copy。也就是会给该指 针指向的内存传入 JNI\_TRUE 或 JNI\_FALSE 标示是否进行了拷贝、NULL表示不关心是否拷贝，就不想内存赋值。

2)、这两个函数可以两种方式进行字符串的 Copy

开辟新内存，Java 中的 String 拷贝到新内存中，最后返回指向这个内存地址的指针。

直接返回指向 Java 中 String 的内存指针，此时 千万不能改变这个内存的内容，这会破坏String 在 Java 中始终是常量的原则(我们都知道 Java 中 String 为不可变对象,一旦被创建, 就不能修改它的值。需要修改则需要重新创建对象 )。

3)、释放拷贝的内存

在 取 得 字 符 串 后 ， 当 已 经 不 再 之 用 ， 需 要 通 过 ReleaseStringChars 、ReleaseStringUTFChars 来释放拷贝所用的内存或释放Java 的String 对象的引用。(上面代码中)

参数一：指定 jstring 变量—>本地要释放的字符串来源。

参数二：释放本地字符串。

2、GetStringCritical 与 ReleaseStringCritical

为了增加直接传回 Java 字符串的指针的可能性( 不是拷贝)JDK1.2 之后增加

GetStringCritical、ReleaseStringCritical

const jchar \* GetStringCritical( jstring string, jboolean \*isCopy) {

return functions->GetStringCritical(this,string,isCopy);

}

void ReleaseStringCritical( jstring string, const jchar \*cstring) {

functions->ReleaseStringCritical(this,string,cstring);

}

这两个函数之间是一个关键区，在这个关键区中不能调用 JNI 的其它函数或可能造成当前线程中断、等待的任何本地代码 , 否则将造成关键区代码执行期间垃圾回收器停止运作，任何触发垃圾回收器的线程也将暂停，其它触发垃圾回收器的线程不能前进，直到当前线程结束来激活垃圾回收器关键区中不能出现中断操作，不能在 JVM 中分配新对象，否则造成 JVM 死锁。

虽然增加了直接传回 Java 字符串指针的可能性，不过还是需要根据实际情况传回拷贝过的字符串，如 Java 字符串是 UTF-16 ，要想转成 UTF-8 编码还是需要进行拷贝，所以也就没有 GetStringUTFCritical 这个函数。

3、GetStringRegion 与 GetStringUTFRegion

Java 1.2 中新增函数，可以实现把 Java 字符串直接拷贝到 C++的字符串数组中，在调用这个函数之前必须有一个 C++分配出来的字符串，然后通过这个函数进行字符串的拷贝。

优点：函数拷贝不需要分配内存，Java 中的 String 内容的拷贝的开销可以忽略，所以不会造成 OutOFMemoryError 异常。

void GetStringRegion( jstring str, jsize start, jsize len, jchar \*buf) {

functions->GetStringRegion(this,str,start,len,buf);

}

void GetStringUTFRegion( jstring str, jsize start, jsize len, char \*buf) {

functions->GetStringUTFRegion(this,str,start,len,buf);

}

将指向 Java 字符串的 jstring 以相应编码 截取之后 传入 C++分配出来存储字符串的空间 buf。

四.三 C++操作 Java 字符串实例

前面一篇讲了 C++字符串三种方式，下面就给出这三种方式的例子通过 C++代码将 Java 控制台输入的字符串倒序输出到 Java 控制台：

Java 代码都是一样的

public class TestNative {

    public native void sayHello(); // C++本地代码实现

    public String message=null;

    public static void main(String[] args) throws IOException {

        System.loadLibrary("NativeCode");// 加载动态链接库l

        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

        String str = reader.readLine();

        TestNative test = new TestNative(); test.message=str;

        test.sayHello();

        System.out.println("Java Output: "+test.message);

    }

}

1、GetStringChars

#include "com\_jouav\_jni\_TestNative.h"

#include <iostream>

using namespace std;

//对 com\_jouav\_jni\_TestNative.h 中声明的方法定义

JNIEXPORT void JNICALL Java\_com\_jouav\_jni\_TestNative\_sayHello (JNIEnv \*env, jobject obj)

{

  jclass clazz\_TestNative = env->GetObjectClass(obj);

  jfieldID  fid\_msg=

  env->GetFieldID(clazz\_TestNative,"message","Ljava/lang/String;");// 注意分号

  jstring j\_msg=( jstring) env->GetObjectField(obj,fid\_msg);

  const jchar\* jstr=env->GetStringChars( j\_msg,NULL);//UTF-16 编码的宽字符串( jchar\*)

  wstring wstr(((const wchar\_t\*)jstr));

  env->ReleaseStringChars( j\_msg,jstr);//释放

  reverse(wstr.begin(),wstr.end());//进行字符串的倒序

  jstring j\_new\_str=env->NewString((const jchar\*)wstr.c\_str(),( jint)wstr.size());//创建新字符串对象

  env->SetObjectField(obj,fid\_msg,j\_new\_str); //对创建的字符串进行赋值

}

注：

wstring 是宽 char，Unicode 编码，一个字符占两个字节大小

string 是窄 char，ASCII 编码，一个字符占一个字节大小

string/wstring，它们两分别对应着 char 和 wchar\_t。

2、GetStringCritical

效果和第一种是一样的，只是将调用方法和释放方法改变

JNIEXPORT void JNICALL Java\_com\_jouav\_jni\_TestNative\_sayHello (JNIEnv \*env, jobject obj)

{

  jclass clazz\_TestNative = env->GetObjectClass(obj);

  jfieldID  fid\_msg=

  env->GetFieldID(clazz\_TestNative,"message","Ljava/lang/String;");

  jstring j\_msg=( jstring) env->GetObjectField(obj,fid\_msg);

  const jchar\* jstr=env->GetStringCritical( j\_msg,NULL);

  wstring wstr(((const wchar\_t\*)jstr));

  env->ReleaseStringCritical( j\_msg,jstr);

  reverse(wstr.begin(),wstr.end());

  jstring j\_new\_str=env->NewString((const jchar\*)wstr.c\_str(),( jint)wstr.size());

  env->SetObjectField(obj,fid\_msg,j\_new\_str);

}

3、GetStringRegion

JNIEXPORT void JNICALL Java\_com\_jouav\_jni\_TestNative\_sayHello (JNIEnv \*env, jobject obj)

{

  jclass clazz\_TestNative = env->GetObjectClass(obj);

  jfieldID  fid\_msg=

  env->GetFieldID(clazz\_TestNative,"message","Ljava/lang/String;");

  jstring j\_msg=( jstring) env->GetObjectField(obj,fid\_msg);

  jsize len=env->GetStringLength( j\_msg);

  jchar\* jstr=new jchar[len+1]; //这里需要注意，C++中字符串以'\0'结尾jstr[len]=L'\0'; //宽字符串

  env->GetStringRegion( j\_msg,0,len,jstr);

  wstring wstr(((const wchar\_t\*)jstr));

  delete [] jstr; //释放分配出来存储字符串的空间

  reverse(wstr.begin(),wstr.end());

  jstring j\_new\_str=env->NewString((const jchar\*)wstr.c\_str(),( jint)wstr.size());

  env->SetObjectField(obj,fid\_msg,j\_new\_str);

}

五. C++处理 Java 的数组

数组的处理分为以下两种方式

1. 基本类型

2. 指向Java类型的引用对象类型(Object[])的数组

通用于两种类型的数组的函数 GetArrayLength(jarray array)获取数组的长度。

处理基本类型数组

1. Get<TYPE>ArrayElements(<TYPE>Array array, jboolean\* isCopy)，此函数可以把Java基本类型数组转换到C/C++中的数组，两种方式：

第一种：拷贝一份传回本地代码

第二种：把指向 Java 数组的指针直接传回到本地代码，处理完本地化的数组，通过Release<TYPE>ArrayElements 释放数组。

2. Release<TYPE>ArrayElements(<TYPE>Array arr, <TYPE>\* elems, jint mode)，可以选择如何处理Java和C++的数组：提交、撤销、内存释放、不释放等mode 取值：

0 -> 对Java的数组更新并释放 C++数组

JNI\_CONNIT -> 对Java的数组进行更新但不释放C++数组

JNI\_ABORT -> 对Java数组不进行更新，释放C++数组。

3. GetPrimitiveArrayCritical与ReleasePrimitiveArrayCritical

JDK1.2 之后，增加直接传回指向Java数组的指针而加入的函数，同样也会有同GetStringCritical 的死锁的问题。

4.1 Get<TYPE>ArrayRegion(<TYPE>Array array, jsize start, jsize len, <TYPE>\* buf); 在 C++ 中开辟一段内存，然后把 Java 基本类型的数组拷贝到这段内存类似GetStringRegion。

4.2 Set<TYPE>ArrayRegion(<TYPE>Array array, jsize start, jsize len, const <TYPE>\* buf); 把 Java 基本类型的数组中的指定范围的元素用 C++的数组中的元素来赋值。

5. <TYPE>Array New<TYPE>Array( jsize len)

指定一个长度然后返回相应的 Java 基本类型的组。

处理对象类型数组

1. Get/SetObjectArrayElements

JNI没有提供直接把Java的对象类型数组(Object[])直接转到jobject[]数组的函数，而是直接通过Get/SetObjectArrayElements这样的函数来对Java的Object[]数组进行操作，取回的位 jobject 对象，不用释放任何资源。

jobject GetObjectArrayElement( jobjectArray array, jsize index) {

return functions->GetObjectArrayElement(this,array,index);

}

void SetObjectArrayElement( jobjectArray array, jsize index,jobject val) {

functions->SetObjectArrayElement(this,array,index,val);

}

2. NewObjectArray

可以通过指定长度跟初始值来创建某个类的数组

操作基本类型实例，通过C++代码获取Java中数组并对数组排序更新，在Java端输出。

Java 代码

package com.jouav.jni;

public class TestNative {

    int[] arrays={1,2,9,4,6,7,8,5,3,2};

    public native void sayHello(); // C++本地代码实现

    public static void main(String[] args){

        System.loadLibrary("NativeCode");// 加载动态链接库，不能加 .dll

        TestNative test = new TestNative();

        test.sayHello();

        for(int each : test.arrays){

            System.out.println(each);

        }

    }

}

C++代码

#include "com\_jouav\_jni\_TestNative.h"

#include <iostream>

#include <algorithm>

using namespace std;

//对 com\_jouav\_jni\_TestNative.h 中声明的方法定义

JNIEXPORT void JNICALL Java\_com\_jouav\_jni\_TestNative\_sayHello (JNIEnv \*env, jobject obj)

{

  jclass clazz\_TestNative = env->GetObjectClass(obj);

  jfieldID fid\_arrays= env->GetFieldID(clazz\_TestNative,"arrays","[I"); //根据签名获取 ID

  jintArray jint\_arr=( jintArray) env->GetObjectField(obj,fid\_arrays); //根据 ID 获取数组变量

  jint\* int\_arr= env->GetIntArrayElements( jint\_arr,NULL);  //Java 数组的地址直接传回到 C++中

  jsize len= env->GetArrayLength( jint\_arr);  //获取数组长度

  /\*for (int i = 0; i < len; i++)

  {

  cout<<int\_arr[i]<<endl;

  }\*/

  sort(int\_arr,int\_arr+len);  //排序，int\_arr 指针变量指向的起始地址

  env->ReleaseIntArrayElements( jint\_arr,int\_arr,0);

}

六.一 JNI 中的三种引用

局部引用、全局引用、弱全局引用

1. 从Java虚拟机创建的对象传到C/C++代码时会产生引用，根据Java的垃圾回收机制，只要有引用存在就不会触发该引用指向的Java对象的垃圾回收。

2. 这些引用在JNI中分为3种：

全局引用 (Global Reference)

局部引用 (Local Reference)

弱全局引用 (Week Global Reference) since JDK1.2

**局部引用**

最常见的引用类型，基本上通过 JNI 返回来的引用都是局部引用,例如使用 NewObject 就会返回创建出来的实例的局部引用， 局部引用只在该 native 函数中有效，所有在该函数中产生的局部引用，都会在函数返回的时候自动释放(freed)，也可以使用 DeleteLocalRef 函数手动释放该引用。之所以用 DeleteLocalRef 函数：实际上局部引用存在，就会防止其指向的对象被垃圾回收，尤其是当一个局部变量引用指向一个很庞大的对象，或是在一个循环中生成了局部引用，最好的做法就是在使用完该对象后，或在该循环尾部把这个引用释放掉，以确保在垃圾回收器被触发的时候被回收。

在局部引用的有效期中，可以传递到别的本地函数中，要强调的是它的有效期仍然只是在第一次的 Java 本地函数调用中，所以千万不能用 C++全局变量保存它或是把它定义为 C++ 静态局部变量。

**全局引用**

全局引用可以跨越当前线程，在多个 native 函数中有效，不过需要编程人员手动释放该引用。全局引用存在期间会防止在 Java 的垃圾回收器的回收，与局部引用不同，全局引用的创建不是由 JNI 自动创建的，全局引用时需要 NewGlobalRef函数，而释放它需要使用 ReleaseGlobalRef。

**函数弱全局引用**

JDK 1.2 新增加功能，与全局引用相似，创建跟删除都需要由编程人员来进行，这种引用与全局引用一样可以在多个本地代码有效，不一样的是，弱全局引用将不会阻止垃圾回收器回收这个引用所指向的对象，所以在使用时需要多加小心，它所引用的对象可能是不存在的或者已经被回收。使用 NewWeakGlobalRef、ReleaseWeakGlobalRef 来产生和解除引用。

引用相关函数

jobject NewGlobalRef( jobject obj)

jobject NewLocalRef( jobject obj)

jobject NewWeakGlobalRef( jobject obj)

void DeleteGlobalRef( jobject obj)

void DeleteLocalRef( jobject obj)

void DeleteWeakGlobalRef( jobject obj)

jboolean IsSameObject( jobject obj1,jobject obj2)

IsSameObject 可以用于判断两个引用是否指向同一个 Java 对象。

注：对于弱全局引用来说，可以传入一个 NULL 来判断弱全局所指向的 Java 对象是否被回收

六.二 缓存 jfieldID、jmethodID

取得 jfieldID 和 jmethodID 的时候通过该属性/方法名称加上签名来查询相应的 ID，这种查询相对来说开销较大。我们可以将这些 ID 缓存起来，这样只需要查询一次，以后使用已经缓存的ID。

两种缓存方式

1. 在第一次使用的时候缓存 (Caching at the Point of Use)

在native code 中使用static 局部变量来保存已经查询过得id.这样就不会在每次的函数调用时查询，只要在查询成功后就保存起来了,在这种情况下就不得不考虑多线程同时调用此函数时可能会招致同时查询的危机，不过这种方式是无害的，因为查询同一个属性/方法的ID 通常返回的是一样的值，下面是一个简单的代码：

//对 com\_jouav\_jni\_TestNative.h 中声明的方法定义

JNIEXPORT void JNICALL Java\_com\_jouav\_jni\_TestNative\_sayHello (JNIEnv \*env, jobject obj)

{

  static jfieldID fieldID\_string=NULL;

  jclass clazz=env->GetObjectClass(obj);

  if (fieldID\_string==NULL)

  {

    fieldID\_string=env->GetFieldID(clazz,"string","Ljava/lang/String;");

  }

}

2.在 Java 类初始化时缓存 (Caching at the Defining Class’s Initializer)

更好的一个方式就是在任何 native 函数调用前把 id 全部存起来我们可以让 Java 在第一次加载这个类的时候首先调用本地代码初始化所有的 jfieldID/jmethodID，这样的话就可以省去多次的确定 ID 是否存在的语句，当然这些 jfieldID/jmethodID 是定义在C/C++的全局。使用这种方式的好处，当 Java 类卸载或是重新加载的时候也会重新调用本地代码来重新 计 算 IDs Java 不完整代码：

package com.jouav.jni;

public class TestNative {

static{

sayHello();

}

static native void sayHello(); // C++本地代码实现int propInt=0;

String propStr="";

public native void otherNative();

//other code...

}

C++ 不 完 整 代 码 ：

jfieldID g\_propInt\_id=0; jfieldID g\_propStr\_id=0;

//对 com\_jouav\_jni\_TestNative.h 中声明的方法定义

JNIEXPORT void JNICALL Java\_com\_jouav\_jni\_TestNative\_sayHello (JNIEnv \*env, jobject obj)

{

jclass clazz=env->GetObjectClass(obj); g\_propInt\_id=env->GetFieldID(clazz,"propInt","I");

g\_propStr\_id=env->GetFieldID(clazz,"propStr","Ljava/lang/String;");

}

JNIEXPORT void JNICALL Java\_com\_jouav\_jni\_TestNative\_otherNative (JNIEnv \*env, jobject obj)

{

//get field with g\_propInt\_id/g\_propStr\_id....

}

最后

还需要学习的知识： 异常处理

C/C++如何启动 JVM

JNI 跟多线程

推荐的书籍读物：

The Java Native Interface Programmer’s Guide and Specification JNI++ User Guide