Java Native Interface 应用研究*)

许晓宁
(扬州职业大学计算机科学系　扬州 225002)

摘　要　本文研究在 Java 虚拟机下加快系统的执行速度，实现调用本地操作系统的内核，访问硬件设备接口，执行非 Java 代码集及临界代码段等技术。通过这些技术的实现，在保持 Java 平台无关性的同时又充分发挥了本地平台的优势。

关键词　Java，JNI，Java 方法，JVM

Java Native Interface Application Research

XU Xiao-Ning
(Department of Computer Scienie, Yangzhou Polytechnic University, Yangzhou225009)

Abstract　The characteristic that Java is with its cross-platform deep suffers people to like, but again positive because of the purpose of its cross-platform, make it to contact with the every kind of inner part of the native machine to become few, controlled its function . This text research is under the Java Virtual Machine quickly implementing, realizing of access operating-system-specific features, interface with special hardware devices, reuse a preexisting, non-Java code base, or implement time-critical sections of code. Pass realizing of these techniqueses, it is important to keeping Java platform irrelevant and to develop the advantage of the native platformed at the same time.

Keywords　Java, JNI, Java method, JVM

1　引言

JavaSoft 公司提供的 Java Native Interface(JNI)是 Java JDK 的本地编程接口。JNI 允许运行在 Java 虚拟机中的 Java 代码调用其他用 C、C++、汇编语言编写的应用代码和库函数，还可通过引用 API 将 Java 虚拟机嵌入本地平台应用中。使用 JNI 编写的程序代码将保证跨越所有平台，充分发挥本地平台的特性。

程序员采用 JNI 编写 native 方法实现下述应用：(1)标准 Java 类库无法支持的特定平台的某些应用；(2)在 Java 虚拟机中继续使用早期开发的非 Java 代码应用系统；(3)在 Java 中调用汇编语言编写的临界代码。

在 JNI 框架里可以用 native 方法使用 Java 对象，而这种方式与直接用 Kava 代码处理对象具有相同的效果。一个 native 方法可以创建 Java 对象，观察并使用对象完成其任务。一个 native 方法还可以观察和使用 Java 应用程序创建的对象。因此，本地语言和 Java 语言都能创建、更新、访问与共享 Java 对象。

JNI 可以在 native 方法中实现 Java 编程语言的所有特点。例如，native 方法能够捕捉、抛出 Java 例外，还可实现在 Java 应用程序中的例外处理。通过调用 JNI 函数，native 方法可装载 Java 类并获取 Java 类信息。

2　Java 应用程序中使用 native 方法

(1)编写一段 Java 程序，创建一个声明 native 方法的类。这个类中包括 native 方法的声明和 native 代码，以及调用 native 方法的 main 方法。代码段如下所示：

```
public class ApiCalling{
  private native void ApiCalling (String msg);
  static{
    System. loadLibrary("MsgImpl");
      }
  pustatic void main(String [] args){
    ApiCalling app = new ApiCalling ();
    app. ApiCalling ("Generated with JNI");
      }
}
```

Native 方法调用 System. loadLibrary()完成动态链接库的装载和连接。必须指定动态连接库(DLL)所在目录，其扩展名视 JVM 所在平台自动生成。

(2)将该类源文件用 Java 类编译器编译成二进制字节码文件。由于采用了 native 关键字声明，编译器会忽视没有代码体的 JNI 方法部分。

(3)编译生成 ApiCalling 类后，运行以下命令：

Javah jni ApiCalling

Javah 命令读取 Java 类文件中的每一个 native 方法声明，生成一个 C 或 C++ 函数原型 ApiCalling. h。

```
#include <jni. h>
/* Header for class ApiCalling */
#ifndef _Included_ ApiCalling
#define _Included_ ApiCalling
#ifdef __cplusplus
extern C"{
    #endif
/* CApiCalling
     Method: ApiCalling */
JNIEXPORT void JNICALL
Java_ ApiCalling _ ApiCalling
  (JNIEnv *, jobject, jstring);
#ifdef __cplusplus
}
#endif
#endif
```

当出现 #ifdef __cplusplus 与处理程序指示时，表示这

*)江苏省高校自然科学研究基金资助项目(No. 05KJD520270)。许晓宁　副教授，主要研究方向：计算机网络与 Web 应用开发，虚拟现实技术应用。

个文件可以用 C 或 C++ 编译器进行编译。#include 语句指明包含的 jni.h 头文件定义了可使用的数据类型。JNIEXPORT 和 JNICALL 是匹配特殊平台的宏定义，JNIenv、jobject 和 jstring 是宏中定义的 JNI 数据类型。

(4)根据头文件编写相应方法的实现代码。在编码过程中，需要注意变量的长度问题，例如 Java 的整型变量长度为 32 位，而 C 语言为 16 位，所以要仔细核对变量类型映射表，防止在传值过程中出现问题。

(5)在 Win32 环境下，可以利用 Visual C++ 或其他能产生 DLL 文件的 C 或 C++ 编译器将 JNI 实现代码编译成动态链接库。

DLL 是依赖于某一特定平台的，下述代码段在 Windows 下编译连接成 ApiCalling.dll 文件，在 Unix/Linux 下编译连接成 ApiCalling.so 文件。

```
#include <jni.h>
#include <stdio.h>
#include "ApiCalling.h"
extern "C" JNIEXPORT void JNICALL
Java_ApiCalling_ApiCalling (JNIEnv * env,
jobject, jstring jMsg){
  const char * msg=env->GetStringUTFChars(jMsg,0);
  printf("Java Native Interface JNI: %s\n", msg);
  env->ReleaseStringUTFChars(jMsg, msg);
}
```

上述 native 方法中的参数能通过网络返回 Java。JNIEnv 包含了所有调用返回 JVM 的陷阱。jobject 参数相对于不同类型的方法有不同的含义，它涉及到调用 native 方法的对象，在上述非 static 方法例子中，该参数等效于 C++ 中的"this"参数，类似于 Java 中的"this"关键字，对于 static 方法，该参数涉及到方法被执行的类对象。

3 Native 平台与 Java 的整合

JNI 提供了一系列标准接口函数，可以通过在 native 方法中调用 JNI 函数，实现访问 Java 对象、释放 Java 对象、创建新对象以及调用 Java 方法等操作。

(1)利用 JNIEnv 参数访问 JNI 函数

JNI 函数是程序员在 JVM 中的 native 方法中定义与使用的，上例说明 JNIEnv 参数指向一个 JNIEnv 类型的 JNI 数据结构，一个 JNI 数据结构的元素就是指向 JVM 产生的矩阵的指针，矩阵中的每一个元素指向一个 JNI 函数。也可以使用间接指针从 native 方法中调用 JNI 函数。

通过 JNIEnv 参数，程序员可以访问大量 JNI 函数集，这些函数可以被归并为下属各类：

- 获取版本信息；
- 执行类和对象的操作；
- 处理引用全局和局部的 Java 对象；
- 访问实例属性和静态属性；
- 调用实例方法和静态方法；
- 执行串和矩阵操作；
- 产生和处理 Java 例外；

(2)从 native 代码中调用 Java 方法

例程 Callbacks.Java 如下：

```
class Callbacks{
   private native void nativeMethod(int depth);
   private void callback(int depth){
      if(depth<5){
        System.out.println("In Java,depth= "+depth+",
about to enter C");
        nativeMethod(depth+1);
        System.out.println("In Java,depth="+depth+",
back from C");
      }else
      System.out.println("In Java,depth="+depth+",limit
exceeded");
   }
   public static void main(String args []){
    Callbacks c = new Callbacks();
    c.nativeMethod(0);
      }
   static {
    System.loadLibrary("MyImpOfCallbacks");
   }
}
```

该 Java 程序包含了一个 native 方法，native 方法的功能为调用返回 Java 方法。

通常可通过以下步骤调用实例方法：

①在 native 方法中调用 JNI 函数 GetObjectClass，它返回 Java 类对象。

②然后，在 native 方法中调用 JNI 函数 GetMethodID，它完成在给定类中查找 Java 方法。这个查找是基于方法标识或方法素质签名的。如果方法不存在，GetMethodID 返回 0。

③最后，在 native 方法中调用 JNI 函数 CallVoidMethod，CallVoidMethod 函数包括一个无返回值类型的实例方法，它接收对象、方法 ID 和实际参数。

(3)使用方法的 ID 号调用 Java 方法

当 JNI 包含方法时，须将方法 ID 传递给实际方法引用的函数。因为获取一个方法 ID 要分别针对不同的方法引用，所以其操作是相对复杂的。但方法 ID 一经获取便可多次引用。

只要相应的 Java 类未被卸载，方法 ID 就一直保持有效。若想捕捉方法 ID，就要确信保持相对于 Java 类的生存周期，只要相应的 Java 类的生存周期存在，native 代码保持对该类的有效性。

(4)调用类方法

从 native 代码中调用 Java 类方法类似于调用实例方法，其步骤如下：

(5)使用 JNI 函数 GetStaticMethodID 获取方法 ID。

(6)传递类、方法 ID 和参数给类方法引用函数：GetStaticMethodID，CallStaticBooleanMethod 等。

比较实例方法引用函数与类方法引用函数，我们将注意到实例方法引用函数接收的是对象而不是类。

4 调用 Java 虚拟机

Java 虚拟机嵌入 native 应用就使 native 应用具有了虚拟机的完全功能。如果 native 应用是 Web 浏览器，那么浏览器就支持 Java applets 的执行。JDK 发送 JVM 作为共享库或 win32 下的 DLL，通过 native 应用与共享库链接的方法将 JVM 嵌入 native 应用。

JNI 支持引用 API 加载、初始化、调用 Java 虚拟机，通过引用 API 解析命令行参数及 Java 虚拟机。下例的 C 程序实现调用 JVM 和定义在 Prog.Java 中的 Prog.main 方法：

```
/* invoke.c */
#include <jni.h>
#ifdef WIN32
#define PATH_SEPARATOR ';'
#else /* UNIX */
#define PATH_SEPARATOR ':'
#endif
#define USER_CLASSPATH "." /* where Prog.class is */
main () {
     JNIEnv *env;
     JavaVM *jvm;
      JDK1_1InitArgs vm_args;
      jint res;
      jclass cls;
```

(下转第封三页)

上例经过上述算法(1)由表1可构造如下集合(见表2和表3)：

表2 集合A

W(1,3)=θ	W(7,8)=θ
W(1,2)=θ	W(9,11)=θ
W(2,5)=θ	W(9,13)=θ
W(4,2)=θ	W(10,8)=θ
W(4,3)=θ	W(10,9)=θ
W(6,7)=θ	W(12,10)=θ
W(7,9)=θ	W(14,10)=θ

表3 集合B

W(8,Y)=E
W(3,4)=a
W(5,6)=b
W(11,12)=a
W(13,14)=c

上述结果经过上述算法(2)可得到终态集合C(表4)和DFA图，如图2。

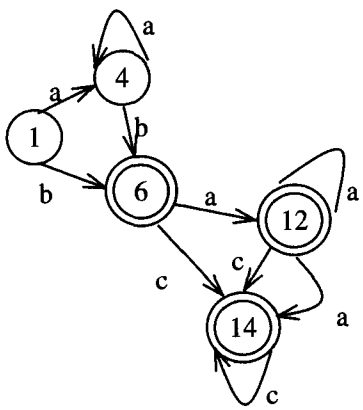

图2 DFA图

表4 集合C

W(1,4)=a	W(12,12)=a
W(1,6)=b	W(12,14)=c
W(4,4)=a	W(12,Y)=E
W(4,6)=b	W(14,12)=a
W(6,Y)=E	W(14,14)=c
W(6,12)=a	W(14,Y)=E
W(6,15)=c	

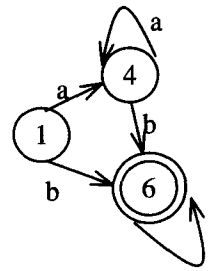

图3 最简DFA图

经过上述算法(3)可得到表达式 $a^* \cdot b \cdot (a|c)^*$ 的最简DFA图，如图3。

结束语 有限状态自动机的并行确定化算法的设计及过程分析有其理论和实践意义，其算法思想及表列中间存储结构也可用于类似的并行转换。在并行环境中，多机的并行结构模型和中间存储结构都直接影响算法的运行结果。此外，并行确定化转换算法并未被证明是最优的，这方面还有许多工作可做，优化和存储结构的改进以及与不同并行结构模型的匹配等需要进一步的研究与讨论。

参 考 文 献

1 陈火旺，刘春林，谭庆平，等. 程序设计编译原理. 北京：国防工业出版社，2003

2 Gibbons A, Rytter W. Efficient parallel algorithms. Cambridge University Press ,1990

3 Ra Dong-Yul, Kim Jong-Hyun. A parallel parsing algorithm for arbitrary contexe-free grammars. Information Processing Letters, 1996,58:87～96

4 Matsumoto Y. A parallel parsing system for natural language analysis. New Generation Comput,1987,15:63～78

5 Wyard P J, Ninghtingale C. A single layer higher order neural net and its application to context-free grammar recognition. In: Sharkey N, ed

6 陈国良. 并行计算—结构、算法、编程. 北京：高等教育出版社，1999

(上接第292页)

```
    ……
    JNI_GetDefaultJavaVMInitArgs(&vm_args);
        sprintf (classpath, "%s%c%s",
                 vm_args. classpath, PATH_SEPARATOR,
USER_CLASSPATH);
    vm_args. classpath = classpath;
    /* Create the Java VM */
    res= JNI_CreateJavaVM(&jvm,&env,&vm_args);
    if (res < 0) {
        fprintf(stderr, "Can't create Java VM\n");
        exit(1);
    }
     cls = (*env)->FindClass(env, "Prog");
    ……
     (*env)->CallStaticVoidMethod(env, cls, mid, args);
     (*jvm)->DestroyJavaVM(jvm);
}
//Prog. Java
public class Prog {
    public static void main(String [] args) {
        System. out. println("Hello World" + args [0]);
}
}
```

invoke. c代码开始调用JNI_GetDefaultJavaVMInitArgs获得初始化设置，接着调用JNI_CreateJavaVM加载和初始化虚拟机。在JNI_CreateJavaVM成功返回之后，当前native线程已将自身捆绑在Java虚拟机上，但无结果返回JVM。

结束语 本文围绕在Java中调用native方法，native方法与Java的整合，在native方法中调用Java虚拟机等方面阐述了技术实现方案，使Java通过JNI调用本地的库文件的内部方法，实现和本地机器的紧密联系，调用系统级的各接口。

Java JNI技术的不断完善给基于Java技术的应用系统开发带来了便利，随着Java软件技术的不断提高，“一次编写，多处可用”的理念将会发挥巨大作用。

参 考 文 献

1 飞思科技产品研发中心. JavaWeb服务应用开发指南. 电子工业出版社，2002

2 靳慧峰，等. 新概念JSP网络应用. 北京科海集团公司，2001

3 王立冬，张凯. Java虚拟机分析. 北京理工大学学报，2002(8)

4 陈传波，蒋湘. 网络管理中JAVA技术应用的探讨. 湖北工学院学报，2001(3)