B: El Interfaz Nativo Java (JNI[1])

El material de este apéndice se incorporó y usó con el permiso de Andrea Provaglio (www.AndreaProvaglio.com).

El lenguaje Java y su API estándar son lo suficientemente ricos como para escribir aplicaciones completas. Pero en ocasiones hay que llamar a código no-Java; por ejemplo, si se desea acceder a aspectos específicos del sistema operativo, interactuar con dispositivos hardware especiales, reutilizar código base preexistente no-Java, o implementar secciones de código críticas en el tiempo.

Interactuar con código no-Java requiere de soporte dedicado por parte del compilador y de la Máquina Virtual, y de herramientas adicionales para establecer correspondencias entre el código Java y el no-Java. La solución estándar para invocar a código no-Java proporcionada por JavaSoft es el *Interfaz Nativo Java*, que se presentará en este apéndice. No se trata de ofrecer aquí un tratamiento en profundidad, y en ocasiones se asume que uno tiene un conocimiento parcial de los conceptos y técnicas involucrados.

JNI es una interfaz de programación bastante rica que permite la creación de métodos nativos desde una aplicación Java. Se añadió en Java 1.1, manteniendo cierto nivel de compatibilidad con su equivalente en Java 1.0: el interfaz nativo de métodos (NMI). NMI tiene características de diseño que lo hacen inadecuado para ser adoptado en todas las máquinas virtuales. Por ello, las versiones futuras del lenguaje podrían dejar de soportar NMI, y éste no se cubrirá aquí.

Actualmente, JNI está diseñado para interactuar con métodos nativos escritos únicamente en C o C++. Utilizando JNI, los métodos nativos pueden:

- Crear, inspeccionar y actualizar objetos Java (incluyendo arrays y **String**s).
- Invocar a métodos Java
- Capturar y lanzar excepciones
- Cargar clases y obtener información de clases
- Llevar a cabo comprobación de tipos en tiempo de ejecución.

Por tanto, casi todo lo que se puede hacer con las clases y objetos ordinarios de Java también puede lograrse con métodos nativos.

---

[1] N. del traductor: En inglés: *Java Native Interface.*

Invocando a un método nativo

Empezaremos con un ejemplo simple: un programa en Java que invoca a un método nativo, que de hecho llama a la función ejemplo **printf( )** de la biblioteca estándar de C.

El primer paso es escribir el código Java declarando un método nativo y sus argumentos:

```
//: apendiceb:MostrarMensaje.java
public class MostrarMensaje {
  private native void MostrarMensaje(String msj);
  static {
    System.loadLibrary("MsgImpl");
    // Truco Linux, si no puedes acceder a la ruta de tu biblioteca
    // configura tu entorno:
    // System.load(
    //   "/home/bruce/tij2/appendixb/MsgImpl.so");
  }
  public static void main(String[] args) {
    MostrarMensaje app = new MostrarMensaje();
    app.MostrarMensaje("Generado con JNI");
  }
} ///:~
```

A la declaración del método nativo sigue un bloque **static** que invoca a **System.loadLibrary( )** (a la que se podría llamar en cualquier momento, pero este estilo es más adecuado). **System.loadLibrary( )** carga una DLL en memoria enlazándose a ella. La DLL debe estar en la ruta de la biblioteca del sistema. La JVM añade la extensión del nombre del archivo automáticamente en función de la plataforma.

En el código de arriba también se puede ver una llamada al método **System.load( )**, marcada como comentario. La trayectoria especificada en este caso es absoluta en vez de radicar en una variable de entorno. Naturalmente, usar ésta última aporta una solución más fiable y portable, pero si no se puede averiguar su valor se puede comentar **loadLibrary( )** e invocar a esta línea, ajustando la trayectoria al directorio en el que resida el archivo en cada caso.

El generador de cabeceras de archivo: javah

Ahora, compile el archivo fuente Java y ejecute **javah** sobre el archivo **.class** resultante, especificando la opción **-jni** (el *makefile* que acompaña a la distribución de código fuente de este libro se encarga de hacer esto automáticamente):

```
javah -jni MostrarMensaje
```

javah lee el archivo de clase Java y genera un prototipo de función en un archivo de cabecera C o C++ por cada declaración de método nativo. He aquí la salida: el archivo fuente **MostrarMensaje.h** (editado de forma que entre en este libro):

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

#ifndef _Included_MostrarMensaje
#define _Included_MostrarMensaje
#ifdef _cplusplus
extern "C" {
#endif
/*
 * Class:        MostrarMensaje
 * Method:       MostrarMensaje
 * Signature:    (Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL
Java_MostrarMensaje_MostrarMensaje
   (JNIEnv *, jobject, jstring);

#ifdef _cplusplus
}
#endif
#endif
```

Como puede verse por la directiva del preprocesador **#ifdef__cplusplus**, este archivo puede compilarse con un compilador de C o de C++. La primera directiva **#include** incluye **jni.h**, un archivo de cabecera que, entre otras cosas, define los tipos que pueden verse utilizados en el resto del archivo. **JNIEXPORT** y **JNICALL** son macros que se expanden para hacer coincidir directivas específicas de la plataforma. **JNIEnv**, **jobject** y **jstring** son definiciones de tipos de datos JNI, que se explicarán en breve.

Renombrado de nombres y signaturas de funciones

JNI impone una convención de nombres (denominada *renombrado de nombres*) a los métodos nativos. Esto es importante, pues es parte del mecanismo por el que la máquina virtual enlaza las llamadas a Java con métodos nativos. Básicamente, todos los métodos nativos empiezan con la palabra "Java", seguida del nombre de la clase en la que aparece la declaración nativa Java, seguida del nombre del método Java. El carácter "guión bajo" se usa como separador. Si el método nativo Java está sobrecargado, se añade también la signatura de la función al nombre; puede verse la signatura nativa en los comentarios que preceden al prototipo. Para obtener más información sobre *renombrado de nombres* y signaturas de métodos nativos, por favor acudir a la documentación JNI.

Implementando la DLL

En este momento, todo lo que hay que hacer es escribir archivos de código fuente en C y C++ que incluye el archivo de cabecera generado por **javah**, que implementa el método nativo, después compilarlo y generar una biblioteca de enlace dinámico. Esta parte es dependiente de la plataforma. El código de debajo se compila y enlaza a un archivo que en Windows se llama **MsgImpl.dll** y en Unix/Linux **MsgImpl.so** (el *makefile* empaquetado junto con los listados de código contiene los comandos para hacer esto —está disponible en el CD ROM vinculado a este libro, o como descarga gratuita de *www.BruceEcKel.com*):

```
//: apendiceb:MsgImpl.cpp
//# Probado con VC++ & BC++. Hay que ajustar la ruta de
//# los includes para encontrar las cabeceras JNI. Ver
//# el makefile de este capítulo (en el
//# código fuente descargable) si se desea tener un ejemplo.
#include <jni.h>
#include <stdio.h>
#include "MostrarMensaje.b.h"

extern "C" JNIEXPORT void JNICALL
Java_MostrarMensaje_MostrarMensaje(JNIEnv* env,
jobject, jstring jMsg) {
  const char* msg=env->GetStringUTFChars(jMsg,0);
  printf("Piensa en Java, JNI: %s\n", msg);
  env->ReleaseStringUTFChars(jMsg, msg);
} ///:~
```

Los parámetros que se pasan al método nativo son la pasarela que permite la vuelta a Java. El primero, de tipo **JNIEnv**, contiene todos los anzuelos que te permiten volver a llamar a la JVM. (Echaremos un vistazo a esto en la sección siguiente.) El segundo parámetro tiene significado distinto en función del tipo de método. En el caso de métodos *no* **static** como el del ejemplo de arriba, el segundo parámetro es equivalente al puntero "this" de C++ y similar a **this** en Java: es una referencia al objeto que invocó al método nativo. En el caso de métodos **static**, es una referencia al objeto **Class** en el que está implementado el método.

Los demás parámetros representan los objetos Java que se pasan a la llamada al método nativo. También se pasan tipos de datos primitivos así, pero vienen por valor.

En las secciones siguientes, explicaremos este código mirando a las formas de acceder y controlar la JVM desde dentro de un método nativo.

Accediendo a funciones JNI: el parámetro **JNIEnv**

Las funciones JNI son aquéllas que usa el programador para interactuar con la JVM desde dentro de un método nativo. Como puede verse en el ejemplo de arriba, todo método JNI nativo recibe un parámetro especial en primer lugar: el parámetro **JNIEnv**, que es un puntero a una estructura de datos especial de JNI de tipo **JNIENv_**. Un elemento de la estructura de datos JNI es un puntero a un array generado por la JVM. Cada elemento de este array es un puntero a una función JNI. Las funciones JNI pueden ser invocadas desde el método nativo desreferenciando estos punteros (es más simple de lo que parece). Toda JVM proporciona su propia implementación de las funciones JNI, pero sus direcciones siempre estarán en desplazamientos predefinidos.

A través del parámetro **JNIENv**, el programador tiene acceso a un gran conjunto de funciones. Estas funciones pueden agruparse en las siguientes categorías:

- Obtener información de la versión
- Llevar a cabo operaciones de clase y objetos
- Acceder a campos de instancia y campos estáticos
- Llamar a métodos de instancia y estáticos
- Llevar a cabo operaciones de Strings y arrays
- Generar y gestionar excepciones Java

El número de funciones JNI es bastante grande y no se cubrirá aquí. Sin embargo, mostraré el razonamiento que hay tras el uso de estas funciones. Para obtener información más detallada, consulte la documentación JNI del compilador.

Si se echa un vistazo al archivo de cabecera **jni.h**, se verá que dentro de la condicional de preprocesador **#ifdef__cplusplus**, se define la estructura **JNIEnv_** como una clase cuando se compile por un compilador de C++. Esta clase contiene varias funciones que te permiten acceder a las funciones JNI con una sintaxis sencilla y familiar. Por ejemplo, la línea de código C++ del ejemplo anterior:

```
env->ReleaseStringUTFChars(jMsg, msg);
```

También podría haber sido invocada desde C así:

```
(*env)->ReleaseStringUTFChars(env, jMsg, msg);
```

Se verá que el estilo de C es (naturalmente) más complicado —se necesita una desreferencia doble del puntero **env**, y se debe pasar también el nombre del puntero como primer parámetro a la llamada a la función JNI. Los ejemplos de este apéndice usan el ejemplo de C++.

Accediendo a Strings Java

Como ejemplo de acceso a una función JNI, considérese el código de **MsgImpl.cpp**. Aquí, se usa el argumento **JNIEnv env** para acceder a un **String** Java. Éstos están en formato Unicode, por lo que si se recibe uno y se desea pasarlo a una función no-Unicode (**printf( )**, por ejemplo) primero hay que convertirlo a caracteres ASCII con la función JNI **GetStringUTFChars( )**. Esta función toma un **String** Java y lo convierte a UTF de 8 caracteres. (Estos 8 bits son suficientes para almacenar valores ASCII, o 16 bits para almacenar Unicode. Si el contenido de la cadena de caracteres original sólo estaba compuesta de caracteres ASCII, la cadena resultante también estará en ASCII.)

GetStringUTFChars( ) es una de las funciones miembro de **JNIEnv**. Para acceder a la función JNI usamos la sintaxis típica de C++ para llamar a funciones miembro mediante un puntero. La forma de arriba se usa para acceder a todas las funciones JNI.

Pasando y usando objetos Java

En el ejemplo anterior, se pasaba un **String** al método nativo. También se pueden pasar objetos Java de tu creación al método nativo. Dentro de éste, se puede acceder a los campos y métodos del objeto que se recibió.

Para pasar objetos, puede usarse la sintaxis general de Java al declarar el método nativo. En el ejemplo de abajo, **MiClaseJava** tiene un campo **public** y un método **public**. La clase **UsarObjetos** declara un método nativo que toma un objeto de clase **MiClaseJava**. Para ver si el método nativo manipula su parámetro, se pone el campo **public** del parámetro, se invoca al método nativo, y después se imprime el valor del campo **public**:

```
//: apendiceb:UsarObjetos.java
class MiClaseJava {
  public int unValor;
  public void dividirPorDos() { unValor /= 2; }
}

public class UsarObjetos {
  private native void
    cambiarObjecto(MiClaseJava obj);
  static {
    System.loadLibrary("UsarObjImpl");
    // Truco de Linux, si no se puede lograr la ruta de
    // la biblioteca, configure su entorno:
    // System.load(
    //"/home/bruce/tij2/appendixb/UsarObjImpl.so");
  }
  public static void main(String[] args) {
    UsarObjetos app = new UsarObjetos();
    MiClaseJava unObj = new MiClaseJava();
```

```
    unObj.unValor = 2;
    app.cambiarObjeto(unObj);
    System.out.println("Java: " + unObj.unValor);
  }
} ///:~
```

Tras compilar el código y ejecutar **javah**, se puede implementar el método nativo. En el ejemplo de debajo, una vez que se obtienen el campo y el ID del método, se acceden a través de funciones JNI:

```
//: apendiceb:UsarObjImpl.cpp
//# Probado con VC++ & BC++. Hay que ajustar la ruta de
//# los includes para encontrar las cabeceras JNI. Ver
//# el makefile de este capítulo (en el
//# código fuente descargable) si se desea tener un ejemplo.
#include <jni.h>
extern "C" JNIEXPORT void JNICALL
Java_UsarObjetos_cambiarObjeto(
JNIEnv* env, jobject, jobject obj) {
  jclass cls = env->GetObjectClass(obj);
  jfieldID fid = env->GetFieldID(
    cls, "unValor", "I");
  jmethodID mid = env->GetMethodID(
    cls, "dividirPorDos", "()V");
  int valor = env->GetIntField(obj, fid);
  printf("Nativo: %d\n", valor);
  env->SetIntField(obj, fid, 6);
  env->CallVoidMethod(obj, mid);
  valor = env->GetIntField(obj, fid);
  printf("Nativo: %d\n", valor);
} ///:~
```

Ignorando el equivalente "this", la función C++ recibe un **jobject**, que es el lado nativo de la referencia al objeto Java que le pasamos desde el código Java. Simplemente leemos **unValor**, la imprimimos, cambiamos el valor, se llama al método **dividirPorDos( )** del objeto, y se imprime de nuevo el valor.

Para acceder a un campo o método Java, primero hay que obtener su identificador usando **GetFieldID( )** en el caso de los campos y **GetMethodID( )** en el caso de los métodos. Estas funciones toman el objeto clase, una cadena de caracteres que contiene el nombre del elemento, y otra cadena que proporciona información de tipos: el tipo de datos del campo, o información de signatura en el caso de un método (pueden encontrarse detalles en la documentación de JNI). Estas funciones devuelven un identificador que se usa para acceder al elemento. Este enfoque podría parecer complicado, pero el método nativo desconoce la disposición interna del objeto Java. En su lugar, debe acceder a los campos y métodos a través de índices devueltos por la JVM. Esto permite que distintas JVMs implementen disposiciones internas de objetos distintas sin que esto afecte a los métodos nativos.

Si se ejecuta el programa Java, se verá que el objeto pasado desde el lado Java es manipulado por el método nativo. Pero ¿qué es exactamente lo que se pasa? ¿Un puntero o una referencia Java? Y ¿qué hace el recolector de basura durante llamadas a métodos nativos?

El recolector de basura sigue operando durante la ejecución de métodos nativos, pero está garantizado que no eliminará ningún objeto durante la llamada a un método nativo. Para asegurar esto, se crean previamente *referencias locales*, que son destruidas inmediatamente después de la llamada al método nativo. Dado que su tiempo de vida envuelve la llamada, se sabe que los objetos serán válidos durante toda la llamada al método nativo.

Dado que estas referencias son creadas y destruidas subsecuentemente cada vez que se llama a la función, no se pueden hacer copias locales en los métodos nativos, en variables **static**. Si se desea una referencia que perdure a través de invocaciones a funciones, se necesita una referencia global. Éstas no las crea la JVM, pero el programador puede hacer una referencia global a partir de una local llamando a funciones de JNI específicas. Cuando se crea una referencia global, uno es responsable de la vida del objeto referenciado. La referencia global (y el objeto al que hace referencia) estarán en memoria hasta que el programador libere la referencia explícitamente con la función JNI apropiada. Es semejante al **malloc( )** y **free( )** de C.

JNI y las excepciones Java

Con JNI, pueden lanzarse, capturarse, imprimirse y relanzarse excepciones Java exactamente igual que si se estuviera dentro de un programa Java. Pero depende del programador el invocar a funciones JNI dedicadas a tratar las excepciones. He aquí las funciones JNI para la gestión de excepciones:

- **Throw( )**

  Lanza un objeto excepción existente. Se usa en los métodos nativos para relanzar una excepción.

- **ThrowNew( )**

  Genera un nuevo objeto excepción y lo lanza.

- **ExceptionOccurred( )**

  Determina si se lanzó una excepción aún sin eliminar.

- **ExceptionDescribe( )**

  Imprime una excepción y la traza de la pila.

- **ExcepcionClear( )**

  Elimina una excepción pendiente.

- **FatalError( )**

  Lanza un error fatal. No llega a devolver nada.

Entre éstas, no se puede ignorar **ExceptionOcurred( )** y **ExceptionClear( )**. La mayoría de funciones JNI pueden generar excepciones y no hay ninguna faceta del lenguaje que pueda usarse en el lugar de un bloque *try* de Java, por lo que hay que llamar a **ExceptionOccurred( )** tras cada llamada a función JNI para ver si se lanzó alguna excepción. Si se detecta una excepción, se puede elegir manejarla (y posiblemente relanzarla). Hay que asegurarse, sin embargo, de que la excepción sea siempre eliminada. Esto puede hacerse dentro de la función **ExceptionClear( )** o en alguna otra función si se relanza la excepción, pero hay que hacerlo.

Hay que asegurar que se elimine la excepción, pues si no, los resultados serían impredecibles si se llama a una función JNI mientras está pendiente una excepción. Hay pocas funciones JNI que pueden ser invocadas de forma segura durante una excepción; entre éstas, por supuesto, se encuentran las funciones de manejo de excepciones.

JNI y los hilos

Dado que Java es un lenguaje multihilo, varios hilos pueden invocar a métodos nativos de forma concurrente. (El método nativo podría suspenderse en el medio de su operación al ser invocado por un segundo hilo.) Depende enteramente del programador el garantizar que la llamada nativa sea inmune a los hilos; por ejemplo, no modifica datos compartidos de forma no controlada. Básicamente, se tienen dos opciones: declarar el método nativo como **synchronized**, o implementar alguna otra estrategia dentro del método nativo para asegurar una manipulación de datos concurrentes correcta.

También se podría no pasar nunca el puntero **JNIEnv** por los hilos, puesto que la estructura interna a la que apunta está ubicada en una base hilo a hilo y contiene información que sólo tiene sentido en cada hilo en particular.

Usando un código base preexistente

La forma más sencilla de implementar métodos JNI nativos es empezar a escribir prototipos de métodos nativos en una clase Java, compilar la clase y ejecutar el archivo **.class** con **javah**. Pero ¿qué ocurre si se tiene un código base grande preexistente al que se desea invocar desde Java? Renombrar todas las funciones de las DLLs para que coincidan con la convención de nombres de JNI no es una solución viable. El mejor enfoque es escribir una DLL envoltorio "fuera" del código base original. El código Java llamaría a funciones de esta nueva DLL, que a su vez invoca a funciones de la DLL original. Esta solución no es sólo un rodeo; en la mayoría de casos hay que hacerlo así siempre porque hay que invocar a funciones JNI con referencias a objetos antes de su uso.

Información adicional

Puede encontrarse más material introductorio, incluyendo un ejemplo en C (en vez de en C++) y una discusión de aspectos Microsoft en el Apéndice A de la primera edición de este libro, que puede encontrarse en el CD ROM que acompaña a este libro, o como descarga gratuita de

www.BruceEckel.com. Hay información más extensa disponible en *java.sun.com*. (en el motor de búsqueda, seleccione las palabras clave "native methods" dentro de "training & tutorials"). El Capítulo 11 de *Core Java 2, Volume II*, por Horstmann & Cornell (Prentice-Hall, 2000) da una cobertura excelente a los métodos nativos.