Contents | Next

Chapter 1: Introduction

This chapter introduces the *Java Native Interface* (JNI). The JNI is a native programming interface. It allows Java code that runs inside a Java Virtual Machine (VM) to interoperate with applications and libraries written in other programming languages, such as C, C++, and assembly.

The most important benefit of the JNI is that it imposes no restrictions on the implementation of the underlying Java VM. Therefore, Java VM vendors can add support for the JNI without affecting other parts of the VM. Programmers can write one version of a native application or library and expect it to work with all Java VMs supporting the JNI.

This chapter covers the following topics:

Java Native Interface Overview

While you can write applications entirely in Java, there are situations where Java alone does not meet the needs of your application. Programmers use the JNI to write *Java native methods* to handle those situations when an application cannot be written entirely in Java.

The following examples illustrate when you need to use Java native methods:

- The standard Java class library does not support the platform-dependent features needed by the application.
- You already have a library written in another language, and wish to make it accessible to Java code through the JNI.
- You want to implement a small portion of time-critical code in a lower-level language such as assembly.

By programming through the JNI, you can use native methods to:

- Create, inspect, and update Java objects (including arrays and strings).
- Call Java methods.
- Catch and throw exceptions.
- Load classes and obtain class information.
- Perform runtime type checking.

You can also use the JNI with the *Invocation API* to enable an arbitrary native application to embed the Java VM. This allows programmers to easily make their existing applications Java-enabled without having to link with the VM source code.

Historical Background

VMs from different vendors offer different native method interfaces. These different interfaces force programmers to produce, maintain, and distribute multiple versions of native method libraries on a given platform.

We briefly examine some of the native method interfaces, such as:

- JDK 1.0 native method interface
- Netscape's Java Runtime Interface
- Microsoft's Raw Native Interface and Java/COM interface

JDK 1.0 Native Method Interface

JDK 1.0 was shipped with a native method interface. Unfortunately, there were two major reasons that this interface was unsuitable for adoption by other Java VMs.

First, the native code accessed fields in Java objects as members of C structures. However, the *Java Language Specification* does not define how objects are laid out in memory. If a Java VM lays out objects differently in memory, then the programmer would have to recompile the native method libraries.

Second, JDK 1.0's native method interface relied on a conservative garbage collector. The unrestricted use of the `unhand` macro, for example, made it necessary to conservatively scan the native stack.

Java Runtime Interface

Netscape had proposed the Java Runtime Interface (JRI), a general interface for services provided in the Java virtual machine. JRI was designed with portability in mind---it makes few assumptions about the implementation details in the underlying Java VM. The JRI addressed a wide range of issues, including native methods, debugging, reflection, embedding (invocation), and so on.

Raw Native Interface and Java/COM Interface

The Microsoft Java VM supports two native method interfaces. At the low level, it provides an efficient Raw Native Interface (RNI). The RNI offers a high degree of source-level backward compatibility with the JDK's native method interface, although it has one major difference. Instead of relying on conservative garbage collection, the native code must use RNI functions to interact explicitly with the garbage collector.

At a higher level, Microsoft's Java/COM interface offers a language-independent standard binary interface to the Java VM. Java code can use a COM object as if it were a Java object. A Java class can also be exposed to the rest of the system as a COM class.

Objectives

We believe that a uniform, well-thought-out standard interface offers the following benefits for everyone:

- Each VM vendor can support a larger body of native code.
- Tool builders will not have to maintain different kinds of native method interfaces.
- Application programmers will be able to write one version of their native code and this version will run on different VMs.

The best way to achieve a standard native method interface is to involve all parties with an interest in Java VMs. Therefore we organized a series of discussions among the Java licensees on the design of a uniform native method interface. It is clear from the discussions that the standard native method interface must satisfy the following requirements:

- Binary compatibility - The primary goal is binary compatibility of native method libraries across all Java VM implementations on a given platform. Programmers should maintain only one version of their native method libraries for a given platform.
- Efficiency - To support time-critical code, the native method interface must impose little overhead. All known techniques to ensure VM-independence (and thus binary compatibility) carry a certain amount of overhead. We must somehow strike a compromise between efficiency and VM-independence.
- Functionality - The interface must expose enough Java VM internals to allow native methods to accomplish useful tasks.

Java Native Interface Approach

We hoped to adopt one of the existing approaches as the standard interface, because this would have imposed the least burden on programmers who had to learn multiple interfaces in different VMs. Unfortunately, no existing solutions are completely satisfactory in achieving our goals.

Netscape's JRI is the closest to what we envision as a portable native method interface, and was used as the starting point of our design. Readers familiar with the JRI will notice the similarities in the API naming convention, the use of method and field IDs, the use of local and global references, and so on. Despite our best efforts, however, the JNI is not binary-compatible with the JRI, although a VM can support both the JRI and the JNI.

Microsoft's RNI was an improvement over JDK 1.0 because it solved the problem of native methods working with a nonconservative garbage collector. The RNI, however, was not suitable as a VM-independent native method interface. Like the JDK, RNI native methods access Java objects as C structures, leading to two problems:

- RNI exposed the layout of internal Java objects to native code.
- Direct access of Java objects as C structures makes it impossible to efficiently incorporate "write barriers," which are necessary in advanced garbage collection algorithms.

As a binary standard, COM ensures complete binary compatibility across different VMs. Invoking a COM method requires only an indirect call, which carries little overhead. In addition, COM objects are a great improvement over dynamic-link libraries in solving versioning problems.

The use of COM as the standard Java native method interface, however, is hampered by a few factors:

- First, the Java/COM interface lacks certain desired functions, such as accessing private fields and raising general exceptions.
- Second, the Java/COM interface automatically provides the standard IUnknown and IDispatch COM interfaces for Java objects, so that native code can access public methods and fields. Unfortunately, the IDispatch interface does not deal with overloaded Java methods and is case-insensitive in matching method names. Furthermore, all Java methods exposed through the IDispatch interface are wrapped to perform dynamic type checking and coercion. This is because the IDispatch interface is designed with weakly-typed languages (such as Basic) in mind.
- Third, instead of dealing with individual low-level functions, COM is designed to allow software components (including full-fledged applications) to work together. We believe that it is not appropriate to treat all Java classes or low-level native methods as software components.
- Fourth, the immediate adoption of COM is hampered by the lack of its support on UNIX platforms.

Although Java objects are not exposed to the native code as COM objects, the JNI interface itself is binary-compatible with COM. JNI uses the same jump table structure and calling convention that COM does. *This means that, as soon as cross-platform support for COM is available, the JNI can become a COM interface to the Java VM.*

JNI is not believed to be the only native method interface supported by a given Java VM. A standard interface benefits programmers, who would like to load their native code libraries into different Java VMs. In some cases, the programmer may have to use a lower-level, VM-specific interface to achieve top efficiency. In other cases, the programmer might use a higher-level interface to build software components. Indeed, as the Java environment and component software technologies become more mature, native methods will gradually lose their significance.

Programming to the JNI

Native method programmers should program to the JNI. Programming to the JNI insulates you from unknowns, such as the vendor's VM that the end user might be running. By conforming to the JNI standard, you will give a native library the best chance to run in a given Java VM.

If you are implementing a Java VM, you should implement the JNI. JNI has been time tested and ensured to not impose any overhead or restrictions on your VM implementation, including object representation, garbage collection scheme, and so on. Please send us your feedback if you run into any problems we may have overlooked.

Changes

As of Java SE 6.0, the deprecated structures JDK1_1InitArgs and JDK1_1AttachArgs have been removed, instead JavaVMInitArgs and JavaVMAttachArgs are to be used.

Contents | Previous | Next