Java Virtual Machine

A Java Virtual Machine (JVM) enables a set of computer software programs and data structures to use a virtual machine model for the execution of other computer programs and scripts. The model used by a JVM accepts a form of computer intermediate language commonly referred to as Java bytecode. This language conceptually represents the instruction set of a stack-oriented, capability architecture. Sun Microsystems states there are over 4.5 billion JVM-enabled devices.[1]

Overview

A JVM can also implement programming languages other than Java. For example, Ada source code can be compiled to Java bytecode, which may then be executed by a JVM. JVMs can also be released by other companies besides Oracle (the developer of Java) — JVMs using the "Java" trademark may be developed by other companies as long as they adhere to the JVM specification published by Oracle and to related contractual obligations.

Java was conceived with the concept of WORA: "write once, run anywhere". This is done using the Java Virtual Machine. The JVM is the environment in which Java programs execute. It is software that is implemented on non-virtual hardware and on standard operating systems.

JVM is a crucial component of the Java platform, and because JVMs are available for many hardware and software platforms, Java can be both middleware and a platform in its own right, hence the trademark write once, run anywhere. The use of the same bytecode for all platforms allows Java to be described as "compile once, run anywhere", as opposed to "write once, compile anywhere", which describes cross-platform compiled languages. A JVM also enables such features as automated exception handling, which provides "root-cause" debugging information for every software error (exception), independent of the source code.

A JVM is distributed along with a set of standard class libraries that implement the Java application programming interface (API). Appropriate APIs bundled together form the Java Runtime Environment (JRE).

Execution environment

Programs intended to run on a JVM must be compiled into a standardized portable binary format, which typically comes in the form of .class files. A program may consist of many classes in different files. For easier distribution of large programs, multiple class files may be packaged together in a .jar file (short for Java archive).

The Java application launcher, java, offers a standard way of executing Java code. Compare javaw.[2]

The JVM runtime executes .class or .jar files, emulating the JVM instruction set by interpreting it, or using a just-in-time compiler (JIT) such as Oracle's HotSpot. JIT compiling, not interpreting, is used in most JVMs today to achieve greater speed. There are also ahead-of-time compilers that enable developers to precompile class files into native code for particular platforms.

Like most virtual machines, the Java Virtual Machine has a stack-based architecture akin to a microcontroller/microprocessor. However, the JVM also has low-level support for Java-like classes and methods, which amounts to a highly idiosyncratic memory model and capability-based architecture.
JVM languages

Versions of non-JVM languages

<table>
<thead>
<tr>
<th>Language</th>
<th>On JVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Erlang</td>
<td>Erjang</td>
</tr>
<tr>
<td>JavaScript</td>
<td>Rhino</td>
</tr>
<tr>
<td>PHP</td>
<td>Resin</td>
</tr>
<tr>
<td>Python</td>
<td>Jython</td>
</tr>
<tr>
<td>REXX</td>
<td>NetRexx</td>
</tr>
<tr>
<td>Ruby</td>
<td>JRuby</td>
</tr>
<tr>
<td>Tcl</td>
<td>Jacl</td>
</tr>
</tbody>
</table>

Languages designed expressly for JVM

<table>
<thead>
<tr>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIDletPascal</td>
</tr>
<tr>
<td>Clojure</td>
</tr>
<tr>
<td>Groovy</td>
</tr>
<tr>
<td>Scala</td>
</tr>
</tbody>
</table>

Although the JVM was primarily aimed at running compiled Java programs, many other languages can now run on top of it.\(^4\) The JVM has currently no built-in support for dynamically typed languages: the existing JVM instruction set is statically typed,\(^5\) although the JVM can be used to implement interpreters for dynamic languages. The JVM has a limited support for dynamically modifying existing classes and methods; this currently only works in a debugging environment, where new classes and methods can be added dynamically. Built-in support for dynamic languages is currently planned for Java 7.\(^6\)

Bytecode verifier

A basic philosophy of Java is that it is inherently "safe" from the standpoint that no user program can "crash" the host machine or otherwise interfere inappropriately with other operations on the host machine, and that it is possible to protect certain methods and data structures belonging to "trusted" code from access or corruption by "untrusted" code executing within the same JVM. Furthermore, common programmer errors that often lead to data corruption or unpredictable behavior such as accessing off the end of an array or using an uninitialized pointer are not allowed to occur. Several features of Java combine to provide this safety, including the class model, the garbage-collected heap, and the verifier.

The JVM verifies all bytecode before it is executed. This verification consists primarily of three types of checks:

- Branches are always to valid locations
- Data is always initialized and references are always type-safe
- Access to "private" or "package private" data and methods is rigidly controlled.

The first two of these checks take place primarily during the "verification" step that occurs when a class is loaded and made eligible for use. The third is primarily performed dynamically, when data items or methods of a class are first accessed by another class.

The verifier permits only some bytecode sequences in valid programs, e.g. a jump (branch) instruction can only target an instruction within the same method. Furthermore, the verifier ensures that any given instruction operates on
a fixed stack location, allowing the JIT compiler to transform stack accesses into fixed register accesses. Because of this, that the JVM is a stack architecture does not imply a speed penalty for emulation on register-based architectures when using a JIT compiler. In the face of the code-verified JVM architecture, it makes no difference to a JIT compiler whether it gets named imaginary registers or imaginary stack positions that must be allocated to the target architecture's registers. In fact, code verification makes the JVM different from a classic stack architecture whose efficient emulation with a JIT compiler is more complicated and typically carried out by a slower interpreter. Code verification also ensures that arbitrary bit patterns cannot get used as an address. Memory protection is achieved without the need for a memory management unit (MMU). Thus, JVM is an efficient way of getting memory protection on simple architectures that lack an MMU. This is analogous to managed code in Microsoft's .NET Common Language Runtime, and conceptually similar to capability architectures such as the Plessey 250, and IBM System/38.

**Bytecode instructions**

The JVM has instructions for the following groups of tasks:

- Load and store
- Arithmetic
- Type conversion
- Object creation and manipulation
- Operand stack management (push / pop)
- Control transfer (branching)
- Method invocation and return
- Throwing exceptions
- Monitor-based concurrency

The aim is binary compatibility. Each particular host operating system needs its own implementation of the JVM and runtime. These JVMs interpret the bytecode semantically the same way, but the actual implementation may be different. More complex than just emulating bytecode is compatibly and efficiently implementing the Java core API that must be mapped to each host operating system.

**Secure execution of remote code**

A virtual machine architecture allows very fine-grained control over the actions that code within the machine is permitted to take. This is designed to allow safe execution of untrusted code from remote sources, a model used by Java applets. Applets run within a VM incorporated into a user's browser, executing code downloaded from a remote HTTP server. The remote code runs in a restricted "sandbox", which is designed to protect the user from misbehaving or malicious code. Publishers can purchase a certificate with which to digitally sign applets as "safe", giving them permission to ask the user to break out of the sandbox and access the local file system, clipboard or network.
C to bytecode compilers

From the point of view of a compiler, the Java Virtual Machine is just another processor with an instruction set, Java bytecode, for which code can be generated. The JVM was originally designed to execute programs written in the Java language. However, the JVM provides an execution environment in the form of a bytecode instruction set and a runtime system that is general enough that it can be used as the target for compilers of other languages.

Because of its close association with the Java language, the JVM performs the strict runtime checks mandated by the Java specification. That requires C to bytecode compilers to provide their own "lax machine abstraction", for instance producing compiled code that uses a Java array to represent main memory (so pointers can be compiled to integers), and linking the C library to a centralized Java class that emulates system calls. Most or all of the compilers listed below use a similar approach.

Several C to bytecode compilers exist:

- NestedVM translates C to MIPS machine language first before converting to Java bytecode.
- Cibyl works similarly to NestedVM but targets J2ME devices.
- LLVM compiles C to LLVM IR, which is then translated to JVM bytecode.
- C2J is also GCC-based, but it produces intermediary Java source code before generating bytecode.
- Supports the full ANSI C runtime.
- Axiomatic Multi-Platform C supports full ANSI C 1989, SWT, and J2ME CDC 1.1 for mobile devices.
- Java Backend for GCC, possibly the oldest project of its kind, was developed at The University of Queensland in 1999.
- Javum is an attempt to port the full GNU environment to the JVM, and includes one of the above compilers packaged with additional utilities.
- egcs-jvm appears to be an inactive project.

Compilers targeting Java bytecode have been written for other programming languages, including Ada and COBOL.

Licensing

Starting with J2SE 5.0, changes to the JVM specification have been developed under the Java Community Process as JSR 924. As of 2006, changes to specification to support changes proposed to the class file format (JSR 202) are being done as a maintenance release of JSR 924. The specification for the JVM is published in book form, known as "blue book". The preface states:

We intend that this specification should sufficiently document the Java Virtual Machine to make possible compatible clean-room implementations. Oracle provides tests that verify the proper operation of implementations of the Java Virtual Machine.

Oracle's JVM is called HotSpot. Clean-room Java implementations include Kaffe, IBM J9 and Dalvik. Oracle retains control over the Java trademark, which it uses to certify implementation suites as fully compatible with Oracle's specification.
Heap

The Java Virtual Machine heap is the area of memory used by the JVM (and specifically HotSpot) for dynamic memory allocation. The heap is split up into "generations":

- The young generation stores short-lived objects that are created and immediately garbage collected.
- Objects that persist longer are moved to the old generation (also called the tenured generation).
- The permanent generation (or permgen) is used for class definitions and associated metadata.

Originally there was no permanent generation, and objects and classes were stored together in the same area. But as class unloading occurs much more rarely than objects are collected, moving class structures to a specific area allows significant performance improvements.

Notes

3. 1996, possibly the first new language specifically designed to run on the JVM
17. JSR 924 ([http://www.jcp.org/en/jsr/detail?id=924](http://www.jcp.org/en/jsr/detail?id=924)), specifies changes to the JVM specification starting with J2SE 5.0
18. JSR 202 ([http://www.jcp.org/en/jsr/detail?id=202](http://www.jcp.org/en/jsr/detail?id=202)), specifies a number of changes to the class file format
References

- Clarifications and Amendments to the Java Virtual Machine Specification, Second Edition (http://java.sun.com/docs/books/vmspec/2nd-edition/jvms-clarify.html) includes list of changes to be made to support J2SE 5.0 and JSR 45
- JSR 45 (http://www.jcp.org/en/jsr/detail?id=45), specifies changes to the class file format to support source-level debugging of languages such as JavaServer Pages (JSP) and SQLJ that are translated to Java

External links

- The Java Virtual Machine Specification (http://java.sun.com/docs/books/vmspec/)
- Java implementations (http://www.dmoz.org/Computers/Programming/Languages/Java/Implementations/) at the Open Directory Project
- Sun to build virtual machine for iPhone - ComputerWorld (http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9067358)

License

Creative Commons Attribution-Share Alike 3.0 Unported
http://creativecommons.org/licenses/by-sa/3.0/