Theory: Inside the JVM

9 minutes 0 / 3 problems solved

Skip this topic Start practicing

§1. What is Java Virtual Machine?

The **Java Virtual Machine (JVM)** is a virtual simulation of a physical computer that executes compiled Java programs (bytecode). JVM runs as an application on top of an operating system and provides an environment for Java programs.

Because they use virtual machines, Java programs are platform-independent and can be executed on different hardware and operating systems according to the **WORA** (***Write Once Run Anywhere***) principle.

There are a lot of different JVM implementations. HotSpot is the primary reference Java VM implementation. It's used by Oracle Java and OpenJDK.

Many JVMs (including HotSpot) are implemented according to the Java Virtual Machine Specification. You do not need to read it now, just remember that this specification exists.

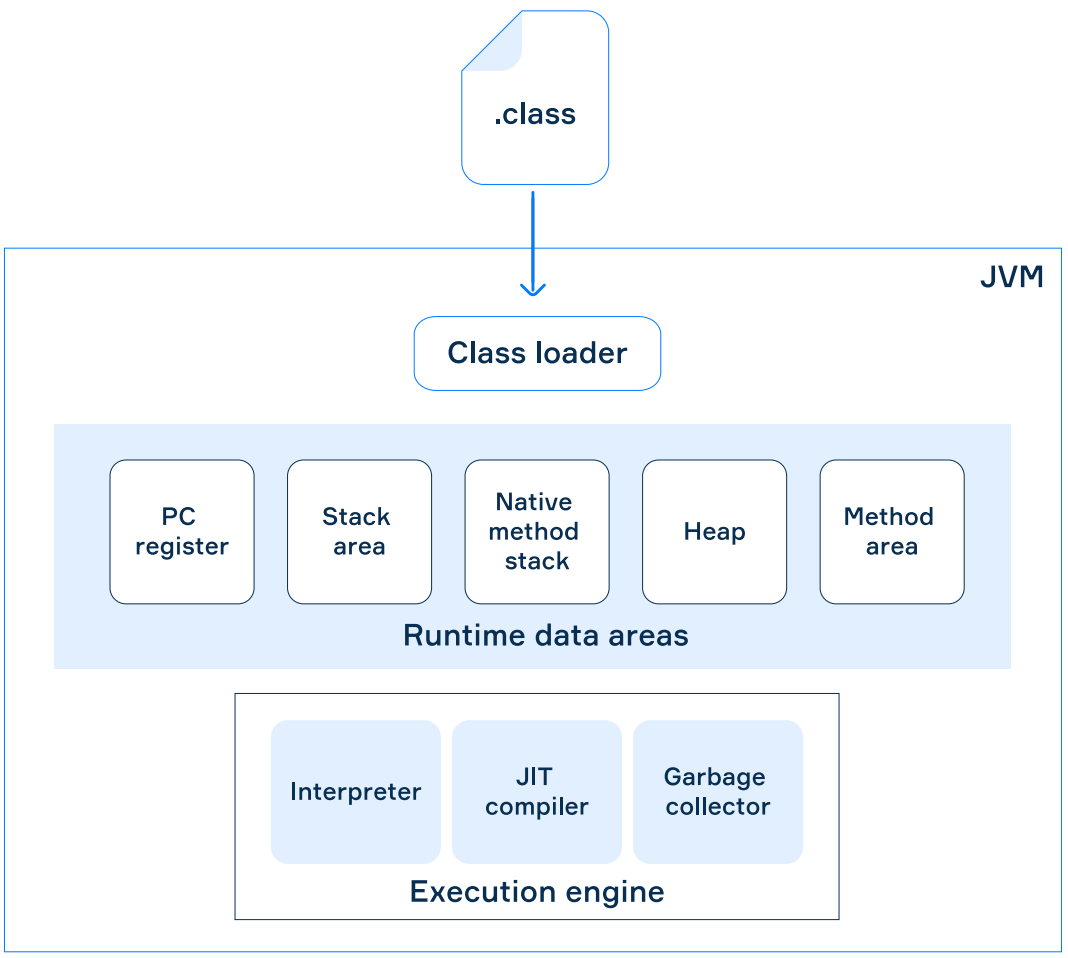
§2. The JVM internals overview

After the compilation of a Java program, there is a file with the **.class** extension. It contains the Java bytecode. In order to execute the code, you need to load it into JVM. When JVM executes a program, it translates the bytecode into the platform native code.

JVM mainly performs the following activities:

- loads bytecode;
- verifies bytecode;
- executes bytecode;
- provides the runtime environment.

The following image illustrates the common JVM architecture:



The common JVM architecture

Let's consider every subsystem in more details.

§3. The class loader subsystem

Current topic:

Inside the JVM

Topic depends on:

✕ Introduction to Java Platform

Table of contents:

This subsystem loads the Java bytecode for execution, verifies it and then allocates memory for the bytecode. We cover the subsystem in another topic. To verify bytecode there is a module called **bytecode verifier**. It checks that the instructions don't require any dangerous actions like accessing private fields and methods of classes and objects.

§4. The runtime data areas

This **subsystem** represents **JVM memory**. The areas are used for different purposes during the program execution.

- **PC register** holds the address of the currently executing instruction;
- **stack area** is a memory place where methods' calls and local variables are stored;
- **native method stack** stores native method information;
- **heap** stores all created objects (instances of classes);
- **method area** stores all the class level information like class name, immediate parent class name, method information and all static variables.

Every thread has its own **PC register**, **stack**, and **native method stack**, but all threads share the same **heap** and **method area.**

§5. Execution engine

It is responsible for executing the program (bytecode). It interacts with various data areas of JVM when executing a bytecode.

The execution engine has the following parts:

- **bytecode interpreter** interprets the bytecode line by line and executes it (rather slowly);
- **just-in-time compiler** (JIT compiler) translates bytecode into native machine language while executing the program (it executes the program faster than the interpreter);
- **garbage collector** cleans unused objects from the heap.

Different JVM implementations can contain both a **bytecode interpreter** and a **just-in-time compiler**, or only one of them. Do not confuse them with the javac (source code to bytecode compiler); it's not included in JVM.

§6. Interfaces and libraries

Other important parts of JVM for execution:

- **native method interface** provides an interface between Java code and the native method libraries;
- **native method library** consists of (C/C++) files that are required for the execution of native code.

Thus, JVM has a lot of parts. We won't cover all of them because it's enough to understand the JVM in general. The class loader working principles and garbage collection algorithms will be discussed in separate topics.

Report a typo

88 users liked this theory. **1** didn't like it. **What about you?**

Start practicing

Comments (5) Hints (0) Useful links (0) Show discussion