JIT vs Interpreters

Ask Question

I couldn't find the difference between JIT and Interpreters.

Jit is intermediary to Interpreters and Compilers. During runtime, it converts byte code to machine code (JVM or Actual Machine ?) For the next time, it takes from the cache and runs Am i right?

Interpreters will directly execute bytecode without transforming it into machine code. Is that right?

How the real processor in our pc will understand the instruction.?

Please clear my doubts.

java

asked Sep 15 '10 at 13:26

Manoj
1,873 10 34 62

6 Answers

- Interpreter: Reads your source code or some intermediate representation (bytecode) of it, and executes it *directly*.
- JIT compiler: Reads your source code, or more typically some intermediate representation (bytecode) of it, compiles that on the fly and executes *native code*.

answered Sep 15 '10 at 13:55

gpeche
16.3k 2 23 41

> Jit is intermediary to Interpreters and Compilers. During runtime, it converts byte code to machine code (JVM or Actual Machine ?) For the next time, it takes from the cache and runs Am i right?

Yes you are.

> Interpreters will directly execute bytecode without transforming it into machine code. Is that right?

Yes, it is.

> How the real processor in our pc will understand the instruction.?

In the case of interpreters, the virtual machine executes a native JVM procedure corresponding to each instruction in byte code to produce the expected behaviour. But your code isn't actually compiled to native code, as with Jit compilers. The JVM emulates the expected behaviour for each instruction.

answered Sep 15 '10 at 13:34

Vivien Barousse
17k 2 45 64

First thing first:
With JVM, both interpreter and compiler (the JVM compiler and not the source-code compiler like javac) *produce native code* (aka Machine language code for the underlying physical CPU like x86) from *byte code*.

What's the difference then:
The difference is in how they generate the native code, how optimized it is as well how costly the optimization is. Informally, an interpreter pretty much converts each byte-code instruction to corresponding native instruction by looking up a predefined JVM-instruction to machine instruction mapping (see below pic). Interestingly, a further speedup in execution can be achieved, if we take a section of byte-code and convert it into machine code - because considering a whole logical section often provides rooms for optimization as opposed to converting (*interpreting*) each line in isolation (*to machine instruction*). This very act of converting a section of byte-code into (presumably optimized)

machine instruction is called compiling (in the current context). When the compilation is done at run-time, the compiler is called JIT compiler.

Bytecode	ARM Instruction
ILOAD_3	LDR R0, [R7, #12]
ILOAD_4	LDR R1 [R7, #16]
IADD	ADD R0, R1
ISTORE_3	STR R0, [R7, #12]

Table 1: An example of bytecode hardware-translation.

The co-relation and co-ordination:
Since Java designer went for (*hardware & OS*) portability, they had chosen interpreter architecture (*as opposed to c style compiling, assembling, and linking*). However, in order to achieve more speed up, a compiler is also optionally added to a JVM. Nonetheless, as a program goes on being interpreted (and executed in physical CPU) "hotspot"s are detected by JVM and statistics are generated. Consequently, using statistics from interpreter, those sections become candidate for compilation (optimized native code). It is in fact done on-the-fly (thus JIT compiler) and the compiled machine instructions are used subsequently (rather than being interpreted). In a natural way, JVM also caches such compiled pieces of code.

Words of caution:
These are pretty much the fundamental concepts. If an actual implementer of JVM, does it a bit different way, don't get surprised. So could be the case for VM's in other languages.

Words of caution:
Statements like "interpreter executes byte code in virtual processor", "interpreter executes byte code directly", etc. are all correct as long as you understand that in the end there is a set of machine instructions that have to run in a physical hardware.

Some Good References: [I've not done extensive search though]

- [paper] Instruction Folding in a Hardware-Translation Based Java Virtual Machine by Hitoshi Oi
- [book] Computer organization and design, 4th ed, D. A. Patterson. (*see Fig 2.23*)
- [web-article] JVM performance optimization, Part 2: Compilers, by Eva Andreasson (JavaWorld)

PS: I've used following terms interchangebly - 'native code', 'machine language code', 'machine instructions', etc.

edited Feb 17 '17 at 7:57

answered Nov 29 '16 at 21:28

KGhatak
3,465 1 12 15

A JIT Compiler translates byte code into machine code and then execute the machine code.

Interpreters read your high level language (interprets it) and execute what's asked by your program. Interpreters are normally not passing through byte-code and jit compilation.

But the two worlds have melt because numerous interpreters have take the path to internal byte-compilation and jit-compilation, for a better speed of execution.

answered Sep 15 '10 at 13:34

Jérôme Radix
7,917 3 24 35

I'm pretty sure that JIT turns byte code into machine code for whatever machine you're running on right as it's needed. The alternative to this is to run the byte code in a java virtual machine. I'm not sure if this the same as interpreting the code since I'm more familiar with that term being used to describe the execution of a scripting (non-compiled) language like ruby or perl.

answered Sep 15 '10 at 13:37

Aaron
5,525 4 46 86

The first time a class is referenced in JVM the JIT Execution Engine re-compiles the .class files (primary Binaries) generated by Java Compiler containing JVM Instruction Set to Binaries containing HOST system's Instruction Set. JIT stores and reuses those recompiled binaries from Memory going forward, there by reducing interpretation time and benefits from Native code execution.

And there is another flavor which does Adaptive Optimization by identifying most reused part of the app and applying JIT only over it, there by optimizing over memory usage.

On the other hand a plain old java interpreter interprets one JVM instruction from class file at a time and calls a procedure against it.

Find a detail comparison here

answered Feb 21 '14 at 2:38

bitan

133 1 7