Demystifying the JVM: Interpretation, JIT and AOT Compilation

Jan 22, 2017
13 minute read

I guess anyone coding Java has heard about JIT (Just In Time) and maybe also AOT (Ahead Of Time) compilation. We also hear "interpreted" languages a lot. It is the purpose of this post to show how all of these are used in the Java Virtual Machine, JVM.

As you probably know, when you code in the Java programming language, you run the Java compiler (the one triggered by the "javac" program), which compiles the Java source code (.java files), into Java bytecodes (.class files). Java bytecodes are the intermediate language. It is called intermediate because it is not understandable and executable by the real computing device, the CPU, so it is an intermediate form between the source code and the CPU-executable "native" machine code.

In order for Java bytecodes to do some actual work, there are 3 options:

1. Executing the intermediate code directly. In other and better words, "interpreting" the intermediate code. JVM is (or more properly has) a Java language interpreter. As you know, JVM is executed by running the "java" program.
2. Just before executing the intermediate code, compiling it into the native code, and letting the CPU run the very freshly generated native code. So the compilation is Just In Time of the execution.
3. Before everything, before running the program, compiling the intermediate code to the native code and letting the CPU run it from the beginning. So the compilation is before/Ahead Of execution Time.

So, (1) is the job of an Interpreter, (2) is the result of JIT compilation and (3) is the result of AOT compilation.

For the sake of completeness, there is actually one more approach, to interpret the source code directly, but Java does not work like that, but for example Python does.

So now our question is how "java" works, as an (1) Interpreter, (2) JIT compiler and/or (3) AOT compiler and when.

The short answer is, normally, "java" does both (1) and (2). With Java 9, which is planned to be released in the summer of 2017, (3) will be possible as well. If you want to try this now, in order for 3 (AOT) to work, you need JDK 9, so you need to get the snapshot builds. I believe Ubuntu 16 has openjdk-9-jdk package available, however, last time I checked, it was build 134. However, the AOT support in JDK 9, JEP 295: Ahead-of-Time Compilation, is integrated in build 150, so you should not use the package manager, but download the latest JDK 9 build. I am using Ubuntu 16.10 and JDK 9 build 153 for all the examples in this post.

Here is our Test class, to be used in the examples later.

```java
public class Test {

  public int f() throws Exception {
    int a = 5;
    return a;
  }

  public static void main(String[] args) throws Exception {
    for (int i = 1; i <= 10; i++) {
      System.out.println("call " + Integer.valueOf(i));
      long a = System.nanoTime();
      new Test().f();
      long b = System.nanoTime();
      System.out.println("elapsed= " + (b-a));
    }

  }

}
```

As you see, it has the main method, which instantiates a Test object and calls function f, 10 times in a loop. Function f does almost nothing.

So if you compile and run the code above, the output will be, as very much expected (and of course elapsed times will be different for you):

```
call 1
elapsed= 5373
call 2
elapsed= 913
call 3
elapsed= 654
```

```
call 4
elapsed= 623
call 5
elapsed= 680
call 6
elapsed= 710
call 7
elapsed= 728
call 8
elapsed= 699
call 9
elapsed= 853
call 10
elapsed= 645
```

Now, here is the question, is this output a result of "java" as an interpreter, as in option (1), "java" as a JIT compiler, as in option (2) or does it have anything to do with AOT, as in option (3) ? The aim of this post is finding the proper answer to these questions.

My first answer is, most probably, it is only (1). The reason I say "most probably" is because I do not know if any environment variable is set to modify any default JVM options. If nothing is set and this is the default run of "java", then it is 100% option (1), which means it is purely interpreted. So how do I know this, because:

- According to "java" documentation, -XX:CompileThreshold=invocations option runs with default invocations=1500 on client JVM (see below for more about client JVM). Since I run it only 10 < 1500 times, it should not be JIT compiled. This command line option basically sets the maximum number of times the function should be interpreted before (JIT)compiled. I will elaborate more on this later.
- I have actually run the code with the diagnostic flags, so I know if it is (JIT)compiled or not. I will show this very soon.

  Note: JVM can run in client or server mode, which does change the default values of the options. This is normally decided automatically depending on the environment/computer JVM has been triggered. From now on, I am going to add -client option to my runs to be sure it is running in client mode. This option does not change anything I would like to show in this post.

If you run "java" with -XX:PrintCompilation option, it will write a line, when a function is JIT(compiled). Keep in mind, JIT compilation is per function, so in a class, some functions can still be bytecode (so will be interpreted) and others can be JIT compiled (so will be run by CPU directly).

Below, I am also adding -Xbatch option. The reason for -Xbatch option is only to make output more presentable, otherwise the JIT compilation occurs concurrently (together with interpretation) and the output of compilation (due to -XX:PrintCompilation) may appear at strange times during the execution. However, with -Xbatch, we disable the background compilation so the JIT compilation will be done by stopping our execution.

(For the sake of making it easy to read, I will write the options into new lines.)

```
$ java -client -Xbatch
-XX:+PrintCompilation
Test
```

I am not going to paste the output of this command here, because by default, JVM compiles many internal functions (e.g. some functions in java, sun, jdk packages) so you will see many lines of output (I see 274 lines for internal functions, and a few more for the program output itself). In order to make this study simpler, I would like to disable JIT compilation for internal classes or selectively enable the JIT compilation only for my method (Test.f). To do this, we need to provide another option -XX:CompileCommand. You can provide many (compile) commands so it is easier to provide this as a file and luckily we have -XX:CompileCommandFile option too. So lets create a file, I name it hotspot_compiler for the reason I will tell soon, and typed the following:

```
quiet
exclude java/* *
exclude jdk/* *
exclude sun/* *
```

This should be very clear, we are excluding all functions (last *) in all classes in all packages starting with java, jdk and sun (package names are separated by / and you can use *). quiet command tells JVM to not write anything about excluded classes so only the ones that are compiled will be written to the console. Now, I run:

```
java -client -Xbatch
-XX:+PrintCompilation
-XX:CompileCommandFile=hotspot_compiler
Test
```

Before telling you the output of this command, I named the file hotspot_compiler, because it seems (I did not check this) for the Oracle JDK, .hotspot_compiler is the default name for the compiler command file.

Now, the output:

```
many lines like this 111    1     n 0       java.lang.invoke.MethodHandle::linkToStatic(LLLLLL)L (native)   (static)
call 1
some more lines like this 161   48     n 0       java.lang.invoke.MethodHandle::linkToStatic(ILIJL)I (native)   (static)
elapsed= 7558
call 2
elapsed= 1532
call 3
elapsed= 920
call 4
elapsed= 732
call 5
elapsed= 774
call 6
elapsed= 815
call 7
elapsed= 767
call 8
elapsed= 765
call 9
elapsed= 757
call 10
elapsed= 868
```

First, I do not know why still some of the java.lang.invoke.MethodHandler methods are compiled. It should be something you cannot disable -I will update this post when I figure out-. However, as you see, all other compilations (the 274 lines before) has disappeared. (from now on, I am going to omit the compilation logs of java.lang.invoke.MethodHandler in the outputs)

Lets see where we are now. We have a simple code, where we run our function 10 times. I told you before that this function is interpreted and not compiled, because, the documentation says so and now we see it in the logs (meaning we do not see it in the compilation logs this means no JIT compilation). Great, you have just seen the “java” tool, interpreting and only interpreting our function 100% of time. So we put a check to our option (1). Now, lets move to (2), JIT compilation.

As the documentation says, we can run our function 1500 times and see that JIT compilation really happens. However, we can also use the -XX:CompileThreshold=invocations option to set what we want instead of 1500. So lets set it now to 5, we expect to see after 5 “interpretation” of our function f, JVM should compile our method, and then run the compiled version.

```
java -client -Xbatch
-XX:+PrintCompilation
-XX:CompileCommandFile=hotspot_compiler
-XX:CompileThreshold=5
Test
```

If you have run the command, you may have realized, nothing has changed from the above, so there is still no compilation. The reason is, according to the documentation, -XX:CompileThreshold only works if TieredCompilation is disabled, which is enabled by default, and it can be disabled by -XX:-TieredCompilation. Tiered Compilation is a feature introduced in Java 7 to improve both startup and regular speed of JVM, it is not relevant for the nature of this post so it is safe to disable it. So lets run the following command now:

```
java -client -Xbatch
-XX:+PrintCompilation
-XX:CompileCommandFile=hotspot_compiler
-XX:CompileThreshold=5
-XX:-TieredCompilation
Test
```

The output is (remember, I am omitting the lines about java.lang.invoke.MethodHandle):

```
call 1
elapsed= 9411
call 2
elapsed= 1291
call 3
elapsed= 862
call 4
elapsed= 1023
call 5
```

```
    227   56    b        Test::<init> (5 bytes)
    228   57    b        Test::f (4 bytes)
elapsed= 1051739
call 6
elapsed= 18516
call 7
elapsed= 940
call 8
elapsed= 769
call 9
elapsed= 855
call 10
elapsed= 838
```

Say hello! to JIT compiled function Test.f (or `Test::f`), just after the call number 5, as I set the CompileThreshold to 5. JVM interprets the function 5 times and then compiles it, then runs the compiled version. Since it is compiled, it should run faster but we cannot test it here since function does not do anything. It could be a good topic for another post.

As you probably realized, another function is compiled as well, and it is `Test::<init>`, which is the constructor of Test class. Since the code calls constructor (new Test()) every time it runs the f, it is compiled at the same time as the function f, exactly after 5 calls.

This actually concludes the option (2), JIT compilation. Here, you see, the function is first interpreted by JVM, then Just-In-Time compiled after 5 times it is interpreted. I would like to add one last point to JIT compilation, which brings us to `-XX:+PrintAssembly`option. As the name suggests, this prints the compiled version of the function (compiled version=native machine code=assembly code). However, this only works if there is a disassemler in the library path. The disassembler, which I believe can be different for different JVMs, for openjdk is hsdis. There are different places where you can reach to the source code or binary of hsdis library, I have used the one here and compiled it, and put `hsdis-amd64.so` to `JAVA_HOME/lib/server`.

Now, we can run the command, but before, I should add, in order to run `-XX:+PrintAssembly`, you also need to add `-XX:+UnlockDiagnosticVMOptions` option as well and it should be before the PrintAssembly option. If you do not do this, you will see JVM actually gives a warning on proper usage of PrintAssembly option. So lets run this now:

```
java -client -Xbatch
-XX:+PrintCompilation
-XX:CompileCommandFile=hotspot_compiler
-XX:CompileThreshold=5
-XX:-TieredCompilation
-XX:+UnlockDiagnosticVMOptions
-XX:+PrintAssembly
Test
```

The output is long, and there are lines like:

```
0x00007f4b7cab1120: mov    0x8(%rsi),%r10d
0x00007f4b7cab1124: shl    $0x3,%r10
0x00007f4b7cab1128: cmp    %r10,%rax
```

So as you see, the corresponding functions are compiled to native machine code.

Finally, the option 3, AOT. Ahead-of-Time, AOT compilation was not available with Open JDK (or Sun/Oracle JDK) before and it is just added to JDK 9 which is still under development. There are some 3rd party tools and other JVMs that supports AOT.

JDK 9 brings a new tool jaotc, as you guess, java AOT compiler. The idea is, you run the Java compiler “javac”, then Java AOT compiler “jaotc” and then run JVM “java” as usual. JVM does the usual interpret and JIT compile as before, however, if a function has an AOT compiled code, it uses that one directly, instead of interpreting or JIT compiling it. Just to be clear, you do not need to run AOT compiler, it is optional, and if you run, you can run it only for the classes you want to be AOT compiled.

Lets build a library consisting of AOT compiled version of Test::f. Keep in mind you need JDK 9 build 150+ to do this on your own.

```
jaotc --output=libTest.so Test.class
```

This generates libTest.so, the library that contains the AOT compiled native code of the functions in Test class. You can see the symbols defined in this library:

```
nm libTest.so
```

Besides many others, this will show:

```
0000000000002120 t Test.f()I
00000000000021a0 t Test.<init>()V
00000000000020a0 t Test.main([Ljava/lang/String;)V
```

So, all of our functions, the constructor, f and the static method main is there in the libTest.so library.

Similar to corresponding “java” option, it is possible to provide options with a file, with –compile-commands option to `jaotc`. JEP 295 gives examples for this, which I will not show here.

Now, it is time to run “java” and see if AOT compiled methods are used. If you run “java” as before, not surprisingly it will run as before, so no AOT library will be used. To use this feature, there is a new option, `-XX:AOTLibrary`, which you need to provide:

```
java -XX:AOTLibrary=./libTest.so Test
```

You can provide multiple AOT libraries separated by commas.

The output of this command is not different than running “java” without AOTLibrary, since nothing is changed for the behaviour of Test program. To see if AOT compiled functions are used, we can add another new option, `-XX:+PrintAOT`.

```
java
-XX:AOTLibrary=./libTest.so
-XX:+PrintAOT
Test
```

Before the output of Test program, this command shows the following:

```
 9    1     loaded    ./libTest.so  aot library
99    1     aot[ 1]   Test.main([Ljava/lang/String;)V
99    2     aot[ 1]   Test.f()I
99    3     aot[ 1]   Test.<init>()V
```

As expected, AOT library is loaded, and the AOT compiled functions are used.

If you wonder, you can run the following command to see if there is any JIT compilation occurs.

```
java -client -Xbatch
-XX:+PrintCompilation
-XX:CompileCommandFile=hotspot_compiler
-XX:CompileThreshold=5
-XX:-TieredCompilation
-XX:AOTLibrary=./libTest.so
-XX:+PrintAOT
Test
```

As expected, no JIT compilation happens, because the methods in Test class is AOT compiled and provided as a library.

You may wonder since we provide the native code for the functions, how JVM detects if the native code is old/stale. As a final example, lets modify the f function and set a to 6.

```
public int f() throws Exception {
  int a = 6;
  return a;
}
```

I did this only to modify the class file. Now, we will “javac” compile and run the same command above.

```
javac Test.java
java -client -Xbatch
-XX:+PrintCompilation
-XX:CompileCommandFile=hotspot_compiler
-XX:CompileThreshold=5
-XX:-TieredCompilation
-XX:AOTLibrary=./libTest.so
-XX:+PrintAOT
Test
```

As you see, I did not run “jaotc” after “javac”, so the code in AOT library is old and incorrect now, the function f has a=5.

The output of “java” command above shows the following:

```
228   56    b        Test::<init> (5 bytes)
229   57    b        Test::f (5 bytes)
```

which means these functions are JIT compiled, thus AOT compiled code is not used. So the change in class file is detected. When you “javac” compile, a fingerprint is put into the class and class fingerprint is also kept in AOT library. Since the class fingerprint is different now, than the one in AOT library, AOT compiled native code is not used. This also concludes all the things I would like to say about the last option, AOT compilation.

In this post, I tried to explain and give simple real world examples of how JVM executes the Java code, by interpreting it, by JIT compiling it and, by AOT compiling it as a new feature of JDK 9. I hope you have learned something new.