深入JVM系列-JIT编译详解

JIT 简介

JIT 是 (Just In Time - Compliter) 即时编译编译器。使用即时编译器技术，能够加速 Java 程序的执行速度。

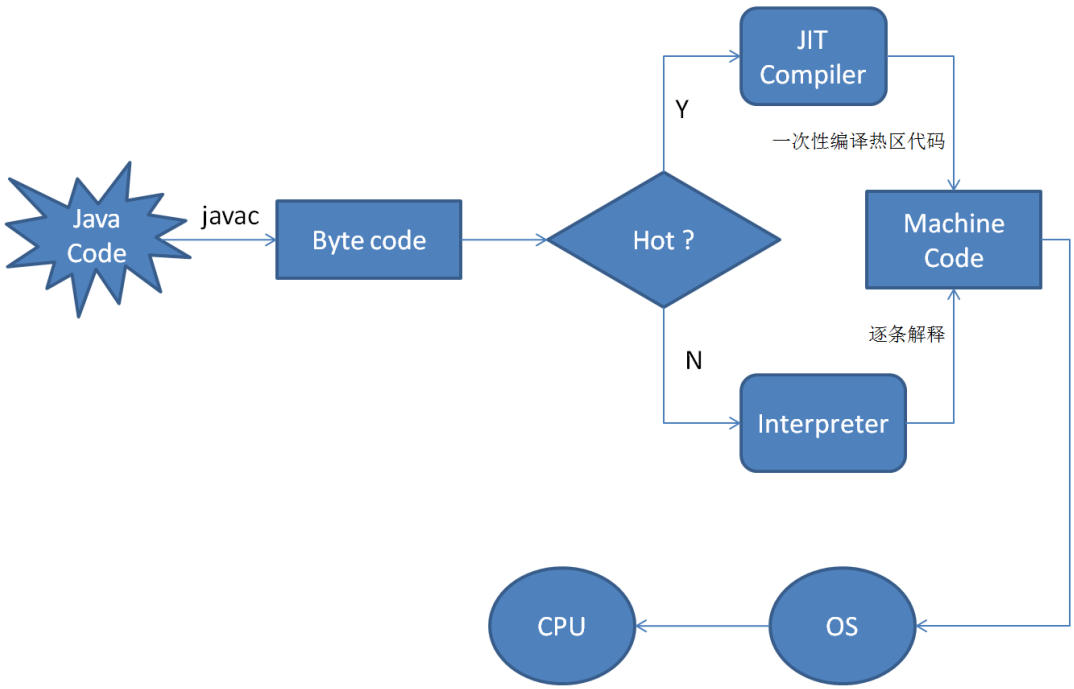
首先，我们大家都知道，通常通过 javac 将程序源代码编译，转换成 java 字节码，JVM 通过解释字节码将其翻译成对应的机器指令，逐条读入，逐条解释翻译。很显然，经过解释执行，其执行速度必然会比可执行的二进制字节码程序慢很多。为了提高执行速度，引入了 JIT 技术。

当虚拟机发现某个方法或代码块运行特别频繁时，就会把这些代码认定为“Hot Spot Code”（热点代码），为了提高热点代码的执行效率，在运行时，虚拟机将会把这些代码编译成与本地平台相关的机器码，并进行各层次的优化，完成这项任务的正是JIT编译器。

在运行时 JIT 会把翻译过的机器码保存起来，以备下次使用，因此从理论上来说，采用该 JIT 技术可以接近以前纯编译技术。下面我们看看，JIT 的工作过程。

JIT 编译过程

当 JIT 编译启用时（默认是启用的），JVM 读入.class 文件解释后，将其发给 JIT 编译器。JIT 编译器将字节码编译成本机机器代码，下图展示了该过程。



运行过程中会被即时编译器编译的“热点代码”有两类：

- 被多次调用的方法。
- 被多次调用的循环体。

两种情况，编译器都是以整个方法作为编译对象，这种编译也是虚拟机中标准的编译方式。要知道一段代码或方法是不是热点代码，是不是需要触发即时编译，需要进行Hot Spot Detection（热点探测）。目前主要的热点判定方式有以下两种：

- 基于采样的热点探测：采用这种方法的虚拟机会周期性地检查各个线程的栈顶，如果发现某些方法经常出现在栈顶，那这段方法代码就是“热点代码”。这种探测方法的好处是实现简单高效，还可以很容易地获取方法调用关系，缺点是很难精确地确认一个方法的热度，容易因为受到线程阻塞或别的外界因素的影响而扰乱热点探测。

- 基于计数器的热点探测：采用这种方法的虚拟机会为每个方法，甚至是代码块建立计数器，统计方法的执行次数，如果执行次数超过一定的阈值，就认为它是“热点方法”。这种统计方法实现复杂一些，需要为每个方法建立并维护计数器，而且不能直接获取到方法的调用关系，但是它的统计结果相对更加精确严谨。

Hot Spot 编译

当 JVM 执行代码时，它并不立即开始编译代码。这主要有两个原因：

首先，如果这段代码本身在将来只会被执行一次，那么从本质上看，编译就是在浪费精力。因为将代码翻译成 java 字节码相对于编译这段代码并执行代码来说，要快很多。

当然，如果一段代码频繁的调用方法，或是一个循环，也就是这段代码被多次执行，那么编译就非常值得了。因此，编译器具有的这种权衡能力会首先执行解释后的代码，然后再去分辨哪些方法会被频繁调用来保证其本身的编译。其实说简单点，就是 JIT 在起作用，我们知道，对于 Java 代码，刚开始都是被编译器编译成字节码文件，然后字节码文件会被交由 JVM 解释执行，所以可以说 Java 本身是一种半编译半解释执行的语言。Hot Spot VM 采用了 JIT compile 技术，将运行频率很高的字节码直接编译为机器指令执行以提高性能，所以当字节码被 JIT 编译为机器码的时候，要说它是编译执行的也可以。也就是说，运行时，部分代码可能由 JIT 翻译为目标机器指令（以 method 为翻译单位，还会保存起来，第二次执行就不用翻译了）直接执行。

第二个原因是最优化，当 JVM 执行某一方法或遍历循环的次数越多，就会更加了解代码结构，那么 JVM 在编译代码的时候就做出相应的优化。

我们将在后面讲解这些优化策略，这里，先举一个简单的例子：我们知道 equals() 这个方法存在于每一个 Java Object 中（因为是从 Object class 继承而来）而且经常被覆写。当解释器遇到 b = obj1.equals(obj2) 这样一句代码，它则会查询 obj1 的类型从而得知到底运行哪一个 equals() 方法。而这个动态查询的过程从某种程度上说是很耗时的。

寄存器和主存

其中一个最重要的优化策略是编译器可以决定何时从主存取值，何时向寄存器存值。考虑下面这段代码：

```
public class RegisterTest {
 private int sum;

 public void calculateSum(int n) {
 for (int i = 0; i < n; ++i) {
 sum += i;
 }
 }
}
```

在某些时刻，sum 变量居于主存之中，但是从主存中检索值是开销很大的操作，需要多次循环才可以完成操作。正如上面的例子，如果循环的每一次都是从主存取值，性能是非常低的。相反，编译器加载一个寄存器给 sum 并赋予其初始值，利用寄存器里的值来执行循环，并将最终的结果从寄存器返回给主存。这样的优化策略则是非常高效的。但是线程的同步对于这种操作来说是至关重要的，因为一个线程无法得知另一个线程所使用的寄存器里变量的值，线程同步可以很好的解决这一问题。

寄存器的使用是编译器的一个非常普遍的优化。

回到之前的例子，JVM 注意到每次运行代码时，obj1 都是 java.lang.String 这种类型，那么 JVM 生成的被编译后的代码则是直接调用 String.equals() 方法。这样代码的执行将变得非常快，因为不仅它是被编译过的，而且它会跳过查找该调用哪个方法的步骤。

当然过程并不是上面所述这样简单，如果下次执行代码时，obj1 不再是 String 类型了，JVM 将不得不再生成新的字节码。尽管如此，之后执行的过程中，还是会变的更快，因为同样会跳过查找该调用哪个方法的步骤。这种优化只会在代码被运行和观察一段时间之后发生。这也就是为什么 JIT 编译器不会理解编译代码而是选择等待然后再去编译某些代码片段的第二个原因。

参考

IBM Developer Java blog文章：

https://www.ibm.com/developerworks/cn/java/j-lo-just-in-time/index.html

深入Java虚拟机 之七：Javac编译与JIT编译

https://blog.csdn.net/ns_code/article/details/18009455

https://www.jianshu.com/p/04fcd0ea5af7