李 伟军, 宋 翰瀛, 和 杨 翔宇  
2016 年 5 月 31 日发布

[Weibo](http://service.weibo.com/share/share.php?url=http%3A%2F%2Fwww.ibm.com%2Fdeveloperworks%2Fcn%2Fjava%2Fj-lo-just-in-time%2Findex.html%25&title=%E6%B7%B1%E5%85%A5%E6%B5%85%E5%87%BA%20JIT%20%E7%BC%96%E8%AF%91%E5%99%A8&language=zh_cn)[Google+](https://plus.google.com/share?url=http%3A%2F%2Fwww.ibm.com%2Fdeveloperworks%2Fcn%2Fjava%2Fj-lo-just-in-time%2Findex.html&t=%E6%B7%B1%E5%85%A5%E6%B5%85%E5%87%BA%20JIT%20%E7%BC%96%E8%AF%91%E5%99%A8)[用电子邮件发送本页面](mailto:?subject=%E6%B7%B1%E5%85%A5%E6%B5%85%E5%87%BA%20JIT%20%E7%BC%96%E8%AF%91%E5%99%A8&body=http%3A%2F%2Fwww.ibm.com%2Fdeveloperworks%2Fcn%2Fjava%2Fj-lo-just-in-time%2Findex.html)

[Comments](https://www.ibm.com/developerworks/cn/java/j-lo-just-in-time/index.html#icomments)

[4](https://www.ibm.com/developerworks/cn/java/j-lo-just-in-time/index.html" \l "icomments)

JIT 简介

JIT 是 just in time 的缩写, 也就是即时编译编译器。使用即时编译器技术，能够加速 Java 程序的执行速度。下面，就对该编译器技术做个简单的讲解。

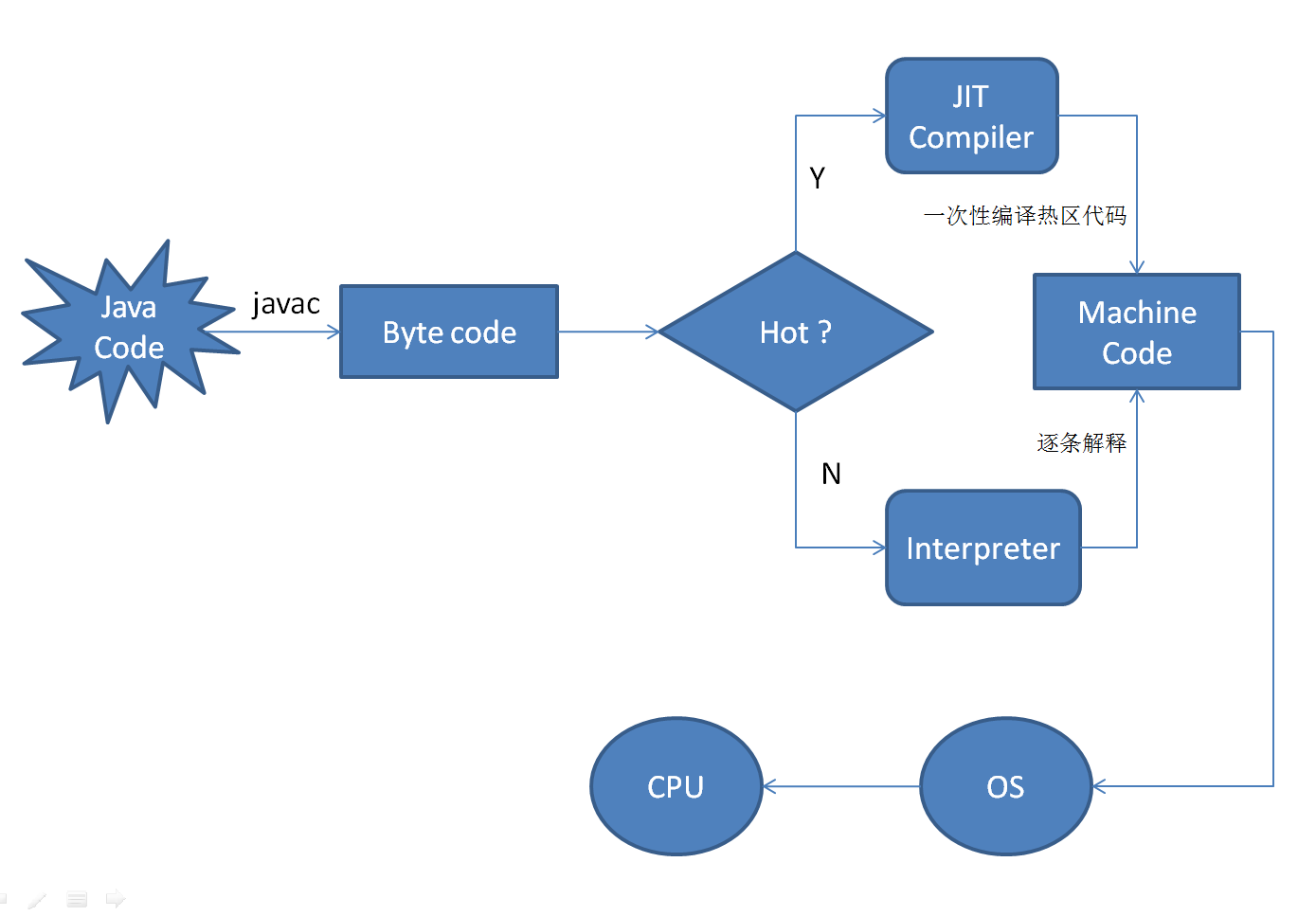
首先，我们大家都知道，通常通过 javac 将程序源代码编译，转换成 java 字节码，JVM 通过解释字节码将其翻译成对应的机器指令，逐条读入，逐条解释翻译。很显然，经过解释执行，其执行速度必然会比可执行的二进制字节码程序慢很多。为了提高执行速度，引入了 JIT 技术。

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。

图 1. JIT 工作原理图



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：

清单 1. 主存 or 寄存器测试代码

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 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，有关于线程同步的知识，我们将在后续文章中进行讲解。

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。

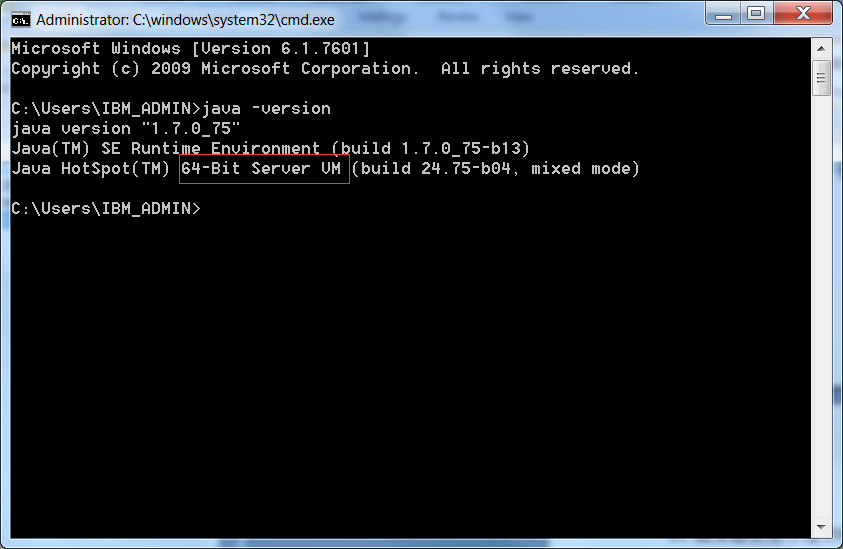
当然过程并不是上面所述这样简单，如果下次执行代码时，obj1 不再是 String 类型了，JVM 将不得不再生成新的字节码。尽管如此，之后执行的过程中，还是会变的更快，因为同样会跳过查找该调用哪个方法的步骤。这种优化只会在代码被运行和观察一段时间之后发生。这也就是为什么 JIT 编译器不会理解编译代码而是选择等待然后再去编译某些代码片段的第二个原因。

初级调优：客户模式或服务器模式

JIT 编译器在运行程序时有两种编译模式可以选择，并且其会在运行时决定使用哪一种以达到最优性能。这两种编译模式的命名源自于命令行参数（eg: -client 或者 -server）。JVM Server 模式与 client 模式启动，最主要的差别在于：-server 模式启动时，速度较慢，但是一旦运行起来后，性能将会有很大的提升。原因是：当虚拟机运行在-client 模式的时候，使用的是一个代号为 C1 的轻量级编译器，而-server 模式启动的虚拟机采用相对重量级代号为 C2 的编译器。C2 比 C1 编译器编译的相对彻底，服务起来之后，性能更高。

通过 java -version 命令行可以直接查看当前系统使用的是 client 还是 server 模式。例如：

图 2. 查看编译模式



中级编译器调优

大多数情况下，优化编译器其实只是选择合适的 JVM 以及为目标主机选择合适的编译器（-cient，-server 或是-xx:+TieredCompilation）。多层编译经常是长时运行应用程序的最佳选择，短暂应用程序则选择毫秒级性能的 client 编译器。

优化代码缓存

当 JVM 编译代码时，它会将汇编指令集保存在代码缓存。代码缓存具有固定的大小，并且一旦它被填满，JVM 则不能再编译更多的代码。

我们可以很容易地看到如果代码缓存很小所具有的潜在问题。有些热点代码将会被编译，而其他的则不会被编译，这个应用程序将会以运行大量的解释代码来结束。

这是当使用 client 编译器模式或分层编译时很频繁的一个问题。当使用普通 server 编译器模式时，编译合格的类的数量将被填入代码缓存，通常只有少量的类会被编译。但是当使用 client 编译器模式时，编译合格的类的数量将会高很多。

在 Java 7 版本，分层编译默认的代码缓存大小经常是不够的，需要经常提高代码缓存大小。大型项目若使用 client 编译器模式，则也需要提高代码缓存大小。

现在并没有一个好的机制可以确定一个特定的应用到底需要多大的代码缓存。因此，当需要提高代码缓存时，这将是一种凑巧的操作，一个通常的做法是将代码缓存变成默认大小的两倍或四倍。

可以通过 –XX:ReservedCodeCacheSize=Nflag（N 就是之前提到的默认大小）来最大化代码缓存大小。代码缓存的管理类似于 JVM 中的内存管理：有一个初始大小（用-XX:InitialCodeCacheSize=N 来声明）。代码缓存的大小从初始大小开始，随着缓存被填满而逐渐扩大。代码缓存的初始大小是基于芯片架构（例如 Intel 系列机器，client 编译器模式下代码缓存大小起始于 160KB，server 编译器模式下代码缓存大小则起始于 2496KB）以及使用的编译器的。重定义代码缓存的大小并不会真正影响性能，所以设置 ReservedCodeCacheSize 的大小一般是必要的。

再者，如果 JVM 是 32 位的，那么运行过程大小不能超过 4GB。这包括了 Java 堆，JVM 自身所有的代码空间（包括其本身的库和线程栈），应用程序分配的任何的本地内存，当然还有代码缓存。

所以说代码缓存并不是无限的，很多时候需要为大型应用程序来调优（或者甚至是使用分层编译的中型应用程序）。比如 64 位机器，为代码缓存设置一个很大的值并不会对应用程序本身造成影响，应用程序并不会内存溢出，这些额外的内存预定一般都是被操作系统所接受的。

编译阈值

在 JVM 中，编译是基于两个计数器的：一个是方法被调用的次数，另一个是方法中循环被回弹执行的次数。回弹可以有效的被认为是循环被执行完成的次数，不仅因为它是循环的结尾，也可能是因为它执行到了一个分支语句，例如 continue。

当 JVM 执行一个 Java 方法，它会检查这两个计数器的总和以决定这个方法是否有资格被编译。如果有，则这个方法将排队等待编译。这种编译形式并没有一个官方的名字，但是一般被叫做标准编译。

但是如果方法里有一个很长的循环或者是一个永远都不会退出并提供了所有逻辑的程序会怎么样呢？这种情况下，JVM 需要编译循环而并不等待方法被调用。所以每执行完一次循环，分支计数器都会自增和自检。如果分支计数器计数超出其自身阈值，那么这个循环（并不是整个方法）将具有被编译资格。

这种编译叫做栈上替换（OSR），因为即使循环被编译了，这也是不够的：JVM 必须有能力当循环正在运行时，开始执行此循环已被编译的版本。换句话说，当循环的代码被编译完成，若 JVM 替换了代码（前栈），那么循环的下个迭代执行最新的被编译版本则会更加快。

标准编译是被-XX:CompileThreshold=Nflag 的值所触发。Client 编译器模式下，N 默认的值 1500，而 Server 编译器模式下，N 默认的值则是 10000。改变 CompileThreshold 标志的值将会使编译器相对正常情况下提前（或推迟）编译代码。在性能领域，改变 CompileThreshold 标志是很被推荐且流行的方法。事实上，您可能知道 Java 基准经常使用此标志（比如：对于很多 server 编译器来说，经常在经过 8000 次迭代后改变次标志）。

我们已经知道 client 编译器和 server 编译器在最终的性能上有很大的差别，很大程度上是因为编译器在编译一个特定的方法时，对于两种编译器可用的信息并不一样。降低编译阈值，尤其是对于 server 编译器，承担着不能使应用程序运行达到最佳性能的风险，但是经过测试应用程序我们也发现，将阈值从 8000 变成 10000，其实有着非常小的区别和影响。

检查编译过程

中级优化的最后一点其实并不是优化本身，而是它们并不能提高应用程序的性能。它们是 JVM（以及其他工具）的各个标志，并可以给出编译工作的可见性。它们中最重要的就是--XX:+PrintCompilation（默认状态下是 false）。

如果 PrintCompilation 被启用，每次一个方法（或循环）被编译，JVM 都会打印出刚刚编译过的相关信息。不同的 Java 版本输出形式不一样，我们这里所说的是基于 Java 7 版本的。

编译日志中大部分的行信息都是下面的形式：

清单 2. 日志形式

|  |  |
| --- | --- |
| 1 | timestamp compilation\_id attributes (tiered\_level) method\_name size depot |

这里 timestamp 是编译完成时的时间戳，compilation\_id 是一个内部的任务 ID，且通常情况下这个数字是单调递增的，但有时候对于 server 编译器（或任何增加编译阈值的时候），您可能会看到失序的编译 ID。这表明编译线程之间有些快有些慢，但请不要随意推断认为是某个编译器任务莫名其妙的非常慢。

用 jstat 命令检查编译

要想看到编译日志，则需要程序以-XX:+PrintCompilation flag 启动。如果程序启动时没有 flag，您可以通过 jstat 命令得到有限的可见性信息。

Jstat 有两个选项可以提供编译器信息。其中，-compile 选项提供总共有多少方法被编译的总结信息（下面 6006 是要被检查的程序的进程 ID）：

清单 3 进程详情

|  |  |
| --- | --- |
| 1  2  3 | % jstat -compiler 6006  CompiledFailedInvalid TimeFailedTypeFailedMethod  206 0 0 1.97 0 |

注意，这里也列出了编译失败的方法的个数信息，以及编译失败的最后一个方法的名称。

另一种选择，您可以使用-printcompilation 选项得到最后一个被编译的方法的编译信息。因为 jstat 命令有一个参数选项用来重复其操作，您可以观察每一次方法被编译的情况。举个例子：

Jstat 对 6006 号 ID 进程每 1000 毫秒执行一次： %jstat –printcompilation 6006 1000，具体的输出信息在此不再描述。

高级编译器调优

这一节我们将介绍编译工作剩下的细节，并且过程中我们会探讨一些额外的调优策略。调优的存在很大程度上帮助了 JVM 工程师诊断 JVM 自身的行为。如果您对编译器的工作原理很感兴趣，这一节您一定会喜欢。

编译线程

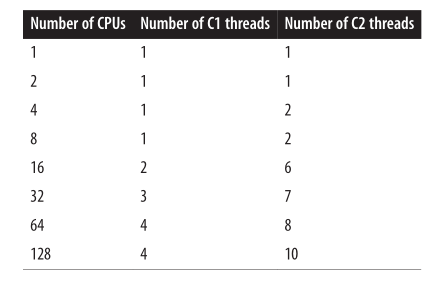
从前文中我们知道，当一个方法（或循环）拥有编译资格时，它就会排队并等待编译。这个队列是由一个或很多个后台线程组成。这也就是说编译是一个异步的过程。它允许程序在代码正在编译时被继续执行。如果一个方法被标准编译方式所编译，那么下一个方法调用则会执行已编译的方法。如果一个循环被栈上替换方式所编译，那么下一次循环迭代则会执行新编译的代码。

这些队列并不会严格的遵守先进先出原则：哪一个方法的调用计数器计数更高，哪一个就拥有优先权。所以即使当一个程序开始执行，并且有大量的代码需要编译，这个优先权顺序将帮助并保证最重要的代码被优先编译（这也是为什么编译 ID 在 PrintComilation 的输出结果中有时会失序的另一个原因）。

当使用 client 编译器时，JVM 启动一个编译线程，而 server 编译器有两个这样的线程。当分层编译生效时，JVM 会基于某些复杂方程式默认启动多个 client 和 server 线程，涉及双日志在目标平台上的 CPU 数量。如下图所示：

分层编译下 C1 和 C2 编译器线程默认数量：

图 3. C1 和 C2 编译器默认数量



编译器线程的数量可以通过-XX:CICompilerCount=N flag 进行调节设置。这个数量是 JVM 将要执行队列所用的线程总数。对于分层编译，三分之一的（至少一个）线程被用于执行 client 编译器队列，剩下的（也是至少一个）被用来执行 server 编译器队列。

在何时我们应该考虑调整这个值呢？如果一个程序被运行在单 CPU 机器上，那么只有一个编译线程会更好一些：因为对于某个线程来说，其对 CPU 的使用是有限的，并且在很多情况下越少的线程竞争资源会使其运行性能更高。然而，这个优势仅仅局限于初始预热阶段，之后，这些具有编译资格的方法并不会真的引起 CPU 争用。当一个股票批处理应用程序运行在单 CPU 机器上并且编译器线程被限制成只有一个，那么最初的计算过程将比一般情况下快 10%（因为它没有被其他线程进行 CPU 争用）。迭代运行的次数越多，最初的性能收益就相对越少，直到所有的热点方法被编译完性能收益也随之终止。

结束语

本文详细介绍了 JIT 编译器的工作原理。从优化的角度讲，最简单的选择就是使用 server 编译器的分层编译技术，这将解决大约 90%左右的与编译器直接相关的性能问题。最后，请保证代码缓存的大小设置的足够大，这样编译器将会提供最高的编译性能。