Java 理论与实践: 动态编译与性能测量

**简介：** 为动态编译的语言（例如 Java）编写和解释性能评测，要比为静态编译的语言（例如 C 或 C++）编写困难得多。在这期的 Java 理论与实践 中，Brian Goetz 介绍了动态编译使性能测试复杂的诸多原因中的一些。

这个月，我着手撰写一篇文章，分析一个写得很糟糕的微评测。毕竟，我们的程序员一直受性能困扰，我们也都想了解我们编写、使用或批评的代码的性能特征。当我偶然间写到性能这个主题时，我经常得到这样的电子邮件：“我写的这个程序显示，动态 frosternation 要比静态 blestification 快，与您上一篇的观点相反!”许多随这类电子邮件而来的所谓“评测“程序，或者它们运行的方式，明显表现出他们对于 JVM 执行字节码的实际方式缺乏基本认识。所以，在我着手撰写这样一篇文章（将在未来的专栏中发表）之前，我们先来看看 JVM 幕后的东西。理解动态编译和优化，是理解如何区分微评测好坏的关键（不幸的是，好的微评测很少）。

**动态编译简史**

Java 应用程序的编译过程与静态编译语言（例如 C 或 C++）不同。静态编译器直接把源代码转换成可以直接在目标平台上执行的机器代码，不同的硬件平台要求不同的编译器。 Java 编译器把 Java 源代码转换成可移植的 JVM 字节码，所谓字节码指的是 JVM 的“虚拟机器指令”。与静态编译器不同，javac 几乎不做什么优化 —— 在静态编译语言中应当由编译器进行的优化工作，在 Java 中是在程序执行的时候，由运行时执行。

第一代 JVM 完全是解释的。JVM 解释字节码，而不是把字节码编译成机器码并直接执行机器码。当然，这种技术不会提供最好的性能，因为系统在执行解释器上花费的时间，比在需要运行的程序上花费的时间还要多。

**即时编译**

对于证实概念的实现来说，解释是合适的，但是早期的 JVM 由于太慢，迅速获得了一个坏名声。下一代 JVM 使用即时 (JIT) 编译器来提高执行速度。按照严格的定义，基于 JIT 的虚拟机在执行之前，把所有字节码转换成机器码，但是以惰性方式来做这项工作：JIT 只有在确定某个代码路径将要执行的时候，才编译这个代码路径（因此有了名称“ *即时* 编译”）。这个技术使程序能启动得更快，因为在开始执行之前，不需要冗长的编译阶段。

JIT 技术看起来很有前途，但是它有一些不足。JIT 消除了解释的负担（以额外的启动成本为代价），但是由于若干原因，代码的优化等级仍然是一般般。为了避免 Java 应用程序严重的启动延迟，JIT 编译器必须非常迅速，这意味着它无法把大量时间花在优化上。所以，早期的 JIT 编译器在进行内联假设(inlining assumption)方面比较保守，因为它们不知道后面可能要装入哪个类。

虽然从技术上讲，基于 JIT 的虚拟机在执行字节码之前，要先编译字节码，但是 JIT 这个术语通常被用来表示任何把字节码转换成机器码的动态编译过程 —— 即使那些能够解释字节码的过程也算。

**HotSpot 动态编译**

HotSpot 执行过程组合了编译、性能分析以及动态编译。它没有把所有要执行的字节码转换成机器码，而是先以解释器的方式运行，只编译“热门”代码 —— 执行得最频繁的代码。当 HotSpot 执行时，会搜集性能分析数据，用来决定哪个代码段执行得足够频繁，值得编译。只编译执行最频繁的代码有几项性能优势：没有把时间浪费在编译那些不经常执行的代码上；这样，编译器就可以花更多时间来优化热门代码路径，因为它知道在这上面花的时间物有所值。而且，通过延迟编译，编译器可以访问性能分析数据，并用这些数据来改进优化决策，例如是否需要内联某个方法调用。

为了让事情变得更复杂，HotSpot 提供了两个编译器：客户机编译器和服务器编译器。默认采用客户机编译器；在启动 JVM 时，您可以指定 -server 开关，选择服务器编译器。服务器编译器针对最大峰值操作速度进行了优化，适用于需要长期运行的服务器应用程序。客户机编译器的优化目标，是减少应用程序的启动时间和内存消耗，优化的复杂程度远远低于服务器编译器，因此需要的编译时间也更少。

HotSpot 服务器编译器能够执行各种样的类。它能够执行许多静态编译器中常见的标准优化，例如代码提升( hoisting)、公共的子表达式清除、循环展开(unrolling)、范围检测清除、死代码清除、数据流分析，还有各种在静态编译语言中不实用的优化技术，例如虚方法调用的聚合内联。

**持续重新编译**

HotSpot 技术另一个有趣的方面是：编译不是一个全有或者全无(all-or-nothing)的命题。在解释代码路径一定次数之后，会把它重新编译成机器码。但是 JVM 会继续进行性能分析，而且如果认为代码路径特别热门，或者未来的性能分析数据认为存在额外的优化可能，那么还有可能用更高一级的优化重新编译代码。JVM 在一个应用程序的执行过程中，可能会把相同的字节码重新编译许多次。为了深入了解编译器做了什么，请用 -XX:+PrintCompilation 标志调用 JVM，这个标志会使编译器（客户机或服务器）每次运行的时候打印一条短消息。

**栈上(On-stack)替换**

HotSpot 开始的版本编译的时候每次编译一个方法。如果某个方法的累计执行次数超过指定的循环迭代次数（在 HotSpot 的第一版中，是 10,000 次），那么这个方法就被当作热门方法，计算的方式是：为每个方法关联一个计数器，每次执行一个后向分支时，就会递增计数器一次。但是，在方法编译之后，方法调用并没有切换到编译的版本，需要退出并重新进入方法，后续调用才会使用编译的版本。结果就是，在某些情况下，可能永远不会用到编译的版本，例如对于计算密集型程序，在这类程序中所有的计算都是在方法的一次调用中完成的。重量级方法可能被编译，但是编译的代码永远用不到。

HotSpot 最近的版本采用了称为 *栈上(on-stack)替换* (OSR) 的技术，支持在循环过程中间，从解释执行切换到编译的代码（或者从编译代码的一个版本切换到另一个版本）。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp12214/#ibm-pcon)

**那么，这与评测有什么关系？**

我向您许诺了一篇关于评测和性能测量的文章，但是迄今为止，您得到的只是历史的教训和 Sun 的 HotSpot 白皮书的老调重谈。绕这么大的圈子的原因是，如果不理解动态编译的过程，就不可能正确地编写或解释 Java 类的性能测试。（即使深入理解动态编译和 JVM 优化，也仍然是非常困难的。）

**为 Java 代码编写微评测远比为 C 代码编写难得多**

判断方法 A 是否比方法 B 更快的传统方法，是编写小的评测程序，通常叫做 *微评测*。这个趋势非常有意义。科学的方法不能缺少独立的调查。魔鬼总在细节之中。为动态编译的语言编写并解释评测，远比为静态编译的语言难得多。为了了解某个结构的性能，编写一个使用该结构的程序一点也没有错，但是在许多情况下，用 Java 编写的微评测告诉您的，往往与您所认为的不一样。

使用 C 程序时，您甚至不用运行它，就能了解许多程序可能的性能特征。只要看看编译出的机器码就可以了。编译器生成的指令就是将要执行的机器码，一般情况下，可以很合理地理解它们的时间特征。（有许多有毛病的例子，因为总是遗漏分支预测或缓存，所以性能差的程度远远超过查看机器码所能够想像的程度，但是大多数情况下，您都可以通过查看机器码了解 C 程序的性能的很多方面。）

如果编译器认为某段代码不恰当，准备把它优化掉（通常的情况是，评测到它实际上不做任何事情），那么您在生成的机器码中可以看到这个优化 —— 代码不在那儿了。通常，对于 C 代码，您不必执行很长时间，就可以对它的性能做出合理的推断。

而在另一方面，HotSpot JIT 在程序运行时会持续地把 Java 字节码重新编译成机器码，而重新编译触发的次数无法预期，触发重新编译的依据是性能分析数据积累到一定数量、装入新类，或者执行到的代码路径的类已经装入，但是还没有执行过。持续的重新编译情况下的时间测量会非常混乱、让人误解，而且要想获得有用的性能数据，通常必须让 Java 代码运行相当长的时间（我曾经看到过一些怪事，在程序启动运行之后要加速几个小时甚至数天），才能获得有用的性能数据。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp12214/#ibm-pcon)

**清除死代码**

编写好评测的一个挑战就是，优化编译器要擅长找出死代码 —— 对于程序执行的输出没有作用的代码。但是评测程序一般不产生任何输出，这就意味着有一些，或者全部代码都有可能被优化掉，而毫无知觉，这时您实际测量的执行要少于您设想的数量。具体来说，许多微评测在用 -server 方式运行时，要比用 -client 方式运行时好得多，这不是因为服务器编译器更快（虽然服务器编译器一般更快），而是因为服务器编译器更擅长优化掉死代码。不幸的是，能够让您的评测工作非常短（可能会把评测完全优化掉）的死代码优化，在处理实际做些工作的代码时，做得就不会那么好了。

**奇怪的结果**

清单 1 的评测包含一个什么也不做的代码块，它是从一个测试并发线程性能的评测中摘出来的，但是它实际测量的根本不是要评测的东西。（这个示例是从 JavaOne 2003 的演示 “The Black Art of Benchmarking” 中借用的。请参阅 [参考资料](http://www.ibm.com/developerworks/cn/java/j-jtp12214/#resources)。）

**清单 1. 被意料之外的死代码弄乱的评测**

|  |
| --- |
| public class StupidThreadTest {  public static void doSomeStuff() {  double uselessSum = 0;  for (int i=0; i<1000; i++) {  for (int j=0;j<1000; j++) {  uselessSum += (double) i + (double) j;  }  }  }  public static void main(String[] args) throws InterruptedException {  doSomeStuff();    int nThreads = Integer.parseInt(args[0]);  Thread[] threads = new Thread[nThreads];  for (int i=0; i<nThreads; i++)  threads[i] = new Thread(new Runnable() {  public void run() { doSomeStuff(); }  });  long start = System.currentTimeMillis();  for (int i = 0; i < threads.length; i++)  threads[i].start();  for (int i = 0; i < threads.length; i++)  threads[i].join();  long end = System.currentTimeMillis();  System.out.println("Time: " + (end-start) + "ms");  }  } |

表面上看， doSomeStuff() 方法可以给线程分点事做，所以我们能够从 StupidThreadBenchmark 的运行时间推导出多线程调度开支的一些情况。但是，因为 uselessSum 从没被用过，所以编译器能够判断出 doSomeStuff 中的全部代码是死的，然后把它们全部优化掉。一旦循环中的代码消失，循环也就消失了，只留下一个空空如也的 doSomeStuff。表 1 显示了使用客户机和服务器方式执行 StupidThreadBenchmark 的性能。两个 JVM 运行大量线程的时候，都表现出差不多是线性的运行时间，这个结果很容易被误解为服务器 JVM 比客户机 JVM 快 40 倍。而实际上，是服务器编译器做了更多优化，发现整个doSomeStuff 是死代码。虽然确实有许多程序在服务器 JVM 上会提速，但是您在这里看到的提速仅仅代表一个写得糟糕的评测，而不能成为服务器 JVM 性能的证明。但是如果您没有细看，就很容易会把两者混淆。

**表 1. 在客户机和服务器 JVM 中 StupidThreadBenchmark 的性能**

|  |  |  |
| --- | --- | --- |
| 线程数量 | 客户机 JVM 运行时间 | 服务器 JVM 运行时间 |
| 10 | 43 | 2 |
| 100 | 435 | 10 |
| 1000 | 4142 | 80 |
| 10000 | 42402 | 1060 |

对于评测静态编译语言来说，处理过于积极的死代码清除也是一个问题。但是，在静态编译语言中，能够更容易地发现编译器清除了大块评测。您可以查看生成的机器码，查看是否漏了某块程序。而对于动态编译语言，这些信息不太容易访问得到。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp12214/#ibm-pcon)

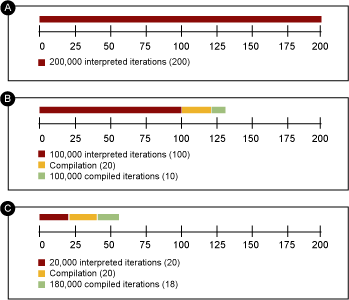
**预热**

如果您想测量 X 的性能，一般情况下您是想测量它编译后的性能，而不是它的解释性能（您想知道 X 在赛场上能跑多快）。要做到这样，需要“预热” JVM —— 即让目标操作执行足够的时间，这样编译器在为执行计时之前，就有足够的运行解释的代码，并用编译的代码替换解释代码。

使用早期 JIT 和没有栈上替换的动态编译器，有一个容易的公式可以测量方法编译后的性能：运行多次调用，启动计时器，然后执行若干次方法。如果预热调用超过方法被编译的阈值，那么实际计时的调用就有可能全部是编译代码执行的时间，所有的编译开支应当在开始计时之前发生。

而使用今天的动态编译器，事情更困难。编译器运行的次数很难预测，JVM 按照自己的想法从解释代码切换到编译代码，而且在运行期间，相同的代码路径可能编译、重新编译不止一次。如果您不处理这些事件的计时问题，那么它们会严重歪曲您的计时结果。

图 1 显示了由于预计不到的动态编译而造成的可能的计时歪曲。假设您正在通过循环计时 200,000 次迭代，编译代码比解释代码快 10 倍。如果编译只在 200,000 次迭代时才发生，那么您测量的只是解释代码的性能（时间线(a)）。如果编译在 100,000 次迭代时发生，那么您总共的运行时间是运行 200,000 次解释迭代的时间，加上编译时间（编译时间非您所愿），加上执行 100,000 次编译迭代的时间（时间线(b)）。如果编译在 20,000 次迭代时发生，那么总时间会是 20,000 次解释迭代，加上编译时间，再加上 180,000 次编译迭代（时间线(c)）。因为您不知道编译器什么时候执行，也不知道要执行多长时间，所以您可以看到，您的测量可能受到严重的歪曲。根据编译时间和编译代码比解释代码快的程度，即使对迭代数量只做很小的变化，也可能造成测量的“性能”有极大差异。

**图 1. 因为动态编译计时造成的性能测量歪曲**  
 

那么，到底多少预热才足够呢？您不知道。您能做到的最好的，就是用 -XX:+PrintCompilation 开关来运行评测，观察什么造成编译器工作，然后改变评测程序的结构，以确保编译在您启动计时之前发生，在计时循环过程中不会再发生编译。

**不要忘记垃圾收集**

那么，您已经看到，如果您想得到正确的计时结果，就必须要让被测代码比您想像的多运行几次，以便让 JVM 预热。另一方面，如果测试代码要进行对象分配工作（差不多所有的代码都要这样），那么垃圾收集器也肯定会运行。这是会严重歪曲计时结果的另一个因素 —— 即使对迭代数量只做很小的变化，也意味着没有垃圾收集和有垃圾收集之间的区别，就会偏离“每迭代时间”的测量。

如果用 -verbose:gc 开关运行评测，您可以看到在垃圾收集上耗费了多少时间，并相应地调整您的计时数据。更好一些的话，您可以长时间运行您的程序，这可以保证触发许多垃圾收集，从而更精确地分摊垃圾收集的成本。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp12214/#ibm-pcon)

**动态反优化(deoptimization)**

许多标准的优化只能在“基本块”内执行，所以内联方法调用对于达到好的优化通常很重要。通过内联方法调用，不仅方法调用的开支被清除，而且给优化器提供了更大的优化块可以优化，会带来相当大的死代码优化机会。

清单 2 显示了一个通过内联实现的这类优化的示例。 outer() 方法用参数 null 调用 inner()，结果是 inner() 什么也不做。但是通过把 inner() 的调用内联，编译器可以发现 inner() 的 else 分支是死的，因此能够把测试和 else 分支优化掉，在某种程度上，它甚至能把整个对 inner() 的调用全优化掉。如果 inner() 没有被内联，那么这个优化是不可能发生的。

**清单 2. 内联如何带来更好的死代码优化**

|  |
| --- |
| public class Inline {  public final void inner(String s) {  if (s == null)  return;  else {  // do something really complicated  }  }  public void outer() {  String s=null;  inner(s);  }  } |

但是不方便的是，虚方法对内联造成了障碍，而虚函数调用在 Java 中要比在 C++ 中普遍。假设编译器正试图优化以下代码中对 doSomething() 的调用：

|  |
| --- |
| Foo foo = getFoo();  foo.doSomething(); |

从这个代码片断中，编译器没有必要分清要执行哪个版本的 doSomething() —— 是在类 Foo 中实现的版本，还是在 Foo 的子类中实现的版本？只在少数情况下答案才明显 —— 例如 Foo 是 final 的，或者 doSomething() 在 Foo 中被定义为 final方法 —— 但是在多数情况下，编译器不得不猜测。对于每次只编译一个类的静态编译器，我们很幸运。但是动态编译器可以使用全局信息进行更好的决策。假设有一个还没有装入的类，它扩展了应用程序中的 Foo。现在的情景更像是doSomething() 是 Foo 中的 final 方法 —— 编译器可以把虚方法调用转换成一个直接分配（已经是个改进了），而且，还可以内联 doSomething()。（把虚方法调用转换成直接方法调用，叫做 *单形(monomorphic)调用变换*。)

请稍等 —— 类可以动态装入。如果编译器进行了这样的优化，然后装入了一个扩展了 Foo 的类，会发生什么？更糟的是，如果这是在工厂方法 getFoo() 内进行的会怎么样？ getFoo() 会返回新的 Foo 子类的实例？那么，生成的代码不就无效了么？对，是无效了。但是 JVM 能指出这个错误，并根据目前无效的假设，取消生成的代码，并恢复解释（或者重新编译不正确的代码路径）。

结果就是，编译器要进行主动的内联决策，才能得到更高的性能，然后当这些决策依据的假设不再有效时，就会收回这些决策。实际上，这个优化如此有效，以致于给那些不被覆盖的方法添加 final 关键字（一种性能技巧，在以前的文章中建议过）对于提高实际性能没有太大作用。

**奇怪的结果**

清单 3 中包含一个代码模式，其中组合了不恰当的预热、单形调用变换以及反优化，因此生成的结果毫无意义，而且容易被误解：

**清单 3. 测试程序的结果被单形调用变换和后续的反优化歪曲**

|  |
| --- |
| public class StupidMathTest {  public interface Operator {  public double operate(double d);  }  public static class SimpleAdder implements Operator {  public double operate(double d) {  return d + 1.0;  }  }  public static class DoubleAdder implements Operator {  public double operate(double d) {  return d + 0.5 + 0.5;  }  }  public static class RoundaboutAdder implements Operator {  public double operate(double d) {  return d + 2.0 - 1.0;  }  }  public static void runABunch(Operator op) {  long start = System.currentTimeMillis();  double d = 0.0;  for (int i = 0; i < 5000000; i++)  d = op.operate(d);  long end = System.currentTimeMillis();  System.out.println("Time: " + (end-start) + " ignore:" + d);  }  public static void main(String[] args) {  Operator ra = new RoundaboutAdder();  runABunch(ra); // misguided warmup attempt  runABunch(ra);  Operator sa = new SimpleAdder();  Operator da = new DoubleAdder();  runABunch(sa);  runABunch(da);  }  } |

StupidMathTest 首先试图做些预热（没有成功），然后测量 SimpleAdder、 DoubleAdder、 RoundaboutAdder 的运行时间，结果如表 2 所示。看起来好像先加 1，再加 2 ，然后再减 1 最快。加两次 0.5 比加 1 还快。这有可能么？（答案是：不可能。）

**表 2. StupidMathTest 毫无意义且令人误解的结果**

|  |  |
| --- | --- |
| 方法 | 运行时间 |
| SimpleAdder | 88ms |
| DoubleAdder | 76ms |
| RoundaboutAdder | 14ms |

这里发生什么呢？在预热循环之后， RoundaboutAdder 和 runABunch() 确实已经被编译了，而且编译器 Operator 和RoundaboutAdder 上进行了单形调用转换，第一轮运行得非常快。而在第二轮( SimpleAdder)中，编译器不得不反优化，又退回虚函数分配之中，所以第二轮的执行表现得更慢，因为不能把虚函数调用优化掉，把时间花在了重新编译上。在第三轮( DoubleAdder)中，重新编译比第二轮少，所以运行得就更快。（在现实中，编译器会在 RoundaboutAdder 和 DoubleAdder上进行常数替换(constant folding)，生成与 SimpleAdder 几乎相同的代码。所以如果在运行时间上有差异，那么不是因为算术代码）。哪个代码首先执行，哪个代码就会最快。

那么，从这个“评测”中，我们能得出什么结论呢？实际上，除了评测动态编译语言要比您可能想到的要微妙得多之外，什么也没得到。

[**回页首**](http://www.ibm.com/developerworks/cn/java/j-jtp12214/#ibm-pcon)

**结束语**

这个示例中的结果错得如此明显，所以很清楚，肯定发生了什么，但是更小的结果能够很容易地歪曲您的性能测试程序的结果，却不会触发您的“这里肯定有什么东西有问题”的警惕。虽然本文列出的这些内容是微评测歪曲的一般来源，但是还有许多其他来源。本文的中心思想是：您正在测量的，通常不是您以为您正在测量的。实际上，您通常所测量的，不是您以为您正在测量的。对于那些没有包含什么实际的程序负荷，测试时间不够长的性能测试的结果，一定要非常当心。