## 介绍

这是一本关于java性能优化艺术与科学的书。说它是门科学并不让人感到惊讶，关于性能的讨论包括：许多的数据、测量及分析。大多数性能工程师都有科学的背景，并应用科学的严谨性成了最大化优化性能。

艺术部分是什么？性能优化一部分是艺术，一部分是科学的概念几乎是全新的。但是对性能优化的考虑很少明确化。这是因为艺术的理念违背了我们的训练。

一部分原因是因为艺术一般是基于个人丰富的知识与经验。

性能优化的知识包括两方面：JVM本身性能，JVM的配置都会影响很多方面java程序的性能。有其它语言开发经验的开发人员都知道，去调整一些参数是很麻烦的事件，但调整JVM的话就像测试及为C++程序选择编译器，或者在php.ini中设置相应的变量一样。

第二个方面是要理解JAVA平台的功能如何影响性能。请注意，这里是说选择平台，有些功能，如线程、同步是语言的一部分，而有些功能，如处理xml是java api的一部分。虽然java语言和API有很重要的区别，但在这里我们看作一样的。JVM的性能优化是基于调整相应标识，但更多的是依赖于实际开发的应用代码。

## 性能测试方法

本章讨论获取性能测试结果的四个原则。

**测试一个真正的应用**

第一个原则是应该在一个真实的产品上进行测试。通俗的讲，有三种类型的代码可以进行测试，每种都有其相应的优缺点。而实际的应用将会提供最好的测试结果。

**微观测试**

这些类型的第一个就是微观测试。微观测试是一种测试设计，用来测试很多小单元的性能：测试调用一个同步和非同步的方法所需要的时间；测试创建线程与使用线程池所消耗的开销；测试对已经实施的方案换用另一种算法的时间等等。

微观测试看起来是一个不错的方法，但它有时很难正确的表达效果，考虑如下代码，它试图测试一个计算50的Fibonacci数方法的两种实现的性能。

|  |
| --- |
| **public void** doTest() {  *// Main Loop*  **double** l;  **long** then = System.currentTimeMillis();  **for** (**int** i = 0; i < nLoops; i++) {  l = fibImpl1(50);  }  **long** now = System.currentTimeMillis();  System.out.println("Elapsed time: " + (now - then));  }  ...  **private double** fibImpl1(**int** n) {  **if** (n < 0) **throw new** IllegalArgumentException("Must be > 0");  **if** (n == 0) **return** 0d;  **if** (n == 1) **return** 1d;  **double** d = fibImpl1(n - 2) + fibImpl(n - 1);  **if** (Double.isInfinite(d)) **throw new** ArithmeticException("Overflow");  **return** d;  } |

这个看起来很简单，但这段代码中却有许多的问题。

**微观测试需要使用它们的结果**

这代码最大的问题是从未实际改变程序的状态。因为这Fibonacci计算的结果从未被使用，所以编译器会自动丢掉其计算的结果，智能的编译器（包括java7与8）会执行如下代码：

|  |
| --- |
| **long** then = System.currentTimeMillis();  **long** now = System.currentTimeMillis();  System.out.println("Elapsed time: " + (now - then)); |

其结果，将只有几毫秒，不管Fibonacci循环多少次。怎样消除这种循环将在第4章给出。这有一个方法围绕特定的方式：确保每个结果都被读取，而不是简单的写。改变l的定义，从一个局部变量改成实例变量（用volatile进行声明），就可以允许测量方法的性能。把l改成volatile实例变量的原因将在第9章给出。

|  |
| --- |
| 线程化的微观测试  在此例中volatile 变量也适用于单线程中。在考虑使用线程化的微观测试时要特别的小心。当多个线程执行小段代码时，同步瓶颈的潜力是相当的大的。结果是导致在线程化的同步中花费大量的时间进行优化，而问题却很在出在那。  考虑下在微量测试中，调用一个同步方法的两种情况：  因为基本的代码很少，大部分都会在同步方法中执行，即使只有微量测量50%的代码在同步方法中，即使只有两个线程去同时执行这个同步方法的概率也是很高的。主线程运行结果会相当的缓慢，并且主线程作为额外的线程，会导致线程竞争使性能更慢。其工作是进行解决JVM竞争，而不是于以主线程目标。 |

**微观测试不应该包含其它的操作**

即使这样，潜在的隐患依然存在。这段代码只执行了一次操作：计算50的Fibonacci数。聪明的编译器能明白这一点，只执行一次循环，或者丢掉一些没操作多余的循环。

此外，执行fibImpl1（1000）与执行fibImpl1（1）所表现的性能效果是有很大不同的。测试方法不同实现的性能，输入的范围也必须考虑。

这了解决上述所说，传递参数给fibImpl1必须有所不同，最好是选择一个随机数，但也必须小心进行。

最简单的方式是如下传入一个随机数：

|  |
| --- |
| **for** (**int** i = 0; i < nLoops; i++) {  l = fibImpl1(random.nextInteger());  } |

现在执行时间包括了执行循环的时间，因为此测试时间为计算Fibonacci循环时间，加上生成随机数的时间。但那看起来不是想要的目标。

在微观测试中，输入的值必须预先计算，如下例：

|  |
| --- |
| **int**[] input = **new int**[nLoops];  **for** (**int** i = 0; i < nLoops; i++) {  input[i] = random.nextInt();  }  **long** then = System.currentTimeMillis();  **for** (**int** i = 0; i < nLoops; i++) {  **try** {  l = fibImpl1(input[i]);  } **catch** (IllegalArgumentException iae) {  }  }  **long** now = System.currentTimeMillis(); |

**微观测试必须测试正确的输入**

第三个缺陷是选择随机值不一定代表代码将被如何使用.在方法运行一半时异常的抛出（任何负数都会）。当输入值大于1476也会抛出，因为这是double可以表示的最大Fibonacci数。这样会导致运行的速度会大大的加快，但异常状态并没有被检测到，直到程序运行结束。考虑下改进的方案：

|  |
| --- |
| **public double** fibImplSlow(**int** n) {  **if** (n < 0) **throw new** IllegalArgumentException("Must be > 0");  **if** (n > 1476) **throw new** ArithmeticException("Must be < 1476");  **return** verySlowImpl(n);  } |

很难想像这个实现比原始的实现还要慢，但假设有人设计并在这在使用这代码。

和原来的实现做比较，原来的实现比新实现输入的范围要宽很多以致比原来的实现快很多—简单，因为在方法的开始处就进行了范围的检查。

如果，在现实世界中，用户仅传一个小于100的值给这方法，那么这比较将会给我们一个错误的答案。通常情况下，fibImpl()方法都比较快，像第1章解释的那样，要优化一般的情况。（这显示是个人刑为的例子，想通过简单的加入一个边界测试来使原来的实现比较其它实现都好，那是不太可能的）

|  |
| --- |
| 怎么热身呢？ |