17 | Benchmark测试（上）：如何做好微基准测试？

2021-06-24 尉刚强

《性能优化高手课》 课程介绍 >

讲述：尉刚强
时长 18:40 大小 17.10M

你好，我是尉刚强。从这节课开始，我们就进入了课程的第三个模块：性能看护篇。接下来，我们会用 5 节课的时间，来学习和掌握性能测试的核心理论、测试工具的选择和使用，并理解如何才能更好地集成在流水线中监控软件产品性能的能力。

今天，我们先来了解下基准测试（Benchmark）的分类，并重点学习下在进行微基准测试时都会碰到哪些问题，以及高效实现微基准测试的方法步骤和手段。

现在，我想先问你一个问题：软件为什么要进行基准测试呢？

实际上，从软件生命周期的视角来看，由于新需求的不断引入，导致软件实现在持续不断地演进与变化，而在这个过程中，软件的熵会不断增大，同时软件的性能也很容易被不断

地劣化。所以说，性能优化是一个持续改进的过程，如果没有好的措施来看护软件的性能基线，就很容易导致软件系统的性能长期处于不稳定的状态。

那么，**基准测试的目的，就是为软件系统获取一个已知的基线水平。**这样，当软件修改变化导致性能发生劣化的时候，我们就可以在第一时间发现问题。

但是，如何对软件系统做好基准测试，是一件非常有挑战的事情！我举个简单的例子，有些互联网 SaaS 服务在进行性能测试时，需要很大规模的用户接入，可是这在测试场景下是很难构造的。

另外，基准测试按照被测系统规模，可以分为微基准测试与宏基准测试。其中，**微基准测试**主要针对的是**软件编码实现层面**上的性能基线测试，而**宏基准测试**则是针对**产品系统级**所开展的性能基线测试。

所以今天这节课，我会先给你介绍下微基准测试中面临的一些核心挑战与难点，带你分析如何才能做好微基准测试。至于宏基准测试的相关知识点，我会在下节课给你讲解。

不过在开始之前，我还要说明一点，就是由于微基准测试与编程语言实现的相关性比较大，所以接下来，我主要是从程序员使用非常多的 Java 语言为出发点，来给你介绍微基准测试面临的问题。

OK，下面我们就从 Java 软件程序的微基准测试开始，来了解下即时编译对代码实现性能测试的影响吧。

JIT 对代码实现性能测试影响

事实上，对于 Java 软件程序来说，进行微基准测试其实存在很大的挑战，而这其中最大的挑战就来自于 **JIT**（Just In Time），也就是 JDK 中的 HotSpot 虚拟机的即时编译技术。

JIT 技术会在程序运行过程中，寻找到热点代码，并将这部分代码提前编译成机器码保存起来，这样在下次运行时就可以避免解释执行，而是可以直接运行机器码，以此提升系统性能。

那么 JIT 又是如何影响微基准测试呢？下面我就通过几个场景案例，来给你介绍说明下。

首先，在代码运行的过程中，JIT 中会对一些比较小的函数方法实施**内联优化**，也就是将一个函数方法（对象方法）生成的指令直接插入到被调用函数的指令内，这样就可以通过减少函数调用开销来提升执行性能。

然后，针对程序中 For 循环频繁执行的代码块，JIT 也会根据循环执行次数来决定是否启动编译优化，当满足一定的次数门限后，就会实施**栈上替换（OSR）**，也就是把循环体内生成的字节码替换为编译好的机器码来加速执行，从而导致 For 循环在不同遍历中的执行代码和运行时间不一致。

同时，JIT 的代码优化是实时动态的行为，会受制于 Code Cache 的大小限制。所以，如果优化后的运行效果不理想，JIT 还会触发**逆优化**，它的功能是把原来放到 Code Cache 中的机器码删除掉，这部分代码又回退为 Java 字节码执行。

所以综上所述，这些技术手段其实都会造成代码的执行时间发生变化，进一步就会影响微基准测试（但这只是 JIT 即时优化技术中很小的一部分，这里我们只需明白 JIT 技术会影响到代码的微基准测试结果即可）。

而除了各种技术手段的影响之外，还有一个原因，就是 Java 虚拟机在运行期存在两种模式：Client 模式和 Server 模式。Client 模式主要追求编译期的优化速度，而 Server 模式更关注运行期的性能，所以**针对这两种模式，JIT 进行热点代码优化的默认策略并不一样**，这也会直接影响到微基准测试的结果。

那么根据以上的分析，我们怎样才能避免 JIT 对微基准性能测试带来如此大的干扰呢？

答案就是**使用充足的代码预热**。也就是说，你首先需要将 Java 的被测代码循环执行很多次，以确保代码已经被 JIT 优化过，然后再对该段代码进行微基准测试，来获取测量值（如何更方便地进行预热，我会在后面的 JMH 测试框架部分讲解）。

> 补充：在 C/C++ 语言中，由于在编译期间，所有代码都被编译转换成了汇编指令，所以在对代码段进行性能测试时，并不需要这个单独的预热阶段。

所以简而言之，微基准测试就是对代码执行时间的一项测量活动，而既然是对时间的测量，肯定就会受到测量精度的影响。

那么，针对 Java 而言，测量时间的精度是否需要满足微基准测试的需求呢？下面我们就一起来探讨下这个问题。

测量时间的精度问题

在现实世界中，我们会使用手表来计算时间间隔，如果手表上的时间最小单位是秒，那么你可以大致认为测量出的时间间隔误差小于秒。而在计算机系统中，当测量时间使用更小的单位之后，那测量时间间隔的误差是否仍然小于最小的时间单位呢？

这个答案其实是否定的。因为**对于计算机系统来说，通常测量获取的时间不是准确的**。这要怎么理解呢？接下来我给你举个具体的例子。

在 Java 语言中，测试时间的方法通常会使用 **System.currentTimeMillis()**，这是一个获取系统当前时刻距离 1970 年 1 月 1 日的毫秒偏移量值，因为返回值是一个 long 类型的数字，所以可以帮助我们更方便地计算时间间隔。

不过，虽然这个接口获取的时间偏移是基于 ms（毫秒）单位的，但受制于底层实现的差异，每次获取时间的准确度并不确定，甚至有些场景下获取的时间偏差可能会超过 10ms。

因此为了解决这个问题，Java 语言中后来引入了一个 **System.nanoTime() 方法**，这是一个获取系统当前时刻与之前某一个时刻的偏移值，可以支持我们记录更精准的时间间隔。它可以获取更小的时间单位 ns（纳秒），但同样的，这并不代表误差会小于 ns。

> 补充：目前测量时间间隔的最精确方法是，通过指令获取代码运行期间，CPU 中的时钟寄存器差值，再根据 CPU 的时钟周期频率来计算出时间间隔。这种方式在做 C/C++ 实时系统的运行时间分析时，使用得比较多，但它也受制于 CPU 的指令级发射机制和编译乱序优化的影响，测试出来的时间间隔也会存在一定的误差。

实际上，针对较小的代码段运行时间测不准的问题，**微基准测试的一种可行方式**，就是迭代、累积运行多次后获取的测试时间间隔，然后再平均到每一次的运行时间上，这样就可以减少获取的时间间隔误差对测量结果的影响。

但这里仍然存在一个问题，就是**对代码段迭代很多次，又容易触发 JIT 中的栈上替换（OSR）优化**，可真实的业务代码在执行过程中并没有出现 JIT，也没有触发 OSR。所以

这样就会导致基准测试值不能反映真实的业务性能水平问题，你也需要注意规避。

总而言之，针对 Java 语言，在进行微基准测试时，我们不能太依赖底层接口获取的测量时间精度，因为 Java 的底层无法保证测量精度是非常准确的。

不过，除了测量时间精度会对测量结果产生影响以外，由于软件代码本身的运行时间也是不确定的，所以针对这种情况，我们在做微基准测试的时候，还需要在基于波动的测量结果的前提下，来尽量准确地获取平均测量结果，以此支撑性能分析。

那么接下来，我们就具体来看看测量结果数据的波动现象。

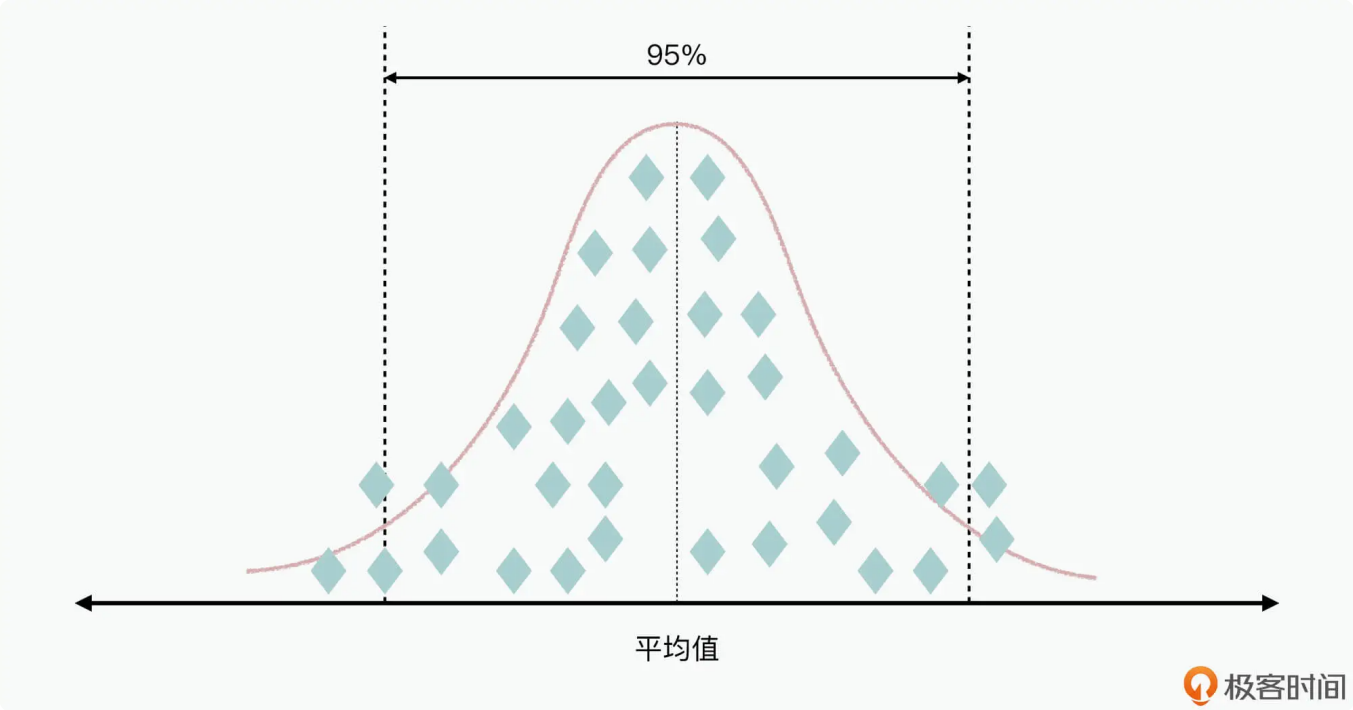
测量结果数据波动现象

这里我们要先明确一点，就是我们不可能完全剥离掉测试时软硬件运行环境的影响，也不可能完全避免测试结果的计算误差，**我们必须客观接受获取的测量结果存在波动的这种现象**。

那么，由于测试性能获取的结果会是一直波动的，所以根据单次结果去判断性能是否退化，其实也会比较困难。

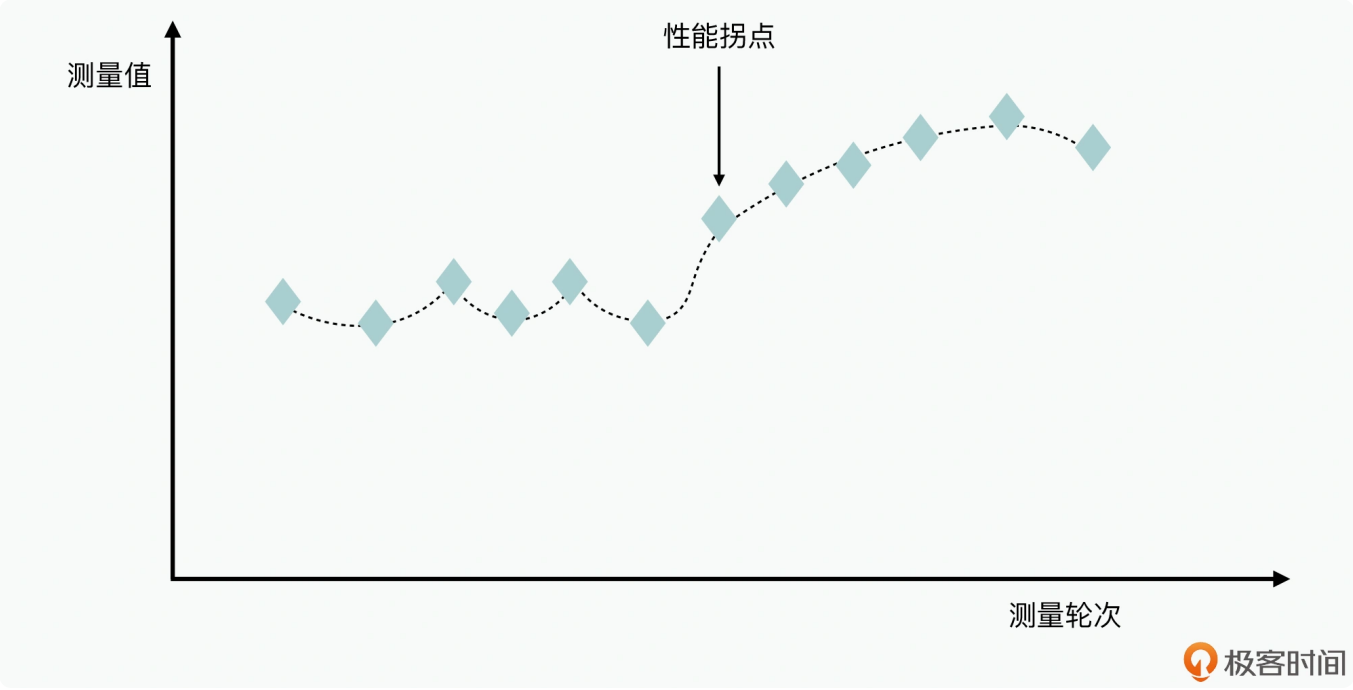
所以在这个基础上，我们可以基于统计学方法，先测量计算出性能测试结果的波动范围区间，也就是**置信区间**，然后根据测试结果是否落在置信区间，来判断性能基线是否发生变化。

可是这样问题就来了：如何计算出测试结果的波动范围区间呢？我们先来看一张示意图：



如上图所示，你可以获取大量的测试值并计算出平均值，假设你觉得 95% 左右的测量结果为可信数据，那么你就可以选择平均值周围 95% 的测量结果的最大值与最小值范围，作为置信区间。

实际上，判断微基准测试的性能是否发生变化，还有一个更有效的手段，就是**使用图表**协助分析测试结果的变化趋势。



如上图所示，绿色菱形为每一轮基准测量结果，其中你会比较容易看到一个性能拐点。这是因为图表携带了比置信区间更多的有效信息，更容易进行准确判断。另外，对于性能基

线微基准测试而言，它的目标也并不在于追求单次测试结果的准确性，而是要测试出性能变化走势的准确性。

OK，在基于以上微基准测试所面临的问题分析之后，现在我们就知道该如何规避这些因素，以避免影响到微基准测试结果。而接下来我们要讨论的，就是如何更好地实施执行微基准测试的具体方法。

实施微基准测试的步骤方法

一般来说，在实施微基准测试的时候，你需要根据具体的被测试代码片段，手动编码很多代码逻辑来获取测量值。但这里存在一个问题，就是你会很容易忽略前面提到的一些实现因素，从而导致测量结果不能准确反映性能。

那么，有没有什么更快速、有效的测试步骤流程呢？这里我根据以往的实践经验，给你总结了一个微基准测试的基本步骤流程，可以帮助你更好地实现微基准测试。

这个步骤方法主要分为四步：

第一步，确定被测程序的软硬件运行环境、运行器配置等，都与真实的产品环境保持一致。

第二步，合理选择**被测方法**。针对 Java 而言，首先建议你针对包级别的对外接口方法进行测试，这种类型接口方法的性能更加稳定；其次，由于本身微基准测试有一定的成本，因此仅对性能影响比较大的关键方法进行测试才更划算；最后，由于执行时间越短的方法，测试准确的困难越大，建议选择被测方法的执行时间要超过一定的门限，比如 10us 等。

第三步，开发微基准测试用例，并验证**正确性**和**准确性**。正确性不仅需要确保被测方法被正常执行，已经完成预热阶段，还需要保证被测方法运行方式与产品上线时一致；准确性需要验证测试结果值是否在一个有效的区间范围内波动，才具有指导意义。

第四步，执行测试，并导出测试结果，并通过可视化手段分析变化趋势。

不过，如果是自己手动来规避微基准测试的各种问题的话，实施起来会比较复杂。好在每种编程语言都有现成的微基准测试框架可供选择，比如对于 Java 语言来说，JMH 就是首选的微基准性能测试框架；而对 C/C++ 语言而言，Google Benchmark 则是首选的微基准测试框架。

所以接下来，我就主要来给你介绍下 Java 的 JMH 框架。

JMH 测试框架是如何帮助完成微基准测试的？

JMH（Java Macrobenchmark Harness）是一个测试 Java 或 JVM 上其他语言的微基准测试工具，它把支撑微基准测试的标准过程机制与手段都内置到了框架中，从而可以支持我们**通过注解的方式，来高效率开发微基准测试用例**。

我们来看一个例子。如以下代码段所示，我们可以**使用 @Benchmark** 来标记需要基准测试的方法，然后写一个 **main 方法**来启动基准测试：

复制代码

```
@Warmup(iterations = 3, time = 1)
@Measurement(iterations = 2, time = 1)
@BenchmarkMode({Mode.Throughput})
public class Sample {

    @Benchmark    //这里标注的方法就是一个被测函数方法
    public void helloworld() {
        System.out.println("hello world")
    }
    //
    public static void main(String[] args) throws RunnerException {
        Options opt = new OptionsBuilder()
                .include(Sample.class.getSimpleName())
                .forks(1)
                .build();

        new Runner(opt).run();  //启动基准测试
    }
}
```

另外，在 JMH 中，我们还可以**使用 @Warmup 注解来配置预热时间**。下面的代码示例中，就表示配置预热 3 轮，每轮 1 秒钟，这样就可以跳过预热阶段，来规避 JIT 编译优化对测试结果的影响。

复制代码

```
@Warmup(iterations = 3, time = 1)
```

然后，我们还可以**使用 @Measurement 注解来配置基准测试运行时间**。下面代码中表示的是配置测试 2 轮，每轮 1 秒钟，在每轮执行期间还会不断地迭代执行。因此，我们会得到两轮执行之后的一个测试结果：

```
复制代码
1 Benchmark              Mode  Cnt       Score       Error         Units
2 Sample.helloworld      thrpt   2   2703833258.555 ± 354675008.250  us/op
```

除此之外，JMH 还支持以下几种测试模式：

Throughput，表示吞吐量，测试每秒可以执行操作的次数；

Average Time，表示平均耗时，测试单次操作的平均耗时；

Sample Time，表示采样耗时，测试单次操作的耗时，包括最大、最小耗时，以及百分位耗时等；

Single Shot Time，表示只计算一次的耗时，一般用来测试冷启动的性能（不设置 JVM 预热）；

All，表示测试以上的所有指标。

这样，我们就可以通过如下的方式来选择配置前面提到的测试模式：

```
复制代码
1 @BenchmarkMode({Mode.Throughput})
```

最后，**JMH 还支持多种格式的结果输出**，比如 TEST、CSV、SCSV、JSON、LaTeX 等。如下所示，这是一个打印出 JSON 格式的命令：

```
复制代码
1 java -jar benchmark.jar -rf json
```

而且 JMH 的测试结果在导出后，还可以使用 JMH Visual 进行显示，但这个工具只显示单个测试导出结果。所以在通常情况下，为了更好地监控被测方法的性能变化趋势，我们还需要持续地导出并保存 JMH 结果，这样才能通过其他可视化手段去分析其变化趋势。

当然了，今天这节课，我主要目的是带你理解做好微基准测试的方法与步骤，所以并不会给你详细介绍 JMH 的构建配置过程，这里我给你推荐一个基于 Gradle 构建的 JMH 的样例库，你可以直接下载下来，参考开发测试用例或配置构建工程。

小结

热力学之父开尔文男爵（Lord Kelvin）曾经说过一句对性能优化领域有哲学指导意义的话：If you cannot measure it, you cannot improve it. 这句话的大致意思是，你只能优化你能测量到的性能问题。不仅如此，你也只能看护你能测量到的软件性能。

而微基准测试，正是你支撑与看护高性能编码实现的重要手段。

今天这节课，我带你理解了微基准测试会碰到问题与挑战、高效开展微基准测试的方法步骤，以及借助微基准性能测试框架来更好地协助测试的方法。其中，你需要重点关注的是做好微基准测试的理论和方法，这样当具体的测量结果不准确时，你就可以做到有的放矢，找到应对方案。

另外，通过学习今天的课程，你还可以在深入理解基线性能面临的问题与挑战的基础上，来指导在核心高性能模块软件开发的过程中，准确高效地开发微基准测试，并能够及时发现测试中存在的问题。

思考题

在真实的软件产品中，你有没有发现过哪些被测方法代码，很难保持测试态与运行态的执行方式一致的呢?

欢迎在留言区分享你的看法。如果觉得有收获，也欢迎你把今天的内容分享给更多的朋友。

上一篇 16 | 技术探索：你真的把CPU的潜能都挖掘出来了吗？

下一篇 18 | Benchmark测试（下）：如何做好宏基准测试？

精选留言 (1)

写留言

公号-技术夜未眠

2021-06-24

老师好，昨天线上出现了一个整体，在5分钟内，数据库出现卡顿，在该数据库上的所有表操作上的所有sql写执行都出现慢操作，持续了5分钟左右，5分钟后又都恢复了正常。分析了半天，没有得出有效结论，请问老师可能是什么原因？谢谢老师

作者回复: 这个可能原因比较多，不同数据库原因也不一样，可能数据库内部并发锁的问题，或者磁盘IO问题等等，可以通过数据库的慢查询日志分析下。