29 | 基准测试框架JMH（下）

2018-09-26 郑雨迪

29 | 基准测试框架JMH（下）
朗读人：郑雨迪　09′42″ | 4.45M

今天我们来继续学习基准测试框架 JMH。

@Fork 和 @BenchmarkMode

在上一篇的末尾，我们已经运行过由 JMH 项目编译生成的 jar 包了。下面是它的输出结果：

```
$ java -jar target/benchmarks.jar
...
# JMH version: 1.21
# VM version: JDK 10.0.2, Java HotSpot(TM) 64-Bit Server VM, 10.0.2+13
# VM invoker: /Library/Java/JavaVirtualMachines/jdk-10.0.2.jdk/Contents/Home/bin/java
# VM options: <none>
# Warmup: 5 iterations, 10 s each
# Measurement: 5 iterations, 10 s each
# Timeout: 10 min per iteration
# Threads: 1 thread, will synchronize iterations
```

```
# Threads: 1 thread, will synchronize iterations
# Benchmark mode: Throughput, ops/time
# Benchmark: org.sample.MyBenchmark.testMethod

# Run progress: 0,00% complete, ETA 00:08:20
# Fork: 1 of 5
# Warmup Iteration   1: 1023500,647 ops/s
# Warmup Iteration   2: 1030767,909 ops/s
# Warmup Iteration   3: 1018212,559 ops/s
# Warmup Iteration   4: 1002045,519 ops/s
# Warmup Iteration   5: 1004210,056 ops/s
Iteration   1: 1010251,342 ops/s
Iteration   2: 1005717,344 ops/s
Iteration   3: 1004751,523 ops/s
Iteration   4: 1003034,640 ops/s
Iteration   5: 997003,830 ops/s

# Run progress: 20,00% complete, ETA 00:06:41
# Fork: 2 of 5
...

# Run progress: 80,00% complete, ETA 00:01:40
# Fork: 5 of 5
# Warmup Iteration   1: 988321,959 ops/s
# Warmup Iteration   2: 999486,531 ops/s
# Warmup Iteration   3: 1004856,886 ops/s
# Warmup Iteration   4: 1004810,860 ops/s
# Warmup Iteration   5: 1002332,077 ops/s
Iteration   1: 1011871,670 ops/s
Iteration   2: 1002653,844 ops/s
Iteration   3: 1003568,030 ops/s
Iteration   4: 1002724,752 ops/s
Iteration   5: 1001507,408 ops/s



Result "org.sample.MyBenchmark.testMethod":
  1004801,393 ±(99.9%) 4055,462 ops/s [Average]
  (min, avg, max) = (992193,459, 1004801,393, 1014504,226), stdev = 5413,926
```

```
  (min, avg, max) = (992193,459, 1004801,393, 1014504,226), stdev = 5413,926
  CI (99.9%): [1000745,931, 1008856,856] (assumes normal distribution)



# Run complete. Total time: 00:08:22


...


Benchmark                Mode  Cnt        Score      Error  Units
MyBenchmark.testMethod  thrpt   25  1004801,393 ± 4055,462  ops/s
```

在上面这段输出中，我们暂且忽略最开始的 Warning 以及打印出来的配置信息，直接看接下来貌似重复的五段输出。

```
# Run progress: 0,00% complete, ETA 00:08:20
# Fork: 1 of 5
# Warmup Iteration   1: 1023500,647 ops/s
# Warmup Iteration   2: 1030767,909 ops/s
# Warmup Iteration   3: 1018212,559 ops/s
# Warmup Iteration   4: 1002045,519 ops/s
# Warmup Iteration   5: 1004210,056 ops/s
Iteration   1: 1010251,342 ops/s
Iteration   2: 1005717,344 ops/s
Iteration   3: 1004751,523 ops/s
Iteration   4: 1003034,640 ops/s
Iteration   5: 997003,830 ops/s
```

你应该已经留意到`Fork: 1 of 5`的字样。这里指的是 JMH 会 Fork 出一个新的 Java 虚拟机，来运行性能基准测试。

之所以另外启动一个 Java 虚拟机进行性能基准测试，是为了获得一个相对干净的虚拟机环境。

在介绍反射的那篇文章中，我就已经演示过因为类型 profile 被污染，而导致无法内联的情况。使用新的虚拟机，将极大地降低被上述情况干扰的可能性，从而保证更加精确的性能数据。

在介绍虚方法内联的那篇文章中，我讲解过基于类层次分析的完全内联。新启动的 Java 虚拟机，其加载的与测试无关的抽象类子类或接口实现相对较少。因此，具体是否进行完全内联将交由开发人员来决定。

关于这种情况，JMH 提供了一个性能测试案例 [1]。如果你感兴趣的话，可以下载下来自己跑一遍。

除了对即时编译器的影响之外，Fork 出新的 Java 虚拟机还会提升性能数据的准确度。

这主要是因为不少 Java 虚拟机的优化会带来不确定性，例如 TLAB 内存分配（TLAB 的大小会变化），偏向锁、轻量锁算法，并发数据结构等。这些不确定性都可能导致不同 Java 虚拟机中运行的性能测试的结果不同，例如 JMH 这一性能的测试案例 [2]。

在这种情况下，通过运行更多的 Fork，并将每个 Java 虚拟机的性能测试结果平均起来，可以增强最终数据的可信度，使其误差更小。在 JMH 中，你可以通过`@Fork`注解来配置，具体如下述代码所示：

```
@Fork(10)
public class MyBenchmark {
  ...
}
```

让我们回到刚刚的输出结果。每个 Fork 包含了 5 个预热迭代（warmup iteration，如`# Warmup Iteration 1: 1023500,647 ops/s`）以及 5 个测试迭代（measurement iteration，如`Iteration   1: 1010251,342 ops/s`）。

每个迭代后都跟着一个数据，代表本次迭代的吞吐量，也就是每秒运行了多少次操作（operations/s，或 ops/s）。默认情况下，一次操作指的是调用一次测试方法`testMethod`。

除了吞吐量之外，我们还可以输出其他格式的性能数据，例如运行一次操作的平均时间。具体的配置方法以及对应参数如下述代码以及下表所示：

```
@BenchmarkMode(Mode.AverageTime)
public class MyBenchmark {
  ...
}
```

一般来说，默认使用的吞吐量已足够满足大多数测试需求了。

@Warmup 和 @Measurement

之所以区分预热迭代和测试迭代，是为了在记录性能数据之前，将 Java 虚拟机带至一个稳定状态。

这里的稳定状态，不仅包括测试方法被即时编译成机器码，还包括 Java 虚拟机中各种自适配优化算法能够稳定下来，如前面提到的 TLAB 大小，亦或者是使用传统垃圾回收器时的 Eden 区、Survivor 区和老年代的大小。

一般来说，预热迭代的数目以及每次预热迭代的时间，需要由你根据所要测试的业务逻辑代码来调配。通常的做法便是在首次运行时配置较多次迭代，并监控性能数据达到稳定状态时的迭代数目。

不少性能评测框架都会自动检测稳定状态。它们所采用的算法是计算迭代之间的差值，如果连续几个迭代与前一迭代的差值均小于某个值，便将这几个迭代以及之后的迭代当成稳定状态。

这种做法有一个缺陷，那便是在达到最终稳定状态前，程序可能拥有多个中间稳定状态。例如通过 Java 上的 JavaScript 引擎 Nashorn 运行 JavaScript 代码，便可能出现多个中间稳定状态的情况。（具体可参考 Aleksey Shipilev 的 devoxx 2013 演讲 [3] 的第 21 页。）

总而言之，开发人员需要自行决定预热迭代的次数以及每次迭代的持续时间。

通常来说，我会在保持 5-10 个预热迭代的前提下（这样可以看出是否达到稳定状况），将总的预热时间优化至最少，以便节省性能测试的机器时间。（这在持续集成 / 回归测试的硬件资源跟不上代码提交速度的团队中非常重要。）

当确定了预热迭代的次数以及每次迭代的持续时间之后，我们便可以通过`@Warmup`注解来进行配置，如下述代码所示：

```
@Warmup(iterations=10, time=100, timeUnit=TimeUnit.MILLISECONDS, batchSize=10)
public class MyBenchmark {
  ...
}
```

`@Warmup`注解有四个参数，分别为预热迭代的次数`iterations`，每次迭代持续的时间`time`和`timeUnit`（前者是数值，后者是单位。例如上面代码代表的是每次迭代持续 100 毫秒），以及每次操作包含多少次对测试方法的调用`batchSize`。

测试迭代可通过`@Measurement`注解来进行配置。它的可配置选项和`@Warmup`的一致，这里就不再重复了。与预热迭代不同的是，每个 Fork 中测试迭代的数目越多，我们得到的性能数据也就越精确。

@State、@Setup 和 @TearDown

通常来说，我们所要测试的业务逻辑只是整个应用程序中的一小部分，例如某个具体的 web app 请求。这要求在每次调用测试方法前，程序处于准备接收请求的状态。

我们可以把上述场景抽象一下，变成程序从某种状态到另一种状态的转换，而性能测试，便是在收集该转换的性能数据。

JMH 提供了`@State`注解，被它标注的类便是程序的状态。由于 JMH 将负责生成这些状态类的实例，因此，它要求状态类必须拥有无参数构造器，以及当状态类为内部类时，该状态类必须是静态的。

JMH 还将程序状态细分为整个虚拟机的程序状态，线程私有的程序状态，以及线程组私有的程序状态，分别对应`@State`注解的参数`Scope.Benchmark`，`Scope.Thread`和`Scope.Group`。

需要注意的是，这里的线程组并非 JDK 中的那个概念，而是 JMH 自己定义的概念。具体可以参考`@GroupThreads`注解 [4]，以及这个案例 [5]。

`@State`的配置方法以及状态类的用法如下所示：

```
public class MyBenchmark {
    @State(Scope.Benchmark)
    public static class MyBenchmarkState {
      String message = "exception";
    }

    @Benchmark
    public void testMethod(MyBenchmarkState state) {
        new Exception(state.message);
    }
}
```

我们可以看到，状态类是通过方法参数的方式传入测试方法之中的。JMH 将负责把所构造的状态类实例传入该方法之中。

不过，如果`MyBenchmark`被标注为`@State`，那么我们可以不用在测试方法中定义额外的参数，而是直接访问`MyBenchmark`类中的实例变量。

和 JUnit 测试一样，我们可以在测试前初始化程序状态，在测试后校验程序状态。这两种操作分别对应`@Setup`和`@TearDown`注解，被它们标注的方法必须是状态类中的方法。

而且，JMH 并不限定状态类中`@Setup`方法以及`@TearDown`方法的数目。当存在多个`@Setup`方法或者`@TearDown`方法时，JMH 将按照定义的先后顺序执行。

JMH 对@Setup方法以及@TearDown方法的调用时机是可配置的。可供选择的粒度有在整个性能测试前后调用，在每个迭代前后调用，以及在每次调用测试方法前后调用。其中，最后一个粒度将影响测试数据的精度。

这三种粒度分别对应@Setup和@TearDown注解的参数Level.Trial, Level.Iteration, 以及Level.Invocation。具体的用法如下所示：

```
public class MyBenchmark {
  @State(Scope.Benchmark)
  public static class MyBenchmarkState {
    int count;

    @Setup(Level.Invocation)
    public void before() {
      count = 0;
    }

    @TearDown(Level.Invocation)
    public void after() {
      // Run with -ea
      assert count == 1 : "ERROR";
    }
  }

  @Benchmark
  public void testMethod(MyBenchmarkState state) {
    state.count++;
  }
}
```

即时编译相关功能

JMH 还提供了不少控制即时编译的功能，例如可以控制每个方法内联与否的@CompilerControl注解 [6]。

另外一个更小粒度的功能则是Blackhole类。它里边的consume方法可以防止即时编译器将所传入的值给优化掉。

具体的使用方法便是为被@Benchmark注解标注了的测试方法增添一个类型为Blackhole的参数，并且在测试方法的代码中调用其实例方法Blackhole.consume，如下述代码所示：

```
@Benchmark
public void testMethod(Blackhole bh) {
  bh.consume(new Object()); // prevents escape analysis
}
```

需要注意的是，它并不会阻止对传入值的计算的优化。举个例子，在下面这段代码中，我将3+4的值传入Blackhole.consume方法中。即时编译器仍旧会进行常量折叠，而Blackhole将阻止即时编译器把所得到的常量值 7 给优化消除掉。

```
@Benchmark
public void testMethod(Blackhole bh) {
  bh.consume(3+4);
}
```

除了防止死代码消除的consume之外，Blackhole类还提供了一个静态方法consumeCPU，来消耗 CPU 时间。该方法将接收一个 long 类型的参数，这个参数与所消耗的 CPU 时间呈线性相关。

总结与实践

今天我介绍了基准测试框架 JMH 的进阶功能。我们来回顾一下。

- @Fork允许开发人员指定所要 Fork 出的 Java 虚拟机的数目。
- @BenchmarkMode允许指定性能数据的格式。
- @Warmup和@Measurement允许配置预热迭代或者测试迭代的数目，每个迭代的时间以及每个操作包含多少次对测试方法的调用。
- @State允许配置测试程序的状态。测试前对程序状态的初始化以及测试后对程序状态的恢复或者校验可分别通过@Setup和@TearDown来实现。

今天的实践环节，请逐个运行 JMH 的官方案例 [7]，具体每个案例的意义都在代码注释之中。

最后给大家推荐一下 Aleksey Shipilev 的 devoxx 2013 演讲（Slides[8]；视频 [9]，请自备梯子）。如果你已经完成本专栏前面两部分，特别是第二部分的学习，那么这个演讲里的绝大部分内容你应该都能理解。

[1] http://hg.openjdk.java.net/code-tools/jmh/file/3769055ad883/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_12_Forking.java
[2] http://hg.openjdk.java.net/code-tools/jmh/file/3769055ad883/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_13_RunToRun.java
[3] https://shipilev.net/talks/devoxx-Nov2013-benchmarking.pdf
[4] http://hg.openjdk.java.net/code-tools/jmh/file/3769055ad883/jmh-core/src/main/java/org/openjdk/jmh/annotations/GroupThreads.java
[5] http://hg.openjdk.java.net/code-tools/jmh/file/3769055ad883/jmh-samples/src/main/java/org/openjdk/jmh/samples/JMHSample_15_Asymmetric.java
[6] http://hg.openjdk.java.net/code-tools/jmh/file/3769055ad883/jmh-core/src/main/java/org/openjdk/jmh/annotations/CompilerControl.java
[7] http://hg.openjdk.java.net/code-tools/jmh/file/3769055ad883/jmh-samples/src/main/java/org/openjdk/jmh/samples
[8] https://shipilev.net/talks/devoxx-Nov2013-benchmarking.pdf
[9] https://www.youtube.com/watch?v=VaWgOCDBxYw

精选留言

HELSING 0

郑老师，能不能介绍一下，为什么ZGC会快那么多啊

2018-09-26