<https://github.com/jackyu86/learnSpace/tree/master/benchmarkdemo>

在pom文件中加入依赖(在[Maven Central](http://central.maven.org/maven2/org/openjdk/jmh/jmh-core/)查看jmh-core的最新版本)：

<dependencies>

<dependency>

<groupId>org.openjdk.jmh</groupId>

<artifactId>jmh-core</artifactId>

<version>0.4.2</version>

</dependency>

</dependencies>

生成一个包含main方法的Java类。main方法中加入以下代码：

Options opt = new OptionsBuilder()

.include(".\*" + YourClass.class.getSimpleName() + ".\*")

.forks(1)

.build();

new Runner(opt).run();

测试方法使用@GenerateMicroBenchmark注解。运行该类。

**测试模式**

测试方法上@BenchmarkMode注解表示使用特定的测试模式：

| **名称** | **描述** |
| --- | --- |
| Mode.Throughput | 计算一个时间单位内操作数量 |
| Mode.AverageTime | 计算平均运行时间 |
| Mode.SampleTime | 计算一个方法的运行时间(包括百分位) |
| Mode.SingleShotTime | 方法仅运行一次(用于冷测试模式)。或者特定批量大小的迭代多次运行(具体查看后面的“@Measurement“注解)——这种情况下JMH将计算批处理运行时间(一次批处理所有调用的总时间) |
| 这些模式的任意组合 | 可以指定这些模式的任意组合——该测试运行多次(取决于请求模式的数量) |
| Mode.All | 所有模式依次运行 |

**时间单位**

使用@OutputTimeUnit指定时间单位，它需要一个标准Java类型java.util.concurrent.TimeUnit作为参数。可是如果在一个测试中指定了多种测试模式，给定的时间单位将用于所有的测试(比如，测试SampleTime适宜使用纳秒，但是throughput使用更长的时间单位测量更合适)。

**测试参数状态**

测试方法可能接收参数。这需要提供单个的参数类，这个类遵循以下4条规则：

* 有无参构造函数(默认构造函数)
* 是公共类
* 内部类应该是静态的
* 该类必须使用@State注解

@State注解定义了给定类实例的可用范围。JMH可以在多线程同时运行的环境测试，因此需要选择正确的状态。

| **名称** | **描述** |
| --- | --- |
| Scope.Thread | 默认状态。实例将分配给运行给定测试的每个线程。 |
| Scope.Benchmark | 运行相同测试的所有线程将共享实例。可以用来测试状态对象的多线程性能(或者仅标记该范围的基准)。 |
| Scope.Group | 实例分配给每个线程组(查看后面的线程组部分) |

除了将单独的类标记@State，也可以将你自己的benchmark类使用@State标记。上面所有的规则对这种情况也适用。

**状态设置和清理**

与JUnit测试类似，使用@Setup和@TearDown注解标记状态类的方法(这些方法在JMH文档中称为\_fixtures\_)。setup/teardown方法的数量是任意的。这些方法不会影响测试时间(但是Level.Invocation可能影响测量精度)。

@Setup/@TearDown注解使用Level参数来指定何时调用fixture：

| **名称** | **描述** |
| --- | --- |
| Level.Trial | 默认level。全部benchmark运行(一组迭代)之前/之后 |
| Level.Iteration | 一次迭代之前/之后(一组调用) |
| Level.Invocation | 每个方法调用之前/之后(不推荐使用，除非你清楚这样做的目的) |

**冗余代码**

冗余代码消除是microbenchmark中众所周知的问题。通常的解决方法是以某种方式使用计算结果。JMH本身不会实施对冗余代码的消除。但是如果你想消除冗余代码——**要做到测试程序返回值不为**void。**永远返回你的计算结果**。JMH将完成剩余的工作。

如果测试程序需要返回多个值，将所有这些返回值使用省时操作结合起来(省时是指相对于获取到所有结果所做操作的开销)，或者使用BlackHole作为方法参数，将所有的结果放入其中(注意某些情况下BlockHole.consume可能比手动将结果组合起来开销更大)。BlackHole是一个thread-scoped类：

@GenerateMicroBenchmark

public void testSomething( BlackHole bh )

{

bh.consume( Math.sin( state\_field ));

bh.consume( Math.cos( state\_field ));

}

**常量处理**

如果计算结果是可预见的并且不依赖于状态对象，它可能被JIT优化。因此，最好总是从状态对象读取测试的输入并且返回计算的结果。这条规则大体上用于单个返回值的情形。使用BlackHole对象JVM更难优化它(但不是不可能被优化)。下面测试的所有方法都不会被优化：

private double x = Math.PI;

@GenerateMicroBenchmark

public void bhNotQuiteRight( BlackHole bh )

{

bh.consume( Math.sin( Math.PI ));

bh.consume( Math.cos( Math.PI ));

}

@GenerateMicroBenchmark

public void bhRight( BlackHole bh )

{

bh.consume( Math.sin( x ));

bh.consume( Math.cos( x ));

}

返回单个值的情形更加复杂。下面的测试不会被优化，但是如果使用Math.log替换Math.sin，那么testWrong方法将被常量值替换。

private double x = Math.PI;

@GenerateMicroBenchmark

public double testWrong()

{

return Math.sin( Math.PI );

}

@GenerateMicroBenchmark

public double testRight()

{

return Math.sin( x );

}

因此，为使测试更可靠要严格遵守以下规则：**永远从状态对象读取测试输入并返回计算的结果**。

**循环**

**不要在测试中使用循环。JIT非常聪明，在循环中经常出现不可预料的处理。要测试真实的计算，让JMH处理剩余的部分。**

在非统一开销操作情况下(比如测试处理列表的时间，这个列表在每个测试后有所增加)，你可能使用@BenchmarkMode(Mode.SingleShotTime) 和@Measurement(batchSize = N)。但是不允许你自己实现测试的循环。

**分支**

默认JMH为每个试验(迭代集合)fork一个新的java进程。这样可以防止前面收集的“资料”——其他被加载类以及它们执行的信息对当前测试的影响。比如，实现了相同接口的两个类，测试它们的性能，那么第一个实现(目标测试类)可能比第二个快，因为JIT发现第二个实现类后就把第一个实现的直接方法调用替换为接口方法调用。

因此，**不要把**forks**设为0**，**除非你清楚这样做的目的**。

极少数情况下需要指定JVM分支数量时，使用@Fork对方法注解，就可以设置分支数量，预热(warmup)迭代数量和JVM分支的其他参数。

可能通过JMH API调用来指定JVM分支参数也有优势——可以使用一些JVM  
-XX:参数，通过JMH API访问不到它。这样就可以根据你的代码自动选择最佳的JVM设置(new Runner(opt).run()以简便的形式返回了所有的测试结果)。

**编译器提示**

可以为JIT提供关于如何使用测试程序中任何方法的提示。“任何方法”是指任何的方法——不仅仅是@GenerateMicroBenchmark注解的方法。使用@CompilerControl模式(还有更多模式，但是我不确定它们的有用程度)：

| **名称** | **描述** |
| --- | --- |
| CompilerControl.Mode.DONT\_INLINE | 该方法不能被内嵌。用于测量方法调用开销和评估是否该增加JVM的inline阈值 |
| CompilerControl.Mode.INLINE | 要求编译器内嵌该方法。通常与“Mode.DONT\_INLINE“联合使用，检查内嵌的利弊。 |
| CompilerControl.Mode.EXCLUDE | 不编译该方法——解释它。在该JIT有多好的圣战中作为有用的参数:) |

**注解控制测试**

通过注解指定JMH参数。这些注解用在类或者方法上。方法注解总是优先于类的注解。

| **名称** | **描述** |
| --- | --- |
| @Fork | 需要运行的试验(迭代集合)数量。每个试验运行在单独的JVM进程中。也可以指定(额外的)JVM参数。 |
| @Measurement | 提供真正的测试阶段参数。指定迭代的次数，每次迭代的运行时间和每次迭代测试调用的数量(通常使用@BenchmarkMode(Mode.SingleShotTime)测试一组操作的开销——而不使用循环) |
| @Warmup | 与@Measurement相同，但是用于预热阶段 |
| @Threads | 该测试使用的线程数。默认是Runtime.getRuntime().availableProcessors() |

**CPU消耗**

有时测试消耗一定CPU周期。通过静态的BlackHole.consumeCPU(tokens)方法来实现。Token是一些CPU指令。这样编写方法代码就可以达到运行时间依赖于该参数的目的(不被任何JIT/CPU优化)。

**多参数的测试运行**

很多情况下测试代码包含多个参数集合。幸运的是，要测试不同参数集合时JMH不会要求写多个测试方法。或者准确来说，测试参数是基本类型，基本包装类型或者String时，JMH提供了解决方法。

程序需要完成：

1. 定义@State对象
2. 在其中定义所有的参数字段
3. 每个字段都使用@Param注解

@Param注解使用String数组作为参数。这些字符串在任何@Setup方法被调用前转换为字段类型。然而，JMH文档中声称这些字段值在@Setup方法中不能被访问。

JMH使用所有@Param字段的输出结果。因此，如果第一个字段有2个参数，第二个字段有5个参数，测试将运行2 \* 5 \* Forks次。

**线程组——非统一的多线程**

我们已经提到@State(Scope.Benchmark)用来测试多线程访问状态对象的情形。并发程度通过用来测试的线程数量设置。

可能也需要定义对状态对象非统一访问的情况——比如测试“读取——写入”场景时，读线程数通常高于写线程数量。JMH使用线程组来应对这种情形。

为设置测试组，需要：

1. 使用@Group(name)注解标记所有的测试方法，为同一个组中的所有测试设置相同的名称(否则这些测试将独立运行——没有任何警告提示！)
2. 使用@GroupThreads(threadsNumber)注解标记每个测试，指定运行给定方法的线程数量。

JMH将启动给定组的所有@GroupThreads，并发运行相同实验中同一组的所有测试。组和每个方法的结果将单独给出。

**多线程——伪共享字段访问**

你可能知道这样一个事实，大多数现代x86 CPU有64字节的cache line(缓存行)。CPU缓存提高了数据读取速率，但同时，如果你需要从多个线程同时读写两个邻近的字段，也会产生性能瓶颈。这种情况称为“伪共享”——字段似乎是独立访问的，但是实际上它们在硬件层面的相互竞争。

这个问题通常的解决方案是两边都增加至少128字节的虚拟数据。因为JVM可以将类的字段重排序，在相同的类内部增加可能不能正确运行。

更加健壮的方法是使用类层次——JVM通常将属于同一个类的字段放在一起。比如，定义类A有一个只读字段，类B继承类A且定义16个long字段，类C继承类B定义可写字段，最后类D继承类C定义另一个16个long字段——这就防止了被分配在下一个内存中对象的写变量竞争。

以防读写的字段类型相同，也可以使用两个数据位置相互距离很远的稀疏数组。**在前面的情况中不要使用数组**——它们是对象特定类型，仅需要增加4或8字节(取决于JVM设置)。

这个问题的另一种解决方法是如果你已经用到了Java 8：在可写字段上使用@sun.misc.Contended以及-XX:-RestrictContended的JVM设置。更多细节，参见[Aleksey Shipilev的说明](http://shipilev.net/talks/jvmls-July2013-contended.pdf)。

JMH是如何解决竞争字段访问的呢？它在两边都增加了@State对象，但是这并不能在单一对象内部对个别的字段增加——需要自己来处理。

**总结**

* JMH用于各种类型的microbenchmark——每个测试从纳秒到毫秒。它关注所有可测量的逻辑，测试人员只需编写测试方法的任务代码。JMH也包含对所有类型多线程测试的内在支持——统一(所有线程运行相同代码)和非统一(线程分组，每个组运行自己的代码)。
* 如果仅仅一条规则需要记住的话，那就是——**永远从**@State**对象读取测试输入并返回计算的结果(无论结果是明确的还是通过**  
  BlackHole**对象返回)**。