codecentric Blog (https://blog.codecentric.de/)

Overview (https://blog.codecentric.de/en/category/java-en/)

Performance measurement with JMH – Java Microbenchmark Harness (https://blog.codecentric.de/en/2017/10/performance-measurement-with-jmh-java-microbenchmark-harness/)

10/23/17 by **Kevin Peters (https://blog.codecentric.de/en/author/kevin-peters/)**

No Comments (https://blog.codecentric.de/en/2017/10/performance-measurement-with-jmh-java-microbenchmark-harness/#comments)

What is benchmarking and why should we do that?
If there are multiple ways to implement a feature or if we have serious doubts about performance while using a certain technology, special implementation patterns or a new “cutting edge” library, we have to make decisions. There might be assumptions about performance effects of a certain way of implementing something, but in the end – if we do not measure and compare the different approaches – we will not be sure if our decision was correct. This is true for the big architectural topics, but also for smaller-scoped concerns such as preferring a certain API method although multiple alternatives exist. So we should stop guessing and start measuring performance! In other words, we should use benchmarks! This article introduces Java Microbenchmark Harness (JMH), an OpenJDK project which aims to ease setting up a benchmark environment for Java performance tests.

Benchmark == Benchmark?

To categorize benchmarks in a more fine-grained manner, people invented benchmark categories such as "micro", "macro", or even "meso", which separate performance tests by scope. Maybe we can say the separation is done by the affected layers or complexity level of code under test.

Microbenchmarks are performance metrics on the lowest level. You can compare them to unit tests, which means they invoke single methods or execute small pieces of business logic without "more (cross-cutting) stuff" around.

Macrobenchmarks are the opposite of that. They test entire applications similar to end-to-end tests.

Mesobenchmarks represent something in between, which means they are written to measure entire actions (features, workflows) related to bigger parts of our applications using different layers in interaction with each other without spinning up the entire application. This could be a single feature which uses authentication/authorization, reads data from a database or calls external services and so on. We could range mesobenchmarks with integration tests.

In this post I will put the focus on the smallest kind of these. So let's concentrate on the microbenchmarks.

How to implement microbenchmarks

If we want to know which methods are performing better than others, we should give it a try and compare them under equal conditions. A naive approach would be to call the different methods within some kind of common simple unit test and take a look at the time the execution takes, maybe implemented using `java.lang.System.currentTimeMillis()`. Then we could just compute the difference between start and stop timestamps and on the first view that's sufficient to get an idea about its performance – but taking a second look, it's not. We should take into account how the JVM executes and especially optimizes code. Regarding this point of view, our results would not be reliable using values we got after a single execution. There are many JVM-related optimization topics we have to keep in mind and I will give some further hints afterwards. For now it's important that the more often a line of code is executed, the more information the JVM will get about that code and it will optimize it (if possible). So if we want to measure code which will be invoked frequently in later production environments (and that code is the crucial part of our software we should measure), we should also measure it after some warmup iterations to simulate "real" production conditions. And now it's getting interesting (some people would rather say "complicated").

The question now is: How should the warmup be implemented? Use a boolean flag which separates warmup iterations from measurement iterations and switch that flag after some time? Maybe, but doing that again and again would be a cumbersome, error-prone task.

Using JHM for benchmarks

Blessedly, there is the Java Microbenchmark Harness (http://openjdk.java.net/projects/code-tools/jmh/). This is an OpenJDK project which aims to ease setting up a benchmark environment for Java performance tests. If you are already familiar with JUnit (http://junit.org/junit4/) tests (and you should be) it will be very comfortable to get started with JMH.

Set up the JMH environment

To create a maven benchmark project, just use the maven archetype (https://maven.apache.org/guides/introduction/introduction-to-archetypes.html) and provide your preferred groupId, artifactId and version.

```
mvn archetype:generate \
-DinteractiveMode=false \
-DarchetypeGroupId=org.openjdk.jmh \
-DarchetypeArtifactId=jmh-java-benchmark-archetype \
-DgroupId=com.example \
-DartifactId=jmh-number-verification-performance-test \
-Dversion=1.0
```

That command will create a skeleton project which can execute your benchmarks. After you wrote your tests (as described below), build it with `mvn clean install` . The project creates a *benchmarks.jar* in the target folder which should be used to run the measurements:

```
java -jar target/benchmarks.jar
```

Although you could use your IDE to run the tests, you should prefer this standalone JAR. It provides great portability – you can execute it on different machines – and there is no performance penalty due to any IDE overhead.

Writing benchmarks

Writing benchmarks is as simple as writing JUnit tests. The main difference is that you have to annotate a test method with `@Benchmark` instead of `@Test` . Just use the archetype-generated class (MyBenchmark), rename it or write your own class and invoke the suspicious code you want to measure within a `@Benchmark` method. JHM is doing everything else and generates a performance report for you.

As with JUnit, it is also possible to use parameterized tests. This is the purpose of the `@Param` annotation. There are a lot of examples for a bunch of use cases (http://hg.openjdk.java.net/code-tools/jmh/file/tip/jmh-samples/src/main/java/org/openjdk/jmh/samples/) available on the project site.

Long story short, to try it out, I created a benchmark which compares different approaches to check if a String represents a valid Integer value. It compares following implementations:

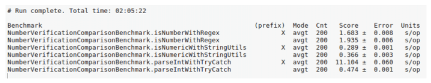
– using `try-catch` with `Integer.parseInt(String)`
– `StringUtils.isNumeric(String)`
– `String.matches("\\d+")`

Check out the example project on Github (https://github.com/kevin-peters/jmh-number-verification-performance-test). With that benchmark, we could find out which approach produces the best performance result.

Performance results

Per default, JMH executes 10 Forks (separate execution environments), 20 warmup cycles (without measurement, providing the opportunity to the JVM to optimize the code before the measurement starts) and 20 real measurement iterations for every test. Of course, this behavior can be overidden on annotation basis (`@Fork` , `@Warmup` , `@Measurement`).

The results will vary depending on the configuration of the target machine they are running on. After the runs are finished, a report will be generated. The following output is a result report generated by the mentioned comparison benchmark, running on my i7 laptop (7700 HQ, 4 x 2.8 GHz, 32 MB RAM):



```
# Run complete. Total time: 02:05:22

Benchmark                                                       (prefix)  Mode  Cnt   Score    Error  Units
NumberVerificationComparisonBenchmark.isNumberWithRegex                X  avgt  200   1.683 ±  0.008   s/op
NumberVerificationComparisonBenchmark.isNumberWithRegex                   avgt  200   1.935 ±  0.006   s/op
NumberVerificationComparisonBenchmark.isNumericWithStringUtils         X  avgt  200   0.289 ±  0.001   s/op
NumberVerificationComparisonBenchmark.isNumericWithStringUtils            avgt  200   0.366 ±  0.003   s/op
NumberVerificationComparisonBenchmark.parseIntWithTryCatch             X  avgt  200  11.104 ±  0.060   s/op
NumberVerificationComparisonBenchmark.parseIntWithTryCatch                avgt  200   0.474 ±  0.001   s/op
```

(https://blog.codecentric.de/files/2017/10/result.png)

Since you can execute your benchmarks in different `@BenchmarkModes` you have to read the results differently. E.g. in `Mode.AverageTime` a lower score is preferred, while using `Mode.Throughput` a higher value points to better performance.

Beware of the JVM optimizations

As mentioned before, the JVM will optimize code based on collected information during execution. Usually this is a good thing we should appreciate, at least for production environments. But under artificial conditions (our microbenchmark definitely is one) this could cause problems. Here are some topics you should be aware of:

Warmup:

The first big obstacle is conquered using JHM itself – it delivers the warmup cycles out of the box. So the JVM can collect some information about the code under test and the effectively executed code will be more "production-like" than a once executed method ever could be.

Always read computed results:

If you don't use code (e.g. if you never read a private variable), the JVM is free to discard that code during compilation. This is called "Dead Code Elimination", which means that even the entire computation of these dead results will probably be eliminated if no one is interested in. This will definitely distort your benchmark results and can lead to false conclusions. So take an interest (or at least pretend) and read your computation results even if they are not relevant for your test. This could be done either by returning result variables or throwing it into a so-called `Blackhole` injected by declaring it as input parameter for your benchmark method.

Differences to production code:

We're done with this short introduction to JMH and although we talked about *reliable* results, please be aware of the fact that code in tests will always behave differently to the same code executed during production. There are a lot of optimizations the JVM will do afterwards, e.g. depending how often methods are invoked (hot code), call hierarchies and stack depth. So performance tests are at most a good hint, but no guarantee. The best thing you can do is measure performance in production using metrics or with profiling.

Blockchain in the Supply Chain – A Practical Introduction [blockcentric #2]
(https://blog.codecentric.de/en/2017/10/blockchain-in-the-supply-chain/)

Tags

BENCHMARK (HTTPS://BLOG.CODECENTRIC.DE/EN/TAG/BENCHMARK/) JAVA (HTTPS://BLOG.CODECENTRIC.DE/EN/TAG/JAVA-EN-2/)

JMH (HTTPS://BLOG.CODECENTRIC.DE/EN/TAG/JMH/) PERFORMANCE TESTING (HTTPS://BLOG.CODECENTRIC.DE/EN/TAG/PERFORMANCE-TESTING/)

Kevin Peters (https://blog.codecentric.de/en/author/kevin-peters/)

"It's all about data!" – Every application reads and writes information, so Kevin places great emphasis on the smooth and high-performance handling of these data. He has specialized in the use of the Spring Framework and Hibernate and he is also interested in new technologies. Agile software development and the "Clean Code" concept are further foundations of his work.

(http://www.facebook.com/sharer.php?u=https://blog.codecentric.de/en/2017/10/performance-measurement-with-jmh-java-microbenchmark-harness/)

(http://twitter.com/share?url=https://blog.codecentric.de/en/2017/10/performance-measurement-with-jmh-java-microbenchmark-harness/&text=Performance+measurement+with+JMH+%26%238211%3B+Java+Microbenchmark+Harness)

(https://www.linkedin.com/shareArticle?mini=true&url=https://blog.codecentric.de/en/2017/10/performance-measurement-with-jmh-java-microbenchmark-harness/)

(http://reddit.com/submit?url=https://blog.codecentric.de/en/2017/10/performance-measurement-with-jmh-java-microbenchmark-harness/&title=Performance+measurement+with+JMH+%26%238211%3B+Java+Microbenchmark+Harness)

Post by **Kevin Peters**

ARCHITECTURE

Polite HTTP API design – "Use the headers, Luke!" (https://blog.codecentric.de/en/2017/09/polite-http-api-design-use-the-headers-luke/)

SPRING BOOT

How to enable the Spring Boot 'Run Dashboard' in IntelliJ IDEA 2017.2.1 (https://blog.codecentric.de/en/2017/09/how-to-enable-the-spring-boot-dashboard-in-intellij-idea-2017-2-1/)

More content about **Java**

JAVA

Improve your test structure with Lambdas and Mockito's Answer (https://blog.codecentric.de/en/2018/02/improve-test-structure-lambdas-mockitos-answer/)

JAVA

DRY in the 21st Century (https://blog.codecentric.de/en/2018/01/dry-in-the-21st-century/)

Comment

Nachricht

Name

E-Mail

SUBMIT

☐ Notify me of followup comments via e-mail