Java Performance Tuning Guide

Java performance tuning guide – various tips on improving performance of your Java code

Introduction to JMH

by Mikhail Vorontsov

11 Sep 2014: Article was updated for JMH 1.0.

10 May 2014: Original version.

Introduction

This article will give you an overview of basic rules and abilities of JMH. The second article will give you an overview of JMH profilers.

JMH is a new microbenchmarking framework (first released late-2013). Its distinctive advantage over other frameworks is that it is developed by the same guys in Oracle who implement the JIT. In particular I want to mention Aleksey Shipilev and his brilliant blog. JMH is likely to be in sync with the latest Oracle JRE changes, which makes its results very reliable.

You can find JMH examples here.

JMH has only 2 requirements (everything else are recommendations):

- You need to create a maven project using a command from the JMH official web page
- You need to annotate test methods with `@Benchmark` annotation

In some cases, it is not convenient to create a new project just for the performance testing purposes. In this situation you can rather easily add JMH into an existing project. You need to make the following steps:

1. Ensure your project directory structure is recognizable by Maven (your benchmarks are at src/main/java at least)
2. Copy 2 JMH maven dependencies and `maven-shade-plugin` from the JMH archetype. No other plugins mentioned in the archetype are required at the moment of writing (JMH 1.0).

How to run

Run the following maven command to create a template JMH project from an archetype (it may change over the time, check for the latest version near the start of the the official JMH page):

```
$ mvn archetype:generate \
          -DinteractiveMode=false \
          -DarchetypeGroupId=org.openjdk.jmh \
          -DarchetypeArtifactId=jmh-java-benchmark-archetype \
          -DgroupId=org.sample \
          -DartifactId=test \
          -Dversion=1.0
```

Alternatively, copy 2 JMH dependencies and `maven-shade-plugin` from the JMH archetype (as described above).

Create one (or a few) java files. Annotate some methods in them with `@Benchmark` annotation – these would be your performance benchmarks.

You have at least 2 simple options to run your tests::

Follow the procedure from the official JMH page):

```
$ cd your_project_directory/
$ mvn clean install
$ java -jar target/benchmarks.jar
```

The last command should be entered **verbatim** – regardless of your project settings you will end up with `target/benchmarks.jar` sufficient to run all your tests. This option has a slight disadvantage – it will use the default JMH settings for all settings not provided via annotations (`@Fork, @Warmup` and `@Measurement` annotations are getting nearly mandatory in

this mode). Use `java -jar target/benchmarks.jar -h` command to see all available command line options (there are plenty).

Or use the old way: add `main` method to some of your classes and write a JMH start script inside it. Here is an example:

```
Options opt = new OptionsBuilder()
                .include(".*" + YourClass.class.getSimpleName() + ".*")
                .forks(1)
                .build();
new Runner(opt).run();
```

After that you can run it with `target/benchmarks.jar` as your classpath:

```
$ cd your_project_directory/
$ mvn clean install
$ java -cp target/benchmarks.jar your.test.ClassName
```

Now after extensive “how to run it” manual, let’s look at the framework itself.

Test modes

You can use the following test modes specified using `@BenchmarkMode` annotation on the test methods:

Name	Description
Mode.Throughput	Calculate number of operations in a time unit.
Mode.AverageTime	Calculate an average running time.
Mode	Calculate how long does it take for a method to run (including percentiles).

e.SampleTime	
Mode.SingleShotTime	Just run a method once (useful for cold-testing mode). Or more than once if you have specified a batch size for your iterations (see `@Measurement` annotation below) – in this case JMH will calculate the batch running time (total time for all invocations in a batch).
Any set of these modes	You can specify any set of these modes – the test will be run several times (depending on number of requested modes).
Mode.All	All these modes one after another.

Time units

You can specify time unit to use via `@OutputTimeUnit`, which requires an argument of the standard Java type `java.util.concurrent.TimeUnit`. Unfortunately, if you have specified several test modes for one test, the given time unit will be used for all tests (for example, it may be convenient to measure `SampleTime` in nanoseconds, but throughput should better be measured in the longer time units).

State of test arguments

Your test methods can accept arguments. You could provide a single argument of a class which complies to 4 following rules:

- There should be a no-arg constructor (default constructor).
- It should be a public class.
- Inner classes should be static.
- Class must be annotated with `@State` annotation.

`@State` annotation defines the scope in which an instance of a given class will be available. JMH allows you to run tests in multiple threads simultaneously, so choose the right state:

Name	Description
Scope.Thread	This is a default state. An instance will be allocated for each thread running the given test.
Scope.Benchmark	An instance will be shared across all threads running the same test. Could be used to test multithreaded performance of a state object (or just mark your benchmark with this scope).
Scope.Group	An instance will be allocated per thread group (see Groups section down below).

Besides marking a separate class as a `@State`, you can also mark your own benchmark class as `@State`. All above scope rules apply to this case as well.

State housekeeping

Like JUnit tests, you can annotate your state class methods with `@Setup` and `@TearDown` annotations (these methods called *fixtures* in JMH documentation. You can have any number of setup/teardown methods. These methods do not contribute anything to test times (but `Level.Invocation` may affect precision of measurements).

You can specify when to call fixtures by providing a `Level` argument for `@Setup`/`@TearDown` annotations:

Name	Description
Level.Trial	This is a default level. Before/after entire benchmark run (group of iteration)
Level.Iteration	Before/after an iteration (group of invocations)
Level.Invocation	Before/after every method call (this level is not recommended until you know what you are doing)

Dead code

Dead code elimination is a well known problem among microbenchmark writers. The general solution is to use the result of calculations somehow. JMH does not do any magic tricks on its own. If you want to defend against dead code elimination – **never write `void` tests. Always return the result of your calculations.** JMH will take care of the rest.

If you need to return more than one value from your test, either combine all return values with some cheap operation (cheap compared to the cost of operations by which you got your results) or use a `BlackHole` method argument and sink all your results into it (note that `BlockHole.consume` may be more expensive than manual combining of results in some cases). `BlackHole` is a thread-scoped class:

```
1 @Benchmark
2 public void testSomething( BlackHole bh )
3 {
4     bh.consume( Math.sin( state_field ));
5     bh.consume( Math.cos( state_field ));
6 }
```

Constant folding

If result of your calculation is predictable and does not depend on state objects, it is likely to be optimized by JIT. So, always read the test input from a state object and return the result of your calculations. This rule is mostly related to the case of a single return value. Using `BlackHole` object makes it much harder for JVM to optimize it (but not impossible!). Both methods in the following test will not be optimized.

```
private double x = Math.PI;

@Benchmark
public void bhNotQuiteRight( BlackHole bh )
{
    bh.consume( Math.sin( Math.PI ));
    bh.consume( Math.cos( Math.PI ));
}

@Benchmark
public void bhRight( BlackHole bh )
{
    bh.consume( Math.sin( x ));
    bh.consume( Math.cos( x ));
}
```

Things are getting more complicated in case of a method returning a single value. The following tests will not be optimized, but if you will replace `Math.sin` with `Math.log`, then `testWrong` method will be replaced with a constant value:

```
private double x = Math.PI;

@Benchmark
public double testWrong()
{
    return Math.sin( Math.PI );
}

@Benchmark
public double testRight()
{
    return Math.sin( x );
}
```

So, in order to make your tests reliable, stick to the following rule: **always read the test input from a state object and return the result of your calculations.**

Loops

Do not use loops in your tests. JIT is too smart and often does magic tricks with loops. Test the actual calculation and let JMH to take care of the rest.

In case of non-uniform cost operations (for example, you test time to process a list which grows after each test) you may want to use `@BenchmarkMode(Mode.SingleShotTime)` with `@Measurement(batchSize = N)`. But you must not implement test loops yourself!

Forks

By default JHM forks a new java process for each trial (set of iterations). This is required to defend the test from previously collected "profiles" – information about other loaded classes and their execution information. For example, if you have 2 classes implementing the same interface and test the performance of both of them, then the first implementation (in order of testing) is likely to be faster than the second one (in the same JVM), because JIT replaces direct method calls to the first implementation with interface method calls after discovering the second implementation.

So, **do not set `forks` to zero until you know what you are doing.**

In the rare cases when you need to specify number of forked JVMs, use `@Fork` test method annotation, which allows you to set number of forks, number of warmup iterations and the (extra) arguments for the forked JVM(s).

It may be useful to specify the forked JVM arguments via JMH API calls – it may allow you to provide JVM some `-XX:` arguments, which are not accessible via JMH API. It will allow you to automatically choose the best JVM settings for your critical code (remember that `new Runner(opt).run()` returns all test results in a convenient form).

Compiler hints

You can give the JIT a hint how to use any method in your test program. By "any method" I mean any method – not just those annotated by `@Benchmark`. You can use following `@CompilerControl` modes (there are more, but I am not sure about their usefulness):

Name	Description
CompilerControl.Mode.DONT_INLINE	This method should not be inlined. Useful to measure the method call cost and to evaluate if it worth to increase the inline threshold for the JVM.
CompilerControl.Mode.INLINE	Ask the compiler to inline this method. Usually should be used in conjunction with `Mode.DONT_INLINE` to check pros and cons of inlining.
CompilerControl.Mod-	Do not compile this method – interpret it instead. Useful in holy wars as an argument how good

e.EXCLUDE	is the JIT 🙂

Test control annotations

You can specify JMH parameters via annotations. These annotations could be applied to either classes or methods. Method annotations always win.

Name	Description
`@Fork`	Number of trials (sets of iterations) to run. Each trial is started in a separate JVM. It also lets you specify the (extra) JVM arguments.
`@Measurement`	Allows you to provide the actual test phase parameters. You can specify number of iterations, how long to run each iteration and number of test invocations in the iteration (usually used with `@BenchmarkMode(Mode.SingleShotTime)` to measure the cost of a **group** of operations – instead of using loops).
`@Warmup`	Same as `@Measurement`, but for warmup phase.
`@Threads`	Number of threads to use for the test. The default is `Runtime.getRuntime().availableProcessors()`.

CPU burning

From time to time you may want to burn some CPU cycles inside your tests. This could be done via a static `BlackHole.consumeCPU(tokens)` method. Token is a few CPU instructions. Method code is written so that the time to run this method will depend linearly on its argument (defensive against any JIT/CPU optimizations).

Running a test with a set of parameters

In many situations you need to test your code with several sets of parameters. Luckily, JMH does not force you to write N test methods if you need to test N sets of parameters. Or, to be more precise, JMH will help you if your test parameters are primitives, primitive wrappers or Strings.

All you need to do is:

1. Define a `@State` object
2. Define all your parameters fields in it
3. Annotate each of these fields with `@Param` annotation

`@Param` annotation expects an array of `String` arguments. These strings will be converted to the field type before any `@Setup` method invocations. Nevertheless, JMH documentation claims that these field values may not be accessible in `@Setup` methods.

JMH will use an outer product of all `@Param` fields. So, if you have 2 parameters on the first field and 5 parameters on the second field, your test will be executed `2 * 5 * Forks` times.

Thread groups – non uniform multithreading

We have already mentioned that `@State(Scope.Benchmark)` annotation could be used to test the case of multithreaded access to the state object. The degree of concurrency will be set by the number of threads which should be used for testing.

You may also need to define the non-uniform access to your state object – for example to test the "readers-writers" scenario where the number of readers is usually higher than the number of writers. JMH uses the notion of thread groups for this case.

In order to setup a group of tests, you need:

1. Mark all your test methods with `@Group(name)` annotation, providing the same string name for all tests in a group (otherwise these tests will be run independently – no warning will be given!).
2. Annotate each of your tests with `@GroupThreads(threadsNumber)` annotation, specifying a number of threads which will run the given method.

JMH will start a sum of all your `@GroupThreads` for the given group and will run all tests in a group concurrently in the same trial. The results will be given for the group and for each method independently.

Multithreading – False shared field access

You probably know about the fact that most modern x86 CPUs have 64 byte cache lines. CPU cache allows you to read data at great rates, but at the same time it creates a performance bottleneck if you have to read and write 2 adjacent fields from 2 or more threads at the same time. Such event is called "false sharing" – while fields seem to be accessed independently, they actually contend with each other on the hardware level.

The general solution to this problem is to pad such fields with at least 128 bytes of dummy data on both sides. Padding inside the same class may not work properly because JVM is allowed to reorder class fields in any order.

The more robust solution is to use class hierarchies – JVM usually puts all fields which belong to the same class together. For example, we can define class A with a *read* access field, extend it with a class B defining 16 `long` fields, extend class B with class C defining a *write* access field and finally (that's important) extend class C with class D defining another 16 `long` fields – this will prevent contended access to a *write* variable from the object which will be located next in memory.

In case when *read* and *write* fields have the same type, you can also use a sparse array with 2 cells located far enough from each other. **Do not use arrays as padding in the previous case** – they are a special type of ob-

ject and will contribute only 4 or 8 bytes (depending on your JVM settings) to padding.

There is another way to solve this problem if you are already using Java 8: use `@sun.misc.Contended` annotation for *write* fields and use `-XX:-RestrictContended` JVM key. For more details, take a look at Aleksey Shipilev's presentation.

How JMH can help you with contended field access? It pads your `@State` objects from both sides, but it can not help you to pad individual fields inside a single object – this is left to yourself.

Summary

- JMH is useful for all sorts of microbenchmarking – from nanoseconds to seconds per test. It takes care of all measurement logic, leaving you just a task of writing the test method(s). JMH also contains built-in support for all sorts of multithreaded tests – both uniform (all threads run the same code) and non-uniform (there are several groups of threads, each of them is running each own code).
- If you have to remember just one JMH rule, it should be: **always read test input from `@State` objects and return the result of your calculations (either explicitly or via a `BlackHole` object).**
- JMH is started differently since JMH 0.5: now you have to add one more dependency to your pom file and use `maven-shade-plugin`. It generates `target/benchmarks.jar` file, which contains all the code required to run all tests in your project.

This entry was posted in CPU optimization, Intermediate, Memory optimization, Overviews and tagged JMH, microbenchmarking, Oracle on September 13, 2014 [http://java-performance.info/jmh/] .

---

12 thoughts on "Introduction to JMH"

Pingback: Introduction to JMH Profilers | Java Performance Tuning Guide

Pingback: Java Annotated Monthly – May 2014 | JetBrains IntelliJ IDEA Blog

Pingback: String switch performance | Java Performance Tuning Guide

Pingback: Exceptions are slow in Java - Java Performance Tuning Guide