Java Microbenchmarks with JMH, Part 2

blog.avenuecode.com/java-microbenchmarks-with-jmh-part-2

perm_identity Andre Brait
schedule 4/3/19 2:00 PM

In last week's blog, we discussed the basics of creating a JMH project and getting it to run. We also proposed a small benchmark and showed what the implementation would look like. Today, we'll be looking into how to configure JMH, how it works, and why certain parameters are necessary. (Be sure to join us next week for tips on how **not** to create benchmarks!)

Code Recap

In the first part of this series, we showed a comparison of different ways to use multiple threads to get the total sum of a list of integers, as in the code below:

```
package com.avenuecode.snippet;

import org.openjdk.jmh.annotations.Benchmark;
import org.openjdk.jmh.annotations.BenchmarkMode;
import org.openjdk.jmh.annotations.Fork;
import org.openjdk.jmh.annotations.Level;
import org.openjdk.jmh.annotations.Mode;
import org.openjdk.jmh.annotations.Param;
import org.openjdk.jmh.annotations.Scope;
import org.openjdk.jmh.annotations.Setup;
import org.openjdk.jmh.annotations.State;
import org.openjdk.jmh.annotations.Warmup;
import org.openjdk.jmh.infra.Blackhole;

import java.util.List;
import java.util.Random;
import java.util.concurrent.atomic.AtomicLong;
```

```
import java.util.concurrent.atomic.LongAdder;
import java.util.stream.Collectors;

public class MyBenchmark {

@State(Scope.Benchmark)
public static class BenchmarkState {
@Param({"1000000", "10000000", "100000000"})
public int listSize;

public List<Integer> testList;

@Setup(Level.Trial)
public void setUp() {
testList = new Random()
.ints()
.limit(listSize)
.boxed()
.collect(Collectors.toList());
}
}

public static class VolatileLong {
private volatile long value = 0;

public synchronized void add(long amount) {
this.value += amount;
}

public long getValue() {
```

```
return this.value;
}
}

@Fork(value = 1, warmups = 1)
@Warmup(iterations = 1)
@Benchmark
@BenchmarkMode(Mode.AverageTime)
public void longAdder(Blackhole blackhole, BenchmarkState state) {
LongAdder adder = new LongAdder();
state.testList.parallelStream().forEach(adder::add);
blackhole.consume(adder.sum());
}

@Fork(value = 1, warmups = 1)
@Warmup(iterations = 1)
@Benchmark
@BenchmarkMode(Mode.AverageTime)
public void atomicLong(Blackhole blackhole, BenchmarkState state) {
AtomicLong atomicLong = new AtomicLong();
state.testList.parallelStream().forEach(atomicLong::addAndGet);
blackhole.consume(atomicLong.get());
}

@Fork(value = 1, warmups = 1)
@Warmup(iterations = 1)
@Benchmark
@BenchmarkMode(Mode.AverageTime)
```

```
    public void volatileLong(Blackhole blackHole, BenchmarkState state) {
    VolatileLong volatileLong = new VolatileLong();
    state.testList.parallelStream().forEach(volatileLong::add);
    blackHole.consume(volatileLong.getValue());
    }

    @Fork(value = 1, warmups = 1)
    @Warmup(iterations = 1)
    @Benchmark
    @BenchmarkMode(Mode.AverageTime)
    public void longStreamSum(Blackhole blackHole, BenchmarkState state) {
    long sum = state.testList.parallelStream().mapToLong(s -> s).sum();
    blackHole.consume(sum);
    }

    }
```

view raw MyBenchmark-2.java hosted with ❤ by GitHub
We'll refer to this code throughout the rest of this article.

Configuring Your Benchmark

First things first: we can see there's a number of annotations that have been added to our code there. Those are JMH-specific annotations that allow us to tune and configure the benchmarks to suit our needs and fit our constraints. Let's explore what each one means:

Benchmark Modes:

JMH allows users to determine what they want to measure. There are 5 modes in which JMH can run a benchmark:

1. **Throughput**: measures the number of times your benchmark method could be executed (an operation, in the JMH idiom) per time unit.
2. **Average time**: measures the average time it takes for your benchmark method to be executed.
3. **Sample time**: measures the time it takes for the benchmark method to run, including the minimum, maximum, etc. times.

4. **Single shot time**: measures how long a single benchmark method execution takes to run; this is helpful for testing how it performs under a cold start (no JVM warm up).
5. **All**: all the above; "all" is mostly used to test JMH itself while developing it.

You can specify in which mode each benchmark is to be executed using the `@BenchmarkMode` annotation. Each benchmark can have its own configuration here. If the annotation is not used, the default mode is throughput.

You can see we used the **Average time** mode for the benchmarks in our code since we're interested in knowing which is the fastest approach.

Warmup:

JMH performs a few runs of a given benchmark and then discards the results. That constitutes the warmup phase, and its role is to allow the JVM to perform any class loading, compilation to native code, and caching steps it would normally do in a long-running application before starting to collect actual results.

The difference between running the code with and without a warmup can be quite noticeable, so it's recommended that the benchmark is allowed to warm up at least a few times.

The number of iterations per warmup phase can be configured using the `@Warmup` annotation. The default value is 5.

Fork:

There are many things that can affect the performance of an application in the Java world, such as OS-related things like memory alignment when a process is created, the moments the GC pauses occur, differences in how the JIT behaves (both as a whole and in each execution), etc.

For instance, if two benchmarks deal with classes that implement the same interface, the benchmark that gets executed first might tend to be faster than the other one since the JIT compiler may replace direct method calls to the first implementation with interface method calls when it discovers the second implementation. That could produce inaccurate results since the order of execution of the benchmarks can affect the performance.

That is precisely what happened in the **bad benchmark example** in Part 1. Each one of those methods ran from inside a class that implemented `Runnable`. Doing nothing was slower than doing something merely because the former was run after the latter. (A more detailed explanation can be found here.)

So, how can JMH deal with these issues? Simply put, it avoids them by forking the JVM process for each set of benchmark trials. The number of times this is done can be controlled with the `@Fork` annotation. It's highly recommended that at least 1 fork is used. Here again,

the default is 5.

State

Sometimes you'll want to initialize some variables that your benchmark code needs, but which you do not want to be part of the code your benchmark measures. Such variables are called **state** variables. State variables are declared in special state classes, and an instance of that state class can then be *provided as parameter to the benchmark method*, like in the example.

The state class needs to be annotated with the `@State` annotation, and it needs to adhere to some standards:

1. The class must be declared **public**;
2. If the class is a nested class, it must be declared `static` (e.g. `public static class`);
3. The class must have a public no-arg constructor (no parameters to the constructor).

State Scope:

A state object can be reused across multiple calls to your benchmark method. JMH provides different **scopes** that the state object can be reused in. These are:

1. **Thread**: each thread running the benchmark will create its own instance of the state object.
2. **Group**: each thread group running the benchmark will create its own instance of the state object.
3. **Benchmark**: all threads running the benchmark share the same state object.

Since this is a single-threaded benchmark (even though it uses multiple threads inside each benchmark), the scope is not all that important, but we'll be using the benchmark scope in this example.

Setup and Teardown:

State objects can provide methods that are used by JMH either during the setup of the state object, before it's passed as argument to a benchmark, or after the benchmark is run. To specify such methods, annotate them with the `@Setup` and the `@Teardown` annotations.

The `@Setup` and `@Teardown` annotations can receive an argument that controls when to run the corresponding methods. Those arguments, called Levels, can be:

1. **Trial**: the method is called once for each full run of the benchmark, that is, a full fork including all warmup and benchmark iterations.
2. **Iteration**: the method is called once for each iteration of the benchmark.
3. **Invocation**: the method is called once for each call to the benchmark method.

In the example, we've annotated the setup method with `@Setup(Level.Trial)` in order to create a new list of random integers for each trial of benchmark executions.

Parameterizing the Benchmark

It might be interesting, for a number of reasons, to run the same benchmark for a given set of parameters. One common reason to do this is scaling. Even slow solutions can appear to be fast if the number of items on which they operate is small enough. Another common use-case is testing different parameters for a configurable algorithm or data structure, like the max depth of a queue or the number of reader and writer threads.

There's no need to run the benchmark separately for each one of those cases. A state object can contain a field on which JMH will inject a set of parameterized values, one for each set of benchmark runs. This field needs to be annotated with the `@Param` annotation, which receives as argument an array of strings.

In the example, the `@Param` was used to set a different list size for each set of executions (1 million, 10 million and 100 million).

The Results

Finally, it's time! Upon running the example, we get something analogous to the following:

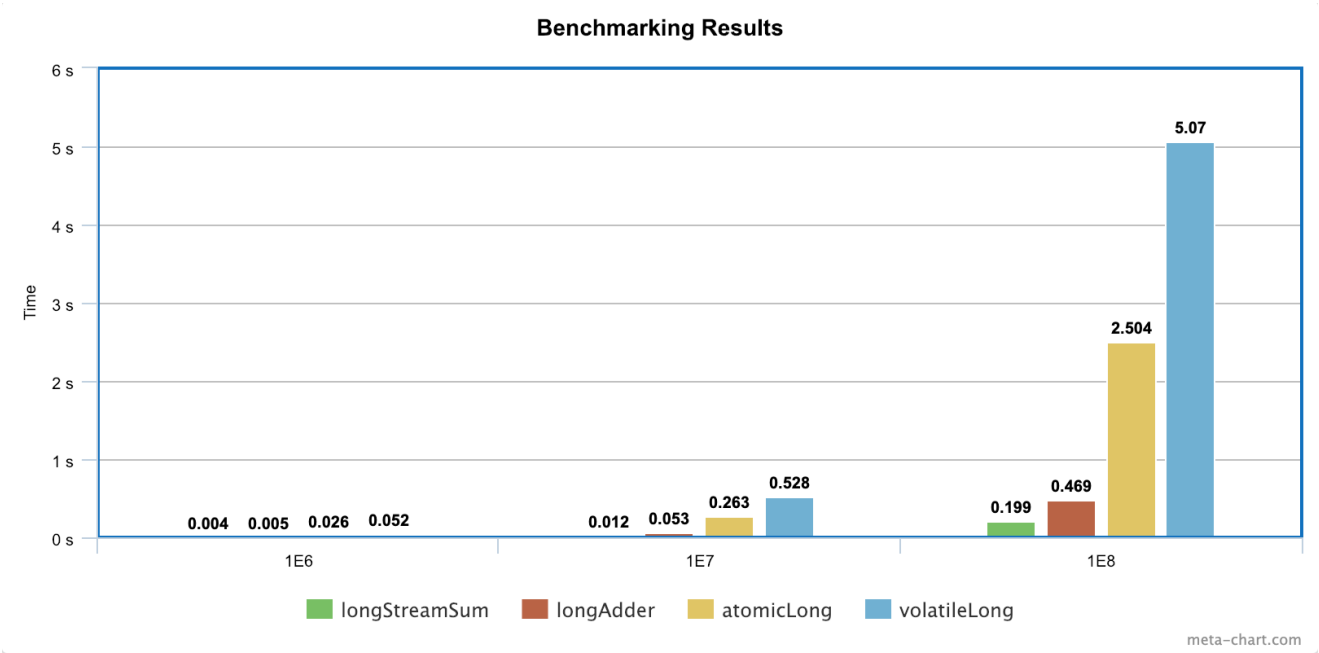
REMEMBER: The numbers below are just data. To gain reusable insights, you need to follow up on
why the numbers are the way they are. Use profilers (see -prof, -lprof), design factorial
experiments, perform baseline and negative tests that provide experimental control, make sure
the benchmarking environment is safe on JVM/OS/HW level, ask for reviews from the domain experts.
Do not assume the numbers tell you what you want them to tell.
Benchmark (listSize) Mode Cnt Score Error Units
MyBenchmark.atomicLong 1000000 avgt 5 0.026 ± 0.001 s/op
MyBenchmark.atomicLong 10000000 avgt 5 0.263 ± 0.003 s/op
MyBenchmark.atomicLong 100000000 avgt 5 2.504 ± 0.466 s/op
MyBenchmark.longAdder 1000000 avgt 5 0.005 ± 0.002 s/op
MyBenchmark.longAdder 10000000 avgt 5 0.053 ± 0.016 s/op
MyBenchmark.longAdder 100000000 avgt 5 0.469 ± 0.065 s/op

MyBenchmark.longStreamSum 1000000 avgt 5 0.004 ± 0.001 s/op
MyBenchmark.longStreamSum 10000000 avgt 5 0.012 ± 0.001 s/op
MyBenchmark.longStreamSum 100000000 avgt 5 0.199 ± 0.425 s/op
MyBenchmark.volatileLong 1000000 avgt 5 0.052 ± 0.019 s/op
MyBenchmark.volatileLong 10000000 avgt 5 0.528 ± 0.098 s/op
MyBenchmark.volatileLong 100000000 avgt 5 5.070 ± 0.302 s/op

view raw benchmark-result.txt hosted with ❤ by GitHub

The first paragraph is extremely important: do not take these numbers for granted. Follow that advice, always. I think you might better understand why I need to stress this after you read part three of this series. Until then, keep in mind that benchmarking is tricky, so always take those numbers with a grain of salt.

So, what do the results tell us? Let's put those numbers on a graph so we can visualize them:



It's quite clear that the fastest way was using a LongStream's `sum` method, followed by the LongAdder, then the AtomicLong and, finally, the `synchronized volatile long`. While the difference is always there, we see that LongAdder started just barely slower than LongStream's sum, but as soon as we started increasing the size of the list, the difference became more pronounced.

What did you expect? More importantly, why does this happen, and why is `volatile` + `synchronized` slower than AtomicLong?

I guess we have some homework to do now ;-)

The Blackhole

There's a section above that I did not explain. I did that on purpose ;-)

The reason why it's there, and specifically why it sometimes **needs** to be there, will be fully explained in Part 3. Stay tuned!

Author

Andre Brait

Andre Brait is a Software Engineer at Avenue Code. He enjoys learning about almost every field in computer science and engineering and is frequently involved in Free and Open Source Software development, especially those related to the GNU/Linux operating system.

Related Posts

How to Use k6 Load/Performance Testing for Web Pages

READ MORE

Java Microbenchmarks with JMH, Part 3

READ MORE

Java Microbenchmarks with JMH, Part 1

READ MORE