Java Microbenchmarks with JMH, Part 1

perm_identity Andre Brait
schedule 3/27/19 2:00 PM

You have an application running. It's doing pretty well, except for the fact that it's somewhat slow. You have identified the bottleneck, and now you're trying to come up with the ideal solution. There are a myriad of mechanisms you could try, but you don't want to deploy each one of your attempts to the development environment and then check the performance there. And running the application locally might not be the ideal solution either. So, what do you do?

You create a local benchmark for that small piece of code, of course! But there are problems with this approach—namely, the JVM and its optimizations might differ between your small local benchmark and the significantly larger application you have running in production, leading you to believe a solution is faster than it really will be when you deploy the application. How do we avoid this? Enter the Java Microbenchmark Harness, or JMH.

While JMH itself is quite simple to use, the devil is in the details. For that reason, this will be a three-part series. In this first part, we'll start by discussing why you'll want to use the JMH. We'll also be touching on how to get started with it and how to implement a simple benchmark. This part is designed to serve as an introduction for our next two articles.

- In Part 2, we'll go over implementation and how to configure JMH to suit your needs.
- In Part 3, we'll go over the most important aspect of benchmarking with the JMH: how **not** to do it.

The Issue with Naive Java Benchmarking

Benchmarking has always been a source of debate since benchmarks might not correspond to real-world usage scenarios. Because of this, it can be easy to get results that you *want* rather than results that are *accurate*. Yet, benchmarking is a necessary practice in a number of areas. Evaluating the performance of a piece of code can be crucial to achieving the best possible performance in a critical application.

Benchmarking appears to be a simple task. After all, it should consist of compiling your code, running it in a loop, and measuring how long that takes. The faster the better, right? That can be the case for statically compiled languages, such as C or C++, but for Java developers, there is an additional layer of complexity: the JVM.

Stressing that specific piece of code and extracting meaningful metrics can be difficult in the JVM world, because the JVM is an adaptive virtual machine and can perform optimizations that can produce misleading—or even completely useless—results. Unlike with statically compiled languages, the actual machine code that is executed is determined by the JVM while it is running, and it can change the code depending on what happens during the execution. This allows for the above-mentioned optimizations, but it also means Java developers need to be extra careful when writing benchmarks!

The need for specific precautions when writing benchmarks for Java culminated in the development of the JMH.

...But How Bad Can Manual Benchmarking Be?

That's a good question. According to an article published in Java Magazine, given the following code:

```
static double distance(double x1, double y1, double x2, double y2) {
double dx = x2 - x1;
double dy = y2 - y1;
return Math.sqrt((dx * dx) + (dy * dy));
}

static double constant(double x1, double y1, double x2, double y2) {
return 0.0d;
}

static void nothing() {
// this really does nothing
}
```

view raw bad-benchmarks.java hosted with ❤ by GitHub
...manual benchmarks can be this bad:

```
Running: distance
[ ~29975598 ops/ms ]

Running: constant
[ ~421092 ops/ms ]

```

```
Running: nothing
[ ~274938 ops/ms ]
```

view raw bad-benchmarks-result.txt hosted with ❤ by GitHub
As we can see, we have three computational operations going on here:

1. one that operates on **double precision floating-point numbers;**
2. one that just **returns a constant;**
3. one that does, literally, **nothing**.

And the results say *doing nothing* is **slower** than *returning a constant*, which in turn is **slower** than *operating on double precision floating-point numbers* - **including a square root** operation that's quite expensive! Taking the first and last numbers, we see that **doing nothing is more than 100x slower than doing something**! That is literally the opposite of what should be happening.

So What is JMH?

JMH is a Java harness library for writing benchmarks on the JVM, and it was developed as part of the OpenJDK project. JMH provides a very solid foundation for writing and running benchmarks whose results are not erroneous since they are not affected by unwanted virtual machine optimizations. **JMH itself does not prevent the pitfalls** that were briefly mentioned earlier, but it greatly helps in mitigating them. (The third part of this series is dedicated to explaining practices that help you avoid such pitfalls, which is why it's arguably the most important part.)

JMH is popular for writing *microbenchmarks*, that is, benchmarks that stress a very specific piece of code. JMH also excels at concurrent benchmarks. That being said, JMH is a general-purpose benchmarking harness, so it is useful for larger benchmarks, too.

Getting Started

The recommended - and easiest - way to get started with JMH is to generate a JMH Maven project using the JMH Maven Archetype. You can do this by using your favorite IDE, but I'll present an IDE-agnostic method, which is, of course, the good old command line interface.

Assuming you have Maven installed and in your `PATH`, run the following command:

```
mvn archetype:generate
-DinteractiveMode=false
-DarchetypeGroupId=org.openjdk.jmh
-DarchetypeArtifactId=jmh-java-benchmark-archetype
```

```
-DgroupId=com.avenuecode.snippet
-DartifactId=first-benchmark
-Dversion=1.0
```

view raw maven.sh hosted with ❤ by GitHub
This will generate a new folder called first-benchmark, inside which you'll find a `pom.xml` file and a Maven source directory structure. This project will already have declared in it the correct dependencies to compile and run the benchmarks (namely the JMH libraries and annotation processor, as well as some Maven plugins to make things easier).

Writing Your Own Benchmark

Now it's time for us to write the actual benchmarks. In the generated source structure, you'll find the `MyBenchmark` class, which will look something like the following:

```
package com.avenuecode.snippet;

import org.openjdk.jmh.annotations.Benchmark;

public class MyBenchmark {

@Benchmark
public void testMethod() {
// This is a demo/sample template for building your JMH benchmarks. Edit as needed.
// Put your benchmark code here.
}

}
```

view raw MyBenchmark-1.java hosted with ❤ by GitHub
The first thing we see here is the `@Benchmark` annotation. Every method annotated with `@Benchmark` is considered by JMH to be a benchmark to be executed. You can have any number of benchmark methods inside the benchmark class. Just be aware that the more benchmarks you add, the more time it'll take.

In order to implement our own benchmarks, all we have to do is put our code inside one of the benchmark methods. Easy, right?

Running Your Benchmark

The JMH Maven archetype includes all the necessary dependencies and plugins that make it easy to build and run our benchmarks. In order to build it, all we have to do is run:

```
mvn clean install
```

view raw maven-2.sh hosted with ❤ by GitHub
This will produce an executable jar for us in the targets directory. To run our benchmarks, just use the command below:

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

view raw maven-3.sh hosted with ❤ by GitHub
Proposed Benchmark

In order to give you an example of what a somewhat decent implementation of a benchmark looks like, let's use a simple yet interesting performance test.

There are a number of ways in which you can obtain the total sum of a list of integers while leveraging the power of multiple processing cores to speed up the process, even in scenarios where some level of synchronization is needed. But which one is the fastest? It's time to find out!

First, let's enumerate a few ways one could obtain this total sum while keeping things thread-safe and taking visibility into account as well:

1. Using a `volatile` field of type `long` to store the total value and synchronizing access to it in each sum;
2. Using an AtomicLong to achieve the same effect as above;
3. Using a LongAdder;
4. Using LongStream's `sum` method (our baseline, since there is no synchronization here).

Let's test this by writing a benchmark class like the one below:

```
package com.avuecode.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

...so, there's quite a number of new annotations here!

Part 2 will be all about these new annotations, how JMH works, and the results of our benchmark.

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 2

READ MORE