Java Microbenchmarks with JMH, Part 3

blog.avenuecode.com/java-microbenchmarks-with-jmh-part-3

perm_identity Andre Brait
schedule 4/10/19 2:00 PM

In Part 1 of this series, we covered the basics of creating a JMH project and getting it to run, proposed a small benchmark, and showed what the implementation would look like. In Part 2, we looked at how to configure JMH, how it works, and why certain parameters are necessary. In this last part, we'll focus on how **not** to write a benchmark for JMH and why some precautions are necessary. This may be the most important part of the three, as it'll give you insights into some quirks of the JVM.

Pitfalls

We know how to write benchmarks. We know why the JMH does certain things to prevent inaccuracies in the benchmarks. But there are still a plethora of pitfalls and quirks that can get us. One common pitfall is the set of **optimizations** the JVM can end up applying to our benchmarks that it would not apply in our original application, making our code appear to be faster than it really is. There are a number of publications addressing these issues, but I'll try to summarize some of the most frequent pitfalls that can make our benchmarks produce inaccurate results.

Loop Optimizations

You might be tempted to put your code inside a loop so that it will be executed more times per benchmark iteration and hence reduce the overhead of the benchmark method call. **Don't**. Only ever put loops in if they're part of the code you want to benchmark, that is, the loop is ***in*** the code and not ***around*** it.

The JVM is quite good at optimizing loops, so it may produce a different result if you wrap your code in a loop, and you may end up with a test result that does not correspond to the code running outside of a loop.

Dead Code Elimination

Another optimization the JVM commonly uses is getting rid of dead code. Dead code is - as the name implies - code that does nothing or is never used. The JVM can detect such pieces of code, and it'll happily **remove** them for you.

In the example below, the JVM will detect that the calculation `a + b`, which is assigned to sum, is never used, and it will remove the `a + b` operation from the method. Since `a` and `b` are also never used, they too can be removed. Because of this, you'll end up with a benchmark that is measuring how fast Java can do - literally - *nothing*.

```
@Benchmark
public void testMethod() {
int a = 1;
int b = 2;
int sum = a + b;
}
```

view raw sum-1.java hosted with ❤ by GitHub
To avoid dead code elimination, a couple of things can be done:

1. Return the result of the operation from the benchmark method.
2. Pass the value to a blackhole provided by JMH as a method argument.

We'll look at some examples of both approaches in the next subsections.

Return Value from Benchmark Method

The point here is to trick the JVM into thinking the code is being used by returning the result of the calculation from the benchmark method. This way, the JVM cannot just eliminate the addition, because the return value might be used by the caller. JMH will take care of tricking the JVM into thinking the value is going to be used.

The change itself is trivial, as shown below:

```
@Benchmark
public int testMethod() {
int a = 1;
int b = 2;
int sum = a + b;

return sum;
}
```

view raw sum-2.java hosted with ❤ by GitHub
However, it does not solve the problem of having dead code inside the method, say intermediate values or other calculations, that won't be returned. That code would still be eliminated. For such cases, JMH provides what it calls the Blackhole.

Passing Value to a Blackhole

In order to avoid returned combined values, JMH includes a Blackhole class that contains a consume method. This method can be used to trick the JVM into thinking code is not dead by passing the intermediate and non-returned values to it.

```
@Benchmark
public void testMethod(Blackhole blackhole) {
int a = 1;
int b = 2;
int sum = a + b;
blackhole.consume(sum);
}
```

view raw sum-3.java hosted with ❤ by GitHub
Notice two things here: the method now takes a Blackhole as an argument, and the blackhole object is used to consume the result of the addition operation. This will trick the JVM into thinking the sum variable is being used, and it'll avoid eliminating it as dead code.

If your benchmark method produces multiple results, you can pass each of these results to a black hole, meaning calling consume on the Blackhole instance for each value.

Constant Folding

The JVM is quite smart. So smart that even with the Blackhole or returning a value, our benchmark would still produce inaccurate results because both `a` and `b` are constants. A calculation that is based on constants will always produce the exact same result, regardless of how many times the calculation is performed. The JVM may detect that and replace the calculation with the result of the calculation. This process is an optimization known as constant folding.

Our benchmark could become:

```
@Benchmark
public int testMethod() {
int sum = 3;
return sum;
}
```

view raw sum-4.java hosted with ❤ by GitHub
It might go even further and replace the variable `sum` with `return 3` since it's able to detect the method always returns 3. It can even replace every external call to the method with a literal, since it already knows the method always returns 3.

Avoiding Constant Folding

To avoid constant folding, you must not hardcode constants into your benchmark methods. Instead, the input to your calculations should come from a state object. This makes it harder for the JVM to see that the calculations are based on constant values.

```
@State(Scope.Thread)
public static class MyState {
public int a = 1;
public int b = 2;
}

@Benchmark
public int testMethod(MyState state) {
int sum = state.a + state.b;
return sum;
}
```

view raw constant-1.java hosted with ❤ by GitHub
Remember, if your benchmark method calculates multiple values, you can pass them through a black hole instead of returning them, avoiding the dead code elimination optimization.

```
@Benchmark
public void testMethod(MyState state, Blackhole blackhole) {
int sum1 = state.a + state.b;
int sum2 = state.a + state.a + state.b + state.b;

blackhole.consume(sum1);
blackhole.consume(sum2);
```

```
}
```

view raw constant-2.java hosted with ❤ by GitHub

Whew! That's it, guys. Thanks for coming along! I hope this series provided you with what it takes to write those benchmarks, as well as some food for thought about how the JVM works.

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 2

READ MORE

Java Microbenchmarks with JMH, Part 1

READ MORE