Microbenchmarking

Introducing JMH

JHUG @Trasys Greece HQ

Kalfopoulos Georgios, Senior Solution Architect @Trasys Greece

15/03/2017

Micro.. what?

- Ever heard the term microservices?
- Containers?
- Docker?

- Absolutely no relation, I just had to mention Docker to make this a cool presentation ☺

Microbenchmarks

What are they?

- A benchmark designed to measure a “small” and very specific piece of code.

- “Small” could be one or two lines of code (e.g. is it better to use an Iterator on an ArrayList or is a standard indexed for loop in combination with .get(i) better)
- “Small” could also be a full algorithm, a component, a service, half your program…

Microbenchmarks

Any caveats?

- "It's a dangerous business, Frodo, going out your door. You step onto the road, and if you don't keep your feet, there's no knowing where you might be swept off to."
  — **J.R.R. Tolkien**, **The Lord of the Rings**

- Microbenchmarks are hard to write correctly
- A wrong microbenchmark may lead you to wrong decisions about what needs optimizing and what not

Microbenchmark example

Measuring a for loop

- for(int i = 0; i < 1000000; ++i){ }
- Hotspot will optimize it away
- int sum = 0;
  for(int i = 0; i < 1000000; ++i){ sum++; }
- Hotspot will figure out sum is not used optimize it away, and then optimize away the loop
- int sum = 0;
  for(int i = 0; i < 1000000; ++i){ sum++; }
  System.out.println(sum);
- Hotspot will figure out the loop ends up in a constant value, use that and optimize all else away.

Microbenchmarking

Hmm, difficult. VERY difficult. (*Sorting Hat*, **Harry Potter and the Sorcerer's Stone**)

- Dead Code Elimination
  - If the JVM detects that the result of some computation is never used, the JVM may consider this computation *dead code* and eliminate it
- Constant Folding
  - A calculation which is based on constants will result in the exact same result, regardless of how many times the calculation is performed
- Pre/post code compilation behavior (-XX:CompileThreshold)
- Loop unrolling
- On-Stack Replacement
- Hardware, OS, JVM optimizations
- Other applications competing for resources

Introducing JMH

Java Microbenchmark Harness

- For a correct Java microbenchmark we should prevent the optimizations the JVM and hardware may apply during microbenchmark execution which could not have been applied in a real production system.

- JMH to the rescue. JMH is a toolkit that helps you implement Java microbenchmarks correctly. JMH is developed by the same people who implement the JVM.

JMH

What exactly does it do

- Runs your code:
  - In multiple forks (10 by default), using a new JVM instance for each fork
  - Runs warmup iterations (20 by default) which are measured but not included in averages
  - Runs "regular" iterations (20 by default) which are averaged
  - On each iteration it simply runs your code as many times as it can for a minimum of 1 second (and a default maximum of 10 minutes)
  - Reports in the end how many runs it was able to perform measured as ops per second (one op = one run of your code)

Microbenchmark

An Example

- Let's try to sum an ArrayList of Longs with various ways and see what method measures best depending on the list overall size.
- We would like to try variations with plain indexed for loop and get(), for…each, iterators, plus a few variations with streams
- We would also like to try the tests with a variety of sizes in the list (10, 10.000, 10.000.000)

Microbenchmarking Arraylists

First we need a project

```
mvn archetype:generate
        -DinteractiveMode=false
        -DarchetypeGroupId=org.openjdk.jmh
        -DarchetypeArtifactId=jmh-java-benchmark-archetype
        -DgroupId=gr.trasys
        -DartifactId=first-benchmark
        -Dversion=1.0
```

Which produces a simple project with one java file ready to fill in your benchmark code and correctly configured pom.xml that will produce a "fat" jar ready to run in the end.

Microbenchmarking Arraylists

Then we need state to work with

```
@State(Scope.Thread)
public static class BenchmarkState {
  @Param({ "10", "10000", "10000000" })
  public int listSize;

  public ArrayList<Long> list = new ArrayList<>();

  @Setup(Level.Trial)
  public void toSetup() {
    for (long i = 1; i <= listSize; i++) {
      list.add(i);
    }
  }
}
```

JMH Annotations

Part one

- @State
  - Initializing and setting variables needed for your test, but that are not part of the test itself (defeats Constant Folding as well)
  - Scope can be Thread, Group, or Benchmark indicating how many instances are created and shared between threads during the benchmark run
- @Setup
  - Marks initialization methods
  - Level indicates when to call the method
    - Trial = once for the full Benchmark run (full means one "fork" including all warmups and iterations)
    - Iteration = once every iteration
    - Invocation = every time the test method is called
- @Param
  - Used on primitives/Strings (pass String values that will be autoconverted)
  - The harness will automatically run the benchmark once for each values
  - If more than one @Param are defined, the benchmark will run for each combination

Microbenchmarking Arraylists

Finally, we need benchmarks

```
@Benchmark
@BenchmarkMode(Mode.Throughput)
public long sumUsingGet(BenchmarkState state, Blackhole blackhole) {

long sum = 0;
for (int i = 0; i < state.list.size(); i++) {
  sum += state.list.get(i).longValue();
}

blackhole.consume(sum);

return sum;
}
```

JMH Annotations and BlackHole

Part two

- @Benchmark
  - Think of this as similar to @Test from Junit and you are in the right track
- @BenchmarkMode
  - Throughput, Average Time, Sample Time, Single Shot Time, All
- Blackhole (avoiding Dead Code Elimination)
  - If your benchmark produces just one value you can simply return it
  - If you produce more than one values, use Blackhole.consume()

Microbenchmarking Arraylists

A few alternatives

- ```
  for (Long number : state.list) {
      sum += number.longValue();
  }
  ```
- ```
  Iterator<Long> iter = state.list.iterator();
  while (iter.hasNext()) {
      sum += iter.next().longValue();
  }
  ```
- `state.list.stream().mapToLong(Long::longValue).sum();`
- `state.list.parallelStream().mapToLong(Long::longValue).sum();`
- `state.list.stream().reduce(0L, (a, b) -> (a.longValue() + b.longValue()));`
- `state.list.parallelStream().reduce(0L, (a, b) -> (a.longValue() + b.longValue()));`

Microbenchmarking Arraylists

Ready, Set, Go!

- mvn clean package
- java –jar .\target\benchmarks.jar

```
# JMH 1.17.4 (released 27 days ago)
# VM version: JDK 1.8.0_121, VM 25.121-b13
# VM invoker: C:\Program Files\Java\jre1.8.0_121\bin\java.exe
# VM options: <none>
# Warmup: 20 iterations, 1 s each
# Measurement: 20 iterations, 1 s each
# Timeout: 10 min per iteration
# Threads: 1 thread, will synchronize iterations
# Benchmark mode: Throughput, ops/time
# Benchmark: gr.trasys.MyBenchmark2.sumUsingGet
# Parameters: (listSize = 10)

# Run progress: 57,14% complete, ETA 01:49:48
# Fork: 1 of 10
# Warmup Iteration   1: 56399628,007 ops/s
# Warmup Iteration   2: 57475680,988 ops/s
# Warmup Iteration   3: 61688471,269 ops/s
```

Microbenchmarking Arraylists

Hours later

Benchmark	(listSize)	(variation)	Mode	Cnt	Score	Error	Units
MyBenchmark2.sumUsingForEach	10	N/A	thrpt	200	57654813,898 ±	437705,820	ops/s
MyBenchmark2.sumUsingForEach	10000	N/A	thrpt	200	111544,886 ±	204,746	ops/s
MyBenchmark2.sumUsingForEach	10000000	N/A	thrpt	200	42,253 ±	0,106	ops/s
MyBenchmark2.sumUsingGet	10	N/A	thrpt	200	61156168,395 ±	97415,234	ops/s
MyBenchmark2.sumUsingGet	10000	N/A	thrpt	200	112517,387 ±	141,998	ops/s
MyBenchmark2.sumUsingGet	10000000	N/A	thrpt	200	42,116 ±	0,100	ops/s
MyBenchmark2.sumUsingIterator	10	N/A	thrpt	200	58339346,730 ±	96620,503	ops/s
MyBenchmark2.sumUsingIterator	10000	N/A	thrpt	200	111239,521 ±	188,824	ops/s
MyBenchmark2.sumUsingIterator	10000000	N/A	thrpt	200	42,450 ±	0,091	ops/s
MyBenchmark2.sumUsingParallelStreamMap	10	N/A	thrpt	200	163853,288 ±	3073,882	ops/s
MyBenchmark2.sumUsingParallelStreamMap	10000	N/A	thrpt	200	46309,720 ±	3656,783	ops/s
MyBenchmark2.sumUsingParallelStreamMap	10000000	N/A	thrpt	200	59,084 ±	1,639	ops/s
MyBenchmark2.sumUsingParallelStreamReduce	10	N/A	thrpt	200	157311,180 ±	1702,657	ops/s
MyBenchmark2.sumUsingParallelStreamReduce	10000	N/A	thrpt	200	25141,630 ±	552,714	ops/s
MyBenchmark2.sumUsingParallelStreamReduce	10000000	N/A	thrpt	200	23,902 ±	0,127	ops/s
MyBenchmark2.sumUsingStreamMap	10	N/A	thrpt	200	19796604,699 ±	58854,520	ops/s
MyBenchmark2.sumUsingStreamMap	10000	N/A	thrpt	200	95272,491 ±	348,331	ops/s
MyBenchmark2.sumUsingStreamMap	10000000	N/A	thrpt	200	43,021 ±	0,069	ops/s
MyBenchmark2.sumUsingStreamReduce	10	N/A	thrpt	200	9608975,231 ±	47973,749	ops/s
MyBenchmark2.sumUsingStreamReduce	10000	N/A	thrpt	200	12578,832 ±	36,326	ops/s
MyBenchmark2.sumUsingStreamReduce	10000000	N/A	thrpt	200	14,644 ±	0,061	ops/s

Microbenchmarking

Resources and follow up

- OpenJDK JMH homepage (make sure to look at samples)
- Stack Overflow: How do I write a correct microbenchmark?

- In-depth discussion and presentation of JMH features