Java Microbenchmarking mit JMH

Alexander Gehrke

January 24, 2018

Microbenchmarks?

- Messung der Performance eines kleinen Codeteils (z.B. einer Funktion)
- Kein Ersatz für Benchmarks des gesamten Programms

Naiver Ansatz

Messe für zufälliges Int nötige Zeit

```
final Random random = new Random();
final long start = System.nanoTime();
random.nextInt();
final long end = System.nanoTime();
System.out.println(end - start);
```

Naiver Ansatz

Messe für ~~zufälliges Int~~ nötige Zeit

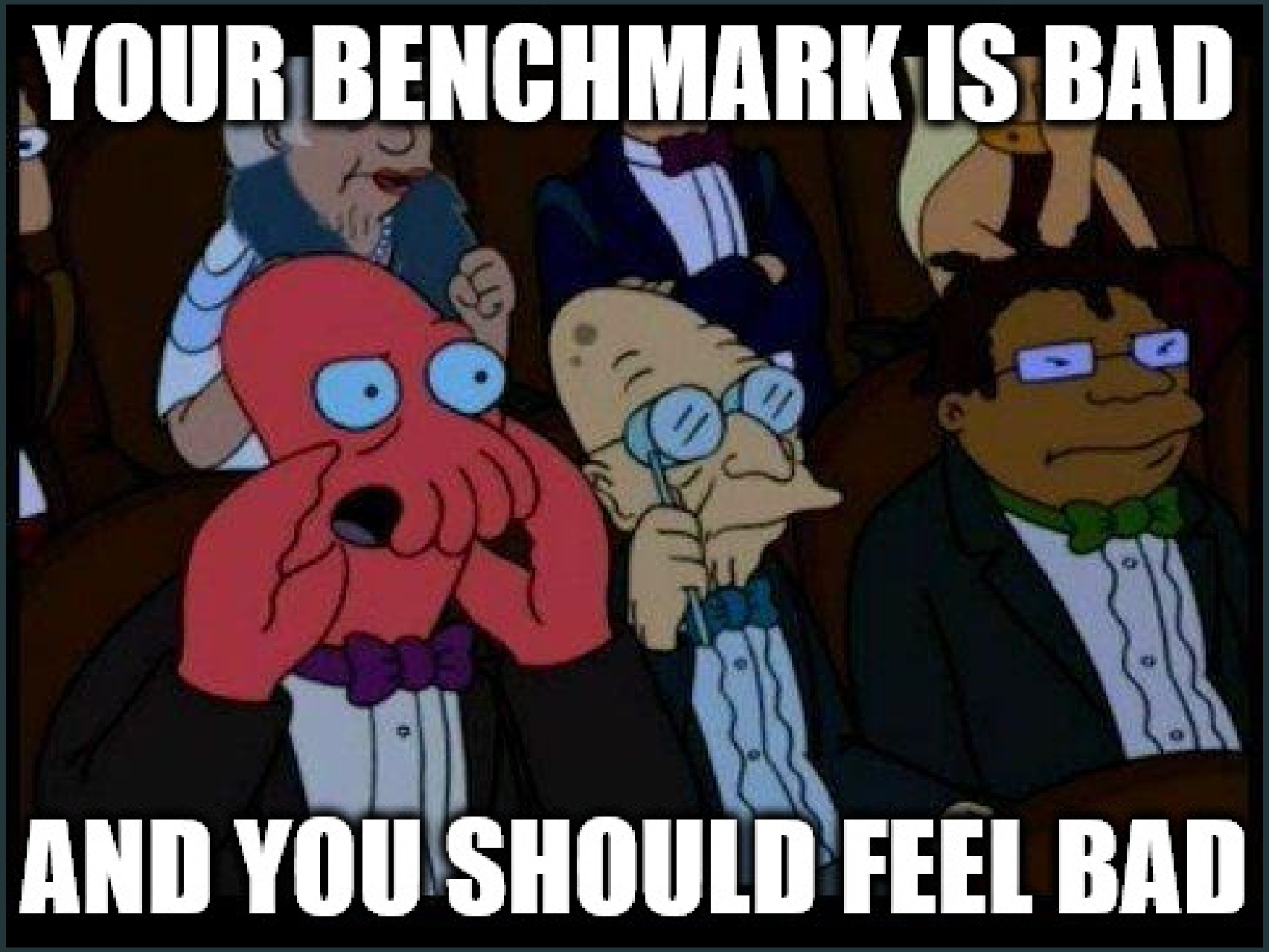
```
final Random random = new Random();
final long start = System.nanoTime();
random.nextInt();
final long end = System.nanoTime();
System.out.println(end - start);
```

Naiver Ansatz

Messe ziemlich zufällige Zeit

```
final Random random = new Random();
final long start = System.nanoTime();
random.nextInt();
final long end = System.nanoTime();
System.out.println(end - start);
```

- JVM benutzt JIT-Compilation, aber standardmäßig erst nach 1000 Aufrufen
- `Random.nextInt()` dauert nur sehr kurz $\Longrightarrow$ Betriebssystemaktivität (z.B. Interrupts, Scheduling) fällt ins Gewicht
- Statistische Schwankungen


YOUR BENCHMARK IS BAD
AND YOU SHOULD FEEL BAD

Etwas besserer Ansatz

```
static void doMeasure() {
  final long start = System.nanoTime();
  for(int i=0; i<1_000_000; i++) {
    random.nextInt();
  }
  final long end = System.nanoTime();
  System.out.println(end - start);
}
```

- Methode mehrfach aufrufen - erster Durchlauf für JIT-Warmup

Etwas besserer Ansatz

Weniger schlecht als vorheriges Ergebnis, aber immer noch ungenau:

- Keine Trennung vom Testframework, JIT optimiert uns evtl. die Schleife.
- Vergleich mehrerer Varianten in einem Benchmark nicht möglich wegen fehlender Isolation.

JMH to the rescue!

- Java Microbenchmarking Harness aus dem OpenJDK-Projekt
- Anwendung ähnlich wie Unittests: Code in Methode packen, Annotation dran $\rightarrow$ Benchmark!
- Prinzipiell für jede JVM-Sprache, Vorlagen für Java, Scala, Groovy und Kotlin

Einbindung in bestehendes Projekt

Maven Dependencies:

```
org.openjdk.jmh : jmh-core : 1.19

org.openjdk.jmh : jmh-generator-annprocess : 1.19
```

Empfohlen: separates Maven-Projekt

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

`archetypeArtifactId` in der vierten Zeile an gewünschte Programmiersprache anpassen

Benchmark in JMH

```
@Warmup(iterations=3, time=3, timeUnit=TimeUnit.SECONDS)
@Measurement(iterations=5, time=10,
↪   timeUnit=TimeUnit.SECONDS)
@Fork(1)
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.NANOSECONDS)
@State(Scope.Benchmark)
@Threads(1)
class MyBenchmark {
  private Random random = new Random();

  @Benchmark
  public void runRandomInt() {
    random.nextInt();
  }
}
```

Ausführen

Bei Verwendung des Maven-Archetype:

```
mvn clean install
java -jar target/benchmarks.jar
```

Dann etwas (je nach Konfiguration auch lange) warten:

```
# Run progress: 0.00% complete, ETA 00:00:59
# Fork: 1 of 1
# Warmup Iteration   1: 12.098 ns/op
# Warmup Iteration   2: 12.079 ns/op
# Warmup Iteration   3: 12.067 ns/op
Iteration   1: 12.122 ns/op
Iteration   2: 12.092 ns/op
...
```

Ergebnis

```
Result "org.sample.MyBenchmark.testMethod":
12.096 ±(99.9%) 0.074 ns/op [Average]
(min, avg, max) = (12.071, 12.096, 12.122), stdev = 0.019
CI (99.9%): [12.022, 12.170] (assumes normal distribution)

# Run complete. Total time: 00:00:59

Benchmark                  Mode  Cnt   Score   Error  Units
MyBenchmark.testMethod     avgt    5  12.096 ± 0.074  ns/op
```

Parametrisierte Benchmarks

```java
@Param({"1", "31", "65", "101", "103"})
public int arg;

@Param({"0", "1", "2", "4", "8", "16", "32"})
public int certainty;

@Benchmark
public boolean bench() {
    return BigInteger.valueOf(arg)
      .isProbablePrime(certainty);
}
```

Parametrisierte Benchmarks

```
Benchmark                (arg)  (certainty)      Score      Error
MyBenchmark.bench            1            0      4.419 ±    0.129
MyBenchmark.bench            1            1      7.754 ±    0.425
MyBenchmark.bench            1            2      8.440 ±    1.462
MyBenchmark.bench            1            4      7.720 ±    0.213
...
MyBenchmark.bench           31            0      6.236 ±    0.351
MyBenchmark.bench           31            1    660.332 ±   11.083
MyBenchmark.bench           31            2    660.559 ±   34.127
MyBenchmark.bench           31            4   1269.245 ±   67.708
MyBenchmark.bench           31            8   2439.929 ±   52.919
...
```

(Spalten `Mode`, `Cnt` und `Units` fehlen aus Platzgründen)

Mehr Features

- 4 verschiedene Benchmark-Modes (Throughput, Avg. Time, Sampled Time, Single Shot Time)
- State-Scoping (für welche Benchmark-Durchläufe gilt ein Parameter)
- Fixtures (`@Setup, @Teardown`)
- …

Fallstricke

- Unerwartete Optimierungen der JVM
  - „Black Holes“ benutzen, um Wegoptimierung von unbenutzten Ergebnissen zu verhindern (bei nur einem Wert: `return`)
  - auch konstante Parameter nicht final machen
  - der IDE nicht glauben, wenn sie sagt „Die Variable geht auch lokal.“
- zusätzliche Probleme bei multi-treaded Benchmarks (z.B. „false sharing“)

Further Reading / Sources

- JMH Homepage inkl. vielen Beispielen
- JAXenter: Aus der Java-Trickkiste: Microbenchmarking
- baeldung.com: Microbenchmarking with Java