8 Neues und Änderungen in Java 12

Bei Oracle folgt man mittlerweile der Strategie, Java in kleinen, aber feinen Iterationen um nützliche Funktionalität zu ergänzen und auch bei der Syntax zu modernisieren. Zudem soll ein sechsmonatiger Releasezyklus eingehalten werden. Demnach erfolgte die Veröffentlichung von Java 12 im März 2019. Für uns als Entwickler finden sich folgende interessante Erweiterungen in JDK 12:

- Switch Expressions (Preview) (`https://openjdk.java.net/jeps/325`)
- Microbenchmark Suite (`https://openjdk.java.net/jeps/230`)
- Einige kleinere Neuerungen und Anpassungen in den APIs

Auf die Switch Expressions gehe ich im Folgenden nicht näher ein, da eine aktualisierte und leicht modifizierte Variante wiederum als Preview Einzug in Java 13 gefunden hat und final in Java 14 enthalten ist. Diese Neuerung wird in dem dazu korrespondierenden Kapitel 9 beschrieben.

8.1 Microbenchmark Suite

Mitunter sind einige Teile der eigenen Software nicht so performant, wie man es sich erhofft hat bzw. wie es vom Nutzer erwartet wird. Zur Optimierung der Performance gibt es verschiedene Hilfsmittel. Generell sollte man zunächst gründlich messen und nur mit Bedacht und vor allem nicht direkt optimieren. Das gilt unter anderem deshalb, weil zum einen die JVM bereits diverse Optimierungen eingebaut hat und zum anderen man rein auf Basis von Vermutungen häufig falsch liegt und so nur Programmteile schlechter les- und wartbar macht, jedoch die Gesamtperformance kaum oder überhaupt nicht verbessert.

Als Abhilfe gibt es oftmals verschiedene Möglichkeiten. Manchmal reicht ein anderer Algorithmus und manchmal kann Caching helfen. Immer jedoch empfiehlt es sich, die für die Performance kritischen Stellen zu ermitteln und auch nur dort zu optimieren. Diese Aussage gilt für die Architektur und das Design bis hin zur Low-Level-Implementierung. Details zum Thema Optimierung finden Sie in meinem Buch »Der Weg zum Java-Profi« [4].

Wie schon erwähnt, sollte man die kritischen Stellen keinesfalls nur aufgrund von Vermutungen identifizieren, sondern basierend auf Messungen. Dazu kann man beispielsweise einfache Start-Stopp-Messungen einsetzen, empfehlenswerter sind ausgeklügeltere Verfahren mit mehreren Durchläufen, um temporäre Störeffekte, verursacht durch höhere Systemlast oder von Garbage Collections, zu vermeiden. Dabei hilft die mit Java 12 in das JDK aufgenommene Microbenchmark Suite, die auf dem Toolkit Java Microbenchmark Harness (JMH) basiert.

8.1.1 Eigene Microbenchmarks und Varianten davon

Nachfolgend schauen wir uns die Optimierung auf der Ebene von einzelnen bzw. wenigen Anweisungen an. Dazu nutzt man sogenannte Microbenchmarks. Bedenken Sie bitte aber vorab, dass derartige Optimierungen nur dann sinnvoll sind, wenn auf den höheren Ebenen wie Architektur, Design und Algorithmus keine Verbesserungen mehr erreicht werden können oder die zu optimierenden Anweisungen extrem häufig ausgeführt werden und somit die Gesamtperformance maßgeblich beeinflussen.

Nehmen wir an, wir wollten Aussagen über die Performance für einige Varianten einer Implementierung treffen. Grundsätzlich ist das nicht so einfach, wie man zunächst annehmen könnte, da die Messungen unter möglichst gleichen Bedingungen (CPU-Last, Speicherverbrauch usw.) geschehen sollten, damit man vergleichbare Resultate erhält. Aber auch Optimierungsmechanismen in der JVM wie Loop-Unrolling, Hotspot-Kompilierung usw. führen potenziell dazu, dass sich Messwerte signifikant voneinander unterscheiden.

Einfache Start-Stopp-Messungen

Im einfachsten Fall nutzt man eine Art Stoppuhr, indem man den zu messenden Programmteil mit Aufrufen von `System.currentTimeMillis()` wie folgt umklammert und die Differenz zwischen den Werten ermittelt:

```
// ACHTUNG: keine gute Variante
final long startTimeMs = System.currentTimeMillis();

someCodeToMeasure();

final long stopTimeMs = System.currentTimeMillis();
final long duration = stopTimeMs - startTimeMs;
```

Das scheint vernünftig. Jedoch nutzt `System.currentTimeMillis()` – zumindest unter Windows – ungenaue, zu grobgranulare Zeitgeber des Betriebssystems. Verlässlichere Werte erhält man mithilfe der folgenden Methode:

```
System.nanoTime()
```

Wiederholte Start-Stopp-Messungen

Unabhängig vom Zeitgeber liefern einfache Start-Stopp-Messungen eher unzuverlässige Ergebnisse, da die Messungen durch unterschiedlichste Effekte verfälscht werden können. Deswegen ist es oftmals sinnvoller, eine gewisse Anzahl an Iterationen zu nutzen und dann den Durchschnittswert als Referenzmesswert zu errechnen:

```
// bessere Variante
final long startTimeNs = java.lang.System.nanoTime();

for (int i = 0; i < MAX_ITERATIONS; i++)
{
   someCodeToMeasure();
}

final long stopTimeNs = java.lang.System.nanoTime();

final long avgDurationNs = (stopTimeNs - startTimeNs) / MAX_ITERATIONS;
```

Diese Variante mit mehreren Durchläufen und Durchschnittsbildung ist weniger anfällig für Systemlastschwankungen oder sonstige Störeinflüsse. Zudem ist es recht einfach möglich, auch Minimal- und Maximaldauer oder die Standardabweichung zu ermitteln. Bei diesen erweiterten Anforderungen lagert man diese Funktionalität aber sinnvollerweise in eine oder mehrere separate Klassen aus.

Wiederholte Start-Stopp-Messungen mit Warm-up

Neben kleineren externen Störungen beobachtet man immer mal wieder Einschwing-Effekte: Erst nach einer gewissen Anzahl an Durchläufen zeigt eine Funktionalität ihre optimale Laufzeit: Das kann etwa dadurch verursacht werden, dass Caches zunächst gefüllt werden oder aber der Hotspot-Optimierer verschiedene Codestellen in Maschinensprache übersetzt. Für diese Fälle empfiehlt sich dann ein Warm-up vor der eigentlichen Messung.

Nachfolgend zeige ich exemplarisch ein der eigentlichen Performance-Messung vorangestelltes Warm-up in Form einiger Schleifendurchläufe mit Aufrufen an die später zu messende Funktionalität:

```
// noch bessere Variante der Messung durch Warm-up
for (int i = 0; i < MAX_WARMUP_ITERATIONS; i++)
{
   someCodeToMeasure();
}

// eigentliche Messung nach Warm-up
final long startTimeNs = java.lang.System.nanoTime();

for (int i = 0; i < MAX_ITERATIONS; i++)
{
   someCodeToMeasure();
}
final long stopTimeNs = java.lang.System.nanoTime();

final long avgDurationNs = (stopTimeNs - startTimeNs) / MAX_ITERATIONS;
```

Man erkennt gut, dass die Implementierung der Messung immer unhandlicher wird. Das ist oftmals ein Zeichen dafür, dass man nach einem unterstützenden Framework Ausschau halten sollte. Glücklicherweise gibt es das JMH, das wir nun genauer kennenlernen.

Tipp: Hotspot-Optimierer

Wenn man ein Java-Programm startet, so liegt dieses zunächst in Bytecode vor. Dieser wird von der JVM interpretiert. Obwohl dies in den letzten Jahren immer performanter geworden ist, reicht die Ausführungsgeschwindigkeit interpretierten Bytecodes natürlich nicht ganz an die von Maschinensprache heran. In die JVM ist der Hotspot-Optimierer inklusive eines JIT-Compilers (JIT = Just In Time) integriert. Beide sorgen dafür, dass besonders häufig durchlaufene Teile des Bytecodes erkannt und on the fly in Maschinensprache transferiert werden.

Aufgrund dieser Besonderheit betrachtet man bei naiven Messungen der Performance eventuell nur die interpretierte Ausführung, gegebenenfalls aber auch eine Mischform, wenn bereits ein Teil des Bytecodes in Maschinensprache übersetzt wurde.

Mithilfe des Aufrufparameters `-XX:+PrintCompilation` kann man die Kompilierung protokollieren lassen und `-XX:CompileThreshold` erlaubt es, die Anzahl der benötigten Ausführungen eines Programmstücks bis zu dessen Kompilierung festzulegen. Standardmäßig benötigt es in der seit Java 11 nur noch vorhandenen Servervariante 10.000 Abarbeitungen – die früher verfügbare Client-JVM hat bereits nach 1.500 Abarbeitungen des Bytescodes kompiliert.

8.1.2 Microbenchmarks mit JMH

Seit Java 12 ist das Open-Source-Framework Java Microbenchmark Harness (JMH) ins JDK aufgenommen worden und adressiert die oben angedeuteten sowie weitere Schwierigkeiten bei Performance-Messungen.

Einstieg in JMH

JMH arbeitet mit Annotations und integriert basierend darauf verschiedene Messungen. Hierzu bietet sich die Nutzung eines Maven-Archetypes an. Folgender Aufruf erzeugt ein neues Verzeichnis mit einem vollständig initialisierten JMH-Benchmark:

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

Schauen Sie doch kurz in Anhang C, um etwas mit Maven vertraut zu werden.

Dadurch entstehen folgende Verzeichnisse und Dateien:

```
.
`-- jmh-test
    |-- pom.xml
    `-- src
        `-- main
            `-- java
                `-- org
                    `-- sample
                        `-- MyBenchmark.java
```

Als Grundgerüst wird eine Klasse `MyBenchmark` erzeugt, weshalb sich dort auch auch ein englischer Kommentar findet. In der generierten Klasse können wir nun die zu beobachtenden Funktionalitäten aufrufen. Ähnlich zu JUnit lassen sich die gewünschten Einstellungen zu den Messungen per Annotations vorgeben. Die wichtigste ist `@Benchmark`, die eine zu untersuchende Methode kennzeichnet:

```
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.
    }
}
```

Um dann den Benchmark auszuführen, muss man in das durch den obigen Aufruf erzeugte Verzeichnis wechseln und dort – wie von Maven gewohnt – kompilieren und paketieren:

```
mvn package
```

Dadurch entsteht ein entsprechendes Artefakt `benchmarks.jar` im Verzeichnis `target`. Weil dort insbesondere alle benötigten Klassen und Bibliotheken bereits enthalten sind, kann man das JAR ohne weitere Angaben folgendermaßen starten:

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

Da wir bislang noch keine sinnvolle Funktionalität, sondern nur eine leere Methode messen würden, heben wir uns den Start für das nächste Beispiel auf.

Der erste Benchmark in JMH

Wir haben gerade gesehen, wie man eine Performance-Suite mit JMH erzeugt. Allerdings weiß das Framework ja noch nicht, was gemessen werden soll. Deshalb wird auch nur eine Klasse `MyBenchmark` als Grundgerüst erzeugt. Dort können wir die zu beobachtenden Funktionalitäten aufrufen und die gewünschten Messeinstellungen per Annotations vorgeben. Generell empfiehlt es sich aber, einen sprechenderen Namen für

die Benchmark-Klasse zu nutzen, nachfolgend etwa `FirstIntegerBenchmark`, der die Performance der Methoden `toHexString()` und `toBinaryString()` misst.

Insgesamt wird das Schreiben von Microbenchmarks ziemlich einfach: Praktischerweise kann man zwar eine recht feingranulare Steuerung vornehmen, aber dem Gedanken *Convention over Configuration* folgend ist es für erste Messungen oftmals ausreichend, wenn man lediglich die Annotation `@Benchmark` wie folgt nutzt – die Zahlen sind mithilfe des Unterstrichs an den Positionen der Tausenderpunkte etwas besser lesbar:

```
public class FirstIntegerBenchmark
{
    @Benchmark
    public String numberAsHexString()
    {
        final int number = 1_234_567;
        return Integer.toHexString(number);
    }

    @Benchmark
    public String numberAsBinaryString()
    {
        final int number = 123_456_789;
        return Integer.toBinaryString(number);
    }
}
```

Nachdem wir beide Klassen kompiliert und paketiert haben, starten wir das ausführbare JAR. Es ergeben sich in etwa folgende gekürzten Konsolenausgaben:

```
# JMH version: 1.23
# VM version: JDK 14, OpenJDK 64-Bit Server VM, 14+36-1461
# VM invoker: /Library/Java/JavaVirtualMachines/jdk-14.jdk/Contents/Home/bin/
      java
# VM options: <none>
# Warmup: 5 iterations, 10 s each
# Measurement: 5 iterations, 10 s each
# Timeout: 10 min per iteration
# Threads: 1 thread, will synchronize iterations
# Benchmark mode: Throughput, ops/time
# Benchmark: org.sample.FirstIntegerBenchmark.numberAsBinaryString

# Run progress: 0.00% complete, ETA 00:16:40
# Fork: 1 of 5
# Warmup Iteration   1: 56326650.457 ops/s
# Warmup Iteration   2: 59780600.908 ops/s
# Warmup Iteration   3: 64051264.973 ops/s
# Warmup Iteration   4: 64008812.168 ops/s
# Warmup Iteration   5: 64186426.089 ops/s
Iteration   1: 61304141.790 ops/s
Iteration   2: 61953075.846 ops/s
Iteration   3: 63036185.662 ops/s
Iteration   4: 64322247.337 ops/s
Iteration   5: 64872604.082 ops/s
...
```

Nach längerer Ausführung erhält man dann ungefähr folgende Ergebnisse:

```
Result "org.sample.FirstIntegerBenchmark.numberAsBinaryString":
  64224040.592 ±(99.9%) 978073.744 ops/s [Average]
  (min, avg, max) = (61298595.526, 64224040.592, 65767851.332), stdev =
       1305700.466
  CI (99.9%): [63245966.848, 65202114.336] (assumes normal distribution)

Result "org.sample.FirstIntegerBenchmark.numberAsHexString":
  145802842.871 ±(99.9%) 1765144.419 ops/s [Average]
  (min, avg, max) = (140986090.679, 145802842.871, 149109246.667), stdev =
       2356417.299
  CI (99.9%): [144037698.452, 147567987.290] (assumes normal distribution)
```

Möchte man den Benchmark noch ein wenig feintunen, beispielsweise die Anzahl an Iterationen oder das Warm-up und die Anzahl an Messdurchgängen konfigurieren, so könnte man dies folgendermaßen mithilfe der Annotations `@Warmup` sowie `@Measurement` tun:

```
import org.openjdk.jmh.annotations.Measurement;
import org.openjdk.jmh.annotations.Warmup;
import org.openjdk.jmh.annotations.Fork;

@Measurement(iterations = 3, time = 1000, timeUnit = TimeUnit.MILLISECONDS)
@Warmup(iterations = 7, time = 1000, timeUnit = TimeUnit.MICROSECONDS)
@Fork(4)
public class MyBenchmark
{
     ....
}
```

Wendet man diese Werte auf den ersten Benchmark an, dann ändert sich die Ausgabe wie folgt:

```
# Benchmark: org.sample.FirstIntegerBenchmark.numberAsBinaryString

# Run progress: 0.00% complete, ETA 00:33:44
# Fork: 1 of 4
# Warmup Iteration   1: 2612321.666 ops/s
# Warmup Iteration   2: 6316193.097 ops/s
# Warmup Iteration   3: 7323288.665 ops/s
# Warmup Iteration   4: 4430635.983 ops/s
# Warmup Iteration   5: 6554600.103 ops/s
# Warmup Iteration   6: 4660414.309 ops/s
# Warmup Iteration   7: 5267248.079 ops/s
Iteration   1: 32894213.089 ops/s
Iteration   2: 34024432.225 ops/s
Iteration   3: 46131122.469 ops/s
```

Man erkennt sehr schön den zuvor beschriebenen Einfluss der Annotations: Es gibt erwartungsgemäß vier Durchläufe mit jeweils sieben Warm-ups und drei Iterationen.

Der zweite Benchmark in JMH

Um noch ein wenig vertrauter mit JMH zu werden, erstellen wir eine Messung für die arithmetischen Operationen * und / sowie für das Potenzieren von `int`-Werten:

```
import org.openjdk.jmh.annotations.Benchmark;
import org.openjdk.jmh.annotations.BenchmarkMode;
import org.openjdk.jmh.annotations.Mode;
import org.openjdk.jmh.annotations.OutputTimeUnit;
import org.openjdk.jmh.annotations.State;
import org.openjdk.jmh.annotations.Scope;

import java.util.concurrent.TimeUnit;

@State(Scope.Benchmark)
public class SimpleMathBenchmark
{
    int x = 7271;
    int y = 1234;

    @Benchmark
    @BenchmarkMode({Mode.Throughput, Mode.AverageTime})
    @OutputTimeUnit(TimeUnit.NANOSECONDS)
    public long mult()
    {
        return x * y;
    }

    @Benchmark
    @BenchmarkMode(Mode.AverageTime)
    @OutputTimeUnit(TimeUnit.NANOSECONDS)
    public long div()
    {
        return x / y;
    }

    @Benchmark
    @BenchmarkMode(Mode.AverageTime)
    @OutputTimeUnit(TimeUnit.MILLISECONDS)
    public double pow()
    {
        return Math.pow(x, y);
    }
}
```

Im Listing lernen wir einige neue Annotations kennen:

- `@State` – Man kann spezifizieren, wie der Zustand zwischen den Threads des Benchmarks geteilt werden soll. Im Thread-Scope erhält jeder Thread seinen unabhängigen Zustand. Hier nutzen wir den Zustand pro Benchmark: Damit ist dieser für alle Threads gleich.
- `@BenchmarkMode` – Oftmals ist zur Performance-Beurteilung die Anzahl an Operationen pro Zeiteinheit (ops/time = Default) oder die durchschnittliche Ausführungszeit (time/ops) von Interesse. Beides verwenden wir im diesem Beispiel.
- `@OutputTimeUnit` – Mit JMH werden die Laufzeiten normalerweise in Nanosekunden berechnet. Mitunter ist man aber an anderen Zeitskalen interessiert, wie oben an Millisekunden für die Potenzierung.

Nachdem wir beide Klassen kompiliert und paketiert haben, starten wir das Executable JAR. Das produziert Konsolenausgaben etwa wie folgt:

```
# Benchmark mode: Throughput, ops/time
# Benchmark: org.sample.SimpleMathBenchmark.mult

Result "org.sample.SimpleMathBenchmark.mult":
  0.401 ±(99.9%) 0.012 ops/ns [Average]
  (min, avg, max) = (0.348, 0.401, 0.414), stdev = 0.017
  CI (99.9%): [0.388, 0.413] (assumes normal distribution)

# Benchmark mode: Average time, time/op
# Benchmark: org.sample.SimpleMathBenchmark.mult

Result "org.sample.SimpleMathBenchmark.mult":
  2.568 ±(99.9%) 0.108 ns/op [Average]
  (min, avg, max) = (2.414, 2.568, 2.931), stdev = 0.144
  CI (99.9%): [2.460, 2.676] (assumes normal distribution)
```

Zunächst werden zwischen den einzelnen Iterationen die Ergebnisse präsentiert und zum Abschuss erfolgt die Ausgabe einer Zusammenfassung.

```
Benchmark                                    Mode  Cnt         Score
      Error   Units
FirstIntegerBenchmark.numberAsBinaryString  thrpt   12  36490099.791 ±
      7938126.104   ops/s
FirstIntegerBenchmark.numberAsHexString     thrpt   12  78930550.252 ±
      18442012.239  ops/s
SimpleMathBenchmark.mult                    thrpt   25         0.401 ±
      0.012  ops/ns
SimpleMathBenchmark.div                      avgt   25         4.445 ±
      0.115   ns/op
SimpleMathBenchmark.mult                     avgt   25         2.568 ±
      0.108   ns/op
SimpleMathBenchmark.pow                      avgt   25        ≈ 10^5
                   ms/op
```

Mit deren Hilfe kann man verschiedene Aussagen zur Performance ableiten. So sieht man beispielsweise, dass die Multiplikation rund 2,5 ns benötigt, eine Division jedoch rund 4,4 ns. Das Potenzieren hat eine Laufzeit von 0.00005 ms (50 ns) und ist damit um Größenordnungen langsamer als die einfachen Operationen * und /. Das war zu erwarten, lässt sich aber so auch gut nachvollziehen.

8.1.3 Fazit zu JMH

Mitunter ist es wichtig, Aussagen über die Laufzeit spezieller Programmteile zu erhalten, um Optimierungspotenziale zu ermitteln. Neben einem Profiler, der Messungen zur Laufzeit ermöglicht, ist es manchmal auch wünschenswert, die Performance spezieller Anweisungen oder Methoden feingranular zu messen. Implementiert man dies jedoch von Hand, so wird man mit einigen Schwierigkeiten und Tücken konfrontiert. Sinnvoller ist es, hierzu JMH zu nutzen, das das Erstellen korrekter Microbenchmarks deutlich erleichtert. In diesem Unterkapitel konnte ich Ihnen lediglich einen ersten Eindruck von den Möglichkeiten von JMH vermitteln, es gibt noch sehr viel mehr zu entdecken.